- Contents
- Quick start
- Concepts
  - Glossary
  - Analytical Features
    - Overview
    - Detailed
      - Data Ingestion
      - Data Storage
      - Query Execution
      - Federated queries
      - Data transformation and preparation (ETL/ELT)
      - BI analytics and data visualization
      - Machine Learning
    - Key Features
  - Stream processing
    - Streaming queries
    - Watermarks
  - Architecture
    - Architecture overview
    - Metadata distribution services
  - Connecting to a database
  - Schema objects
    - Cluster namespace
    - Directories
    - Tables
    - Views
    - Topics
    - Coordination nodes
    - Secrets
    - External tables
    - External data source
    - Service objects
      - Backup collections
  - Cluster topology
  - Bridge mode
  - Query execution
    - Overview
    - Query optimizer
    - Multi-Version Concurrency Control (MVCC)
    - Secondary indexes
    - Vector search
    - Fulltext search
    - Hybrid search
    - JSON Search
    - Local indexes
    - Federated query
      - Overview
      - Architecture
      - Working with ClickHouse databases
      - Working with Greenplum databases
      - Working with Microsoft SQL Server databases
      - Working with MySQL databases
      - Working with PostgreSQL databases
      - Working with YDB databases
      - Working with S3 buckets
        - Reading data through connections
        - Reading data through external tables
        - Writing data to S3 buckets
        - Data formats and compression algorithms
        - Mapping types for Parquet read and write
        - Data partitioning
        - Extended data partitioning
      - Data import and export
    - Spilling
    - Scan queries
    - YQL queries to topics
  - Transactions
  - Change Data Capture (CDC)
  - Time to live and eviction
  - Database limits

    - Backup and restore
    - Asynchronous replication
    - Bridge mode
    - Data transfer
- For Database Administrators
    - Concepts
        - System requirements
        - CPU settings for production
        - Storage capacity planning
        - Versioning
        - Maintenance
        - SelfHeal
            - Storage SelfHeal
            - Metadata Distribution SelfHeal
    - Cluster configuration management
        - Configuration V1
            - Configuration overview
            - Static configuration
            - Dynamic configuration
            - Volatile configuration
            - Cluster configuration DSL
            - Changing configurations via CMS
            - Changing actor system configuration
            - Cluster expansion
            - Configuring State Storage
            - State Storage move
            - Static group move
            - Replacing node FQDN
            - Database node authentication and authorization
        - Configuration V2
            - Configuration overview
            - Configuration update
            - Cluster configuration DSL
            - Configuration parameters
            - Cluster expansion
            - Configuring State Storage
            - State Storage move
            - Static group move
            - Replacing node FQDN
            - Database node authentication and authorization
            - Static group SelfHeal
        - Migration
            - Migration to configuration V2
            - Migration to configuration V1
        - Checking configuration version
        - Comparing configurations V1 and V2
    - Deployment options
        - Ansible
            - Initial deployment
                - Deployment preparation
                - Configuration V1
                - Configuration V2
            - Preparing VMs with Terraform
            - Restart
            - Update config
            - Update executable
            - Observability
                - Logging
        - Kubernetes
            - Initial deployment
        - Manual
            - Initial deployment
                - Deployment preparation
                - Deploying a cluster using configuration V1
                - Deploying a cluster using configuration V2
            - Updating configuration
            - Cluster maintenance
            - Updating executable
            - Database node authentication and authorization
            - Managing a cluster in bridge mode
            - Managing a cluster's disk subsystem
                - Overview
                - Adding storage groups

            - Safe restart and shutdown of nodes
            - Enabling/disabling Scrubbing
            - Data Center Decommissioning
            - Decommissioning groups using virtual groups
            - Moving VDisks
            - Staying within the failure model
            - Disk load balancing
            - Freeing up space on physical devices
            - Replacing a node's FQDN
            - BlobDepot
            - BlobDepot decommit
        - Federated queries
            - Connector deployment
    - Enterprise Manager
        - Initial deployment
        - Configuring S3 backups
    - Observability
        - Monitoring
        - Logging
        - Minidumps
        - System views
        - Tracing
        - Alerting
        - Diagnostics
    - Backup and Restore
- For Application Developers
    - Getting started
    - Streaming queries
        - Typical patterns
        - Writing to Tables
        - Data Enrichment
        - Data formats for reading/writing topics
        - Data Delivery Guarantees
        - Checkpoints
        - Watermarks
    - Example applications
        - C++
        - C# (.NET)
        - Go
        - Java
        - JavaScript
        - PHP
        - Python
        - Rust
    - Generate code from YQL
    - YQL tutorial
        - Creating a table
        - Adding data to a table
        - Selecting data from all columns
        - Selecting data from specific columns
        - Sorting and filtering
        - Data aggregation
        - Additional selection criteria
        - Joining tables with JOIN
        - Inserting and updating data with REPLACE
        - Inserting and updating data with UPSERT
        - Inserting data with INSERT
        - Updating data with UPDATE
        - Deleting data
        - Adding and deleting columns
        - Deleting a table
    - Primary keys
        - Row-oriented
        - Column-oriented
    - Secondary indexes
    - Vector indexes
    - Vector index type vector_kmeans_tree
    - Fulltext indexes
    - Hybrid search
    - JSON indexes
    - Local indexes
        - Bloom skip indexes
        - min-max index

    - Query execution optimization
        - Using query plans for query optimization
        - Optimizer hints
        - Parameterized queries and recompilation
    - Batch upload
    - Paging
    - Timeouts
    - System views
    - Change Data Capture
    - Terraform
    - Custom attributes
    - Resource consumption management
    - Structure and limitations of a shared topic reader
- For Analysts
    - Quick start
    - Dataset import
        - Overview
        - Chess Position Evaluations
        - Video Game Sales
        - E-Commerce Behavior Data
        - COVID-19 Open Research Dataset
        - Netflix Movies and TV Shows
        - Animal Crossing New Horizons Catalog
    - Practical guides
        - Data versioning (SCD)
            - Overview
            - SCD1 with TRANSFER
            - SCD2 with TRANSFER
            - SCD2 with merge of changes
    - Limitations
- For Security Engineers
    - Authentication
    - Authorization
    - Initial security configuration
    - Audit log
    - Encryption
        - Data at rest
        - Data in transit
    - Short access control notation
- For Contributors
    - Working on a change
    - Ya Make build system
    - Releases
    - Documentation
        - Review process
        - Style guide
        - Structure
        - Genres
    - Core
        - General schema
        - Distributed storage
        - Metadata Distribution Subsystems
        - LocalDB
            - Persistent uncommitted changes
        - Hive
            - Tablet Boot Process
        - DataShard
            - Locks and transaction change visibility
            - Distributed transactions
        - Configuration V2
        - Bridge mode
        - Spilling Service
        - Testing with load actors
            - KqpLoad
            - KeyValueLoad
            - StorageLoad
            - VDiskLoad
            - PDiskWriteLoad
            - PDiskReadLoad
            - PDiskLogLoad
            - MemoryLoad
            - Stop
    - Testing

        - Chaos Testing
- Reference
    - YQL
        - Overview
        - Data types
            - Overview
            - Simple
            - Serial
            - Optional
            - Containers
            - Special
            - Type casting
            - Text representation of data types
            - JSON
        - Syntax
            - Overview
            - Lexical structure
            - Expressions
            - Limitations
                - Correlated subqueries
                - Unsupported statements
            - ACTION
            - ALTER ASYNC REPLICATION
            - ALTER DATABASE
                - Overview
                - OWNER TO
                - SET
            - ALTER GROUP
            - ALTER TABLE
                - Overview
                - INDEX
                - COLUMN
                - FAMILY
                - NOT NULL
                - SET
                - CHANGEFEED
                - RENAME
                - COMPACT
            - ALTER TRANSFER
            - ALTER SECRET
            - ALTER VIEW
            - ALTER TOPIC
            - ALTER STREAMING QUERY
            - ALTER USER
            - ALTER RESOURCE POOL
            - ALTER RESOURCE POOL CLASSIFIER
            - ANALYZE
            - BATCH DELETE
            - BATCH UPDATE
            - BACKUP
            - CREATE ASYNC REPLICATION
            - CREATE BACKUP COLLECTION
            - CREATE GROUP
            - CREATE EXTERNAL DATA SOURCE
            - CREATE STREAMING QUERY
            - CREATE OBJECT TYPE SECRET
            - CREATE OBJECT TYPE SECRET_ACCESS
            - CREATE EXTERNAL TABLE
            - CREATE OBJECT TYPE SECRET
            - CREATE RESOURCE POOL
            - CREATE RESOURCE POOL CLASSIFIER
            - CREATE SECRET
            - CREATE TABLE
                - Overview
                - SECONDARY INDEX
                - VECTOR INDEX
                - FULLTEXT INDEX
                - JSON INDEX
                - BLOOM INDEX
                - min_max INDEX
                - FAMILY
                - WITH
                - AS SELECT

- CREATE TRANSFER
- CREATE VIEW
- CREATE RESOURCE POOL
- CREATE RESOURCE POOL CLASSIFIER
- CREATE TOPIC
- CREATE USER
- COMMIT
- DECLARE
- DELETE
- DISCARD
- DROP ASYNC REPLICATION
- DROP BACKUP COLLECTION
- DROP GROUP
- DROP EXTERNAL DATA SOURCE
- DROP EXTERNAL TABLE
- DROP STREAMING QUERY
- DROP OBJECT TYPE SECRET
- DROP OBJECT TYPE SECRET_ACCESS
- DROP TABLE
- DROP SECRET
- DROP TRANSFER
- DROP TOPIC
- DROP VIEW
- DROP RESOURCE POOL
- DROP RESOURCE POOL CLASSIFIER
- DROP USER
- GRANT
- INSERT
- INTO RESULT
- PRAGMA
- REPLACE
- REVOKE
- RESTORE
- SELECT
  - Overview
  - FROM
  - FROM AS_TABLE
  - FROM SELECT
  - FLATTEN
  - GROUP BY
  - JOIN
  - WINDOW
  - DISTINCT
  - UNIQUE DISTINCT
  - UNION
  - VIEW SECONDARY INDEX
  - VIEW VECTOR INDEX
  - VIEW FULLTEXT INDEX
  - VIEW JSON INDEX
  - ORDER BY HybridRank
  - WITH
  - WITHOUT
  - WHERE
  - ORDER BY
  - ASSUME ORDER BY
  - LIMIT OFFSET
  - SAMPLE
  - MATCH_RECOGNIZE
  - STREAMING
- SHOW CREATE
- TRUNCATE TABLE
- UPDATE
- UPSERT
- UPSERT OBJECT TYPE SECRET
- VALUES

- Built-in functions
  - Overview
  - Basic
  - Fulltext search
  - Aggregate
  - Window
  - For lists
  - For dictionaries

        - For structures
        - For types
        - For code generation
        - For JSON
        - C++ libraries
            - Overview
            - DateTime
            - Digest
            - Histogram
            - Hyperscan
            - Ip
            - Knn
            - Math
            - Pcre
            - Pire
            - Re2
            - Roaring
            - String
            - Unicode
            - Url
            - Yson
    - Query plans
- PostgreSQL compatibility
- YDB UI
    - Overview
    - YDB Monitoring
    - Hive web-viewer
    - Connections overview
    - Logs
    - Charts
- Integrations
    - Graphical user interfaces
        - DBeaver
        - DBeaver Plugin
        - DataGrip
        - Jupyter Notebook
    - Data visualization
        - Apache Superset
        - DataLens
        - Grafana data source
    - Orchestration
        - Apache Airflow™
    - Data ingestion
        - Log records collection using FluentBit
        - Logstash plugins
    - Data migrations
        - Importing from JDBC data sources
        - Import from MySQL
    - Schema migrations
        - dbt
        - Flyway
        - goose
        - Liquibase
    - Query engines
        - Apache Spark™
    - Object–relational mapping
        - Hibernate
        - Spring Data JDBC
        - JOOQ
        - Dapper
        - Entity Framework
        - Linq To Db
        - SQLAlchemy
        - Django
        - Kotlin Exposed
        - Drizzle ORM
    - Vector search
        - LangChain
        - Mem0
    - SQL Dialect Converter to YQL
- YDB CLI
    - Overview
    - Installation

- Connecting to and authenticating with a database
- All commands in alphabetical order
- Service commands
- Global options
- YDB CLI commands
  - table attribute add
  - table attribute drop
- Working with the DB schema
  - List of objects
  - Information about the object
  - Permissions
  - Directories
  - Secondary indexes
  - Copying tables
  - Renaming tables
  - Setting TTL parameters
  - Resetting TTL parameters
  - Deleting a table
  - Generating CREATE TABLE from a file
- Operations with data
  - Getting a query execution plan and AST
  - Streaming table reads
- Importing and exporting data
  - Overview
  - File structure of data export
  - Exporting data to the file system
  - Importing data from the file system
  - Connecting to and authenticating with S3
  - Exporting data to S3
  - Importing data from S3
  - Export to NFS
  - Import from NFS
  - Importing data from a file to an existing table
  - Backup collections
- Working with topics
  - Commands for topics
  - Creating a topic
  - Updating a topic
  - Deleting a topic
  - Adding a topic consumer
  - Deleting a topic consumer
  - Saving a consumer offset
  - Reading messages from a topic
  - Writing messages to a topic
  - Message pipeline processing
- SQL query execution
  - Overview
  - Query execution
  - Parameterized query execution
  - Interactive query execution mode
  - AI assistant mode
  - Scan queries
- Managing background operations
  - Getting a list of long-running operations
  - Obtaining the status of long-running operations
  - Canceling long-running operations
  - Deleting long-running operations from the list
- Managing profiles
  - Overview
  - Creating and updating profile
  - Using a profile in requests
  - Getting profile information
  - Deleting a profile
  - Activated profile
- Information services
  - List of endpoints
  - Authentication
  - Displaying connection parameters
  - Getting the YDB CLI version
  - Health check
- Load testing
  - Overview
  - Stock load

        - ClickBench load
        - Key-Value load
        - Topic load
        - Transfer load
        - TPC-C load
        - TPC-H load
        - TPC-DS load
        - Query load
        - Fulltext load
        - Vector load
    - Managing configuration
        - Overview
        - Cluster commands
            - Generate dynamic configuration
            - Fetch current dynamic configuration
            - Replace dynamic configuration
        - Node commands
            - Initialize config directory
    - Bridge cluster management
        - Overview
        - ydb admin cluster bridge list
        - ydb admin cluster bridge switchover
        - ydb admin cluster bridge failover
        - ydb admin cluster bridge takedown
        - ydb admin cluster bridge rejoin
- YDB Native SDK
    - Overview
    - Installation
    - Authentication
    - Parameterized queries
    - Data formats
        - Protobuf (Value)
        - Apache Arrow
    - Working with topics
    - Working with coordination nodes
    - Handling errors in the API
        - YDB server status codes
        - gRPC status codes
    - gRPC API
        - Overview
        - gRPC headers
        - Health Check API
    - Observability
        - Overview
        - Logging
            - Enabling logging
            - SDK logs in OpenTelemetry
        - Metrics
            - SDK Metrics in OpenTelemetry
            - SDK Metrics in Prometheus
        - Tracing
            - SDK tracing in OpenTelemetry
            - SDK tracing in Jaeger
    - Comparison of SDK features
- Languages and APIs
    - ADO.NET
        - Getting Started
        - Installation
        - Basic Usage
        - Connection Parameters
        - Type Mapping
        - Connect to Yandex Cloud
    - JDBC driver
        - Quick start
        - Using with Maven
        - Authentication modes
        - Properties
        - Building
    - Model Context Protocol
- AI tools
    - YDB AI Skills
- Kafka API
    - Overview

        - Authentication
        - Usage examples
        - Kafka Connect
            - Overview
            - Kafka Connect setup. A step-by-step guide
            - Examples of connector configuration
        - Constraints
    - Amazon SQS API
        - Overview
        - Authentication
        - Examples
    - Configuration
        - actor_system_config
        - auth_config
        - aws_client_config
        - blob_storage_config
        - bridge_config
        - client_certificate_authorization
        - cms_config
        - domains_config
        - feature_flags
        - query_service_config
        - healthcheck_config
        - host_configs
        - hosts
        - log_config
        - memory_controller_config
        - node_broker_config
        - resource_broker_config
        - security_config
        - self_management_config
        - system_tablet_backup_config
        - tls
        - table_service_config
        - tli_config
        - hive_config
        - kafka_proxy_config
    - Observability
        - Metrics
            - Metrics reference
            - Grafana dashboards
        - Tracing
            - Setup
            - External traces
    - YDB DStool
        - Overview
        - Installation
        - Global options
        - Device list
        - Populate a PDisk
    - ydbops
        - Overview
        - Installation
        - Configuration
        - Cluster restart scenario
        - Reference
    - Docker
        - Tag naming
        - Installation
        - Start
        - Configuration
        - Init scripts
        - Cleanup
- Recipes
    - Backup collections recipes
        - Getting started
        - Multi-environment setup
        - Microservices backup strategy
        - Exporting to external storage
        - Importing and restoring
        - Maintenance and cleanup
        - Validation and testing
    - System tablet backup recipes

        - Enable and configure
        - Recovery
    - YDB SDK
        - Overview
        - Initialize the driver
        - Authentication
            - Overview
            - Using a token
            - Anonymous
            - Service account file
            - Metadata service
            - Using environment variables
            - Username and password based
        - Balancing
            - Overview
            - Random choice
            - Prefer the nearest data center
            - Prefer the specific availability zone
            - Prefer a pile with a specific state
        - Running repeat queries
        - Setting the session pool size
        - Upserting data
        - Bulk-upserting data
        - Setting up the transaction execution mode
        - Configuring time to live (TTL)
        - Vector search
        - Coordination
            - Distributed lock
            - Leader election
            - Service discovery
            - Configuration publication
        - Troubleshooting
            - Overview
            - Enable logging
            - Enable metrics in Prometheus
            - Tracing with OpenTelemetry
            - Connect tracing in Jaeger
    - YDB CLI
        - Convert table type
        - Benchmarks
        - Time to live (TTL)
    - YQL
        - Overview
        - Accessing JSON
        - Modifying JSON
        - Configuring TTL
    - Vector search
        - Vector index - quick start
        - Vector index with a prepared external dataset
    - Fulltext search
        - Fulltext index - quick start
    - JSON search
        - JSON index — quick start
        - Directory with nested attributes
        - Parameterized queries and JsonPath variables
        - Checking field type and path existence
    - Local indexes
        - Bloom skip index quickstart
        - min_max index — quick start
    - Transfer
        - Transfer — quick start
        - Transfer — streaming NGINX access logs to a table
    - Backup to NFS
    - Streaming queries
        - Quickstart: read and write topics
        - Debug reads from a topic
- Troubleshooting
    - Performance issues
        - Infrastructure
            - Network issues
            - Data center outages
            - Data center maintenance and drills
            - Hardware issues

        - Insufficient resources
            - CPU
            - Memory
            - I/O bandwidth
            - Disk space
        - Operating system
            - System clock drift
        - YDB configuration
            - Rolling restart
            - Frequent tablet moves between nodes
        - Schema design
            - Overloaded shards
            - Excessive tablet splits and merges
        - Client application
            - Transaction lock invalidation
            - TLI logging
            - OVERLOADED errors
            - Retry cascade effect
    - Spilling issues
        - Permission denied
        - Spilling Service not started
        - Total size limit exceeded
    - Diagnostic examples
        - Overloaded shard
- Questions and answers
    - General questions
    - SDK
    - Errors
    - YQL
    - Analytics
    - Known issues
    - All questions on one page
- Public materials
    - Videos
        - 2025
        - 2024
        - 2023
        - 2022
    - Articles
        - 2024
        - 2023
- Downloads
    - YDB Open-Source Database
    - Yandex Enterprise Database
    - YDB CLI
    - YDB DSTool
    - YDB Ops
    - YDB Ansible
- Changelog
    - YDB Server
    - Yandex Enterprise Database
    - YDB CLI
    - Security changelog

YDB Quick Start

In this guide, you will install a single-node local YDB cluster and execute simple queries against your database.

Normally, YDB stores data on multiple SSD/NVMe or HDD raw disk devices without any filesystem. However, for simplicity, this guide emulates disks in RAM or using a file in a regular filesystem. Thus, this setup is unsuitable for any production usage or even benchmarks. See the documentation for DevOps Engineers to learn how to run YDB in a production environment.

Install and start YDB

Linux x86_64

> **Note**
>
> The recommended environment to run YDB is x86_64 Linux. If you don't have access to one, feel free to switch to the instructions on the "Docker" tab.

1. Create a directory for YDB and use it as the current working directory:

   ```
   mkdir ~/ydbd && cd ~/ydbd
   ```

2. Download and run the installation script:

   ```
   curl https://install.ydb.tech | bash
   ```

   This will download and unpack the archive containing the `ydbd` executable, libraries, configuration files, and scripts needed to start and stop the local cluster.

   The script is executed entirely with the current user privileges (notice the lack of `sudo`). Therefore, it can't do much on the system. You can check which exactly commands it runs by opening the same URL in your browser.

3. Start the cluster in one of the following storage modes:
   - In-memory data:

     ```
     ./start.sh ram
     ```

     In this case, all data is stored only in RAM, it will be lost when the cluster is stopped.
   - Data on disk:

     ```
     ./start.sh disk
     ```

     When you run this command an 80GB `ydb.data` file will be created in the working directory if it weren't there before. Make sure there's enough disk space available to create it. This file will be used to emulate a raw disk device, which would have been used in production environments.
   - Data on a real disk drive:

     ```
     ./start.sh drive "/dev/$DRIVE_NAME"
     ```

     Replace `/dev/$DRIVE_NAME` with an actual device name that is not used for anything else, for example `/dev/sdb`. The first time you run this command, the specified disk drive will be fully wiped and then used for YDB data storage. It is recommended to use a NVMe or SSD drive with at least 800Gb data volume. Such setup can be used for single-node performance testing or other environments that do not have any fault-tolerance requirements.

   Result:

   ```
   Starting storage process...
   Initializing storage ...
   Registering database ...
   Starting database process...

   Database started. Connection options for YDB CLI:

   -e grpc://localhost:2136 -d /Root/test
   ```

Docker x86_64

> **Note**
>
> If you are using a Mac with an Apple Silicon processor, emulate the x86_64 CPU instruction set with Rosetta:
>
> - colima with the `colima start --arch aarch64 --vm-type=vz --vz-rosetta` options.
> - Docker Desktop with installed and enabled Rosetta 2.
>
> If Rosetta 2 is not enabled, add the `-e YDB_USE_IN_MEMORY_PDISKS=true` parameter to the command for running the Docker container. For more information, see Configuring the YDB Docker container.

1. Create a directory for YDB and use it as the current working directory:

```
mkdir ~/ydbd && cd ~/ydbd
mkdir ydb_data
mkdir ydb_certs
```

2. Run the Docker container:

```
docker run -d --rm --name ydb-local -h localhost \
  --platform linux/amd64 \
  -p 2135:2135 -p 2136:2136 -p 8765:8765 -p 9092:9092 \
  -v $(pwd)/ydb_certs:/ydb_certs -v $(pwd)/ydb_data:/ydb_data \
  -e GRPC_TLS_PORT=2135 -e GRPC_PORT=2136 -e MON_PORT=8765 \
  -e YDB_KAFKA_PROXY_PORT=9092 \
  ydbplatform/local-ydb:latest
```

If the container starts successfully, you'll see the container ID. The container might take a few seconds to initialize. The database will not be available until container initialization is complete.

Minikube

1. Install the Kubernetes CLI kubectl and Helm 3 package manager.
2. Install and run Minikube.
3. Clone the repository with YDB Kubernetes Operator:

```
git clone https://github.com/ydb-platform/ydb-kubernetes-operator && cd ydb-kubernetes-operator
```

4. Install the YDB controller in the cluster:

```
helm upgrade --install ydb-operator deploy/ydb-operator --set metrics.enabled=false
kubectl rollout status deployment/ydb-operator --timeout=5m
```

5. Apply the manifest for creating a YDB cluster:

   First, check that the API server can call the controller's admission webhook and accept the manifest without creating the resource:

```
kubectl apply --dry-run=server -f samples/minikube/storage.yaml
```

   The webhook may still be unavailable after `rollout status` completes. If the check reports `connection refused` or `no endpoints available` when calling the webhook, wait a few seconds and retry the check. Resolve the cause of any other errors before proceeding. Apply the manifest only after the check succeeds:

```
kubectl apply -f samples/minikube/storage.yaml
```

6. Wait for `kubectl get storages.ydb.tech` to become `Ready`.
7. Apply the manifest for creating a database:

```
kubectl apply -f samples/minikube/database.yaml
```

8. Wait for `kubectl get databases.ydb.tech` to become `Ready`.
9. After processing the manifest, a StatefulSet object that describes a set of dynamic nodes is created. Use the following separate services to connect from the same Kubernetes namespace:
   - `database-minikube-sample-grpc:2135` (gRPC);
   - `database-minikube-sample-status:8765` (YDB UI).

   Database path: `/Root/database-minikube-sample`.
10. To continue, get access to port 8765 from outside Kubernetes using `kubectl port-forward database-minikube-sample-0 8765`.

Kind

1. Install the Kubernetes CLI kubectl and Helm 3 package manager.
2. Install Kind.
3. Clone the repository with YDB Kubernetes Operator:

```
git clone https://github.com/ydb-platform/ydb-kubernetes-operator && cd ydb-kubernetes-operator
```

4. Create a Kind cluster and wait until it is ready:

```
kind create cluster --config=samples/kind/kind-config.yaml  --wait 5m
```

5. Install the YDB controller in the cluster:

```
helm upgrade --install ydb-operator deploy/ydb-operator --set metrics.enabled=false
kubectl rollout status deployment/ydb-operator --timeout=5m
```

6. Apply the manifest for creating a storage:

   First, check that the API server can call the controller's admission webhook and accept the manifest without creating the resource:

```
kubectl apply --dry-run=server -f samples/kind/storage.yaml
```

The webhook may still be unavailable after `rollout status` completes. If the check reports `connection refused` or `no endpoints available` when calling the webhook, wait a few seconds and retry the check. Resolve the cause of any other errors before proceeding. Apply the manifest only after the check succeeds:

```
kubectl apply -f samples/kind/storage.yaml
```

7. Wait for `kubectl get storages.ydb.tech` to become `Ready`.
8. Apply the manifest for creating a database:

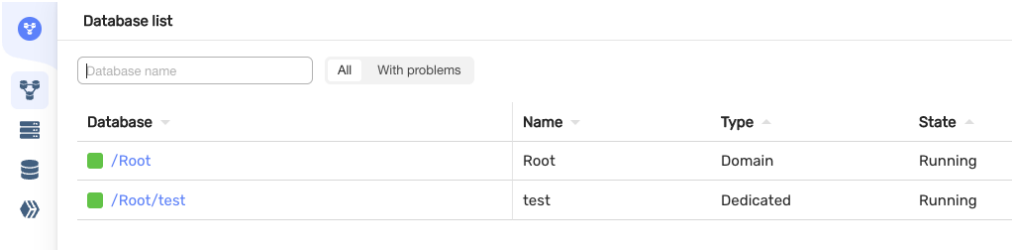
```
kubectl apply -f samples/kind/database.yaml
```

9. Wait for `kubectl get databases.ydb.tech` to become `Ready`.
10. After processing the manifest, a StatefulSet object that describes a set of dynamic nodes is created. Use the following separate services to connect from the same Kubernetes namespace:
    - `database-kind-sample-grpc:2135` (gRPC);
    - `database-kind-sample-status:8765` (YDB UI).

    Database path: `/Root/database-kind-sample`.
11. To continue, get access to port 8765 from outside Kubernetes using `kubectl port-forward database-kind-sample-0 8765`.

Run your first "Hello, world!" query

The simplest way to launch your first YDB query is via YDB UI. It is launched by default on port 8765 of the YDB server. If you have launched it locally, open localhost:8765 in your web browser. If not, replace `localhost` with your server's hostname in this URL or use `ssh -L 8765:localhost:8765 my-server-hostname-or-ip.example.com` to set up port forwarding and still open localhost:8765. You'll see a page like this:



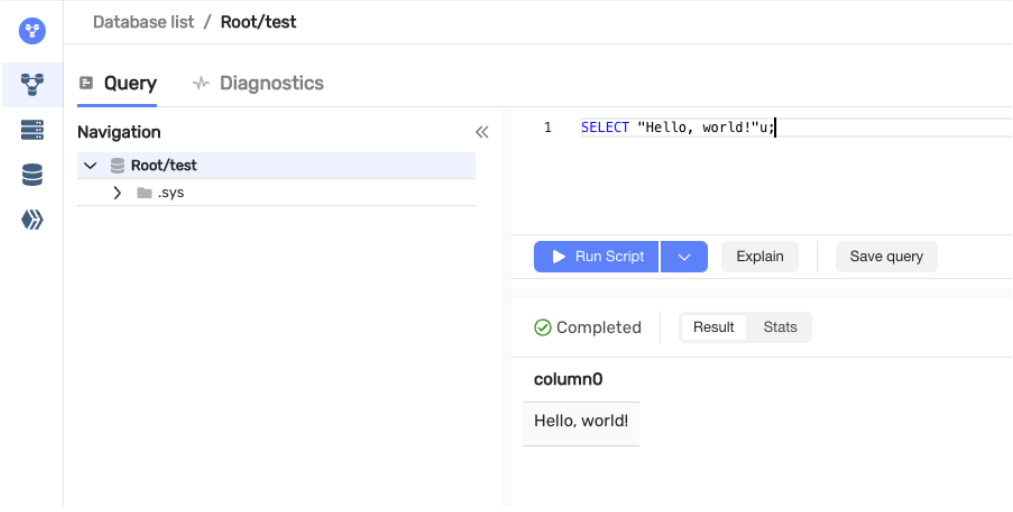
YDB is designed to be a multi-tenant system, with potentially thousands of users working with the same cluster simultaneously. Hence, most logical entities inside a YDB cluster reside in a flexible hierarchical structure more akin to Unix's virtual filesystem rather than a fixed-depth schema you might be familiar with from other database management systems. As you can see, the first level of hierarchy consists of databases running inside a single YDB process that might belong to different tenants. `/Root` is for system purposes. To run queries, select the database created in the previous step:

Installation method	Database
Linux x86_64	`/Root/test`
Docker x86_64	`/local`
Minikube	`/Root/database-minikube-sample`
Kind	`/Root/database-kind-sample`

Click on the corresponding database, enter your first query, and hit the "Run" button:

```
SELECT "Hello, world!"u;
```

The query returns the greeting, as it is supposed to:



Note

Did you notice the odd `u` suffix? YDB and its query language, YQL, are strongly typed. Regular strings in YDB can contain any binary data, while this suffix indicates that this string literal is of the `Utf8` data type, which can only contain valid UTF-8 sequences. Learn more about YDB's type system.

The second simplest way to run a SQL query with YDB is the command line interface (CLI), while most real-world applications will likely communicate with YDB via one of the available software development kits (SDK). Feel free to follow the rest of the guide using either the CLI or one of the SDKs instead of the web UI if you feel comfortable doing so.

Create your first table

The main purpose of database management systems is to store data for later retrieval. As an SQL-based system, YDB's primary abstraction for data storage is a table. To create our first one, run the following query:

```
CREATE TABLE example
(
    key UInt64,
    value String,
    PRIMARY KEY (key)
);
```

As you can see, it is a simple key-value table. Let's walk through the query step-by-step:

- Each SQL statement kind like `CREATE TABLE` has more detailed explanation in YQL reference.
- `example` is the table name identifier, while `key` and `value` are column name identifiers. It is recommended to use simple names for identifiers like these, but if you need one that contains non-trivial symbols, wrap the name in backticks.
- `UInt64` and `String` are data type names. `String` represents a binary string, and `UInt64` is a 64-bit unsigned integer. Thus, our example table stores string values identified by unsigned integer keys. More details about data types.
- `PRIMARY KEY` is one of the fundamental concepts of SQL that has a significant impact on both application logic and performance. Following the SQL standard, the primary key also implies an unique constraint, meaning the table cannot have multiple rows with equal primary keys. In this example table, it's quite straightforward which column should be chosen as the primary key, which we specify as `(key)` in round brackets after the respective keyword. In real-world scenarios, tables often have dozens of columns, and primary keys can be compound (consisting of multiple columns in a specified order), making choosing the right primary key more of an art. If you are interested in this topic, there's a guide on choosing the primary key for maximizing performance. YDB tables are required to have a primary key.

Add sample data

Now let's fill our table with some data. The simplest way is to just use literals:

```
INSERT INTO example (key, value)
VALUES (123, "hello"),
       (321, "world");
```

Step-by-step walkthrough:

- `INSERT INTO` is the classic SQL statement for adding new rows to a table. However, it is not the most performant, as according to the SQL standard, it has to check whether the table already has rows with the given primary key values, and raise an error if they exist. Thus, if you run this query multiple times, all attempts except the first will return an error. If your application logic doesn't require this behavior, it is better to use `UPSERT INTO` instead of `INSERT INTO`. Upsert (which stands for "update or insert") will blindly write the provided values, overwriting existing rows if there were any. The rest of the syntax will be the same.
- `(key, value)` specifies the names of the columns we're inserting and their order. The values provided next need to match this specification, both in the number of columns and their data types.
- After the `VALUES` keyword, there's a list of tuples, each representing a table row. In this example, we have two rows identified by 123 and 321 in the `key` column, and "hello" and "world" values in the `value` column, respectively.

To double-check that the rows were indeed added to the table, there's a common query that should return `2` in this case:

```
SELECT COUNT(*) FROM example;
```

A few notable details in this one:

- The `FROM` clause specifies a table to retrieve data from.
- `COUNT` is an aggregate function that counts the number of values. By default, when there are no other special clauses around, the presence of any aggregate function collapses the result to one row containing aggregates over the whole input data (the `example` table in this case).
- Asterisk `*` is a placeholder that normally means "all columns"; thus, `COUNT` will return the overall row count.

Another common way to fill a table with data is by combining `INSERT INTO` (or `UPSERT INTO`) and `SELECT`. In this case, values to be stored are calculated inside the database instead of being provided by the client as literals. We'll use a slightly more realistic query to demonstrate this:

```
$subquery = SELECT ListFromRange(1000, 10000) AS keys;

UPSERT INTO example
SELECT
    key,
    CAST(RandomUuid(key) AS String) AS value
FROM $subquery
FLATTEN LIST BY keys AS key
```

There's quite a lot going on in this query; let's dig into it:

- `$subquery` is a named expression. This syntax is YQL's extension to the SQL standard that allows making complex queries more readable. It behaves the same as if you wrote that first `SELECT` inline where `$subquery` is later used on the last row, but it allows comprehending what's going on piece by piece, like variables in regular programming languages.
- `ListFromRange` is a function that produces a list of consecutive integers, starting from the value provided in the first argument and ending with the value provided in the second argument. There's also a third optional argument that can allow skipping integers with a specified step, but we omit it in our example, which defaults to returning all integers in the given range. `List` is one of the most common container data types.
- `AS` is a keyword used to give a name to the value we're returning from `SELECT`; in this example, `keys`.
- `FROM ... FLATTEN LIST BY ... AS ...` has a few notable things happening:
  - Another `SELECT` used in the `FROM` clause is called a subquery. That's why we chose this name for our `$subquery` named expression, but we could have chosen something more meaningful to explain what it is. Subqueries normally aren't materialized; they just pass the output of one `SELECT` to the input of another on the fly. They can be used as a means to produce arbitrarily complex execution graphs, especially if used in conjunction with other YQL features.
  - `FLATTEN LIST BY` clause modifies input passed via `FROM` in the following way: for each row in the input data, it takes a column of list data type and produces multiple rows according to the number of elements in that list. Normally, that list column is replaced by the column with the current single element, but the `AS` keyword in this context allows access to both the whole list (under the original name) and the current element (under the name specified after `AS`), or just to make it more clear what is what, like in this example.
- `RandomUuid` is a function that returns a pseudorandom UUID version 4. Unlike most other functions, it doesn't actually use what is passed as an argument (the `key` column); instead, it indicates that we need to call the function on each row. See the reference for more examples of how this works.
- `CAST(... AS ...)` is a common function for converting values to a specified data type. In this context, the type specification is expected after `AS` (in this case, `String`), not an arbitrary name.
- `UPSERT INTO` will blindly write the values to the specified tables, as we discussed previously. Note that it didn't require `(key, value)` column names specification when used in conjunction with `SELECT`, as now columns can just be matched by names returned from `SELECT`.

Quick question!

What will the `SELECT COUNT(*) FROM example;` query return now?

Stop the cluster

Stop the local YDB cluster after you have finished experimenting:

Linux x86_64

To stop the local cluster, run the following command:

```
~/ydbd/stop.sh
```

Optionally, you can then clean up your filesystem by removing your working directory with the `rm -rf ~/ydbd` command. All data inside the local YDB cluster will be lost.

Docker

To stop the Docker container with the local cluster, run the following command:

```
docker kill ydb-local
```

Optionally, you can then clean up your filesystem by removing your working directory with the `rm -rf ~/ydbd` command. All data inside the local YDB cluster will be lost.

Minikube

To delete the YDB database, it is enough to delete the Database resource associated with it:

```
kubectl delete database.ydb.tech database-minikube-sample
```

To delete the YDB cluster, execute the following commands (all data will be lost):

```
kubectl delete storage.ydb.tech storage-minikube-sample
```

To remove the YDB controller from the Kubernetes cluster, delete the release created by Helm:

```
helm delete ydb-operator
```

Kind

To delete the YDB database, it is enough to delete the Database resource associated with it:

```
kubectl delete database.ydb.tech database-kind-sample
```

To delete the YDB cluster, execute the following commands (all data will be lost):

```
kubectl delete storage.ydb.tech storage-kind-sample
```

To remove the YDB controller from the Kubernetes cluster, delete the release created by Helm:

```
helm delete ydb-operator
```

To delete `kind` cluster, run the following command:

```
kind delete cluster
```

Done! What's next?

After getting a hold of some basics demonstrated in this guide, you should be ready to jump into more advanced topics. Choose what looks the most relevant depending on your use case and role:

- Walk through a more detailed YQL tutorial that focuses on writing queries.
- Try to build your first app storing data in YDB using one of the SDKs.
- Learn how to set up a production-ready deployment of YDB.
- Read about YDB concepts.

Concepts YDB

This documentation section covers the fundamental concepts and architectural principles of YDB. This information will help you better understand what happens when you study more practical content that can be found in other sections designed for specific roles.

The following topics provide comprehensive coverage of the core functionality of YDB, from high-level architecture and data model to advanced capabilities such as transactions, indexing, and federated queries.

- Glossary
- Analytical capabilities
- Stream processing
- Architecture
- Database connection
- Schema objects
- Cluster topology
- Query execution
- Transactions
- Change Data Capture (CDC)
- Time to Live (TTL) and data eviction to external storage
- Database limitations
- Asynchronous replication
- Bridge mode
- Data transfer

See also

- YDB for DevOps Engineers
- YDB for Application Developers / Software Engineers
- YDB for Security Engineers

YDB glossary

This article is an overview of terms and definitions used in YDB and its documentation. It starts with key terms that will be useful to get acquainted with early when you start working with YDB, while the rest of it is more advanced and might be helpful later on.

Key terminology

This section describes terms that are useful to anyone working with YDB, regardless of their role or use case.

Cluster

A **cluster** YDB is a set of interconnected nodes YDB that exchange data to execute user queries and ensure reliable data storage. These nodes form one of the supported cluster topologies, which directly affects its reliability and performance characteristics.

Clusters YDB are multi-tenant and can contain several isolated databases.

Database

Like in most database management systems, a **database** in YDB is a logical container for other entities like tables. However, in YDB, the namespace inside databases is hierarchical like in virtual file systems, and thus folders allow for further organization of entities.

Another essential characteristic of YDB databases is that they typically have dedicated compute resources allocated to them. Hence, creating a database requires additional operations from DevOps engineers.

Node

YDB A **node** is a server process that runs an executable called `ydbd`. Multiple nodes YDB can run on a single physical server or virtual machine, which is common practice. Thus, in the context of YDB, nodes are **not** synonymous with hosts.

Since YDB uses an approach with separate storage and compute layers (storage and compute separation), `ydbd` has several operation modes that define the node type. The available node types are described below.

Database node

Database nodes (also known as **tenant nodes** or **compute nodes**) process user queries addressed to a specific logical database. Their state is stored only in RAM and can be restored from distributed storage. The set of database nodes of a given cluster YDB can be considered the compute layer of that cluster. Thus, adding database nodes and allocating additional resources (CPU and RAM) to them are the main ways to increase the compute resources of a database.

The main role of database nodes is to run various tablets and actors, as well as accept incoming requests via various endpoints.

Storage node

Storage nodes are stateful nodes responsible for long-term storage of data fragments. The set of storage nodes in a given cluster YDB is called distributed storage and can be considered as the storage layer of that cluster. Thus, adding additional storage nodes and their disks is the primary way to increase the storage capacity and I/O throughput of the cluster.

Hybrid node

A **Hybrid node** is a process that simultaneously performs both the roles of a database node and a storage node. Hybrid nodes are often used for development purposes. For example, you can run a container with a full-featured YDB containing only one `ydbd` process in hybrid mode. They are rarely used in production environments.

Static node

Static nodes are configured manually during initial cluster initialization or reconfiguration. Typically, they serve as storage nodes, but they can technically be configured as database nodes.

Dynamic node

Dynamic nodes are added and removed from the cluster on the fly. They can only act as database nodes.

Distributed storage

Distributed storage, **Blob storage**, or **BlobStorage** is a distributed fault-tolerant data persistence layer of YDB. It has a specialized API designed for storing immutable pieces of tablet's data.

Multiple terms related to the distributed storage implementation are covered below.

Storage group

Storage group, **distributed storage group**, or **Blob storage group** is a place for reliable data storage, similar to RAID, but using disks from multiple servers. Depending on the selected cluster topology, storage groups use different algorithms to ensure high availability, similar to standard RAID levels.

Distributed storage typically manages a large number of relatively small storage groups. Each group can be assigned to a specific database to increase disk capacity and input/output throughput available to this database.

Static and dynamic storage groups are physical, meaning their data is placed directly on VDisks.

Static group

A **static group** is a special storage group created during the initial deployment of the cluster. Its main role is to store data of system tablets, which can be considered as cluster-level metadata.

A static group might require special attention during major maintenance, such as decommissioning an availability zone.

Dynamic group

Ordinary storage groups that are not static are called **dynamic groups**. They are called dynamic because they can be created and removed on the fly while the cluster is running.

Virtual storage group

A **virtual storage group** is an entity that is not actually a storage group but appears as one from the outside (provides a similar external interface). It can store its data in other storage groups or in S3.

Storage pool

A **storage pool** is a set of data storage devices with similar characteristics. Each storage pool is assigned a unique name within the YDB cluster. Technically, each storage pool consists of multiple physical disks (PDisk). Each storage group is created in a specific storage pool, which determines the performance characteristics of the storage group through the selection of appropriate storage devices. Typically, separate storage pools are created for devices of different types (e.g., NVMe, SSD, and HDD) or for specific models of these devices with different capacity and access speed.

Actor

The Actor model is one of the main approaches to execution parallelism used in YDB. In this model, **actors** are lightweight user-space processes that can have and modify their private state, but can only influence each other indirectly through message passing. YDB has its own implementation of this model, which is described below.

In YDB, actors with the reliably persisted state are called tablets.

Tablet

A **tablet** is one of the main building blocks and abstractions of YDB. It represents an entity responsible for a relatively small segment of user or system data. Typically, a tablet manages up to several gigabytes of data, although some types of tablets can handle larger volumes.

For example, a row-oriented user table is managed by one or more DataShard tablets, with each tablet responsible for a continuous range of primary keys and the corresponding data.

End users sending queries to a YDB cluster aren't expected to know much about tablets, their kinds, or how they work, but it might still be helpful, for example, for performance optimizations.

Technically, tablets are actors with state reliably stored in distributed storage. This state allows a tablet to continue operating on a different database node if the previous one fails or becomes overloaded.

Tablet implementation details and related terms, as well as main tablet types, are covered below in the advanced section.

Transactions

YDB implements **transactions** on two main levels:

- Local database and the rest of the tablet infrastructure allow tablets to manipulate their state using **local transactions** with serializable isolation level. Technically, they are not local to a single node, since this state is stored remotely in distributed storage.
- In the context of YDB, the term **distributed transactions** usually refers to transactions that span multiple tablets. For example, transactions between tables or even rows of the same table are often distributed.
- **Single-shard** transactions cover one tablet and execute faster. For example, transactions between rows of the same table partition are often single-shard.

Together, these mechanisms allow YDB to provide strict consistency.

The implementation of distributed transactions is covered in a separate article DataShard: distributed transactions, while below there's a list of several related terms.

Sessions

Logical connections to the database that store the context needed for executing queries and managing transactions. Sessions are described in more detail in the section Sessions.

Client-side timeout

Client-side timeout — a time limit that an application or YDB SDK waits for a database operation to complete (for example, executing a query or receiving a response via a gRPC call). After this time expires, the client usually aborts the wait: closes the connection or data stream, receives a transport or SDK error — even before the server has returned an explicit response (see YDB server response codes).

If the client-side timeout is shorter than the query execution time on the YDB side, then due to the specifics of query processing in the cluster, a query interrupted on the client side may continue to execute on the server for some time. If this situation occurs on a large scale, the server becomes overloaded with queries for which the client is not waiting for a response. Therefore, frequent retries of the same query immediately after a timeout can exacerbate the overload. For more details, see the articles Retry cascade effect and Overloaded errors; retry policies in the SDK are described in the section Handling errors.

Implicit transactions

Implicit transaction is a query execution mode in which the transaction mode is not specified. In this case, YDB independently determines whether to wrap them in a transaction. This mode is described in more detail in Implicit Transactions.

Interactive transactions

An **interactive transaction** is a transaction in which multiple queries are executed, with application logic potentially running between them. It allows the program to send multiple queries and make decisions about the further course of the transaction based on the data received. Such a transaction can explicitly begin with a separate step (or open together with the first query) and complete separately. Unlike a single query that is executed and committed atomically, an interactive transaction enables building complex workflows in which multiple related operations are treated as a single atomic unit. At the same time, it incurs overhead on the system, as it requires holding optimistic locks and uncommitted changes for longer, and also increases the complexity of the application-side code.

Multi-version concurrency control

Multi-version concurrency control, **multi-version concurrency control** or **MVCC** is a method used by YDB for concurrent access of multiple parallel transactions to the database without interfering with each other. It is described in more detail in a separate article Multi-Version Concurrency Control (MVCC).

Streaming queries

A type of query designed for stream processing of an unbounded data stream. Unlike regular queries, streaming queries have no restrictions on execution duration, automatically restart on errors, and periodically save their state as checkpoints to ensure fault tolerance.

Streaming queries are described in more detail in Streaming queries.

Streaming query checkpoints

A periodically saved state of a streaming query, necessary for automatically restoring its operation after failures in a distributed system. For more details about checkpoints, see the article Checkpoints.

Topology

YDB supports several **topologies** of a cluster (or **topology**), described in more detail in a separate article YDB Cluster Topology. Below, several related terms are explained.

Availability zones and regions

An **availability zone** is a data center or an isolated segment thereof with minimal physical distance between nodes and minimal risk of failure at the same time as other availability zones. Thus, availability zones are expected not to share any infrastructure like power, cooling, or external network connections.

A **region** is a large geographic area containing several availability zones. The distance between availability zones in one region should be about 500 km or less. YDB performs data writes to each availability zone in the region synchronously, ensuring reasonable latency and uninterrupted operation in case of failure of one of the availability zones.

Rack

A **rack** or **server rack** is equipment used to organize the placement of multiple servers. Servers in the same rack are more likely to become unavailable simultaneously due to rack-level issues related to power supply, cooling, etc. YDB can take into account information about which server is in which rack when placing each data fragment in environments based on physical servers.

Pile

A **Pile** is a set of nodes that can fail or be shut down simultaneously while maintaining the operability of other parts of the cluster (pile). A Pile can remain operational when other cluster nodes are shut down. Piles are used in bridge mode to split the cluster into several parts between which synchronous replication is performed. A Pile can consist of nodes from one or more regions.

Bridge mode

Bridge mode is a special cluster topology in which data is stored with synchronous replication between several piles. The features of this mode are described in Bridge mode and also in Bridge cluster operation mode.

Table

A **table** is a structured piece of information arranged in rows and columns. Each row represents a single record or entry, while each column represents a specific attribute or field with a particular data type.

There are two main approaches to representing tabular data in memory or on disks: row-based (row by row) and column-based (column by column). The chosen approach greatly affects the performance characteristics of operations on this data: the former is more suitable for transactional workloads (OLTP), and the latter for analytical workloads (OLAP). YDB supports both approaches.

Row-oriented table

Row-oriented tables store data for all or most columns of a given row physically close to each other. They are explained in more detail in Row-Oriented Tables.

Column-oriented table

Column-oriented tables or **columnar tables** store data for each column separately. They are optimized for building aggregates over a small number of columns, but are less suitable for accessing specific rows, as rows need to be reconstructed from their cells on the fly. They are described in more detail in Column-Oriented Tables.

Primary key

A **primary key** is an ordered list of columns, the values of which uniquely identify rows. It is used to build the table's primary index. It is provided by the YDB user during table creation and dramatically impacts the performance of workloads interacting with that table.

Guidance on choosing primary keys is provided in Choosing a primary key.

Primary index

Primary index or **primary key index** is the main data structure used to find rows in a table. It is created based on the selected primary key and determines the physical order of rows in the table; thus, each table can have only one primary index. The primary index is unique.

Secondary index

A **secondary index** is an additional data structure used to find rows in a table, typically when this cannot be done efficiently using the primary index. Unlike a primary index, secondary indexes are managed independently of the table's main data. Thus, a table can have multiple secondary indexes for different scenarios. The capabilities of YDB regarding secondary indexes are described in a separate article Secondary indexes. A secondary index can be either unique or non-unique.

Special types of secondary indexes include vector index, full-text index, and JSON index.

Vector Index

Vector index is an additional data structure used to speed up the solution of the vector search problem when there is a lot of data and exact vector search without an index does not work satisfactorily.
The capabilities of YDB for approximate nearest neighbor search (ANN search) using vector indexes are described in a separate article Vector Indexes.

A **vector index** is a specialized type of secondary index designed for similarity search, as opposed to traditional secondary indexes optimized for equality or range search.

Fulltext index

Full-text index is an additional data structure used to speed up text search on a table column (by words and phrases, and when using N-grams, also by substrings).

The features of full-text search and index parameters are described in the articles Fulltext Indexes and Fulltext search.

JSON-index

JSON index is an additional data structure used to speed up predicates with the JSON_EXISTS and JSON_VALUE functions on a column of type `Json` or `JsonDocument`. Unlike traditional secondary indexes, which are optimized for equality or range searches on individual table columns, the JSON index works with arbitrary JsonPath paths within a JSON document.

A JSON index, like a full-text index, is built on top of an inverted index, but uses its own JSON document tokenizer. JSON search capabilities are described in the articles JSON indexes and Searching JSON document contents.

Local index

A local index is an auxiliary structure stored together with table data (unlike a global secondary index, which materializes a separate index table). It is applied while reading the main table in storage. See local indexes.

Bloom filter

A Bloom filter is a probabilistic data structure that allows you to quickly check whether an element belongs to a set. False positives are possible, but false negatives are not.

Local bloom index

A local Bloom index is a special case of a local index: a probabilistic filter on column values based on a Bloom filter that speeds up selective queries by skipping data fragments where the searched value is guaranteed to be absent. For more information, see Bloom indexes and local indexes.

Column family

Column family or **column group** is a feature that allows storing subsets of columns of a row table separately in a family or group. The main use case is storing some columns on other disk types (moving less important columns to HDD) or with different compression settings. If the workload requires many column families, consider using column tables.

Column encoding

Column encoding is a data storage optimization mechanism for table columns that reduces disk space usage and speeds up some operations.

Time to live

Time to live or **TTL** is a mechanism for automatically removing old rows from a table asynchronously in the background. It is explained in a separate article Time to Live (TTL) and Eviction to External Storage.

View

View is a way to save a query and access its results as if they were a real table. The view itself does not store any data except the query text. The query stored in the view is executed each time a SELECT is performed on it, generating the returned result. Any changes to the tables referenced by the view are immediately reflected in the read results.

Views can be user-defined or system.

User-defined views

User-defined views are created by the user using the CREATE VIEW command. They are described in more detail in View.

System view

System views are special views automatically created by the system for monitoring the state of the database and cluster. They are located in a special directory `.sys` in the root folder of each database. System views for databases are described in System database views; system views for the cluster and access management issues are described in Cluster System Views.

Topic

Message queue is used for reliable asynchronous communication between different systems by passing messages. YDB provides infrastructure that ensures exactly-once semantics in such communications. Using it, you can guarantee no lost messages or accidental duplicates.

Topic is a named entity in a message queue for interaction between writers and readers.

Several terms related to topics are listed below. How topics work in YDB is explained in more detail in a separate article Topic.

Partition

For horizontal scaling, topics are divided into separate elements called **partitions**. Thus, partitions are the unit of parallelism within a topic. Messages within each partition are ordered.

However, subsets of data managed by a single data shard or column shards can also be called partitions.

Offset

Offset is a sequence number that identifies a message within a partition.

Producer

Writer or **producer** is an entity that writes new messages to a topic.

Consumer

Reader or **consumer** is an entity that reads messages from a topic.

Change data capture

Change data capture or **CDC** is a mechanism that allows subscribing to a **change feed** for a specific table. Technically, it is implemented on top of topics. It is described in more detail in a separate article Change Data Capture (CDC).

Change feed

Change feed is an ordered list of changes to a table placed in a topic.

Backup collection

Backup collection is a schema object that organizes full and incremental backups for selected row tables. Collections provide point-in-time recovery by maintaining backup chains and ensuring consistent recovery of multiple tables. A table can belong to only one backup collection at a time.

For more information, see Backup collection.

Backup

Backup is a copy of data at a specific point in time that can be used for data recovery. In the context of backup collections, there are two types:

- **Full backup**: A complete snapshot of all data in the collection. It serves as the basis for backup chains and can be restored independently.
- **Incremental backup**: Captures only changes (inserts, updates, deletes) since the previous backup. Requires the entire backup chain for restoration.

Backup chain

Backup chain is an ordered sequence of backups starting with a full backup followed by zero or more incremental backups. Each incremental backup depends on all previous backups in the chain. Deleting any backup in the chain makes subsequent incremental backups unrecoverable.

Asynchronous replication instance

Async replication instance is a named entity that stores async replication settings (connection settings, list of replicated objects, etc.). It can also be used to obtain information about the async replication status: initial scan progress, lag, errors, etc.

Replicated object

Replicated object is an object (for example, a table) for which async replication is configured.

Replica object

Replica object is a mirror copy of the replicated object, automatically created by the async replication instance. Typically, it is read-only.

Transfer instance

Transfer instance is a named entity that stores transfer settings, including connection settings and data transformation rules. It can also be used to obtain information about the transfer status, for example errors.

Coordination node

Coordination node is a schema object that allows client applications to create semaphores for coordinating their actions. Coordination nodes are used to implement distributed locks, service discovery, leader election, and other scenarios. For more information about coordination nodes.

Semaphore

Semaphore is an object inside a coordination node that provides a synchronization mechanism for distributed applications. Semaphores can be persistent or temporary and support create, acquire, release, and monitor operations. For more information about semaphores in YDB.

Resource pool

Resource pool is a schema object that describes the limits imposed on resources (CPU, RAM, etc.) available for executing queries in this resource pool. A query is always executed in some resource pool. By default, all queries are executed in a resource pool named `default`, which does not impose any limits. For more information about using resource pools, see Workload Manager — resource consumption management.

Resource pool classifier

Resource pool classifier is an object designed to manage the distribution of queries among resource pools. It describes the rules by which a resource pool is selected for each query. These classifiers are global for the entire database and apply to all queries that come into it. For more information about their usage, see Workload Manager — resource consumption management.

YQL

YQL (YDB Query Language) is a high-level language for working with the system. It is a dialect of ANSI SQL. There's a lot of content covering YQL, including a tutorial, reference, and recipes.

Federated queries

Federated queries is a feature that allows executing queries to data stored in systems external to the YDB cluster.

A few terms related to federated queries are listed below. How YDB federated queries work is explained in more detail in a separate article Federated query.

External data source

External data source or **external connection** is metadata describing how to connect to a supported external system to execute federated queries.

External table

External table is metadata describing a specific dataset that can be retrieved from an external data source.

Secret

Secret is confidential metadata that requires special handling. For example, secrets can be used in definitions of external data sources and represent entities such as passwords and tokens.

Auth token

Auth token is a token used for authentication in YDB.

YDB supports various authentication modes and token types.

Cluster scheme

A **YDB cluster scheme** is a hierarchical namespace of a YDB cluster. The top-level element of the namespace is the cluster scheme root that contains databases as its children. Scheme objects inside databases can use nested directories to form a hierarchy.

Database scheme

A **database scheme** is a subset of the hierarchical namespace of a YDB cluster that belongs to a database.

Database root

A **database root** is a path to a database in a YDB cluster scheme.

Scheme root

Cluster schema root is the root element of the YDB namespace, whose child elements are databases.

Scheme object

A database schema consists of **scheme objects**, which can be databases, tables (including external tables), topics, folders, and so on.

For organizational convenience, scheme objects form a hierarchy using folders.

Folder

As in file systems, a **folder** or **directory** is a container for scheme objects.

Folders can contain subfolders, and this nesting can have arbitrary depth.

Access object

An **access object** during authorization is an entity for which access rights and restrictions are configured. In YDB, access objects are schema objects.

Each schema object has an owner and an access control list on that object, granted to users and groups (access subjects).

Access subject

An **access subject** is an entity that can access access objects and perform certain actions in the system.

Gaining access during these requests and actions depends on the configured access control lists and the access level of the subject.

An access subject can be a user or a group.

Access right

An **access right** is an entity that represents permission for an access subject to perform a specific set of operations in a cluster or database on a specific access object.

Access right inheritance

Access rights inheritance is a mechanism where access rights granted on parent access objects are inherited by child objects in the hierarchical database structure. This ensures that permissions granted at a higher level of the hierarchy apply to all lower levels unless they are explicitly overridden.

Permissions list

access control list or **ACL** — a list of all rights granted to access subjects (users and groups) on a specific access object.

Access level

An **access level** determines additional privileges of an access subject for scheme objects as well as privileges that are not related to scheme objects.

- Database
- Viewer
- Monitoring
- Administration.

The access level for a subject is configured using access control lists.

Access level list

Access Control List or **Permission List** — a list of SIDs of access subjects that are allowed a certain access level.

YDB provides several access level lists that collectively determine access levels in the system.

Detailed information about access control lists, their hierarchy, and operating principles is provided in the Access control lists section of the authorization documentation.

Owner

Owner — an access subject (user or group) that has full rights to a specific access object.

User

User — a person who uses YDB to perform a specific function.

YDB has the following types of users depending on their source:

- Local users in databases YDB.
- external users from third-party directories.

A user is identified by SID.

Local user

A **local user** is an individual whose YDB account is created directly in YDB using the `CREATE USER` command or during the initial security configuration.

External user

A user YDB whose account is created in an external directory, for example, an LDAP directory or IAM system.

Group

Group or **access group** - a named set of users and other groups with equal permissions for their members.

A group is identified by SID.

Role

A **role** is a named collection of access rights that can be granted to users or groups.

Roles in YDB are implemented using groups, which are created during the initial deployment of the cluster and are assigned a specific access rights list at the cluster schema root. For more information about roles, see the Initial cluster security configuration article.

SID

SID or **security identifier** — a string of the form `<name>` or `<name>@<auth-domain>` that identifies an access subject. It is used in authentication, authorization, access control lists, and access level lists.

SID identifies an individual user or group of users.

The optional suffix `@<auth-domain>` identifies the source of the access subject, i.e., the external directory or system from which it was obtained. For example, users or groups from an LDAP directory may have the suffix `@ldap`. The absence of a suffix means that the user or group was created and exists directly in YDB.

Query optimizer

Query optimizer is a set of YDB components responsible for converting the logical representation of a query into a specific physically executable plan for obtaining the requested result. The main goal of the optimizer is to select, among all possible query execution plans, one that is sufficiently efficient in terms of predicted execution time and cluster resource consumption. It is described in more detail in a separate article Query Optimization in YDB.

Compilation cache

compile cache is a cache of compiled queries on each node of the cluster. It is used to avoid recompilation: if the query text is already in the node's cache, no additional compilation is performed. For more details, see the Query Compilation Cache section.

Advanced Terminology

This section explains terms that are useful for YDB contributors and users who want to understand what happens inside the system more deeply.

Actor Implementation

Actor System

actor system is a C++ library with an implementation of the actor model for the needs of YDB.

Actor Service

actor service is an actor that has a well-known name and usually runs as a single instance on a node.

ActorId

ActorId is a unique identifier of an actor or a tablet in a cluster.

Actor System Interconnect

actor system interconnect or **interconnect** is the internal network layer of the cluster. All actors communicate with each other in the system via the interconnect.

Local

local is an actor service running on each node. It directly manages tablets on its node and interacts with Hive. It registers with Hive and receives commands to start tablets.

Tablet Implementation

tablet is an actor with persistent state. It includes a set of data that this tablet is responsible for and a state machine through which the tablet's data (or state) is modified. The tablet is a fault-tolerant entity because the tablet's data is stored in distributed storage, which survives disk and node failures. The tablet is automatically restarted on another node in case of failure or overload of the previous one. Data in the tablet is modified sequentially, as the system infrastructure guarantees that there is no more than one tablet leader through which tablet data changes are performed.

A tablet solves the same problem as the Paxos and Raft algorithms in other systems, namely the problem of distributed consensus. From a technical point of view, the tablet implementation can be described as a replicated state machine (RSM) on top of a shared log, since the tablet state is fully described by an ordered log of commands stored in a distributed and fault-tolerant storage.

At runtime, the tablet state machine is managed by three components:

1. The common tablet part ensures log consistency and recovery in case of failures.
2. **executor** is an abstraction of a local database, namely data structures and code that organize work with data stored by the tablet.
3. An actor with user code that implements the specific logic of a particular tablet type.

In YDB, there are multiple kinds of specialized tablets storing all kinds of data for all sorts of tasks. Many YDB features like tables and topics are implemented as specific tablets. Thus, reusing tablet infrastructure is one of the key means of YDB extensibility as a platform.

Typically, a YDB cluster runs orders of magnitude more tablets compared to the processes or threads that other systems would use for a cluster of similar size. In a YDB cluster, hundreds of thousands or millions of tablets can easily run simultaneously.

Since a tablet stores its state in distributed storage, it can be (re)started on any node of the cluster. Tablets are identified by TabletID, a 64-bit number assigned when the tablet is created.

Tablet Leader

Tablet leader is the current active leader of a given tablet. The tablet leader accepts commands, assigns them an order, and confirms them to the outside world. It is guaranteed that at any given time there is at most one leader for each tablet.

Tablet candidate

Tablet candidate is one of the election participants that wants to become the leader of a given tablet. If the candidate wins the election, it becomes the tablet leader.

Tablet follower

Tablet follower or **hot standby** is a copy of the tablet leader that applies the command log accepted by the leader (with some delay). A tablet can have zero or more replicas. Replicas perform two main functions:

- In case of termination or failure of the leader, replicas are preferred candidates for the new leader role, as they can become the leader much faster than other candidates because they have applied most of the log.
- Replicas can answer read-only requests if the client explicitly chooses an optional relaxed transaction mode that allows stale reads.

Tablet generation

Tablet generation is a number that identifies the reincarnation of the tablet leader. It changes only when a new leader is elected and always increases.

Tablet local database

Tablet local database or **local database** is a set of data structures and associated code that manage the state of the tablet and the data it stores. Logically, the state of the local database is represented by a set of tables, very similar to relational tables. Modification of the local database state is performed by local tablet transactions created by the user tablet actor.

Each local database table is stored using the LSM tree data structure.

Log-structured merge-tree

Log-structured merge-tree is a data structure designed to optimize write and read performance in storage systems. It is used in YDB to store tables of the local database and data of VDisks.

MemTable

All data written to the tables of the local database is initially stored in an in-memory data structure called **MemTable**. When the MemTable reaches a specified size, it is flushed to disk as an immutable data structure SST.

Sorted string table

Sorted string table is an immutable data structure that stores table rows sorted by key, facilitating efficient key lookup and range scans. Each SST consists of a continuous series of small data pages, typically about 7 KiB each, which further optimizes reading data from disk. An SST is usually part of an LSM tree.

Tablet pipe

Tablet pipe or **TabletPipe** is a virtual connection that can be established with a tablet. It involves finding the tablet leader by TabletID. This is the recommended way to work with a tablet. The term **open a pipe to a tablet** describes the process of resolving (finding) a tablet in the cluster and establishing a virtual communication channel with it.

TabletID

A **TabletID** is a cluster-wide unique tablet identifier.

Bootstrapper

Bootstrapper is the main mechanism for starting tablets, used for system tablets (e.g., Hive, DS controller, root SchemeShard). Hive initializes the other tablets.

Shared cache

Shared cache is an actor that stores data pages recently read from distributed storage. Caching these pages reduces the number of disk I/O operations and speeds up data retrieval, improving overall system performance.

Memory controller

A **memory controller** is an actor that manages YDB memory limits.

Spilling

Spilling is a memory management mechanism in YDB that temporarily offloads intermediate query data to external storage when such data exceeds the available RAM of the node. In YDB, disk is currently used for spilling.

For more details on spilling, see Spilling.

Tablet types

Tablets can be considered as a framework for building reliable components operating in a distributed system. Many components of YDB — both system and those working with user data — are implemented using this framework; the main ones are listed below.

Scheme shard

A **Scheme shard** or **SchemeShard** is a tablet that stores a database schema, including metadata of user tables, topics, etc.

In addition, there is a **root SchemeShard** that stores information about databases created in the cluster.

Data shard

DataShard or **Data shard** is a tablet that manages a segment of a row-based user table. A logical user table is divided into segments by continuous ranges of the table's primary key. Each such range is managed by a separate DataShard tablet. The range itself is also called a partition. The DataShard tablet stores data row by row, which is efficient for OLTP workloads.

Column shard

A **column shard** or **ColumnShard** is a tablet that stores a data segment of a column-oriented user table.

KV Tablet

KeyValue, **KV Tablet** is a tablet that implements a simple key → value mapping, where keys and values are strings. It also has several specific features, such as locks.

PQ Tablet

A **PQ Tablet** or **persistent queue tablet** is a tablet that implements the concept of a topic. Each topic consists of one or more partitions, and each partition is managed by a separate PQ tablet instance.

TxAllocator

A **TxAllocator** or **transaction allocator** is a system tablet that allocates unique transaction identifiers (TxID) within the cluster. Typically, a cluster has several such tablets, from which transaction proxy pre-allocates and caches ranges for local issuance within a single process.

Coordinator

Coordinator is a system tablet that ensures global ordering of transactions. The coordinator's task is to assign a logical time PlanStep to each transaction planned through this coordinator. Each transaction is assigned exactly one coordinator, selected by hashing its TxId.

Mediator

The **Mediator** is a system tablet that distributes the transactions planned by coordinators to the transaction participants (usually, DataShards). Mediators ensure the advancement of global time. Each transaction participant is associated with exactly one mediator. Mediators allow to avoid the need for a full mesh of connections between all coordinators and all participants in all transactions.

Hive

Hive is a system tablet responsible for launching and managing other tablets. Its responsibilities include moving tablets between nodes in case of failure or overload of a node. For more details about Hive, see the dedicated article.

CMS

CMS or **cluster management system** is a system tablet responsible for managing information about the current state of the YDB cluster. This information is used for performing rolling restarts of the cluster without affecting user workloads, maintenance, cluster reconfiguration, etc. CMS includes Sentinel, a component that implements SelfHeal.

SelfHeal

SelfHeal is a set of mechanisms that automatically maintain and restore cluster fault tolerance. Storage SelfHeal relocates VDisks of storage groups after prolonged node or disk failures. Metadata Distribution SelfHeal relocates State Storage, Board, and SchemeBoard replicas after failures and can add replicas when new nodes appear.

NodeBroker

NodeBroker is a system tablet responsible for registering dynamic nodes in the cluster.

BSController

BSController (also known as **blob storage controller**) manages the dynamic configuration of the distributed storage, including information about PDisk, VDisk, and storage groups. It interacts with node warden to start various distributed storage components. It interacts with Hive to allocate channels to tablets.

Console

Console is a system tablet responsible for storing the dynamic configuration and delivering it to cluster nodes.

Kesus

Kesus is a tablet that implements a coordination node.

SysViewProcessor

SysViewProcessor is a tablet that stores data for some of the system views.

SequenceShard

SequenceShard is a tablet that serves Sequence objects used to implement serial data types.

ReplicationController

ReplicationController is a tablet responsible for the asynchronous replication process.

StatisticsAggregator

StatisticsAggregator is a tablet responsible for collecting statistics used in cost optimization.

Slot

A **slot** in YDB can be used in two contexts:

- **Slot** is a portion of server resources allocated to run one node of YDB. A typical slot size is 10 CPU cores and 50 GB of RAM. Slots are used when the YDB cluster is deployed on servers or virtual machines with sufficient resources to host multiple slots.
- **VDisk slot** or **VSlot** is a portion of a PDisk that can be allocated to one of the VDisk instances.

State storage

State storage (also known as **StateStorage**) is a distributed service that stores information about tablets, namely:

- The current leader of the tablet or its absence.
- Tablet followers.
- Generation and step of the tablet `(generation:step)`.

State storage is used as a service for resolving tablet names, i.e., to obtain an ActorId from a TabletID. StateStorage is also used in the tablet leader election process.

The information in the state storage is volatile. Thus, it is lost on power failure or process restart. Despite its name, this service is not a permanent long-term storage. It only contains information that is easy to recover and that does not need to be durable. However, the state storage stores information on multiple nodes to minimize the impact of node failures. This service can also be used to gather a quorum, which is used for tablet leader election.

Due to its nature, the state storage service operates on a best-effort basis. For example, the absence of multiple tablet leaders is guaranteed through the leader election protocol on the distributed storage, not on the state storage.

Board

Board is a distributed service designed to store metadata as key-value pairs. It is used, among other things, to store information about endpoints.

SchemeBoard

SchemeBoard is a distributed service designed to store metadata as key-value pairs. It is used, among other things, to store information about schemas.

Compaction

Compaction is the internal background process of rebuilding LSM tree data. The data in VDisks and local databases are organized in the form of an LSM tree. Therefore, there is a distinction between **VDisk compaction** and **Tablet compaction**. The compaction process is usually quite resource-intensive, so efforts are made to minimize the overhead associated with it, for example, by limiting the number of concurrent compactions.

gRPC proxy

gRPC proxy — a proxy system for external user requests. Client requests enter the system via the gRPC protocol, then the proxy component translates them into internal calls to execute these requests, transmitted via interconnect. This proxy provides an interface for both request-response and bidirectional streaming data.

Distributed configuration

Distributed configuration or **DistConf** — an internal mechanism of cluster configuration that ensures the startup and configuration of static nodes, automatic management of the static storage group and State Storage. Distributed configuration starts before any tablets, storage groups, or State Storage.

For more on how distributed configuration works, see Internals of the V2 configuration mechanism.

Distributed storage implementation

Distributed storage — a distributed fault-tolerant data storage layer that stores binary records called LogoBlob, addressed by a specific type of identifier called LogoBlobID. Thus, Distributed storage is a key-value store that maps a LogoBlobID to a string of up to 10 MB. Distributed storage consists of many storage groups, each of which is an independent data repository.

Distributed storage stores immutable data, with each immutable data block identified by a specific LogoBlobID key. The Distributed storage API is very specific, intended only for use by tablets to store their data and change logs. Thus, it is not intended for general-purpose data storage. Data in Distributed storage is deleted using special barrier commands. Due to the absence of mutations in its interface, Distributed storage can be implemented without implementing distributed consensus. Distributed storage is just one of the components that tablets use to implement distributed consensus.

LogoBlob

LogoBlob — a set of binary immutable data, identified by LogoBlobID and stored in Distributed storage. The data block size is limited at the VDisk level and above in the stack. Currently, the maximum data block size that a VDisk can handle is 10 MB.

LogoBlobID

LogoBlobID — an identifier of a LogoBlob in Distributed storage. It has a structure of the form `[TabletID, Generation, Step, Channel, Cookie, BlobSize, PartID]`. The main elements of LogoBlobID:

- `TabletID` — the ID of the tablet that owns the LogoBlob.
- `Generation` — the generation of the tablet in which the data block was written.
- `Channel` — the channel of the tablet on which the LogoBlob is written.
- `Step` — an incremental counter, usually within the tablet generation.
- `Cookie` — a unique identifier of a data block within a single `Step`. Cookie is typically used when writing multiple data blocks into one `Step`.
- `BlobSize` — the size of the LogoBlob.
- `PartID` — the identifier of a data block part. It is important when the original LogoBlob is split into parts using erasure coding, and the parts are written to the corresponding VDisk and storage groups.

Replication

Replication — a process that ensures a sufficient number of copies (replicas) of data to maintain the desired availability characteristics of the cluster YDB. Typically used in geo-distributed clusters YDB.

Erasure Coding

Erasure coding is a data encoding method where the original data is supplemented with redundancy and split into multiple fragments, enabling recovery of the original data if one or more fragments are lost. It is widely used in clusters YDB with a single availability zone, as opposed to replication with 3 replicas. For example, the most popular erasure coding scheme 4+2 provides the same reliability as three replicas, with a space overhead of 1.5 versus 3.

PDisk

PDisk, **physical disk** is a component that controls a physical disk drive (block device). In other words, PDisk is a subsystem that implements an abstraction similar to a specialized file system on top of block devices (or files emulating a block device for testing purposes). PDisk provides data integrity control (including erasure coding of sector groups to recover data on individual damaged

sectors, integrity checking using checksums), transparent encryption of all data on the disk, and transactional guarantees for disk operations (write confirmation strictly after `fsync`).

PDisk contains a scheduler that ensures shared use of the device's bandwidth among multiple clients (VDisk). PDisk divides the block device into blocks called slots (about 128 megabytes in size; smaller blocks are also allowed). At any given time, no more than one VDisk can own each slot. PDisk also maintains a recovery log shared by PDisk service records and all VDisks.

VDisk

VDisk, **virtual disk** is a component that implements data storage of distributed storage LogoBlob on PDisk. VDisk stores all its data on PDisk. One VDisk corresponds to one PDisk, but typically several VDisks are associated with one PDisk. Unlike PDisk, which hides blocks and logs behind it, VDisk provides an interface at the LogoBlob and LogoBlobID level, for example writing a LogoBlob, reading LogoBlobID data, and deleting a set of LogoBlobs using a special command. VDisk is a member of a storage group. VDisk itself is local, but many VDisks in a given group provide reliable data storage. VDisks in a group synchronize data with each other and replicate data in case of losses. The set of VDisks in a storage group forms a distributed RAID.

Yard

Yard is the name of the PDisk API. It allows VDisk to read and write data to blocks and logs, reserve blocks, delete blocks, and transactionally acquire and release ownership of blocks. In some contexts, Yard can be considered a synonym for PDisk.

Skeleton

A **Skeleton** is an actor that provides an interface to a VDisk.

SkeletonFront

SkeletonFront is a proxy actor for Skeleton that controls the flow of messages coming to Skeleton.

Distributed storage controller

The **distributed storage controller** or **DS controller** manages the dynamic configuration of distributed storage, including information about PDisks, VDisks, and storage groups. It interacts with node wardens to launch various distributed storage components. It interacts with Hive to allocate channels to tablets.

Proxy

Distributed storage proxy, **DS-proxy**, or **BS-proxy** acts as a client library for performing operations with distributed storage. The users of DS-proxy are tablets that write to and read from distributed storage. DS-proxy hides the distributed nature of distributed storage from the user. The task of DS-proxy is to write to a quorum of VDisk, perform retries when necessary, and control the write/read flow to prevent VDisk overload.

Technically, the DS proxy is implemented as an actor service launched by node warden on each node for each storage group, handling all requests to the group (writing, reading, and deleting LogoBlob, and group locking). When writing data, the DS proxy performs erasure coding of the data, splitting the LogoBlob into parts that are then sent to the corresponding VDisks. The DS proxy performs the reverse process when reading, receiving parts from VDisks and reconstructing the LogoBlob from them.

Node warden

Node warden or `BS_NODE` is an actor service on each cluster node that launches PDisks, VDisks, and DS proxies of static storage groups when the node starts. It also interacts with the DS controller to launch PDisk, VDisk, and DS proxies of dynamic groups. The DS proxy of dynamic groups is launched on demand: node warden processes undelivered messages to DS proxies, launches the corresponding DS proxies, and receives group configuration from the DS controller.

Failure realm

A **fail realm** is a set of failure domains that can fail simultaneously due to a common cause. A correlated failure of two VDisks in the same fail realm is more likely than a failure of two VDisks from different fail realms.

An example of a fail realm is a set of equipment located in a single data center, or availability zone, which can fail entirely due to a natural disaster, large-scale power outage, or other similar event.

Failure domain

A **fail domain** is a set of hardware that may fail simultaneously. The correlated failure of two VDisks within the same fail domain is more probable than the failure of two VDisks from different fail domains. In the case of different fail domains, this probability is also affected by whether these domains belong to the same fail realm or not.

An example of a fail domain is a set of disks connected to a single server, since all disks of a particular server may become unavailable if the server's power supply or network controller fails. Typically, all servers located in a single server rack are considered a common fail domain, because power or network issues at the rack level cause all equipment in it to become unavailable. Thus, a typical fail domain corresponds to a server rack (if the cluster is configured with rack-aware topology) or an individual server.

Failures at the fail domain level are automatically handled by YDB without stopping the cluster.

Distributed storage channel

A **distributed storage channel**, **DS channel**, or **channel** is a logical connection between a tablet and a storage group. A tablet can write data to different channels, and each channel maps to a specific storage group. Having multiple channels allows a tablet to:

- Write more data than a single storage group can hold.
- Store different LogoBlobs in different storage groups, with different properties, such as erasure coding or on different media (HDD, SSD, NVMe).

Distributed transactions implementation

Below are the terms related to the implementation of distributed transactions. The implementation itself is described in a separate article DataShard: distributed transactions.

Deterministic transactions

Distributed transactions in YDB are inspired by the research paper Building Deterministic Transaction Processing Systems without Deterministic Thread Scheduling by Alexander Thomson and Daniel J. Abadi from Yale University. The paper introduced the concept of **deterministic transaction processing**, which allows efficient processing of distributed transactions. The original paper imposed restrictions on the types of operations that could be performed this way. Since these restrictions hindered real-world user scenarios, YDB evolved its algorithms to handle them, using deterministic transactions as stages for executing user transactions with additional orchestration and locking.

Optimistic locking

As in many other database management systems, queries in YDB can place locks on certain data fragments, such as table rows, to ensure that concurrent changes do not lead to an inconsistent state. However, YDB checks these locks not at the start of transactions, but when attempting to commit them. The first approach is called **pessimistic locking** (for example, used in PostgreSQL), and the second is called **optimistic locking** (used in YDB).

Transaction lock invalidation

Transaction Lock Invalidation (**TLI**) is the normal behavior of YDB when parallel transactions conflict under optimistic locks. If one transaction (the violator) writes data and thereby breaks the locks of another transaction (the victim), YDB detects this when the victim commits and aborts it with error `transaction locks invalidated`. For more details on TLI diagnostics, see Transaction lock invalidation.

Prepare stage

Preparation phase is the transaction phase during which the transaction body is registered on all participating shards.

Execute stage

Execution phase is the transaction phase during which the scheduled transaction is executed and a response is generated.

In some cases, instead of prepare and execute, the transaction is immediately executed, and a response is generated. For example, this happens for transactions involving only one shard or consistent reads from a snapshot.

Dirty operations

In the case of read-only transactions, similar to "read uncommitted" in other database management systems, it might be necessary to read data that has not yet been committed to disk. This is called **dirty operations**.

Read-write set

Read-write set or **RW-set** is a data set that will participate in the execution of a distributed transaction. It combines the read set data that will be read and the write set for which modifications will be performed.

Read set

Read set or **ReadSet data** is what participating shards send during transaction execution. In the case of data transactions, it may contain information about the state of optimistic locks, the shard's readiness to commit, or a decision to abort the transaction.

Transaction proxy

Transaction proxy or `TX_PROXY` is a service that orchestrates the execution of many distributed transactions: sequential phases, phase execution, scheduling, and result aggregation. In the case of direct orchestration by other actors (for example, QP data transactions), it is used for caching and allocating unique TxIDs.

Transaction flags

Transaction flags or **TxFlags** is a bitmask of flags that somehow modify the execution of a transaction.

Transaction ID

TxID is a unique identifier assigned to each transaction when it is accepted by YDB.

Transaction order ID

Transaction order id is a unique identifier assigned to each transaction during scheduling. It consists of PlanStep and Transaction ID.

PlanStep

PlanStep or **Step** is the logical time at which the execution of a set of transactions is scheduled.

Mediator time

During the distributed query execution, **mediator time** is the logical time before which (inclusive) the shard participant must know the entire execution plan. It is used to advance the time in the absence of transactions on a particular shard, to determine whether it can read from a snapshot.

MiniKQL

MiniKQL is a language that allows expressing a single deterministic transaction in the system. It is a functional, strongly typed language. Conceptually, the language describes a graph of reading from the database, performing computations on the read data, and writing results to the database and/or to a special document representing the query result (for display to the user). A MiniKQL transaction must explicitly specify its read set (data to be read) and assume deterministic selection of execution branches (for example, no randomness).

MiniKQL is a low-level language. The system's end users only see queries in the YQL language, which relies on MiniKQL in its implementation.

Query Processor

Query Processor or **QP** (formerly **KQP**) is a YDB component responsible for orchestrating the execution of user queries and generating the final response.

Global schema

Global scheme, **global schema**, or **database schema** is the schema of all data stored in a database. It consists of tables and other entities such as topics. The metadata about these entities is called the global schema. The term is used in contrast to **local schema**, which refers to the data schema inside a tablet. YDB users never see the local schema and work only with the global schema.

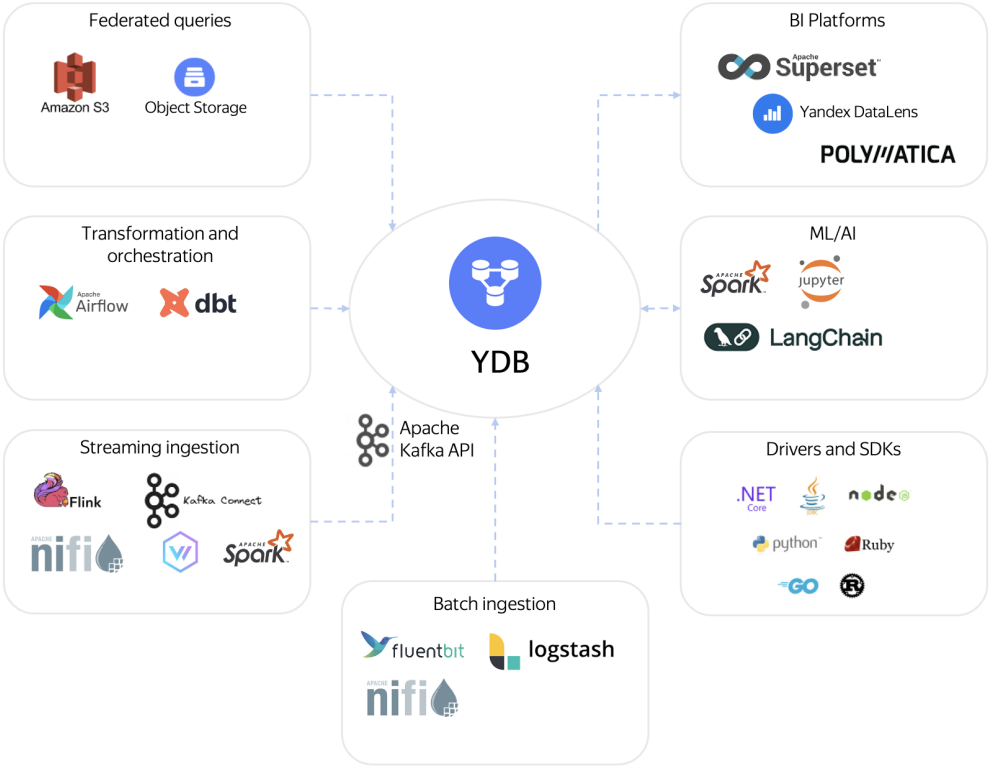
KiKiMR

KiKiMR is the former name of YDB, used before it became an open source product. It can still be found in source code, old articles, videos, etc.

Overview

YDB is a distributed, fault-tolerant database that provides components for building a data warehouse (DWH) on a unified platform.

Using YDB allows you to consolidate the functionality of several technologies (for example, separate systems for streaming, storage, and analytics) into a single solution. You can use familiar tools and approaches while gaining the benefits of a distributed system.



Data Ingestion

The platform is designed to handle streaming and batch data ingestion at large scale.

- Streaming processing: Built-in topic system with Kafka API support for integration with existing systems. Plugins for Fluent Bit and Logstash are available for log collection.
- Batch loading: BulkUpsert API for efficiently loading large data sets and a connector for Apache Spark to integrate with data processing platforms.
- Connection via standard interfaces: JDBC-driver and native SDK.

Data Storage

The core of the storage system is — column-oriented tables with built-in compression, optimized for analytical workloads.

- Separation of storage and compute: A key feature of YDB, enabling independent scaling of disk space and computing resources.
- Minimized administration: Background processes for (compaction) and TTL-based data removal reduce the need for manual operations.

Query Execution

YDB is an Massively Parallel Processing (MPP) DBMS with no dedicated master node. All nodes perform the same roles, and the system scales horizontally by dynamically adding or removing compute resources.

- Cost-Based Optimizer (CBO): Selects the optimal query execution plan by analyzing data statistics.
- Spilling mechanism: Enables execution of queries whose intermediate results do not fit in RAM by offloading them to disk.
- Workload Manager: Manages resource allocation among queries, isolating different types of workloads.

Data Transformation

Data transformation is supported using standard approaches and tools.

- ELT with SQL: Use `INSERT INTO ... SELECT` to build data marts. For managing complex SQL pipelines, integration with dbt is available.
- ETL with Apache Spark: Run ETL jobs on Apache Spark using the parallel connector.
- Orchestration: Automate pipelines with Apache Airflow.

Federated Queries

YDB allows you to run queries on data stored in S3-compatible storage without preloading it. This simplifies working with data stored in a data lake.

Data Analysis and Visualization (BI and ML)

You can use industry-standard tools for data analysis:

- BI tools: Yandex DataLens, Apache Superset, Grafana.
- ML tools: Use Jupyter Notebook and Apache Spark for data preparation and machine learning model training.

Data Ingestion

YDB is designed to ingest both streaming and batch data. The absence of dedicated master nodes allows for parallel writes to all database nodes, enabling write throughput to scale linearly as the cluster grows. The choice of tool depends on the requirements for latency, delivery guarantees, and data volume.

Streaming Ingestion (Real-time)

For scenarios requiring minimal latency, such as logs, metrics, and CDC streams.

- Topics with Kafka API: the primary and recommended method for streaming ingestion. Topics are the YDB built-in equivalent of Apache Kafka. Thanks to Kafka API support, you can use existing clients and systems (Apache Flink, Spark Streaming, Kafka Connect) without any changes. The key advantage is the ability to perform transactional writes from a topic to a table, which guarantees `exactly-once` semantics at the database level.
- Plugins for Fluent Bit / Logstash: if you use Fluent Bit or Logstash for log collection, specialized plugins allow you to write data directly to YDB, bypassing intermediate message brokers.
- Built-in data transfer (Transfer): the Transfer service allows you to transform and move data from topics to tables in streaming mode.

Batch Ingestion (Batch)

For loading large volumes of historical data, exports from other systems, or the results of batch jobs.

- BulkUpsert - the most performant method for batch inserts. BulkUpsert is a specialized API optimized for maximum throughput. It requires fewer resources compared to transactional operations, allowing you to load large datasets at maximum speed.
- Federated queries to data in S3 / Data Lakes - YDB allows you to execute SQL queries directly against data stored in S3-compatible object storage or other external systems. This is a convenient way to load data without using separate ETL tools.
- Apache Spark connector saves data directly to YDB tables in multi-threaded mode for the most performant writes.
- JDBC driver and native SDKs- with these, you can connect any applications or pipelines, including Apache Spark, Apache NiFi, and other solutions.

Data Storage

Efficient data storage is the foundation of any analytical warehouse. YDB uses a columnar format, a storage and compute disaggregation architecture, and automatic maintenance processes to ensure high performance and a low total cost of ownership.

Columnar tables

Data in columnar tables is stored by columns instead of rows. This approach is the standard for OLAP systems and offers two key advantages:

1. Reduced read volume: when a query (e.g., `SELECT column_a, column_b FROM...`) is executed, only the data from the columns involved in the query is read from the disk.
2. Data compression: columnar storage lets you apply compression and encoding per column, which compresses data better than heterogeneous row data. By default, the `LZ4` compression algorithm is used without special encoding; both settings can be configured per column. For details, see CREATE TABLE request parameters.

Architecture with storage and compute disaggregation

Storage and compute disaggregation is an architectural principle of YDB. The nodes responsible for data storage (storage nodes) and the nodes that execute queries (dynamic nodes) are separate. This allows you to:

- scale resources independently: if you run out of disk space, you add storage nodes. If you lack CPU for queries, you add compute nodes. This differs from systems where storage and compute resources are tightly coupled;
- redistribute load quickly: redistributing compute load between nodes does not require physical data movement; only metadata is transferred.

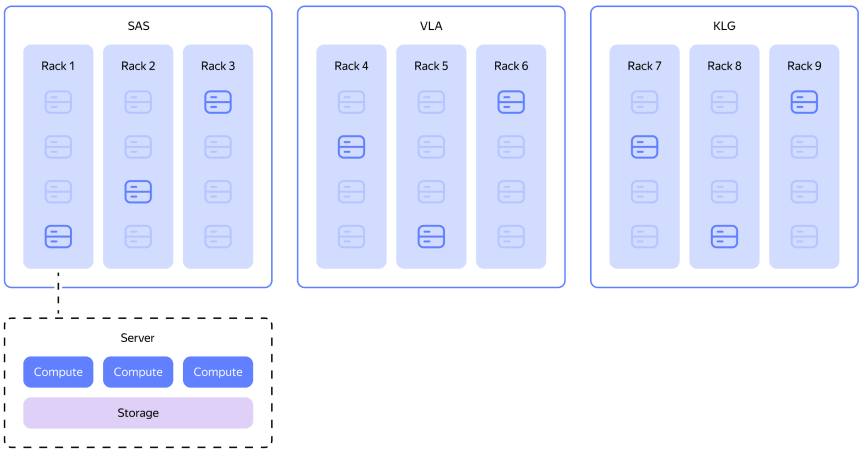
Automatic storage optimization

YDB is designed to minimize manual maintenance operations.

- Automatic data compaction: Data is stored in LSM-like structures; data merging and optimization processes run continuously in the background. You do not need to run VACUUM or similar commands.
- Automatic data deletion: To manage the data lifecycle, use the TTL-based deletion mechanism.

Built-in fault tolerance

YDB was designed from the ground up as a fault-tolerant system and supports various data placement modes to protect against hardware, rack, or even entire data center failures.



Query Execution

YDB is a distributed Massively Parallel Processing (MPP) database designed for executing complex analytical queries on large volumes of data. Each query is automatically parallelized across all available compute nodes in the cluster, enabling efficient use of compute resources.

YDB supports several key technologies that ensure high performance and stability:

- Decentralized MPP architecture
- Cost-Based Query Optimizer
- Handling data that exceeds RAM
- Workload Isolation and Resource Management

Decentralized MPP architecture

Unlike MPP systems with a dedicated master node, YDB's architecture is completely decentralized. This provides two main advantages:

- High fault tolerance: any node in the cluster can accept and coordinate query execution. There is no single point of failure (SPOF). The failure of some nodes does not halt the cluster—the load is automatically redistributed among the remaining nodes.
- Compute scalability: you can add or remove compute nodes without downtime, and the system automatically adapts, distributing the load to account for the new cluster composition.

Cost-Based Query Optimizer

Before executing a query, YDB uses a Cost-Based Optimizer (CBO). It analyzes the query text, metadata, and statistics on data distribution in tables to build a physical execution plan with the lowest estimated cost.

The optimizer can:

- choose the join order for queries with dozens of `JOIN`s;
- select distributed `JOIN` algorithms (e.g., Grace Join, Broadcast Join) depending on table sizes;
- push down filters (`WHERE` clauses) as close as possible to the data sources to reduce the amount of data processed in subsequent stages.

Handling data that exceeds RAM

Analytical queries can require large amounts of RAM, especially for `JOIN` and `GROUP BY` operations. YDB is designed to work with data that may not fit into RAM.

- Spilling: if the intermediate results of a query exceed the memory limit, YDB automatically spills them to the node's local disk. This prevents the query from failing with an "Out of Memory" error and allows queries to be executed on large volumes of data.
- Distributed JOIN algorithms: for joining tables that exceed the memory of a single node, distributed algorithms are used that process data in chunks across different nodes.

Workload Isolation and Resource Management

In a corporate DWH, different teams often run different types of workloads. To prevent these workloads from interfering with each other, YDB has a built-in resource manager.

- Workload Manager: the built-in `workload manager` allows you to create resource pools and, using classifiers, assign queries from different user groups to different pools. This mechanism solves the "noisy neighbor" problem, where a single resource-intensive query can slow down the system for all other users.

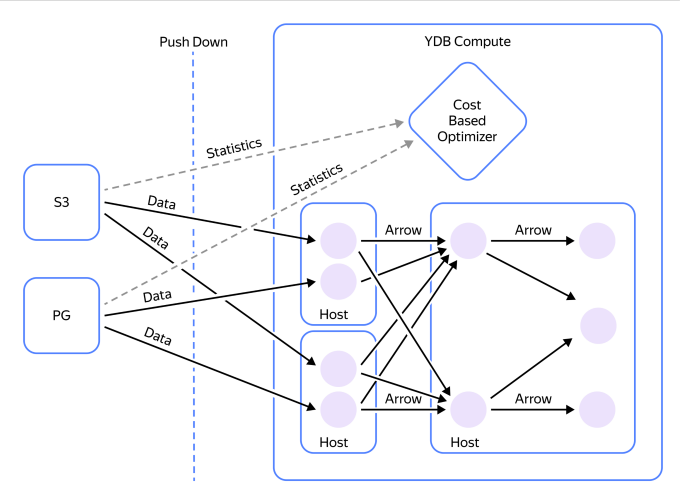
Federated queries

Federated queries allow you to query data stored in external systems without first loading it (ETL) into YDB. The most popular use case is working with data in S3-compatible object storage.

How it works

You can create an external table in YDB that references data in S3. When you execute a SELECT query against such a table, YDB initiates a parallel read from all compute nodes. Each node reads and processes only the portion of data it needs.

- Supported formats: Parquet, CSV, JSON with various compression algorithms.
- Read optimization: YDB uses S3 data read optimization mechanisms (partition pruning) for Hive-style partitioning and for more complex partitioning schemes.



Data transformation and preparation (ETL/ELT)

Data preparation for analysis is a key stage in building a data warehouse. YDB supports all standard data transformation approaches, allowing you to choose the most suitable tool for a specific task: from pure SQL to complex pipelines on Apache Spark.

ELT

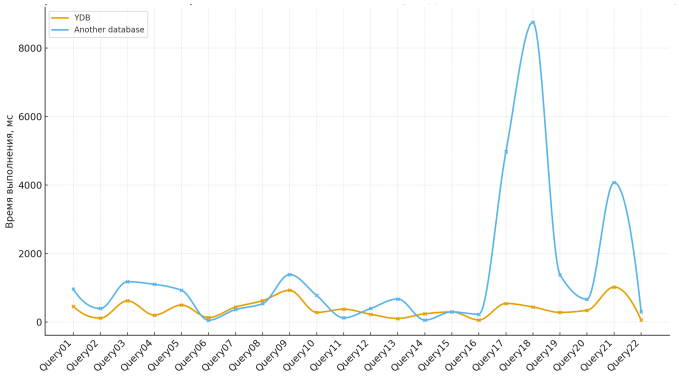
Data transformations using SQL are often the most performant, since all processing occurs directly within the YDB engine without moving data to and from external systems. The logic is described in SQL and executed by the distributed MPP engine, which is optimized for analytical operations.

Performance in the TPC-H benchmark

The performance of ELT operations directly depends on the execution speed of complex analytical queries. The industry-standard benchmark for evaluating such queries is TPC-H.

A comparison with another distributed analytical DBMS on the TPC-H query set shows that YDB demonstrates more stable performance, especially when executing queries that contain:

- connections (`JOIN`) of a large number of tables (five or more);
- nested subqueries used for filtering;
- aggregations (`GROUP BY`) followed by complex filtering of the results.



This stability indicates the high efficiency of the YDB cost-based query optimizer in building execution plans for complex SQL patterns typical of real-world ELT processes. For a data warehouse (DWH) platform, this means predictable data update times and a reduced risk of uncontrolled performance degradation in the production environment.

Key use cases

- Building data marts: use the familiar `INSERT INTO ... SELECT FROM ...` syntax to create aggregated tables (data marts) from raw data;
- joining OLTP and OLAP data: YDB allows you to join data from both transactional (row-based) and analytical (column-based) tables in a single query. This enables you to enrich "cold" analytical data with up-to-date information from the OLTP system without the need for duplication;
- bulk updates: for "blind" writes of large volumes of data without existence checks, you can use the `UPSERT INTO` operator.

Managing SQL pipelines with dbt

To manage complex SQL pipelines, use the dbt plugin. This plugin allows data engineers to describe data models as `SELECT` queries, and dbt automatically builds a dependency graph between models and executes them in the correct order. This approach helps implement software engineering principles (testing, documentation, versioning) when working with SQL code.

ETL

Complex transformations using external frameworks

For tasks that require complex logic in programming languages (Python, Scala, Java), integration with ML pipelines, or processing large volumes of data, it is convenient to use external frameworks for distributed processing.

Apache Spark is one of the most popular tools for such tasks, and a dedicated connector to YDB has been developed for it. If your company uses other similar solutions (e.g., Apache Flink), they can also be used to build ETL processes using the JDBC driver.

A key advantage of YDB when working with such systems is its architecture, which allows for parallel data reading. YDB has no dedicated master node for exports, so external tools can read information directly from all storage nodes. This ensures high-speed reads and linear scalability.

Pipeline orchestration

Orchestrators are used to run pipelines on a schedule and manage dependencies.

- Apache Airflow: an Apache Airflow provider is supported for orchestrating data in YDB. It can be used to create DAGs that run `dbt run` , execute YQL scripts, or initiate Spark jobs.
- built-in mechanisms: For some tasks, an external orchestrator is not required. YDB can perform some operations automatically:
  - TTL-based data expiration: automatically cleans up partitions after a specified time;
  - automatic compaction: data merging and optimization processes for the LSM tree run in the background, eliminating the need to regularly run commands like `VACUUM` ;
- other orchestrators: if your company uses a different tool (e.g., Dagster, Prefect) or a custom scheduler, you can use it to run the same commands. Most orchestrators can execute shell scripts, allowing you to call the YDB CLI, dbt run, and other utilities.

Integration with other ETL tools via JDBC

YDB provides a JDBC driver, enabling the use of a wide range of existing ETL tools, such as Apache NiFi and other JDBC-compliant systems.

BI analytics and data visualization

The interactivity of BI dashboards directly depends on the performance of the underlying database. YDB was designed as a high-performance analytical platform that executes queries in sub-second time, enabling analysts to work with data interactively.

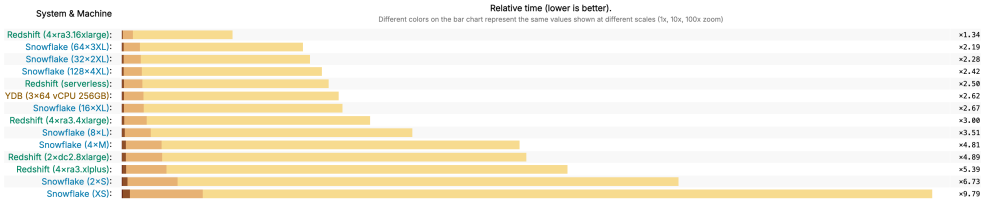
This is achieved through key architectural features:

- columnar storage: queries read only the columns specified in the request from disk, which reduces the volume of I/O operations;
- MPP architecture: each query is parallelized across all available compute nodes of the cluster, harnessing all available resources for its execution;
- decentralized architecture: the absence of a single master node enables efficient processing of multiple concurrent queries from BI systems.

Performance in independent benchmarks

Although synthetic tests do not always reflect real-world workloads, they serve as a good starting point for performance comparison. ClickBench is an independent benchmark for analytical DBMSs, developed by the creators of ClickHouse.

On a set of 43 analytical queries, YDB shows competitive results, outperforming many popular open-source and cloud analytical databases. This confirms the engine's high performance on typical OLAP queries.



Integrations with BI platforms

YDB supports the following BI platforms:

- Yandex DataLens;
- Apache Superset;
- Grafana;
- Polymatica.

Machine Learning

YDB serves as an effective platform for storing and processing data in ML pipelines. You can use familiar tools, such as Jupyter Notebook and Apache Spark, throughout all stages of the ML model lifecycle.

Feature Engineering

Use YDB as an engine for feature engineering:

- SQL and dbt: execute complex analytical queries to aggregate raw data and create new features. Materialize feature sets into row-based tables for fast access;
- Apache Spark: for more complex transformations that require Python or Scala logic, use the Apache Spark connector to read data, process it, and save the results back to YDB.

Model Training

YDB can serve as a fast and scalable data source for model training:

- Jupyter Integration: connect to YDB from Jupyter Notebook for ad-hoc analysis and model prototyping;
- distributed training: the Apache Spark connector enables parallel reading of data from all cluster nodes directly into a Spark DataFrame. This allows you to load training sets for models in PySpark MLlib, CatBoost, Scikit-learn, and other libraries.

Online Feature Store

The combination of row-based (OLTP) and columnar (OLAP) tables in YDB allows you to implement not only an analytical warehouse but also an Online Feature Store on a single platform.

- Use row-based (OLTP) tables to store features that require low-latency point reads; this allows ML models to retrieve features in real time for inference.
- Use columnar (OLAP) tables to store historical data and for the batch calculation of these features.

Key Analytics Capabilities: Quick Reference

This page is a map of the documentation for the analytical features of YDB. The content is organized by the stages of the data lifecycle to help you quickly find the information you need for designing, developing, and operating analytical solutions.

Data Warehouse Design (Concepts & Design)

Core concepts for organizing, scaling, and managing data.

Main Concepts and Data Types

- Column-Oriented Tables: Storage architecture optimized for OLAP workloads.
- Data Types: Comprehensive reference for supported types.

Scalability and Performance

- Primary Key Design for Maximum Performance: How to choose `PRIMARY KEY` and `PARTITION BY`.
- Table Partitioning: Mechanism for distributing data across nodes.

Data Lifecycle Management

- TTL (Time-to-Live): Automatic deletion of outdated data after expiration.

Streaming Ingestion

- Topics (Kafka API): Native streaming using the Kafka protocol.
- Connector Fluent Bit: Direct log ingestion.

Batch Ingestion

- Apache Spark Connector: Read and write data for ETL/ELT tasks.
- BulkUpsert API: High-performance bulk inserts via SDK.

Integration with External Systems

- Federated Queries: Run queries on data in external systems (S3, ClickHouse, Postgres).
- Working with S3 via External Tables: Read and write Parquet/CSV data in Object Storage.

Data Processing and Transformation (ETL/ELT)

Query language and integration with orchestration tools.

YQL Query Language

- Full Reference for YQL: Syntax, functions, and operators.
- Date and Time Functions: Complete list and typical use cases.
- JSON Functions: Extracting data from JSON documents.

Pipeline Building Tools

- dbt (Data Build Tool) Integration: Managing ELT pipelines with SQL.
- Apache Airflow Integration: Orchestrating complex ETL/ELT processes.

Application Development & Integration (Development & SDKs)

Tools for application developers.

- YDB SDK Overview: Native SDKs for Go, Python, Java, C++, JavaScript.
- JDBC driver: Standard connectivity for the Java ecosystem.
- YDB CLI: Command-line tool for administration and query execution.

Data Analysis and Visualization

Integration with end-user tools.

BI Tools

- Apache Superset
- Grafana
- Yandex DataLens

Data Science Tools

- Jupyter Notebooks: Running YQL queries and interactive data analysis.

Operations & Performance Management

Administration, monitoring, security, and optimization.

Performance Management

- Query Plan Analysis (EXPLAIN): How to understand query execution plans and identify bottlenecks.
- Cost-Based Optimizer: Overview of how the query planner works.

Monitoring and Diagnostics

- YDB UI: Web interface for cluster monitoring and diagnostics.
- Metrics Reference: Full list of metrics for monitoring systems.
- Ready-to-use Grafana Dashboards: Templates for quick monitoring setup.

Security and Fault Tolerance

- Authentication and Authorization: User access configuration, including LDAP support.

Streaming data processing

This section describes the key concepts of stream processing (stream processing) in YDB:

- Streaming queries — a type of query for continuous processing of incoming events. Sources and sinks of data, delivery guarantees, limitations, and query management are described.
- Watermarks — a mechanism for tracking the progress of event processing over time.

Streaming queries

A streaming query is a type of query designed for continuous processing of an unbounded data stream (stream processing). Unlike regular queries, a streaming query does not terminate after returning a result but runs continuously, processing data as it arrives.

Stream data processing is a widely used approach implemented in systems such as Apache Flink, Apache Kafka Streams, Amazon Kinesis Data Analytics. Typical use cases include monitoring and alerting, metric aggregation over time windows, real-time event transformation and filtering, enriching events with reference data, and pattern detection in event sequences.

YDB implements stream processing as part of a unified data platform. In addition to typical streaming query use cases, integration into the YDB common platform allows you to read data from topics YDB, process change streams from row tables via CDC, and write processing results to output topics or directly to tables. This enables building data processing pipelines within YDB with minimal latency.

Differences from regular queries

Regular queries work with data already stored in tables. A query executes, returns a result, and terminates. A streaming query is created and runs indefinitely until explicitly canceled by the user. Data continuously arrives in a topic, flows through the query, and is written to a sink — another topic or table.

Characteristic	Regular queries	Streaming queries
Data	Finite set in tables	Infinite stream of events
Lifetime	Terminates after processing	Runs continuously
Result	Available after completion	Updated as data arrives
Recovery on failures	Manual restart	Automatic recovery from checkpoint

Data sources and sinks

Streaming queries read data from topics YDB and write results to topics or tables YDB. Data can arrive in a topic from external systems (for example, an application writes events via the SDK), but the streaming query itself only works with YDB entities. Direct reading from external systems, such as Apache Kafka, or writing to tables in other databases, such as PostgreSQL, is not supported.

Sources

Topics are the main source of streaming data. The query reads messages from one or more topics and processes them as they arrive. Data can be written to a topic by:

- **External applications** — via the YDB SDK or Kafka API. For example, a service sends telemetry events or logs to a YDB topic, and the streaming query processes them.
- **CDC (Change Data Capture)** — change streams from tables, implemented via built-in topics. They allow reacting to inserts, updates, and deletes of records in real time. For more information, see Change Data Capture (CDC).

Sinks

Topics — for passing results to other systems or subsequent processing stages.

Tables — for materializing results. Data is saved via UPSERT and is available for regular SQL queries.

Guarantees

Under normal operation, streaming queries provide an at-least-once data delivery guarantee — on failures, the query automatically recovers from a checkpoint and resumes reading from saved offsets.

However, it is important to consider the limitations of the current implementation:

- **Offset reset on query recreation:** changing the query text via DROP + CREATE deletes the checkpoint. The new query starts reading from the end of the topic, skipping events that arrived between the deletion of the old version and the start of the new one.
- **Incomplete aggregates due to lack of watermarks:** due to the absence of a watermarks mechanism, time windows are closed based on system time (wall-clock). Events that arrive late (e.g., due to slow partitions or network lag) are not included in the aggregates of already closed windows.

Detailed description of guarantees, anomalies, and ways to minimize them — in the Data delivery guarantees section.

Note

We are constantly working on developing streaming processing mechanisms. In future versions, the guarantees provided will be improved.

Limitations

Warning

- The query must contain at least one read from a topic, as streaming processing requires a continuous input data stream.
- `JOIN` of two streams is not supported (temporary architectural limitation).

The following are also not supported in the current version:

- The important reader flag for consumers used by streaming queries.
- Autopartitioning (split/merge of partitions) of topics used by streaming queries. When the number of partitions in a topic from which a running streaming query reads increases, new partitions will not be processed.
- Changing the text of a running query. To change a query, you need to recreate it — delete it using DROP STREAMING QUERY and create it again using CREATE STREAMING QUERY.

Management

Streaming queries are created, modified, and deleted using YQL commands:

- CREATE STREAMING QUERY — creation.
- ALTER STREAMING QUERY — modification and state management.
- DROP STREAMING QUERY — deletion.

Query state is available in the system table `.sys/streaming_queries`.

Query language

Streaming queries are written in YQL and support familiar SQL constructs: SELECT, WHERE, GROUP BY, JOIN. For working with time windows, GROUP BY HOP is used.

A single streaming query can read multiple input topics, use the UNION ALL construct to combine data streams, and write the result to multiple output topics and/or tables (see more details in the articles Formats for writing data and Writing to tables).

See also

- Data delivery guarantees — guarantees, anomalies in window aggregation, and recommendations.
- Quick start: reading and writing to topics — step-by-step guide.
- Data formats for reading/writing from topics — data formats.

Watermarks

A watermark in stream processing (stream processing) is a monotonically increasing lower bound on the event times that may still arrive in the stream. When the watermark reaches a value X, the system declares that all events with time less than X have most likely been received.

YDB implements the watermark mechanism in streaming queries: they are used to correctly close time-based aggregation windows (HoppingWindow) and ensure that the window result is emitted only when the system has confirmed the completeness of input data for that period.

In stream processing, each event has two timestamps: **event time** — the moment when the event occurred in the real world, and **processing time** — the moment when the system received the event. Due to network delays, failures, and uneven load, these two values can differ significantly. This is precisely why the system needs a watermark mechanism: without it, the system does not know when a past time range can be considered sufficiently complete to emit a result.

Accuracy vs. Latency Trade-off

In YDB, this trade-off is explicitly controlled: the `WATERMARK` parameter in the `GROUP BY` clause specifies the expression for computing the watermark, including the amount of lag behind the last event time. This allows you to balance between result freshness and completeness of accounting for late events.

A watermark cannot simultaneously account for arbitrarily large event delays and advance quickly: the longer the system waits for late events, the later it emits results. This is a fundamental trade-off in stream processing.

Events that arrive after the watermark has passed the corresponding time range are considered late and are discarded. For more details on handling late events and configuring watermarks, see Watermarks.

YDB Architecture Overview

Introduction

YDB is a horizontally scalable, distributed, and fault-tolerant database system designed as a versatile platform for high performance — for example, a typical cluster node can process tens of thousands of queries per second. The system supports geographically distributed (cross-datacenter) configurations, ranging from small clusters with a few nodes to large-scale deployments of thousands of servers capable of efficiently handling hundreds of petabytes of data.

Key Features and Capabilities of YDB

- **Horizontal scaling and automatic sharding**: data and workload are dynamically distributed across available hardware resources as data volume or query intensity grows.
- **Fault tolerance**: automatic recovery from failures of nodes, racks, or availability zones.
- **Data high availability and durability**: ensured through automatic synchronous data replication within the cluster.
- **Strong consistency and ACID transactions**: the system provides distributed transactions with *serializable* isolation. Consistency and isolation levels can be relaxed when higher performance is required.
- **YQL**: a SQL dialect optimized for large-scale data and complex processing scenarios.
- **Relational data model**: supports both row-oriented and column-oriented tables, enabling efficient handling of both transactional (OLTP) and analytical (OLAP) workloads within a single system.
- **Hierarchical namespace**: tables, topics, and other objects are organized in a hierarchical namespace, similar to a filesystem.
- **Asynchronous replication**: near real-time data synchronization between YDB databases — both within a single cluster and across different clusters.
- **Streaming data processing and distribution**:
  - **Topics**: storage and streaming delivery of unstructured messages to multiple subscribers. Supports the Kafka protocol.
  - **Change Data Capture (CDC)**: built-in stream of table data changes published as a topic.
  - **Transfers**: automated data delivery from topics to tables.
- **Federated queries**: execute queries against external data sources (e.g., S3) as part of YQL queries, without prior data import to YDB storage.
- **Vector indexes**: support for storing and searching vector embeddings — ideal for semantic search, similarity matching, and ML use cases.
- **Observability**: built-in metrics, logs, and dashboards.
- **Security and audit**: data encryption (at-rest and in-transit), operation auditing, and support for authentication and authorization — see Security.
- **Tools, integrations, and APIs**: YDB CLI for running queries, administration, and debugging. SDKs for C++, C#, Go, Java, JavaScript, PHP, Python, and Rust. Integrations with various third-party systems. Learn more in Integrations YDB and Languages and APIs.
- **Open architecture**: source code is available under the Apache License 2.0. The system uses the open gRPC protocol, enabling client implementations in any programming language.

Key Use Cases

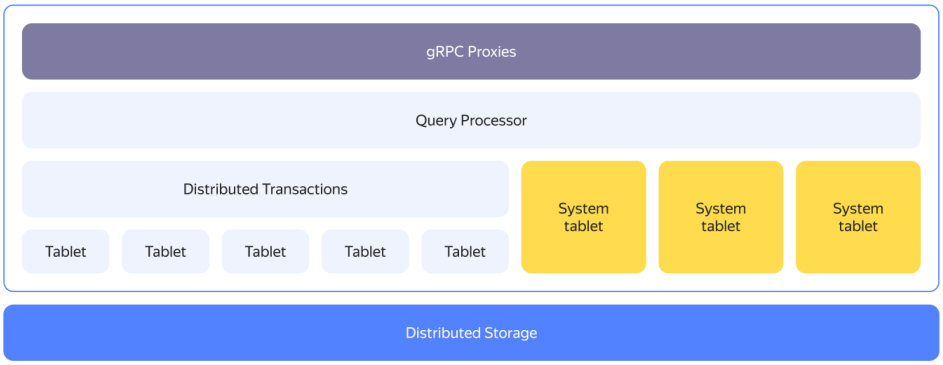
YDB is a versatile platform suitable for a wide range of scenarios requiring scalability, reliability, and flexibility. Typical use cases include:

- In distributed systems requiring **strong consistency or support for multi-row and multi-table transactions**. YDB combines NoSQL-like scalability with the consistency and integrity guarantees of relational databases.
- Systems that store and process **very large datasets** and require nearly unlimited horizontal scaling (production clusters with thousands of nodes, handling millions of RPS and petabytes of data).
- High-load systems relying on **manual sharding** of relational databases. YDB simplifies architecture by automatically handling the sharding logic, re-sharding, query routing, and cross-shard transactions out of the box.
- New product development with **uncertain load patterns** or expected scale beyond the limits of traditional relational database management systems (RDBMS).
- Projects requiring a **flexible platform** capable of handling diverse workloads and use cases — including transactional, streaming, and analytical.

How It Works?

Fully explaining how YDB works in detail takes quite a while. Below you can review several key highlights and then continue exploring the documentation to learn more.

YDB Architecture

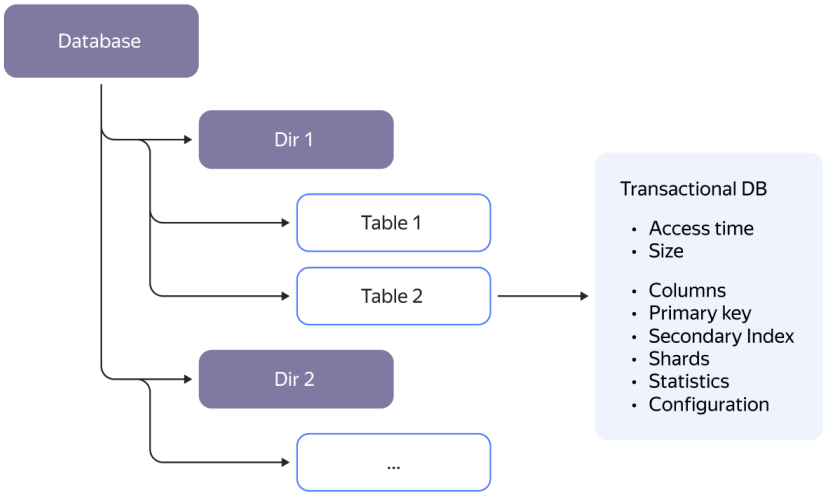


YDB clusters typically run on commodity hardware with a shared-nothing architecture. From a bird's eye view, YDB exhibits a layered architecture. The compute and storage layers are disaggregated; they can either run on separate sets of nodes or be co-located.

One of the key building blocks of YDB's compute layer is called a *tablet*. Tablets are stateful logical components implementing various aspects of YDB.

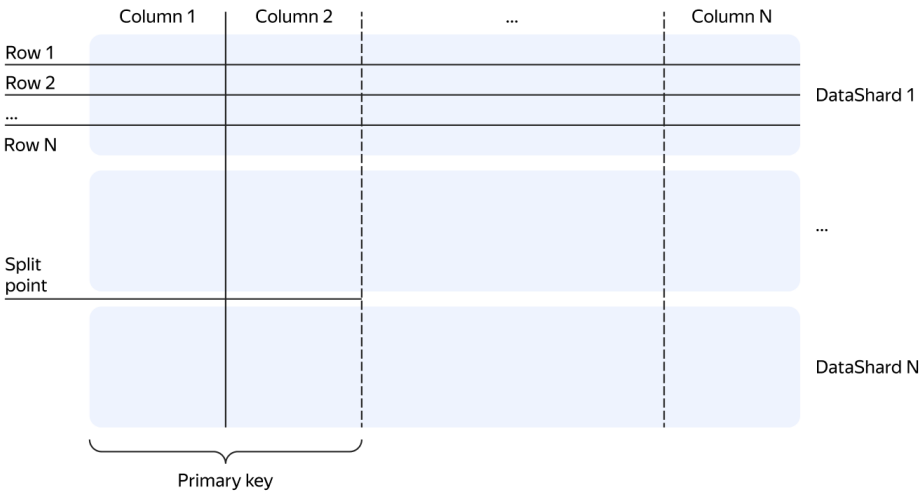
The next level of detail of the overall YDB architecture is explained in the General YDB schema article.

Hierarchy



From the user's perspective, everything inside YDB is organized in a hierarchical structure using directories. It can have arbitrary depth depending on how you choose to organize your data and projects. Even though YDB does not have a fixed hierarchy depth like in other SQL implementations, it will still feel familiar as this is exactly how any virtual filesystem looks.

Table



YDB provides users with a well-known abstraction — tables. In YDB, there are two main types of tables:

- Row-oriented tables are designed for OLTP workloads.
- Column-oriented tables are designed for OLAP workloads.

Logically, from the user's perspective, both types of tables look the same. The main difference between row-oriented and column-oriented tables lies in how the data is physically stored. In row-oriented tables, the values of all columns in each row are stored together. In contrast, in column-oriented tables, each column is stored separately, meaning that cells from different rows are stored next to each other within the same column.

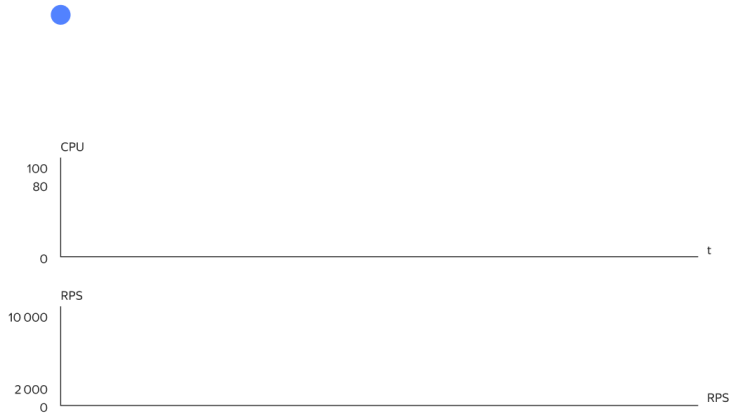
Regardless of the type, each table must have a primary key. Column-oriented tables can only have `NOT NULL` columns in primary keys. Table data is physically sorted by the primary key.

Partitioning works differently in row-oriented and column-oriented tables:

- Row-oriented tables are automatically partitioned by primary key ranges, depending on the data volume.
- Column-oriented tables are partitioned by the hash of the partitioning columns.

Each partition of a table is processed by a specific tablet, called a data shard for row-oriented tables and a column shard for column-oriented tables.

Split by Load

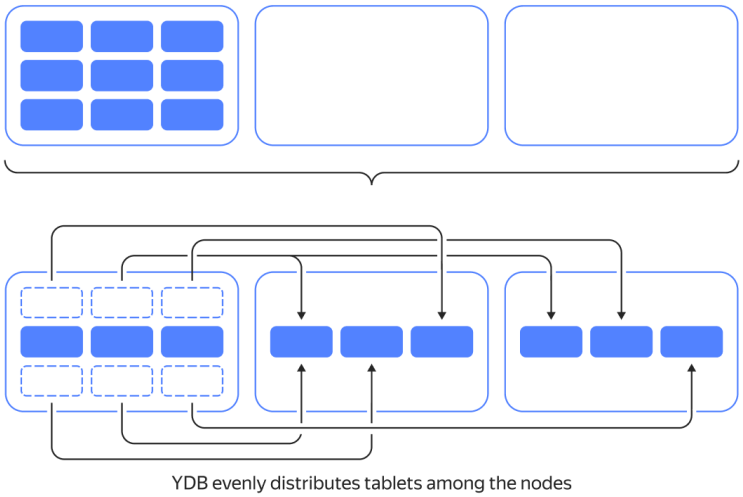


Data shards will automatically split into more as the load increases. They automatically merge back to the appropriate number when the peak load subsides.

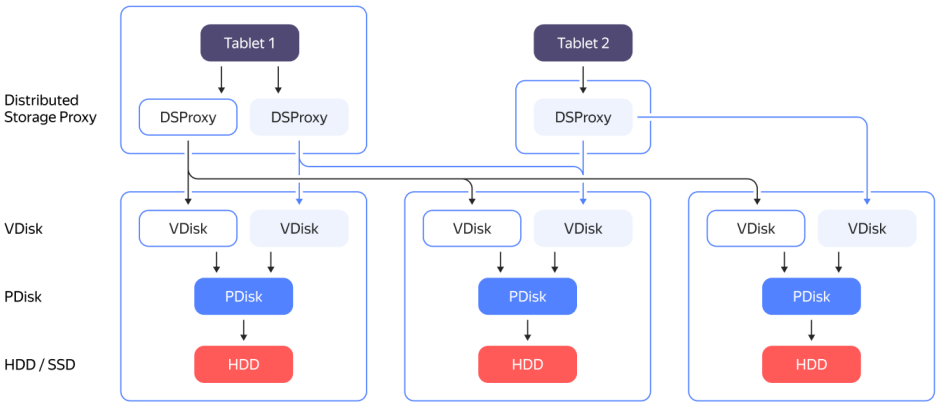
Split by Size

Data shards will also automatically split when the data size increases. They automatically merge back if enough data is deleted.

Automatic Balancing

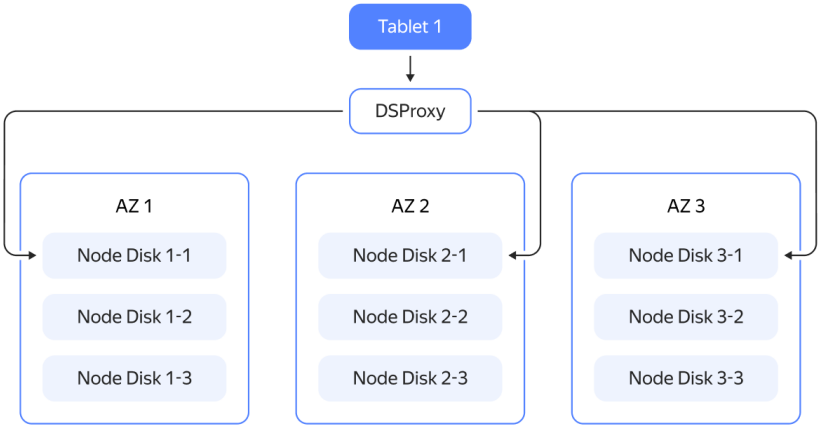


YDB evenly distributes tablets among available nodes. It moves heavily loaded tablets from overloaded nodes. CPU, memory, and network metrics are tracked to facilitate this.



YDB doesn't rely on any third-party filesystem. It stores data by directly working with disk drives as block devices. All major disk kinds are supported: NVMe, SSD, or HDD. The PDisk component is responsible for working with a specific block device. The abstraction layer above PDisk is called VDisk. There is a special component called DSProxy between a tablet and VDisk. DSProxy analyzes disk availability and characteristics and chooses which disks will handle a request and which won't.

Distributed Storage Proxy (DSProxy)



YDB writes data to 3 Availability Zones

A common fault-tolerant setup of YDB spans three datacenters or availability zones (AZ). When YDB writes data to three AZs, it doesn't send requests to obviously bad disks and continues to operate without interruption even if one AZ and a disk in another AZ are lost.

What's Next?

If you are interested in more specifics about various aspects of YDB, check out neighboring articles in this documentation section. If you are ready to jump into more practical content, you can continue to the quick start or YQL tutorials.

YDB Architecture Overview

Introduction

YDB is a horizontally scalable, distributed, and fault-tolerant database system designed as a versatile platform for high performance — for example, a typical cluster node can process tens of thousands of queries per second. The system supports geographically distributed (cross-datacenter) configurations, ranging from small clusters with a few nodes to large-scale deployments of thousands of servers capable of efficiently handling hundreds of petabytes of data.

Key Features and Capabilities of YDB

- **Horizontal scaling and automatic sharding**: data and workload are dynamically distributed across available hardware resources as data volume or query intensity grows.
- **Fault tolerance**: automatic recovery from failures of nodes, racks, or availability zones.
- **Data high availability and durability**: ensured through automatic synchronous data replication within the cluster.
- **Strong consistency and ACID transactions**: the system provides distributed transactions with *serializable* isolation. Consistency and isolation levels can be relaxed when higher performance is required.
- **YQL**: a SQL dialect optimized for large-scale data and complex processing scenarios.
- **Relational data model**: supports both row-oriented and column-oriented tables, enabling efficient handling of both transactional (OLTP) and analytical (OLAP) workloads within a single system.
- **Hierarchical namespace**: tables, topics, and other objects are organized in a hierarchical namespace, similar to a filesystem.
- **Asynchronous replication**: near real-time data synchronization between YDB databases — both within a single cluster and across different clusters.
- **Streaming data processing and distribution**:
  - **Topics**: storage and streaming delivery of unstructured messages to multiple subscribers. Supports the Kafka protocol.
  - **Change Data Capture (CDC)**: built-in stream of table data changes published as a topic.
  - **Transfers**: automated data delivery from topics to tables.
- **Federated queries**: execute queries against external data sources (e.g., S3) as part of YQL queries, without prior data import to YDB storage.
- **Vector indexes**: support for storing and searching vector embeddings — ideal for semantic search, similarity matching, and ML use cases.
- **Observability**: built-in metrics, logs, and dashboards.
- **Security and audit**: data encryption (at-rest and in-transit), operation auditing, and support for authentication and authorization — see Security.
- **Tools, integrations, and APIs**: YDB CLI for running queries, administration, and debugging. SDKs for C++, C#, Go, Java, JavaScript, PHP, Python, and Rust. Integrations with various third-party systems. Learn more in Integrations YDB and Languages and APIs.
- **Open architecture**: source code is available under the Apache License 2.0. The system uses the open gRPC protocol, enabling client implementations in any programming language.

Key Use Cases

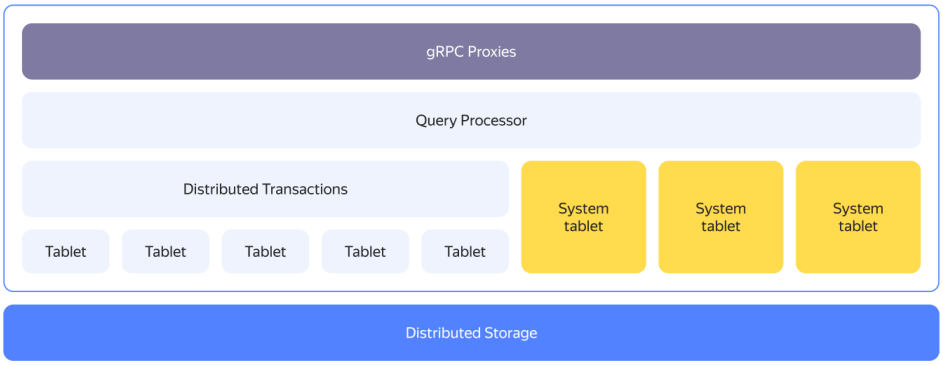
YDB is a versatile platform suitable for a wide range of scenarios requiring scalability, reliability, and flexibility. Typical use cases include:

- In distributed systems requiring **strong consistency or support for multi-row and multi-table transactions**. YDB combines NoSQL-like scalability with the consistency and integrity guarantees of relational databases.
- Systems that store and process **very large datasets** and require nearly unlimited horizontal scaling (production clusters with thousands of nodes, handling millions of RPS and petabytes of data).
- High-load systems relying on **manual sharding** of relational databases. YDB simplifies architecture by automatically handling the sharding logic, re-sharding, query routing, and cross-shard transactions out of the box.
- New product development with **uncertain load patterns** or expected scale beyond the limits of traditional relational database management systems (RDBMS).
- Projects requiring a **flexible platform** capable of handling diverse workloads and use cases — including transactional, streaming, and analytical.

How It Works?

Fully explaining how YDB works in detail takes quite a while. Below you can review several key highlights and then continue exploring the documentation to learn more.

YDB Architecture

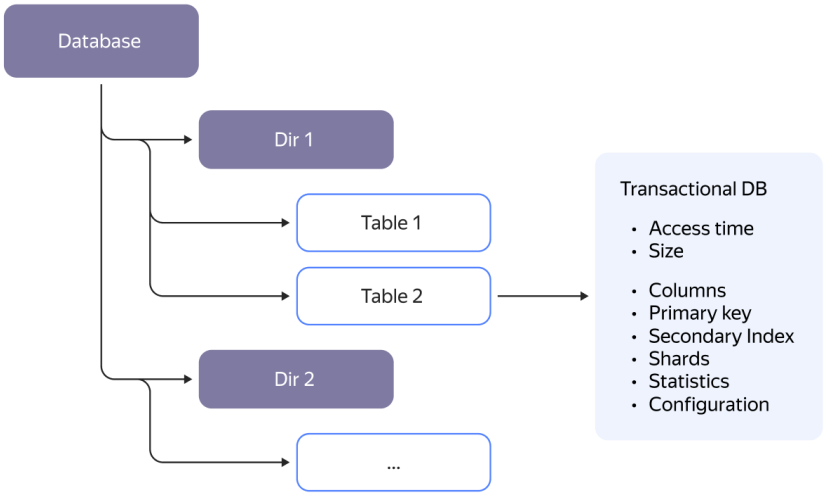


YDB clusters typically run on commodity hardware with a shared-nothing architecture. From a bird's eye view, YDB exhibits a layered architecture. The compute and storage layers are disaggregated; they can either run on separate sets of nodes or be co-located.

One of the key building blocks of YDB's compute layer is called a *tablet*. Tablets are stateful logical components implementing various aspects of YDB.

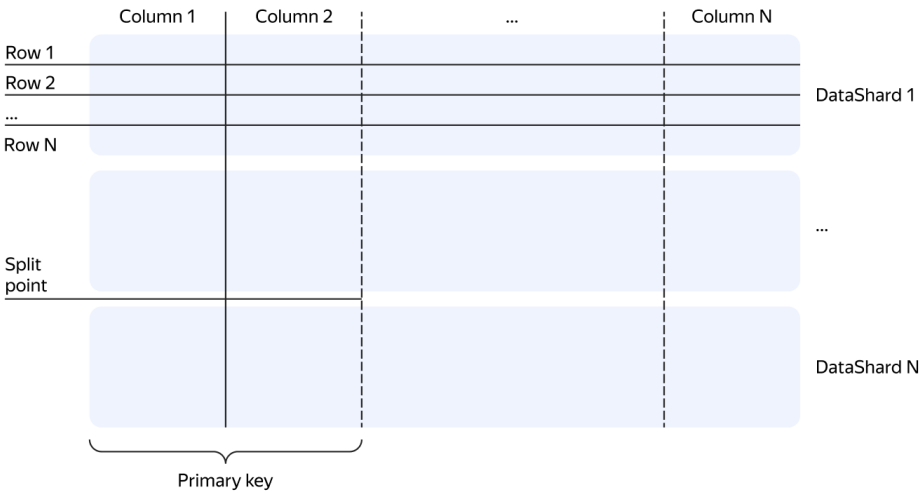
The next level of detail of the overall YDB architecture is explained in the General YDB schema article.

Hierarchy



From the user's perspective, everything inside YDB is organized in a hierarchical structure using directories. It can have arbitrary depth depending on how you choose to organize your data and projects. Even though YDB does not have a fixed hierarchy depth like in other SQL implementations, it will still feel familiar as this is exactly how any virtual filesystem looks.

Table



YDB provides users with a well-known abstraction — tables. In YDB, there are two main types of tables:

- Row-oriented tables are designed for OLTP workloads.
- Column-oriented tables are designed for OLAP workloads.

Logically, from the user's perspective, both types of tables look the same. The main difference between row-oriented and column-oriented tables lies in how the data is physically stored. In row-oriented tables, the values of all columns in each row are stored together. In contrast, in column-oriented tables, each column is stored separately, meaning that cells from different rows are stored next to each other within the same column.

Regardless of the type, each table must have a primary key. Column-oriented tables can only have `NOT NULL` columns in primary keys. Table data is physically sorted by the primary key.

Partitioning works differently in row-oriented and column-oriented tables:

- Row-oriented tables are automatically partitioned by primary key ranges, depending on the data volume.
- Column-oriented tables are partitioned by the hash of the partitioning columns.

Each partition of a table is processed by a specific tablet, called a data shard for row-oriented tables and a column shard for column-oriented tables.

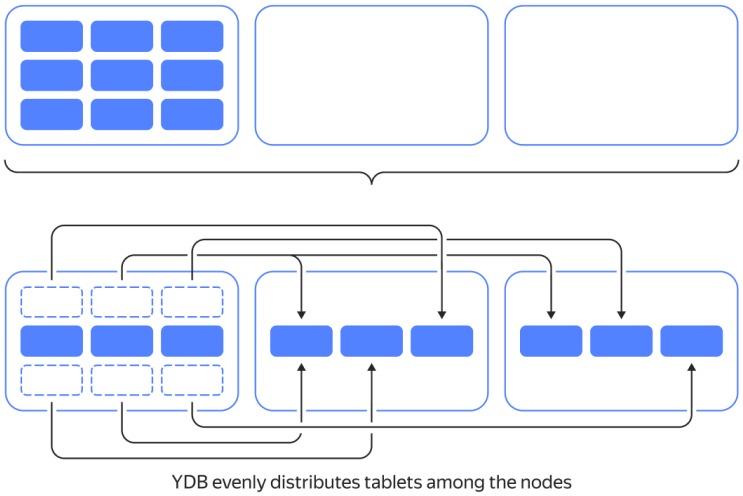
Split by Load

Data shards will automatically split into more as the load increases. They automatically merge back to the appropriate number when the peak load subsides.

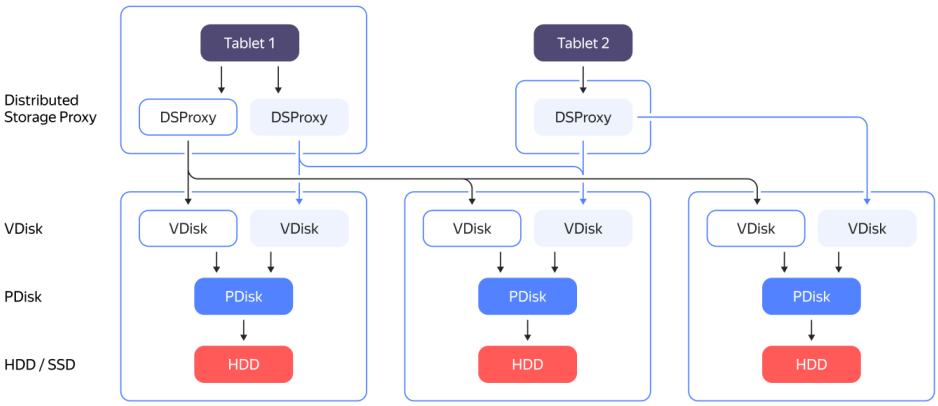
Split by Size

Data shards will also automatically split when the data size increases. They automatically merge back if enough data is deleted.

Automatic Balancing

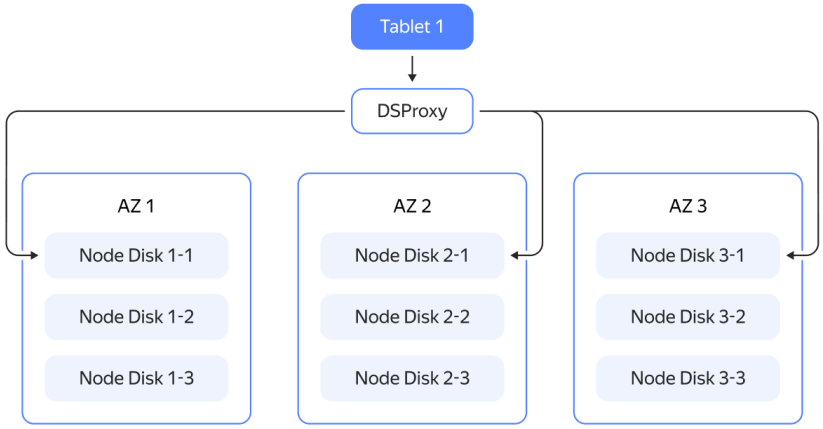


YDB evenly distributes tablets among available nodes. It moves heavily loaded tablets from overloaded nodes. CPU, memory, and network metrics are tracked to facilitate this.



YDB doesn't rely on any third-party filesystem. It stores data by directly working with disk drives as block devices. All major disk kinds are supported: NVMe, SSD, or HDD. The PDisk component is responsible for working with a specific block device. The abstraction layer above PDisk is called VDisk. There is a special component called DSProxy between a tablet and VDisk. DSProxy analyzes disk availability and characteristics and chooses which disks will handle a request and which won't.

Distributed Storage Proxy (DSProxy)



YDB writes data to 3 Availability Zones

A common fault-tolerant setup of YDB spans three datacenters or availability zones (AZ). When YDB writes data to three AZs, it doesn't send requests to obviously bad disks and continues to operate without interruption even if one AZ and a disk in another AZ are lost.

What's Next?

If you are interested in more specifics about various aspects of YDB, check out neighboring articles in this documentation section. If you are ready to jump into more practical content, you can continue to the quick start or YQL tutorials.

Overview of metadata distribution services

Metadata distribution services are three interconnected subsystems of the YDB cluster that deliver service information between nodes: **StateStorage**, **Board**, and **SchemeBoard**. All of them are built on a distributed quorum service with deterministic replica placement.

This article provides an overview of the purpose of the services without diving into internal identifiers and core mechanisms. A detailed description for core developers is in the Metadata distribution subsystems section. Configuration instructions are in the Configuring metadata distribution subsystems section.

Why metadata distribution services are needed

The YDB cluster is a distributed system in which millions of tablets can run simultaneously on thousands of nodes. Cluster components need to know:

- where the leader of a specific tablet runs and how to reach it (**StateStorage**).
- which nodes provide services, such as connection points for clients (**Board**).
- what the current database schema is (**SchemeBoard**).

Distributing this data from a single cluster node is bad — it creates high load and problems when that node fails. Therefore, metadata is distributed across many cluster nodes using three specialized subsystems.

StateStorage

StateStorage stores the current state of tablets: who is the leader now, the generation and step of leader election, and the list of replicas. By tablet id, through this service you can find out the current actor id, which allows interacting with it through the actor system.

> **Note**
>
> Data in StateStorage is **volatile**: it is stored in the memory of replicas and is restored on restart. This is not a long-term storage.

Board

Board is a service for publishing and subscribing to metadata in the format «path → set of records». The main use case is storing database endpoints: nodes publish addresses, clients and other components subscribe to changes.

SchemeBoard

SchemeBoard distributes schema metadata: tables, indexes, topics, access rights. Database nodes use it as a schema cache to avoid contacting SchemeShard on every query.

Comparison of services

Characteristic	StateStorage	Board	SchemeBoard
Purpose	Tablet state and leadership	Publishing service metadata	Schema distribution
Data type	Tablet leader state	Path → payload pairs	Schema object descriptions
Key	Tablet identifier	Path (string)	Path to schema object
Main consumers	Components interacting with tablets	gRPC proxies, clients	Database nodes

General principle

All three services use one architectural approach: a record is addressed by key, the set of replicas for a key is computed deterministically, and operations are performed by quorum. This ensures scalability and fault tolerance during rolling restarts and rack failures.

Details (replica rings, quorum, configuration changes, placement across failure domains) are described in the article Metadata distribution subsystems.

Related materials

- Metadata distribution subsystems — a detailed description for core contributors.
- Configuring metadata distribution subsystems.
- Metadata Distribution SelfHeal.
- Bridge mode.
- Cluster topology.
- Glossary.

Connecting to a Database

To connect to a YDB database from the YDB CLI or an app running the YDB SDK, specify your endpoint and database path.

Endpoint

An endpoint is a string structured as `protocol://host:port` and provided by a YDB cluster owner for proper routing of client queries to its databases by way of a network infrastructure as well as for proper network connections. Cloud databases display the endpoint in the management console on the requisite DB page and also normally send it via the cloud provider's CLI. In corporate environments, YDB endpoint names are provided by the administration team or obtained in the internal cloud management console.

Examples:

- `grpc://localhost:7135` is an unencrypted data interchange protocol (gRPC) with the server running on port 7135 of the same host as the client.
- `grpcs://ydb.example.com` is an encrypted data interchange protocol (gRPCs) with the server running on the ydb.example.com host on an isolated corporate network and listening for connections on YDB default port 2135.
- `grpcs://ydb.serverless.yandexcloud.net:2135` is an encrypted data interchange protocol (gRPCs), public Yandex.Cloud Serverless YDB server at ydb.serverless.yandexcloud.net, port 2135.

Database Path

Database path (`database`) is a string that defines where the queried database is located in the YDB cluster. It has the format and uses the `/` character as separator. It always starts with a `/`.

A YDB cluster may have multiple databases deployed, with their paths determined by the cluster configuration. Like the endpoint, `database` for cloud databases is displayed in the management console on the desired database page, and can also be obtained via the CLI of the cloud provider.

For cloud solutions, databases are created and hosted on the YDB cluster in self-service mode without the involvement of the cluster owner or administrators.

> **Warning**
>
> Applications should not in any way interpret the number and value of `database` directories, since they are set in the YDB cluster configuration. When using YDB in Yandex.Cloud, `database` has the format `region_name/cloud_id/database_id`; however, this format may change going forward for new DBs.

Examples:

- `/ru-central1/b1g8skpblkos03malf3s/etn01q5ko6sh271beftr` is a Yandex.Cloud database with `etn01q3ko8sh271beftr` as ID deployed in the `b1g8skpbljhs03malf3s` cloud in the `ru-central1` region.
- `/local` is the default database for custom deployment using Docker.

Connection String

A connection string is a URL-formatted string that specifies the endpoint and path to a database using the following syntax:

```
<endpoint>?database=<database>
```

Examples:

- `grpc://localhost:7135?database=/local`
- `grpcs://ydb.serverless.yandexcloud.net:2135?database=/ru-central1/b1g8skpblkos03malf3s/etn01q5ko6sh271beftr`

Using a connection string is an alternative to specifying the endpoint and database path separately and can be used in tools that support this method.

A Root Certificate for TLS

When using an encrypted protocol (gRPC over TLS, or gRPCS), a network connection can only be continued if the client is sure that it receives a response from the genuine server that it is trying to connect to, rather than someone in-between intercepting its request on the network. This is assured by verifications through a chain of trust, for which you need to install a root certificate on your client.

The OS that the client runs on already includes a set of root certificates from the world's major certification authorities. However, the YDB cluster owner can use its own CA that is not associated with any of the global ones, which is often the case in corporate environments, and is almost always used for self-deployment of clusters with connection encryption support. In this case, the cluster owner must somehow transfer its root certificate for use on the client side. This certificate may be installed in the operating system's certificate store where the client runs (manually by a user or by a corporate OS administration team) or built into the client itself (as is the case for Yandex.Cloud in YDB CLI and SDK).

YDB Schema Objects

YDB supports the following types of schema objects:

- Directories
- Tables
- Views
- Topics
- Coordination nodes
- Secrets
- External tables
- External data sources

Schema objects are placed within the cluster namespace and are named according to the common naming rules that apply to all schema objects.

See Also

- Concepts YDB
- YDB Cluster Topology

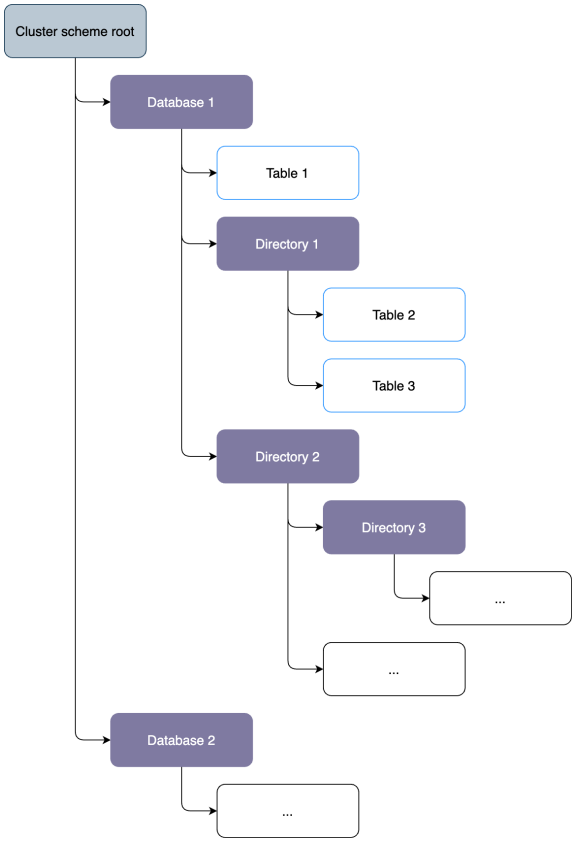
Cluster Namespace and Object Naming

In YDB, all schema objects are organized into a hierarchical namespace called the **cluster schema**. This structure determines how databases, directories, and other objects are arranged and named, providing a clear and scalable way to manage data.

Cluster Schema

The cluster schema forms a tree-like hierarchy where:

- The **root** represents the top level of the namespace.
- At the first level are databases.
- Inside each database, you can create nested directories to build a custom hierarchy.
- Schema objects—such as tables, views, and indexes—can reside either at the database root or within any directory.



Naming Rules for Schema Objects

Every schema object in YDB has a name. In YQL expressions, object names are used as identifiers, which can be unquoted or enclosed in backticks (`` ` ``). For details on identifier syntax, see Keywords and identifiers.

Valid names for schema objects must meet the following criteria:

- **Allowed characters**:
  - Latin letters (`A-Z` , `a-z`)
  - Digits (`0-9`)
  - Special characters: `.` , `-` , and `_`
- **Maximum length**: 255 characters
- **Placement restrictions**: Objects cannot be created in directories with names that start with a dot (for example, `.sys` , `.metadata` , `.sys_health`)

Directory

For convenience, YDB supports creating directories similar to a filesystem, meaning the entire database consists of a directory tree, and scheme objects, such as tables, are located in the leaves of this tree. A directory can host multiple subdirectories and several scheme objects. The names of scheme objects within a single directory are unique.

Table

A table is a relational table containing a set of related data, composed of rows and columns. Tables represent entities. For instance, a blog article can be represented by a table named `article` with columns: `id`, `date_create`, `title`, `author`, and so on. Rows contain data, and columns define data types. For example, the `id` column cannot be empty (`NOT NULL`) and should contain only unique integer values. A record in YQL might look like this:

```
CREATE TABLE article (
    id Int64 NOT NULL,
    date_create Date,
    author String,
    title String,
    PRIMARY KEY (id)
)
```

YDB supports creating row-based and column-based tables. Their main difference lies in the use case and the data storage format on the hard disk. For row-based tables, data is stored sequentially as rows, while for column-based tables, data is stored as columns. Each table type has its own purpose.

Column Naming Rules

Column names in YDB must meet the following requirements:

- A column name can consist of the following characters:
  - Uppercase Latin letters
  - Lowercase Latin letters
  - Digits
  - Special characters: `-` and `_`.
- The length of a column name must not exceed 255 characters.
- Column names must not start with the system prefix `__ydb_`.

Row-Oriented Tables

Row-based tables are well suited for transactional queries generated by Online Transaction Processing (OLTP) systems, such as backends for weather services or online stores. Row-based tables provide efficient access to a large number of columns at once. Search in row-based tables is very fast due to the use of indexes.

An index is a data structure that increases the speed of data retrieval operations based on one or more columns. It is analogous to an index in a book: instead of scanning every page to find the needed section, you can refer to the index at the end of the book and quickly jump to the relevant page.

When a query is executed based on a column (or columns) for which an index has been created, the DBMS can use that index to quickly find the matching rows, avoiding a full scan of all data. For example, if you have an index on the "author" column and you search for all articles written by the author "Gray", the DBMS uses the index to quickly find all rows with that surname.

You can create a row-based table via the YDB web interface, using the CLI or SDK. Regardless of how you interact with YDB, remember the general rule for creating a row-based table: the table must have at least one key column, and creating a table consisting only of key columns is allowed.

By default, when creating a row-oriented table, all columns are optional and can have `NULL` values. This behavior can be modified by setting the `NOT NULL` conditions for key columns that are part of the primary key. Primary keys are unique, and row-oriented tables are always sorted by this key. This means that point reads by the key, as well as range queries by key or key prefix, are efficiently executed (essentially using an index). It's permissible to create a table consisting solely of key columns. When choosing a key, it's crucial to be careful, so we recommend reviewing the article: "Choosing a Primary Key for Maximum Performance".

Partitioning Row-Oriented Tables

A row-based table in the database can be partitioned by ranges of primary key values. Each partition of the table is responsible for its own range of primary keys. The key ranges served by different partitions do not overlap. Different partitions of the table can be served by different servers of the distributed database (including those located in different locations), and can also move independently between servers for rebalancing or to maintain partition health in case of server or network equipment failures.

With a small amount of data or low load, a table can consist of a single partition. As the data volume of a partition or the load on a partition grows, YDB automatically splits it into two:

- If the data volume exceeds the partition size threshold, splitting occurs by the median value of the primary key.
- If load increases, the partition first collects a sample of requested keys (read, written, and deleted) and, based on this sample, selects a key for splitting so that the load is distributed evenly between the new partitions. Thus, in the case of load-based splitting, the new partitions may have substantially different sizes.

The partition size threshold for splitting and enabling or disabling automatic splitting can be configured individually for each database table.

Regardless of the AUTO_PARTITIONING_PARTITION_SIZE_MB parameter value, YDB performs partition splitting and merging based on a size of 2000 MB. At the same time, YDB does not restrict user settings and works in parallel with them. For example, if you set the partition size to 100 MB and the limit to 10 partitions, the table stops splitting when it reaches 10 partitions of 100 MB each, and partition sizes begin to grow. When any partition exceeds 2000 MB, it splits, and the table has 11 partitions. If the size of two adjacent partitions drops to 1000 MB, they merge, and the table has 10 partitions again.

In addition to automatic splitting, you can create an empty table with a predefined number of partitions. You can manually set exact key split boundaries for partitions or specify uniform distribution across a predefined number of partitions. In the latter case, boundaries are created based on the first component of the primary key. Uniform distribution can be specified for tables whose first primary key component is an integer with data type `Uint64` or `Uint32`.

Partitioning parameters apply only to the table itself, not to secondary indexes built on its data. Each index is served by its own set of partitions, and decisions to split or merge its partitions are made independently based on default settings. In the future, these settings may become available to users, similar to the settings of the main table.

The typical duration of a split or merge operation is about 500 ms. During this time, the data involved in the operation becomes briefly unavailable for reads and writes. Specialized wrapper methods in the YDB SDK automatically retry when they detect that a partition is in the process of splitting or merging, without surfacing this information to the application level. It is important to note that if the system

is overloaded for any reason (for example, due to insufficient CPU or bandwidth of the allocated disk resources), split and merge operations may take longer.

The data schema defines the following table partitioning parameters:

AUTO_PARTITIONING_BY_SIZE

- Type: `Enum` (`ENABLED`, `DISABLED`).
- Default value: `ENABLED`.

Automatic partitioning by partition size. If a partition size exceeds the value specified by the AUTO_PARTITIONING_PARTITION_SIZE_MB parameter, it is enqueued for splitting. If the total size of two or more adjacent partitions is less than 50% of the AUTO_PARTITIONING_PARTITION_SIZE_MB value, they are enqueued for merging.

AUTO_PARTITIONING_BY_LOAD

- Type: `Enum` (`ENABLED`, `DISABLED`).
- Default value: `DISABLED`.

Load-based automatic partitioning mode. If a partition consumes more than 50% CPU for several tens of seconds, it is queued for splitting (split). If the total load on two or more adjacent partitions is less than 35% of one CPU core for an hour, they are queued for merging (merge).

When deciding to split a partition by load or merge multiple partitions by load, YDB considers the CPU load on both the leader of the partition and all its replicas.

Performing split or merge operations uses the CPU and takes time. Therefore, when dealing with a variable load, we recommend both enabling this mode and setting AUTO_PARTITIONING_MIN_PARTITIONS_COUNT to a value other than 1. This ensures that a decreased load does not cause the number of partitions to drop below the required value, resulting in a need to split them again when the load increases.

When choosing the minimum number of partitions, it makes sense to consider that a single table partition can reside on only one server and use no more than 1 CPU core for data modification operations. Based on this, for a table that may experience high load, you can set the minimum number of partitions to at least the number of nodes (servers), or better, on the order of the number of CPU cores allocated to the database.

AUTO_PARTITIONING_PARTITION_SIZE_MB

- Type: `Uint64`.
- Default value: `2000 MB` (`2 GB`).

The desired partition size threshold in megabytes. Recommended values range from `10 MB` to `2000 MB`. If this threshold is exceeded, a shard may split.
The specified value serves only as a recommendation for splitting. Splitting may not occur even if the configured size is exceeded.
This setting applies when the `AUTO_PARTITIONING_BY_SIZE` mode is enabled.

AUTO_PARTITIONING_MIN_PARTITIONS_COUNT

- Type: `Uint64`.
- Default value: `1`.

Partition merging (merge) is performed only if the actual number of partitions exceeds the value set by this parameter. When using load-based automatic partitioning, it is recommended to set this parameter to a value other than 1 so that periodic load drops do not reduce the number of partitions below the required level.

AUTO_PARTITIONING_MAX_PARTITIONS_COUNT

- Type: `Uint64`.
- Default value: `50`.

Partition splitting (split) is performed only if the number of partitions does not exceed the value set by this parameter. With any enabled automatic partitioning modes, it is recommended to set a meaningful value for this parameter and monitor the actual number of partitions approaching it; otherwise, partitions will eventually stop splitting as data or load grows, leading to a failure.

UNIFORM_PARTITIONS

- Type: `Uint64`.
- Default value: not applicable.

The number of partitions for uniform initial table partitioning. The primary key's first column must have type `Uint64` or `Uint32`. A created table is immediately divided into the specified number of partitions.

When automatic partitioning is enabled, make sure to set the correct value for AUTO_PARTITIONING_MIN_PARTITIONS_COUNT to avoid merging all partitions into one immediately after creating the table.

PARTITION_AT_KEYS

- Type: `Expression`.
- Default value: not applicable.

Boundary values of keys for initial table partitioning. It's a list of boundary values separated by commas and surrounded with brackets. Each boundary value can be either a set of values of key columns (also separated by commas and surrounded with brackets) or a single value if only the values of the first key column are specified. Examples: `(100, 1000)`, `((100, "abc"), (1000, "cde"))`.

When automatic partitioning is enabled, make sure to set the correct value for AUTO_PARTITIONING_MIN_PARTITIONS_COUNT to avoid merging all partitions into one immediately after creating the table.

Reading Data from Replicas

When executing queries in YDB, the actual execution of a query to each partition is performed at a single point that serves the distributed transaction protocol. However, thanks to storing data on shared storage, it is possible to launch one or more replicas of a partition without allocating additional storage space — the data is already stored replicated, and more than one reader can be served (but the writer is still strictly one at any given moment).

Using read replicas provides the following capabilities:

- Serve clients that are critical to minimal latencies that cannot be achieved otherwise in a multi-DC cluster. This is achieved by bringing the query execution point closer to the query submission point, which eliminates the inter-DC transfer latency. As a result, while preserving all the multi-DC cluster's storage reliability guarantees, point read queries can be answered in milliseconds.
- Serve read requests from replicas without affecting modifying requests running on the partition. This can be useful both for isolating different scenarios and for increasing the partition's throughput.
- Continue serving during partition leader moves (both routine during balancing and during failures). This allows surviving cluster processes without affecting read clients.
- Overall, increase the read performance limit of partitions if many read requests hit the same keys.

In the table data schema, you can specify the need to launch read replicas for each table partition. Read replica accesses typically occur without leaving the data center network, which ensures response times in milliseconds:

Parameter name	Description	Type	Allowed values	Can be changed	Can be reset
`READ_REPLICAS_SETTINGS`	`PER_AZ` means using the specified number of replicas in each AZ and `ANY_AZ` in all AZs in total.	String	`"PER_AZ:<count>"`, `"ANY_AZ:<count>"`, where `<count>` is the number of replicas. To remove replicas, set the value to 0. For example: "PER_AZ:0".	Yes	No

The internal state of each replica is restored exactly from the leader's state and is fully consistent.

Besides the data state in storage, followers also receive a stream of updates from the leader. Updates are sent in real time, immediately after the commit to the log. However, they are sent asynchronously, resulting in some delay (usually no more than dozens of milliseconds, but sometimes longer in the event of cluster connectivity issues) in applying updates to followers relative to their commit on the leader. Therefore, reading data from followers is only supported in the transaction mode `StaleReadOnly()`.

If there are multiple replicas, their lag behind the leader may differ, i.e., although each replica of each partition maintains internal consistency, artifacts may be observed between different partitions. Application code must be prepared for this. For the same reason, it is currently impossible to execute cross-shard transactions from replicas.

Deleting Expired Data (TTL)

YDB supports automatic background deletion of expired data. A table data schema may define a column of a suitable type; the value in this column is compared with the current time for all rows in the background. Rows for which the current time becomes greater than the column value plus the specified delay are deleted.

Parameter name	Type	Allowed values	Can be changed	Can be reset
`TTL`	Expression	`Interval("<literal>") ON <column> [AS <unit>]` or `Interval("literal1") action1, ..., Interval("literal1") action1 ON <column> [AS <unit>]`	Yes	Yes

Syntax of TTL value is described in the article Time to Live (TTL). For more information about deleting expired data, see Time to Live (TTL).

Renaming a Table

YDB allows you to rename an existing table, move it to another directory in the same database, and replace one table with another, in which case the data of the replaced table is deleted. When performing operations, only the table metadata changes, such as its path and name. The table data is not moved or overwritten.

Operations are performed in isolation; the external process sees only two states of the table: before and after the operation. This is critical, for example, for table replacement: the data of the replaced table is deleted by the same transaction that renames the replacing table. During the replacement, there might be errors in queries to the replaced table that have retryable statuses.

The speed of the rename operation is determined by the type of data transactions currently running on the table and does not depend on the amount of data in the table.

- Renaming a table in YQL
- Renaming a table via the CLI

Local Bloom skip indexes

Using a Bloom filter allows more efficiently determining the absence of keys in a table during multiple point lookups by primary key, reducing the number of required disk I/O operations at the cost of increased memory consumption.

The recommended way to manage bloom filters on row-oriented tables is via local Bloom skip indexes (`LOCAL USING bloom_filter`), created with ALTER TABLE ... ADD INDEX and removed with ALTER TABLE ... DROP INDEX. For details, see Bloom skip indexes.

Warning

Parameter name	Type	Allowed values	Can be changed	Can be reset
`KEY_BLOOM_FILTER`	Enum	`ENABLED`, `DISABLED`	Yes	No

The `KEY_BLOOM_FILTER` setting is deprecated. It enables a bloom filter over the full primary key and, when set to `DISABLED`, clears all bloom filters on the table.

Column Groups

YDB allows you to group columns in a table to optimize their storage and usage. The column group mechanism improves the performance of partial row read operations by splitting the storage of table columns into multiple groups. The most common scenario

is storing rarely used attributes in a separate column group. This way, you can enable data compression and/or store them on slower storage devices.

Each column group has its own name, unique within the table. Column group composition is set during table creation and can be modified later. Removing column groups from an existing table is not supported.

A column group can contain any number of columns from its table. Each table column belongs to one and only one column group (column groups do not overlap).

Every table has a primary column group named `default` containing all columns not explicitly assigned to another group. Primary key columns always belong to the primary column group and cannot be moved to another group. Also, if there are no other column groups, changing attributes of the primary column group lets you set attributes for the entire table.

For a column group, attributes that affect data storage and caching are set:

- the type of storage devices used (SSD or HDD, with availability depending on the YDB cluster configuration)
- Data compression mode (no compression or LZ4 algorithm compression);
- caching mode.

Column group attributes are set during table creation and can be modified later. Storage attribute changes aren't immediately applied to existing data; instead, they take effect during subsequent background LSM compaction. Caching attribute changes take effect immediately.

Access to data stored in the fields of the main column group is faster and requires fewer resources than access to data of the same table row stored in the fields of additionally created column groups. Primary key lookup is always performed in the main column group. When accessing fields in other column groups, in addition to the primary key lookup, additional lookup operations are required to determine the specific storage position of these fields. Changing the caching mode does not affect the need for additional operations for non-main columns, so with the same caching modes, access to the main column group will also be faster.

Thus, moving some table columns to a separate column group speeds up reading of the most important and frequently used columns (those in the main column group) at the cost of somewhat slower access to the remaining columns. In addition, column groups allow you to manage data storage parameters, such as the type of storage devices, compression mode, and caching parameters.

Caching modes

You can set caching modes for column groups. Caching modes in YDB allow you to control the strategy for caching table data in RAM. This is especially important for optimizing response time when reading small, frequently used tables. Two caching modes are supported:

- `regular` (default);
- `in_memory`.

In the `regular` caching mode, column group data is loaded into the shared cache only on first access. If a column group uses the `in_memory` mode, its data is automatically preloaded into the cache when the system starts.

When the total amount of data in the cache exceeds the configured limit, memory occupied by pages with the `regular` mode is freed first. If that is not enough, pages with the `in_memory` mode are evicted. Thus, the `in_memory` mode provides priority storage in RAM, but the cache size remains limited by the configuration — memory is not used without limit.

The `in_memory` caching mode can be useful in various scenarios. One common scenario is storing small lookup tables that fit entirely in RAM. In this case, using the `in_memory` mode minimizes access latency because the table always remains in memory, even if it is accessed irregularly.

However, this is not the only use case. For example, in some cases it may be advisable to store all data of a particular table in memory, regardless of its purpose, if the application business logic requires constant fast access to the full version of the table.

When using the `in_memory` mode, remember that memory allocated to such tables reduces the amount available for caching regular tables, which may negatively affect their performance (increase latency).

Note that the `in_memory` caching mode affects read operations only. Write operations are written to disk regardless of the mode, so data durability guarantees remain unchanged. This distinguishes YDB caching modes from some other in-memory DBMSs that may keep entire tables in memory at the cost of reduced data durability guarantees.

Custom attributes

User attributes allow you to add arbitrary information to table metadata. This information is not interpreted by the server, but can be interpreted by the database client (a person or, more often, a program).

Attributes are specified as a key-value pair. The key and value of an attribute can only be a string or a type that can be represented as a string (for example, using base64 encoding).

User attribute keys and values have the following restrictions:

- Key length: 1–100 bytes
- Value length: 1–4096 bytes
- Maximum total size of attributes (sum of lengths of all keys and values): 10240 bytes.

To add, change, retrieve current values, or delete attributes, see Custom attributes in tables.

Column-Oriented Tables

Columnar tables YDB store data of each column separately (independently) from other columns. This data storage principle is optimized for Online Analytical Processing (OLAP) workloads, because when executing a query, only the columns that directly participate in the query are read. Another advantage of this approach is a high degree of data compression, since columns often store repeated or similar data. The downside is that performing operations on rows becomes more expensive.

At the moment, the main use case for YDB column-oriented tables is writing data with an increasing primary key (for example, event time), analyzing this data, and deleting outdated data based on TTL. The optimal way to add data to YDB column-oriented tables is batch upload, performed in MB-sized blocks. Data packet insertion is atomic: data will be written either to all partitions or none.

In most cases, working with columnar tables YDB is similar to working with row-based tables, but there are differences:

- Only `NOT NULL` columns can be used as the primary key.
- Data is partitioned not by the primary key, but by the hash of the partitioning columns.
- Columnar tables support a limited set of data types:

- Available in both the primary key and other columns: `Date`, `Datetime`, `Timestamp`, `Int32`, `Int64`, `Uint8`, `Uint16`, `Uint32`, `Uint64`, `Utf8`, `String`;
- Available only in columns not included in the primary key: `Decimal`, `Double`, `Float`, `Int8`, `Int16`, `JsonDocument`, `Json`, `Yson`.

- You can configure compression and encoding individually for each column. For details, see CREATE TABLE request parameters.

Let's recreate the `article` table, this time in column-oriented format, using the following YQL command:

```
CREATE TABLE article_column_table (
    id Int64 NOT NULL,
    author String,
    title String,
    PRIMARY KEY (id)
)
WITH (STORE = COLUMN);
```

Local Bloom skip indexes

In column-oriented and row-oriented tables you can define local Bloom skip indexes on columns with `LOCAL USING bloom_filter` or `LOCAL USING bloom_ngram_filter`, either during table creation or with ALTER TABLE ADD INDEX. See local indexes and Bloom skip indexes.

Local min_max index

In columnar tables, you can define a local min_max index on columns using `LOCAL USING min_max`. The index is defined when creating a table or added via ALTER TABLE ADD INDEX. For more details: local indexes, min_max index.

Partitioning Column-Oriented Tables

Unlike row-based tables YDB, columnar tables partition data not by primary keys, but by dedicated keys: partitioning keys. Partitioning keys are a subset of the table's primary keys.

Unlike data partitioning in row-based tables YDB, data partitioning for columnar tables is performed not by key values, but by hash values of the keys, which allows data to be evenly distributed across all existing partitions. Such partitioning helps avoid hotspots during insertion and speeds up analytical queries that process (read) large amounts of data.

The choice of partitioning keys substantially affects the performance of column-oriented tables. For more information, see "Choosing a primary key for maximum performance of column-oriented tables".

To manage data partitioning, an additional partitioning parameter is used: AUTO_PARTITIONING_MIN_PARTITIONS_COUNT. Other partitioning parameters are ignored for columnar tables.

AUTO_PARTITIONING_MIN_PARTITIONS_COUNT defines the minimum physical number of partitions for storing data.

Type: `Uint64`.

Default value: 64.

Given that other partitioning parameters are ignored, this same value also determines the maximum number of partitions.

Column-oriented table limitations

Currently, not all columnar table functionality is implemented. The following are not supported:

- Reading from replicas.
- Global secondary indexes.
- Vector and full-text indexes.
- Bloom filter for the primary key (`KEY_BLOOM_FILTER`; see Bloom filter for row-oriented tables).
- Change Data Capture.
- User-defined table attributes.

View

A view logically represents a table formed by a given query. The view itself contains no data. The content of a view is generated every time you `SELECT` from it. Thus, any changes in the underlying tables are reflected immediately in the view.

Views are often used to:

- Hide query complexity
- Limit access to underlying data
- Provide a backward-compatible interface to emulate a table that used to exist but whose schema has changed

> **Warning**
>
> The scenario of creating a view to grant other users partial `SELECT` privileges on a table that has sensitive data has not been implemented yet. In YDB, the view's stored query can only be executed on behalf of the user querying the view. A user cannot access data from a view that reads from a table that they don't have privileges to `SELECT` from. See the `security_invoker` option description on the `CREATE VIEW` page for details.

View Invalidation

If you drop a table that a view references, the view will become invalid. Queries against it will fail with an error caused by referencing a table that does not exist. To make the view valid again, you must provide a queryable entity with the same name (by creating or renaming a table or another view). It needs to have a schema compatible with the deleted one. The dependencies of views on tables and other views are not tracked. A `SELECT` from a view is executed like a `SELECT` from a subquery would, without any prior checks of validity. You would know that the view's query became invalid only at the moment of its execution. This approach will change in future releases: YDB will start tracking the view's dependencies, and the default behavior would be to forbid dropping a table if there's a view referencing it.

Performance

Queries are executed in two steps:

1. Compilation
2. Execution of the compiled code

The resulting compiled code contains no evidence that the query was made using views because all the references to views should have been replaced during compilation by the queries that they represent. In practice, there must be no difference in the execution time of the compiled code (step 2) for queries made using views versus queries directly reading from the underlying tables.

However, users might notice a slight increase in the compilation time of the queries made using views compared to the compilation time of the same queries written directly. It happens because a statement reading from a view:

```
SELECT * FROM a_view;
```

is compiled similarly to a statement reading from a subquery:

```
SELECT * FROM (SELECT * FROM underlying_table);
```

but with an additional overhead of loading data from the schema object `a_view`.

Please note that if you execute the same query over and over again, like:

```
-- execute multiple times
SELECT * FROM hot_view;
```

compilation results will be cached on the YDB server side, and you will not notice any difference in the performance of queries using views and direct queries.

View Redefinition Lag

> **Warning**
>
> Execution plans of queries containing views are currently cached. It might lead to the usage of an old query plan for a short while after a given view has been redefined. This is going to be fixed in future releases. See below for a more detailed explanation.

Query Compilation Cache

YDB caches query compilation results on the server side for efficiency. For small queries like `SELECT 1;` compilation can take significantly more time than the execution.

The cache entry is searched by the text of the query and some additional information, such as a user SID.

The cache is automatically updated by YDB to stay on track with the changes made to the objects the query references. However, in the case of views, the cache is not updated in the same transaction in which the object's definition changes. It happens with a slight delay.

Example of the Problem

Let's consider the following situation. Alice repeatedly executes the following query:

```
-- Alice's session
SELECT * FROM some_view_which_is_going_to_be_redefined;
```

while Bob redefines the view's query like this:

```
-- Bob's session
DROP VIEW some_view_which_is_going_to_be_redefined;
CREATE VIEW some_view_which_is_going_to_be_redefined ...;
```

The text of Alice's query does not change, which means that the compilation will happen only once, and the results are going to be taken from the cache since then. Bob changes the definition of the view, and the cache entry for Alice's query should theoretically be evicted from the cache in the same transaction in which the view was redefined. However, this is not the case. Alice's query will be recompiled with a slight delay, which means that for a short period of time, Alice's query will produce results that are inconsistent with the updated definition of the view. This is going to be fixed in future releases.

See Also

- CREATE VIEW
- ALTER VIEW
- DROP VIEW

Topic

A topic in YDB is an entity for storing unstructured messages, designed to deliver them to multiple subscribers. In fact, a topic is a named set of messages.

A writer application writes messages to a topic. Reader applications are independent of each other; they receive, or "read", messages from the topic in the order they were written. The topic implements the publisher-subscriber architectural pattern.

The YDB topic has the following properties:

- At-least-once guarantees when reading messages by subscribers.
- Exactly-once guarantees when publishing messages (message deduplication).
- Guarantees of sequential message processing (FIFO) for messages published with the same source identifier.
- Scaling of message throughput for messages published with different sequence identifiers.

Messages

Data is transmitted as message streams. A message is the minimum indivisible unit of user information. Messages consist of a body and attributes, as well as additional system properties. The content of messages is a set of bytes that is not interpreted by YDB in any way.

Messages can contain user attributes in the "key-value" format. They are returned together with the message body when reading. User attributes allow the reader to decide whether to process the message without unpacking the message body. Message attributes are set when initializing a write session. This means that all messages written within a single write session will have the same attributes when read.

Partitioning

For horizontal scaling, a topic is divided into separate elements, *partitions*, which are units of parallelism. Each partition has limited throughput; the recommended write speed is up to 1 MB/s.

> **Note**
>
> Currently, reducing the number of partitions in a topic is only supported by deleting and recreating the topic with fewer partitions.

There are two types of partitions:

- **Active.** All partitions by default; both writing and reading are supported.
- **Inactive.** Only reading is possible from them. Inactive partitions appear after a partition split when auto-partitioning is enabled. Inactive partitions are automatically deleted when all messages from such a partition have been deleted after the retention period expires.

Offset

All messages within a partition have a unique sequence number called `offset` (offset). The offset increases monotonically as new messages are written.

Auto-partitioning

The number of topic partitions and their throughput are set when creating the topic and determine the total write throughput of the topic. If the maximum required write speed to the topic is unknown at creation time or will change over time, you can use auto-partitioning to dynamically scale the topic. If auto-partitioning up is enabled on the topic, the number of partitions in such a topic automatically increases as the write speed grows (see Auto-partitioning modes for details).

Guarantees

1. The SDK and server provide exactly-once write guarantees in case of partition splits. This means that any message will be written either to the parent partition or to one of the child partitions. A message cannot be written to both the parent and child partitions simultaneously. Moreover, a message cannot be written to the same partition multiple times.
2. The SDK and server ensure read order. Data will be read from parent partitions first, and only then from child partitions.
3. Thus, the exactly-once write and read order guarantees continue to hold for a specific source identifier (producer-id).

Auto-partitioning modes

The following auto-partitioning modes are available for any topic.

Disabled (DISABLED)

Auto-partitioning is disabled. In this case, the number of partitions remains unchanged, and automatic scaling does not occur.

The initial number of partitions is specified when creating the topic. When manually changing the number of partitions in this mode, new partitions are added. All previously existing partitions remain active.

Increase (UP)

The topic has auto-partitioning 'up' enabled, meaning that when the write speed increases, the number of partitions increases. When the write speed decreases, the number of partitions remains unchanged.

Algorithm for increasing the number of partitions: if within a specified time the write speed to a partition exceeds the specified threshold (as a percentage of the maximum write speed to the partition), that partition is split into two. The original partition becomes inactive, and data can only be read from it. When the message retention period for that partition expires and all messages are deleted, the partition itself is also deleted. The two new child partitions become active, and both reading and writing are possible in them.

Pause (PAUSED)

Auto-partitioning on the topic is paused. Automatic increase in the number of partitions does not occur. If necessary, you can re-enable the partition increase mode.

Examples of YQL queries for switching the topic to various auto-partitioning modes can be found here.

Note

The DOWN and UP_AND_DOWN auto-partitioning modes are currently under development and are not supported in the current version.
The UP_AND_DOWN mode is currently implemented as UP.

Limitations

The following limitations apply when using auto-partitioning:

1. If auto-partitioning is enabled on a topic, it cannot be disabled, only paused.
2. If auto-partitioning is enabled on a topic, writing to or reading from such a topic via the Kafka API protocol is not possible.
3. Auto-partitioning cannot be enabled on a topic with the local storage mode.

Sources

A source identifier, `producer_id`, is a way to order a set of messages. The order of written messages is preserved within `producer_id`.

On first use, the source identifier `producer_id` is bound to a topic partition using a round-robin algorithm, and all messages with this `producer_id` end up in the same partition. The binding is removed if no new messages using this source identifier appear within 14 days.

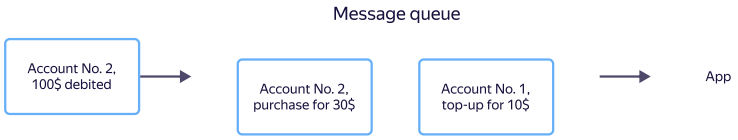
Warning

The recommended maximum number of `producer_id` is up to 100,000 per partition over the last 14 days. For reference, this is approximately 5 new `producer_id` per minute around the clock.

When message processing order matters

Consider a financial application whose task is to calculate the user's account balance and allow or deny debiting funds.

To solve such problems, you can use a message queue. When a deposit, withdrawal, or purchase is made, a message with the account ID, amount, and operation type is written to the queue. The application processes incoming messages and calculates the balance.

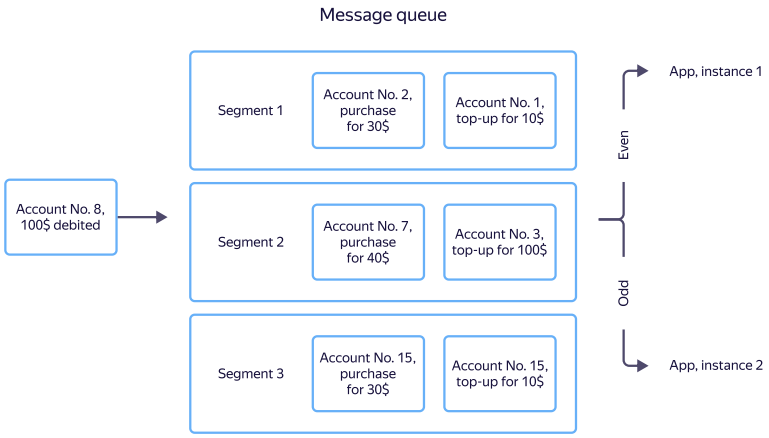


For correct balance calculation, the order of message processing is important. If a user first deposits money and then makes a purchase, the messages with information about these operations must be processed by the application in the same sequence. Otherwise, a business logic error may occur, and, for example, the application will reject the purchase due to insufficient funds. Message queues have mechanisms for guaranteed delivery order, but they cannot ensure the order of messages within a single queue for arbitrary data volumes.

When multiple application instances read messages from a stream, one instance may receive the deposit message and another the withdrawal message. In this case, there is no instance that is guaranteed to have the correct balance information. To solve this problem, you can store data in a DBMS, exchange information between application instances, build a distributed cache, etc.

In YDB, you can write data so that messages from one source arrive at the same application instance. To do this, messages from each source are written with their unique source identifiers (`producer_id`), and a message sequence number from the source (`seqno`) is used to protect against duplicates. In YDB, messages with the same `producer_id` end up in the same partition. When reading from a topic, each reader instance serves its own subset of partitions, thus eliminating the need for synchronization between instances. For example, using this approach, you can ensure that messages about transactions for the same account always end up in the same partition and are processed by the application instance associated with that partition.

Below is an example where all transactions for accounts with even identifiers are sent to the first application instance, and those with odd identifiers to the second.



If processing order is not important

For some tasks, the order of message processing is not critical. For example, sometimes it is important just to deliver the data, and the storage system will handle ordering.

For such cases, you can use a simplified write mode called "write without deduplication". In this mode, you do not need to specify message source identifiers (`producer_id`) and message sequence numbers — `sequence number`. Write without deduplication is faster and consumes fewer server resources, but message ordering and deduplication do not occur on the server. This means that if you send the same message again (for example, after a crash and subsequent restart of the writing process), it may be written more than once.

Message sequence numbers

All messages from one source have a sequence number, `sequence number`, used for deduplication. The message sequence number must increase monotonically within the pair `topic`, `source`. When the server receives a message with a sequence number less than or equal to the maximum recorded for the pair `topic`, `source`, the message will be skipped as a duplicate. Gaps in the sequence of message numbers are allowed. Message sequence numbers must be unique only within the pair `topic`, `source`.

Not used if the write mode without deduplication is selected.

Examples of message sequence numbers

Type	Example	Description
File	Offset of transmitted data from the beginning of the file	You cannot delete rows from the beginning of the file, as this will lead to skipping part of the data, duplicates, or loss of part of the data.
Database table	Auto-increment record identifier	

Message retention time

Each topic has a defined message retention time. After the retention time expires, messages are automatically deleted. An exception is data that has not yet been acknowledged by an "important" reader — it will be stored until the reader processes it. If there is a reader with an explicitly specified availability time, the retention time of unprocessed messages is increased to the specified value.

Data compression

During transmission, the writer application indicates that the message can be compressed using one of the supported codecs. The codec name is passed during writing and stored with the message, and is also returned on reading. Message compression is performed individually for each message; batch compression is not supported. Data compression and decompression operations are performed on the reader and writer application side.

The list of supported codecs is explicitly specified in each topic. Attempting to write data to a topic with an unsupported codec will result in a write error.

Codec	Description
`raw`	No compression.
`gzip`	Compression using the gzip algorithm.

| | Compression using the `lzop` algorithm. |

| | Compression using the `zstd` algorithm. |

Reader

A reader is a named entity for reading data from a topic. The reader contains read positions confirmed by the reader for each topic read on its behalf.

A read session is a client connection to a topic to receive messages on behalf of a reader. Read sessions work through the Topic API. A single reader can establish multiple read sessions: in this case, topic partitions are distributed among these sessions. To work with read sessions, it is recommended to use the YDB SDK (see Working with topics).

In YDB, readers can be of the following types:

- **streaming** — a reader type where a partition is exclusively assigned to one consumer, and all messages from that partition are processed by that consumer.
- **shared (common)** — a reader type that allows multiple consumers to jointly process messages from a single topic without exclusive assignment of partitions to each of them.

Streaming reader

A streaming reader is designed for streaming data processing from a topic. In this mode, multiple instances of the consumer application read messages in parallel and process them as they arrive. Typical scenarios include collecting and analyzing application logs, building real-time aggregates, and replicating events to external systems.

You can read from a topic using a streaming reader via the Topic API and the Kafka protocol.

In streaming mode, each partition of a topic is assigned to at most one consumer at any given time. All messages from the assigned partition are processed by that consumer.

The YDB server is responsible for distributing partitions among consumers. When consumers connect and disconnect, the server redistributes partitions and aims to distribute them as evenly as possible among active consumers.

The number of partitions determines the maximum degree of parallelism: no more consumers than partitions can read from a topic simultaneously. If there are more consumers than partitions, some instances remain without assigned partitions and are idle until free partitions become available or the number of consumers decreases.

Shared (common) reader

A shared (common) reader in YDB is a reading model from a topic where one or more messages from the topic are assigned to the reader, rather than an entire topic partition. This allows a single partition to be processed simultaneously by a large number of

consumers.

A typical use case is message exchange between microservices. Reading from a topic using a shared (common) reader is possible via the Amazon SQS protocol.

A message is locked for a consumer for a certain period of time. If during this time the consumer has not reported completion of message processing and has not extended its processing time, it means that the consumer could not process the message, and the message becomes available for reading and will be read again. The processing time is configured in the reader parameters, but can also be set for each message read request.

A Dead letter policy is applied to messages that cannot be processed. The Dead letter policy can be configured in one of the following ways:

- **none** — the message is returned to the queue after unsuccessful processing and will later be given for re-reading.
- **move** — the message is moved to a DLQ after several processing attempts; any other database topic can be specified as the DLQ.
- **delete** — the message is deleted after several processing attempts.

By default, the Dead letter policy is disabled (none). The number of message processing attempts before moving it to a DLQ (move) or deleting it (delete) is configured on the reader. If the message cannot be moved to the DLQ, it is returned to the queue (as when the Dead letter policy is disabled), and on the next unsuccessful attempt to process the message, another attempt will be made to move the message to the DLQ. For example, if you delete the topic specified as the DLQ, messages after unsuccessful processing will be returned to the queue until the topic specified as the DLQ is created or another existing topic is specified.

Depending on the configuration of the shared (common) reader, reading from a topic can be performed either with or without preserving message order. Message order is preserved among messages with the same message-group-id, which can be set for each message when it is written. If the shared (common) reader is configured to read with message order preservation, then at any given time, such a reader delivers no more than one message from a single message-group-id to consumers for processing.

When placing a message in the queue, you can specify a delay before the message is given for processing for the first time. Message order is determined by the time the message was added to the topic — if reading is performed with message order preservation, a delayed message will pause the delivery for processing of all subsequent messages with the same message-group-id.

See also Shared topic consumer: architecture and limitations.

Reading position

The reading position is the saved offset of the reader for each partition of the topic. The reading position is saved by the reader after sending an acknowledgment of the read data. When establishing a new reading session, messages are delivered to the reader starting from the saved reading position. This allows users not to store the reading position on their side.

Reading limit from a single partition

The `partition_max_in_flight_bytes` parameter can be set when creating a reading session.

This parameter sets the upper limit of in-flight - the amount of data for a single partition that has already been issued by the server in read responses but not yet confirmed by an offset commit.

- `0` — no separate in-flight limit per partition is applied.
- positive value — when the limit is reached, the server temporarily stops delivering new portions for this partition until the client performs a commit and reduces in-flight.

A value that is too small, with slow processing or infrequent commits, can cause reading pauses on a partition. In such a case, increase the limit or perform commits more often.

Important reader

A reader can have the "important" attribute. The presence of this attribute means that messages in the topic will not be deleted until the reader reads and confirms the messages. This attribute can be set for the most critical readers that must process all data even during long downtime.

> **Warning**
>
> Since a prolonged idle period of an important reader can lead to using all available data storage space with unread messages, it is necessary to monitor the read lag of important readers.

Message availability time for the reader

A time availability period can be set for the reader, during which messages from the topic that have not yet been processed will be available to them.

This option allows extending the message retention time in the topic from the retention time up to the specified availability time, if the reader does not acknowledge processing.
Unlike the "important" reader attribute, this parameter limits the maximum age of messages that will be stored in the topic.
If the option is not set or all readers acknowledge processing without significant delay (less than the topic's retention time), then data will be deleted according to the usual rules.

Protocols for working with topics

The YDB SDK is used for working with topics (see Working with topics).

Also, the Kafka API version 3.4.0 (see Kafka API) and Amazon SQS API (see Amazon SQS API) protocols are partially supported.

Work with a single topic can be carried out simultaneously using multiple protocols. For example, writing can be done using the Topic API, and reading using the Amazon SQS API, and vice versa.

Transactions involving topics

YDB supports working with topics within transactions.

Transactional reading from a topic

Data in topics is not modified when reading from a topic. Therefore, when reading from a topic in a transaction, the only transactional operation is changing the offset. When reading transactionally via the SDK, offsets are not committed. Deferred offset commit occurs

automatically when the transaction is committed; the SDK hides this from the user.

Transactional writing to a topic

When writing transactionally to a topic, data is stored outside the partition before commit, and then published (becomes visible) at the moment of transaction commit. In this case, data will be added to the end of the partition at sequential offsets. Visibility of own changes in topics within transactions involving topics is not provided.

Limitations when working with topics in a transaction

Transactions do not impose additional restrictions on working with topics. Within a transaction, you can write large amounts of data to a topic, write to multiple partitions, and read with multiple consumers.

Nevertheless, it is recommended to choose the transaction mode taking into account the specifics of transactional work with topics: data is published at the moment of transaction commit. That is, if the transaction is long-running, data will become visible only after a significant amount of time.

YQL queries to topics

To work with topics, you can use YQL queries with familiar constructs: SELECT for reading messages from a topic and INSERT for writing messages to a topic.

Coordination Node

A coordination node is an object in YDB that allows client applications to coordinate their actions in a distributed manner. Typical use cases for coordination nodes include:

- Distributed semaphores and mutexes.
- Service discovery.
- Leader election.
- Task queues.
- Publishing small amounts of data with the ability to receive change notifications.
- Ephemeral locking of arbitrary entities not known in advance.

Semaphores

Coordination nodes allow you to create and manage semaphores within them. Typical operations with semaphores include:

- Create.
- Acquire.
- Release.
- Describe.
- Subscribe.
- Delete.

A semaphore can have a counter that limits the number of simultaneous acquisitions, as well as a small amount of arbitrary data attached to it.

YDB supports two types of semaphores: persistent and ephemeral. A persistent semaphore must be created before acquisition and will exist either until it is explicitly deleted or until the coordination node in which it was created is deleted. Ephemeral semaphores are automatically created at the moment of their first acquisition and deleted at the last release, which is convenient to use, for example, in distributed locking scenarios.

> **Note**
>
> Semaphores in YDB are **not** recursive. Thus, semaphore acquisition and release are idempotent operations.

Usage

Working with coordination nodes and semaphores is done through dedicated methods in YDB SDK.

Similar Systems

YDB coordination nodes can solve tasks that are traditionally performed using systems such as Apache Zookeeper, etcd, Consul, and others. If a project uses YDB for data storage along with one of these third-party systems for coordination, switching to YDB coordination nodes can reduce the number of systems that need to be operated and maintained.

Secrets

Various access credentials are used for authentication in external systems. These credentials are stored in separate objects called secrets. Secrets are only available for writing and updating; their values cannot be retrieved.
In YDB, secrets are used, for example, in federated queries and data transfers.

Syntax

The following YQL operators are used to manage secrets:

- CREATE SECRET — create a secret.
- ALTER SECRET — modify an existing secret.
- DROP SECRET — delete a secret.

Usage

Examples of using secrets and working with them are provided in the following sections:

- Configuring Time to Live (TTL)
- Working with S3 Buckets (Yandex Object Storage)

Access management

Secrets are schema objects, so rights to them are granted using the GRANT command and revoked using the REVOKE command. To use a secret in a query, for example, when creating an external data source or data transfer, the right `SELECT ROW` is required.

External Tables

Some external data sources, such as database management systems, store data in a structured format, while others, like S3 (Yandex Object Storage), store data as individual files. To work with file-based data sources, you need to understand both the file placement rules and the formats of the stored data.

A special entity, `EXTERNAL TABLE`, describes the stored data in such sources. External tables allow you to define the schema of the stored files and the schema of file placement within the source.

A record in YQL might look like this:

```
CREATE EXTERNAL TABLE s3_test_data (
  key Utf8 NOT NULL,
  value Utf8 NOT NULL
) WITH (
  DATA_SOURCE="bucket",
  LOCATION="folder",
  FORMAT="csv_with_names",
  COMPRESSION="gzip"
);
```

Data can be inserted into external tables just like regular tables. For example, to write data to an external table, you need to execute the following query:

```
INSERT INTO s3_test_data
SELECT * FROM Table
```

More details on working with external tables describing S3 buckets (Object Storage) can be found in section Reading Data from an External Table Pointing to S3 (Object Storage).

External data sources

An external data source is an object in YDB that describes connection parameters to an external data source. For example, in the case of ClickHouse, the external data source describes the network address, login, and password for authentication in the ClickHouse cluster, and in the case of S3 (Object Storage), it describes access credentials and the path to the bucket.

The following example shows how to create an external data source pointing to a ClickHouse cluster:

```
CREATE EXTERNAL DATA SOURCE test_data_source WITH (
  SOURCE_TYPE="ClickHouse",
  LOCATION="192.168.1.1:8123",
  DATABASE_NAME="default",
  AUTH_METHOD="BASIC",
  USE_TLS="TRUE",
  LOGIN="login",
  PASSWORD_SECRET_PATH="test_password_path",
  PROTOCOL="NATIVE"
);
```

After creating an external data source, you can read data from the created `EXTERNAL DATA SOURCE` object. The example below illustrates reading data from the `test_table` table in the `default` database in the ClickHouse cluster:

```
SELECT * FROM test_data_source.test_table;
```

Using external data sources, you can run federated queries for cross-system data analytics tasks.

You can use the following data sources:

- S3

Experimental sources data

- ClickHouse
- Greenplum
- Microsoft SQL Server
- MySQL
- PostgreSQL
- YDB (federated queries between YDB databases)

Backup collection

A backup collection organizes full and incremental backups of selected row tables into manageable chains. It allows you to **restore data to the state of the latest backup in the chain**, providing protection against accidental data loss — erroneous deletions or modifications.

> **Note**
>
> For practical instructions on creating and managing backup collections, see the Backup collections section of the 'Backup and Restore' document.

Overview

Backup collections solve typical backup tasks for production workloads:

- **Storage efficiency**: Incremental backups capture only changes since the previous backup, significantly reducing storage requirements compared to multiple full backups.
- **Consistent restore**: All tables in the collection are created from a single global snapshot, ensuring referential integrity between tables when restoring.
- **Chain restore**: Restore to the state of the **latest** backup in the current chain — see `RESTORE`.

For comparison with other backup methods (export/import, dump/restore), see Backup concepts.

Key concepts

These terms are essential for understanding backup collections. For detailed definitions, see the glossary.

- **Full backup**: A complete snapshot of all data in the collection at a specific point in time. Serves as the basis for subsequent incremental backups.
- **Incremental backup**: Captures only changes (inserts, updates, deletes) since the previous backup. Requires the entire backup chain for restoration.
- **Backup chain**: An ordered sequence starting with a full backup, followed by zero or more incremental backups.

Limitations

Before using backup collections, consider these limitations:

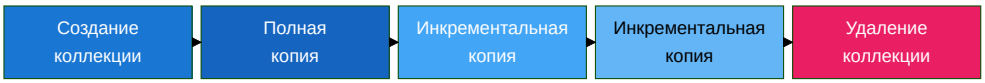
- **Row tables only**: Column tables are not supported.
- **One collection per table**: A table can belong to only one backup collection. To include a table in another collection, run `DROP BACKUP COLLECTION` for the current collection and create a new one with the desired set of tables.
- **Immutable composition**: After creation, the list of tables in a collection cannot be changed. To add new tables, create a new collection with all the required tables.
- **No partial restore**: You cannot restore individual tables from a collection; the entire collection is restored.
- **External scheduling**: YDB does not provide built-in backup scheduling. Use external tools, such as cron, for automation.

Architecture

Backup collections use a copy-on-write mechanism combined with change streams for efficient incremental backups. This section explains how the components work together.

How backup collections work

The following diagram illustrates the backup process:



Creating a collection defines which tables to include and creates a schema object. This is a fast, metadata-only operation.

Full backup creates a consistent snapshot of all tables in the collection. Key characteristics:

- Uses a **global snapshot**, which ensures consistency across all tables in the collection.
- Creates **change streams** for each table to track subsequent modifications.
- Uses **copy-on-write**: The backup is created quickly by referencing existing data; actual data copying occurs only when the source data is modified.

Incremental backup captures all changes since the previous backup:

- Uses a **distributed transaction** to read change streams from all tables at a consistent point, ensuring referential integrity of the entire collection.
- Reads accumulated changes from change streams created during the full backup and accumulating changes **since the previous backup (full or incremental)**.
- Records all modifications: inserts, updates, and deletes (including as deletion records).
- Saves change stream data to incremental backup tables.

> **Warning**
>
> Schema changes (ALTER TABLE) of tables in the backup collection are not tracked by incremental backups. If you need to change the schema of a table with a backup, create a new full backup after the schema change so that the backup chain reflects the new structure.

> **Note**
>
> Change streams created during a full backup are automatically deleted when the backup collection is deleted. They cannot be manually deleted or used for other purposes while the collection exists.

Storage

Backup collections are stored in the YDB cluster in a dedicated directory structure:

```
/Root/database/.backups/collections/
├── my_collection/
│   ├── 20250821141425Z_full/
│   │   ├── table_1/
│   │   └── table_2/
│   └── 20250821151519Z_incremental/
│       ├── table_1/
│       └── table_2/
```

Note

The `.backups` directory is created automatically when the first backup collection is created and is managed by the system. After it is created, you can manage backup tables inside it (for example, when exporting or importing backups).

Cluster storage

By default, backups are stored in the cluster. Backups in the cluster are intended for recovery from **logical errors**, such as accidental `DROP TABLE`, `TRUNCATE TABLE`, or erroneous data changes. Advantages:

- Fast backup and restore operations.
- Integrated security mechanisms.
- No external infrastructure required.

Warning

Backups in the cluster are in the same failure zone as the protected data. If the cluster fails beyond its fault tolerance (for example, complete cluster loss or catastrophic events in the data center), both data and backups may be lost. To protect against such scenarios, use external storage.

External storage

For **disaster recovery** and protection against failures of the entire cluster, regularly export backup collections to external storage (S3-compatible storage or file system) using export/import operations.

To export backups to external storage, use the YDB CLI:

- `ydb export s3` for S3-compatible storage.
- `ydb export nfs` for NFS on cluster hosts.
- `ydb tools dump` for file system on a local computer.

Each backup in the chain must be exported separately. Preserve the chain order during export/import for successful recovery.

Background operations

All backup and restore operations are performed asynchronously, allowing you to continue normal database operations. Track progress using `ydb operation list incbackup`.

Restore from backups

Restore returns data to the state of the latest backup in the chain that is in the cluster. To restore to an earlier state, import only the required chain prefix from external storage — see Import and restore.

Restore process

1. **Import from external storage** (if necessary): If backups were exported, import the full backup and all incremental backups up to the desired restore point.
2. **Perform restore**: Run `RESTORE collection_name` to restore all tables from the backup collection. The system applies the full backup and all incremental backups sequentially to reach the latest backup point.

Warning

The restore operation will fail if at least one of the tables being restored already exists at the same path. Rename or delete conflicting tables before restoring.

All tables in the collection are restored to the same point in time: data from different tables will be mutually consistent.

Note

During the restore operation, the target tables are unavailable for modifications. Partially restored data may be visible to read workloads. Plan the restore for a maintenance window or disable application access to the affected tables until the operation completes.

See also

- Backup concepts: Overview of all backup approaches in YDB
- Backup and restore: backup collections: step-by-step scenarios and CLI commands
- Recipes and examples: Typical scenarios and examples
- YQL reference:
  - CREATE BACKUP COLLECTION

- BACKUP
- RESTORE
- DROP BACKUP COLLECTION

YDB Cluster Topology

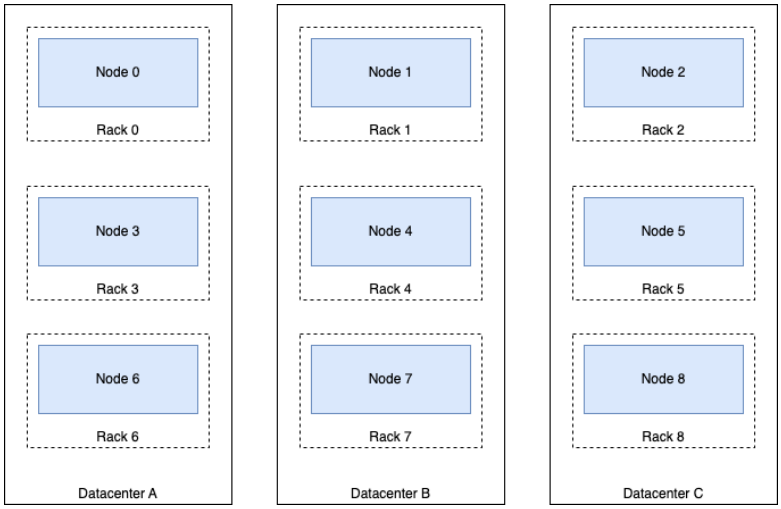
A YDB cluster consists of storage and database nodes. As the data stored in YDB is available only via queries and API calls, both types of nodes are essential for database availability. However, distributed storage consisting of storage nodes has the most impact on the cluster's fault tolerance and ability to persist data reliably. During the initial cluster deployment, an appropriate distributed storage operating mode needs to be chosen according to the expected workload and database availability requirements. The operation mode cannot be changed after the initial cluster setup, making it one of the key decisions to consider when planning a new YDB deployment.

Cluster Operating Modes

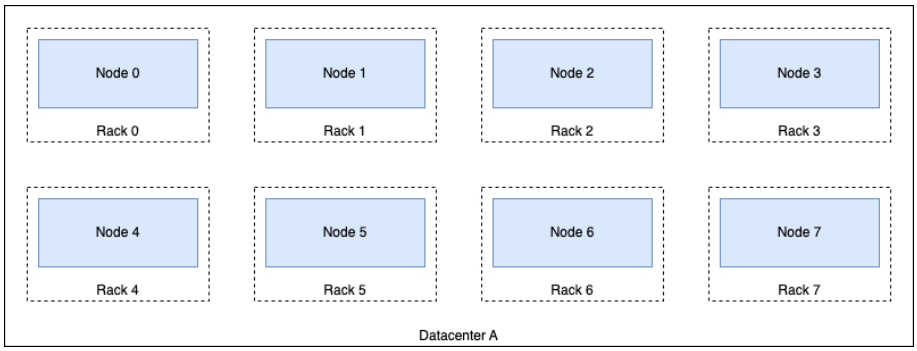
Cluster topology is based on the chosen distributed storage operating mode, which needs to be determined according to the fault tolerance requirements. YDB's failure model is based on the concepts of fail domain and fail realm.

The following YDB distributed storage operating modes are available:

- `mirror-3-dc`. Data is replicated to 3 failure realms (usually availability zones or data centers) using 3 failure domains (usually racks) within each realm. YDB cluster remains available even if one failure realm completely fails; additionally, one failure domain in the remaining zones can fail at the same time. This mode is recommended for multi-datacenter clusters with high availability requirements.



- `block-4-2`. Erasure coding is applied with two blocks of redundancy added to the four blocks of source data. Storage nodes are placed in at least 8 failure domains (usually racks). YDB cluster remains available if any two domains fail, continuing to record all 6 data parts in the remaining domains. This mode is recommended for clusters deployed within a single availability zone or data center.



- `none`. There is no redundancy. Any hardware failure causes data to become unavailable or permanently lost. This mode is only recommended for development and functional testing.

> **Note**
>
> Server failure refers to both total and partial unavailability. For example, the failure of a single disk is also considered a server failure in this context.

Fault-tolerant operation modes of distributed storage require a significant amount of hardware to provide the maximum level of high availability guarantees supported by YDB. However, for some use cases, the investment into hardware might be too high upfront. Therefore, YDB offers variations of these operation modes that require less hardware while still providing a reasonable level of fault tolerance. The requirements and guarantees of all these operation modes and their variants are shown in the table below, while the implications of choosing a particular mode are discussed further in the article.

Mode	Storage volume multiplier	Minimum number of nodes	Fail domain	Fail realm	Number of data centers	Number of server racks
`mirror-3-dc`, can stand a failure of a data center and 1 rack in one of the remaining data centers	3	9 (12 recommended[1])	Rack	Data center	3	3 in each data center
`mirror-3-dc` *(reduced)*, can stand a failure of a data center and 1 server in one of the two other data centers	3	12	½ a rack	Data center	3	6

`mirror-3-dc` *(3 nodes)*, can stand a failure of a single server, or a failure of a data center	3	3	Server	Data center	3	Doesn't matter
`block-4-2`, can stand a failure of 2 racks	1.5	8 (10 recommended[1])	Rack	Data center	1	8
`block-4-2` *(reduced)*, can stand a failure of 1 rack	1.5	10	½ a rack	Data center	1	5
`none`, no fault tolerance	1	1	Node	Node	1	1

> **Note**
>
> The storage volume multiplier specified above only applies to the fault tolerance factor. Other influencing factors (for example, slot fragmentation and granularity) must be taken into account for storage capacity planning.

For information about how to set the YDB cluster topology, see Blob Storage Configuration.

Bridge mode

Bridge mode is a special cluster operating mode that differs significantly from the distributed storage modes listed above. In bridge mode, cluster nodes are divided into several pile (typically corresponding to data centers), each of which stores data using one of the distributed storage modes described above, with synchronous replication organized between pile.

It is important to understand that pile are not independent YDB clusters but parts of a single cluster with complex topology.

Bridge mode provides explicit control over replication stop and resume. The YDB cluster becomes unavailable when any pile fails until the replication stop command is executed for that pile. After it is executed, the cluster resumes operation. Thus, the cluster remains available until the last pile fails.

Resuming replication in a pile after it has been disconnected may take significant time, since YDB synchronizes storage in that pile with the others, replicating missing data. The cluster remains available during synchronization.

Bridge mode is recommended for clusters deployed in two data centers, as well as for systems with high fault tolerance requirements — for example, when it is necessary to maintain availability when three out of four data centers fail.

When using bridge mode, each pile must have enough nodes, domains, and fail realms for the chosen storage mode to work correctly. The resulting storage volume multiplier will equal the sum of storage volume multipliers of all pile.

Cluster response time in bridge mode for most operations is limited by the response time of the slowest pile.

Reduced Configurations

If it is impossible to use the recommended amount of hardware, you can divide servers within a single rack into two dummy fail domains. In this configuration, the failure of one rack results in the failure of two domains instead of just one. In such reduced configurations, YDB will continue to operate if two domains fail. The minimum number of racks in a cluster is five for `block-4-2` mode and two per data center (e.g., six in total) for `mirror-3-dc` mode.

The minimal fault-tolerant configuration of a YDB cluster uses the 3 nodes variant of `mirror-3-dc` operating mode, which requires only three servers with three disks each. In this configuration, each server acts as both a fail domain and a fail realm, and the cluster can withstand the failure of only a single server. Each server must be located in an independent data center to provide reasonable fault tolerance.

YDB clusters configured with one of these approaches can be used for production environments if they don't require stronger fault tolerance guarantees.

Capacity and Performance Considerations

The system can function with fail domains of any size. However, if there are few domains with varying numbers of disks, the number of storage groups that can be created will be limited. In such cases, hardware in overly large fail domains may be underutilized. If all hardware is fully utilized, significant differences in domain sizes may prevent reconfiguration.

For example, consider a cluster in `block-4-2` mode with 15 racks. The first rack contains 20 servers, while the other 14 racks each contain 10 servers. To fully utilize the 20 servers from the first rack, YDB will create groups that include 1 disk from this largest fail domain in each group. Consequently, if any other fail domain's hardware fails, the load cannot be redistributed to the hardware in the first rack.

YDB can group disk drives of different vendors, capacities, and speeds. The resulting characteristics of a group depend on the set of the worst characteristics of the hardware serving the group. Generally, the best results can be achieved by using homogeneous hardware.

> **Note**
>
> Hardware from the same batch is more likely to have similar defects and may fail simultaneously. It is essential to consider this when building large-scale YDB clusters.

Therefore, the optimal initial hardware configurations for production YDB clusters are as follows:

- **A cluster hosted in one availability zone**: This setup uses the `block-4-2` mode and consists of nine or more racks, each with an identical number of servers.
- **A cluster hosted in three availability zones**: This setup uses the `mirror-3-dc` mode and is distributed across three data centers, with four or more racks in each, all containing an identical number of servers.

Database Availability

A database within a YDB cluster is available if both its storage and compute resources are operational:

- All storage groups allocated for the database must be operational, i.e., stay within the allowed level of failures.

- The compute resources of the currently available database nodes (primarily the amount of main memory) must be sufficient to start all the tablets managing objects like tables or topics within the database and to handle user sessions.

To survive an entire data center outage at the database level, assuming a cluster configured with the `mirror-3-dc` operating mode:

- The storage nodes need to have at least double the I/O bandwidth and disk capacity compared to what is required for normal operation. In the worst case, the load on the remaining nodes during the maximum allowed outage might triple, but that's only temporary until self-heal restores failed disks in operating data centers.
- The database nodes must be evenly distributed between all 3 data centers and include sufficient resources to handle the entire workload when running in just 2 of the 3 data centers. To achieve this, database nodes in each datacenter need at least 35% extra spare CPU and RAM resources when running normally without ongoing failures. If database nodes are typically utilized above this threshold, consider adding more of them or moving them to servers with more resources.

See Also

- Documentation for DevOps Engineers
- Blob Storage Configuration
- Example Cluster Configuration Files
- YDB distributed storage

[1] Using fewer nodes will limit the cluster's ability to automatically recover.

Bridge cluster operation mode

> **Feature of Yandex Enterprise Database**
>
> This functionality is available only in the Yandex Enterprise Database. In the open-source version of YDB it is absent.

This article describes a special operating mode of the YDB cluster in which data is stored with synchronous replication between multiple parts of the cluster called pile.

The bridge mode is an extension on top of the basic distributed storage modes: each pile is built using one of the YDB operating modes (for example, `mirror-3-dc` or `block-4-2`) and is fault-tolerant on its own; bridge adds synchronous replication between piles on top of that, plus managed failover/switchover of load between them. This provides an extra layer of fault tolerance on top of the chosen topologies. Bridge mode requires more hardware and is more complex to operate; it is especially useful for clusters deployed across two data centers and for systems with higher availability requirements — for example, when you need to remain available if three out of four data centers fail.

Description of the mode

In bridge mode, cluster nodes are split into several piles (typically one pile per datacenter). If one of the piles fails, the cluster becomes unavailable until the command to disable that pile is executed. After it is disabled, the cluster resumes operation.

YDB supports an arbitrary number of piles; however, for simplicity, the following discussion considers the case of two piles.

At the distributed storage level, bridge mode implements synchronous Active/Active replication between piles. At the database node level, Active/Passive redundancy is possible.

Data is stored in pairs of storage groups from different piles. Within each pile, any YDB operating mode supported for data storage may be used.

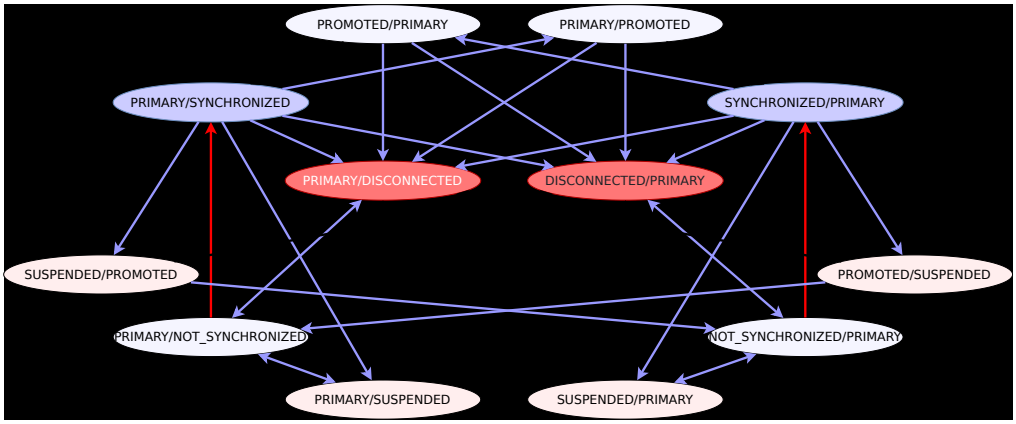
Bridge mode can achieve zero-downtime switchover. On failover, the cluster requires explicit disabling of replication to the failed pile before it can resume. When any pile is disabled, request processing is suspended until the command to disable replication to that pile is executed.

Piles are not standalone YDB clusters; they are parts of a single cluster with a complex topology:

- Tablets in bridge mode run as a single instance;
- each pile runs its own static group and set of independent storage groups with regular VDisks. Storage groups are accessed via DS-proxy-proxy, which exposes the DS-proxy interface to tablets and performs operations using two DS-proxy instances — one for the groups in each pile;
- each pile runs its own set of replicas of StateStorage, SchemeBoard, and Board. Access to StateStorage, SchemeBoard, and Board is through similar proxy-proxy components.

Pile states

Cluster operation is determined by the states of all its piles. For example, for a cluster of pile A and pile B, the pile states are written as (A, B), where order matters.



Each pile can be in one of the following states:

- `PRIMARY`. The main pile where tablets run. Only one pile can be in the `PRIMARY` state.
- `SYNCHRONIZED`. A follower pile whose storage groups are fully synchronized with `PRIMARY`.
- `DISCONNECTED`. Disconnected from `PRIMARY`; does not participate in operations.
- `NOT_SYNCHRONIZED`. A pile whose storage groups are not yet synchronized with `PRIMARY`. When synchronization completes, YDB automatically transitions the pile to `SYNCHRONIZED`.
- `PROMOTED`. A pile being smoothly transitioned from `SYNCHRONIZED` to `PRIMARY`. The pile remains in this state until the transition completes, after which YDB automatically transitions it to `PRIMARY`.
- `SUSPENDED`. A pile being smoothly transitioned to `DISCONNECTED`. When the transition completes, YDB automatically moves the pile to `DISCONNECTED`.

Normally, a pair of piles operates in `PRIMARY/SYNCHRONIZED` or `SYNCHRONIZED/PRIMARY`. That is, one pile is `PRIMARY` (tablets run there) and the other is `SYNCHRONIZED` (all its storage groups are synchronized with `PRIMARY`). In this configuration, a write is considered committed only after it has been successfully written to the storage groups of each pile with the required redundancy.

Transitions between states

From state	To state	How	Description
`PRIMARY`	`SYNCHRONIZED`	Automatically	Planned switchover complete.

PRIMARY	DISCONNECTED	Manually — failover	Emergency disable of unavailable PRIMARY and selection of new PRIMARY.
PRIMARY	SUSPENDED	Manually — takedown	Planned disable of current PRIMARY and selection of new PRIMARY.
SYNCHRONIZED	PRIMARY	Manually — failover	Emergency disable of unavailable PRIMARY and selection of new PRIMARY.
SYNCHRONIZED	DISCONNECTED	Manually — failover	Emergency disable of unavailable SYNCHRONIZED.
SYNCHRONIZED	PROMOTED	Manually — switchover	Start planned change of PRIMARY.
SYNCHRONIZED	SUSPENDED	Manually — takedown	Start planned disable of SYNCHRONIZED pile.
DISCONNECTED	NOT_SYNCHRONIZED	Manually — rejoin	Return pile to cluster after maintenance or recovery.
NOT_SYNCHRONIZED	SYNCHRONIZED	Automatically	Data synchronization complete.
NOT_SYNCHRONIZED	DISCONNECTED	Manually — failover	Emergency disable of unavailable NOT_SYNCHRONIZED.
PROMOTED	PRIMARY	Automatically	Planned switchover complete.
PROMOTED	PRIMARY	Manually — failover	Emergency disable of unavailable PRIMARY and selection of new PRIMARY.
PROMOTED	DISCONNECTED	Manually — failover	Emergency disable of unavailable PROMOTED.
SUSPENDED	NOT_SYNCHRONIZED	Manually — rejoin	Cancel planned disable of pile.
SUSPENDED	DISCONNECTED	Automatically	Planned disable complete.
SUSPENDED	DISCONNECTED	Manually — failover	Emergency disable of unavailable SUSPENDED.

State change scenarios

Pile failure

When a pile fails, the cluster becomes unavailable. To resume cluster operation, the failed pile must be disabled by moving it to the DISCONNECTED state.

The administrator can issue a transition from PRIMARY/SYNCHRONIZED or SYNCHRONIZED/PRIMARY to either PRIMARY/DISCONNECTED or DISCONNECTED/PRIMARY. The command updates the storage group configuration and changes how data is written to those groups. Interconnect tears down sessions with all nodes of the pile in DISCONNECTED; data exchange sessions are not established (TCP connections for exchanging pile state metadata may remain active). Subsequent read and write operations run without the pile in DISCONNECTED. If the PRIMARY pile has changed, tablets are restarted on it.

The same kind of transition applies when the failed pile was in any state and the pile being designated PRIMARY is in one of the allowed states: PRIMARY, SYNCHRONIZED, or PROMOTED.

If the pile being designated PRIMARY was in DISCONNECTED, NOT_SYNCHRONIZED, or SUSPENDED, a normal transition is not possible, because that pile may not hold a complete, up-to-date replica of the data.

Pile recovery

After the nodes of the failed pile are back up, the pile must be reattached to the cluster by moving it to NOT_SYNCHRONIZED.

The administrator can issue a transition from PRIMARY/DISCONNECTED to PRIMARY/NOT_SYNCHRONIZED or from DISCONNECTED/PRIMARY to NOT_SYNCHRONIZED/PRIMARY. Configuration is exchanged between nodes; interconnect sessions are established only between nodes with the same configuration. The command starts storage group synchronization. When synchronization finishes, an automatic transition occurs from PRIMARY/NOT_SYNCHRONIZED to PRIMARY/SYNCHRONIZED or from NOT_SYNCHRONIZED/PRIMARY to SYNCHRONIZED/PRIMARY.

Planned PRIMARY pile switchover

To change the PRIMARY pile in a planned way, the pile that will become the new PRIMARY must be moved to the PROMOTED state.

The administrator can issue a transition from PRIMARY/SYNCHRONIZED to PRIMARY/PROMOTED or from SYNCHRONIZED/PRIMARY to PROMOTED/PRIMARY. The command does not change how data is written to storage groups but updates their configuration and initiates the change of PRIMARY pile with a smooth migration of tablets to the new PRIMARY. When the transition completes, the former PRIMARY becomes SYNCHRONIZED and PROMOTED becomes PRIMARY.

Planned pile disable

To disable a pile in a planned way, it must be moved to the SUSPENDED state.

The administrator can issue a transition from PRIMARY/SYNCHRONIZED to PRIMARY/SUSPENDED or from SYNCHRONIZED/PRIMARY to SUSPENDED/PRIMARY. A planned disable of the nodes of the pile moved to SUSPENDED is performed, and they are moved to DISCONNECTED. The system aims to minimize the impact of this process on cluster operation. After the pile has been moved to DISCONNECTED, its nodes can be shut down for maintenance.

Then perform normal pile recovery by moving it to NOT_SYNCHRONIZED.

Recovery from cluster split with incompatible configuration (split brain)

A situation can occur where the administrator has put the cluster in a state where pile A and pile B are isolated from each other, and then pile A was reconfigured so that it became PRIMARY and pile B became SYNCHRONIZED, while at the same time pile B was reconfigured so that it became PRIMARY and pile A became SYNCHRONIZED. This results in a cluster split with incompatible configuration (split-brain). In this state, each part of the cluster may remain operational.

To restore a single cluster, choose which pile will be wiped. Then stop all nodes of that pile, wait for them to stop completely, wipe all disks on all nodes, and bring the nodes of the pile being wiped back up.
Then perform normal pile recovery by moving it to `NOT_SYNCHRONIZED`.

Non-standard pile recovery

In complex situations (for example, after sequential pile failures), the following scenario can occur: first pile A failed, was moved to `DISCONNECTED`, and the cluster continued operating; then pile B failed irreversibly; after that, pile A was recovered. In such cases, the cluster can be recovered based on pile A.

If the cluster must be brought back up when only a pile in `DISCONNECTED`, `NOT_SYNCHRONIZED`, or `SUSPENDED` is available, the administrator can run a transition from `PRIMARY/DISCONNECTED` to `DISCONNECTED/PRIMARY` (or similarly for `NOT_SYNCHRONIZED` or `SUSPENDED`) with the special `force` parameter. This causes a cluster split with incompatible configuration. Depending on the actual state of data in the pile being moved to `PRIMARY`, the cluster may be restored to a correct or internally inconsistent state. Correct operation of such a cluster is not guaranteed.

If the cluster turns out to be operational, you can proceed to restore it to normal state. Before recovering the disabled pile, fully wipe data and metadata on all its nodes, then move the pile to `NOT_SYNCHRONIZED`.

Bridge mode implementation details

For more on how bridge mode works, see Bridge mode.

Query Execution

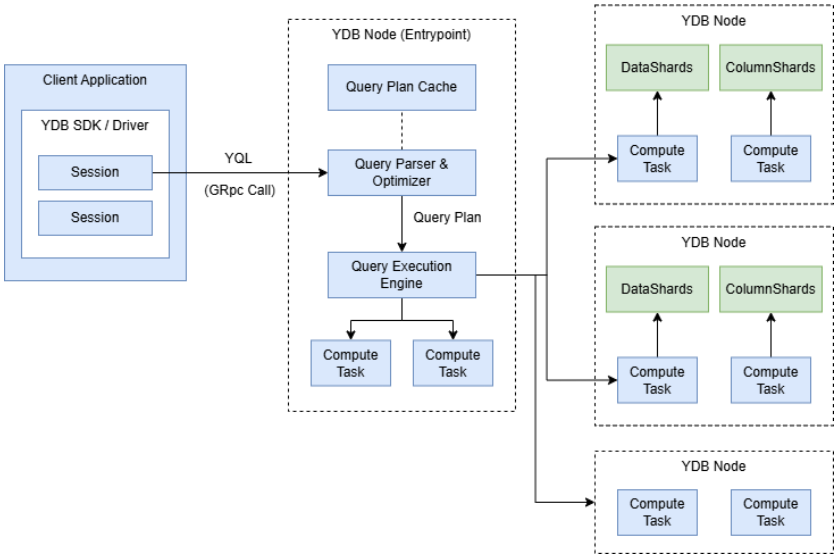
This article provides an overview of query execution in YDB. It is intended to familiarize users with the capabilities and limitations of YDB's query execution engine, including key features such as the supported query language and execution flow. The article also introduces essential terminology and concepts related to query processing, which are used throughout the rest of the documentation.

YDB provides a unified query interface capable of efficiently handling diverse workloads — from high-throughput Online Transaction Processing (OLTP) to large-scale analytical Online Analytical Processing (OLAP) queries. With this approach, applications can run transactional and analytical queries transparently, without having to use different APIs for different workloads.

YDB uses a distributed query execution engine designed for high scalability and efficiency in large, distributed environments. When you run a query, YDB automatically breaks the work down across multiple nodes, taking advantage of data locality — processing data where it is stored whenever possible. This reduces unnecessary data movement across the network. Additionally, YDB leverages advanced features like compute pushdown, where filters and computations are pushed closer to the data storage layer, further improving performance. These techniques enable YDB to efficiently handle complex queries and large workloads across clusters of machines.

General Workflow

This section provides a step-by-step overview of how SQL queries are handled in YDB. Understanding this process helps you become familiar with YDB components and gain insight into what happens under the hood.



1. **Connecting to the Database**
   Your application uses one of the official YDB SDKs to connect to the database. The SDK automatically manages a pool of sessions, logical connections required to execute queries. Behind the scenes, each session is physically connected to one of the nodes in the YDB cluster. When you need to run a query, the SDK provides a session from this pool, so you don't need to manage connections manually.
2. **Starting a Transaction and Sending a Query**
   With your session in hand, your application can begin a new transaction. You then issue your query in the YQL query language based on your application logic and send it to the YDB cluster using the session.
3. **Parsing and Plan Cache Lookup**
   On the server side, the YDB node that receives your query first parses and analyzes it for correctness. Before planning execution, YDB checks whether a physical execution plan for this query already exists in the query cache. If a cached plan is found, it can be reused to save time and resources.
4. **Query Optimization and Plan Preparation**
   If no existing plan is found in the cache, YDB's query optimizer creates a new physical query plan. This plan determines the most efficient way to execute your query across the distributed cluster. For more detailed information about query optimization and query plans, see the Query Optimization in YDB article.
5. **Distributed Query Execution**
   Using the prepared physical plan, YDB starts distributed execution of the query. Work is distributed across multiple nodes in the database, with each node undertaking a part of the computation or data access as defined by the plan. This parallel processing enables fast and scalable query execution, even on large datasets.
6. **Streaming Results Back to the Client**
   For queries that produce results (such as `SELECT` statements), they are returned to your application as one or more result sets, which look like strongly typed tables. Instead of sending all results at once, YDB streams the data back in portions (parts). This allows your application to start processing results immediately and handle large result sets efficiently without needing to load everything into memory at once.
7. **Continuing or Completing the Transaction**
   After receiving and processing the results, your application can choose to continue the transaction by sending additional queries within the same transaction context, or complete the transaction by committing it to save changes.

Further details and explanations of the concepts introduced in this section are provided in the following sections.

Sessions

A session in YDB is a logical "connection" to the database that maintains the context required to execute queries and manage transactions. Sessions store transaction state and other essential context, enabling a series of related queries to be executed as part of a single transaction. Most query execution operations occur within the context of an active session.

Sessions are designed to be long-living objects. One of their key roles is to enable efficient load balancing: by distributing sessions and their associated queries across different nodes in the cluster, YDB can make better use of resources and achieve high availability and scalability.

In practice, you don't need to worry about creating, reusing, or closing sessions yourself. All official YDB SDKs provide session pooling out of the box. A session pool automatically manages the lifecycle of sessions—creating them when needed, reusing existing ones, and returning them to the pool—so that you can focus on writing your application's logic rather than handling session management details.

Transactions

Every query in YDB is executed within the context of a transaction, ensuring data consistency and reliability. Transactions can be managed either explicitly, or by specifying appropriate transaction control parameters during query execution.

YDB also supports Interactive transactions, which give you the flexibility to execute multiple queries within the same transaction, while allowing your application to perform custom logic between those queries. This makes it possible to build complex workflows that require several related operations to be treated as a single atomic unit.

For comprehensive information on transactions and the available transaction modes in YDB, see the Transactions article.

Retries

YDB employs Optimistic concurrency control for transaction management. This means that a transaction may be aborted during execution if YDB detects a conflict and cannot guarantee the requested isolation level — for example, when two transactions attempt to modify the same data concurrently. Additionally, because YDB operates as a distributed system across potentially large clusters, some nodes may become temporarily unavailable due to network partitions, hardware failures, or maintenance. Such events can also cause transaction failures that require retries.

Retries should always be handled at the transaction level, not at the level of individual queries. In Interactive transactions, the sequence of queries and their intermediate results may influence subsequent operations, making it unsafe or impossible to retry only a single failed query. Therefore, if a query fails due to a conflict or a transient error, the entire transaction should be retried from the beginning to ensure correctness and consistency.

All official YDB SDKs provide built-in retry logic and transaction management helpers to simplify application development. By using the standard transaction methods provided by your SDK, you automatically get correct and robust retry behavior without having to implement it manually. For details about retry mechanisms in specific SDKs, see the Handling errors.

Query language

Queries for YDB are written in YQL — an SQL dialect designed with scalable distributed databases in mind. While YQL is not fully ANSI SQL compatible, it closely follows familiar SQL syntax and concepts for most common use cases, making it easy to learn for those with SQL experience. The complete language reference is available in the YQL documentation.

Most interactions with YDB are performed using YQL, making it the primary tool for querying and managing data in YDB. Because of this, understanding YQL's features and capabilities is essential for effectively working with YDB. Learning YQL enables you to take full advantage of the database's advanced query functionality, express complex business logic, and utilize YDB's distributed architecture efficiently.

YQL supports most common SQL constructs, including:

- Data Manipulation Language (DML) — `SELECT`, `INSERT`, `REPLACE`, `UPDATE`, `DELETE`, `UPSERT`.
- Data Definition Language (DDL) — `CREATE`, `ALTER`, `DROP` for tables, indexes, and other schema objects.
- Joins — all standard `JOIN` types, plus special joins such as `LEFT SEMI`, `RIGHT SEMI`, and `ANY` joins.
- Aggregations — `GROUP BY` and window functions.
- Named expressions for better query text organization.
- A collection of built-in functions for processing various data types, empowering users to handle complex logic directly in queries.
- Pragmas and hints to fine-tune execution plans.

Result Sets

When you execute a query in YDB, the result can consist of one or more result sets. Each result set is similar to a table: it contains rows and columns, where every column has a defined, explicit data type. This strong typing guarantees that the structure of the returned data is always predictable and consistent.

Result sets in YDB can be arbitrarily large. To efficiently handle large amounts of data, YDB streams result sets back to the client in parts (chunks). This streaming approach lets clients begin processing the results right away without waiting for the entire result set to be transferred. As a result, applications can handle large datasets quickly and with minimal memory usage.

Limitations

When working with queries in YDB, there are several important limitations to keep in mind:

- **No schema transactions**
  YDB does not support schema transactions. This means that DDL statements (such as creating or altering tables) cannot be combined with DML statements (such as inserts, updates, or deletes) within the same transaction or query.
- **Large updates and optimistic locking**
  YDB uses optimistic concurrency control. When performing very large updates or deletes within a transaction, the probability of lock conflicts increases, making such operations impractical. For bulk modifications, it is recommended to use `BATCH UPDATE` / `BATCH DELETE` statements.
- **Transaction size limits**
  The amount of data that can be written in a single transaction is limited. For detailed thresholds, see the Limits on Query Execution section.

For a full overview of YDB limitations, see Database Limits.

Query Execution

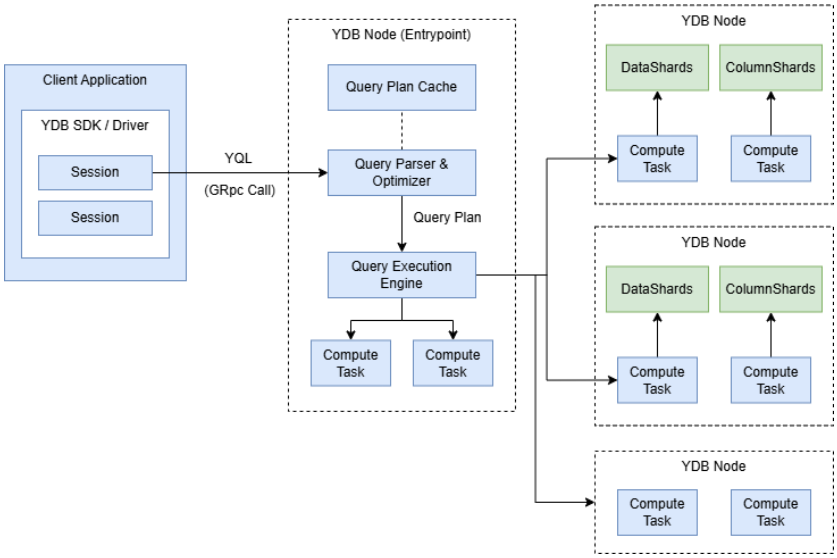
This article provides an overview of query execution in YDB. It is intended to familiarize users with the capabilities and limitations of YDB's query execution engine, including key features such as the supported query language and execution flow. The article also introduces essential terminology and concepts related to query processing, which are used throughout the rest of the documentation.

YDB provides a unified query interface capable of efficiently handling diverse workloads — from high-throughput Online Transaction Processing (OLTP) to large-scale analytical Online Analytical Processing (OLAP) queries. With this approach, applications can run transactional and analytical queries transparently, without having to use different APIs for different workloads.

YDB uses a distributed query execution engine designed for high scalability and efficiency in large, distributed environments. When you run a query, YDB automatically breaks the work down across multiple nodes, taking advantage of data locality — processing data where it is stored whenever possible. This reduces unnecessary data movement across the network. Additionally, YDB leverages advanced features like compute pushdown, where filters and computations are pushed closer to the data storage layer, further improving performance. These techniques enable YDB to efficiently handle complex queries and large workloads across clusters of machines.

General Workflow

This section provides a step-by-step overview of how SQL queries are handled in YDB. Understanding this process helps you become familiar with YDB components and gain insight into what happens under the hood.



1. **Connecting to the Database**
   Your application uses one of the official YDB SDKs to connect to the database. The SDK automatically manages a pool of sessions, logical connections required to execute queries. Behind the scenes, each session is physically connected to one of the nodes in the YDB cluster. When you need to run a query, the SDK provides a session from this pool, so you don't need to manage connections manually.
2. **Starting a Transaction and Sending a Query**
   With your session in hand, your application can begin a new transaction. You then issue your query in the YQL query language based on your application logic and send it to the YDB cluster using the session.
3. **Parsing and Plan Cache Lookup**
   On the server side, the YDB node that receives your query first parses and analyzes it for correctness. Before planning execution, YDB checks whether a physical execution plan for this query already exists in the query cache. If a cached plan is found, it can be reused to save time and resources.
4. **Query Optimization and Plan Preparation**
   If no existing plan is found in the cache, YDB's query optimizer creates a new physical query plan. This plan determines the most efficient way to execute your query across the distributed cluster. For more detailed information about query optimization and query plans, see the Query Optimization in YDB article.
5. **Distributed Query Execution**
   Using the prepared physical plan, YDB starts distributed execution of the query. Work is distributed across multiple nodes in the database, with each node undertaking a part of the computation or data access as defined by the plan. This parallel processing enables fast and scalable query execution, even on large datasets.
6. **Streaming Results Back to the Client**
   For queries that produce results (such as `SELECT` statements), they are returned to your application as one or more result sets, which look like strongly typed tables. Instead of sending all results at once, YDB streams the data back in portions (parts). This allows your application to start processing results immediately and handle large result sets efficiently without needing to load everything into memory at once.
7. **Continuing or Completing the Transaction**
   After receiving and processing the results, your application can choose to continue the transaction by sending additional queries within the same transaction context, or complete the transaction by committing it to save changes.

Further details and explanations of the concepts introduced in this section are provided in the following sections.

Sessions

A session in YDB is a logical "connection" to the database that maintains the context required to execute queries and manage transactions. Sessions store transaction state and other essential context, enabling a series of related queries to be executed as part of a single transaction. Most query execution operations occur within the context of an active session.

Sessions are designed to be long-living objects. One of their key roles is to enable efficient load balancing: by distributing sessions and their associated queries across different nodes in the cluster, YDB can make better use of resources and achieve high availability and scalability.

In practice, you don't need to worry about creating, reusing, or closing sessions yourself. All official YDB SDKs provide session pooling out of the box. A session pool automatically manages the lifecycle of sessions—creating them when needed, reusing existing ones, and returning them to the pool—so that you can focus on writing your application's logic rather than handling session management details.

Transactions

Every query in YDB is executed within the context of a transaction, ensuring data consistency and reliability. Transactions can be managed either explicitly, or by specifying appropriate transaction control parameters during query execution.

YDB also supports Interactive transactions, which give you the flexibility to execute multiple queries within the same transaction, while allowing your application to perform custom logic between those queries. This makes it possible to build complex workflows that require several related operations to be treated as a single atomic unit.

For comprehensive information on transactions and the available transaction modes in YDB, see the Transactions article.

Retries

YDB employs Optimistic concurrency control for transaction management. This means that a transaction may be aborted during execution if YDB detects a conflict and cannot guarantee the requested isolation level — for example, when two transactions attempt to modify the same data concurrently. Additionally, because YDB operates as a distributed system across potentially large clusters, some nodes may become temporarily unavailable due to network partitions, hardware failures, or maintenance. Such events can also cause transaction failures that require retries.

Retries should always be handled at the transaction level, not at the level of individual queries. In Interactive transactions, the sequence of queries and their intermediate results may influence subsequent operations, making it unsafe or impossible to retry only a single failed query. Therefore, if a query fails due to a conflict or a transient error, the entire transaction should be retried from the beginning to ensure correctness and consistency.

All official YDB SDKs provide built-in retry logic and transaction management helpers to simplify application development. By using the standard transaction methods provided by your SDK, you automatically get correct and robust retry behavior without having to implement it manually. For details about retry mechanisms in specific SDKs, see the Handling errors.

Query language

Queries for YDB are written in YQL — an SQL dialect designed with scalable distributed databases in mind. While YQL is not fully ANSI SQL compatible, it closely follows familiar SQL syntax and concepts for most common use cases, making it easy to learn for those with SQL experience. The complete language reference is available in the YQL documentation.

Most interactions with YDB are performed using YQL, making it the primary tool for querying and managing data in YDB. Because of this, understanding YQL's features and capabilities is essential for effectively working with YDB. Learning YQL enables you to take full advantage of the database's advanced query functionality, express complex business logic, and utilize YDB's distributed architecture efficiently.

YQL supports most common SQL constructs, including:

- Data Manipulation Language (DML) — `SELECT`, `INSERT`, `REPLACE`, `UPDATE`, `DELETE`, `UPSERT`.
- Data Definition Language (DDL) — `CREATE`, `ALTER`, `DROP` for tables, indexes, and other schema objects.
- Joins — all standard `JOIN` types, plus special joins such as `LEFT SEMI`, `RIGHT SEMI`, and `ANY` joins.
- Aggregations — `GROUP BY` and window functions.
- Named expressions for better query text organization.
- A collection of built-in functions for processing various data types, empowering users to handle complex logic directly in queries.
- Pragmas and hints to fine-tune execution plans.

Result Sets

When you execute a query in YDB, the result can consist of one or more result sets. Each result set is similar to a table: it contains rows and columns, where every column has a defined, explicit data type. This strong typing guarantees that the structure of the returned data is always predictable and consistent.

Result sets in YDB can be arbitrarily large. To efficiently handle large amounts of data, YDB streams result sets back to the client in parts (chunks). This streaming approach lets clients begin processing the results right away without waiting for the entire result set to be transferred. As a result, applications can handle large datasets quickly and with minimal memory usage.

Limitations

When working with queries in YDB, there are several important limitations to keep in mind:

- **No schema transactions**
  YDB does not support schema transactions. This means that DDL statements (such as creating or altering tables) cannot be combined with DML statements (such as inserts, updates, or deletes) within the same transaction or query.
- **Large updates and optimistic locking**
  YDB uses optimistic concurrency control. When performing very large updates or deletes within a transaction, the probability of lock conflicts increases, making such operations impractical. For bulk modifications, it is recommended to use `BATCH UPDATE` / `BATCH DELETE` statements.
- **Transaction size limits**
  The amount of data that can be written in a single transaction is limited. For detailed thresholds, see the Limits on Query Execution section.

For a full overview of YDB limitations, see Database Limits.

Query Optimization in YDB

YDB uses two types of query optimizers: a rule-based optimizer and a cost-based optimizer. The cost-based optimizer is used for complex queries, typically analytical (OLAP), while rule-based optimization works on all queries.

A query plan is a graph of operations, such as reading data from a source, filtering a data stream by a predicate, or performing more complex operations such as JOIN and GROUP BY. Optimizers in YDB take an initial query plan as input and transform it into a more efficient plan that is equivalent to the initial one in terms of the returned result.

Rule-Based Optimizer

A significant part of the optimizations in YDB applies to almost any query plan, eliminating the need to analyze alternative plans and their costs. The rule-based optimizer consists of a set of heuristic rules that are applied whenever possible. For example, it is beneficial to filter out data as early as possible in the execution plan for any query. Each optimizer rule comprises a condition that triggers the rule and a rewriting logic that is executed when the plan is applied. Rules are applied iteratively as long as any rule conditions match.

Cost-Based Query Optimizer

The cost-based optimizer is used for more complex optimizations, such as choosing an optimal join order and join algorithms. The cost-based optimizer considers a large number of alternative execution plans for each query and selects the best one based on the cost estimate for each option. Currently, this optimizer only works with plans that contain JOIN operations. It chooses the best order for these operations and the most efficient algorithm implementation for each join operation in the plan.

The cost-optimizer consists of three main components:

- Plan enumerator
- Cost estimation function
- Statistics module, which is used to estimate statistics for the cost function

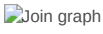
Plan Enumerator

The current Cost-based optimizer in YDB enumerates all useful join trees, for which the join conditions are defined. It first builds a join hypergraph, where the nodes are tables and edges are join conditions. Depending on how the original query is written, the join hypergraph may have quite different topologies, ranging from simple chain-like graphs to complex cliques. The resulting topology of the join graph determines how many possible alternative plans need to be considered by the optimizer.

For example, a star is a common topology in analytical queries, where a main fact table is joined to multiple dimension tables:

```
SELECT
    P.Brand,
    S.Country AS Countries,
    SUM(F.Units_Sold)

FROM Fact_Sales F
INNER JOIN Dim_Date D    ON (F.Date_Id = D.Id)
INNER JOIN Dim_Store S   ON (F.Store_Id = S.Id)
INNER JOIN Dim_Product P ON (F.Product_Id = P.Id)

WHERE D.Year = 1997 AND  P.Product_Category = 'tv'

GROUP BY
    P.Brand,
    S.Country
```

In this query graph, all `Dim...` tables are joined to the `Fact_Sales` fact table:



Common topologies also include chains and cliques. A "chain" is a topology where tables are connected to each other sequentially and each table participates in no more than one join. A "clique" is a fully connected graph where each table is connected to every other table.

In practice, OLAP queries often have a topology that is a combination of "star" and "chain" topologies, while complex topologies like "cliques" are very rare.

The topology significantly impacts the number of alternative plans that the optimizer needs to consider. Therefore, the cost-based optimizer limits the number of joins that are compared by exhaustive search, depending on the topology of the original plan. The capabilities of exact optimization in YDB are listed in the following table:

Topology	Number of supported joins
Chain	110
Star	18
Clique	15

YDB uses a modification of the DPHyp algorithm to search for the best join order. DPHyp is a modern dynamic programming algorithm for query optimization that avoids enumerating unnecessary alternatives and allows you to optimize plans with `JOIN` operators, complex predicates, and even `GROUP BY` and `ORDER BY` operators.

Cost Estimation Function

To compare plans, the optimizer needs to estimate their costs. The cost function estimates the time and resources required to execute an operation in YDB. The primary parameters of the cost function are estimates of the input data size for each operator and the size of its output. These estimates are based on statistics collected from YDB tables, along with an analysis of the plan itself.

Statistics for the Cost-Based Optimizer

The cost-based optimizer relies on table statistics and individual column statistics. YDB collects and maintains these statistics in the background. You can manually force statistics collection using the ANALYZE query.

The current set of table statistics includes:

- Number of records
- Table size in bytes

The current set of column statistics includes:

- Count-min sketch

Cost Optimization Levels

In YDB, you can configure the cost optimization level via the CostBasedOptimizationLevel pragma.

To force the cost-based optimizer to be enabled for the current query, set the pragma `PRAGMA ydb.CostBasedOptimization = "on";`. To disable the cost-based optimizer, set the pragma to `off`.

Multi-Version Concurrency Control (MVCC)

This article describes how YDB uses MVCC.

YDB Transactions

YDB transactions run at serializable isolation level by default, which in layman's terms means it's as if they executed in some serial order without overlapping. While technically any order is allowed, in practice, YDB also guarantees non-stale reads (modifications committed before the transaction started will be visible). With YDB you may run these transactions interactively (users may have client-side logic between queries), which uses optimistic locks for conflict detection. When two transactions overlap in time and have conflicts (e.g., both transactions read the same key, observe its current value, and then try to update it), one of them will commit successfully, but the other will abort and will need to be retried.

YDB is a distributed database that splits OLTP tables into DataShard tablets, partitioned using the table's primary key, and each storing up to ~2GB of user data. Tablets are fault-tolerant replicated state machines over the shared log and shared storage, which may quickly migrate between different compute nodes. DataShards tablets implement low-level APIs for accessing corresponding partition data and support distributed transactions.

Distributed transactions in YDB are based on the ideas of Calvin, distributing deterministic transactions across multiple participants using a predetermined order of transactions. Every participant receives a substream of the global transaction stream (that involves this participant). Since each participant receives deterministic transactions in the same relative order, the database as a whole eventually reaches a consistent deterministic state (where each participant reached the same decision to commit or abort), even when different participants execute their substreams at different speeds. It's important to note that this does not make YDB eventually consistent and transactions always observe a consistent state at their point in time. Deterministic distributed transactions have a limitation in that they need to know all participants ahead of time, but YDB uses them as a building block, executing multiple deterministic transactions as phases of a larger user transaction. In practice, only the final commit phase runs as a distributed transaction since YDB tries to transform other phases into simple single-shard operations as much as possible while preserving serializable isolation guarantees.

Having a predetermined order of transactions becomes a concurrency bottleneck, as when you have a slow transaction that has to wait for something, all transactions down the line have to wait as well. This necessitates a good out-of-order execution engine that is able to reorder transactions that don't conflict while preserving externally visible guarantees. Out-of-order execution alone can't help when transactions actually conflict, however. One example from the past is when a wide read was waiting for data from a disk, it blocked all writes that fell into the same range, stalling the pipeline. Implementing MVCC reads lifted that restriction.

What Is MVCC

MVCC (Multi-Version Concurrency Control) is a way to improve database concurrency by storing multiple row versions as they have been at different points in time. This allows readers to keep reading from a database snapshot without blocking writers. Databases don't overwrite rows but make a modified copy of them instead, tagged with some version information, keeping older row versions intact. Older row versions are garbage collected eventually, e.g., when there are no readers left that could possibly read them.

A simple and naive way of adding MVCC to a sorted KV store is to store multiple values for each key, e.g., using keys tagged with a version suffix and skipping versions that are not visible to the current transaction/snapshot. Many databases go this route one way or another. Bear in mind that such a naive approach leads to data bloat, as multiple copies of data need to be stored and older versions may be difficult to find and remove. It also causes degraded range query performance, as the query execution engine needs to skip over many unnecessary versions until it finds the right one or until it finds the next key.

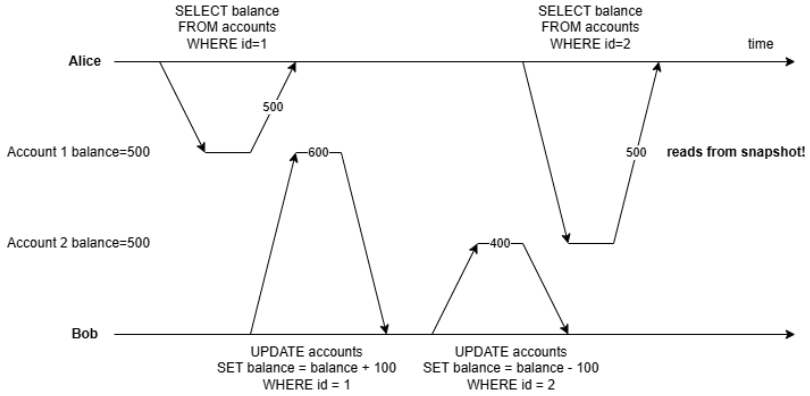
Why YDB Needs MVCC

YDB table shards store data in a sorted KV store, implemented as a write-optimized LSM tree, and historically they did not use MVCC. Since the order of transactions is predetermined externally (using Coordinators, somewhat similar to sequencers in the original Calvin paper), YDB heavily relies on reordering transaction execution at each participant, which is correct as long as such reordering cannot be observed externally, and it doesn't change the final outcome. Without MVCC, reordering is impeded by read-write conflicts, e.g., when a write cannot start execution until a particularly wide read is complete. With MVCC, writes no longer need to wait for conflicting reads to complete, and reads only ever need to wait for preceding conflicting writes to commit. This makes the out-of-order engine's job easier and improves the overall throughput.

Timestamp	Statement	Without MVCC	With MVCC	Description
v1000:123	`UPSERT`	☑️	☑️	Executed
v1010:124	`SELECT`	⏳	⏳	Reading from disk
v1020:126	`UPSERT`	❌	☑️	UPSERT without MVCC has to wait for SELECT to finish
v1030:125	`SELECT`	❌	⏳	SELECT needs to wait for UPSERT to finish before reading from disk

YDB without MVCC also had to take additional steps to preserve data consistency. Interactive transactions may consist of multiple read operations, where each read was performed at a different point in time, and YDB had to perform special checks to ensure conflicting transactions did not perform writes between the read and commit time. This was not ideal for our users, as even pure read-only transactions were often failing with serializability errors and had to be retried.

After implementing MVCC using global versions (shared with deterministic distributed transactions), it became possible to perform reads in a transaction using a global snapshot. This means pure read-only transactions no longer fail with serializability errors and rarely need to be retried by the user, improving throughput in read-heavy use cases. Transactions also acquired a "repeatable read" property, which means if you perform several selects from the same table, the transaction will not fail and will observe the same (globally consistent) result.

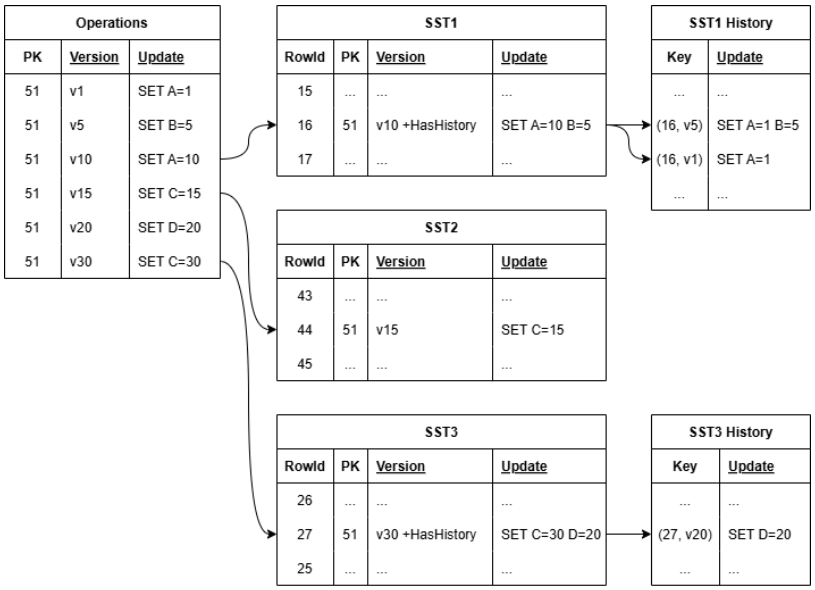


How YDB Stores MVCC Data

A DataShard tablet stores one table shard as a write-optimized LSM tree in which, for each primary key, we keep a list of operations that change column values in the row. The current row state is built on every access by merging individual change operations. Background compaction likewise merges value changes across LSM levels and writes an aggregated state for each row.

When MVCC was added to YDB, one goal was to avoid hurting performance for existing workloads. A typical example is queries for the latest row state in a table over a range of primary-key values. A naive MVCC design that appends a version suffix to the primary key was ruled out by the developers because it inevitably hurts query performance. Instead, YDB always keeps the latest row versions in the main SST structure (sorted string table, part of the LSM tree). When several versions of a row exist, the newest version is marked with a flag meaning "change history" is present. Older versions of rows marked this way live in an additional SST; version data for each row is sorted in descending version order.

When reading from a snapshot, YDB checks whether the latest row version in the main SST is "too new" and, if needed, runs a binary search in the SST that holds historical data. The changes in the row version that matches the current snapshot are used to reconstruct the row's data state. Because LSM levels roughly correspond to when they were created, the search for row versions in the SST for the current level can stop as soon as a suitable version is found. To build the full row state at the snapshot, we take the newest version of that row in each older LSM level. The number of row-version merge operations is therefore bounded by the number of LSM levels, which is usually small.



Rows stored in SSTs do not hold a full set of attributes; in practice they store only data changes. Each row stored in a given SST contains the full set of data changes for the row relative to the latest version of the same row in the previous SST in the LSM tree. This redundant storage format makes lookups and compaction more efficient.

Consider a hypothetical case where a user applies one million updates to a row with key K, each time changing only one of many table columns. To keep writes efficient, YDB avoids reading the row state at update time and instead records a change in the form "update column C for key K". Row versions accumulate in memory in a MemTable and are periodically flushed to new SSTs, each of which in this example will hold a (typically large) number of versions of the updated row. If each row version stored only a delta compared to the previous version, building the current row state would require merging data from one million accumulated versions and scanning a large amount of data. Instead, at each SST level YDB stores aggregated row changes, starting from the MemTable as LSM level 0 (where the previous row state is always available in RAM, so disk reads are not needed).

During compaction, when several SST levels are merged into a new SST, each changed row is merged with its latest versions from older SSTs. That reduces the cost of row merges (still bounded per row by the number of LSM levels) for both compaction and reads, while still allowing "blind" writes without reconstructing the full row when updating its columns.

PK=51 at different versions			
Version	**SST3**	**SST2**	**SST1**
v35	SET C=30 D=20	no updates (C=15 overridden by C=30)	SET A=10 B=5
v25	SET D=20 found (27, v20) <= v25	SET C=15	SET A=10 B=5
v19	no data for v19	SET C=15	SET A=10 B=5
v7	no data for v7	no data for v7	SET A=1 B=5 found (16, v5) <= v7

Eventually, we mark version ranges as deleted and no longer readable, after which compactions allow us to garbage collect unnecessary row versions automatically (unreachable versions are skipped over and not emitted when writing new SSTs). We also

store a small per-version histogram for each SST, so we can detect when too much unnecessary data accumulates in the LSM tree and trigger additional compactions for garbage collection.

How YDB Uses MVCC

MVCC allows DataShards to improve the reordering of transactions, but we can do even better by leveraging global snapshots, so we use global timestamps as version tags, which correspond to global order already used by deterministic distributed transactions. This allows us to create global snapshots by choosing a correct global timestamp. Using such snapshots to read from DataShards effectively allows the database to observe a consistent snapshot at that point in logical time.

When we perform the first read in an interactive or multi-stage transaction, we choose a snapshot timestamp that is guaranteed to include all previously committed transactions. Currently, this is just a timestamp that corresponds to the last timestamp sent out by Coordinators. This timestamp may be slightly in the future (as some transactions in-flight from Coordinators to DataShards may not have started executing yet), but usually not by much, and since transactions are inflight, they are expected to be executed soon. Most importantly, this timestamp is guaranteed to be equal to or larger than any commit timestamp already observed by the client, so when this timestamp is generated for new reads, those reads are guaranteed to include all previously finished writes.

We then use this snapshot for reads without any additional coordination between participants, as snapshots are already guaranteed to be globally consistent across the database. To guarantee snapshots include all relevant changes and are not modified later, DataShards may need to wait until they received all write transactions that must have happened before that snapshot, but only conflicting writes need to execute before the read may start. DataShards also guarantee that any data observed by the snapshot is frozen and won't be modified later by concurrent transactions (effectively guaranteeing repeatable read), but this only applies to observed changes. Anything not yet observed is in a state of flux and is free to be modified until logical time advances well past the snapshot at every participant in the database. An interesting consequence of this is that some later writes may be reordered before the snapshot, which is allowed under serializable snapshot isolation.

When interactive transactions perform writes, their changes are buffered and the final distributed commit transaction checks (using optimistic locks) that the transaction did not have any conflicting changes between its read (snapshot) and commit time, which effectively simulates moving all reads forward to the commit point in time, only committing when it's possible. Interactive transactions also detect when they read from "history" data and mark such transactions as read-only, since we already know moving this read forward in time (to a future commit timestamp) would be impossible. If a user attempts to perform writes in such a transaction, we return a serialization error without any additional buffering or communication. If it turns out a transaction was read-only, there's no serialization violation (we have been reading from a consistent snapshot after all), and we only need to perform some cleanup, returning success to the user. Before YDB introduced global MVCC snapshots, we had to always check locks at commit time, which made it a struggle to perform wide reads under a heavy write load.

Keeping Fast KeyValue Performance

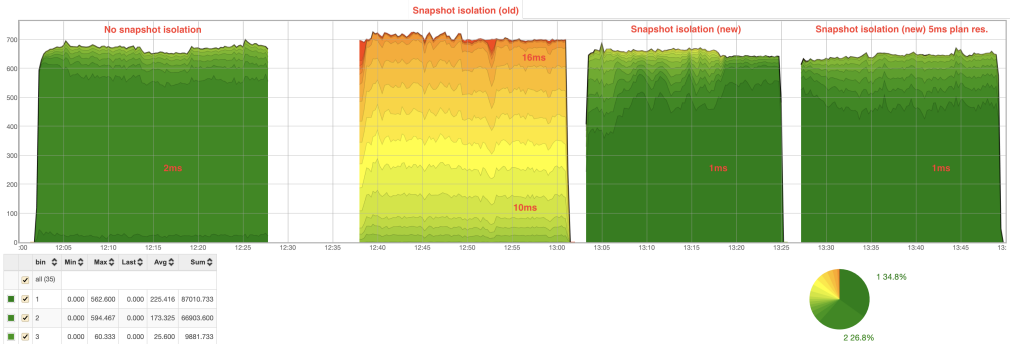
While YDB supports complicated distributed transactions between multiple shards and tables, it's important to also support fast single-shard transactions. Internally such transactions are called "immediate" and they historically don't use any additional coordination. For example, when a transaction only reads or writes a single key in a table, we only want to communicate with a single shard that the key belongs to and nothing else. At first glance, it runs contrary to the use of global MVCC timestamps; however, we may use local uncoordinated timestamps (possibly different at each DataShard), which are still correct global timestamps. Each distributed transaction is assigned a timestamp that consists of two numbers (step, txId), where a step is a coordination tick, often configured to increase every 10ms, and txId is a globally unique transaction id. Immediate transactions don't have a step, since they are not processed by coordinators, and while txId is unique, it is not guaranteed to always increase, so we cannot use it for write timestamps without risking these timestamps going back in time. There's an important distinction, however, since immediate transactions don't have to be repeatable, their read/write timestamp does not have to be unique. That's why we allow them to share a timestamp with some next transaction in the queue (effectively executing "just before" that transaction). When the queue is empty, we use a broadcasted coordinator time (the maximum step acknowledged by shards, so shard is guaranteed there will be no transactions up to that step's end) and a maximum number as txId (effectively executing "just before" that step ends).

Assigning such non-unique timestamps to immediate transactions guarantees there's always some new non-decreasing timestamp that allows them to execute without delay (subject to other out-of-order execution restrictions). It also helps with compactions, as we may have thousands of these transactions executed at the same timestamp, and all of that history is compacted into a single final row version for that step. This allows us to only keep versions in history that need to be consistent across multiple shards.

Prioritizing Reads

An interesting conundrum happened when we tried to enable MVCC snapshots for the first time: there was a severe degradation in read latency under some workloads! This was because time in YDB usually "ticks" every 10ms (tuned to be similar to typical multi-datacenter commit time of 4ms), and single-shard writes use "current time" as part of their timestamp. This meant snapshot reads had to wait for the next tick until the read timestamp is "closed" for new writes, as well as make sure it is actually safe. This added quite a bit of latency to reads, which in hindsight was entirely expected.

We had to go back to the drawing board and find ways to prioritize reads without penalizing writes too much. What we ended up doing was to "close" the read timestamp as soon as we perform a repeatable snapshot read, while choosing some "future" timestamp for new single-shard writes that is guaranteed to not corrupt previously read snapshots (ensuring repeatable read). However, since that write timestamp is in the future, and we must ensure any new snapshot includes committed data, we delay responses until that future timestamp matches the current timestamp, and pretend this new data is not committed (e.g., when running concurrent single-shard reads) until timestamps finally match. Interestingly, since writes are already in flight to storage and must be committed first in at least one other datacenter, this wait did not add much to average write latency even with many snapshot reads, but it did wonders to read latency:



Secondary indexes

In YDB, an index is automatically created on the primary key, so queries with a condition on the primary key are always executed efficiently, affecting only the required rows. A query with a condition on one or more non-key columns typically results in a full table scan. To make such queries efficient, you should use *secondary indexes* — global structures with a separate index table.

Separately, there are local indexes: auxiliary structures that are stored together with the table data and are used when reading on the storage side, without materializing a separate index table (see the Local indexes section below).

In the current version of YDB, *synchronous* and *asynchronous* global secondary indexes are implemented. Each index is a hidden table that is updated:

- for synchronous indexes — transactionally when the main table is modified;
- for asynchronous indexes — in the background, receiving the necessary changes from the main table.

When a user sends an SQL query to insert, modify, or delete data, the database transparently generates commands to modify the index table. A table can have multiple secondary indexes. An index can include multiple columns, and the order of columns in the index is important. A single column can be included in multiple indexes. In addition to the specified columns, the values of the table's primary key columns are always implicitly stored in the index, so that from a found index entry you can navigate to the table entry.

Synchronous secondary index

A synchronous index is updated simultaneously with the table it indexes. Such an index provides strong data consistency and uses the distributed transactions mechanism for this. Thus, while read and blind write operations on a table without an index can be performed without a planning stage, thereby significantly reducing latency, such optimization is not possible for writes to a table with a synchronous index.

Asynchronous secondary index

An asynchronous index, unlike a synchronous one, does not use the distributed transaction mechanism, but receives changes from the indexed table in the background. Write transactions to a table with such an index are performed without additional planning overhead, at the cost of reduced guarantees: an asynchronous index provides eventual data consistency, but not strong consistency. Using an asynchronous index in read transactions is only possible in Stale Read Only mode.

Covering secondary index

It is possible to copy the contents of columns into the index (covering index), thus eliminating the need to read from the main table in index read operations, which significantly reduces latency. At the same time, such denormalization leads to increased disk space consumption and possible slowdown of insert and update operations due to the need for additional data copying.

Unique secondary index

This type of index implements the semantics of a unique value in a column or set of columns, and, like other indexes, allows efficient point reads on the set of indexed columns. YDB uses it to perform additional checks to ensure that each unique value of the indexed columns appears in the table no more than once. If a modifying query violates this constraint, it is aborted with the status `PRECONDITION_FAILED`. Therefore, user code must be prepared to handle this status.

A unique secondary index is a synchronous index, so from a transactional perspective, its update process is the same as that of the synchronous secondary index described above.

Vector index

Vector index is a special type of secondary index.

Unlike traditional secondary indexes, which optimize equality or range search, vector indexes allow performing vector search based on distance or similarity functions.

Full-text index

Full-text index is a special type of secondary index.

Unlike traditional secondary indexes, which optimize search by equality or range, full-text indexes allow scalable text search for words and phrases (and when using N-grams, also for substrings). See also: Full-text search.

JSON-index

JSON index is a special type of secondary index, like full-text index — both are built on top of an inverted index, but use different tokenizers.

JSON indexes allow you to speed up predicates with the JSON_EXISTS and JSON_VALUE functions on the content of a column of type `Json` or `JsonDocument`. The index is built by splitting JSON documents into path tokens and pairs of the form "path + value", which allows you to find matching rows by JsonPath paths without a full table scan. See also: JSON search.

Local indexes

Local indexes — auxiliary structures stored together with table data and used for reads on the storage side. They do not materialize a separate index table. Currently, Bloom indexes are implemented, and other types are planned in the future.

Online creation of a secondary index

In YDB, you can create a secondary index and delete an existing secondary index without stopping service. For one table, only one index can be created at a time.

The online index creation operation consists of the following steps:

1. Taking a snapshot of the table with data, creating an index table marked as available for writes.

   After this step, write transactions become distributed, and data is written to the main table and the index. The index is not yet available to the user.
2. Reading a snapshot of the main table and writing to the index.

   Implements "write to the past": situations are allowed where data updates in step 1 change data written in step 2.
3. Publishing the result, deleting the snapshot.

   The index is ready to use.

Possible impact on user transactions:

- Increased latency may be observed because transactions become distributed (when creating a synchronous index).
- An increased background of `OVERLOADED` errors is possible because automatic partitioning of index table shards is actively working during data writes.

> **Note**
>
> The data write speed is chosen to minimize the impact of the write process on user transactions. To control the speed, configure limits for the corresponding queue of the resource broker.

Creating an index is an asynchronous operation. If a client-server connection break occurs after starting the operation, the index building will continue. You can manage the asynchronous operation via YDB CLI.

Creating and deleting secondary indexes

A secondary index can be:

- Created when creating a table using the YQL CREATE TABLE command.
- Added to an existing table by the YQL ALTER TABLE command or by the YDB CLI table index add command.
- Deleted from an existing table by the YQL command ALTER TABLE or by the YDB CLI command table index drop.
- Deleted together with the table by the YQL command DROP TABLE or by the YDB CLI command `table drop`.

Using secondary indexes

Detailed information on using secondary indexes in applications is available in the article about them in the developer documentation section.

Vector search

Vector search concept

Vector search, also known as nearest neighbor search (NN), is an optimization problem of finding the nearest vector (or set of vectors) in a given dataset relative to a query vector. Proximity between vectors is determined using distance or similarity metrics.

A common approach, especially with large datasets, is approximate nearest neighbor (ANN) search, which yields faster results at the possible cost of some accuracy.

Vector search is widely used in:

- recommendation systems;
- semantic search;
- image similarity search;
- anomaly detection;
- classification systems.

In addition, **vector search** in YDB is widely applied in machine learning (ML) and artificial intelligence (AI) tasks. It is particularly useful in Retrieval-Augmented Generation (RAG) approaches, which utilize vector search to retrieve relevant information from large volumes of data, significantly enhancing the quality of generative models.

Vector search methods can be divided into three main categories:

- exact methods;
- approximate methods without index;
- approximate methods with index.

The choice of method depends on the number of vectors and the workload. Exact methods are slower to search and faster to update; indexes are the opposite.

Exact vector search

The foundation of the exact method is the calculation of the distance from the query vector to all the vectors in the dataset. This algorithm, also known as the naive approach or brute force method, has a runtime of `O(dn)`, where `n` is the number of vectors in the dataset, and `d` is their dimensionality.

Exact vector search is best utilized if the complete enumeration of the vectors occurs within acceptable time limits. This includes cases where they can be pre-filtered based on some condition, such as a user identifier. In such instances, the exact method may perform faster than the current implementation of vector indexes.

Main advantages:

- No need for additional data structures, such as specialized vector indexes.
- Full support for transactions, including in strict consistency mode.
- Instant execution of data modification operations: insertion, update, deletion.

For more information, see exact vector search examples.

Approximate vector search without index

Approximate methods do not perform a complete enumeration of the initial data. This allows significantly speeding up the search process, although it might lead to some reduction in the quality of the results.

Scalar Quantization is a method of reducing vector dimensionality, where a set of coordinates is mapped into a space of smaller dimensions.

YDB supports vector searching for vector types `Float`, `Int8`, `Uint8`, and `Bit`. Consequently, it is possible to apply scalar quantization to transform data from `Float` to any of these types.

Scalar quantization reduces the time required for reading and writing data by decreasing the number of bytes. For example, when quantizing from `Float` to `Bit`, each vector is reduced by 32 times.

Approximate vector search without an index uses a very simple additional data structure - a set of vectors with other quantization. This allows the use of a simple search algorithm: first, a rough preliminary search is performed on the compressed vectors, followed by refining the results on the original vectors.

Main advantages:

- Full support for transactions, including in strict consistency mode.
- Instant application of data modification operations: insertion, update, deletion.

Learn more about approximate vector search without index.

Approximate vector search with index

When the data volume significantly increases, non-index approaches cease to work within acceptable time limits. In such cases, additional data structures are necessary such as vector indexes, which accelerate the search process.

Main advantage:

- ability to work with a large number of vectors.

Disadvantages:

- index build can take considerable time;
- the current version does not support data modification operations: insert, update, delete.

Fulltext search

Concept of fulltext search

Fulltext search is a way to find documents by text content in a `String` or `Utf8` column by words, phrases, and (with special indexing) by substrings. Unlike simple string predicates, fulltext search typically involves **tokenization** and **normalization** (for example, lowercasing), so queries operate on "terms" rather than raw byte substrings. Typical use cases include:

- search in product catalogs and knowledge bases
- log/message search
- searching document metadata and descriptions

The core idea of a fulltext index is an **inverted index**: for each term, the index stores the list of documents (primary keys) that contain that term. This allows queries like "documents containing terms A and B" to avoid full table scans.

In YDB, fulltext search can be performed in two main ways:

- **without an index** — by scanning the table and applying string predicates (for example, `LIKE`). This approach is simple but does not scale well as data grows.
- **with a fulltext index** — by creating a fulltext index and executing queries through `VIEW IndexName`. This approach is designed for scalable search.

Fulltext search with a fulltext index

Fulltext indexes build an inverted index over a text column and allow:

- fulltext matching via FulltextMatch (query modes `Keywords` / `Query` / `Wildcard`, default operator `And` / `Or`)
- relevance ranking (BM25) via FulltextScore when using fulltext_relevance

Search behavior (what counts as a term, case/word-form handling, wildcard support) is configured at index creation time via tokenizers and filters (for example, lowercase, Snowball stemming, n-grams). For details, see the index creation syntax Fulltext index.

Learn more:

- Fulltext indexes
- Fulltext built-in functions
- VIEW (Fulltext index)

Hybrid search

Hybrid search concept

Hybrid search combines fulltext search and vector search into a single ranked result. Instead of choosing one signal, a hybrid query scores each document by both how well its text matches the query (lexical relevance) and how close its embedding is to the query vector (semantic similarity), and then **fuses** the two rankings into one.

The two signals are complementary:

- **Fulltext search** (BM25) is precise for exact terms, names, codes, and rare keywords, but misses documents that use different wording.
- **Vector search** (nearest neighbor over embeddings) captures semantic similarity and paraphrases, but can rank a topically related document above one that contains the exact term the user typed.

Fusing them yields results that are both lexically and semantically relevant. This is a common building block for search over knowledge bases and for the retrieval stage of Retrieval-Augmented Generation (RAG).

Hybrid search in YDB

In YDB, a hybrid query runs over a table that has **both**:

- a fulltext_relevance index over the text column, and
- a vector_kmeans_tree index over the embedding column.

The query expresses the fusion in the `ORDER BY` clause with the `HybridRank` function, which takes one scoring expression per branch — a FullTextScore for the text branch and a Knn distance or similarity for the vector branch:

```
$queryText = "machine learning";
$queryVector = Knn::ToBinaryStringFloat([0.1, 0.2, 0.3, 0.4]);

SELECT id, title
FROM documents
ORDER BY HybridRank(
    FullTextScore(text, $queryText),
    Knn::CosineDistance(embedding, $queryVector))
LIMIT 10;
```

Each branch runs against its own index and returns its own list of best-matching documents. YDB then merges these lists into a single ranking — in one of two ways:

- **Reciprocal Rank Fusion (RRF)** — the default. Each branch contributes a term based on a document's *rank* within that branch, so the absolute magnitudes of the scores (which are not comparable across branches) do not matter.
- **Linear** — a weighted sum of the (optionally min-max normalized) per-branch scores.

Per-branch weights, the RRF constant, and the candidate-pool sizes are all configurable. For the full query syntax and parameters, see Hybrid search (HybridRank).

Learn more:

- Using hybrid search
- Hybrid search query syntax (HybridRank)
- Fulltext search
- Vector search

Searching JSON document contents

JSON search is a way to find table rows by the content of a JSON document stored in a column of type `Json` or `JsonDocument`: by the existence of a path in the document and by the value located at the specified path. The path is specified using a JsonPath expression, and the checks are performed using the existing functions JSON_EXISTS and JSON_VALUE. Typical scenarios:

- Search for documents with a specific nested field
- Search for documents whose field at a given path equals the required value.
- filtering semi-structured data without a pre-defined schema.

In YDB, JSON search can be performed in two main ways:

- Without an index: scanning the table and applying the `JSON_EXISTS` or `JSON_VALUE` functions to each row. The approach is simple but scales poorly: the amount of scanning work grows with the table size.
- with a JSON index — creating a JSON index on a JSON column. This approach is designed for scalable search.

JSON search with a JSON index

The key idea of a JSON index is building an inverted index over paths and "path + value" pairs. Each path of a JSON document (and for equality checks, the scalar value at that path) is encoded into a token. For each token, the index stores a list of primary key values of the corresponding table rows. Queries to the index are reduced to an inverted search over these tokens — following the same scheme as a full-text index, but with its own JSON tokenizer.

The JSON index does not store a copy of the entire JSON document, but decomposes it into individual tokens:

- For each path in the JSON tree, a token 'path exists' is created.
- if the path leads to a scalar value (string, number, boolean, or `null`), a «path + value» token is additionally created.

Arrays are transparent in this case: array elements are indexed under the same path as the array itself, so the index responds equally to queries to `$.items`, `$.items[0]`, and `$.items[*]`.

For example, for a document:

```
{
    "id": 42042,
    "name": "Michael",
    "email": null,
    "items": [null, "str"],
    "parts": {
        "key": "k1",
        "value": false
    }
}
```

Conceptually, the following tokens are indexed (without regard to the specifics of the internal representation):

Token	What it finds
`$`	document is not equal to `NULL`
`$.id` / `$.id == 42042`	path `id` exists / equals `42042`
`$.name` / `$.name == "Michael"`	path `name` exists / equals `"Michael"`
`$.email` / `$.email == null`	path `email` exists / equals `null`
`$.items` / `$.items == null` / `$.items == "str"`	path `items` exists / contains element `null` / contains element `"str"`
`$.parts`	path `parts` exists
`$.parts.key` / `$.parts.key == "k1"`	nested path exists / equals `"k1"`
`$.parts.value` / `$.parts.value == false`	nested path exists or equals `false`

JSON index allows:

- find rows by path existence using JSON_EXISTS
- find rows by a value in a path using JSON_VALUE or a filter predicate within `JSON_EXISTS`.

When executing a query, the JSON index can be automatically used by the optimizer. It is enough to write a regular condition `WHERE` with expressions `JSON_EXISTS` or `JSON_VALUE` on an indexed JSON column. The optimizer recognizes such a predicate and applies reading by the JSON index instead of scanning records of the source table. Additionally, the JSON index, like any other secondary index, can be forcefully used by specifying its name in the section `VIEW IndexName`.

If the predicate cannot be converted into an index access, the behavior of YDB depends on whether the required JSON index was explicitly specified in the query:

- When the optimizer automatically selects an index, the query is simply executed without index acceleration (the result remains correct).
- When explicitly specifying an index via the `VIEW` expression, an error is returned.

For more information, see VIEW (JSON index).

Additional information:

- JSON indexes — an overview of supported capabilities, predicates, and limitations.
- JSON functions — reference on `JSON_EXISTS`, `JSON_VALUE`, and the JsonPath language.
- VIEW (JSON index) — syntax for queries with a JSON index.

Local indexes

Local index concept

A local index is an auxiliary structure that is stored together with the table data and is used when reading from storage. It does not materialize a separate index table.

Local indexes act as read filters for the main table: the query optimizer and the storage layer use them to skip irrelevant data fragments during scanning.

Currently, YDB implements local Bloom indexes and a min_max index.

Bloom indexes

Bloom indexes are a special case of a local index built on the Bloom filter.

When reading, for each data fragment the index checks whether the sought value (or a set of n-grams) can occur in it. If the filter reports that the value definitely does not occur, the fragment is skipped without reading the indexed columns. If the filter "passes" the check, the value may be present — including due to a false positive — and the fragment must be read. This reduces the amount of data actually read for selective queries.

Bloom index types

- `bloom_filter` — builds a filter on the exact values of the indexed column. Suitable for equality conditions (`=`), list membership checks (`IN`), and other equality comparisons for supported types.
- `bloom_ngram_filter` — builds a filter on n-grams of a string column (`String`, `Utf8`). When searching by substring or pattern (`LIKE`), the query is split into n-grams; if the fragment index lacks at least one of the required n-grams, the sought substring cannot be in it, and the fragment is skipped. Supported only in columnar tables.

Local Bloom indexes

Type `bloom_filter` works in both row-based (OLTP) and columnar (OLAP) tables, but the implementation differs:

- In row-based tables, the filter is built as a prefix Bloom filter on the left prefix of the primary key. The indexed columns must form a continuous leading subset of the primary key columns. This speeds up point reads and range scans that restrict the leading key columns. To create a prefix Bloom filter, use ALTER TABLE ... ADD INDEX; to drop it, use ALTER TABLE ... DROP INDEX.
- In columnar tables, the filter is built on the values of the indexed column in each data fragment (portion) and is used in analytical scans to skip fragments that do not contain the sought value.

Limitations

- `COVER (...)` and extra index columns are not supported.
- Column-oriented tables allow only one indexed column. Row-oriented tables allow multiple indexed columns.
- `bloom_ngram_filter` is not supported on row-oriented tables.
- On row-oriented tables, the indexed columns of a `bloom_filter` must be a left prefix of the primary key. Non-prefix column sets are rejected.
- On row-oriented tables, two `bloom_filter` indexes cannot share the same prefix length (the same set of leading primary-key columns).

Additional materials

- Bloom indexes
- ALTER TABLE ADD INDEX

min_max index

A min_max index is a special case of a local index that stores the minimum and maximum value of one indexed column for each data fragment.

When reading with a special kind of filter on a column with a min_max index, YDB first reads the minimum and maximum values stored for the fragment and checks whether the filter interval intersects this range. If the intervals do not intersect, the predicate is guaranteed to be false on all values of the fragment, so the fragment can be skipped. This is useful for range predicates and equality conditions (a special case of a range predicate) on columns with a small spread between the minimum and maximum values within stored fragments.

Example:

In storage, for the `events` table, one of the fragments of column `level` of type `Int32` contains values `[5, 5, 9, 5, 9, 13]`. Then the minimum value is 5, the maximum is 13. For query `SELECT * FROM events WHERE level = 15`, the filter interval is `[15, 15]`. It does not intersect with interval `[5, 13]`, so such a fragment does not need to be read from storage.

Min-max index predicates

The optimizer can use the min_max index for predicates `=`, `<`, `<=`, `>`, `>=`, `BETWEEN`, as well as compatible combinations with `AND` or `OR`.

Limitations

- Supported only for columnar tables.
- `ON (...)` must specify exactly one column.
- `COVER (...)` and additional data columns are not supported.
- Specific parameters of `WITH (...)` are not supported.
- `ALTER INDEX` is not supported for the min_max index.
- Columns of types `Json` and `JsonDocument` are not supported.

Additional materials

- min_max index
- ALTER TABLE ADD INDEX

Federated queries

Federated queries allow you to retrieve information from various data sources without the need to transfer data from those sources into YDB. Using YQL queries, you can access external databases without duplicating data between systems.

To work with data stored in external DBMSs, simply create an external data source. To work with unschematized data stored in S3 buckets, you additionally need to create an external table. In both cases, you must first create secrets that store confidential data required for authentication in external systems.

You can learn about the internal structure of the federated query processing system in the section on architecture. Detailed information on working with various data sources is provided in the respective sections:

- S3

Experimental sources data

- ClickHouse
- Greenplum
- Microsoft SQL Server
- MySQL
- PostgreSQL
- YDB (federated queries between YDB databases)

Federated query processing system architecture

External data sources and external tables

The key element of the YDB federated query processing system is the concept of an external data source. Such sources can include relational DBMSs, object storages, and other data storage systems. When processing a federated query, YDB streams data from external systems and allows performing the same range of operations on them as on local data.

To work with data stored in external systems, YDB must have information about the internal structure of this data (for example, the number, names, and types of columns in tables). Some sources provide such metadata about the data together with the data itself, while working with other, unschematized sources requires specifying this metadata externally. External tables serve this latter purpose.

After registering external data sources and (if necessary) external tables in YDB, the client can start describing federated queries.

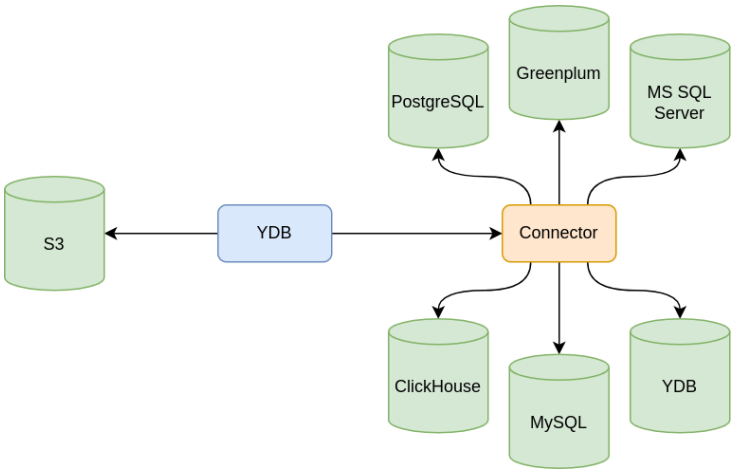
Connectors

During the execution of federated queries, YDB needs to access third-party data storage systems over the network, which requires using their client libraries. The introduction of such dependencies negatively affects the size of the codebase, compilation time, and the size of binary files YDB, as well as the stability of the entire product.

The list of supported data sources for federated queries is constantly expanding.
The most popular sources, such as S3, are natively supported by YDB. However, not all users need support for all sources at the same time. It can be enabled optionally using *connectors* - special microservices that implement a unified interface for accessing external data sources.

The functions of connectors include:

- Translation of YQL queries into queries in a language specific to the external source (for example, into queries in another SQL dialect or into HTTP API calls).
- Establishing network connections to data sources.
- Conversion of data extracted from external sources into a columnar representation in the Arrow IPC Stream format supported by YDB.



Thus, connectors form an abstraction layer that hides the specifics of external data sources from YDB. The conciseness of the connector interface makes it easy to expand the list of supported sources with minimal changes to the YDB code.

Users can deploy one of the ready-made connectors or write their own implementation in any programming language according to the gRPC specification.

List of supported external data sources

YDB has built-in support in `ydbd` for the following external data sources:

- S3

Through the fq-connector-go connector, access to a number of experimental external DBMSs is additionally supported. For details, see the Federated queries section.

Working with ClickHouse databases

> **Warning**
>
> External connectors are an experimental feature of YDB. To work with external DBMS through connectors, you need to deploy fq-connector-go and explicitly enable the corresponding sources in the cluster configuration YDB. The functionality may change, so use in production is not recommended without thorough testing.

This section describes basic information about working with an external database ClickHouse.

To work with an external ClickHouse database, you need to follow these steps:

1. Create a secret containing the password for connecting to the database.

   ```
   CREATE SECRET clickhouse_datasource_user_password WITH (value = "<password>");
   ```

2. Create an external data source that describes the target database inside the ClickHouse cluster. To connect to ClickHouse, you can use either the native TCP protocol (`PROTOCOL="NATIVE"`) or the HTTP protocol (`PROTOCOL="HTTP"`). Enable encryption of connections to the external database using the `USE_TLS="TRUE"` parameter.

   ```
   CREATE EXTERNAL DATA SOURCE clickhouse_datasource WITH (
       SOURCE_TYPE="ClickHouse",
       LOCATION="<host>:<port>",
       DATABASE_NAME="<database>",
       AUTH_METHOD="BASIC",
       LOGIN="<login>",
       PASSWORD_SECRET_PATH="clickhouse_datasource_user_password",
       PROTOCOL="NATIVE",
       USE_TLS="TRUE"
   );
   ```

3. Deploy the connector and configure the YDB dynamic nodes to interact with it. Additionally, ensure network access from the YDB dynamic nodes to the external data source (at the address specified in the `LOCATION` parameter of the `CREATE EXTERNAL DATA SOURCE` request). If network connection encryption to the external source was enabled in the previous step, the connector will use the system's root certificates. More details on TLS configuration can be found in the guide on deploying the connector.
4. Execute a query to the database.

Query syntax

The following SQL query form is used to work with ClickHouse:

```
SELECT * FROM clickhouse_datasource.<table_name>
```

where:

- `clickhouse_datasource`: external data source identifier
- `<table_name>` is the name of the table inside the external data source.

Limitations

When working with ClickHouse clusters, there are a number of limitations:

1. External sources are available only for reading data through `SELECT` queries. The federated query processing engine currently does not support queries that modify tables in external sources.
2. If the date value stored in the external data source is outside the allowed range for YDB (all dates used must be later than 1970-01-01 but earlier than 2105-12-31), such a value in YDB will be converted to `NULL`.
3. The YDB federated query processing system is capable of delegating the execution of certain parts of a query to the system acting as the data source. Query fragments are passed through YDB directly to the external system and processed by them. This optimization, known as "predicate pushdown", significantly reduces the amount of data transferred from the source to the federated query processing engine. This reduces network load and saves computational resources for the federated YDB.

   A specific case of predicate pushdown, when the filtering expressions are specified after the `WHERE` keyword, are passed to the data source, is called "filter pushdown". Filter pushdown is possible when using:

Description	Example
`NULL` checks	`WHERE column1 IS NULL` or `WHERE column1 IS NOT NULL`
Logical conditions `OR`, `NOT`, `AND` and parentheses for controlling calculation priority.	`WHERE column1 IS NULL OR (column2 IS NOT NULL AND column3 > 10)`.
Comparison operators with other columns or constants.	`WHERE column1 > column2 OR column3 <= 10`, `WHERE column1 + column2 > 10`, `WHERE column1 = (10 + 10)`

   When using other types of filters, pushdown to the data source is not performed: filtering of the external table rows will be executed by the federated YDB, which means that YDB will perform a full scan of the external table when processing the query.

   Supported data types for filter pushdown:

Data type YDB
`Bool`
`Int8`

`Uint8`
`Int16`
`Uint16`
`Int32`
`Uint32`
`Int64`
`Uint64`
`Float`
`Double`
`String`

Supported data types

By default, ClickHouse columns physically cannot contain the `NULL` value; however, you can create a table with columns of optional, or nullable, types. The column types displayed by YDB when retrieving data from an external ClickHouse database will depend on whether the ClickHouse table uses primitive or optional types. However, due to the limitations of YDB types used for storing dates and times discussed above, all similar ClickHouse types are displayed in YDB as optional.

Below are the tables of correspondence between ClickHouse and YDB types. All other data types, except those listed, are not supported.

Primitive data types

ClickHouse data type	Data type YDB	Notes
`Bool`	`Bool`	
`Int8`	`Int8`	
`UInt8`	`Uint8`	
`Int16`	`Int16`	
`UInt16`	`Uint16`	
`Int32`	`Int32`	
`UInt32`	`Uint32`	
`Int64`	`Int64`	
`UInt64`	`Uint64`	
`Float32`	`Float`	
`Float64`	`Double`	
`Date`	`Date`	
`Date32`	`Optional<Date>`	Valid date range from 1970-01-01 to 2105-12-31. If the value goes outside the range, `NULL` is returned.
`DateTime`	`Optional<DateTime>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. If the value goes out of range, the value `NULL` is returned.
`DateTime64`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. If the value goes out of range, the value `NULL` is returned.
`String`	`String`	
`FixedString`	`String`	Null bytes `FixedString` are transferred to `String` unchanged.

Optional data types

ClickHouse data type	Data type YDB	Notes
`Nullable(Bool)`	`Optional<Bool>`	
`Nullable(Int8)`	`Optional<Int8>`	
`Nullable(UInt8)`	`Optional<Uint8>`	
`Nullable(Int16)`	`Optional<Int16>`	
`Nullable(UInt16)`	`Optional<Uint16>`	
`Nullable(Int32)`	`Optional<Int32>`	
`Nullable(UInt32)`	`Optional<Uint32>`	

`Nullable(Int64)`	`Optional<Int64>`	
`Nullable(UInt64)`	`Optional<Uint64>`	
`Nullable(Float32)`	`Optional<Float>`	
`Nullable(Float64)`	`Optional<Double>`	
`Nullable(Date)`	`Optional<Date>`	
`Nullable(Date32)`	`Optional<Date>`	Valid date range from 1970-01-01 to 2105-12-31. If the value goes beyond the range boundaries, `NULL` is returned.
`Nullable(DateTime)`	`Optional<DateTime>`	Allowed time value range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. If the value goes beyond the range boundaries, the `NULL` value is returned.
`Nullable(DateTime64)`	`Optional<Timestamp>`	Allowed time value range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. If the value goes beyond the range boundaries, the value `NULL` is returned.
`Nullable(String)`	`Optional<String>`	
`Nullable(FixedString)`	`Optional<String>`	Null bytes `FixedString` are transferred to `String` without changes.

Working with Greenplum databases

> **Warning**
>
> External connectors are an experimental feature of YDB. To work with external DBMS through connectors, you need to deploy fq-connector-go and explicitly enable the corresponding sources in the cluster configuration YDB. The functionality may change, so use in production is not recommended without thorough testing.

This section describes the basic information about working with an external Greenplum database. Since Greenplum is based on PostgreSQL, integrations with them work similarly, and some links below may lead to PostgreSQL documentation.

To work with an external Greenplum database, you need to perform the following steps:

1. Create a secret containing the password for connecting to the database.

   ```
   CREATE SECRET greenplum_datasource_user_password WITH (value = "<password>");
   ```

2. Create an external data source that describes a specific database within a Greenplum cluster. Pass the network address of the Greenplum master node in the `LOCATION` parameter. By default, the namespace `public` is used for reading, but this value can be changed using the optional `SCHEMA` parameter. You can enable encryption of connections to the external database using the `USE_TLS="TRUE"` parameter.

   ```
   CREATE EXTERNAL DATA SOURCE greenplum_datasource WITH (
       SOURCE_TYPE="Greenplum",
       LOCATION="<host>:<port>",
       DATABASE_NAME="<database>",
       AUTH_METHOD="BASIC",
       LOGIN="user",
       PASSWORD_SECRET_PATH="greenplum_datasource_user_password",
       USE_TLS="TRUE",
       SCHEMA="<schema>"
   );
   ```

3. Deploy the connector and configure the YDB dynamic nodes to interact with it. Additionally, ensure network access from the YDB dynamic nodes to the external data source (at the address specified in the `LOCATION` parameter of the `CREATE EXTERNAL DATA SOURCE` request). If network connection encryption to the external source was enabled in the previous step, the connector will use the system's root certificates. More details on TLS configuration can be found in the guide on deploying the connector.
4. Execute a query to the database.

Query syntax

The following SQL query form is used to work with Greenplum:

```
SELECT * FROM greenplum_datasource.<table_name>
```

where:

- `greenplum_datasource`: external data source identifier
- `<table_name>` is the name of the table inside the external data source.

Limitations

When working with Greenplum clusters, there are a number of limitations:

1. External sources are available only for reading data through `SELECT` queries. The federated query processing engine currently does not support queries that modify tables in external sources.
2. If the date value stored in the external data source is outside the allowed range for YDB (all dates used must be later than 1970-01-01 but earlier than 2105-12-31), such a value in YDB will be converted to `NULL`.
3. The YDB federated query processing system is capable of delegating the execution of certain parts of a query to the system acting as the data source. Query fragments are passed through YDB directly to the external system and processed by them. This optimization, known as "predicate pushdown", significantly reduces the amount of data transferred from the source to the federated query processing engine. This reduces network load and saves computational resources for the federated YDB.

   A specific case of predicate pushdown, when the filtering expressions are specified after the `WHERE` keyword, are passed to the data source, is called "filter pushdown". Filter pushdown is possible when using:

Description	Example
`NULL` checks	`WHERE column1 IS NULL` or `WHERE column1 IS NOT NULL`
Logical conditions `OR`, `NOT`, `AND` and parentheses for controlling calculation priority.	`WHERE column1 IS NULL OR (column2 IS NOT NULL AND column3 > 10)`.
Comparison operators with other columns or constants.	`WHERE column1 > column2 OR column3 <= 10`, `WHERE column1 + column2 > 10`, `WHERE column1 = (10 + 10)`

   When using other types of filters, pushdown to the data source is not performed: filtering of the external table rows will be executed by the federated YDB, which means that YDB will perform a full scan of the external table when processing the query.

   Supported data types for filter pushdown:

Data type YDB
`Bool`

`Int8`
`Int16`
`Int32`
`Int64`
`Float`
`Double`

Supported data types

In the Greenplum database, the optionality flag of column values (whether a column is allowed or not allowed to contain `NULL` values) is not part of the data type system. The `NOT NULL` constraint for each column is implemented as the `attnotnull` attribute in the pg_attribute system catalog, i.e., at the table metadata level. Consequently, all basic Greenplum types can contain `NULL` values by default, and in the YDB type system they must be mapped to optional types.

Below is a table of correspondence between Greenplum types and YDB. All other data types, except those listed, are not supported.

Greenplum data type	Data type YDB	Notes
`boolean`	`Optional<Bool>`	
`smallint`	`Optional<Int16>`	
`int2`	`Optional<Int16>`	
`integer`	`Optional<Int32>`	
`int`	`Optional<Int32>`	
`int4`	`Optional<Int32>`	
`serial`	`Optional<Int32>`	
`serial4`	`Optional<Int32>`	
`bigint`	`Optional<Int64>`	
`int8`	`Optional<Int64>`	
`bigserial`	`Optional<Int64>`	
`serial8`	`Optional<Int64>`	
`real`	`Optional<Float>`	
`float4`	`Optional<Float>`	
`double precision`	`Optional<Double>`	
`float8`	`Optional<Double>`	
`json`	`Optional<Json>`	
`date`	`Optional<Date>`	Valid date range from 1970-01-01 to 2105-12-31. If the value goes beyond the range boundaries, `NULL` is returned.
`timestamp`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. If the value goes beyond the range boundaries, the value `NULL` is returned.
`bytea`	`Optional<String>`	
`character`	`Optional<Utf8>`	Default collation rules, the string is padded with spaces to the required length.
`character varying`	`Optional<Utf8>`	Default collation rules.
`text`	`Optional<Utf8>`	Default collation rules.

Working with Microsoft SQL Server databases

> **Warning**
>
> External connectors are an experimental feature of YDB. To work with external DBMS through connectors, you need to deploy fq-connector-go and explicitly enable the corresponding sources in the cluster configuration YDB. The functionality may change, so use in production is not recommended without thorough testing.

This section describes the basic information about working with an external Microsoft SQL Server database.

To work with an external Microsoft SQL Server database, you need to perform the following steps:

1. Create a secret containing the password for connecting to the database.

   ```
   CREATE SECRET ms_sql_server_datasource_user_password WITH (value = "<password>");
   ```

2. Create an external data source that describes a specific Microsoft SQL Server database. The `LOCATION` parameter contains the network address of the Microsoft SQL Server instance to which the connection is made. The `DATABASE_NAME` parameter specifies the database name (for example, `master`). The values of the `LOGIN` and `PASSWORD_SECRET_PATH` parameters are used for authentication to the external database. You can enable encryption of connections to the external database using the `USE_TLS="TRUE"` parameter.

   ```
   CREATE EXTERNAL DATA SOURCE ms_sql_server_datasource WITH (
       SOURCE_TYPE="MsSQLServer",
       LOCATION="<host>:<port>",
       DATABASE_NAME="<database>",
       AUTH_METHOD="BASIC",
       LOGIN="user",
       PASSWORD_SECRET_PATH="ms_sql_server_datasource_user_password",
       USE_TLS="TRUE"
   );
   ```

3. Deploy the connector and configure the YDB dynamic nodes to interact with it. Additionally, ensure network access from the YDB dynamic nodes to the external data source (at the address specified in the `LOCATION` parameter of the `CREATE EXTERNAL DATA SOURCE` request). If network connection encryption to the external source was enabled in the previous step, the connector will use the system's root certificates. More details on TLS configuration can be found in the guide on deploying the connector.
4. Execute a query on the database.

Query syntax

The following SQL query form is used to work with Microsoft SQL Server:

```
SELECT * FROM ms_sql_server_datasource.<table_name>
```

where:

- `ms_sql_server_datasource`: external data source identifier
- `<table_name>` is the name of the table inside the external data source.

Limitations

When working with Microsoft SQL Server clusters, there are a number of limitations:

1. External sources are available only for reading data through `SELECT` queries. The federated query processing engine currently does not support queries that modify tables in external sources.
2. If the date value stored in the external data source is outside the allowed range for YDB (all dates used must be later than 1970-01-01 but earlier than 2105-12-31), such a value in YDB will be converted to `NULL`.
3. The YDB federated query processing system is capable of delegating the execution of certain parts of a query to the system acting as the data source. Query fragments are passed through YDB directly to the external system and processed by them. This optimization, known as "predicate pushdown", significantly reduces the amount of data transferred from the source to the federated query processing engine. This reduces network load and saves computational resources for the federated YDB.

   A specific case of predicate pushdown, when the filtering expressions are specified after the `WHERE` keyword, are passed to the data source, is called "filter pushdown". Filter pushdown is possible when using:

Description	Example
`NULL` checks	`WHERE column1 IS NULL` or `WHERE column1 IS NOT NULL`
Logical conditions `OR`, `NOT`, `AND` and parentheses for controlling calculation priority.	`WHERE column1 IS NULL OR (column2 IS NOT NULL AND column3 > 10)`.
Comparison operators with other columns or constants.	`WHERE column1 > column2 OR column3 <= 10`, `WHERE column1 + column2 > 10`, `WHERE column1 = (10 + 10)`

   When using other types of filters, pushdown to the data source is not performed: filtering of the external table rows will be executed by the federated YDB, which means that YDB will perform a full scan of the external table when processing the query.

Data type YDB
`Bool`
`Int8`

`Int16`
`Int32`
`Int64`
`Float`
`Double`

Supported data types

In a Microsoft SQL Server database, the optionality flag of column values (whether the column is allowed or prohibited from containing `NULL` values) is not part of the data type system. The `NOT NULL` constraint for any column of any table is stored as the value of the `IS_NULLABLE` column of the INFORMATION_SCHEMA.COLUMNS system table, that is, at the table metadata level. Consequently, all basic Microsoft SQL Server data types can by default contain `NULL` values, and in the YDB type system they must be mapped to optional types.

Below is a table of correspondence between Microsoft SQL Server types and YDB. All other data types, except those listed, are not supported.

Microsoft SQL Server data type	Data type YDB	Notes
`bit`	`Optional<Bool>`	
`tinyint`	`Optional<Int8>`	
`smallint`	`Optional<Int16>`	
`int`	`Optional<Int32>`	
`bigint`	`Optional<Int64>`	
`real`	`Optional<Float>`	
`float`	`Optional<Double>`	
`date`	`Optional<Date>`	Valid date range from 1970-01-01 to 2105-12-31. If the value goes beyond the range boundaries, `NULL` is returned.
`smalldatetime`	`Optional<Datetime>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. If the value goes beyond the range boundaries, the value `NULL` is returned.
`datetime`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. If the value goes beyond the range boundaries, the value `NULL` is returned.
`datetime2`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. If the value goes beyond the range boundaries, the value `NULL` is returned.
`binary`	`Optional<String>`	
`varbinary`	`Optional<String>`	
`image`	`Optional<String>`	
`char`	`Optional<Utf8>`	
`varchar`	`Optional<Utf8>`	
`text`	`Optional<Utf8>`	
`nchar`	`Optional<Utf8>`	
`nvarchar`	`Optional<Utf8>`	
`ntext`	`Optional<Utf8>`	

Working with MySQL databases

> **Warning**
>
> External connectors are an experimental feature of YDB. To work with external DBMS through connectors, you need to deploy fq-connector-go and explicitly enable the corresponding sources in the cluster configuration YDB. The functionality may change, so use in production is not recommended without thorough testing.

This section describes basic information about working with an external MySQL database.

To work with an external MySQL database, you need to perform the following steps:

1. Create a secret containing the password for connecting to the database.

   ```
   CREATE SECRET mysql_datasource_user_password WITH (value = "<password>");
   ```

2. Create an external data source describing a specific MySQL database. The `LOCATION` parameter contains the network address of the MySQL instance to connect to. `DATABASE_NAME` specifies the database name (for example, `mysql`). For authentication to the external database, the values of the `LOGIN` and `PASSWORD_SECRET_PATH` parameters are used. You can enable encryption of connections to the external database using the `USE_TLS="TRUE"` parameter.

   ```
   CREATE EXTERNAL DATA SOURCE mysql_datasource WITH (
       SOURCE_TYPE="MySQL",
       LOCATION="<host>:<port>",
       DATABASE_NAME="<database>",
       AUTH_METHOD="BASIC",
       LOGIN="user",
       PASSWORD_SECRET_PATH="mysql_datasource_user_password",
       USE_TLS="TRUE"
   );
   ```

3. Deploy the connector and configure the YDB dynamic nodes to interact with it. Additionally, ensure network access from the YDB dynamic nodes to the external data source (at the address specified in the `LOCATION` parameter of the `CREATE EXTERNAL DATA SOURCE` request). If network connection encryption to the external source was enabled in the previous step, the connector will use the system's root certificates. More details on TLS configuration can be found in the guide on deploying the connector.
4. Execute a query to the database.

Query syntax

The following SQL query form is used to work with MySQL:

```
SELECT * FROM mysql_datasource.<table_name>
```

where:

- `mysql_datasource`: external data source identifier
- `<table_name>` is the name of the table inside the external data source.

Limitations

When working with MySQL clusters, there are a number of limitations:

1. External sources are available only for reading data through `SELECT` queries. The federated query processing engine currently does not support queries that modify tables in external sources.
2. If the date value stored in the external data source is outside the allowed range for YDB (all dates used must be later than 1970-01-01 but earlier than 2105-12-31), such a value in YDB will be converted to `NULL`.
3. The YDB federated query processing system is capable of delegating the execution of certain parts of a query to the system acting as the data source. Query fragments are passed through YDB directly to the external system and processed by them. This optimization, known as "predicate pushdown", significantly reduces the amount of data transferred from the source to the federated query processing engine. This reduces network load and saves computational resources for the federated YDB.

   A specific case of predicate pushdown, when the filtering expressions are specified after the `WHERE` keyword, are passed to the data source, is called "filter pushdown". Filter pushdown is possible when using:

Description	Example
`NULL` checks	`WHERE column1 IS NULL` or `WHERE column1 IS NOT NULL`
Logical conditions `OR`, `NOT`, `AND` and parentheses for controlling calculation priority.	`WHERE column1 IS NULL OR (column2 IS NOT NULL AND column3 > 10)`.
Comparison operators with other columns or constants.	`WHERE column1 > column2 OR column3 <= 10`, `WHERE column1 + column2 > 10`, `WHERE column1 = (10 + 10)`

   When using other types of filters, pushdown to the data source is not performed: filtering of the external table rows will be executed by the federated YDB, which means that YDB will perform a full scan of the external table when processing the query.

   Supported data types for filter pushdown:

Data type YDB
`Bool`
`Int8`

`Uint8`
`Int16`
`Uint16`
`Int32`
`Uint32`
`Int64`
`Uint64`
`Float`
`Double`

Supported data types

In MySQL, the optionality of column values (whether a column is allowed or not allowed to contain `NULL` values) is not part of the data type system. The `NOT NULL` constraint for any column of any table is stored as the value of the `IS_NULLABLE` column of the INFORMATION_SCHEMA.COLUMNS system table, i.e., at the table metadata level. Consequently, all basic MySQL types can by default contain `NULL` values, and in the YDB type system they must be mapped to optional types.

Below is a table of correspondence between MySQL types and YDB. All other data types, except those listed, are not supported.

MySQL data type	Data type YDB	Notes
`bool`	`Optional<Bool>`	
`tinyint`	`Optional<Int8>`	
`tinyint unsigned`	`Optional<Uint8>`	
`smallint`	`Optional<Int16>`	
`smallint unsigned`	`Optional<Uint16>`	
`mediumint`	`Optional<Int32>`	
`mediumint unsigned`	`Optional<Uint32>`	
`int`	`Optional<Int32>`	
`int unsigned`	`Optional<Uint32>`	
`bigint`	`Optional<Int64>`	
`bigint unsigned`	`Optional<Uint64>`	
`float`	`Optional<Float>`	
`real`	`Optional<Float>`	
`double`	`Optional<Double>`	
`date`	`Optional<Date>`	Valid date range from 1970-01-01 to 2105-12-31. If the value goes beyond the range boundaries, `NULL` is returned.
`datetime`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. If the value goes beyond the range boundaries, the value `NULL` is returned.
`timestamp`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. If the value goes beyond the range boundaries, the value `NULL` is returned.
`tinyblob`	`Optional<String>`	
`blob`	`Optional<String>`	
`mediumblob`	`Optional<String>`	
`longblob`	`Optional<String>`	
`tinytext`	`Optional<String>`	
`text`	`Optional<String>`	
`mediumtext`	`Optional<String>`	
`longtext`	`Optional<String>`	
`char`	`Optional<Utf8>`	
`varchar`	`Optional<Utf8>`	
`binary`	`Optional<String>`	
`varbinary`	`Optional<String>`	

`json`	`Optional<Json>`	

Working with PostgreSQL databases

> **Warning**
>
> External connectors are an experimental feature of YDB. To work with external DBMS through connectors, you need to deploy fq-connector-go and explicitly enable the corresponding sources in the cluster configuration YDB. The functionality may change, so use in production is not recommended without thorough testing.

This section describes the basic information about working with an external PostgreSQL database.

To work with an external PostgreSQL database, you need to perform the following steps:

1. Create a secret containing the password for connecting to the database.

   ```
   CREATE SECRET postgresql_datasource_user_password WITH (value = "<password>");
   ```

2. Create an external data source that describes a specific database within a PostgreSQL cluster. By default, the namespace `public` is used for reading, but this value can be changed using the optional parameter `SCHEMA`. The network connection is established via the standard (Frontend/Backend Protocol) over TCP transport (`PROTOCOL="NATIVE"`). Encryption of connections to the external database can be enabled using the parameter `USE_TLS="TRUE"`.

   ```
   CREATE EXTERNAL DATA SOURCE postgresql_datasource WITH (
       SOURCE_TYPE="PostgreSQL",
       LOCATION="<host>:<port>",
       DATABASE_NAME="<database>",
       AUTH_METHOD="BASIC",
       LOGIN="user",
       PASSWORD_SECRET_PATH="postgresql_datasource_user_password",
       PROTOCOL="NATIVE",
       USE_TLS="TRUE",
       SCHEMA="<schema>"
   );
   ```

3. Deploy the connector and configure the YDB dynamic nodes to interact with it. Additionally, ensure network access from the YDB dynamic nodes to the external data source (at the address specified in the `LOCATION` parameter of the `CREATE EXTERNAL DATA SOURCE` request). If network connection encryption to the external source was enabled in the previous step, the connector will use the system's root certificates. More details on TLS configuration can be found in the guide on deploying the connector.
4. Execute a query to the database.

Query syntax

The following SQL query form is used to work with PostgreSQL:

```
SELECT * FROM postgresql_datasource.<table_name>
```

where:

- `postgresql_datasource`: external data source identifier
- `<table_name>` is the name of the table inside the external data source.

Limitations

When working with PostgreSQL clusters, there are a number of limitations:

1. External sources are available only for reading data through `SELECT` queries. The federated query processing engine currently does not support queries that modify tables in external sources.
2. If the date value stored in the external data source is outside the allowed range for YDB (all dates used must be later than 1970-01-01 but earlier than 2105-12-31), such a value in YDB will be converted to `NULL`.
3. The YDB federated query processing system is capable of delegating the execution of certain parts of a query to the system acting as the data source. Query fragments are passed through YDB directly to the external system and processed by them. This optimization, known as "predicate pushdown", significantly reduces the amount of data transferred from the source to the federated query processing engine. This reduces network load and saves computational resources for the federated YDB.

   A specific case of predicate pushdown, when the filtering expressions are specified after the `WHERE` keyword, are passed to the data source, is called "filter pushdown". Filter pushdown is possible when using:

Description	Example
`NULL` checks	`WHERE column1 IS NULL` or `WHERE column1 IS NOT NULL`
Logical conditions `OR`, `NOT`, `AND` and parentheses for controlling calculation priority.	`WHERE column1 IS NULL OR (column2 IS NOT NULL AND column3 > 10)`.
Comparison operators with other columns or constants.	`WHERE column1 > column2 OR column3 <= 10`, `WHERE column1 + column2 > 10`, `WHERE column1 = (10 + 10)`

   When using other types of filters, pushdown to the data source is not performed: filtering of the external table rows will be executed by the federated YDB, which means that YDB will perform a full scan of the external table when processing the query.

   Supported data types for filter pushdown:

Data type YDB
`Bool`

`Int8`
`Int16`
`Int32`
`Int64`
`Float`
`Double`
`Decimal`

Supported data types

In a PostgreSQL database, the optionality of column values (whether a column is allowed or not allowed to contain `NULL` values) is not part of the data type system. The constraint `NOT NULL` for each column is implemented as an attribute `attnotnull` in the system catalog pg_attribute, that is, at the table metadata level. Consequently, all basic PostgreSQL types can by default contain `NULL` values, and in the YDB type system they must be mapped to optional types.

Below is a table of correspondence between PostgreSQL types and YDB. All other data types, except those listed, are not supported.

PostgreSQL data type	Data type YDB	Notes
`boolean`	`Optional<Bool>`	
`smallint`	`Optional<Int16>`	
`int2`	`Optional<Int16>`	
`integer`	`Optional<Int32>`	
`int`	`Optional<Int32>`	
`int4`	`Optional<Int32>`	
`serial`	`Optional<Int32>`	
`serial4`	`Optional<Int32>`	
`bigint`	`Optional<Int64>`	
`int8`	`Optional<Int64>`	
`bigserial`	`Optional<Int64>`	
`serial8`	`Optional<Int64>`	
`real`	`Optional<Float>`	
`float4`	`Optional<Float>`	
`double precision`	`Optional<Double>`	
`float8`	`Optional<Double>`	
`date`	`Optional<Date>`	Valid date range from 1970-01-01 to 2105-12-31. If the value goes outside the range, `NULL` is returned.
`timestamp`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. If the value goes outside the range, the value `NULL` is returned.
`bytea`	`Optional<String>`	
`character`	`Optional<Utf8>`	Default collation rules, the string is padded with spaces to the required length.
`character varying`	`Optional<Utf8>`	Default collation rules.
`text`	`Optional<Utf8>`	Default collation rules.
`json`	`Optional<Json>`	
`numeric(p,s)`	`Optional<Decimal(p,s)>`	`p` (precision) - the total number of digits in the number, `s` (scale) - the number of digits after the decimal point. Types `numeric` without parameters (so-called "unconstrained") are converted to `Optional<Decimal(35, 0)>`. Types `numeric` with `p > 35` or `s < 0` are not supported.

Working with YDB databases

> **Warning**
>
> External connectors are an experimental feature of YDB. To work with external DBMS through connectors, you need to deploy fq-connector-go and explicitly enable the corresponding sources in the cluster configuration YDB. The functionality may change, so use in production is not recommended without thorough testing.

YDB can act as an external data source for another YDB database. This section describes how to set up collaboration between two independent YDB databases in federated query processing mode.

To connect to an external YDB database from another YDB database that acts as a federated query processing engine, you need to perform the following steps on the latter:

1. Prepare authentication credentials to access the remote YDB database. Currently, in federated queries to YDB, the login and password authentication method is available (other methods are not supported). The password to the external database is stored as a secret:

   ```
   CREATE SECRET ydb_datasource_user_password WITH (value = "<password>");
   ```

2. Create an external data source describing an external YDB database. The `LOCATION` parameter contains the network address of the YDB instance to which a network connection is made. In `DATABASE_NAME`, specify the database name (for example, `local`). For authentication to the external database, use the values of the `LOGIN` and `PASSWORD_SECRET_PATH` parameters. You can enable encryption of connections to the external database using the `USE_TLS="TRUE"` parameter. If encryption is enabled, then in the `<port>` field of the `LOCATION` parameter, you must specify the gRPCs port of the external YDB; otherwise, specify the gRPC port.

   ```
   CREATE EXTERNAL DATA SOURCE ydb_datasource WITH (
       SOURCE_TYPE="Ydb",
       LOCATION="<host>:<port>",
       DATABASE_NAME="<database>",
       AUTH_METHOD="BASIC",
       LOGIN="user",
       PASSWORD_SECRET_PATH="ydb_datasource_user_password",
       USE_TLS="TRUE"
   );
   ```

3. Deploy the connector and configure the YDB dynamic nodes to interact with it. Additionally, ensure network access from the YDB dynamic nodes to the external data source (at the address specified in the `LOCATION` parameter of the `CREATE EXTERNAL DATA SOURCE` request). If network connection encryption to the external source was enabled in the previous step, the connector will use the system's root certificates. More details on TLS configuration can be found in the guide on deploying the connector.
4. Execute a query to an external data source.

Query syntax

To retrieve data from tables of an external YDB database, use the following SQL query form:

```
SELECT * FROM ydb_datasource.`<table_name>`
```

where:

- `ydb_datasource`: external data source identifier
- `<table_name>` is the full table name within the directory hierarchy in the YDB database, for example, `table`, `dir1/table1`, or `dir1/dir2/table3`.

If the table is at the top level of the hierarchy (does not belong to any directory), you may omit the backticks around the table name "`":

```
SELECT * FROM ydb_datasource.<table_name>
```

Limitations

When working with external YDB data sources, there are a number of limitations:

1. External sources are available only for reading data through `SELECT` queries. The federated query processing engine currently does not support queries that modify tables in external sources.
2. The YDB federated query processing system is capable of delegating the execution of certain parts of a query to the system acting as the data source. Query fragments are passed through YDB directly to the external system and processed by them. This optimization, known as "predicate pushdown", significantly reduces the amount of data transferred from the source to the federated query processing engine. This reduces network load and saves computational resources for the federated YDB.

   A specific case of predicate pushdown, when the filtering expressions are specified after the `WHERE` keyword, are passed to the data source, is called "filter pushdown". Filter pushdown is possible when using:

Description	Example	Limitation
Filters of the form `IS NULL` / `IS NOT NULL`	`WHERE column1 IS NULL` or `WHERE column1 IS NOT NULL`	
Logical conditions `OR`, `NOT`, `AND` and parentheses to control calculation priority.	`WHERE column1 IS NULL OR (column2 IS NOT NULL AND column3 > 10)`.	

Comparison operators with other columns or constants.	`WHERE column1 > column2 OR column3 <= 10`.	
Pattern matching operator `LIKE`.	`WHERE column1 LIKE '_abc%'`	Currently, only simple patterns based on prefixes (`'abc_'`, `'abc%'`), suffixes (`'_abc'`, `'%abc'`), or substring search (`'_abc_'`, `'%abc%'`, `'_abc%'`, `'%abc_'`) are supported for pushdown. If you need to push down more complex patterns, it is recommended to use `REGEXP`.
String pattern matching operator `REGEXP`.	`WHERE column1 REGEXP '.*abc.*'`	

When using other types of filters, pushdown to the source is not performed: filtering of the external table rows will be performed on the federated YDB side, which means that YDB will perform a full scan of the external table at the time of query processing.

Supported data types for filter pushdown:

Data type YDB
`Bool`
`Int8`
`Uint8`
`Int16`
`Uint16`
`Int32`
`Uint32`
`Int64`
`Uint64`
`Float`
`Double`
`String`
`Utf8`

Supported data types

When working with tables located in an external database YDB, users have access to a limited set of data types. All other types, except those listed below, are not supported. In some cases, type conversion is performed, meaning that columns of a table from the external database YDB change their type after the table is read by the database YDB processing the federated query.

Data type of the external source YDB	Data type in the federated YDB
`Bool`	`Bool`
`Int8`	`Int8`
`Int16`	`Int16`
`Int32`	`Int32`
`Int64`	`Int64`
`Uint8`	`Uint8`
`Uint16`	`Uint16`
`Uint32`	`Uint32`
`Uint64`	`Uint64`
`Float`	`Float`
`Double`	`Double`
`String`	`String`

`Utf8`	`Utf8`
`Date`	`Date`
`Datetime`	`Datetime`
`Timestamp`	`Timestamp`
`Json`	`Json`
`JsonDocument`	`Json`

Working with S3 Buckets (Yandex Object Storage)

To work with S3, you need to set up a data storage connection. There is a DDL for configuring such connections. Next, let's look at the SQL syntax and the management of these settings.

There are two types of buckets in S3: public and private. To connect to a public bucket, use `AUTH_METHOD="NONE"`. To connect to a private bucket, use `AUTH_METHOD="AWS"`. A detailed description of `AWS` can be found here. `AUTH_METHOD="NONE"` means that no authentication is used. If `AUTH_METHOD="AWS"` is specified, several additional parameters are required:

- `AWS_ACCESS_KEY_ID_SECRET_PATH` — the secret where `AWS_ACCESS_KEY_ID` is stored.
- `AWS_SECRET_ACCESS_KEY_SECRET_PATH` — the secret where `AWS_SECRET_ACCESS_KEY` is stored.
- `AWS_REGION` – region from which reading is performed, for example, `ru-central-1`.

To set up a connection to a public bucket, execute the following SQL query. The query creates an external connection named `object_storage`, which points to a specific S3 bucket named `bucket`.

```
CREATE EXTERNAL DATA SOURCE object_storage WITH (
  SOURCE_TYPE="ObjectStorage",
  LOCATION="https://object_storage_domain/bucket/",
  AUTH_METHOD="NONE"
);
```

To set up a connection to a private bucket, you need to run a few SQL queries. First, create secrets containing `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`.

```
CREATE SECRET aws_access_id WITH (value='<id>');
CREATE SECRET aws_access_key WITH (value='<key>');
```

The next step is to create an external connection named `object_storage`, which points to a specific S3 bucket named `bucket` and uses `AUTH_METHOD="AWS"`. The parameters `AWS_ACCESS_KEY_ID_SECRET_PATH`, `AWS_SECRET_ACCESS_KEY_SECRET_PATH`, and `AWS_REGION` are filled in for `AWS`. The values of these parameters are described above.

```
CREATE EXTERNAL DATA SOURCE object_storage WITH (
  SOURCE_TYPE="ObjectStorage",
  LOCATION="https://object_storage_domain/bucket/",
  AUTH_METHOD="AWS",
  AWS_ACCESS_KEY_ID_SECRET_PATH="aws_access_id",
  AWS_SECRET_ACCESS_KEY_SECRET_PATH="aws_access_key",
  AWS_REGION="ru-central-1"
);
```

Using an External Connection to an S3 Bucket

When working with Yandex Object Storage using external data sources, it is convenient to perform prototyping and initial data connection setup.

An example query to read data:

```
SELECT
  *
FROM
 object_storage.`*.tsv`
WITH
(
  FORMAT = "tsv_with_names",
  SCHEMA =
  (
    ts Uint32,
    action Utf8
  )
);
```

The list of supported formats and data compression algorithms for reading data in S3 (Yandex Object Storage) is provided in the section Data formats and compression algorithms.

Data Model

In Yandex Object Storage, data is stored in files. To read data, you need to specify the data format in the files, compression, and lists of fields. This is done using the following SQL expression:

```
SELECT
  <expression>
FROM
  <object_storage_connection_name>.`<file_path>`
WITH(
  FORMAT = "<file_format>",
  COMPRESSION = "<compression>",
  SCHEMA = (<schema_definition>),
  <format_settings>)
WHERE
  <filter>;
```

Where:

- `object_storage_connection_name` — the name of the external data source leading to the S3 bucket (Yandex Object Storage).

- `file_path` — the path to the file or files inside the bucket. Wildcards `*` are supported; more details in the section.
- `file_format` — the data format in the files.
- `compression` — the compression format of the files.
- `schema_definition` — the schema definition of the data stored in the files.
- `format_settings` — optional format settings

Data Schema Description

The data schema description consists of a set of fields:

- Field name.
- Field type.
- Required data flag.

For example, the data schema below describes a schema field named `Year` of type `Int32` with the requirement that this field must be present in the data:

```
Year Int32 NOT NULL
```

If a data field is marked as required (`NOT NULL`) but is missing in the processed file, processing such a file will result in an error. If a field is marked as optional (`NULL`), no error will occur if the field is absent in the processed file, but the field will take the value `NULL`. The keyword `NULL` is optional in this context.

Schema Inference

YDB can determine the data schema of the files inside the bucket so that you do not have to specify these fields manually.

> **Note**
>
> Schema inference is available for all data formats except `raw` and `json_as_string`. For these formats you must describe the schema manually.

To enable schema inference, use the `WITH_INFER` parameter:

```
SELECT
  <expression>
FROM
  <object_storage_connection_name>.`<file_path>`
WITH(
  FORMAT = "<file_format>",
  COMPRESSION = "<compression>",
  WITH_INFER = "true")
WHERE
  <filter>;
```

Where:

- `object_storage_connection_name` — the name of the external data source leading to the S3 bucket (Yandex Object Storage).
- `file_path` — the path to the file or files inside the bucket. Wildcards `*` are supported. For more information, see Data Path Formats Specified in `file_path`.
- `file_format` — the data format in the files. All formats except `raw` and `json_as_string` are supported.
- `compression` — the compression format of the files.

As a result of executing such a query, the names and types of fields will be inferred.

Data Path Formats Specified in `file_path`

In YDB, the followingdata paths are supported:

Path Format	Description	Example

Path ends with a `/`	Path to a directory	The path `/a/` addresses all contents of the directory:
		`/a/b/c/d/1.txt`
		`/a/b/2.csv`

Path contains a wildcard character `*`	Any files nested in the path	The path `/a/*.csv` addresses files in directories:
		`/a/b/c/1.csv`
		`/a/2.csv`
		`/a/b/c/d/e/f/g/2.csv`

Path does not end with `/` and does not contain wildcard characters	Path to a single file	The path `/a/b.csv` addresses the specific file `/a/b.csv`

Format Settings

In YDB, the following format settings are supported:

Setting name	Description	Possible values
`file_pattern`	File name template	File name template string. Wildcards `*` are supported.
`data.interval.unit`	Unit for parsing `Interval` type	`MICROSECONDS`, `MILLISECONDS`, `SECONDS`, `MINUTES`, `HOURS`, `DAYS`, `WEEKS`

`data.datetime.format_name`	Predefined format in which `Datetime` data is stored	`POSIX`, `ISO`
`data.datetime.format`	Strftime-like template which defines how `Datetime` data is stored	Formatting string, for example: `%Y-%m-%dT%H-%M`
`date.timestamp.format_name`	Predefined format in which `Timestamp` data is stored	`POSIX`, `ISO`, `UNIX_TIME_SECONDS`, `UNIX_TIME_MILLISECONDS`, `UNIX_TIME_MICROSECONDS`
`data.timestamp.format`	Strftime-like template which defines how `Timestamp` data is stored	Formatting string, for example: `%Y-%m-%dT%H-%M-%S`
`data.date.format`	The format in which `Date` data is stored	Formatting string, for example: `%Y-%m-%d`
`csv_delimiter`	Delimeter for `csv_with_names` format	Any character (UTF-8)

You can only specify `file_pattern` setting if `file_path` is a path to a directory. Any conversion specifiers supported by `strftime` (C99) function can be used in formatting strings. In YDB, the following `Datetime` and `Timestamp` formats are supported:

Name	Description	Example
`POSIX`	String in `%Y-%m-%d %H:%M:%S` format	2001-03-26 16:10:00
`ISO`	Format, corresponding to the ISO 8601 standard	2001-03-26 16:10:00Z
`UNIX_TIME_SECONDS`	Number of seconds that have elapsed since the 1st of january 1970 (00:00:00 UTC)	985623000
`UNIX_TIME_MILLISECONDS`	Number of milliseconds that have elapsed since the 1st of january 1970 (00:00:00 UTC)	985623000000
`UNIX_TIME_MICROSECONDS`	Number of microseconds that have elapsed since the 1st of january 1970 (00:00:00 UTC)	985623000000000

Example

Example query to read data from S3 (Yandex Object Storage):

```
SELECT
  *
FROM
  connection.`folder/`
WITH(
  FORMAT = "csv_with_names",
  COMPRESSION="gzip"
  SCHEMA =
  (
    Id Int32 NOT NULL,
    UserId Int32 NOT NULL,
    TripDate Date NOT NULL,
    TripDistance Double NOT NULL,
    UserComment Utf8
  ),
  FILE_PATTERN="*.csv.gz",
  `DATA.DATE.FORMAT`="%Y-%m-%d",
  CSV_DELIMITER='/'
);
```

Where:

- `connection` — the name of the external data source leading to the S3 bucket (Yandex Object Storage).
- `folder/filename.csv` — the path to the directory in the S3 bucket (Yandex Object Storage).
- `SCHEMA` — the data schema description in the file.
- `*.csv.gz` — file name template.
- `%Y-%m-%d` — format in which `Date` type is stored in S3.

Reading Data from an External Table Pointing to S3 (Object Storage)

Sometimes, the same data queries need to be executed regularly. To avoid specifying all the details of working with this data every time a query is called, use the mode with external tables. In this case, the query looks like a regular query to YDB tables.

Example query for reading data:

```
SELECT
    *
FROM
    `s3_test_data`
WHERE
    version > 1
```

Creating an External Table Pointing to an S3 Bucket (Object Storage)

To create an external table describing the S3 bucket (Object Storage), execute the following SQL query. The query creates an external table named `s3_test_data`, containing files in the `CSV` format with string fields `key` and `value`, located inside the bucket at the path `test_folder`, using the connection credentials specified by the external data source object `bucket`:

```
CREATE EXTERNAL TABLE `s3_test_data` (
  key Utf8 NOT NULL,
  value Utf8 NOT NULL
) WITH (
  DATA_SOURCE="bucket",
  LOCATION="folder",
  FORMAT="csv_with_names",
  COMPRESSION="gzip"
);
```

Where:

- `key, value` - list of data columns and their types;
- `bucket` - name of the external data source to S3 (Object Storage);
- `folder` - path within the bucket containing the data;
- `csv_with_names` - one of the permitted data storage formats;
- `gzip` - one of the permitted compression algorithms.

You can also specify format settings.

Data Model

Reading data using external tables from S3 (Object Storage) is done with regular SQL queries as if querying a normal table.

```
SELECT
  <expression>
FROM
  `s3_test_data`
WHERE
  <filter>;
```

Limitations

There are a number of limitations when working with S3 buckets (Object Storage).

Limitations:

1. Only data read requests - `SELECT` and `INSERT` are supported; other requests are not.
2. If the date value stored in the external data source is outside the allowed range for YDB (all dates used must be later than 1970-01-01 but earlier than 2105-12-31), such a value in YDB will be converted to `NULL`.

Writing Data to S3 Buckets (Yandex Object Storage)

In YDB, you can use external connections or external tables to write data to the Yandex Object Storage bucket.

Writing Data via External Connection

Using connections for data writing is convenient for prototyping and initial setup. The SQL expression demonstrates writing data directly to an external data source.

```
INSERT INTO `connection`.`test/`
WITH
(
  FORMAT = "csv_with_names"
)
SELECT
    "value" AS value, "name" AS name
```

The data will be written to the specified path. In this mode, the resulting files will not be partitioned. If you need to partition the resulting files, use writing via external tables. The files created during the writing process are assigned random names.

When working with external connections, only read (`SELECT`) and insert (`INSERT`) operations are possible; other types of operations are not supported.

Writing Data via External Tables

If you need to write data regularly, doing this using external tables is convenient. In this case, there is no need to specify all the details of working with this data in each query. To write data to the bucket, create an external table in S3 (Yandex Object Storage) and use the usual SQL `INSERT INTO` statement:

```
INSERT INTO `test`
SELECT
    "value" AS value, "name" AS name
```

Exporting data to S3 object storage

YDB supports exporting table data to S3 using an external data source.

To export, use a write query via external data source and specify the format of the exported files:

```
INSERT INTO external_source.`test/`
WITH
(
    FORMAT = "parquet"
)
SELECT
    *
FROM table
```

You can export both regular YDB tables and any external tables.

> **Tip**
>
> The recommended export format is `parquet` ; import and export paths are optimized for it.

To export all tables under a given directory in the data schema, plus metadata about tables, directories, and table indexes, to S3-compatible storage, use the YDB CLI. For details, see Export to S3-compatible storage.

Data formats and compression algorithms

This section describes the data formats supported in YDB that are stored in S3, the supported compression algorithms, and the list of supported YQL types for each data format.

Supported data formats

The list of data formats supported in YDB is provided in the table below.

Format	Reading	Writing
`csv`	✓	
`csv_with_names`	✓	✓
`tsv_with_names`	✓	✓
`json_list`	✓	✓
`json_each_row`	✓	✓
`json_as_string`	✓	
`parquet`	✓	✓
`raw`	✓	✓

CSV format

This format is based on the CSV format and differs from `csv_with_names` in that it does not have a header with column names. All lines in the file are treated as data, and column names are taken from the schema of the query in the order they are declared in `SCHEMA`.

> **Note**
>
> The `csv` format is available only for reading data from S3. Writing in this format is not supported.

Example of data (without a header row):

```
1997,Man_1,Model_1,3000.00
1999,Man_2,Model_2,4900.00
```

Example query

```
SELECT
    *
FROM external_source.path
WITH
(
    FORMAT = "csv",
    SCHEMA =
    (
        Year Int32,
        Manufacturer Utf8,
        Model Utf8,
        Price Double
    )
)
```

The result of the query execution:

#	Manufacturer	Model	Price	Year
1	Man_1	Model_1	3000	1997
2	Man_2	Model_2	4900	1999

Csv_with_names format

This format is based on the CSV format. The data is arranged in columns, separated by commas, and the first line of the file contains the column names. Unlike the `csv` format, where column names are taken from the query schema and matched positionally, in the `csv_with_names` format, column names are read directly from the header line of the file and matched to the schema by name.

Example data:

```
Year,Manufacturer,Model,Price
1997,Man_1,Model_1,3000.00
1999,Man_2,Model_2,4900.00
```

Example query

```
SELECT
    *
FROM external_source.path
```

```
WITH
(
    FORMAT = "csv_with_names",
    SCHEMA =
    (
        Year Int32,
        Manufacturer Utf8,
        Model Utf8,
        Price Double
    )
)
```

The result of the query execution:

#	Manufacturer	Model	Price	Year
1	Man_1	Model_1	3000	1997
2	Man_2	Model_2	4900	1999

tsv_with_names format

This format is based on the TSV format. The data is arranged in columns and separated by tab characters (code `0x9`). The first line of the file contains the column names.

Example of data:

```
Year    Manufacturer    Model   Price
1997    Man_1   Model_1    3000.00
1999    Man_2   Model_2    4900.00
```

Example query

```
SELECT
    *
FROM external_source.path
WITH
(
    FORMAT = "tsv_with_names",
    SCHEMA =
    (
        Year Int32,
        Manufacturer Utf8,
        Model Utf8,
        Price Double
    )
)
```

The result of the query execution:

#	Manufacturer	Model	Price	Year
1	Man_1	Model_1	3000	1997
2	Man_2	Model_2	4900	1999

json_list format

This format is based on JSON representation of data. In this format, each file must contain an object in a valid JSON representation.

Example of valid data (data is presented as a list of JSON objects):

```
[
    { "Year": 1997, "Manufacturer": "Man_1", "Model": "Model_1", "Price": 3000.0 },
    { "Year": 1999, "Manufacturer": "Man_2", "Model": "Model_2", "Price": 4900.00 }
]
```

Example of **IN**correct data (each separate line contains a separate JSON object, but these objects are not combined into a list):

```
{ "Year": 1997, "Manufacturer": "Man_1", "Model": "Model_1", "Price": 3000.0 }
{ "Year": 1999, "Manufacturer": "Man_2", "Model": "Model_2", "Price": 4900.00 }
```

Example query

```
SELECT
    *
FROM external_source.path
WITH
(
    FORMAT = "json_list",
    SCHEMA =
    (
        Year Int32,
        Manufacturer Utf8,
```

```
        Model Utf8,
        Price Double
    )
)
```

Result of query execution:

#	Manufacturer	Model	Price	Year
1	Man_1	Model_1	3000	1997
2	Man_2	Model_2	4900	1999

json_each_row format

This format is based on JSON representation of data. In this format, each line of the file must contain a separate object in a valid JSON representation, but these objects are not combined into a JSON list. This format is used when transferring data through streaming systems, such as Apache Kafka or YDB topics (../../../datamodel/topic.md).

Example of correct data (each line contains a separate JSON object, but these objects are not combined into a list):

```
{ "Year": 1997, "Manufacturer": "Man_1", "Model": "Model_1", "Price": 3000.0 },
{ "Year": 1999, "Manufacturer": "Man_2", "Model": "Model_2", "Price": 4900.00 }
```

Example query

```
SELECT
    *
FROM external_source.path
WITH
(
    FORMAT = "json_each_row",
    SCHEMA =
    (
        Year Int32,
        Manufacturer Utf8,
        Model Utf8,
        Price Double
    )
)
```

The result of the query execution:

#	Manufacturer	Model	Price	Year
1	Man_1	Model_1	3000	1997
2	Man_2	Model_2	4900	1999

json_as_string format

This format is based on JSON representation. The `json_as_string` format does not split the input JSON document into fields, but represents each line of the file as a single JSON object (or a single string). This format is convenient if the list of fields is not the same in all lines and can vary.

In this format, each file should contain:

- a JSON object in a valid JSON representation on each separate line of the file;
- JSON objects in a valid JSON representation, combined into a list.

Example of valid data (data is presented as a list of JSON objects):

```
{ "Year": 1997, "Attrs": { "Manufacturer": "Man_1", "Model": "Model_1" }, "Price": 3000.0 }
{ "Year": 1999, "Attrs": { "Manufacturer": "Man_2", "Model": "Model_2" }, "Price": 4900.00 }
```

In this format, the schema of the data being read must consist of only one column with one of the allowed data types; see below for details.

Example query

```
SELECT
    CAST(JSON_VALUE(Data, "$.Year") AS Int32) AS Year,
    JSON_VALUE(Data, "$.Attrs.Manufacturer") AS Manufacturer,
    JSON_VALUE(Data, "$.Attrs.Model") AS Model,
    CAST(JSON_VALUE(Data, "$.Price") AS Double) AS Price
FROM external_source.path
WITH
(
    FORMAT = "json_as_string",
    SCHEMA =
    (
        Data Json
    )
)
```

The result of the query execution:

#	Manufacturer	Model	Price	Year
1	Man_1	Model_1	3000	1997
2	Man_2	Model_2	4900	1999

Parquet format

This format allows reading the contents of files in the Apache Parquet format.

Supported data compression algorithms inside Parquet files for reading from S3:

- No compression
- SNAPPY
- GZIP
- BROTLI
- LZ4
- ZSTD
- LZ4_RAW

> **Note**
>
> Parquet format records will be written using the Snappy compression algorithm.

Example query

```
SELECT
    *
FROM external_source.path
WITH
(
    FORMAT = "parquet",
    SCHEMA =
    (
        Year Int32,
        Manufacturer Utf8,
        Model Utf8,
        Price Double
    )
)
```

The result of the query execution:

#	Manufacturer	Model	Price	Year
1	Man_1	Model_1	3000	1997
2	Man_2	Model_2	4900	1999

Raw format

This format allows you to read the contents of files as is, in "raw" form. Data read in this way can be processed using YQL by dividing it into rows and columns.

> **Note**
>
> The size of each of the files read in `raw` format cannot exceed the overall memory consumption limit for a single query in YDB.

Example query

To parse the following data, where rows are separated by a semicolon and values by a comma:

```
1997,Man_1,Model_1,3000.00;
1999,Man_2,Model_2,4900.00;
```

You can use the following query:

```
$input = SELECT
    String::SplitToList(        -- splitting each row by ','
        String::Strip(RowData), -- removing all whitespace characters from the beginning and end of the r
ows
        ","
    ) AS Row
FROM external_source.path
WITH
(
    FORMAT = "raw",
    SCHEMA =
```

```
    (
        FileData Utf8
    )
)
FLATTEN LIST BY (String::SplitToList(FileData, ";", TRUE AS SkipEmpty) AS RowData); -- splitting the file
by ';'

SELECT -- Getting the required columns
    CAST(Row[0] AS Int32) AS Year,
    Row[1] AS Manufacturer,
    Row[2] AS Model,
    CAST(Row[3] AS Double) AS Price
FROM $input
```

The result of the query execution:

#	Manufacturer	Model	Price	Year
1	Man_1	Model_1	3000	1997
2	Man_2	Model_2	4900	1999

Supported compression algorithms

The use of compression algorithms depends on the file formats. For all file formats except Parquet, the following compression algorithms can be used:

Algorithm	Name in YDB	Reading	Writing
Gzip	gzip	✓	✓
Zstd	zstd	✓	✓
LZ4	lz4	✓	✓
Brotli	brotli	✓	✓
Bzip2	bzip2	✓	✓
Xz	xz	✓	✓

For Parquet file format, native internal compression algorithms are supported:

Compression format	Read	Write
None	✓	
Snappy	✓	✓
Gzip	✓	
Brotli	✓	
LZ4	✓	
Zstd	✓	
LZ4_RAW	✓	

In YDB, working with externally compressed Parquet files, such as files of the form `<myfile>.parquet.gz` or similar, is not supported. All Parquet files must be without external compression.

Supported data types

A table of all supported types when reading from S3 in the query schema:

Type	csv	csv_with_names	tsv_with_names	json_list	json_each_row	json_as_string	parquet	raw
`Bool`, `Int8`, `Int16`, `Int32`, `Int64`, `Uint8`, `Uint16`, `Uint32`, `Uint64`, `Float`, `Double`	✓	✓	✓	✓	✓		✓	
`DyNumber`				✓				
`String`, `Utf8`, `Json`	✓	✓	✓	✓	✓	✓	✓	✓
`JsonDocument`							✓	
`Yson`				✓			✓	✓
`Uuid`	✓	✓	✓		✓			

`Date`, `Datetime`, `Timestamp`, `TzDate`, `TzDateTime`, `TzTimestamp`	✓	✓	✓		✓		✓	
`Interval`	✓	✓	✓		✓		✓	
`Date32`, `Datetime64`, `Timestamp64`, `Interval64`, `TzDate32`, `TzDateTime64`, `TzTimestamp64`							✓	
`Optional<T>`	✓	✓	✓	✓	✓	✓	✓	✓

The table of all supported types for writing to S3:

Type	csv_with_names	tsv_with_names	json_list	json_each_row	parquet	raw
`Bool`, `Int8`, `Int16`, `Int32`, `Int64`, `Uint8`, `Uint16`, `Uint32`, `Uint64`, `Float`, `Double`	✓	✓	✓	✓	✓	
`DyNumber`			✓			
`String`, `Utf8`, `Json`	✓	✓	✓	✓	✓	✓
`JsonDocument`						
`Yson`			✓			✓
`Uuid`	✓	✓		✓		
`Date`, `Datetime`, `Timestamp`, `TzDate`, `TzDateTime`, `TzTimestamp`	✓	✓		✓	✓	
`Interval`						
`Date32`, `Datetime64`, `Timestamp64`, `Interval64`, `TzDate32`, `TzDateTime64`, `TzTimestamp64`						
`Optional<T>`	✓	✓	✓	✓	✓	

For all S3 read and write formats, except `json_list`, the `Optional<T>` type can be used only if `T` is a primitive YQL type. For more information about optional types, see the article column0.

Mapping types when reading and writing Parquet data

When reading and writing data in Parquet format, YDB uses the Apache Arrow logical type system — the standard Parquet uses to describe data semantics. The table below shows how YQL types map to Arrow logical types written in Parquet files.

YQL type	Arrow type on export	Arrow type on import	Notes
`Bool`	`UINT8`	`BOOL`, `UINT8`	
`Int8`	`INT8`	`INT8`	
`Int16`	`INT16`	`INT16`	
`Int32`	`INT32`	`INT32`	
`Int64`	`INT64`	`INT64`	
`Uint8`	`UINT8`	`UINT8`	
`Uint16`	`UINT16`	`UINT16`	
`Uint32`	`UINT32`	`UINT32`	
`Uint64`	`UINT64`	`UINT64`	
`Float`	`FLOAT (32)`	`FLOAT (32)`	
`Double`	`FLOAT (64)`	`FLOAT (64)`	
`Date`	`UINT16`	`UINT16`, `INT32`, `UINT32`, `INT64`, `UINT64`, `DATE`, `TIMESTAMP (s, ms, us)`	number of days since the Unix epoch
`Date32`	`DATE (s)`	`DATE (s)`	
`Datetime`	`UINT32`	`UINT16`, `INT32`, `UINT32`, `INT64`, `UINT64`, `DATE`, `TIMESTAMP (s, ms, us)`	number of seconds since the Unix epoch
`Datetime64`	`TIMESTAMP (us)`	`TIMESTAMP (s, ms, us)`	
`Timestamp`	`TIMESTAMP (us)`	`TIMESTAMP (s, ms, us)`	
`Timestamp64`	`TIMESTAMP (us)`	`TIMESTAMP (s, ms, us)`	
`Decimal(s, p)`	`DECIMAL (128, s, p)`	`DECIMAL (128, s, p)`	
`String`	`BINARY`	`BINARY`	
`Utf8`	`BINARY`	`BINARY`	
`Json`	`BINARY`	`BINARY`	

Data Partitioning in S3 (Yandex Object Storage)

In S3 (Yandex Object Storage), it is possible to store very large volumes of data. At the same time, queries to this data may not need to touch all the data but only a part of it. If you describe the rules for marking the storage structure of your data in YDB, then data that is not needed for the query can even be skipped from being read from S3 (Yandex Object Storage). This mechanism significantly speeds up query execution without affecting the result.

For example, data is stored in the following directory structure:

```
year=2021
    month=01
    month=02
    month=03
year=2022
    month=01
```

The query below explicitly implies that only the data for February 2021 needs to be processed, and other data is not needed.

```
SELECT
    *
FROM
    objectstorage.'/'
WITH
(
    SCHEMA =
    (
        data String,
        year Int32,
        month Int32
    )
)
WHERE
    year=2021
    AND month=02
```

If the data partitioning scheme is not specified, then *all* stored data will be read from S3 (Yandex Object Storage), but as a result of processing, data for all other dates will be discarded.

If you explicitly describe the storage structure, specifying that the data in S3 (Yandex Object Storage) is placed in directories by years and months

```
SELECT
    *
FROM
    objectstorage.'/'
WITH
(
    SCHEMA =
    (
        data String,
        year Int32,
        month Int32
    ),
    PARTITIONED_BY = "['year', 'month']"
)
WHERE
    year=2021
    AND month=02
```

then during the query execution, not all data will be read from S3 (Yandex Object Storage), but only the data for February 2021. This will significantly reduce the volume of data processed and speed up processing, while the results of both queries will be identical.

> **Note**
>
> The example above shows working with data at the level of connections. This example is chosen for illustrative purposes only. We strongly recommend using "data bindings" to work with data and not using direct work with connections.

Syntax

When working at the connection level, partitioning is set using the `partitioned_by` parameter, where the list of columns is specified in JSON format.

```
SELECT
    *
FROM
    <connection>.<path>
WITH
(
    SCHEMA=(<field1>, <field2>, <field3>),
    PARTITIONED_BY="['field2', 'field3']"
)
```

In the `partitioned_by` parameter, the columns of the data schema by which the data stored in S3 (Yandex Object Storage) are partitioned are listed. The order of specifying fields in the `partitioned_by` parameter determines the nesting of S3 (Yandex Object Storage) directories within each other.

For example, `PARTITIONED_BY=['year', 'month']` defines the directory structure

```
year=2021
    month=01
    month=02
    month=03
year=2022
    month=01
```

And `partitioned_by=['month', 'year']` defines another directory structure

```
month=01
    year=2021
    year=2022
month=02
    year=2021
month=03
    year=2021
```

Supported Data Types

Partitioning is possible only with the following set of YQL data types:

- Uint16, Uint32, Uint64
- Int16, Int32, Int64
- String, Utf8

When using other types for specifying partitioning, an error is returned.

Supported Storage Path Formats

The storage path format, where the name of each directory explicitly specifies the column name, is called the "Hive-Metastore format" or simply the "Hive format."

This format looks as follows:

```
month=01
    year=2021
    year=2022
month=02
    year=2021
month=03
    year=2021
```

Warning

The basic partitioning mode in YDB supports only the Hive format.

Use the Extended Data Partitioning mode to specify arbitrary storage paths.

Extended Partitioning in S3 (Yandex Object Storage)

Partitioning allows you to specify rules for YDB data placement in S3 (Yandex Object Storage).

Assume the data in S3 (Yandex Object Storage) is stored in the following directory structure:

```
year=2021
    month=01
    month=02
    month=03
year=2022
    month=01
```

When executing the query below, YDB will perform the following actions:

1. Retrieve a full list of subdirectories within '/'.
2. Attempt to process the name of each subdirectory in the format `year=<DIGITS>`.
3. For each subdirectory `year=<DIGITS>`, retrieve a list of all subdirectories in the format `month=<DIGITS>`.
4. Process the read data.

```
SELECT
    *
FROM
    objectstorage.'/'
WITH
(
    SCHEMA =
    (
        data String,
        year Int32,
        month Int32
    ),
    PARTITIONED_BY = "['year', 'month']"
)
WHERE
    year=2021
    AND month=02
```

When working with partitioned data, a complete listing of the contents of S3 (Yandex Object Storage) is performed, which can take a considerable amount of time on large buckets.

To optimize performance on large data volumes, use "advanced partitioning". In this mode, S3 (Yandex Object Storage) directories are not scanned; instead, all paths are calculated in advance, and access is made only to these paths.

To enable advanced partitioning, specify the working rules through a special parameter - "projection". This parameter describes all the rules for data placement in the S3 (Yandex Object Storage) directories.

Syntax

Advanced partitioning is called "partition projection" and is specified through the `projection` parameter.

Example of specifying advanced partitioning:

```
SELECT
    *
FROM
    <connection>.`/`
WITH
(
    SCHEMA =
    (
        data String,
        year Int32,
        month Int32
    ),
    PARTITIONED_BY = "['year', 'month']",
    `projection.enabled` : "true",

    `projection.year.type` : "integer",
    `projection.year.min` : "2010",
    `projection.year.max` : "2022",
    `projection.year.interval` : "1",

    `projection.month.type` : "integer",
    `projection.month.min` : "1",
    `projection.month.max` : "12",
    `projection.month.interval` : "1",
    `projection.month.digits` : "2",

    `storage.location.template` : "${year}/${month}"
)
```

The example above specifies that data exists for each year and each month from 2010 to 2022, with data placed in directories like `2022/12` within the bucket. If data for a certain period is absent within the bucket, this does not cause errors; the query will execute successfully, and the data will be skipped in the calculations.

In general, the advanced partitioning setup looks as follows:

```
SELECT
    *
FROM
    <connection>.<path>
WITH
(
    SCHEMA = (<fields>, <field1>, <field2>),
    PARTITIONED_BY = "'['field1', 'field2']",
    `projection.enabled` : <"true"|"false">,

    `projection.<field1_name>.type` : "<type>",
    `projection.<field1_name>....` : "<extended_properties>",

    `projection.<field2_name>.type` : "<type>",
    `projection.<field2_name>....` : "<extended_properties>",

    `storage.location.template` : ".../${field2}/${field1}/..."
)
```

Field Descriptions

Field name	Description	Allowed values
`projection.enabled`	Whether advanced partitioning is enabled or not	`true`, `false`
`projection.<field1_name>.type`	Data type of the field	`integer`, `enum`, `date`
`projection.<field1_name>.XXX`	Specific properties of the type	

Integer Field Type

It is used for columns whose values can be represented as integers ranging from 2^{-63} to 2^{63}-1.

Field name	Mandatory	Description	Example values
`projection.<field_name>.type`	Yes	Data type of the field	integer
`projection.<field_name>.min`	Yes	Specifies the minimum allowable value as an integer	-100 004
`projection.<field_name>.max`	Yes	Specifies the maximum allowable value as an integer	-10 5000
`projection.<field_name>.interval`	No, default is `1`	Specifies the step between elements within the value range. For example, a step of 3 within the range 2 to 10 will result in the values: 2, 5, 8	2 11
`projection.<field_name>.digits`	No, default is `0`	Specifies the number of digits in the number. If the number of significant digits in the number is less than the specified value, the value is padded with leading zeros up to the specified number of digits. For example, if .digits=3 is specified and the number 2 is passed, it will be converted to 002	2 4

Enum Field Type

It is used for columns whose values can be represented as a set of enumerated values.

Field name	Mandatory	Description	Example values
`projection.<field_name>.type`	Yes	Data type of the field	enum
`projection.<field_name>.values`	Yes	Specifies the allowable values, separated by commas. Spaces are not ignored	1, 2 A,B,C

Date Field Type

It is used for columns whose values can be represented as dates. The allowable date range is from 1970-01-01 to 2105-01-01.

Field name	Mandatory	Description	Example values
`projection.<field_name>.type`	Yes	Data type of the field	date
`projection.<field_name>.min`	Yes	Specifies the minimum allowable date. Allowed values in the format `YYYY-MM-DD` or as an expression containing the special macro substitution NOW	
`projection.<field_name>.max`	Yes	Specifies the maximum allowable date. Allowed values in the format `YYYY-MM-DD` or as an expression containing the special macro substitution NOW	2020-01-01 NOW-5DAYS NOW+3HOURS
`projection.<field_name>.format`	Yes	Date formatting string based on strptime	%Y-%m-%d %D
`projection.<field_name>.unit`	No, default is `DAYS`	Time interval units. Allowed values: `YEARS`, `MONTHS`, `WEEKS`, `DAYS`, `HOURS`, `MINUTES`, `SECONDS`, `MILLISECONDS`	SECONDS YEARS

`projection.<field_name>.interval`	No, default is `1`	Specifies the step between elements within the value range with the specified dimension in `projection.<field_name>.unit`. For example, for the range 2021-02-02 to 2021-03-05 with a step of 15 and the dimension DAYS, the values will be: 2021-02-17, 2021-03-04	2 6

Working with the NOW Macro Substitution

1. A number of arithmetic operations with the NOW macro substitution are supported: adding and subtracting time intervals. For example: `NOW-3DAYS`, `NOW+1MONTH`, `NOW-6YEARS`, `NOW+4HOURS`, `NOW-5MINUTES`, `NOW+6SECONDS`. The possible usage options for the macro substitution are described by the regular expression: `^\s*(NOW)\s*(([\+\-])\s*([0-9]+)\s*(YEARS?|MONTHS?|WEEKS?|DAYS?|HOURS?|MINUTES?|SECONDS?)\s*)?$`
2. Allowed interval dimensions: YEARS, MONTHS, WEEKS, DAYS, HOURS, MINUTES, SECONDS, MILLISECONDS.
3. Only one arithmetic operation is allowed in expressions; expressions like `NOW-5MINUTES+6SECONDS` are not supported.
4. Working with intervals always results in obtaining a valid date, but depending on the dimension, the final results may vary:
   - Adding `MONTHS` to a date adds a calendar month, not a fixed number of days. For example, if the current date is `2023-01-31`, adding `1 MONTHS` will result in the date `2023-02-28`.
   - Adding `30 DAYS` to a date adds a fixed number of days. For example, if the current date is `2023-01-31`, adding `30 DAYS` will result in the date `2023-03-02`.
   - The earliest possible date is `1970-01-01` (time 0 in Unix time). If the result of calculations is a date earlier than the minimum, the entire query fails with an error.
   - The latest possible date is `2105-12-31` (the maximum date in Unix time). If the result of calculations is a date later than the maximum, the entire query fails with an error.

Path Templates

Data in S3 (Yandex Object Storage) buckets can be placed in directories with arbitrary names. The `storage.location.template` setting allows you to specify the naming rules for the directories where the data is stored.

Field name	Description	Example values
`storage.location.template`	Path template for directory names. The path is specified as a text string with parameter macro substitutions `...${<field_name>}...${<field_name>}...`	`root/a/${year}/b/${month}/d${year}/${month}`

If the path contains the characters `$`, `\`, or the characters `{}`, they must be escaped with the `\` character. For example, to work with a directory named `my$folder`, it needs to be specified as `my\$folder`.

Import and export data using federated queries

> **Note**
>
> When importing/exporting data from/to S3 in Parquet format, you must consider the mapping of YQL and Apache Arrow types.

Data import

Federated queries allow you to import data from connected external sources into YDB database tables. To import data, you need to use a read query from an external data source or external table and write to a YDB table.

For columnar tables, massively parallel import from an external source is supported when using UPSERT and INSERT statements — multiple worker threads read data from the external source in parallel and write to the table. For row-based tables, this functionality is under development.

Operation	Write to row-based tables	Write to columnar tables
UPSERT	single-threaded	parallel
REPLACE	single-threaded	parallel
INSERT	single-threaded	parallel

> **Tip**
>
> The recommended import options using federated queries are the `UPSERT` and `REPLACE` operations — the data import scenario is significantly optimized for them.

Example of importing data from a PostgreSQL table into a YDB table:

```
UPSERT INTO target_table
SELECT * FROM postgresql_datasource.source_table
```

For more details on creating external data sources and external tables, as well as reading data from them, see the articles:

- S3

Experimental data sources

- ClickHouse
- Greenplum
- Microsoft SQL Server
- MySQL
- PostgreSQL
- YDB (federated queries between YDB databases)

Data export

Currently, data export using federated queries is supported only for S3-compatible storage; for more details, see the article Exporting data to S3 object storage.

Spilling

Spilling in general

Spilling is a memory management mechanism in which intermediate data generated during query execution that exceeds the available RAM of a node is temporarily offloaded to external storage. In YDB, disk is currently used for spilling. Spilling ensures the execution of user queries that require processing large volumes of data exceeding the available node memory.

In data processing systems, including YDB, spilling is important for:

- Processing queries with large volumes of data when intermediate results do not fit into memory.
- Executing complex analytical operations (aggregations, table joins) on large datasets
- Query performance optimization by intermediate materialization of some data in external memory, which in certain scenarios can speed up the overall execution time.

Spilling operates based on the principle of memory hierarchy:

1. **RAM (Random Access Memory)** is a fast but limited resource.
2. **External memory** — slower, but more capacious.

When memory usage approaches the limit, the system:

- Serializes part of the data.
- Saves them in external memory.
- frees the corresponding memory
- continues processing the query, using data in memory.
- if necessary, loads data back into memory to continue computations.

Spilling in YDB

YDB implements the spilling mechanism via **Spilling Service** — an actor service that provides temporary storage for data blobs. Spilling is performed only on database nodes. Detailed technical information about Spilling Service is available in the Spilling Service section.

Types of spilling in YDB

YDB implements two main types of spilling, operating at different levels of the computing process:

- Spilling in computations
- Spilling in transport

These types operate independently of each other and can be activated simultaneously within a single query, providing comprehensive memory management.

Spilling in computations

Compute cores YDB automatically spill intermediate data to disk when executing operations that require significant memory. This type of spilling is implemented at the level of individual compute operations and is activated when memory limits are reached.

Main use cases:

- **Aggregations** — when grouping large volumes of data, the system offloads intermediate hash tables to disk
- **Join operations** — when joining large tables, the Grace Hash Join algorithm partitions data and spills it to disk.

How it works

Compute nodes contain specialized objects for monitoring memory usage. When the data volume approaches the set limit:

1. The system switches to spilling mode
2. The data is serialized and split into blocks (buckets)
3. Some blocks are transferred to the Spilling Service for saving to disk
4. Metadata about data location is stored in memory
5. The system continues processing the data remaining in memory, which allows freeing up additional space
6. If necessary, the data is loaded back and processed

Spilling in transport

This type of spilling operates at the data transfer level between different query execution stages. The data transfer subsystem automatically buffers and spills data when buffers overflow. This helps avoid blocking operations that generate data, even in situations where the receiving side is temporarily unable to receive data.

How it works

The data transfer subsystem constantly monitors its state:

1. **Buffering**: Incoming data accumulates in the internal buffers of the subsystem
2. **Fill control**: The system monitors the fill level of buffers
3. **Automatic spilling**: When limits are reached, data is automatically serialized and transferred to the Spilling Service
4. **Continued operation**: The subsystem continues to accept new data after freeing up memory space
5. **Recovery**: When the next stage is ready, data is read from external storage and passed on.

Interaction with the memory controller

When executing queries, YDB tries to stay within the specified memory limit set by the memory controller. To continue fitting within this limit even as intermediate computations grow, spilling is used.

At the Resource Manager level, the threshold for transitioning to spilling is set by the `resource_manager.spilling_percent` parameter: when the Query Processor memory pool on a node is filled to the specified percentage, compute operations start preferring offloading data to disk over further growth of RAM consumption. The pool size is set by the `query_execution_limit_percent` / `query_execution_limit_bytes` parameters.

For more details on spilling settings and related memory limits, see the Memory management and Query execution memory limits sections.

See also

- Spilling Service
- Spilling configuration
- Monitoring YDB
- Performance diagnostics

Scan queries in YDB

Warning

Using `ScanQuery` is not recommended for new scenarios. Use standard mechanisms to execute queries.

For more details, see the Executing queries section.

The exception is executing long-running (over 5 minutes) queries against row tables. In this case, we still recommend using `ScanQuery`, as standard query execution mechanisms do not yet fully support this scenario.

Scan queries are a separate data access interface primarily intended for running analytical ad-hoc queries against a database.

This execution mode has the following distinctive properties:

- It is *read-only*.
- In *SERIALIZABLE_RW* mode, a data snapshot is taken, and all further work happens on top of that snapshot. As a result, the impact on the OLTP transaction stream is minimal (only snapshot acquisition).
- The query result is a data stream (gRPC stream). Therefore, scan queries do not have a limit on the number of rows in the result.
- Due to high overhead, it is suitable only for ad-hoc queries.

Note

You can query system views through the *Scan Queries* interface.

Scan queries are not considered a full-fledged way to run OLAP workloads because they currently have a number of technical limitations (which will be lifted over time):

- Query duration is limited to 10 minutes.
- Many operations (including sorting) are executed entirely in memory, so complex queries may fail with an out-of-resources error.
- For joins, only one strategy is currently supported — *MapJoin* (aka *Broadcast Join*), where the "right" table is converted into a map, so it must be no more than a few gigabytes in size.
- Prepared statements are not supported, i.e. the query is compiled on every call.
- There are no optimizations for point reads or small-range reads.
- Automatic retries are not supported in SDKs.

For OLAP workloads, YDB provides a specialized table type — column-oriented tables. They store each column's data separately from other columns, so only the columns directly involved in the query are read during execution.

Note

Even though *Scan Queries* do not directly interfere with OLTP transactions, they still consume shared database resources: CPU, memory, disk, and network. Therefore, running heavy queries **may cause resource starvation**, which will affect the performance of the entire database.

How to use

Like other query types, *Scan Queries* are available through CLI and SDK.

C++ SDK

To run a query via *Scan Queries*, `Ydb::TTableClient` provides two methods:

```
class TTableClient {
 ...
 TAsyncScanQueryPartIterator StreamExecuteScanQuery(const TString& query,
 const TStreamExecScanQuerySettings& settings = TStreamExecScanQuerySettings());

 TAsyncScanQueryPartIterator StreamExecuteScanQuery(const TString& query, const TParams& params,
 const TStreamExecScanQuerySettings& settings = TStreamExecScanQuerySettings());
 ...
};
```

YQL queries to topics

For reading and writing messages to topics, the familiar YQL constructs are used: SELECT for reading and INSERT for writing.

Local and external topics

YQL queries to topics work the same regardless of whether the topic is in the current database or in another database YDB. The source and destination of messages can be either a topic **in the same database** where the query is executed, or a topic **in another database**.

Local topics

Local topics are topics created in the **same database** YDB as the query being executed.

In the query text, they are referred to **by a short name** — just like a table in the current database:

```
SELECT * FROM input_topic WITH (FORMAT = json_each_row, SCHEMA = (...));
```

```
INSERT INTO output_topic SELECT ...;
```

External topics

External topics are topics located **in another database** YDB.

Access to them is performed only through a pre-created external data source with the YDB source type.

After creating a source, for example named `ext_source`, access to topic `input_topic` in the external database is written as follows:

```
SELECT * FROM ext_source.input_topic WITH (FORMAT = json_each_row, SCHEMA = (...));
```

The name `ext_source` in the documentation is **conventional** — in your database, the source may be named differently; it is important that it matches in `CREATE EXTERNAL DATA SOURCE` and in the prefix before the topic name.

Reading from a topic

Reading from a topic can be limited to only the topic's current data, or it can wait for new written messages.

Reading current data

In this mode, reading is performed from the first to the last offset stored in the topic at the time the query is started. If data continues to be written to the topic, the query will stop after reaching the last offset known at startup. Specifying filters by service fields speeds up reading, as reading is performed only over the specified ranges.

```
SELECT
    Data    -- message body
FROM
    input_topic  -- local topic; for external: ext_source.input_topic
LIMIT 10;
```

Reading with data waiting

To wait for new messages, use the `WITH (STREAMING = "TRUE")` option. Reading starts from the current moment and continues until the number of messages specified in the `LIMIT` expression is read. The `LIMIT` parameter is required, without it the query will not finish, as it will wait for new messages indefinitely.

```
SELECT
    Data
FROM
    ext_source.input_topic  -- external topic; for local: input_topic
WITH (STREAMING = "TRUE")
LIMIT 10;
```

For continuous processing of incoming data, use streaming queries.

Message format and schema

When reading from a topic, the message body can be obtained in two ways: raw data and formatted data.

Raw data

Use it when the message content does not need to be parsed; it is enough to read the body as is.

```
SELECT
    Data
FROM
    input_topic  -- local topic; for external: ext_source.input_topic
WITH (
    FORMAT = raw,
    SCHEMA = (
        Data String
    )
)
LIMIT 10;
```

As a result, only the `Data` column is available: the message body in its original form.

The same result can be obtained without the `WITH` block: see table reading.

Formatted data

Use it when messages are serialized in a known format (JSON, CSV, etc.). The `FORMAT` parameter sets the parsing method, and `SCHEMA` sets the names and types of fields that will appear in the `SELECT` result:

```
SELECT
    Id,
    Name
FROM
    input_topic  -- local topic; for external: ext_source.input_topic
WITH (
    FORMAT = json_each_row,
    SCHEMA = (
        Id Uint64 NOT NULL,
        Name Utf8 NOT NULL
    )
);
```

Fields from `SCHEMA` are available in `SELECT` by name, like table columns.

For more details about supported formats: Data formats for reading/writing from topics.

Using the reader

A consumer is a named subscription to a topic that stores the current read position.

A consumer is created via the CLI or when creating a topic using CREATE TOPIC. The consumer name is specified in the query text with a pragma:

```
PRAGMA pq.Consumer="my_consumer";
```

If a consumer is not specified, reading from the topic is performed without one. Specifying a consumer allows tracking the read position and lag from the topic side, for example via the CLI.

Moving data from a topic to a table via UPSERT

Data from a topic can be moved to a table via UPSERT INTO:

```
UPSERT INTO
    table_name
SELECT
    Data            -- you can use any transformations
FROM
    ext_source.input_topic;  -- external topic; for local: input_topic
```

Service fields

When reading, you can request service fields and user-defined message attributes:

Field	Type	Description
`__ydb_create_time`	`Timestamp`	Message creation time
`__ydb_write_time`	`Timestamp`	Time the message was written to the topic
`__ydb_offset`	`Uint64`	Message offset in the partition
`__ydb_partition_id`	`Uint64`	Partition number
`__ydb_message_group_id`	`String`	Message group identifier
`__ydb_seq_no`	`Uint64`	Message sequence number within the group
`__ydb_user_attributes`	`Dict<String,String>`	User-defined message attributes

Example of using service fields:

```
SELECT
    Data,                                               -- message body
    __ydb_create_time AS CreateTime,                    -- message creation time
    __ydb_write_time AS WriteTime,                      -- message write time
    __ydb_offset AS Offset,                             -- message offset in topic
    __ydb_partition_id AS Partition,                    -- partition number
    __ydb_message_group_id AS MessageGroupId,           -- message group identifier
    __ydb_seq_no AS SeqNo                               -- sequence number within partition
FROM
    input_topic  -- local topic; for external: ext_source.input_topic
LIMIT 10;
```

Filters on service fields are evaluated before reading data from the topic and significantly reduce the volume of messages read. Comparison operators (`=` , `<>` , `<` , `<=` , `>` , `>=` , `IN`), logical conditions (`AND` , `OR`), and fields `partition_id` , `write_time` , `offset` are supported. Predicates on other service fields do not limit the read volume.

```
SELECT
    Data
```

```
FROM
    ext_source.input_topic  -- external topic; for local: input_topic
WHERE
    __ydb_partition_id = 42
        AND __ydb_offset >= 1000
        AND __ydb_offset <= 1100
        AND __ydb_write_time > CurrentUtcTimestamp() - Interval("PT2H");
```

Example of using user attributes:

```
SELECT
    COUNT(*) AS ErrorCount
FROM
    input_topic  -- local topic; for external: ext_source.input_topic
WHERE
    __ydb_user_attributes["type"] = "log"
        AND __ydb_user_attributes["level"] = "error";
```

Writing to a topic

Writing a single message

```
INSERT INTO
    output_topic  -- local topic; for external: ext_source.output_topic
SELECT
    "my_message";       -- message body
```

Writing data from a table

To write a table row with multiple columns to a topic, create a JSON object. The `TableRow` function creates a structure from all columns, `Yson::From` converts it to Yson, `Yson::SerializeJson` serializes it to a JSON string, and `ToBytes` converts the result to the `String` type required for writing to the topic:

```
INSERT INTO
    ext_source.output_topic  -- external topic; for local: output_topic
SELECT
    ToBytes(Unwrap(Yson::SerializeJson(Yson::From(TableRow()))))
FROM
    table_name;
```

Limitations

Warning

Writing user attributes via YQL is not supported.

Warning

Transactional writing via YQL/`INSERT INTO` is not supported — partial query results may appear in the topic.

See also

- CREATE TOPIC
- ALTER TOPIC
- DROP TOPIC
- Streaming queries

YDB Transactions and Queries

This section describes the specifics of YQL implementation for YDB transactions.

Query Language

The main tool for creating, modifying, and managing data in YDB is a declarative query language called YQL. YQL is an SQL dialect that can be considered a database interaction standard. YDB also supports a set of special RPCs useful in managing a tree schema or a cluster, for instance.

Transaction Modes

By default, YDB transactions are executed in *Serializable* mode. It provides the strictest isolation level for custom transactions. This mode guarantees that the result of successful parallel transactions is equivalent to their serial execution, and there are no read anomalies for successful transactions.

If consistency or freshness requirement for data read by a transaction can be relaxed, a user can take advantage of execution modes with lower guarantees:

- *Snapshot Read-Only*: All the read operations within a transaction access the database snapshot. All the data reads are consistent. The snapshot is taken when the transaction begins, meaning the transaction sees all changes committed before it began.
- *Stale Read-Only*: Read operations within a transaction may return results that are slightly out-of-date (lagging by fractions of a second). Each individual read returns consistent data, but no consistency between different reads is guaranteed.
- *Online Read-Only*: Each read operation in the transaction is reading the data that is most recent at execution time. The consistency of retrieved data depends on the *allow_inconsistent_reads* setting:
  - *false* (consistent reads): Each individual read operation returns consistent data, but no consistency is guaranteed between reads. Reading the same table range twice may return different results.
  - *true* (inconsistent reads): Even the data fetched by a particular read operation may contain inconsistent results.

> **Limitation for Online Read-Only and Stale Read-Only**
>
> Reading from column-oriented tables is not supported in these modes. Attempts fail with the following error:
>
> ```
> Read from column tables is not supported in Online Read-Only or Stale Read-Only transaction modes.
> Use Serializable or Snapshot Read-Only mode instead.
> ```
>
> For transactions that read column-oriented tables, use:
>
> - Serializable — the default mode.
> - Snapshot Read-Only — provides a read from a consistent snapshot.

Implicit Transactions

Implicit transaction logic applies when a single YQL script is sent to the server without an explicitly selected transaction mode. Typical entry points:

- YDB UI — the **Query** tab on the database page (query form), when run without selecting an explicit transaction mode in the settings.
- YDB CLI — a one-off script via `ydb sql`.
- Applications using the YDB SDK — ImplicitTx mode.

If no transaction mode is specified, YDB automatically manages its behavior. This mode is called an **implicit transaction**.

In this mode, based on the query, YDB decides whether to execute it outside a transaction or wrap it in a transaction with *Serializable* mode. Implicit transactions are a universal way to execute queries, as they support statements of any kind with a certain behavior described below.

Behavior for Different Types of Statements

- **Data Definition Language (DDL) Statements**
  DDL statements (such as `CREATE TABLE`, `DROP TABLE`, etc.) are executed outside a transaction. A query can consist only of DDL statements. If an error occurs, changes made by previous statements in the query are not rolled back.
- **Data Manipulation Language (DML) Statements**
  DML statements (such as `SELECT`, `UPSERT`, `UPDATE`, etc.) are wrapped in a transaction with *Serializable* mode. A query can consist only of DML statements. On success, changes are committed. If an error occurs, all changes are rolled back.
- **Batch Modification Statements**
  Batch modification statements (such as `BATCH UPDATE` and `BATCH DELETE`) are executed outside a transaction. A query can contain only one batch modification statement. If an error occurs, the statement's changes are not rolled back.

Summary Table

Statement Type	Implicit Transaction Handling	Multistatement Support	Rollback on Error
DDL	Outside a transaction	Yes (DDL-only)	No
DML	Auto transaction (Serializable)	Yes (DML-only)	Yes
Batch Modification Statements	Outside a transaction	No	No

To set a transaction mode explicitly, use the corresponding options at each entry point:

- YDB UI — choose a transaction mode in the execution settings on the **Query** tab.
- YDB CLI — for `table query execute` with `data` queries, set the `--tx-mode` parameter (default: `serializable-rw`, which corresponds to *Serializable*).
- YDB SDK — see setting the mode in the YDB SDK.

YQL Language

Statements implemented in YQL can be divided into two classes: Data Definition Language (DDL) and Data Manipulation Language (DML).

For more information about supported YQL constructs, see the YQL documentation.

Listed below are the features and limitations of YQL support in YDB, which might not be obvious at first glance and are worth noting:

- Multi-statement transactions (transactions made up of a sequence of YQL statements) are supported. Transactions may interact with client software, or in other words, client interactions with the database might look as follows: `BEGIN; make a SELECT; analyze the SELECT results on the client side; ...; make an UPDATE; COMMIT`. We should note that if the transaction body is fully formed before accessing the database, it will be processed more efficiently.
- YDB does not support transactions that combine DDL and DML queries. The conventional ACID notion of a transaction is applicable specifically to DML queries, that is, queries that change data. DDL queries must be idempotent, meaning repeatable if an error occurs. If you need to manipulate a schema, each manipulation is transactional, while a set of manipulations is not.
- YQL implementation used in YDB employs the Optimistic Concurrency Control mechanism. If an entity is affected during a transaction, optimistic blocking is applied. When the transaction is complete, the mechanism verifies that the locks have not been invalidated. For the user, locking optimism means that when transactions are competing with one another, the one that finishes first wins. Competing transactions fail with the `Transaction locks invalidated` error.
- All changes made during the transaction accumulate in the database server memory and are applied when the transaction completes. If the locks are not invalidated, all the changes accumulated are committed atomically; if at least one lock is invalidated, none of the changes are committed. The above model involves certain restrictions: changes made by a single transaction must fit inside the available memory.

For efficient execution, a transaction should be formed so that the first part of the transaction only reads data, while the second part of the transaction only changes data. The query structure then looks as follows:

```
SELECT ...;
....
SELECT ...;
UPDATE/REPLACE/DELETE ...;
COMMIT;
```

For more information about YQL support in YDB, see the YQL documentation.

Distributed Transactions

A database table in YDB can be sharded by the range of the primary key values. Different table shards can be served by different distributed database servers (including ones in different locations). They can also move independently between servers to enable rebalancing or ensure shard operability if servers or network equipment goes offline.

A topic in YDB can be sharded into several partitions. Different topic partitions, similar to table shards, can be served by different distributed database servers.

YDB supports distributed transactions. Distributed transactions are transactions that affect more than one shard of one or more tables and topics. They require more resources and take more time. While point reads and writes may take up to 10 ms in the 99th percentile, distributed transactions typically take from 20 to 500 ms.

Transactions with Topics and Tables

Warning

Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

YDB supports transactions involving row-oriented tables and/or topics. This makes it possible to transactionally transfer data from tables to topics and vice versa, as well as between topics. This ensures that data is neither lost nor duplicated in case of a network outage or other issues. This enables the implementation of the transactional outbox pattern within YDB.

For more information about transactions with tables and topics in YDB, see Transactions involving topics and Working with topics.

Transactions with Column and Row Tables

Currently, mixing column-oriented tables and row-oriented tables in a single transaction is supported only if the transaction performs read operations; no writes are allowed. Support for read-write transactions involving both table types is under development.

If a write transaction includes both types of tables, it fails with the following error: `Write transactions that use both row-oriented and column-oriented tables are disabled at current time`.

Change Data Capture (CDC)

> **Warning**
>
> Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Change Data Capture (CDC) captures changes to YDB table rows, uses these changes to generate a *changefeed*, writes them to distributed storage, and provides access to these records for further processing. It uses a topic as distributed storage to efficiently store the table change log.

When adding, updating, or deleting a table row, CDC generates a change record by specifying the primary key of the row and writes it to the topic partition corresponding to this key.

Guarantees

- Change records are sharded across topic partitions by primary key.
- Each change is only delivered once (exactly-once delivery).
- Changes by the same primary key are delivered to the same topic partition in the order they took place in the table.
- Change records are delivered to the topic partition only after the corresponding transaction in the table has been committed.

Limitations

- Changefeeds support records of the following types of operations:
  - Updates: overwriting the values of the specified columns. Query example: UPDATE.
  - Replacements: overwriting the values of the specified columns, the values of the unspecified columns are replaced by their default values. Query example: REPLACE INTO.
  - Erases. Query example: DELETE FROM.

Adding rows is a special update or replace case, and a record of adding a row in a changefeed will look similar to an update or replace record, depending on the original request that led to the change.

Virtual Timestamps

All changes in YDB tables are arranged according to the order in which transactions are performed. Each change is marked with a virtual timestamp which consists of two elements:

1. Global coordinator time.
2. Unique transaction ID.

Using these timestamps, you can arrange records from different partitions of the topic relative to each other or use them for filtering (for example, to exclude old change records).

For single-shard transactions, the timestamp of an existing distributed transaction in the execution queue can be used, which allows the timestamp to be non-unique. If there are no distributed transactions in the execution queue, the first element of the timestamp will be the time obtained from the coordinator, and the unique transaction ID will take the maximum possible value `18446744073709551615` (2^{64}-1, the maximum value of `Uint64`).

> **Note**
>
> By default, virtual timestamps are not emitted to the changefeed. To enable them, use the appropriate parameter when creating a changefeed.

Barriers

Barriers are service records without data about modification or deletion with virtual timestamps that appear in each partition of a topic at a specified interval. A barrier guarantees that any change with a virtual timestamp earlier than the barrier's has been written to this topic partition.

Barriers can be used to ensure strict ordering and global data consistency by buffering data between them.

> **Note**
>
> By default, barriers are not emitted to the changefeed. To set the frequency at which barriers are emitted, use the appropriate parameter when creating a changefeed.

Initial Table Scan

By default, a changefeed only includes records about those table rows that changed after the changefeed was created. Initial table scan enables you to export, to the changefeed, the values of all the rows that existed at the time of changefeed creation.

The scan runs in the background mode on top of the table snapshot. The following situations are possible:

- A non-scanned row changes in the table. The changefeed will receive, one after another: a record with the source value and a record about the update. When the same record is changed again, only the update record is exported.
- A changed row is found during scanning. Nothing is exported to the changefeed because the source value has already been exported at the time of change (see the previous paragraph).
- A scanned row changes in the table. Only an update record exports to the changefeed.

This ensures that, for the same row (the same primary key), the source value is exported first, and then the updated value is exported.

> **Note**
>
> The record with the source row value is labeled as an update record. When using virtual timestamps, records are marked by the snapshot's timestamp.

During the scanning process, depending on the table update frequency, you might see too many `OVERLOADED` errors. This is because, besides the update records, you also need to deliver records with the source row values. When the scan is complete, the changefeed switches to normal operation.

> **Warning**
>
> Automatic partitioning processes are suspended in the table and barriers are not emitted to the changefeed during the initial scan.

Record Structure

Depending on the changefeed parameters, the structure of a record may differ.

JSON Format

A JSON record has the following structure:

```
{
    "key": [<key components>],
    "update": {<columns>},
    "reset": {<columns>},
    "erase": {},
    "newImage": {<columns>},
    "oldImage": {<columns>},
    "ts": [<step>, <txId>],
    "user": "<user SID>",
    "traceId": "<Trace ID>"
}
```

- `key`: An array of primary key component values. Always present. The order of elements matches the order of the columns listed in the primary key of the table.
- `update`: Update flag. Present if a record matches the update operation. In `UPDATES` mode, it also contains the names and values of updated columns.
- `reset`: Replacement flag. Present if a record matches the replacement operation. In `UPDATES` mode, it also contains the names and values of the columns for which a value is set.
- `erase`: Erase flag. Present if a record matches the erase operation.
- `newImage`: Row snapshot that results from its being changed. Present in `NEW_IMAGE` and `NEW_AND_OLD_IMAGES` modes. Contains column names and values.
- `oldImage`: Row snapshot before the change. Present in `OLD_IMAGE` and `NEW_AND_OLD_IMAGES` modes. Contains column names and values.
- `ts`: Virtual timestamp. Present if the `VIRTUAL_TIMESTAMPS` setting is enabled. Contains the value of the global coordinator time (`step`) and the unique transaction ID (`txId`).
- `user`: User identifier. Present if the `USER_SIDS` setting is enabled. Contains the user's SID and is set to `ttl@system` if the record is deleted by the TTL process.
- `traceId`: OpenTelemetry trace identifier. Present if the `TRACE_IDS` setting is enabled.

Sample record of an update in `UPDATES` mode:

```
{
   "key": [1, "one"],
   "update": {
       "payload": "lorem ipsum",
       "date": "2022-02-22"
   }
}
```

Record of an erase:

```
{
   "key": [2, "two"],
   "erase": {}
}
```

Record with row snapshots:

```
{
   "key": [1, 2, 3],
   "update": {},
   "newImage": {
       "textColumn": "value1",
       "intColumn": 101,
       "boolColumn": true
   },
   "oldImage": {
       "textColumn": null,
       "intColumn": 100,
       "boolColumn": false
   }
}
```

Record with virtual timestamps:

```
{
   "key": [1],
   "update": {
       "created": "2022-12-12T00:00:00.000000Z",
       "customer": "Name123"
   },
   "ts": [1670792400890, 562949953607163]
}
```

A barrier record contains a single field `resolved` with a virtual timestamp:

```
{
    "resolved": [1670792500000, 0]
}
```

Note

- The same record may not contain the `update`, `reset` and `erase` fields simultaneously, since these fields are operation flags (you can't update and erase a table row at the same time). However, each record contains one of these fields (any operation is either an update, a replacement, or an erase).
- In `UPDATES` mode, the `update` or `reset` field for update or replacement operations is an operation flag and contains the names and values of updated columns.
- JSON object fields containing column names and values (`newImage`, `oldImage`, and `update` & `reset` in `UPDATES` mode), *do not include* the columns that are primary key components.
- If a record contains the `erase` field (indicating that the record matches the erase operation), this is always an empty JSON object (`{}`).

Debezium-Compatible JSON Format

A Debezium-compatible JSON record structure has the following format:

```
{
    "payload": {
        "op": <op>,
        "before": {<columns>},
        "after": {<columns>},
        "source": {
            "connector": <connector>,
            "version": <version>,
            "ts_ms": <ts_ms>,
            "step": <step>,
            "txId": <txId>,
            "snapshot": <bool>,
            "user": <user SID>,
            "traceId": <Trace ID>
        }
    }
}
```

- `op`: Operation that was performed on a row:
  - `c` — create. Applicable only in `NEW_AND_OLD_IMAGES` mode.
  - `u` — update.
  - `d` — delete.
  - `r` — read from snapshot.
- `before`: Row snapshot before the change. Present in `OLD_IMAGE` and `NEW_AND_OLD_IMAGES` modes. Contains column names and values.
- `after`: Row snapshot after the change. Present in `NEW_IMAGE` and `NEW_AND_OLD_IMAGES` modes. Contains column names and values.
- `source`: Source metadata for the event.
  - `connector`: Connector name. Current name is `ydb`.
  - `version`: Connector version that was used to generate the record. Current version is `1.0.0`.
  - `ts_ms`: Approximate time when the change was applied, in milliseconds.
  - `step`: Global coordinator time. Part of the virtual timestamp.
  - `txId`: Unique transaction ID. Part of the virtual timestamp.
  - `snapshot`: Whether the event is part of a snapshot.
  - `user`: User identifier. Present if the `USER_SIDS` setting is enabled. Contains the user's SID and equals `ttl@system` if the record is deleted by the TTL process.
  - `traceId`: OpenTelemetry trace identifier. Present if the `TRACE_IDS` setting is enabled.

When reading using Kafka API, the Debezium-compatible primary key of the modified row is specified as the message key:

```
{
    "payload": {<columns>}
}
```

- `payload`: Key of a row that was changed. Contains names and values of the columns that are components of the primary key.

Record Retention Period

By default, records are stored in the changefeed for 24 hours from the time they are sent. Depending on usage scenarios, the retention period can be reduced or increased up to 30 days.

> **Warning**
>
> Records whose retention time has expired are deleted, regardless of whether they were processed (read) or not.

Deleting records before they are processed by the client will cause offset skips, which means that the offsets of the last record read from the partition and the earliest available record will differ by more than one.

To set up the record retention period, specify the RETENTION_PERIOD parameter when creating a changefeed.

Topic Partitions

By default, the initial number of topic partitions is equal to the number of table partitions. You can redefine the initial number of topic partitions by specifying the TOPIC_MIN_ACTIVE_PARTITIONS parameter when creating a changefeed. To create a changefeed with a dynamically changing number of partitions, set the TOPIC_AUTO_PARTITIONING parameter when creating the changefeed.

> **Note**
>
> Currently, the ability to explicitly specify the number of topic partitions is available only for tables whose first primary key component is of type `Uint64` or `Uint32`.

Creating and Deleting a Changefeed

You can add a changefeed to an existing table or erase it using the ADD CHANGEFEED and DROP CHANGEFEED directives of the YQL `ALTER TABLE` statement. When erasing a table, the changefeed added to it is also deleted.

Getting and Updating Topic Settings

You can get the settings using an SDK or the YDB CLI by passing the path to the changefeed in the arguments, which has the following format:

```
path/to/table/changefeed_name
```

For example, if a table named `table` contains a changefeed named `updates_feed` in the `my` directory, its path looks as follows:

```
my/table/updates_feed
```

The topic settings can be updated using the expression ALTER TOPIC. Supported actions:

- updating settings:
  - `retention_period`;
  - `retention_storage_mb`;
  - `partition_write_burst_bytes`;
  - `partition_write_speed_bytes_per_second`.
- updating consumers.

CDC Purpose and Use

For information about using CDC when developing apps, see best practices.

Time to Live (TTL) and Eviction to External Storage

This section describes how the TTL mechanism works and what its limits are.

How It Works

The table's TTL is a sequence of storage tiers. Each tier contains an expression (TTL expression) and an action. When the expression is triggered, that tier is assigned to the row. When a tier is assigned to a row, the specified action is automatically performed: moving the row to external storage or deleting it. External storage is represented by the external data source object.

YDB allows you to specify a column (TTL column) whose values are used in TTL expressions. The expression is triggered when the specified number of seconds has passed since the time recorded in the TTL column. For rows with `NULL` value in TTL column, the expression is not triggered.

The timestamp for deleting a table item is determined by the formula:

```
eviction_time = valueof(ttl_column) + evict_after_seconds
```

> **Note**
>
> TTL doesn't guarantee that the item will be deleted exactly at `eviction_time`, it might happen later. If it's important to exclude logically obsolete but not yet physically deleted items from the selection, use query-level filtering.

Data is deleted by the *Background Removal Operation* (*BRO*), consisting of two stages:

1. Checking the values in the TTL column.
2. Deleting expired data.

The *BRO* has the following properties:

- The concurrency unit is a table partition.
- For tables with secondary indexes, the delete stage is a distributed transaction.

Guarantees

- For the same partition *BRO* is run at the intervals set in the TTL settings. The default run interval is 1 hour, the minimum allowable value is 15 minutes.
- Data consistency is guaranteed. The TTL column value is re-checked during the delete stage. This means that if the TTL column value is updated between stages 1 and 2 (for example, with `UPDATE`) and ceases to meet the delete criteria, the row will not be deleted.

Limitations

- The TTL column must be of one of the following types:
  - `Date`.
  - `Datetime`.
  - `Timestamp`.
  - `Uint32`.
  - `Uint64`.
  - `DyNumber`.
- The value in the TTL column with a numeric type (`Uint32`, `Uint64`, or `DyNumber`) is interpreted as a Unix time value. The following units are supported (set in the TTL settings):
  - Seconds.
  - Milliseconds.
  - Microseconds.
  - Nanoseconds.
- You can't specify multiple TTL columns.
- You can't delete the TTL column. However, if this is required, you should first disable TTL for the table.
- Only Object Storage is supported as external storage.
- The delete action can only be specified for the last tier.

Setup

Currently, you can manage TTL settings using:

- YQL.
- YDB console client.
- YDB C++, Go and Python SDK.

Database Limits

This section describes the parameters of limits set in YDB.

Schema Object Limits

The table below shows the limits that apply to schema objects: tables, databases, and columns. The "Object" column specifies the type of schema object that the limit applies to.
The "Error type" column shows the status that the query ends with if an error occurs. For more information about statuses, see Error handling in the API.

Objects	Limit	Value	Explanation	Internal name	Error type	Error message
Database	Maximum path depth	32	Maximum number of nested path elements (directories, tables).	MaxDepth	SCHEME_ERROR	Paths depth limit exceeded
Database	Maximum number of paths (schema objects)	10,000	Maximum number of path elements (directories, tables) in a database.	MaxPaths	PRECONDITION_FAILED	Paths count limit exceeded
Database	Maximum number of tablets	200,000	Maximum number of tablets (table shards and system tablets) that can run in the database. An error is returned if a query to create, copy, or update a table exceeds this limit. When a database reaches the maximum number of tablets, no automatic table sharding takes place.	MaxShards	PRECONDITION_FAILED	Shards count limit exceeded (in subdomain)
Database	Maximum object name length	255	Limits the number of characters in the name of a schema object, such as a directory or a table.	MaxPathElementLength	SCHEME_ERROR	Path part is too long
Database	Maximum ACL size	10 KB	Maximum total size of all access control rules that can be saved for the schema object in question.	MaxAclBytesSize	BAD_REQUEST	ACL size limit exceeded
Directory	Maximum number of objects	100,000	Maximum number of tables and child directories created in a directory.	MaxChildrenInDir	PRECONDITION_FAILED	Children count limit exceeded

Table	Maximum number of table shards	35,000	Maximum number of table shards.	MaxShardsInPath	PRECONDITION_FAILED	Shards count limit exceeded (in dir)
Table	Maximum number of columns	200	Limits the total number of columns in a table.	MaxTableColumns	SCHEME_ERROR on CREATE / BAD_REQUEST on ALTER	Too many columns
Table	Maximum column name length	255	Limits the number of characters in a column name.	MaxTableColumnNameLength	SCHEME_ERROR	Column name too long
Table	Maximum number of columns in a primary key	20	Each table must have a primary key. The number of columns in the primary key may not exceed this limit.	MaxTableKeyColumns	SCHEME_ERROR	Too many key columns
Table	Maximum number of indexes	20	Maximum number of indexes other than the primary key index that can be created in a table.	MaxTableIndices	PRECONDITION_FAILED	Indexes count has reached maximum value in the table
Table	Maximum number of followers	3	Maximum number of read-only replicas that can be specified when creating a table with followers.	MaxFollowersCount	BAD_REQUEST	Too much followers
Table	Maximum number of tables to copy	10,000	Limit on the size of the table list for persistent table copy operations.	MaxConsistentCopyTargets	BAD_REQUEST	Consistent copy object count limit exceeded

Size Limits for Stored Data

Parameter	Value	Error type
Maximum total size of all columns in a primary key	1 MB	GENERIC_ERROR
Maximum size of a string column value	16 MB	GENERIC_ERROR

Analytical Table Limits

Parameter	Value
Maximum row size	8 MB
Maximum size of an inserted data block	8 MB

Limits on Query Execution

The table below lists the limits that apply to query execution.

Parameter	Default	Explanation	Effect in case of a violation of the limit
Query duration	2 hours	The maximum amount of time allowed for a single query to execute.	Returns status code `TIMEOUT`
Maximum number of sessions per cluster node	1,000	The limit on the number of sessions that clients can create with each YDB node.	Returns status code `OVERLOADED`
Maximum query text length	10 KB	The maximum allowable length of YQL query text.	Returns status code `BAD_REQUEST`

Maximum size of parameter values	50 MB	The maximum total size of parameters passed when executing a parameterized query.	Returns status code `BAD_REQUEST`
Maximum size of a row	50 MB	The maximum total size of all fields of a single row returned or produced by the query.	Returns status code `PRECONDITION_FAILED`
Maximum number of locks per DataShard	10,000	The number of lock ranges per DataShard	Converts some locks to whole-shard locks, which use less memory but lock the entire shard instead of just a part.

Legacy Limits

In previous versions of YDB, queries were typically executed using an API called "Table Service". This API had the following limitations, which have been addressed by replacing it with a new API called "Query Service".

Parameter	Default	Explanation	Status in case of a violation of the limit
Maximum number of rows in query results	1,000	The complete results of some queries executed using the `ExecuteDataQuery` method may contain more rows than allowed. In such cases, the query will return the maximum number of rows allowed, and the result will have the `truncated` flag set.	SUCCESS
Maximum query result size	50 MB	The complete results of some queries may exceed the set limit. If this occurs, the query will fail and return no data.	PRECONDITION_FAILED

Resource pool limits

Parameter	Value
Maximum number of resource pool classifiers	1000

Limits on resource counts are also constrained by schema object limits. See above.

Backup concepts

YDB ensures data safety during hardware failures through replication and fault tolerance. However, replication does not protect against **logical errors**: an accidental `DROP TABLE`, an erroneous mass `UPDATE`, or `DELETE` will be replicated to all replicas. To protect against such scenarios, **backup** is required — a separate copy of data that can be restored from.

Full backup

A full backup is a snapshot of table data at a specific point in time. YDB provides several ways to create full backups, from simple to more functional.

Copying tables within a cluster

The simplest way is to create a copy of a table (or several tables) within the same cluster using the `ydb tools copy` command. The copy is created atomically from a consistent snapshot and uses a copy-on-write mechanism, so the operation is fast.

Suitable for:

- Quickly creating a safety copy before a dangerous operation.
- Cloning data for testing.

> **Warning**
>
> The copy is stored in the same cluster as the original data. It protects against logical errors but not against cluster loss.

Dump to file system

The `ydb tools dump` and `ydb tools restore` commands allow you to dump data to a local file system and restore it back.

Suitable for:

- Local development and testing.
- Small databases.
- Creating a copy on a separate medium.

Export to S3-compatible storage

The `ydb export s3` and `ydb import s3` commands allow you to export and import data to an external S3-compatible storage.

Suitable for:

- Disaster recovery (data is stored outside the cluster).
- Data migration between clusters.
- Long-term archiving.

Export to NFS

The `ydb export nfs` and `ydb import nfs` commands allow you to export and import data to a network file system (NFS) mounted on all hosts of the YDB cluster. Unlike dump to file system, the export is performed on the server side and does not require data transfer via CLI.

Suitable for:

- Disaster recovery (data is stored outside the cluster).
- Data migration between clusters.
- Long-term archiving.

For more details on configuring NFS for backup and recovery, see the recipe Backup and recovery via NFS.

Incremental backup

When working with large tables, repeatedly creating full backups can be too costly. Incremental backup solves this problem: after an initial full copy, each subsequent increment captures only the changes (inserts, updates, deletes) that occurred since the previous backup.

Incremental backups are organized into a **chain**:

```
Full backup → Increment₁ → Increment₂ → ... → Incrementₙ
```

To restore, the entire chain is needed: first the full copy is applied, then all increments sequentially. Restoration is performed to the state at the time of the last increment in the chain.

Incremental backup is implemented using backup collections.

> **Note**
>
> Currently, only row tables are supported.

Backup of system tablets

> **Note**
>
> Currently, only backup of cluster system tablets is supported. Backup of database system tablets is not supported.

The system tablet backup mechanism provides incremental copying of cluster metadata — such as Hive, BSController, and SchemeShard — to the local file system of cluster hosts.

This mechanism is used to restore cluster metadata when restoring from database backups is technically possible but not suitable in terms of time or effort. A typical scenario is when the total volume of databases in the cluster is large due to their number, the size of individual databases, or a combination of both; a full `import/restore` of all data into a new cluster in such a case leads to prolonged downtime. In this scenario, you can restore system tablets and bring the cluster back to a working state without performing a bulk restore of user data on the new cluster.

If the volume of databases allows for a standard restore, use export/import or dump/restore first. System tablet backup should be used as a special mechanism for situations where you need to restore cluster metadata and reduce the volume of restore operations.

> **Note**
>
> For practical instructions on enabling and restoring, see recipes for system tablet backup.

> **Warning**
>
> Backups of different system tablets are created independently of each other and are not consistent with each other. After restoration, the state of tablets may be inconsistent, which may negatively affect cluster operation.

How it works

Backup consists of two components:

- **State snapshot** — at each start, the tablet scans all its tables and writes its full state to a backup, including the data schema. The scan is based on a state snapshot and does not block tablet operation.
- **Change log** — with each change to data or schema, the tablet asynchronously writes the change to the log in parallel with writing to the distributed storage. When the log size exceeds the snapshot size, the tablet automatically creates a new snapshot.

> **Warning**
>
> Due to asynchronous writing, the latest changes that did not make it into the backup before the failure may be lost.

Backups are created **locally on the host where the tablet is currently running**. Therefore, the most recent copy is on the host where the tablet was running just before the failure.

The number of stored backups on a host is limited in the configuration. After a successful snapshot, the oldest copy is automatically deleted when the limit is exceeded. Incomplete copies (without a fully written snapshot) are deleted when a new backup is created.

Comparison of approaches

Method	Storage location	Incremental	Use cases
Copying tables within a cluster	Within the cluster	No	Quick copy before a dangerous operation
Dump to file system	File system	No	Development, testing, small databases
Export to S3-compatible storage	S3-compatible storage	No	Disaster recovery, migration, archiving
Export to NFS	Network File System (NFS)	No	Disaster recovery, migration, archiving
Incremental backup	Within the cluster (exported to S3 or file system)	Yes	Regular backups of large production databases
System tablet backup	Local file system of cluster hosts	Yes	Restoring cluster metadata in emergency situations

See also

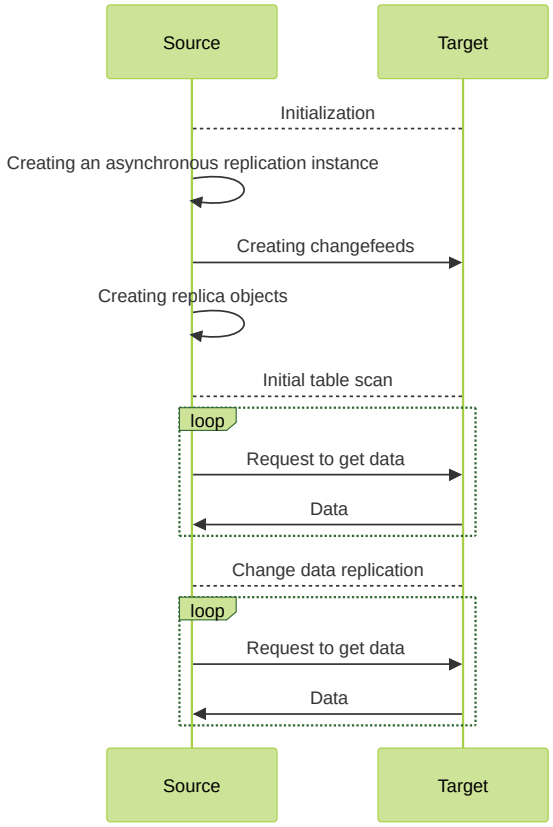
- Backup and restore — practical guide
- Backup collections — architecture and limitations
- Recipes for system tablet backup — enabling and restoring
- Backup and restore — practical guide
- Export/import reference — CLI commands
- YQL reference:
  - `CREATE BACKUP COLLECTION`
  - `BACKUP`
  - `RESTORE`
  - `DROP BACKUP COLLECTION`

Asynchronous Replication

Asynchronous replication allows for synchronizing data between YDB databases in near real time. It can also be used for data migration between databases with minimal downtime for applications interacting with these databases. Such databases can be located in the same YDB cluster as well as in different clusters.

Overview

Asynchronous replication is based on Change Data Capture and operates on logical data. The following diagram illustrates the replication process:



As shown in the diagram above, asynchronous replication involves two databases:

1. **Source**. A database with replicated objects.
2. **Target**. A database where an asynchronous replication instance and replica objects will be created.

Asynchronous replication consists of the following stages:

- Initialization
- Initial table scan
- Change data replication

Initialization

Initialization of asynchronous replication includes the following steps:

- Creating an asynchronous replication instance on the target database using the CREATE ASYNC REPLICATION YQL expression.
- Establishing a connection with the source database. The target database connects to the source using the connection parameters specified during the creation of the asynchronous replication instance.

> **Note**
>
> The user account that is used to connect to the source database must have the following permissions:
>
> - Read permissions for schema objects and directory objects
> - Create, update, delete, and read permissions for changefeeds

- The following objects are created for replicated objects on the source:
  - changefeeds on the source
  - replica objects on the target

> **Note**
>
> Replicas are created under the user account that was used to create the asynchronous replication instance.

Initial Table Scan

During the initial table scan the source data is exported to changefeeds. The target runs consumers that read the source data from the changefeeds and write it to replicas.

You can get the progress of the initial table scan from the description of the asynchronous replication instance.

Change Data Replication

After the initial table scan is completed, the consumers read the change data and write it to replicas.

Each change data block has its *creation time* (). Consumers track the *reception time* of the change data (). Thus, you can use the following formula to calculate the *replication lag*:

You can also get the replication lag from the description of the asynchronous replication instance.

Restrictions

- The set of replicated objects is immutable and is generated when YDB creates an asynchronous replication instance.
- YDB supports the following types of replicated objects:
  - row-based tables
  - directories

    YDB will replicate all row-based tables that are located in the given directories and subdirectories at the time the asynchronous replication instance is created.
- During asynchronous replication, you cannot add or delete columns in the source tables.
- During asynchronous replication, replicas are available only for reading.

Error Handling During Asynchronous Replication

Possible errors during asynchronous replication can be grouped into the following classes:

- **Temporary failures**, such as transport errors, system overload, etc. Requests will be resent until they are processed successfully.
- **Critical errors**, such as access violation errors, schema errors, etc. Replication will be aborted, and the description of the asynchronous replication instance will include the text of the error.

> **Warning**
>
> Currently, asynchronous replication that is aborted due to a critical error cannot be resumed. In this case, you must drop and create a new asynchronous replication instance.

For more information about error classes and how to address them, refer to Error Handling.

Consistency Levels of Replicated Data

Row-Level Data Consistency

The Change Data Capture used for asynchronous replication guarantees that changes to the same primary key are delivered in the exact order in which they occurred at the source. This ensures row-level data consistency.

Data written to replica objects in this mode becomes available for reading immediately.

Global Data Consistency

When transactions are performed at the source, several rows in a table or even rows in different table partitions may be atomically modified. Global data consistency at the target implies preserving atomicity: changes "appear" (become available for reading) in a consistent manner.

To ensure global data consistency, changefeeds must be created with barriers enabled. Data is buffered directly in the partitions of replica objects between barriers. When the next set of barriers from all topics is received, the changes are committed and become available for reading. As a result, the data at the target is consistent as of the timestamp specified in the barrier. If we assume that barriers are created every 10 seconds at the source (the default value), then under normal conditions the target also commits and publishes the set of changes from the source every 10 seconds.

Change Commit Interval

By default changes are committed no more than once every 10 seconds. You can override the change commit interval by specifying the COMMIT_INTERVAL option when creating an asynchronous replication instance.

> **Note**
>
> The change commit interval directly affects the barrier emission interval in the changefeed — the parameter values are synchronized. Thus, changes are usually committed at the same frequency as barriers appear in the changefeeds. However, in some cases, for example, if there is an uneven load across tables, barriers in changefeeds may appear at significantly different times, which can increase the change commit interval. You can find information about the lag in the description of the asynchronous replication instance.

Asynchronous Replication Completion

Completion of asynchronous replication might be an end goal of data migration from one database to another. In this case the client stops writing data to the source, waits for the zero replication lag, and completes replication. After the replication process is completed, replicas become available both for reading and writing. Then you can switch the load from the source database to the target database and complete data migration.

> **Note**
>
> You cannot resume completed asynchronous replication.

> **Warning**
>
> YDB currently supports only **forced** completion of asynchronous replication, when no additional checks are performed for data consistency, replication lag, etc.

To complete asynchronous replication, use the ALTER ASYNC REPLICATION YQL expression.

Dropping an Asynchronous Replication Instance

When you drop an asynchronous replication instance:

- Changefeeds are deleted in the source tables.
- The source tables are unlocked, and you can add and delete columns again.
- Optionally, all replicas are deleted.
- Asynchronous replication instance is deleted.

> **Warning**
>
> If asynchronous replication is dropped without prior completion and replica objects are not deleted, they will remain available only for reading.

To drop an asynchronous replication instance, use the DROP ASYNC REPLICATION YQL expression.

Bridge cluster operation mode

> **Feature of Yandex Enterprise Database**
>
> This functionality is available only in the Yandex Enterprise Database. In the open-source version of YDB it is absent.

This article describes a special operating mode of the YDB cluster in which data is stored with synchronous replication between multiple parts of the cluster called pile.

The bridge mode is an extension on top of the basic distributed storage modes: each pile is built using one of the YDB operating modes (for example, `mirror-3-dc` or `block-4-2`) and is fault-tolerant on its own; bridge adds synchronous replication between piles on top of that, plus managed failover/switchover of load between them. This provides an extra layer of fault tolerance on top of the chosen topologies. Bridge mode requires more hardware and is more complex to operate; it is especially useful for clusters deployed across two data centers and for systems with higher availability requirements — for example, when you need to remain available if three out of four data centers fail.

Description of the mode

In bridge mode, cluster nodes are split into several piles (typically one pile per datacenter). If one of the piles fails, the cluster becomes unavailable until the command to disable that pile is executed. After it is disabled, the cluster resumes operation.

YDB supports an arbitrary number of piles; however, for simplicity, the following discussion considers the case of two piles.

At the distributed storage level, bridge mode implements synchronous Active/Active replication between piles. At the database node level, Active/Passive redundancy is possible.

Data is stored in pairs of storage groups from different piles. Within each pile, any YDB operating mode supported for data storage may be used.

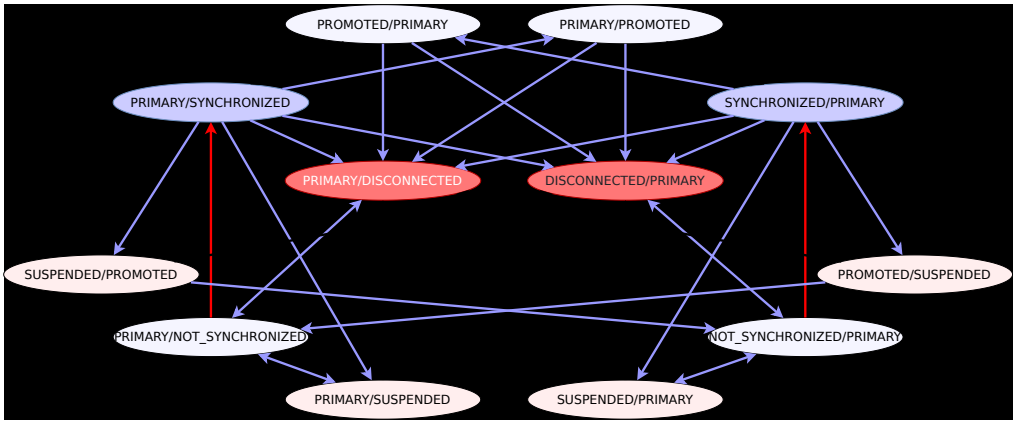
Bridge mode can achieve zero-downtime switchover. On failover, the cluster requires explicit disabling of replication to the failed pile before it can resume. When any pile is disabled, request processing is suspended until the command to disable replication to that pile is executed.

Piles are not standalone YDB clusters; they are parts of a single cluster with a complex topology:

- Tablets in bridge mode run as a single instance;
- each pile runs its own static group and set of independent storage groups with regular VDisks. Storage groups are accessed via DS-proxy-proxy, which exposes the DS-proxy interface to tablets and performs operations using two DS-proxy instances — one for the groups in each pile;
- each pile runs its own set of replicas of StateStorage, SchemeBoard, and Board. Access to StateStorage, SchemeBoard, and Board is through similar proxy-proxy components.

Pile states

Cluster operation is determined by the states of all its piles. For example, for a cluster of pile A and pile B, the pile states are written as (A, B), where order matters.



Each pile can be in one of the following states:

- `PRIMARY`. The main pile where tablets run. Only one pile can be in the `PRIMARY` state.
- `SYNCHRONIZED`. A follower pile whose storage groups are fully synchronized with `PRIMARY`.
- `DISCONNECTED`. Disconnected from `PRIMARY`; does not participate in operations.
- `NOT_SYNCHRONIZED`. A pile whose storage groups are not yet synchronized with `PRIMARY`. When synchronization completes, YDB automatically transitions the pile to `SYNCHRONIZED`.
- `PROMOTED`. A pile being smoothly transitioned from `SYNCHRONIZED` to `PRIMARY`. The pile remains in this state until the transition completes, after which YDB automatically transitions it to `PRIMARY`.
- `SUSPENDED`. A pile being smoothly transitioned to `DISCONNECTED`. When the transition completes, YDB automatically moves the pile to `DISCONNECTED`.

Normally, a pair of piles operates in `PRIMARY/SYNCHRONIZED` or `SYNCHRONIZED/PRIMARY`. That is, one pile is `PRIMARY` (tablets run there) and the other is `SYNCHRONIZED` (all its storage groups are synchronized with `PRIMARY`). In this configuration, a write is considered committed only after it has been successfully written to the storage groups of each pile with the required redundancy.

Transitions between states

From state	To state	How	Description
`PRIMARY`	`SYNCHRONIZED`	Automatically	Planned switchover complete.

PRIMARY	DISCONNECTED	Manually — failover	Emergency disable of unavailable PRIMARY and selection of new PRIMARY.
PRIMARY	SUSPENDED	Manually — takedown	Planned disable of current PRIMARY and selection of new PRIMARY.
SYNCHRONIZED	PRIMARY	Manually — failover	Emergency disable of unavailable PRIMARY and selection of new PRIMARY.
SYNCHRONIZED	DISCONNECTED	Manually — failover	Emergency disable of unavailable SYNCHRONIZED.
SYNCHRONIZED	PROMOTED	Manually — switchover	Start planned change of PRIMARY.
SYNCHRONIZED	SUSPENDED	Manually — takedown	Start planned disable of SYNCHRONIZED pile.
DISCONNECTED	NOT_SYNCHRONIZED	Manually — rejoin	Return pile to cluster after maintenance or recovery.
NOT_SYNCHRONIZED	SYNCHRONIZED	Automatically	Data synchronization complete.
NOT_SYNCHRONIZED	DISCONNECTED	Manually — failover	Emergency disable of unavailable NOT_SYNCHRONIZED.
PROMOTED	PRIMARY	Automatically	Planned switchover complete.
PROMOTED	PRIMARY	Manually — failover	Emergency disable of unavailable PRIMARY and selection of new PRIMARY.
PROMOTED	DISCONNECTED	Manually — failover	Emergency disable of unavailable PROMOTED.
SUSPENDED	NOT_SYNCHRONIZED	Manually — rejoin	Cancel planned disable of pile.
SUSPENDED	DISCONNECTED	Automatically	Planned disable complete.
SUSPENDED	DISCONNECTED	Manually — failover	Emergency disable of unavailable SUSPENDED.

State change scenarios

Pile failure

When a pile fails, the cluster becomes unavailable. To resume cluster operation, the failed pile must be disabled by moving it to the DISCONNECTED state.

The administrator can issue a transition from PRIMARY/SYNCHRONIZED or SYNCHRONIZED/PRIMARY to either PRIMARY/DISCONNECTED or DISCONNECTED/PRIMARY. The command updates the storage group configuration and changes how data is written to those groups. Interconnect tears down sessions with all nodes of the pile in DISCONNECTED; data exchange sessions are not established (TCP connections for exchanging pile state metadata may remain active). Subsequent read and write operations run without the pile in DISCONNECTED. If the PRIMARY pile has changed, tablets are restarted on it.

The same kind of transition applies when the failed pile was in any state and the pile being designated PRIMARY is in one of the allowed states: PRIMARY, SYNCHRONIZED, or PROMOTED.

If the pile being designated PRIMARY was in DISCONNECTED, NOT_SYNCHRONIZED, or SUSPENDED, a normal transition is not possible, because that pile may not hold a complete, up-to-date replica of the data.

Pile recovery

After the nodes of the failed pile are back up, the pile must be reattached to the cluster by moving it to NOT_SYNCHRONIZED.

The administrator can issue a transition from PRIMARY/DISCONNECTED to PRIMARY/NOT_SYNCHRONIZED or from DISCONNECTED/PRIMARY to NOT_SYNCHRONIZED/PRIMARY. Configuration is exchanged between nodes; interconnect sessions are established only between nodes with the same configuration. The command starts storage group synchronization. When synchronization finishes, an automatic transition occurs from PRIMARY/NOT_SYNCHRONIZED to PRIMARY/SYNCHRONIZED or from NOT_SYNCHRONIZED/PRIMARY to SYNCHRONIZED/PRIMARY.

Planned PRIMARY pile switchover

To change the PRIMARY pile in a planned way, the pile that will become the new PRIMARY must be moved to the PROMOTED state.

The administrator can issue a transition from PRIMARY/SYNCHRONIZED to PRIMARY/PROMOTED or from SYNCHRONIZED/PRIMARY to PROMOTED/PRIMARY. The command does not change how data is written to storage groups but updates their configuration and initiates the change of PRIMARY pile with a smooth migration of tablets to the new PRIMARY. When the transition completes, the former PRIMARY becomes SYNCHRONIZED and PROMOTED becomes PRIMARY.

Planned pile disable

To disable a pile in a planned way, it must be moved to the SUSPENDED state.

The administrator can issue a transition from PRIMARY/SYNCHRONIZED to PRIMARY/SUSPENDED or from SYNCHRONIZED/PRIMARY to SUSPENDED/PRIMARY. A planned disable of the nodes of the pile moved to SUSPENDED is performed, and they are moved to DISCONNECTED. The system aims to minimize the impact of this process on cluster operation. After the pile has been moved to DISCONNECTED, its nodes can be shut down for maintenance.

Then perform normal pile recovery by moving it to NOT_SYNCHRONIZED.

Recovery from cluster split with incompatible configuration (split brain)

A situation can occur where the administrator has put the cluster in a state where pile A and pile B are isolated from each other, and then pile A was reconfigured so that it became PRIMARY and pile B became SYNCHRONIZED, while at the same time pile B was reconfigured so that it became PRIMARY and pile A became SYNCHRONIZED. This results in a cluster split with incompatible configuration (split-brain). In this state, each part of the cluster may remain operational.

To restore a single cluster, choose which pile will be wiped. Then stop all nodes of that pile, wait for them to stop completely, wipe all disks on all nodes, and bring the nodes of the pile being wiped back up.
Then perform normal pile recovery by moving it to `NOT_SYNCHRONIZED`.

Non-standard pile recovery

In complex situations (for example, after sequential pile failures), the following scenario can occur: first pile A failed, was moved to `DISCONNECTED`, and the cluster continued operating; then pile B failed irreversibly; after that, pile A was recovered. In such cases, the cluster can be recovered based on pile A.

If the cluster must be brought back up when only a pile in `DISCONNECTED`, `NOT_SYNCHRONIZED`, or `SUSPENDED` is available, the administrator can run a transition from `PRIMARY/DISCONNECTED` to `DISCONNECTED/PRIMARY` (or similarly for `NOT_SYNCHRONIZED` or `SUSPENDED`) with the special `force` parameter. This causes a cluster split with incompatible configuration. Depending on the actual state of data in the pile being moved to `PRIMARY`, the cluster may be restored to a correct or internally inconsistent state. Correct operation of such a cluster is not guaranteed.

If the cluster turns out to be operational, you can proceed to restore it to normal state. Before recovering the disabled pile, fully wipe data and metadata on all its nodes, then move the pile to `NOT_SYNCHRONIZED`.

Bridge mode implementation details

For more on how bridge mode works, see Bridge mode.

Data transfer

Transfer in YDB is an asynchronous mechanism for moving data from a topic to a table. Creating a transfer instance, modifying it, and deleting it is done using YQL. The transfer runs inside the database and works in the background. Transfer is used to solve the task of delivering data from a topic to a table.

In practice, it's often more convenient to write data not directly to a table, but to a topic, and then asynchronously rewrite it from the topic to the table. This approach allows for even load distribution and handling spikes, since writing to a message queue is a lighter operation. Depending on the number of messages written to the topic, the delay in data availability for reading from the table after adding to the topic can range from several seconds to several minutes.

The transfer reads messages from the specified topic, transforms them into a format suitable for writing to the table, and then writes them to the target table. If the table already contains a row with the specified primary key, the row will be updated. Data transformation is performed using a lambda function that takes a message as a parameter and returns a list of structures, each corresponding to a row to be added or replaced in the table. For example, if the table contains a column `example_column` of type `Int32`, the structure must include a named field `example_column` of the same type.
If the lambda function returns an empty list, no rows in the table will be changed. The lambda function is not allowed to access tables and other YDB objects.
The lambda function being used can be obtained from the description of the transfer instance.

Transfer consumes additional server resources, primarily CPU. CPU consumption depends on the complexity of transforming the data written to the topic.

Transfer can read data from topics located both in the database where it's created and in another YDB database or cluster. To read a topic from another database when creating a transfer, you must specify the connection parameters to that database. The target table must be located in the same database where the transfer is created.

Reading from a topic is performed using a consumer. The transfer automatically adds a consumer with a unique name to the topic for reading messages. When the transfer is deleted, the created consumer is automatically deleted. It's also possible to create a consumer manually and specify its name when creating the transfer. A manually created consumer is not automatically deleted when the transfer is deleted. The name of the automatically created consumer can be obtained from the description of the transfer instance.

If data needs to be transferred from one topic to multiple tables, a separate transfer must be created for each table. Similarly, data from multiple topics can be directed to one table, for which each topic must be associated with a separate transfer. If the consumer name is set manually, then when there are multiple transfers reading from one topic, each transfer must be assigned a unique consumer, otherwise the data will be processed by only one of them.

> **Note**
>
> When updating a row in the table, all row values are overwritten with the data from the structure returned by the lambda function. If the structure doesn't contain a named field corresponding to a table column, the value of that column will be set to `NULL`. Named fields that don't have a correspondence in the table are ignored.
>
> When attempting to write a `NULL` value to a `NOT NULL` table column, the transfer ends with a critical error.

Guarantees

- Table writes are guaranteed to follow the write order in the topic partition. It's recommended to group messages related to one table row into one topic partition.
- Writing to the table is performed using bulk upsert without atomicity guarantees. Data writing is split into several independent transactions, each affecting a single table partition, with parallel execution.

Transfer startup

If a transfer is created without explicitly specifying a consumer name, a new consumer will be added to the topic. In this case processing of topic messages will start from the first message in the topic.
If a transfer is created with a previously created consumer specified, processing of topic messages will start from the first message not processed by that consumer.

Required permissions for transfer operation

To create and execute a transfer, the user must have write permissions to the target table and read permissions from the source topic. If the topic is located in another YDB database and the consumer is created automatically, additional permission to modify the topic is required, which allows the transfer to automatically create the consumer and delete it when the transfer is deleted.

Diagnostics

Current transfer parameters, including the lambda function text, can be viewed in the YDB UI and in the description of the transfer instance.

Data processing speed and delays can be monitored using consumer metrics that are used for reading from the topic.

Temporary transfer suspension

Transfer operation can be temporarily paused and then resumed. After resuming transfer operation, messages following the last processed message in the topic will start being processed.

To pause the transfer, you should change the transfer status to `PAUSED`. To resume the transfer, change the status to `ACTIVE`.

> **Warning**
>
> The topic has a message retention time, after which messages are deleted. If the transfer is paused longer than the retention time, the messages are deleted and won't be processed when the transfer is resumed.
>
> To guarantee that unprocessed messages are not deleted, you should make the consumer important.

Error handling during transfer

Different types of errors can occur during the transfer process:

- **Temporary failures**. Transport errors, system overload, and other temporary problems. Requests will be retried until successful execution.

- **Critical errors**. Errors related to access rights, data schema, and other critical aspects. When such errors occur, the transfer will be stopped, and the error text will be displayed on the transfer page in the YDB UI user interface. The error text can also be obtained from the description of the transfer instance.

To resume a transfer operation, eliminate the cause of the error and execute the `ALTER TRANSFER` command. For example, if the error was in the lambda function, change the lambda function. If the error is not related to the transfer configuration, for example, missing read permissions, then after eliminating the cause of the error, the transfer must be restarted by temporarily stopping and then resuming its operation.

See Also

- CREATE TRANSFER
- ALTER TRANSFER
- DROP TRANSFER
- Topic-to-Table Data Transfer Recipes

YDB for DevOps Engineers

This section of YDB documentation covers everything you need to know to work with YDB clusters.

Before getting started, it is recommended to familiarize yourself with YDB system requirements.

Main subsections:

- Concepts for DevOps engineers — additions to the general concepts section, relevant for DevOps engineers.
- YDB Cluster Configuration — YDB cluster configuration management.
- YDB Deployment Options — YDB cluster deployment options.
  - **Ansible**: for deployments on bare metal and virtual machines.
  - **Kubernetes**: for containerized deployments.
  - **Manual**: manual cluster deployment.
- Observability overview — tools for observing YDB clusters.
- Backup and recovery — backup and recovery of YDB clusters.

Concepts for DevOps engineers

This section supplements the general section Concepts YDB with theoretical materials that are primarily relevant for DevOps engineers.

Main topics:

- YDB System Requirements and Recommendations
- CPU settings for production
- Storage capacity planning
- YDB Versioning
- Maintenance without downtime
- SelfHeal

See also:

- YDB Deployment Options
- YDB Cluster Configuration
- Observability overview
- Backup and recovery

YDB System Requirements and Recommendations

This section provides recommendations for deploying YDB clusters that are relevant regardless of the chosen deployment method (Ansible, Kubernetes, or manual).

Hardware Configuration

The fault-tolerance requirements determine the necessary number of servers and disks. For more information, see YDB Cluster Topology.

Processor (CPU)

A YDB server can only run on x86-64 processors with AVX2 instruction support: Intel Haswell (4th generation) and later, AMD EPYC and later.

The ARM architecture is currently not supported for production YDB server deployments. For local development on Apple Silicon Macs, you can run YDB in Docker with x86_64 instruction emulation (Rosetta); see the Docker tab in YDB Quick Start.

> **Note**
>
> CPU power-saving modes (C-states, P-states, and the cpufreq scaling governor) can increase latency due to delays when leaving idle states.
>
> For production workloads, use the **performance** cpufreq governor (and equivalent BIOS settings).
>
> See CPU settings for production for details.

RAM

We recommend using error-correcting code (ECC) memory to protect against hardware failures.

Disk Subsystem

A YDB server can run on servers with any disk type (HDD/SSD/NVMe). However, we recommend using SSD/NVMe disks for better performance.

For YDB to work efficiently, we recommend using physical (not virtual) disks larger than 800 GB as block devices.

The minimum disk size is 80 GB, otherwise the YDB node won't be able to use the device. Correct and uninterrupted operation with minimum-size disks is not guaranteed. We recommend using such disks exclusively for informational purposes.

> **Warning**
>
> Configurations with disks less than 800 GB or any types of storage system virtualization cannot be used for production services or system performance testing.
>
> We don't recommend storing YDB data on disks shared with other processes (for example, the operating system).

YDB works with disk drives directly and does not use any filesystem to store data. Don't mount a file system or perform other operations with partitions used by YDB. Also, avoid sharing the YDB's block device with the operating system and different processes, which can lead to significant performance degradation.

Prefer to use physical local disk drives for YDB instead of virtual or network storage devices.

Remember that YDB uses some disk space for internal needs when planning disk capacity. For example, on a medium-sized cluster of 8 nodes, you can expect approximately 100 GB to be consumed for a static group on the whole cluster. On a large cluster with more than 1500 nodes, this will be about 200 GB. There are also 25.6 GB of logs on each Pdisk and a system area on each Pdisk. Its size depends on the size of the Pdisk, but is no less than 0.2 GB.

The disk is also used for spilling, a memory management mechanism that temporarily saves intermediate query execution results to disk when RAM is insufficient. This is important to consider when planning disk capacity. Detailed spilling configuration is described in the Spilling Configuration section.

Software Configuration

A YDB server can be run on servers with a Linux operating system, kernel 4.19 and higher, and libc 2.30. For example, Ubuntu 20.04, Debian 11, Fedora 34, or newer releases. For optimal performance, we recommend using more recent Linux kernel versions (6.6 or newer), as they include significant improvements in I/O subsystems, scheduling, and memory management that positively impact database workloads.

YDB uses the TCMalloc memory allocator. To make it efficient, enable Transparent Huge Pages and Memory overcommitment.

To improve disk and network I/O performance in a trusted environment, you can disable IOMMU by setting the Linux boot parameter `intel_iommu=off` or `amd_iommu=off`. In an untrusted environment, as well as under strict security requirements or with active virtualization use (for example, PCI passthrough and device isolation), disabling IOMMU is not recommended. In such cases, use `intel_iommu=on iommu=pt` or `amd_iommu=on iommu=pt`.

The environment can be considered trusted if only YDB and user-controlled applications are running on the server. Configurations that run third-party applications or virtual machines should be considered untrusted.

If the server has more than 32 CPU cores, to increase YDB performance, run each dynamic node in a separate taskset/cpuset of 10 to 32 cores. For example, with 128 CPU cores, an optimal setup is to run 4 dynamic nodes, each in its own 32-core taskset. Cores within one taskset/cpuset should belong to the same NUMA node.

MacOS and Windows operating systems are currently unsupported for running production YDB servers. However, running YDB in a Docker container on them is acceptable for development and functional testing.

CPU settings for production

Problem

CPU power-saving modes (such as Intel SpeedStep, AMD Cool'n'Quiet, C-states, and P-states) can hurt YDB performance for several reasons:

1. **Unstable CPU frequency** — YDB expects stable CPU throughput for predictable operation.
2. **Core affinity issues** — changing per-core frequency can break load balancing across threads.
3. **Mode-switching latency** — transitions between power-saving states add extra delays.
4. **Unpredictable throughput** — operations may run at different speeds depending on the current CPU mode.

Typical symptoms:

- Unstable query latency.
- Unexplained performance drops.
- Uneven load across cluster nodes.
- Erratic behavior under heavy load.

Solution

Recommended BIOS/UEFI settings

For YDB servers, configure the following BIOS/UEFI settings:

1. **CPU Power Management** — set to **Performance** or **Max Performance**.
2. **Intel Turbo Boost** — enable for maximum performance.
3. **C-states** — disable or limit to C1/C1E.
4. **P-states** — use a fixed maximum frequency.
5. **Intel SpeedStep / AMD Cool'n'Quiet** — disable.
6. **Package C-State Limit** — set to C0 or C1.

Operating system settings

On Linux, for deployments **not** using the Ansible role (which configures this automatically), run the following commands as root:

```
# Force the performance cpufreq governor
echo 'performance' | sudo tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor >/dev/null

# Pin minimum frequency to maximum
echo $(cat /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_max_freq) | \
sudo tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_min_freq >/dev/null

# Reduce PCIe ASPM power saving
echo 'performance' | sudo tee /sys/module/pcie_aspm/parameters/policy >/dev/null
```

Verifying current settings

Useful checks (after applying settings, the first command should output `performance` for every CPU core):

```
# Current governor
cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor

# Available frequencies
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies

# C-states
cat /sys/devices/system/cpu/cpu0/cpuidle/state*/name
```

Rationale

As a high-performance distributed database, YDB needs stable and predictable CPU performance. Client-style power-saving modes are a poor fit for server workloads that require sustained availability and low latency.

A stable CPU frequency helps with:

- Predictable operation latency.
- Correct behavior of load-balancing mechanisms.
- Steady throughput under load.
- Minimal delays for transaction processing.

Exceptions

Compromises may be acceptable when:

- **Test environments** — energy efficiency matters more than raw performance.
- **Standby nodes** — not serving active traffic.
- **Low-load deployments** — power savings are a priority.

For busy production clusters, follow the recommendations above.

Storage capacity planning

Storage capacity planning is the determination of the required number of physical disks to store a given volume of data, taking into account replication and overhead, as well as the inverse problem: estimating the useful data volume for a given amount of equipment.

Distributed storage model

At the hardware level, the YDB cluster includes a number of physical disks (HDD / SSD / NVMe). Other components (CPU, RAM, network) are not covered in this article.

Clusters YDB are multitenant, and one physical disk can be shared by multiple consumers — isolated databases. Each physical disk has one component running on top of it for working with it — PDisk. The resources of a single PDisk (IOPS, read and write bandwidth, capacity) are divided into equal shares called slots. Slots are reserved for creating logical (virtual) disks VDisk. Each VDisk is assigned one slot.

To manage resources, the user has access to the storage group entity. A storage group is a set of several VDisks on different servers that provide fault tolerance similar to RAID. The number of VDisks in a storage group is determined by the cluster's operating mode:

Mode	Data centers	Racks in the data center (minimum)	VDisk in storage group	Redundancy factor `RF`
`block-4-2`	1	8* (10+ recommended)	8	1.5
`mirror-3-dc`	3	3* (recommended 4+)	9	3 (up to 6**)
`mirror-3of4`	1	8* (10+ recommended)	8	4
`none`	1	1	1	1

- Values of 8 for `block-4-2` and `mirror-3of4`, and 3 for `mirror-3-dc` are the minimum number of racks required to ensure cluster operability. With this configuration, the cluster starts and operates, but has no margin for self-healing: if a rack fails, its VDisks have nowhere to migrate within the failure model, and any additional failure may lead to data unavailability. Additional racks are needed by the SelfHeal mechanism to move VDisks from a failed rack to the remaining equipment while preserving fault tolerance guarantees. Therefore, for practical operation, it is recommended to use at least 10 racks for `block-4-2` and `mirror-3of4`, and at least 4 racks for `mirror-3-dc`. For more details on storage modes and fault tolerance guarantees, see YDB Cluster Topology.

** For the `mirror-3-dc` mode in normal operation `RF = 3` (3 data copies). When one data center is unavailable, writing goes to 4 copies (2 in each remaining data center), and the load on the remaining DCs doubles. If a DC may be unavailable for a long time (or data centers are taken out for maintenance one by one), `RF = 6` must be provisioned. If only short-term failures are acceptable (recovery is faster than the system can overwrite a significant portion of data), `RF = 3` is sufficient. For more details on storage modes and fault tolerance guarantees, see YDB Cluster Topology.

From the user's perspective, a storage group can be viewed as a fixed set of resources: IOPS, read and write bandwidth, capacity. Each database owns one or more storage groups. Each storage group belongs to one specific database.

User tables consist of tablets. Each tablet manages a fragment of user or system data. Tablets store data in groups; each tablet can write data to multiple storage groups, and many tablets write to each storage group.

Quantities used

Tablet Storage is the total volume of data stored by the database tablets.

Database Storage is the total size allocated by the database VDisks. This size includes:

- the actual data written by tablets;
- redundancy from replication or error-correcting encoding;
- service metadata and logs;
- allocated but not actually used space.

For existing databases, the Tablet Storage and Database Storage values can be viewed in the YDB UI on the database page under the Info → Storage tab.

The distributed storage implements complex deferred garbage collection, so at any given time, part of the occupied space is taken up by outdated data that has not yet been removed by the collector.

Other overheads can vary significantly depending on the nature of the stored data and the workload. For a rough estimate, `Overhead = 2` can be considered — this is the median value obtained empirically from a sample of production databases in `mirror-3-dc` mode. Depending on the mode, data structure, and workload nature, overheads may differ substantially. For a more accurate estimate, a pilot experiment is recommended.

These quantities can be approximately related by the ratio:

```
Database Storage = Tablet Storage × RF × Overhead
```

Estimating Required Equipment

The task boils down to selecting a cluster configuration such that the data reliably fits into storage groups and the cluster retains a working reserve of empty slots to ensure fault tolerance.

Input parameters:

- `Tablet Storage` — the useful volume of data to be placed.
- `DriveSize` — the capacity of the disks used.

Desired parameters (cluster configuration):

- `NumRacks` — the number of racks / servers (failure domains)
- `DisksPerRack` — the number of disks in each failure domain.

Reference values:

- `RF` — the redundancy factor, determined by the cluster operating mode (see table above).
- `Overhead` — storage overhead (see section above); for a rough estimate, assume `2`.
- `ExpectedSlotCount` — the number of slots per PDisk; determined by disk characteristics — IOPS and read/write bandwidth. Typical values: `8` for HDD, `16` for SSD/NVMe.
- `VDisksInGroup` — the number of VDisks in a storage group, determined by the cluster operating mode (see table above).

To estimate the required equipment, perform the following steps.

1. Estimate the required Database Storage volume:

   ```
   Database Storage = Tablet Storage × RF × Overhead
   ```

2. Determine the slot size:

   ```
   SlotSize = (DriveSize - 27.65 GB) / ExpectedSlotCount
   ```

   About 27.65 GB of PDisk capacity is reserved for system needs; the remaining space is evenly distributed among slots.

   This formula is applicable for disks with a capacity of 800 GB or more. Using smaller disks is not recommended if optimal performance is required. For more details on disk subsystem requirements, see the YDB System Requirements and Recommendations section.

3. Estimate the number of storage groups:

   ```
   TotalGroups = ceil( Database Storage / (SlotSize × VDisksInGroup × 0.85) )
   ```

   The coefficient 0.85 reflects the recommended VDisk fill percentage (VDisk Raw Usage = 85%).

   When monitoring, it is recommended to focus on the VDisk Slot Usage and Capacity Alert values. For existing groups, the VDisk Raw Usage, VDisk Slot Usage, and Capacity Alert values can be viewed in the YDB UI on the **Info** → **Storage** tab of the database page.

   The threshold at which a group is considered full corresponds to the values VDisk Slot Usage = 100%, Capacity Alert = LightYellow, VDisk Raw Usage ≈ 90% — whereas the formula above leaves a margin up to this threshold.

4. Estimate the number of occupied slots:

   ```
   UsedSlots = TotalGroups × VDisksInGroup
   ```

5. Obtain a lower estimate of the required number of physical disks (without considering the fault tolerance reserve):

   ```
   TotalPDisks ≥ ceil( UsedSlots / ExpectedSlotCount )
   ```

6. Select the cluster configuration — parameters `NumRacks`, `DisksPerRack`, and determine the minimum required total reserve of empty slots in the cluster:

   ```
   TotalSlots    = NumRacks × DisksPerRack × ExpectedSlotCount
   MinEmptySlots = ceil( MaxSlotsInRack + 0.027 × TotalSlots )
   ```

   The reserve of empty slots is necessary for the normal operation of the SelfHeal mechanism, which performs automatic reconfiguration of storage groups to replace failed or long-unavailable disks.

   The first term `MaxSlotsInRack` is the maximum number of slots in one failure domain. For a homogeneous cluster `MaxSlotsInRack = DisksPerRack × ExpectedSlotCount`, for a heterogeneous one — `MaxSlotsInRack = max_i(DisksPerRack_i × ExpectedSlotCount)`. This reserve is necessary so that when the most capacious domain fails, its VDisks can fit on the remaining equipment.

   The second term `0.027 × TotalSlots` is an empirically selected operational reserve (~1 disk per 37), covering unscheduled replacement of individual disks.

7. Evaluate the configuration's compliance with the minimum requirements:

   ```
   EmptySlots = TotalSlots - UsedSlots
   ```

   The condition `EmptySlots ≥ MinEmptySlots` must be met. If the condition is not met, increase `NumRacks` or `DisksPerRack` and return to step 6. See Calculation Example.

Estimating Useful Storage Capacity

The inverse problem is to estimate how much useful data can be placed on a given amount of equipment.

Suppose there are `TotalPDisks` physical disks with a capacity of `DriveSize` each. Then:

1. Total number of slots:

   ```
   TotalSlots = TotalPDisks × ExpectedSlotCount
   ```

2. Some slots remain empty for the SelfHeal reserve, the rest are available for storage groups:

   ```
   MinEmptySlots = ceil( MaxSlotsInRack + 0.027 × TotalSlots )
   UsableSlots = TotalSlots - MinEmptySlots
   ```

   Here `MaxSlotsInRack` is the maximum number of slots in one failure domain (rack or server): for a homogeneous cluster `DisksPerRack × ExpectedSlotCount`, for a heterogeneous one — `max_i(DisksPerRack_i × ExpectedSlotCount)`.

3. Number of storage groups:

```
TotalGroups = floor( UsableSlots / VDisksInGroup )
```

4. Useful capacity of the distributed storage considering the 85% fill threshold:

```
Database Storage = TotalGroups × VDisksInGroup × SlotSize × 0.85
```

5. Estimation of useful data volume:

```
Tablet Storage = Database Storage / (RF × Overhead)
```

Calculation Example

Suppose you need to place 100 TB of data (Tablet Storage) in a cluster with fault tolerance mode `block-4-2`, using SSD disks with a capacity of 3.2 TB.

Initial parameters:

Parameter	Value
Tablet Storage	100 000 GB
Mode	`block-4-2`
RF	1.5
Overhead	2
DriveSize	3 200 GB
ExpectedSlotCount	16
VDisksInGroup	8

Calculation:

1. Database Storage = 100 000 × 1.5 × 2 = **300 000 GB**
2. SlotSize = (3 200 − 27.65) / 16 ≈ **198.27 GB**
3. TotalGroups = ⌈300 000 / (198.27 × 8 × 0.85)⌉ = **223 storage groups**
4. UsedSlots = 223 × 8 = **1 784 slots**
5. TotalPDisks (without reserve) = ⌈1 784 / 16⌉ = **112 disks**
6. Next, we will select a specific cluster configuration, providing a reserve of empty slots to ensure fault tolerance. To do this, we will consider various equipment options and evaluate compliance with the minimum requirements, taking into account that for mode `block-4-2` at least 8 failure domains (servers or racks) are required, and for stable practical operation 10 or more are recommended:
   - Let's take 10 racks with 12 disks each. In one rack, 12 × 16 = 192 slots, TotalSlots = 1 920. MinEmptySlots = ⌈192 + 0.027 × 1 920⌉ = 244. EmptySlots = 1 920 − 1 784 = 136. 136 < 244 — the minimum requirements are not met. It is easy to verify that 10 racks with 14 disks each are sufficient: TotalSlots = 2 240, MinEmptySlots = ⌈224 + 0.027 × 2 240⌉ = 285, EmptySlots = 2 240 − 1 784 = 456 > 285.
   - You can choose a server as the failure domain and consider 30 servers with 4 disks each. In one failure domain, 4 × 16 = 64 slots, TotalSlots = 1 920. MinEmptySlots = ⌈64 + 0.027 × 1 920⌉ = 116. EmptySlots = 1 920 − 1 784 = 136 > 116 — the minimum requirements are met.

Thus, to store 100 TB of data in mode `block-4-2` on SSD disks with a capacity of 3.2 TB, you will need 10 racks with 14 disks each (140 disks) or 30 servers with 4 disks each (120 disks).

Reverse calculation:

Next, let's take the configuration from the previous example — 30 servers with 4 SSD disks each with a capacity of 3.2 TB, and estimate how much useful data will fit in such a cluster.

1. TotalSlots = 120 × 16 = **1,920 slots**
2. MinEmptySlots = ⌈MaxSlotsInRack + 0.027 × TotalSlots⌉ = ⌈4 × 16 + 0.027 × 1,920⌉ = ⌈115.84⌉ = **116 slots**
3. UsableSlots = 1,920 − 116 = **1,804 slots**
4. TotalGroups = ⌊1,804 / 8⌋ = **225 storage groups**
5. SlotSize = (3,200 − 27.65) / 16 ≈ **198.27 GB**
6. Database Storage = 225 × 8 × 198.27 × 0.85 ≈ **303,353 GB**
7. Tablet Storage = 303,353 / (1.5 × 2) ≈ **101,118 GB**

Thus, a cluster of 30 servers and 120 SSD disks of 3.2 TB each in `block-4-2` mode can store about 101.1 TB of data (Tablet Storage).

Additional information

- YDB Cluster Topology
- YDB System Requirements and Recommendations
- Storage SelfHeal

YDB Versioning

YDB releases are named with a version that is a string consisting of several components. Depending on the context, some of the rightmost components may be omitted. The ordered list of components:

1. Last two digits of the release year
2. Major version number within the year
3. Minor version number within the major version
4. Patch number within the minor version
5. Release type

Major releases are usually identified by two components, for example, `24.3` is the third major release in 2024. Minor releases are identified by three components, for example, `24.3.14` is the fourteenth minor release within the third major version of 2024.

The list of available YDB releases is published on the download page. The YDB release policy is described in detail in the Manage YDB Releases article in the documentation section for YDB developers.

Version Compatibility

> **Note**
>
> Previously, the YDB server version consisted of three numbers (for example, `v24.3.3`), starting with major version `25.1`. The fourth number was added to specify the patch number (for example, `v25.1.1.3`). For more details about changes in version naming, see Manage YDB Releases.

You can update YDB to a new version in the following cases:

- If the major version numbers are the same, the minor versions can differ.
- If the major version numbers differ by one, first update to the latest available minor release of the current major version before updating to the next major version.

For example:

- `X.Y.*.* → X.Y.*.*` — update is possible, all minor versions within the same major version are compatible.
- `X.Y.Z.*` (last available `X.Y.*.*`) → `X.Y+1.*.*` - update is possible, major versions are sequential.
- `X.Y.*.*` → `X.Y+2.*.*` — update is not possible, major versions are non-sequential.
- `X.Y.*.* → X.Y-2.*.*` — update is not possible, major versions are non-sequential.

> **Warning**
>
> Also, you cannot downgrade YDB by more than two major versions because the older version might not support newer data formats stored on disk.

Version Compatibility Examples

- `v.25.1.3.2` → `v.25.1.5.5` - update is possible
- `v.25.1.5.5` → `v.25.2.3.1` - update is possible (where `v25.1.5.*` is the last available minor version in `v.25.1`)
- `v.25.1.4.1` → `v.25.2.3.1` - update is not possible, you must first update to the last minor version (`v.25.1.5.*`)
- `v.25.1.5.5` → `v.25.3.5.3` - update is not possible, you must first update to the next major version (`v.25.2.*.*`).

Formal Description of Possible Versions

```
<valid-version> ::= <version-core> "-" <version-type>
                  | <version-core>

<version-core> ::= <year> "." <major>
                 | <year> "." <major> "." <minor>
                 | <year> "." <major> "." <minor> "." <patch>

<year> ::= <digit> <digit>

<major> ::= <digits>

<minor> ::= <digits>

<patch> ::= <digits>

<version-type> ::= "testing"
                 | "stable"
                 | "lts"

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<digits> ::= <digit> | <digit> <digits>
```

Examples of Full Versions

- Testing version: `24.3.13.6-testing`
- Stable version: `24.3.14.2-stable`
- LTS version: `24.3.14.2-lts`

Maintenance without downtime

A YDB cluster periodically needs maintenance, such as upgrading its version or replacing broken disks. Maintenance can cause a cluster or its databases to become unavailable due to:

- Going beyond the expectations of the affected storage groups failure model.
- Going beyond the expectations of the State Storage failure model.
- Lack of computational resources due to stopping too many dynamic nodes.

To avoid such situations, YDB has a system tablet that monitors the state of the cluster - the *Cluster Management System (CMS)*. The CMS allows you to answer the question of whether a YDB node or host running YDB nodes can be safely taken out for maintenance. To do this, create a maintenance task in the CMS and specify in it to acquire exclusive locks on the nodes or hosts that will be involved in the maintenance. The cluster components on which the locks are acquired are considered unavailable from the CMS perspective and can be safely engaged in maintenance. The CMS will check the current state of the cluster and acquire locks only if the maintenance complies with the availability mode and unavailable node limits.

Failures during maintenance

During maintenance activities whose safety is guaranteed by the CMS, failures unrelated to those activities may occur in the cluster. If the failures threaten the cluster's availability, urgently aborting the maintenance can help mitigate the risk of cluster downtime.

Maintenance task

A *maintenance task* is a set of *actions* that the user asks the CMS to perform for safe maintenance.

Supported actions:

- Acquiring an exclusive lock on a cluster component (node, host, or disk).

Actions in a task are divided into groups. Actions from the same group are performed atomically. Currently, groups can consist of only one action.

If an action cannot be performed at the time of the request, the CMS informs you of the reason and time it is worth *refreshing* the task and sets the action status to *pending*. When the task is refreshed, the CMS attempts to perform the pending actions again.

Performed actions have a deadline after which they are considered *completed* and stop affecting the cluster. For example, an exclusive lock is released. An action can be completed early.

Protracted maintenance

If maintenance continues after the actions performed to make it safe have been completed, this is considered a failure in the cluster.

Completed actions are automatically removed from the task.

Availability mode

In a maintenance task, you need to specify the cluster's availability mode to comply with when checking whether actions can be performed. The following modes are supported:

- **Strong**: a mode that minimizes the risk of availability loss.
  - No more than one unavailable VDisk is allowed in each affected storage group.
  - No more than one unavailable State Storage ring is allowed.
- **Weak**: a mode that does not allow exceeding the failure model.
  - For affected storage groups with the block-4-2 scheme, no more than two unavailable VDisks are allowed.
  - For affected storage groups with the mirror-3-dc scheme, up to four unavailable VDisks are allowed, three of which must be in the same data center.
  - No more than `(nto_select - 1) / 2` unavailable State Storage rings are allowed.
- **Force**: a forced mode, the failure model is ignored. *Not recommended for use.*

Priority

You can specify the priority of a maintenance task. A lower value means a higher priority.

The task's actions cannot be performed until all conflicting actions from tasks with a higher priority are completed. Tasks with the same priority have no advantage over each other.

Unavailable node limits

In the CMS configuration, you can configure limits on the number of unavailable nodes for a database (tenant) or the cluster as a whole. Relative and absolute limits are supported.

By default, each database and the cluster as a whole are allowed to have no more than 13% unavailable nodes.

Checking algorithm

To check if the actions of a maintenance task can be performed, the CMS sequentially goes through each action group in the task and checks the action from the group:

- If the action's object is a host, the CMS checks whether the action can be performed with all nodes running on the host.
- If the action's object is a node, the CMS checks:
  - Whether there is a lock on the node.
  - Whether it's possible to lock the node according to the limits of unavailable nodes.
  - Whether it's possible to lock all VDisks of the node according to the availability mode.
  - Whether it's possible to lock the State Storage ring of the node according to the availability mode.
  - Whether it's possible to lock the node according to the limit of unavailable nodes on which cluster system tablets can run.
- If the action's object is a disk, the CMS checks:

- Whether there is a lock on the disk.
- Whether it's possible to lock all VDisks of the disk according to the availability mode.

The action can be performed if the checks are successful, and temporary locks are acquired on the checked nodes, hosts, or disks. The CMS then considers the next group of actions. Temporary locks help to understand whether the actions requested in different groups conflict with each other. Once the check is complete, the temporary locks are released.

Bridge mode

If the cluster runs in bridge mode, unavailable-node limits and availability checks are applied independently for each pile.

Examples

The ydbops utility tool uses CMS for cluster maintenance without downtime. You can also use the CMS directly through the gRPC API.

Rolling restart

To perform a rolling restart of the entire cluster, you can use the command:

```
$ ydbops restart
```

The default availability mode is `strong`. This mode minimizes the risk of availability loss. Use the `--availability-mode` parameter to override the default availability mode.

The `ydbops` utility will automatically create a maintenance task to restart the entire cluster using the given availability mode. As it progresses, the `ydbops` will refresh the maintenance task and acquire exclusive locks on the nodes in the CMS until all nodes are restarted.

Take out a host for maintenance

To take out a host for maintenance, follow these steps:

1. Create a maintenance task using the command:

   ```
   $ ydbops maintenance create --hosts=<fqdn> --duration=<seconds>
   ```

   This command creates a maintenance task that will acquire an exclusive lock for `<seconds>` seconds on the host with the fully qualified domain name `<fqdn>`.
2. After creating a task, refresh its state until the lock is taken, using the command:

   ```
   $ ydbops maintenance refresh --task-id=<id>
   ```

   This command refreshes the task with identifier `<id>` and attempts to acquire the required lock. When a `PERFORMED` response is received, proceed to the next step.
3. Perform host maintenance while the lock is acquired.
4. After the maintenance is complete, release the lock using the command:

   ```
   $ ydbops maintenance complete --task-id=<id> --hosts=<fqdn>
   ```

SelfHeal

SelfHeal automatically restores the fault tolerance of five types of YDB cluster objects:

1. Dynamic storage groups.
2. The static storage group.
3. State Storage replicas.
4. Board replicas.
5. SchemeBoard replicas.

These objects are handled by three mechanisms: SelfHeal for dynamic storage groups, a separate SelfHeal for the static storage group, and a shared SelfHeal for State Storage, Board, and SchemeBoard. The documentation groups these mechanisms into two sections: Storage SelfHeal (dynamic and static groups) and Metadata Distribution SelfHeal (State Storage, Board, and SchemeBoard).

When a node or disk fails, SelfHeal waits for the failure to persist before starting relocation. If the node or disk recovers before the mechanism is triggered, relocation does not start. For disks, the default waiting time is about one hour.

> **Note**
>
> SelfHeal for dynamic storage groups does not depend on the configuration version. SelfHeal for the static group and metadata distribution subsystems is available only with configuration V2 and distributed configuration enabled. Automatic static group management must also be allowed with the `automatic_static_group_management` parameter.

For more details about the mechanisms:

- Storage SelfHeal
- Metadata Distribution SelfHeal

Storage SelfHeal

Storage SelfHeal is a mechanism for automatically restoring YDB storage fault tolerance.

For an overview of the mechanisms and their operating conditions, see SelfHeal. Recovery of State Storage, Board, and SchemeBoard replicas is described in Metadata Distribution SelfHeal.

How storage SelfHeal works

Sentinel, a component of CMS, continuously monitors the state of PDisks and nodes. If a fault persists long enough (about one hour by default), Sentinel initiates relocation of the affected VDisks to healthy hardware so that the failure model is satisfied again.

The Blob Storage Controller executes the command: data is replicated in the background. The relocation itself can take from minutes to a day, depending on the data volume and the hardware. Once the command has been accepted, CMS treats the task as issued; distributed storage is responsible for completing replication.

Storage SelfHeal is enabled by default for dynamic groups. On clusters with configuration V2, you can also enable static group SelfHeal. With configuration V1, static group SelfHeal cannot be enabled.

Enabling and disabling SelfHeal

You can enable and disable SelfHeal using the YDB DSTool utility.

To enable SelfHeal, run the command:

```
ydb-dstool -e <bs_endpoint> cluster set --enable-self-heal
```

`<bs_endpoint>` is the endpoint of any storage node in the cluster.

To disable SelfHeal, run the command:

```
ydb-dstool -e <bs_endpoint> cluster set --disable-self-heal
```

SelfHeal settings

You can configure SelfHeal in **Viewer** → **Cluster Management System** → **CmsConfigItems**.

To create settings for the first time, click **Create**. If you need to change existing settings, click the ✎ button.

The following settings are available:

Parameter	Description
Status	Enabling and disabling SelfHeal in CMS.
Dry run	Enabling and disabling the mode in which CMS does not change the BSC setting.
Config update interval (sec.)	Period of configuration updates from BSC.
Retry interval (sec.)	Period of retries for configuration updates.
State update interval (sec.)	Period of PDisk state updates.
Timeout (sec.)	Timeout for PDisk state updates.
Change status retries	Number of retries to change the PDisk status in BSC (`ACTIVE` , `FAULTY` , `BROKEN` , etc.).
Change status retry interval (sec.)	Delay between attempts to change the PDisk status in BSC. CMS monitors the disk state at an interval of **State update interval**. If the disk remains in one state for several **Status update interval** cycles, CMS changes its status in BSC. Next are the settings for the number of update cycles after which CMS will change the disk status. If the disk state is `Normal` , the disk is moved to status `ACTIVE` ; in other states, the disk is moved to status `FAULTY` . The value `0` disables status change for the state (as implemented for `Unknown` by default). For example, with default settings, if CMS observes disk state `Initial` for 5 `Status update interval` cycles of 60 seconds each, the disk status will be changed to `FAULTY` .
Default state limit	For states for which no setting is specified, this "default" value can be used. For unknown PDisk states for which there is no setting, this value is also used. This value is used if the value is not set for states `Initial` , `InitialFormatRead` , `InitialSysLogRead` , `InitialCommonLogRead` , `Normal` .
Initial	PDisk starts initialization. Transitions to `FAULTY` .
InitialFormatRead	PDisk reads its format record. Transitions to `FAULTY` .
InitialFormatReadError	PDisk received an error while reading its format record. Transitions to `FAULTY` .
InitialSysLogRead	PDisk reads the system log. Transitions to `FAULTY` .
InitialSysLogReadError	PDisk received an error while reading the system log. Transitions to `FAULTY` .
InitialSysLogParseError	PDisk received an error while parsing or checking the consistency of the system log. Transitions to `FAULTY` .
InitialCommonLogRead	PDisk reads the common log of VDisks. Transitions to `FAULTY` .
InitialCommonLogReadError	PDisk received an error while reading the common log of VDisks. Transitions to `FAULTY` .
InitialCommonLogParseError	PDisk received an error while parsing or checking the consistency of the common log. Transitions to `FAULTY` .

CommonLoggerInitError	PDisk received an error while initializing internal structures intended for writing to the common log. Transitions to `FAULTY`.
Normal	PDisk has completed initialization and is operating normally. Transition to `ACTIVE` will occur after the specified number of cycles (for example, if `Normal` persists for 5 minutes, the disk transitions to state `ACTIVE`).
OpenFileError	PDisk received an error while opening the disk file. Transitions to `FAULTY`.
Missing	The node responds, but this PDisk is not in its list. Transitions to `FAULTY`.
Timeout	The node did not respond within the allotted timeout. Transitions to `FAULTY`.
NodeDisconnected	Node disconnection. Transitions to `FAULTY`.
Stopped	PDisk is stopped. Transitions to `FAULTY`.
Unknown	Unexpected response, for example, response `TEvUndelivered` to a state request. Transitions to `FAULTY`.

Working with donor disks

A donor disk is a previous VDisk after data migration that continues to store its data and only responds to read requests from the new VDisk. When migrating with donor disks enabled, previous VDisks continue to function until the data is fully migrated to new disks. To prevent data loss during VDisk migration, enable the use of donor disks:

```
ydb-dstool -e <bs_endpoint> cluster set --enable-donor-mode
```

To disable donor disks, enter the command:

```
ydb-dstool -e <bs_endpoint> cluster set --disable-donor-mode
```

Metadata Distribution SelfHeal

Metadata Distribution SelfHeal automatically manages the replica configurations of State Storage, Board, and SchemeBoard.

The configuration uses the historical name `state_storage_self_heal_config` for this mechanism. It applies to all three subsystems.

> **Warning**
>
> These instructions apply only to YDB clusters with **V2 configuration** and **distributed configuration**. On clusters with **V1 configuration**, these steps and commands (including obtaining configuration via `ydb admin cluster config fetch`) are unavailable or will not produce the expected result. Alternatives for V1 are not provided here — see Migration to V2 configuration.

The mechanism detects node failures and, if the nodes cannot recover quickly, relocates the affected replicas to other nodes. As the cluster grows, it can also automatically increase replica counts based on the configuration and available nodes.

The Sentinel component of the CMS cluster management system triggers the mechanism.

Enabling and disabling Metadata Distribution SelfHeal

You can enable and disable Metadata Distribution SelfHeal by changing the configuration:

1. Get the current cluster configuration using the ydb admin cluster config fetch command:

   ```
   ydb [global options...] admin cluster config fetch > config.yaml
   ```

2. Modify the configuration file `config.yaml` by changing the value of parameter `state_storage_self_heal_config.enable` to `true` or `false`:

   ```
   config:
       self_management_config:
           enabled: true # Enabling distributed configuration
       cms_config:
           sentinel_config:
               enable: true # Enabling Sentinel
               state_storage_self_heal_config:
                   enable: true # Enabling Metadata Distribution SelfHeal
   ```

   > **Note**
   >
   > For the mechanism to work, both CMS Sentinel and distributed configuration must be activated. Make sure they are enabled.
   >
   > See also: Migration to V2 configuration and enabling distributed configuration.

   When the `state_storage_self_heal_config.enable` parameter is set to `true`, the mechanism for maintaining the operability and fault tolerance of State Storage, Board, and SchemeBoard is enabled.

3. Update the cluster configuration taking into account the changes made using ydb admin cluster config replace:

   ```
   ydb [global options...] admin cluster config replace -f config.yaml
   ```

Managing automatic configuration changes

In addition to globally enabling or disabling Metadata Distribution SelfHeal with the `state_storage_self_heal_config.enable` parameter, you can use the `self_management_config` section of `config.yaml` to manage automatic configuration changes separately for each metadata distribution subsystem and restrict the nodes to which SelfHeal can relocate replicas.

```
config:
    self_management_config:
        enabled: true
        automatic_state_storage_management: true
        automatic_state_storage_board_management: true
        automatic_scheme_board_management: true
        state_storage_self_heal_allowed_nodes: [1, 2, 3, 4, 5, 6, 7, 8]
        state_storage_board_self_heal_allowed_nodes: [1, 2, 3, 4, 5, 6, 7, 8]
        scheme_board_self_heal_allowed_nodes: [1, 2, 3, 4, 5, 6, 7, 8]
```

Parameter	Default value	Description
`automatic_state_storage_management`	`true`	Allows SelfHeal to automatically change the State Storage configuration. When set to `false`, SelfHeal does not change the current State Storage configuration.
`automatic_state_storage_board_management`	`true`	Same for Board: allows or disallows SelfHeal to automatically change its configuration.
`automatic_scheme_board_management`	`true`	Same for SchemeBoard: allows or disallows SelfHeal to automatically change its configuration.
`state_storage_self_heal_allowed_nodes`	`[]` (no restrictions)	List of node IDs to which SelfHeal can move or on which it can add State Storage replicas. An empty list means there are no restrictions and any cluster nodes can be used.

`state_storage_board_self_heal_allowed_nodes`	`[]` (no restrictions)	Same for Board replicas.
`scheme_board_self_heal_allowed_nodes`	`[]` (no restrictions)	Same for SchemeBoard replicas.

Additional Metadata Distribution SelfHeal parameters

In the `cms_config.sentinel_config.state_storage_self_heal_config` section of the `config.yaml` configuration file, you can configure additional Metadata Distribution SelfHeal parameters. They affect how quickly the mechanism responds to changes and how many metadata distribution subsystem replicas are created. The example below shows all parameters with their default values:

```
config:
    cms_config:
        sentinel_config:
            enable: true
            state_storage_self_heal_config:
                enable: true
                wait_for_config_step: 60000000
                relax_time: 600000000
                pileup_replicas: false
                override_replicas_in_ring_count: 0
                override_rings_count: 0
                replicas_specific_volume: 200
```

Parameter	Default value	Description
`wait_for_config_step`	`60000000` (microseconds, 60 seconds)	Wait time between intermediate steps of applying a new configuration of the metadata distribution subsystems (adding/removing ring groups, clearing the `WriteOnly` flag, see Configuring State Storage). The value is specified in microseconds.
`relax_time`	`600000000` (microseconds, 600 seconds)	Minimum interval between two consecutive Metadata Distribution SelfHeal activations. Until the specified time has elapsed since the previous activation, a repeated configuration change is not started, even if faulty nodes are detected. The value is specified in microseconds.
`pileup_replicas`	`false`	Allows placing replicas of different subsystems (State Storage, Board, SchemeBoard) on the same set of nodes. When set to `false`, SelfHeal tries to use different nodes for replicas of different subsystems where possible; when set to `true`, nodes already occupied by one subsystem can be reused for the others.
`override_replicas_in_ring_count`	`0` (calculated automatically)	Forcibly sets the number of replicas in one ring. If the value is `0`, the number of replicas in the ring is calculated automatically based on `replicas_specific_volume` and the number of available nodes.
`override_rings_count`	`0` (calculated automatically)	Forcibly sets the number of rings in the configuration. If the value is `0`, the number of rings is calculated automatically based on the number of available nodes and the cluster topology.
`replicas_specific_volume`	`200`	Determines how many cluster nodes should correspond to one additional replica in the ring: one additional replica is added for every `replicas_specific_volume` nodes in the cluster. Used in automatic calculation of the number of replicas if `override_replicas_in_ring_count` is not set (equals `0`).

Checking the result

You can check that the changes have been applied in the `CMS` section of the cluster YDB UI (available on port 8765): go to the `Sentinel` tab to view the status of Sentinel and Metadata Distribution SelfHeal.

YDB Cluster Configuration

This section presents materials on YDB cluster configuration management. You will learn about two versions of the configuration mechanism (V1 and V2), their differences, and how to check which version is used in your cluster.

Main subsections:

- Configuration V1 — section about configuration V1, used in YDB.
- Configuration V2 — section about configuration V2, experimental functionality for YDB versions 25.1 and above.
- Cluster Configuration Migration — migration between configurations V1 and V2.
- Checking Configuration Version — checking which configuration version is used on the cluster.
- Comparing YDB Cluster Configurations: V1 and V2 — detailed comparison of configurations V1 and V2.

Tip

New YDB clusters are recommended to be deployed using configuration V2. If a cluster was deployed using configuration V1, it will still use it after updating to YDB version 25.1 or higher. After such an update, it is recommended to plan and perform migration to V2, because support for V1 will be discontinued in future versions of YDB. For the instructions on how to determine the configuration version of the cluster, see Checking Configuration Version.

Configuration V1

This section of the YDB documentation describes Configuration V1, which is the main way to configure YDB clusters deployed using YDB.

Configuration V1 is a two-level YDB cluster configuration system consisting of static configuration and dynamic configuration:

1. **Static configuration**: a YAML format file that is located locally on each static node and used when starting the `ydbd server` process. This configuration contains, among other things, static group and State Storage settings.
2. **Dynamic configuration**: a YAML format file that is an extended version of static configuration. It is loaded via CLI and reliably stored in the Console tablet, which then distributes the configuration to all dynamic cluster nodes. Using dynamic configuration is optional.

You can learn more about Configuration V1 in the Configuration V1 Overview section.

Starting from version v25.1, YDB supports Configuration V2, a unified approach to configuration in a single file format. When using Configuration V2, automatic configuration of static group and State Storage becomes possible. When deploying new clusters on YDB version v25.1 and above, it is recommended to use Configuration V2.

Main materials:

- Configuration V1 Overview
- YDB Cluster Configuration
- Dynamic Cluster Configuration
- Volatile Configurations
- Cluster Configuration Domain-Specific Language (DSL)
- Changing Configurations via CMS
- Changing Actor System Configuration
- Expanding a Cluster
- Configuring metadata distribution subsystems State Storage, Board, and Scheme Board
- State Storage Move
- Static Group Move
- Replacing Node FQDN
- Database Node Authentication and Authorization

Configuration V1 Overview

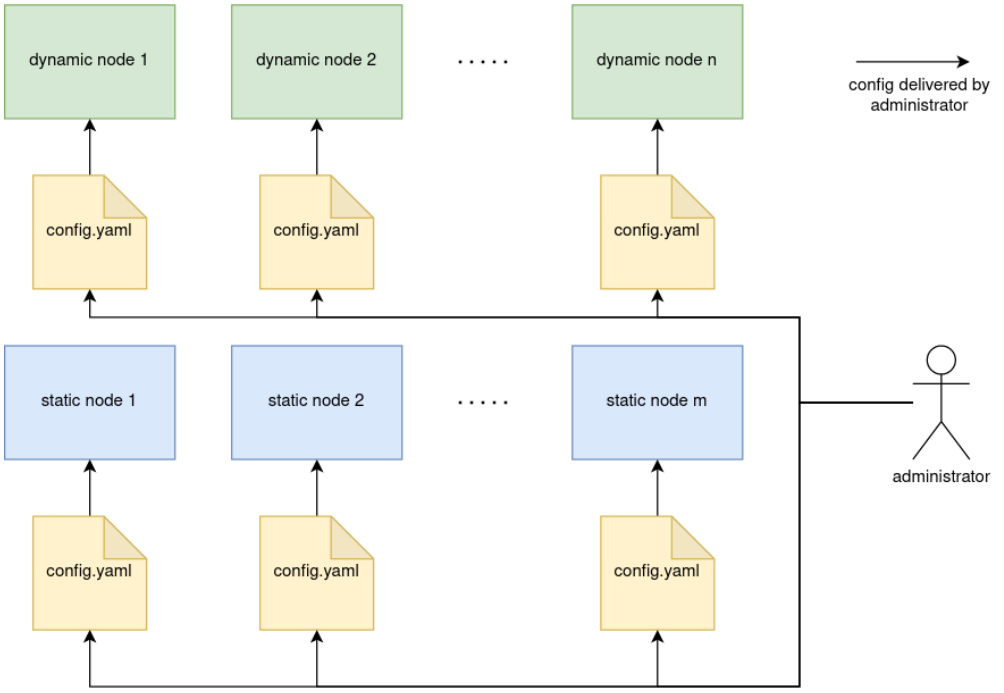
To start a YDB node, configuration is required. There are two types of configuration:

- **Static** — a YAML format file stored on the node's local disk.
- **Dynamic** — a YAML format document stored in the YDB configuration storage.

Static cluster nodes use static configuration. Dynamic nodes can use static configuration, dynamic configuration, or their combination.

Static Configuration

Static configuration is a YAML file stored on cluster nodes. This file lists all system settings. The path to the file is passed to the `ydbd` process at startup through a command line parameter. Distribution of static configuration across the cluster and maintaining it in a consistent state on all nodes is the responsibility of the cluster administrator. Details on using static configuration can be found in the YDB Cluster Configuration section. This configuration is **required** to start static nodes.

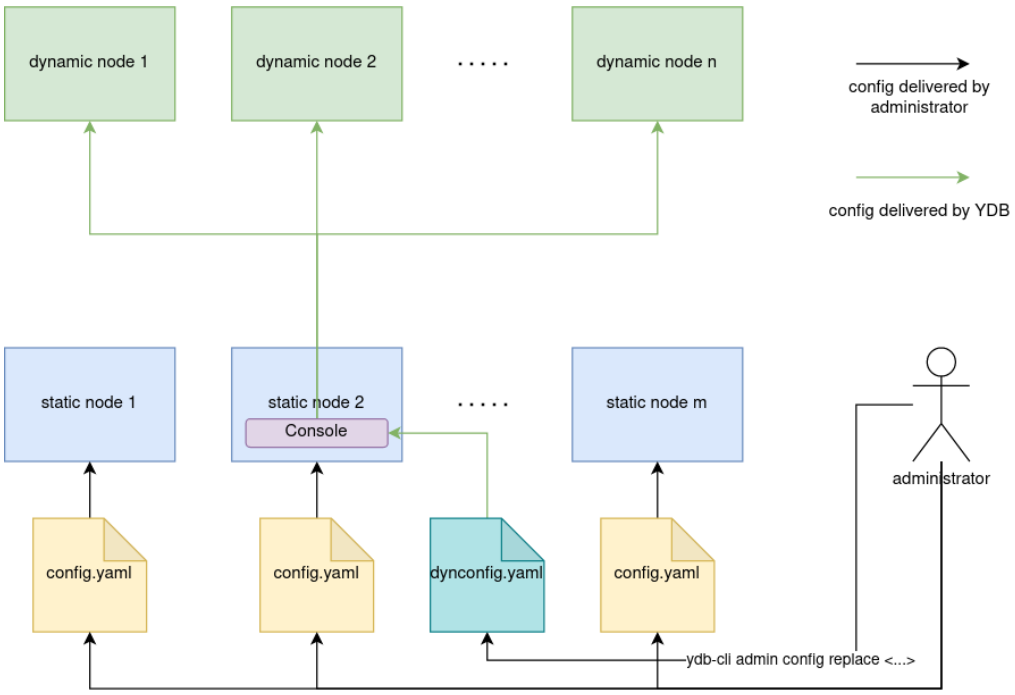


Basic Usage Scenario

1. Copy standard configuration from GitHub.
2. Modify the configuration according to your requirements.
3. Place identical configuration files on all cluster nodes.
4. Start all cluster nodes, explicitly specifying the path to the configuration file using the `--yaml-config` command line argument.

Dynamic Configuration

Dynamic configuration is a YAML document reliably stored in the cluster in the Console tablet. Unlike static configuration, it is sufficient to load it into the cluster, as YDB will be responsible for its distribution and maintaining it in a consistent state. At the same time, dynamic configuration using selectors allows handling complex scenarios while remaining within a single configuration file. A description of dynamic configuration is presented in the Dynamic Cluster Configuration section.



Basic Usage Scenario

1. Copy standard configuration from GitHub.
2. Modify the configuration according to your requirements.
3. Place identical configuration files on all static cluster nodes.
4. Start all static cluster nodes, explicitly specifying the path to the configuration file using the `--yaml-config` command line argument.
5. Supplement the configuration file to dynamic configuration format.
6. Load the resulting configuration to the cluster using `ydb admin config replace -f dynconfig.yaml`.

YDB Cluster Configuration

The cluster configuration is specified in the YAML file passed in the `--yaml-config` parameter when the cluster nodes are run. This article provides an overview of the main configuration sections and links to detailed documentation for each section.

Each configuration section serves a specific purpose in defining how the YDB cluster operates, from hardware resource allocation to security settings and feature flags. The configuration is organized into logical groups that correspond to different aspects of cluster management and operation.

Configuration Sections

The following top-level configuration sections are available, listed in alphabetical order:

Section	Required	Description
actor_system_config	Yes	CPU resource allocation across actor system pools
auth_config	No	Authentication and authorization settings
aws_client_config	No	AWS client default settings
blob_storage_config	No	Static cluster group configuration for system tablets
bridge_config	No	Cluster piles for bridge mode
client_certificate_authorization	No	Client certificate authentication
cms_config	No	Cluster Management System configuration
domains_config	No	Cluster domain configuration including Blob Storage and State Storage
feature_flags	No	Feature flags to enable or disable specific YDB features
healthcheck_config	No	Health check service thresholds and timeout settings
hive_config	No	Hive component configuration for tablet management
host_configs	No	Typical host configurations for cluster nodes
hosts	Yes	Static cluster nodes configuration
kafka_proxy_config	No	Kafka Proxy configuration
log_config	No	Logging configuration and parameters
memory_controller_config	No	Memory allocation and limits for database components
node_broker_config	No	Stable node names configuration
query_service_config	No	Federated query connector configuration
resource_broker_config	No	Resource broker for controlling CPU and memory consumption
security_config	No	Security configuration settings
table_service_config	No	Query processing configuration
tli_config	No	Transaction lock invalidation (TLI) diagnostics parameters
tls	No	TLS configuration for secure connections

Practical Guidelines

While this documentation section focuses on complete reference documentation for available settings, practical recommendations on what to tune and when can be found in the following places:

- As part of the initial YDB cluster deployment:
  - Ansible
  - Kubernetes
  - Manual
- As part of troubleshooting
- As part of security hardening

Sample Cluster Configurations

You can find model cluster configurations for deployment in the repository. Check them out before deploying a cluster.

Dynamic Cluster Configuration

Dynamic configuration allows running dynamic nodes by configuring them centrally, without the need to manually distribute files across nodes. YDB acts as a configuration management system, providing tools for reliable storage, versioning, and configuration delivery, as well as a DSL (Domain Specific Language) for overriding parts of it for specific node groups. The configuration is a YAML document. It is an extended version of static configuration:

- configuration description is moved to the `config` field
- a `metadata` field is added, necessary for validation and versioning
- `allowed_labels` and `selector_config` fields are added for granular setting overrides

This configuration is loaded into the cluster, where it is reliably stored and delivered to each dynamic node at startup. Some settings are updated on the fly, without restarting nodes. Using dynamic configuration, you can centrally solve the following tasks:

- switch component logging settings for both the entire cluster and individual databases or node groups
- enable experimental functionality (feature flags) on individual databases
- change actor system settings on a specific database, individual node, or group of nodes

Preparing to Use Dynamic Configuration

Before starting to use dynamic configuration in the cluster, it is necessary to perform preparatory work:

1. Enable database node authentication and authorization in the cluster.
2. If configuration management via CMS was previously used in the cluster, export existing settings in YAML format. To do this, run the following command:

   ```
   ./ydbd -s grpcs://<node1.ydb.tech>:2135 --ca-file ca.crt --token-file ydbd-token \
        admin console configs dump-yaml > dynconfig.yaml
   ```

   You must first obtain an authentication token using the `ydb auth get-token` command, similar to the cluster initialization procedure.
3. Create the initial dynamic configuration file:
   - If settings were previously made in CMS (exported in the previous step), use the resulting file as a basis and:
     - add a `metadata` section following the pattern from the configuration example
     - add the `yaml_config_enabled: true` parameter to the `config` section
   - If no settings were previously made via CMS, use the minimal content of the dynamic configuration file.
   - If the cluster uses actor system interconnect encryption, add the corresponding TLS settings for interconnect to the `config` section.
4. Apply the created dynamic configuration file to the cluster:

   ```
   # Apply configuration file dynconfig.yaml to the cluster
   ydb admin config replace -f dynconfig.yaml
   ```

> **Note**
>
> After enabling dynamic configuration support in the YDB cluster, the legacy configuration management function via CMS will become unavailable.

Configuration examples

Example of minimal dynamic configuration for a single-datacenter cluster:

```
# Configuration metadata.
# Field managed by server
metadata:
  # Cluster name from cluster_uuid parameter set during cluster installation, or "" if parameter is not set
  cluster: ""
  # Configuration file identifier, always increases by 1 and starts from 0.
  # Automatically increased when loading new configuration to server.
  version: 0
# Main cluster configuration, all values from it are applied by default until overridden by selectors.
# Content is similar to static cluster configuration
config:
  # must always be set to true to use yaml configuration
  yaml_config_enabled: true
  # actor system configuration - since by default this section is used
  # only by DB nodes, configuration is set specifically for them
  actor_system_config:
    # automatic configuration selection for node based on type and number of available cores
    use_auto_config: true
    # HYBRID || COMPUTE || STORAGE — node type, always COMPUTE for database nodes
    node_type: COMPUTE
    # number of allocated cores
    cpu_count: 14
allowed_labels: {}
selector_config: []
```

Configuration parameters are described in more detail on the YDB Cluster Configuration page.

The initially installed cluster dynamic configuration gets version number 1. When applying a new configuration, the version of the stored configuration is compared with that specified in the YAML document and automatically increased by one.

Example of more complex dynamic configuration with typical global parameters and special parameters for one of the databases:

```
---
metadata:
  kind: MainConfig
  cluster: ""
  version: 1
config:
  yaml_config_enabled: true
  table_profiles_config:
    table_profiles:
    - name: default
      compaction_policy: default
      execution_policy: default
      partitioning_policy: default
      storage_policy: default
      replication_policy: default
      caching_policy: default
    compaction_policies:
    - name: default
    execution_policies:
    - name: default
    partitioning_policies:
    - name: default
      auto_split: true
      auto_merge: true
      size_to_split: 2147483648
    storage_policies:
    - name: default
      column_families:
      - storage_config:
          sys_log:
            preferred_pool_kind: ssd
          log:
            preferred_pool_kind: ssd
          data:
            preferred_pool_kind: ssd
    replication_policies:
    - name: default
    caching_policies:
    - name: default
  interconnect_config:
    encryption_mode: REQUIRED
    path_to_certificate_file: "/opt/ydb/certs/node.crt"
    path_to_private_key_file: "/opt/ydb/certs/node.key"
    path_to_ca_file: "/opt/ydb/certs/ca.crt"
allowed_labels:
  node_id:
    type: string
  host:
    type: string
  tenant:
    type: string
selector_config:
- description: Custom settings for testdb
  selector:
    tenant: /cluster1/testdb
  config:
    shared_cache_config:
      memory_limit: 34359738368
    feature_flags: !inherit
      enable_views: true
    actor_system_config:
      use_auto_config: true
      node_type: COMPUTE
      cpu_count: 14
```

Updating Dynamic Configuration

```
# Get cluster configuration
ydb admin config fetch > dynconfig.yaml
# Edit using any text editor
vim dynconfig.yaml
# Apply configuration file dynconfig.yaml to cluster
ydb admin config replace -f dynconfig.yaml
```

Additional configuration capabilities are described on the selectors and volatile configuration pages.
All commands for working with configuration are described in the Fetch the cluster configuration section.

Implementation

Configuration Update from Administrator Perspective

1. The configuration file is loaded by the user using a grpc call or YDB CLI into the cluster.
2. The file is checked for validity, basic constraints are checked, version correctness, cluster name correctness, correctness of configurations obtained after DSL transformation.
3. The configuration version in the file is increased by one.
4. The file is reliably stored in the cluster by the Console tablet.
5. File updates are distributed across cluster nodes.

Configuration Update from Cluster Node Perspective

1. Each node requests full configuration at startup.
2. Upon receiving configuration, the node generates final configuration for its set of labels.
3. The node subscribes to configuration updates by registering with the Console tablet.
4. In case of configuration update, the local service receives it and transforms it for the node's labels.
5. All local services subscribed to updates receive the updated configuration.

Points 1,2 are performed only for dynamic cluster nodes.

Configuration Versioning

This mechanism allows avoiding concurrent configuration modification and making its update idempotent. When receiving a configuration modification request, the server compares the version of the received modification with the saved one. If the version is one less, then configurations are compared — if they are identical, it means the user is trying to load the configuration again, the user gets OK, and the configuration on the cluster is not updated. If the version equals the current one on the cluster, then the configuration is replaced with the new one, while the version field is increased by one. In all other cases, the user gets an error.

Dynamically Updatable Settings

Part of the system settings are updated without restarting nodes. To change them, it is sufficient to load a new configuration and wait for its distribution across the cluster.

List of dynamically updatable settings:

- `immediate_controls_config`
- `log_config`
- `memory_controller_config`
- `monitoring_config`
- `table_service_config`
- `tracing_config.external_throttling`
- `tracing_config.sampling`

The list may be expanded in the future.

Limitations

- Using more than 30 different labels in selectors can lead to delays of tens of seconds during configuration validation, as YDB needs to check the validity of each possible final configuration. At the same time, the number of values of one label has much less impact.
- Using large files (more than 500KiB for a 1000-node cluster) configuration can lead to increased network traffic in the cluster when updating configuration. Traffic volume is directly proportional to the number of nodes and configuration volume.

Volatile Configurations

Volatile configurations are a special type of configurations that complement dynamic ones and are not persistent. That is, these configurations are reset when the Console tablet moves or restarts, as well as when the main configuration is updated.

Main usage scenarios:

- temporary configuration changes for debugging or testing;
- trial enabling of potentially dangerous settings. In case of cluster crash or restart, these settings will be automatically disabled.

These configurations are added to the end of the selector set, the description syntax is identical to selector syntax.

```
# Get all volatile configurations loaded on the cluster
ydb admin volatile-config fetch --all --output-directory <dir>
# Get volatile configuration with id=1
ydb admin volatile-config fetch --id 1
# Apply volatile configuration volatile.yaml to the cluster
ydb admin volatile-config add -f volatile.yaml
# Remove volatile configurations with id=1 and id=3 on the cluster
ydb admin volatile-config drop --id 1 --id 3
# Remove all volatile configurations on the cluster
ydb admin volatile-config drop --all
```

Example of Working with Volatile Configuration

Temporarily enabling `blobstorage` component logging settings to `DEBUG` on node `host1.example.com`:

```
# Request current metadata to form correct volatile configuration header
$ ydb admin config fetch --all
---
kind: MainConfig
cluster: "example-cluster-name"
version: 2
config:
  # ...
---
kind: VolatileConfig
cluster: "example-cluster-name"
version: 2
id: 1
selector_config:
  # ...
# Load configuration with version 2, cluster name example-cluster-name and identifier 2
$ ydb admin volatile-config add -f - <<<EOF
metadata:
  kind: VolatileConfig
  cluster: "example-cluster-name"
  version: 2
  id: 2
selector_config:
- description: Set blobstorage logging level to DEBUG
  selector:
    node_host: host1.example.com
  config:
    log_config: !inherit
      entry: !inherit_key:component
      - component: BLOBSTORAGE
        level: 8
EOF
# ...
# log analysis
# ...
# Remove configuration
$ ydb admin volatile-config drop --id 2
```

Cluster Configuration Domain-Specific Language (DSL)

Cluster Configuration DSL

Selectors

The main entity of the DSL is **selectors**. They allow the overriding of parts of the configuration or the entire configuration for specific nodes or groups of nodes. For example, they can be used to enable experimental functionality for nodes of a particular database. Each selector is an array of overrides and extensions to the main configuration. Each selector has a `description` field, which can be used to store an arbitrary description string. The `selector` field represents a set of rules that determine whether the selector should be applied to a specific node based on a set of labels. The `config` field describes the override rules. Selectors are applied in the order they are defined.

Labels

Labels are special tags used to mark nodes or groups of nodes. Each node has a set of automatically assigned labels:

- `node_id` — the internal identifier of the node in the system
- `node_host` — the node's `hostname` obtained at startup
- `tenant` — the database served by this node
- `dynamic` — whether this node is dynamic (true/false)

Additionally, the user can explicitly define any additional labels when starting the `ydbd` process on the node using command-line arguments, such as `--label example=test`.

Example of Using Selectors

The example below defines the actor system's general configuration and the tenant `large_tenant` configuration. By default, with such a configuration, the actor system assumes that each node has 4 cores, while nodes of the `large_tenant` have 16 cores. The actor system's node type is overridden to `COMPUTE`.

```
metadata:
  cluster: ""
  version: 8
config:
  actor_system_config:
    use_auto_config: true
    node_type: STORAGE
    cpu_count: 4

# This section is used as a hint when generating possible configurations using the resolve command
allowed_labels:
  dynamic:
    type: string

selector_config:
- description: large_tenant has bigger nodes with 16 cpu # arbitrary description string
  selector: # selector for all nodes of the tenant large_tenant
    tenant: large_tenant
  config:
    actor_system_config: !inherit # reuse the original actor_system_config, the semantics of !inherit are des
cribed in the section below
      # in this case, !inherit allows managing the actor_system_config.use_auto_config parameter for the enti
re cluster by changing only the base setting
      cpu_count: 16
      node_type: COMPUTE
```

Permissive Labels

A mapping in which you can set the allowable values for labels. This section is used as a hint when generating possible configurations using the resolve command. Values are not validated at node startup.

There are two types of labels available:

- string;
- enum.

string

It can take any value or be unset.

Example:

```
dynamic:
  type: string
host_name:
  type: string
```

enum

It can take values from the `values` list or be unset.

Example:

```
flavour:
  type: enum
  values:
    ? small
```

```
    ? medium
    ? big
```

Selector Behavior

Selectors represent a simple predicate language. Selectors for each label are combined using the **AND** condition.

Simple Selector

The following selector will select nodes where the `label1` is equal to `value1` **and** the `label2` is equal to `value2`:

```
selector:
  label1: value1
  label2: value2
```

The following selector will select **ALL** nodes in the cluster, as no conditions are specified:

```
selector: {}
```

In

This operator allows for selecting nodes with label values from a list.

The following selector will select all nodes where `label1` is equal to `value1` **or** `value2`:

```
selector:
  label1:
    in:
    - value1
    - value2
```

NotIn

This operator allows selecting nodes where the chosen label does not match any value from a list.

The following selector will select all nodes where `label1` is equal to `value1` **and** `label2` is not equal to `value2` **and** `value3`:

```
selector:
  label1: value1
  label2:
    not_in:
    - value2
    - value3
```

Additional YAML Tags

Tags are necessary for partial or complete reuse of configurations from previous selectors. They allow you to merge, extend, delete, and override parameters set in previous selectors and the main configuration.

!inherit

Scope: YAML mapping
Action: similar to the merge tag in YAML, copy all child elements from the parent mapping and merge with the current ones, overwriting them.
Example:

Original configuration	Override	Resulting configuration

`config:`
`  some_config:`
`  first_entry: 1`
`  second_entry: 2`
`  third_entry: 3`
	`config:`
	`  some_config: !inherit`
	`  second_entry: 100`
		`config:`
		`  some_config:`
		`  first_entry: 1`
		`  second_entry: 100`
		`  third_entry: 3`

!inherit:<key>

Scope: YAML sequence
Action: copy elements from the parent array and overwrite, treating the `key` object in the elements as the key, appending new keys to the end.
Example:

Original configuration	Override	Resulting configuration

`config:`
`  some_config:`
`  array:`
`  - abc: 2`
`  value: 10`
`  - abc: 1`
`  value: 20`
`  another_value: test`
	`config:`
	`  some_config: !inherit`
	`  array: !inherit:abc`
	`  - abc: 1`
	`  value: 30`
	`  - abc: 3`
	`  value: 40`
		`config:`
		`  some_config:`
		`  array:`
		`  - abc: 2`
		`  value: 10`
		`  - abc: 1`
		`  value: 30`
		`  - abc: 3`
		`  value: 40`

!remove

Scope: YAML sequence element under `!inherit:<key>`
Action: remove the element with the corresponding key.
Example:

Original configuration	Override	Resulting configuration

`config:`
`  some_config:`
`  array:`
`  - abc: 2`
`  value: 10`
`  - abc: 1`
`  value: 20`
`  another_value: test`
	`config:`
	`  some_config: !inherit`
	`  array: !inherit:abc`
	`  - !remove`
	`  abc: 1`
		`config:`
		`  some_config:`
		`  array:`
		`  - abc: 2`
		`  value: 10`

!append

Scope: YAML sequence
Action: copy elements from the parent array and append new ones to the end.
Example:

Original configuration	Override	Resulting configuration

`config:`
`  some_config:`
`  array:`
`  - abc: 2`
`  value: 10`
`  - abc: 1`
`  value: 20`
`  another_value: test`
	`config:`
	`  some_config: !inherit`
	`  array: !append`
	`  - abc: 1`
	`  value: 30`
	`  - abc: 3`
	`  value: 40`
		`config:`
		`  some_config:`
		`  array:`
		`  - abc: 2`
		`  value: 10`
		`  - abc: 1`
		`  value: 20`
		`  another_value: test`
		`  - abc: 1`
		`  value: 30`
		`  - abc: 3`
		`  value: 40`

Generating Final Configurations

Configurations can contain complex sets of overrides. With the YDB CLI, you can view the final configurations for:

- specific nodes
- sets of labels
- all possible combinations for the current configuration

```
# Generate all possible final configurations for cluster.yaml
ydb admin config resolve --all -f cluster.yaml
# Generate the configuration for cluster.yaml with labels tenant=/Root/test and canary=true
ydb admin config resolve -f cluster.yaml --label tenant=/Root/test --label canary=true
# Generate the configuration for cluster.yaml with labels similar to those on node 1001
ydb admin config resolve -f cluster.yaml --node_id 1001
# Take the current cluster configuration and generate the final configuration for it with labels similar to t
hose on node 1001
ydb admin config resolve --from-cluster --node_id 1001
```

The configuration transformation command is described in more detail in the section Fetch the cluster configuration.

Example output of `ydb admin config resolve --all -f cluster.yaml` for the following configuration file:

```
metadata:
  cluster: ""
  version: 8
config:
  actor_system_config:
    use_auto_config: true
    node_type: STORAGE
    cpu_count: 4
allowed_labels:
  dynamic:
    type: string
selector_config:
- description: Actorsystem for dynnodes # arbitrary description string
  selector: # selector for all nodes with label dynamic = true
    dynamic: true
  config:
    actor_system_config: !inherit # reuse the original actor_system_config, the semantics of !inherit are des
cribed in the section below
      node_type: COMPUTE
      cpu_count: 8
```

Output:

```
---
label_sets: # sets of labels for which the configuration is generated
- dynamic:
    type: NOT_SET # one of three label types: NOT_SET | COMMON | EMPTY
config: # generated configuration
  invalid: 1
  actor_system_config:
    use_auto_config: true
    node_type: STORAGE
    cpu_count: 4
---
label_sets:
- dynamic:
    type: COMMON
    value: true # label value
config:
  invalid: 1
  actor_system_config:
    use_auto_config: true
    node_type: COMPUTE
    cpu_count: 8
```

Changing Configurations via CMS

> **Note**
>
> This method of changing configuration is deprecated. The recommended configuration method for V1 is described in the cluster dynamic configuration section.

Get Current Settings

The following command will get current settings for the cluster or tenant.

```
ydbd -s <endpoint> admin console configs load --out-dir <config-folder>
```

```
ydbd -s <endpoint> admin console configs load --out-dir <config-folder> --tenant <tenant-name>
```

Update Settings

First, you need to download the required config as indicated above, then you need to prepare a protobuf file with a change request.

```
Actions {
  AddConfigItem {
    ConfigItem {
      Cookie: "<cookie>"
      UsageScope {
        TenantAndNodeTypeFilter {
          Tenant: "<tenant-name>"
        }
      }
      Config {
          <config-name> {
              <full-config>
          }
      }
    }
  }
}
```

The UsageScope field is optional and is needed to apply settings for a specific tenant.

```
ydbd -s <endpoint> admin console configs update <protobuf-file>
```

Changing Actor System Configuration

An actor system is the basis of YDB. Each component of the system is represented by one or more actors.
Each actor is allocated to a specific ExecutorPool corresponding to the actor's task.
Changing the configuration lets you more accurately distribute the number of cores reserved for each type of task.

Actor System Config Description

The actor system configuration contains an enumeration of ExecutorPools, their mapping to task types, and the actor system scheduler configurations.

The following task types and their respective pools are currently supported:

- System: Designed to perform fast internal YDB operations.
- User: Includes the entire user load for handling and executing incoming requests.
- Batch: Tasks that have no strict limit on the execution time, mainly running background operations.
- IO: Responsible for performing any tasks with blocking operations (for example, writing logs to a file).
- IC: Interconnect, includes all the load associated with communication between nodes.

Each pool is described by the Executor field as shown in the example below.

```
Executor {
  Type: BASIC
  Threads: 9
  SpinThreshold: 1
  Name: "System"
}
```

A summary of the main fields:

- **Type**: Currently, two types are supported, such as **BASIC** and **IO**. All pools, except **IO**, are of the **BASIC** type.
- **Threads**: The number of threads (concurrently running actors) in this pool.
- **SpinThreshold**: The number of CPU cycles before going to sleep if there are no tasks, which a thread running as an actor will take (affects the CPU usage and request latency under low loads).
- **Name**: The pool name to be displayed for the node in Monitoring.

Mapping pools to task types is done by setting the pool sequence number in special fields. Pool numbering starts from 0. Multiple task types can be set for a single pool.

List of fields with their respective tasks:

- **SysExecutor**: System
- **UserExecutor**: User
- **BatchExecutor**: Batch
- **IoExecutor**: IO

Example:

```
SysExecutor: 0
UserExecutor: 1
BatchExecutor: 2
IoExecutor: 3
```

The IC pool is set in a different way, via ServiceExecutor, as shown in the example below.

```
ServiceExecutor {
  ServiceName: "Interconnect"
  ExecutorId: 4
}
```

The actor system scheduler is responsible for the delivery of deferred messages exchanged by actors and is set with the following parameters:

- **Resolution**: The minimum time offset step in microseconds.
- **SpinThreshold**: Similar to the pool parameter, the number of CPU cycles before going to sleep if there are no messages.
- **ProgressThreshold**: The maximum time offset step in microseconds.

If, for an unknown reason, the scheduler thread is stuck, it will send messages according to the lagging time, offsetting it by the **ProgressThreshold** value each time.

We do not recommend changing the scheduler config. You should only change the number of threads in the pool configs.

Example of the default actor system configuration:

```
Executor {
  Type: BASIC
  Threads: 9
  SpinThreshold: 1
  Name: "System"
}
Executor {
  Type: BASIC
  Threads: 16
  SpinThreshold: 1
  Name: "User"
}
Executor {
  Type: BASIC
```

```
  Threads: 7
  SpinThreshold: 1
  Name: "Batch"
}
Executor {
  Type: IO
  Threads: 1
  Name: "IO"
}
Executor {
  Type: BASIC
  Threads: 3
  SpinThreshold: 10
  Name: "IC"
  TimePerMailboxMicroSecs: 100
}
SysExecutor: 0
UserExecutor: 1
IoExecutor: 3
BatchExecutor: 2
ServiceExecutor {
  ServiceName: "Interconnect"
  ExecutorId: 4
}
```

On Static Nodes

Static nodes take the configuration of the actor system from the `/opt/ydb/cfg/config.yaml` file.

After changing the configuration, restart the node.

On Dynamic Nodes

Dynamic nodes take the configuration from the CMS. To change it, you can use the following command:

```
ConfigureRequest {
  Actions {
    AddConfigItem {
      ConfigItem {
        // UsageScope: { ... }
        Config {
          ActorSystemConfig {
            <actor system config>
          }
        }
        MergeStrategy: 3
      }
    }
  }
}
```

```
ydbd -s <endpoint> admin console execute --domain=<domain> --retry=10 actorsystem.txt
```

Expanding a Cluster

You can expand a YDB cluster by adding new nodes to its configuration. Below is the list of actions for expanding a YDB cluster installed manually on VM instances or physical servers. In the Kubernetes environment, clusters are expanded by adjusting the YDB controller settings for Kubernetes.

When expanding your YDB cluster, you do not have to pause user access to databases. When the cluster is expanded, its components are restarted to apply the updated configurations. This means that any transactions that were in progress at the time of expansion may need to be executed again on the cluster. The transactions are rerun automatically because the applications leverage the YDB SDK features for error control and transaction rerun.

Preparing New Servers

If you deploy new static or dynamic nodes of the cluster on new servers added to the expanded YDB cluster, on each new server, you need to install the YDB software according to the procedures described in the cluster deployment instructions. Among other things, you need to:

1. Create an account and a group in the operating system to enable YDB operation.
2. Install the YDB software.
3. Generate the appropriate TLS key and certificate for the software and add them to the server.
4. Copy the up-to-date configuration file for the YDB cluster to the server.

The TLS certificates used on the new servers must meet the requirements for filling out the fields and be signed by the same trusted certification authority that signed the certificates for the existing servers of the expanded YDB cluster.

Adding Dynamic Nodes

By adding dynamic nodes, you can expand the available computing resources (CPU cores and RAM) needed for your YDB cluster to process user queries.

To add a dynamic node to the cluster, run the process that serves this node, passing to it, in the command line options, the name of the served database and the addresses of any three static nodes of the YDB cluster, as shown in the cluster deployment instructions.

Once you have added the dynamic node to the cluster, the information about it becomes available on the cluster monitoring page in the YDB UI.

To remove a dynamic node from the cluster, stop the process on the dynamic node.

Adding Static Nodes

By adding static nodes, you can increase the throughput of your I/O operations and increase the available storage capacity in your YDB cluster.

To add static nodes to the cluster, perform the following steps:

1. Format the disks that will be used to store the YDB data by following the procedure for the cluster deployment step
2. Edit the cluster's configuration file:
   - Add the description of the added nodes (in the `hosts` section) and disks used by them (in the `host_configs` section) to the configuration.
   - Use the `storage_config_generation: K` option to set the ID of the configuration update at the top level, where `K` is the integer update ID (for the initial config, `K=0` or omitted; for the first expansion, `K=1`; for the second expansion, `K=2`; and so on).
3. Copy the updated cluster's configuration file to all the existing and added servers in the cluster, overwriting the old version of the configuration file.
4. Restart all the existing static nodes in the cluster one by one, waiting for each restarted node to initialize and become fully operational.
5. Restart all the existing static nodes in the cluster one by one.
6. Start the processes that serve the new static nodes in the cluster, on the appropriate servers.
7. Make sure that all the new static nodes now show up on the cluster monitoring page in the YDB UI.
8. Issue an authentication token using the YDB CLI, for example:

   ```
   ydb -e grpcs://<node1.ydb.tech>:2135 -d /Root --ca-file ca.crt \
         --user root auth get-token --force >token-file
   ```

   The command example above uses the following options:

   - `node1.ydb.tech`: The FQDN of any server hosting the cluster's static nodes.
   - `2135`: Port number of the gRPCs service for the static nodes.
   - `ca.crt`: Name of the file with the certificate authority certificate.
   - `root`: The name of a user who has administrative rights.
   - `token-file`: name of the file where the authentication token is saved for later use.

   When you run the above command, YDB CLI will request the password to authenticate the given user.
9. Allow the YDB cluster to use disks to store data on the new static nodes. For this, run the following command on any cluster node:

   ```
   export LD_LIBRARY_PATH=/opt/ydb/lib
   /opt/ydb/bin/ydbd -f ydbd-token-file --ca-file ca.crt -s grpcs://`hostname -f`:2135 \
       admin blobstorage config init --yaml-file  /opt/ydb/cfg/config.yaml
   echo $?
   ```

   The command example above uses the following options:

   - `ydbd-token-file`: File name of the previously issued authentication token.
   - `2135`: Port number of the gRPCs service for the static nodes.
   - `ca.crt`: Name of the file with the certificate authority certificate.

If the above command results in the error of the configuration ID mismatch, it means that you made an error editing the `storage_config_generation` field in the cluster configuration file. In the error text, you can find the expected configuration ID that can be used to edit the cluster configuration file. Sample error message for the configuration ID mismatch:

```
ErrorDescription: "ItemConfigGeneration mismatch ItemConfigGenerationProvided# 0 ItemConfigGenerationExp
ected# 1"
```

10. Add additional storage groups to one or more databases by running the following commands on any cluster node:

    ```
    export LD_LIBRARY_PATH=/opt/ydb/lib
    /opt/ydb/bin/ydbd -f ydbd-token-file --ca-file ca.crt -s grpcs://`hostname -f`:2135 \
        admin database /Root/testdb pools add ssd:1
    echo $?
    ```

    The command example above uses the following options:

    - `ydbd-token-file`: File name of the previously issued authentication token.
    - `2135`: Port number of the gRPCs service for the static nodes.
    - `ca.crt`: Name of the file with the certificate authority certificate.
    - `/Root/testdb`: Full path to the database.
    - `ssd:1`: Name of the storage pool and the number of storage groups allocated.

11. Make sure that all the new storage groups now show up on the cluster monitoring page in the YDB UI.

To remove a static node from the YDB cluster, use the documented decommissioning procedure.

If the server running the static cluster node is damaged or becomes irreparable, deploy the unavailable static node on a new server with the same or higher number of disks.

Configuring metadata distribution subsystems State Storage, Board, and Scheme Board

Applies if you need to change the metadata distribution subsystems configuration consisting of State Storage, Board, and Scheme Board on the YDB cluster.

> **Warning**
>
> Errors in metadata subsystem configuration (including the `domains_config` section) or an incorrect sequence of changes can lead to YDB cluster unavailability.

To change the metadata distribution subsystems configuration in the YDB cluster, perform the following steps.

1. Make the required changes to the `domains_config` section of the `config.yaml` configuration file on each node of the YDB cluster:
   For rules on changing the `domains_config` section, see Rules for configuring metadata distribution subsystems.
2. Using the rolling-restart procedure, sequentially restart all nodes of the YDB cluster — static and dynamic: metadata subsystem replicas are placed on static nodes, and tablets on dynamic nodes access them. For more on nodes, see Cluster topology.
   Before restarting the next host, wait for the restart on the previous host to complete and for the node to rejoin the cluster.

Rules for configuring metadata distribution subsystems

The rules listed below apply to `state_storage` and to the separate fields `explicit_state_storage_config`, `explicit_state_storage_board_config`, `explicit_scheme_board_config` in the `domains_config` section of the `config.yaml` file (see State Storage configuration). The `explicit_*` keys correspond individually to State Storage, Board, and Scheme Board.

A ring in the configuration refers to the `ring` block inside the `ring_groups` list element (see State Storage Configuration).

The configuration is changed in several steps. First, a new group of rings consisting of properly selected nodes (according to the failure model) is added, and then the old group of rings is removed.

To avoid cluster unavailability, perform the removal and addition of ring groups strictly in the sequence of steps described below.

1. To change the configuration of the metadata distribution subsystems without cluster unavailability, you must do this by adding and removing ring groups.
2. Only ring groups with the `WriteOnly: true` parameter can be added and removed.
3. The new configuration must always contain at least one ring group from the previous configuration without the `WriteOnly` parameter. Such a ring group must come first in the list.
4. If different ring groups use the same cluster nodes, add the `ring_group_actor_id_offset` parameter to the ring group with a unique value (for example, `1`, `2`, ...). The value must be unique among ring groups.

   This parameter will make the replica identifiers in this ring group unique; they will not match the identifiers from other groups, and this will allow placing several replicas of the same type on one cluster node.
5. The transition to the new configuration is performed in 4 sequential steps. At each step, a new configuration is prepared and applied to the cluster.

   Newly created or ready-to-remove ring groups are marked with the `WriteOnly: true` flag. This is necessary so that read requests are handled by the already deployed ring group while the new configuration spreads to the required number of nodes, new replicas are created, or old ones are removed.

   Therefore, you must pause for at least `1 minute` between steps.

   - Add a new ring group with the `WriteOnly: true` parameter corresponding to the target configuration.
   - Remove flag `WriteOnly`.
   - Set the `WriteOnly: true` flag on the ring group corresponding to the old configuration, and move the new ring group to the beginning of the list of ring groups.
   - Delete the old ring group.

Example

Consider the current configuration as an example:

```
config:
  domains_config:
    explicit_scheme_board_config:
      ring:
        nto_select: 5
        node: [1,2,3,4,5,6,7,8]
```

and the target configuration:

```
config:
  domains_config:
    explicit_scheme_board_config:
      ring:
        nto_select: 5
        node: [10,20,30,40,5,6,7,8]
```

We want to move some of the replicas to other cluster nodes.

Step 1
At the first step, prepare `ring_groups` following the configuration rules: the first ring group matches the **current** configuration from the listings above, the second matches the **target** one and is marked with `WriteOnly: true`. Specify the `ring_group_actor_id_offset` parameter as described in the same rules if the node sets of the groups match.

```
config:
  domains_config:
    explicit_scheme_board_config:
      ring_groups:
        - ring:
          nto_select: 5
          node: [1,2,3,4,5,6,7,8]
        - ring:
          nto_select: 5
          node: [10,20,30,40,5,6,7,8]
          write_only: true
          ring_group_actor_id_offset: 1
```

Step 2
Remove the `WriteOnly` flag.

```
config:
  domains_config:
    explicit_scheme_board_config:
      ring_groups:
        - ring:
          nto_select: 5
          node: [1,2,3,4,5,6,7,8]
        - ring:
          nto_select: 5
          node: [10,20,30,40,5,6,7,8]
          ring_group_actor_id_offset: 1
```

Step 3
Make the new ring group first in the list. Set the `WriteOnly: true` flag on the old configuration.

```
config:
  domains_config:
    explicit_scheme_board_config:
      ring_groups:
        - ring:
          nto_select: 5
          node: [10,20,30,40,5,6,7,8]
          ring_group_actor_id_offset: 1
        - ring:
          nto_select: 5
          node: [1,2,3,4,5,6,7,8]
          write_only: true
```

Step 4
Apply the target configuration to the cluster:

```
config:
  domains_config:
    explicit_scheme_board_config:
      ring_groups:
        - ring:
          nto_select: 5
          node: [10,20,30,40,5,6,7,8]
          ring_group_actor_id_offset: 1
```

Checking the result

You can verify that the changes have been applied in the `CMS` section of the cluster's Embedded UI (available on port 8765): go to the `Tablets` tab and check the replicas of the metadata subsystem tablets to make sure the configuration has been picked up.

State Storage Move

If you need to decommission a YDB cluster host that contains part of State Storage, you need to move it to another host.

> **Warning**
>
> Errors in metadata subsystem configuration (including the `domains_config` section) or an incorrect sequence of changes can lead to YDB cluster unavailability.

As an example, consider a YDB cluster with the following State Storage configuration:

```
...
domains_config:
  ...
  state_storage:
  - ring:
      node: [1, 2, 3, 4, 5, 6, 7, 8, 9]
      nto_select: 9
    ssid: 1
  ...
...
```

On the host with `node_id:1`, a cluster static node is configured and running, which serves part of State Storage. Suppose we need to decommission this host.

To replace `node_id:1`, we added a new host with `node_id:10` to the cluster and deployed a static node on it.

To move State Storage from host `node_id:1` to `node_id:10`:

1. Stop the cluster static nodes on hosts with `node_id:1` and `node_id:10`.

   > **Note**
   >
   > A YDB cluster is fault-tolerant. Temporary node shutdown does not lead to cluster unavailability. For more details, see YDB Cluster Topology.

2. In the configuration file `config.yaml`, change the `node` host list, replacing the identifier of the host being removed with the identifier of the host being added:

   ```
   domains_config:
   ...
     state_storage:
     - ring:
         node: [2, 3, 4, 5, 6, 7, 8, 9, 10]
         nto_select: 9
       ssid: 1
   ...
   ```

3. Update the configuration files `config.yaml` for all cluster nodes, including dynamic ones.
4. Using the rolling-restart procedure, restart all cluster nodes, including dynamic ones, except for static nodes on hosts with `node_id:1` and `node_id:10`. Note that a delay of at least 15 seconds is required between host restarts.
5. Start the cluster static nodes on hosts `node_id:1` and `node_id:10`.

Static Group Move

If you need to decommission a YDB cluster host that contains part of the static group, you need to move it to another host.

> **Warning**
>
> Errors in metadata subsystem configuration (including the `domains_config` section) or an incorrect sequence of changes can lead to YDB cluster unavailability.

As an example, consider a YDB cluster where a static node is configured and running on the host with `node_id:1`. This node serves part of the static group.

Static group configuration fragment:

```
...
blob_storage_config:
  ...
  service_set:
    ...
    groups:
      ...
      rings:
        ...
        fail_domains:
        - vdisk_locations:
          - node_id: 1
            path: /dev/vda
            pdisk_category: SSD
        ...
      ...
    ...
  ...
...
```

To replace `node_id:1`, we added a new host with `node_id:10` to the cluster and deployed a static node on it.

To move part of the static group from host `node_id:1` to `node_id:10`:

1. Stop the cluster static node on the host with `node_id:1`.

   > **Note**
   >
   > A YDB cluster is fault-tolerant. Temporary node shutdown does not lead to cluster unavailability. For more details, see YDB Cluster Topology.

2. In the configuration file `config.yaml`, change the `node_id` value, replacing the identifier of the host being removed with the identifier of the host being added:

   ```
   ...
   blob_storage_config:
     ...
     service_set:
       ...
       groups:
         ...
         rings:
           ...
           fail_domains:
           - vdisk_locations:
             - node_id: 10
               path: /dev/vda
               pdisk_category: SSD
           ...
         ...
       ...
     ...
   ...
   ```

   Change the `path` and disk `pdisk_category` if they differ on the host with `node_id: 10`.
3. Update the configuration files `config.yaml` for all cluster nodes, including dynamic ones.
4. Using the rolling-restart procedure, restart all static cluster nodes.
5. Go to the YDB UI monitoring page and ensure that the static group VDisk appeared on the target physical disk and is replicating. For more details, see Monitoring static groups.
6. Using the rolling-restart procedure, restart all dynamic cluster nodes.

Replacing Node FQDN

Replacing Node FQDN

This procedure describes how to replace the FQDN (Fully Qualified Domain Name) of a YDB cluster node without downtime.

Prerequisites

Note

A YDB cluster is fault-tolerant. Temporary node shutdown does not lead to cluster unavailability. For more details, see YDB Cluster Topology.

Warning

Errors in metadata subsystem configuration (including the `domains_config` section) or an incorrect sequence of changes can lead to YDB cluster unavailability.

Procedure Overview

The FQDN replacement process involves:

1. **Preparation**: Verify cluster health and prepare a new node configuration
2. **Node shutdown**: Gracefully stop the node to be replaced
3. **Configuration update**: Update the cluster configuration with a new FQDN
4. **Node restart**: Start the node with new FQDN
5. **Verification**: Confirm successful FQDN change

Step-by-Step Instructions

Step 1: Verify Cluster Health

Before starting the replacement, ensure the cluster is healthy:

```
ydb monitoring healthcheck
```

Step 2: Prepare New Node Configuration

1. Update DNS records to point the new FQDN to the same IP address
2. Update TLS certificates if they include hostname verification
3. Prepare updated configuration files with the new FQDN

Step 3: Stop the Target Node

Gracefully stop the node that needs FQDN replacement:

```
# For systemd-managed nodes
sudo systemctl stop ydbd-storage

# For manually started nodes
kill -TERM <ydbd_pid>
```

Step 4: Update Cluster Configuration

Update the cluster configuration to reflect the new FQDN:

```
# Example configuration update
hosts:
  - host: new-hostname.example.com  # Updated FQDN
    host_config_id: 1
    port: 19001
    location:
      unit: "1"
      data_center: "DC1"
      rack: "1"
```

Step 5: Apply Configuration Changes

Apply the updated configuration to the cluster:

```
ydb admin config replace --config-file updated-config.yaml
```

Step 6: Start Node with New FQDN

Start the node using the new FQDN:

```
# Update hostname if necessary
sudo hostnamectl set-hostname new-hostname.example.com

# Start the node
sudo systemctl start ydbd-storage
```

Step 7: Verify the Change

Confirm the FQDN change was successful:

```
# Check node status
ydb monitoring healthcheck

# Verify node registration
ydb admin config fetch | grep new-hostname
```

Troubleshooting

Common Issues

1. **DNS resolution problems**: Ensure new FQDN resolves correctly
2. **Certificate validation errors**: Update certificates if they include hostname verification
3. **Node registration failures**: Check network connectivity and firewall rules

Recovery Procedures

If the FQDN replacement fails:

1. Revert DNS changes to the original FQDN
2. Restore the original configuration
3. Restart the node with the original settings
4. Investigate and resolve the underlying issue

Best Practices

1. **Test in staging**: Always test FQDN replacement in a non-production environment first
2. **Backup configurations**: Keep backups of working configurations before making changes
3. **Monitor during change**: Watch cluster health metrics during the replacement process
4. **Document changes**: Maintain records of FQDN changes for future reference
5. **Coordinate with the team**: Ensure all team members are aware of the planned change

Database Node Authentication and Authorization

Database Node Authentication and Authorization

Database node authentication in a YDB cluster ensures verification of database node authenticity when making service calls to other nodes via the gRPC protocol. Node authorization ensures verification and provision of necessary permissions when processing service calls, including operations for registering starting nodes in the cluster and accessing configuration. Using database node authentication and authorization is recommended for all YDB clusters, as it helps avoid situations of unauthorized data access through the inclusion of attacker-controlled nodes in the cluster.

Database node authentication and authorization is performed in the following order:

1. The starting database node opens a gRPC connection to one of the cluster storage nodes specified in the `--node-broker` command-line option. The connection uses the TLS protocol, and the starting node's certificate is used as the client certificate in the connection settings.
2. The storage node and database node perform mutual authenticity verification using the TLS protocol: the certificate trust chain is verified and the hostname correspondence to the "Subject Name" field value of the certificate is checked.
3. The storage node verifies that the "Subject" field of the certificate meets the requirements established by settings in the static configuration.
4. Upon successful completion of the above checks, the connection from the database node is considered authenticated, and it is assigned an access subject identifier — SID, determined by the settings.
5. The database node uses the established gRPC connection to register as part of the cluster using the corresponding service call. During registration, the database node transmits its network address intended for interaction with other cluster nodes.
6. The storage node verifies that the SID assigned to the gRPC connection is in the list of allowed ones. Upon successful completion of this check, the storage node registers the database node in the cluster, mapping the received network address to the node identifier.
7. The database node joins the cluster by connecting through its network address and specifying the node identifier received during registration. Attempts to join the cluster by nodes with unknown network addresses or identifiers are blocked by other nodes.

The following describes the settings necessary to enable database node authentication and authorization.

Configuration Prerequisites

1. The deployed YDB cluster must have gRPC traffic encryption configured using the TLS protocol.
2. When preparing node certificates for a cluster where database node authentication and authorization is planned to be used, it is necessary to ensure uniform rules for filling the "Subject" field of certificates, allowing identification of certificates issued for cluster nodes. More information is provided in the certificate verification rules configuration documentation.

   > **Note**
   >
   > The proposed example script for generating self-signed YDB node certificates fills the "Subject" field with the value `O=YDB` for all node certificates. The configuration examples provided below are prepared for certificates with exactly this "Subject" field content.

3. The command-line parameters for starting database nodes must include options that specify paths to certificate files from trusted certificate authorities, a node certificate, and a node key. The list of additional options is provided in the table below.

Command-line Option	Description
`--grpc-ca`	Path to the certificate file `ca.crt` from a trusted certificate authority.
`--grpc-cert`	Path to the node certificate file `node.crt`.
`--grpc-key`	Path to the node private key file `node.key`.

   Example command for starting a database node with options specifying paths to TLS keys and certificates for the gRPC protocol:

   ```
   /opt/ydb/bin/ydbd server --config-dir /opt/ydb/cfg --tenant /Root/testdb \
       --grpcs-port 2136 --grpc-ca /opt/ydb/certs/ca.crt \
       --grpc-cert /opt/ydb/certs/node.crt --grpc-key /opt/ydb/certs/node.key \
       --ic-port 19002 --ca /opt/ydb/certs/ca.crt \
       --mon-port 8766 --mon-cert /opt/ydb/certs/web.pem \
       --node-broker grpcs://<ydb1>:2135 \
       --node-broker grpcs://<ydb2>:2135 \
       --node-broker grpcs://<ydb3>:2135
   ```

Enabling Database Node Authentication and Authorization

> **Note**
>
> Starting from version 25.2, properly configured database node authentication is mandatory.
>
> Node registration in the cluster is impossible without successful authentication.

To enable mandatory database node authorization, the following configuration blocks must be added to the cluster configuration file:

1. At the root level of the configuration, add a `client_certificate_authorization` block specifying requirements for filling the "Subject" field of trusted certificates of connecting nodes, for example:

   ```
   client_certificate_authorization:
     request_client_certificate: true
     client_certificate_definitions:
       - member_groups: ["registerNode@cert"]
         subject_terms:
   ```

```
      - short_name: "O"
        values: ["YDB"]
```

If necessary, add other certificate checks according to the documentation.

After the certificate is verified and its "Subject" field is confirmed to comply with the requirements established in the `subject_terms` block, the connection will be assigned access subjects from the `member_groups` parameter. To distinguish such access subjects, the `@cert` suffix is added to their names.

2. In the `security_config` section of the cluster configuration, add the `register_dynamic_node_allowed_sids` element with a list of access subjects allowed to register database nodes. Example:

```
security_config:
  enforce_user_token_requirement: true
  ...
  register_dynamic_node_allowed_sids:
  - "registerNode@cert"
```

For more information on configuring cluster authentication parameters, see Configuring Administrative and Other Privileges.

3. In the `auth_config` section of the cluster configuration, add the `node_registration_token` as an empty string. Example:

```
auth_config:
  ...
  node_registration_token: ""
```

4. Update the static configuration files on all cluster nodes manually or using the Ansible playbook.
5. Perform a rolling restart of cluster storage nodes using ydbops or using the Ansible playbook.
6. Perform a rolling restart of cluster database nodes using ydbops or using the Ansible playbook.

Configuration V2

This section of the documentation describes the YDB clusters configuration method called V2, which is the experimental way to configure YDB clusters version v25.1 and above.

> **Warning**
>
> This article is dedicated to YDB clusters that use configuration V2. This configuration method is currently experimental and is only available for YDB versions starting from v25.1. For production use, we recommend choosing configuration V1 — it is the main method and is officially supported for all YDB clusters.

Main materials:

- Configuration V2 Overview
- Updating YDB Cluster Configuration
- Cluster configuration DSL
- Configuration parameters
- Cluster Expansion
- Configuring the State Storage, Board, and Scheme Board metadata distribution subsystems
- State Storage Move
- Static Group Move
- Replacing node FQDN
- Authentication and authorization of database nodes

Configuration V2 Overview

To deploy a new YDB cluster, to add new nodes to an existing cluster, or to change parameters, a configuration is required.

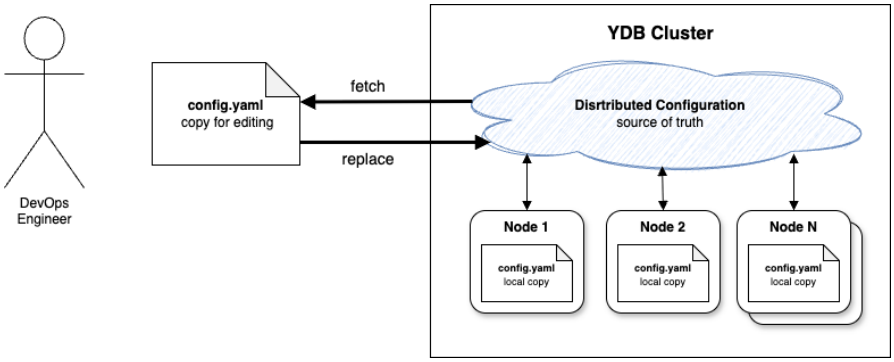
> **Warning**
>
> This article is dedicated to YDB clusters that use configuration V2. This configuration method is currently experimental and is only available for YDB versions starting from v25.1. For production use, we recommend choosing configuration V1 — it is the main method and is officially supported for all YDB clusters.

YDB cluster configuration V2 is a text file in YAML format. In its minimal form, it contains a `config` section with various parameters necessary for starting and configuring cluster nodes, as well as a section with `metadata`. Extended capabilities for flexible configuration are described in the article Cluster Configuration Domain-Specific Language (DSL). You can learn more about available parameters in the configuration reference.

Example configuration V2 file

```
metadata:
  cluster: ""
  version: 0
config:
  hosts:
    - host: localhost
  drive:
    - type: RAM
  grpc_config:
    port: 2136
  monitoring_config:
    monitoring_port: 8765
```

Configuration Management



The YDB cluster itself is responsible for managing the state of the configuration file, and it is also the single source of truth about how it is currently configured. The distributed configuration mechanism is responsible for reliable storage of the current state. You can see the current state of the cluster configuration using the console command ydb admin cluster config fetch, and the state of each specific node through its YDB UI.

Changing the YDB cluster configuration is performed by the administrator as follows:

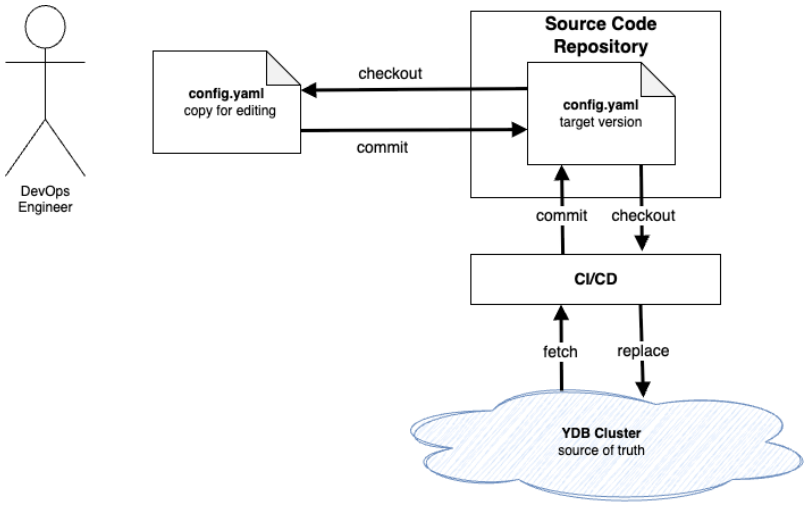
1. Save the current state of the cluster configuration to a local file via ydb admin cluster config fetch.
2. Edit the required parameters in the file in a text editor or any other convenient way.
3. Load changes back to the cluster by calling the ydb admin cluster config replace command.

Configuration change example

```
$ ydb -e grpc://<ydb.example.com>:2135 admin cluster config fetch > config.yaml     # 1
$ vim config.yaml                                                                   # 2
$ ydb -e grpc://<ydb.example.com>:2135 admin cluster config replace -f config.yaml  # 3
```

Loading changes back to the cluster is not always successful. In addition to basic validation of a configuration file, the system has protection against concurrent changes by multiple administrators. The system increments the `metadata.version` field with each configuration change and refuses to accept a new version if its number does not match the expected one, as this means that there was another change between `fetch` and `replace`, and `replace` would erase it. To minimize such conflicts, you can use the "Infrastructure as Code" approach: store a copy of the configuration file in a version control system repository (for example, Git) and run `fetch` and `replace` commands only from a continuous integration and delivery (CI/CD) system linked to this repository, reacting to changes in the YDB configuration file in the repository and ensuring sequential sending of all changes to the YDB cluster.

Infrastructure as Code approach diagram



Each YDB cluster node saves a local copy of the configuration to the directory specified in the `ydbd --config-dir` startup argument. This local copy is used in the following situations:

1. To apply settings that are needed at the very start of the node's operation, even before it has the ability to start communicating with other cluster nodes. Changing such settings may require restarting the node.
2. For initial deployment and expansion of the cluster.
3. In case of force majeure, if problems arose with the main configuration management mechanism that require manual intervention.

The above describes the main mechanism for managing YDB configuration V2. Depending on the preferred infrastructure management method, additional automation may be provided.

Basic Configuration Usage Scenarios

Initial YDB Cluster Deployment

For cluster configuration during initial deployment, it is recommended to use instructions for the selected infrastructure management method:

- Deploying a YDB cluster with Ansible;
- Getting Started with YDB in Kubernetes;
- Deploying YDB Cluster Manually.

Configuration Update

To update the configuration of an already deployed cluster, you need to use the appropriate commands depending on the deployment method:

- Updating Configuration of YDB Clusters Deployed with Ansible;
- Updating configuration of manually deployed YDB clusters.

If configuration changes affect parameters that require restarting cluster nodes, use the rolling restart procedure. More details about it depending on the deployment method:

- Restarting a cluster deployed with Ansible;
- Restarting a manually deployed cluster.

Cluster Expansion

When expanding the cluster, configuration is delivered to old and new nodes differently:

- To nodes that existed before expansion, changes are delivered automatically when calling ydb admin cluster config replace.
- Before the first start of new nodes, a local copy is delivered by the special command ydb admin node config init, not by the node itself.

See Also

- Configuration parameters reference
- Comparing YDB Cluster Configurations: V1 and V2

Updating YDB Cluster Configuration

This article covers changing cluster configuration after initial deployment.

> **Warning**
>
> This article is dedicated to YDB clusters that use configuration V2. This configuration method is currently experimental and is only available for YDB versions starting from v25.1. For production use, we recommend choosing configuration V1 — it is the main method and is officially supported for all YDB clusters.

YDB cluster configuration management is performed using YDB CLI. The standard approach to updating a configuration is to get the current configuration from the cluster using YDB CLI, modify it locally, and then load the updated configuration back to the cluster.

To prevent accidental overwriting of changes when multiple administrators work simultaneously, YDB configuration contains metadata with a version number. With each successful configuration upload, this number is automatically incremented. If another administrator tries to upload a configuration based on a previous (outdated) version (i.e., with the same or lower version number in metadata), the system will reject this attempt. This ensures that changes by one administrator will not be silently overwritten by changes from another.

To update a configuration, you need to get the current configuration, modify it, and apply the updated configuration. During the configuration application process, it will be reliably saved by the cluster and distributed to all its nodes. As configuration is delivered to cluster nodes, the configuration begins to be applied by these nodes. Some configuration parameters change node behavior immediately after configuration delivery, while others only take effect when the node starts, and changing such parameters requires a cluster restart for them to take effect.

Basic Configuration Operations

Getting Current Configuration

To get the current cluster configuration, use the command:

```
ydb -e grpcs://<endpoint>:2135 admin cluster config fetch > config.yaml
```

Specify the address of any cluster node as `<endpoint>`.

Applying New Configuration

To load updated configuration to the cluster, use the following command:

```
ydb -e grpcs://<endpoint>:2135 admin cluster config replace -f config.yaml
```

Specify the address of any cluster node as `<endpoint>`.

Some configuration parameters are applied on the fly after executing the command, however some require performing the cluster restart procedure.

Cluster Configuration Domain-Specific Language (DSL)

Cluster Configuration DSL

Selectors

The main entity of the DSL is **selectors**. They allow the overriding of parts of the configuration or the entire configuration for specific nodes or groups of nodes. For example, they can be used to enable experimental functionality for nodes of a particular database. Each selector is an array of overrides and extensions to the main configuration. Each selector has a `description` field, which can be used to store an arbitrary description string. The `selector` field represents a set of rules that determine whether the selector should be applied to a specific node based on a set of labels. The `config` field describes the override rules. Selectors are applied in the order they are defined.

Labels

Labels are special tags used to mark nodes or groups of nodes. Each node has a set of automatically assigned labels:

- `node_id` — the internal identifier of the node in the system
- `node_host` — the node's `hostname` obtained at startup
- `tenant` — the database served by this node
- `dynamic` — whether this node is dynamic (true/false)

Additionally, the user can explicitly define any additional labels when starting the `ydbd` process on the node using command-line arguments, such as `--label example=test`.

Example of Using Selectors

The example below defines the actor system's general configuration and the tenant `large_tenant` configuration. By default, with such a configuration, the actor system assumes that each node has 4 cores, while nodes of the `large_tenant` have 16 cores. The actor system's node type is overridden to `COMPUTE`.

```
metadata:
  cluster: ""
  version: 8
config:
  actor_system_config:
    use_auto_config: true
    node_type: STORAGE
    cpu_count: 4

# This section is used as a hint when generating possible configurations using the resolve command
allowed_labels:
  dynamic:
    type: string

selector_config:
- description: large_tenant has bigger nodes with 16 cpu # arbitrary description string
  selector: # selector for all nodes of the tenant large_tenant
    tenant: large_tenant
  config:
    actor_system_config: !inherit # reuse the original actor_system_config, the semantics of !inherit are des
cribed in the section below
      # in this case, !inherit allows managing the actor_system_config.use_auto_config parameter for the enti
re cluster by changing only the base setting
      cpu_count: 16
      node_type: COMPUTE
```

Permissive Labels

A mapping in which you can set the allowable values for labels. This section is used as a hint when generating possible configurations using the resolve command. Values are not validated at node startup.

There are two types of labels available:

- string;
- enum.

string

It can take any value or be unset.

Example:

```
dynamic:
  type: string
host_name:
  type: string
```

enum

It can take values from the `values` list or be unset.

Example:

```
flavour:
  type: enum
  values:
    ? small
```

```
    ? medium
    ? big
```

Selector Behavior

Selectors represent a simple predicate language. Selectors for each label are combined using the **AND** condition.

Simple Selector

The following selector will select nodes where the `label1` is equal to `value1` **and** the `label2` is equal to `value2`:

```
selector:
  label1: value1
  label2: value2
```

The following selector will select **ALL** nodes in the cluster, as no conditions are specified:

```
selector: {}
```

In

This operator allows for selecting nodes with label values from a list.

The following selector will select all nodes where `label1` is equal to `value1` **or** `value2`:

```
selector:
  label1:
    in:
    - value1
    - value2
```

NotIn

This operator allows selecting nodes where the chosen label does not match any value from a list.

The following selector will select all nodes where `label1` is equal to `value1` **and** `label2` is not equal to `value2` **and** `value3`:

```
selector:
  label1: value1
  label2:
    not_in:
    - value2
    - value3
```

Additional YAML Tags

Tags are necessary for partial or complete reuse of configurations from previous selectors. They allow you to merge, extend, delete, and override parameters set in previous selectors and the main configuration.

!inherit

Scope: YAML mapping
Action: similar to the merge tag in YAML, copy all child elements from the parent mapping and merge with the current ones, overwriting them.
Example:

Original configuration	Override	Resulting configuration

`config:`
`  some_config:`
`  first_entry: 1`
`  second_entry: 2`
`  third_entry: 3`
	`config:`
	`  some_config: !inherit`
	`  second_entry: 100`
		`config:`
		`  some_config:`
		`  first_entry: 1`
		`  second_entry: 100`
		`  third_entry: 3`

!inherit:<key>

Scope: YAML sequence
Action: copy elements from the parent array and overwrite, treating the `key` object in the elements as the key, appending new keys to the end.
Example:

Original configuration	Override	Resulting configuration

`config:`
`  some_config:`
`  array:`
`  - abc: 2`
`  value: 10`
`  - abc: 1`
`  value: 20`
`  another_value: test`
	`config:`
	`  some_config: !inherit`
	`  array: !inherit:abc`
	`  - abc: 1`
	`  value: 30`
	`  - abc: 3`
	`  value: 40`
		`config:`
		`  some_config:`
		`  array:`
		`  - abc: 2`
		`  value: 10`
		`  - abc: 1`
		`  value: 30`
		`  - abc: 3`
		`  value: 40`

!remove

Scope: YAML sequence element under `!inherit:<key>`
Action: remove the element with the corresponding key.
Example:

Original configuration	Override	Resulting configuration

`config:`
`  some_config:`
`  array:`
`  - abc: 2`
`  value: 10`
`  - abc: 1`
`  value: 20`
`  another_value: test`
	`config:`
	`  some_config: !inherit`
	`  array: !inherit:abc`
	`  - !remove`
	`  abc: 1`
		`config:`
		`  some_config:`
		`  array:`
		`  - abc: 2`
		`  value: 10`

!append

Scope: YAML sequence
Action: copy elements from the parent array and append new ones to the end.
Example:

Original configuration	Override	Resulting configuration

`config:`
`  some_config:`
`  array:`
`  - abc: 2`
`  value: 10`
`  - abc: 1`
`  value: 20`
`  another_value: test`
	`config:`
	`  some_config: !inherit`
	`  array: !append`
	`  - abc: 1`
	`  value: 30`
	`  - abc: 3`
	`  value: 40`
		`config:`
		`  some_config:`
		`  array:`
		`  - abc: 2`
		`  value: 10`
		`  - abc: 1`
		`  value: 20`
		`  another_value: test`
		`  - abc: 1`
		`  value: 30`
		`  - abc: 3`
		`  value: 40`

Generating Final Configurations

Configurations can contain complex sets of overrides. With the YDB CLI, you can view the final configurations for:

- specific nodes
- sets of labels
- all possible combinations for the current configuration

```
# Generate all possible final configurations for cluster.yaml
ydb admin config resolve --all -f cluster.yaml
# Generate the configuration for cluster.yaml with labels tenant=/Root/test and canary=true
ydb admin config resolve -f cluster.yaml --label tenant=/Root/test --label canary=true
# Generate the configuration for cluster.yaml with labels similar to those on node 1001
ydb admin config resolve -f cluster.yaml --node_id 1001
# Take the current cluster configuration and generate the final configuration for it with labels similar to t
hose on node 1001
ydb admin config resolve --from-cluster --node_id 1001
```

The configuration transformation command is described in more detail in the section Fetch the cluster configuration.

Example output of `ydb admin config resolve --all -f cluster.yaml` for the following configuration file:

```
metadata:
  cluster: ""
  version: 8
config:
  actor_system_config:
    use_auto_config: true
    node_type: STORAGE
    cpu_count: 4
allowed_labels:
  dynamic:
    type: string
selector_config:
- description: Actorsystem for dynnodes # arbitrary description string
  selector: # selector for all nodes with label dynamic = true
    dynamic: true
  config:
    actor_system_config: !inherit # reuse the original actor_system_config, the semantics of !inherit are des
cribed in the section below
      node_type: COMPUTE
      cpu_count: 8
```

Output:

```
---
label_sets: # sets of labels for which the configuration is generated
- dynamic:
    type: NOT_SET # one of three label types: NOT_SET | COMMON | EMPTY
config: # generated configuration
  invalid: 1
  actor_system_config:
    use_auto_config: true
    node_type: STORAGE
    cpu_count: 4
---
label_sets:
- dynamic:
    type: COMMON
    value: true # label value
config:
  invalid: 1
  actor_system_config:
    use_auto_config: true
    node_type: COMPUTE
    cpu_count: 8
```

YDB Cluster Configuration

The cluster configuration is specified in the YAML file passed in the `--yaml-config` parameter when the cluster nodes are run. This article provides an overview of the main configuration sections and links to detailed documentation for each section.

Each configuration section serves a specific purpose in defining how the YDB cluster operates, from hardware resource allocation to security settings and feature flags. The configuration is organized into logical groups that correspond to different aspects of cluster management and operation.

Configuration Sections

The following top-level configuration sections are available, listed in alphabetical order:

Section	Required	Description
actor_system_config	Yes	CPU resource allocation across actor system pools
auth_config	No	Authentication and authorization settings
aws_client_config	No	AWS client default settings
blob_storage_config	No	Static cluster group configuration for system tablets
bridge_config	No	Cluster piles for bridge mode
client_certificate_authorization	No	Client certificate authentication
cms_config	No	Cluster Management System configuration
domains_config	No	Cluster domain configuration including Blob Storage and State Storage
feature_flags	No	Feature flags to enable or disable specific YDB features
healthcheck_config	No	Health check service thresholds and timeout settings
hive_config	No	Hive component configuration for tablet management
host_configs	No	Typical host configurations for cluster nodes
hosts	Yes	Static cluster nodes configuration
kafka_proxy_config	No	Kafka Proxy configuration
log_config	No	Logging configuration and parameters
memory_controller_config	No	Memory allocation and limits for database components
node_broker_config	No	Stable node names configuration
query_service_config	No	Federated query connector configuration
resource_broker_config	No	Resource broker for controlling CPU and memory consumption
security_config	No	Security configuration settings
table_service_config	No	Query processing configuration
tli_config	No	Transaction lock invalidation (TLI) diagnostics parameters
tls	No	TLS configuration for secure connections

Practical Guidelines

While this documentation section focuses on complete reference documentation for available settings, practical recommendations on what to tune and when can be found in the following places:

- As part of the initial YDB cluster deployment:
  - Ansible
  - Kubernetes
  - Manual
- As part of troubleshooting
- As part of security hardening

Sample Cluster Configurations

You can find model cluster configurations for deployment in the repository. Check them out before deploying a cluster.

Cluster Expansion

You can expand a YDB cluster by adding new nodes to the cluster configuration. Below are the necessary actions for expanding a YDB cluster installed manually on virtual machines or physical servers. Cluster expansion in a Kubernetes environment is performed by adjusting the YDB controller settings for Kubernetes.

Expanding a YDB cluster does not require suspending user access to databases. When expanding the cluster, its components are restarted to apply configuration changes, which in turn may lead to the need to retry transactions running on the cluster. Transaction retries are performed automatically by applications using YDB SDK capabilities for error control and operation retry.

Preparing New Servers

When placing new static or dynamic cluster nodes on new servers that were not previously part of the expanded YDB cluster, each new server must install YDB software according to the procedures described in the cluster deployment instructions. In particular, it is necessary to:

1. create an account and group in the operating system for the YDB service
2. install YDB software
3. prepare and place the corresponding TLS key and certificate on the server
4. copy the current YDB cluster configuration file to the server

TLS certificates used on new servers must comply with field filling requirements, and be signed by a trusted certificate authority used on existing servers of the expanded YDB cluster.

Adding Dynamic Nodes

Adding dynamic nodes allows increasing available computing resources (processor cores and RAM) for executing user queries by the YDB cluster.

To add a dynamic node to the cluster, it is sufficient to start the process serving this node, passing it the path to the cluster configuration folder, the name of the served database, and addresses of any three static cluster nodes in the command line parameters, as shown in the cluster deployment instructions.

After successfully adding a dynamic node to the cluster, information about it will be available on the cluster monitoring page in the YDB UI.

To remove a dynamic node from the cluster, it is sufficient to stop the dynamic node process.

Adding Static Nodes

Adding static nodes allows increasing throughput when performing input-output operations and increasing available capacity for data storage in the YDB cluster.

To add static nodes to the cluster, you need to perform the following sequence of actions:

1. Format disks that will be used for storing YDB data using the procedure described for the cluster deployment stage.
2. Obtain an authentication token for executing administrative commands using YDB CLI, for example:

   ```
   ydb -e grpcs://<node1.ydb.tech>:2135 -d /Root --ca-file ca.crt \
       --user root auth get-token --force > token-file
   ```

   The example command above uses the following parameters:

   - `node1.ydb.tech` — FQDN of any server hosting static cluster nodes
   - `2135` — grpcs service port number for static nodes
   - `ca.crt` — certificate authority certificate file name
   - `root` — username with administrative rights
   - `token-file` — name of the file where the authentication token is saved for subsequent use

   When executing the above command, YDB CLI will request a password for authenticating the specified user.
3. Get the current cluster configuration by executing the following command on any cluster node:

   ```
   ydb --token-file token-file --ca-file ca.crt -e grpcs://<node1.ydb.tech>:2135 \
       admin cluster config fetch > config.yaml
   ```

4. Correct the cluster configuration file, including in the configuration a description of the added nodes (in the `hosts` section) and disks used on them (in the `host_configs` section).
5. Allow the YDB cluster to use disks on new static nodes for data storage by executing the following command on any cluster node:

   ```
   export LD_LIBRARY_PATH=/opt/ydb/lib
   ydb --token-file ydbd-token-file --ca-file ca.crt -e grpcs://<node1.ydb.tech>:2135 \
       admin cluster config replace -f config.yaml
   echo $?
   ```

   The example command above uses the following parameters:

   - `ydbd-token-file` — name of the previously obtained authentication token file
   - `2135` — grpcs service port number for static nodes
   - `ca.crt` — certificate authority certificate file name

   If the above command returns a configuration version verification error, this means that the current config version is outdated, and you need to get a new one from the cluster by repeating step 3. Example configuration version verification error message:

   ```
   ErrorDescription: "ConfigVersion mismatch ConfigVersionProvided# 0 ConfigVersionExpected# 1"
   ```

6. Create an empty directory `/opt/ydb/cfg` on each new machine for the cluster to work with configuration. If multiple nodes are started on one machine, use the same directory. By executing a special command on each new machine, initialize this directory using any static cluster node as the configuration source.

   ```
   sudo mkdir -p /opt/ydb/cfg
   sudo chown -R ydb:ydb /opt/ydb/cfg
   ydb admin node config init --config-dir /opt/ydb/cfg --seed-node <node.ydb.tech:2135>
   ```

7. Start processes serving new static cluster nodes on the corresponding servers.
8. Ensure that new static nodes are displayed on the cluster monitoring page in the YDB UI.
9. Add additional storage groups to one or more databases by executing commands of the following type on any cluster node:

   ```
   export LD_LIBRARY_PATH=/opt/ydb/lib
   /opt/ydb/bin/ydbd -f ydbd-token-file --ca-file ca.crt -s grpcs://`hostname -f`:2135 \
       admin database /Root/testdb pools add ssd:1
   echo $?
   ```

   The example command above uses the following parameters:

   - `ydbd-token-file` - name of the previously obtained authentication token file
   - `2135` - grpcs service port number for static nodes
   - `ca.crt` - certificate authority certificate file name
   - `/Root/testdb` - full path to the database
   - `ssd:1` - storage pool name and number of allocated storage groups

10. Ensure that added storage groups are displayed on the cluster monitoring page in the YDB UI.

Removing static nodes from the YDB cluster is performed according to the documented decommissioning procedure.

In case of damage and inability to repair a server running a static cluster node, it is necessary to place the unavailable static node on a new server containing a similar or greater number and volume of disks.

Configuring the State Storage, Board, and Scheme Board metadata distribution subsystems

Applies if you need to change the metadata distribution subsystem configuration consisting of State Storage, Board, and Scheme Board on a YDB cluster.

When using configuration V2, the metadata distribution subsystems are partially supported automatically through the SelfHeal mechanism — see Metadata Distribution SelfHeal (moving and adding replicas on topology changes). To disable this behavior, set `state_storage_self_heal_config.enable` to `false`, as described in the same section. Disabling is not required for the steps below: the configuration after `ydb admin cluster config replace` will be applied before the next automatic trigger.

> **Warning**
>
> Errors in metadata subsystem configuration (including the `domains_config` section) or an incorrect sequence of changes can lead to YDB cluster unavailability.

To manually change the State Storage configuration in a YDB cluster, perform the following steps.

1. Get the current cluster configuration using the ydb admin cluster config fetch command:

```
ydb [global options...] admin cluster config fetch --v2-internal-state > config.yaml
```

As a result of running this command, the current configuration will be saved to the file `config.yaml`.

2. Make the required changes to the section `domains_config` of the configuration file `config.yaml`:
   For rules on changing the `domains_config` section, see State Storage configuration rules.
3. Apply the new cluster configuration using the ydb admin cluster config replace command:

```
ydb [global options...] admin cluster config replace -f config.yaml
```

State Storage configuration rules

The rules listed below apply to `state_storage` and to the separate fields `explicit_state_storage_config`, `explicit_state_storage_board_config`, `explicit_scheme_board_config` in the section `domains_config` of the file `config.yaml` (see State Storage configuration). The `explicit_*` keys correspond individually to State Storage, Board, and Scheme Board.

In the configuration, a ring refers to a `ring` block inside a `ring_groups` list element (see State Storage configuration).

Configuration changes are performed in several steps. First, a new group of rings is added, consisting of properly selected nodes (according to the failure model), and then the old group of rings is removed.

To avoid cluster unavailability, add and remove ring groups strictly in the sequence of steps described below.

1. To change the configuration of metadata distribution subsystems without cluster unavailability, do so by adding and removing ring groups.
2. You can only add and remove ring groups with the `WriteOnly: true` parameter.
3. The new configuration must always contain at least one ring group from the previous configuration without the `WriteOnly` parameter. Such a ring group must be first in the list.
4. If different ring groups use the same cluster nodes, add the `ring_group_actor_id_offset:1` parameter to the ring group. The value must be unique among ring groups.

   This parameter makes replica identifiers unique within a given ring group; they will not match identifiers from other groups, allowing multiple replicas of the same type to be placed on a single cluster node.
5. The transition to the new configuration is performed in 4 sequential steps. At each step, a new configuration is prepared and applied to the cluster.

   Newly created or ready-to-remove ring groups are marked with the `WriteOnly: true` flag. This is necessary so that read requests are handled by the already deployed ring group while the new configuration spreads to the required number of nodes, new replicas are created, or old ones are removed.

   Therefore, pause for at least `1 minute` between steps.

   - Add a new ring group with the `WriteOnly: true` parameter corresponding to the target configuration.
   - Remove the `WriteOnly` flag.
   - Set the `WriteOnly: true` flag on the ring group corresponding to the old configuration, and move the new ring group to the beginning of the ring group list.
   - Remove the old ring group.

Example

Consider the current configuration as an example:

```
config:
  domains_config:
    explicit_scheme_board_config:
      ring:
        nto_select: 5
        node: [1,2,3,4,5,6,7,8]
```

and the target configuration:

```
config:
  domains_config:
    explicit_scheme_board_config:
```

```
      ring:
        nto_select: 5
        node: [10,20,30,40,5,6,7,8]
```

We want to move some replicas to other cluster nodes.

Step 1
At the first step, prepare `ring_groups` following the configuration rules: the first ring group matches the **current** configuration from the listings above, the second matches the **target** one and is marked with `WriteOnly: true`. Specify the `ring_group_actor_id_offset` parameter as described in the same rules if the node sets of the groups match.

```
config:
  domains_config:
    explicit_scheme_board_config:
      ring_groups:
        - ring:
          nto_select: 5
          node: [1,2,3,4,5,6,7,8]
        - ring:
          nto_select: 5
          node: [10,20,30,40,5,6,7,8]
          write_only: true
          ring_group_actor_id_offset: 1
```

Step 2
Remove the `WriteOnly` flag.

```
config:
  domains_config:
    explicit_scheme_board_config:
      ring_groups:
        - ring:
          nto_select: 5
          node: [1,2,3,4,5,6,7,8]
        - ring:
          nto_select: 5
          node: [10,20,30,40,5,6,7,8]
          ring_group_actor_id_offset: 1
```

Step 3
Make the new ring group first in the list. Set the `WriteOnly: true` flag on the old configuration.

```
config:
  domains_config:
    explicit_scheme_board_config:
      ring_groups:
        - ring:
          nto_select: 5
          node: [10,20,30,40,5,6,7,8]
          ring_group_actor_id_offset: 1
        - ring:
          nto_select: 5
          node: [1,2,3,4,5,6,7,8]
          write_only: true
```

Step 4
Apply the target configuration to the cluster:

```
config:
  domains_config:
    explicit_scheme_board_config:
      ring_groups:
        - ring:
          nto_select: 5
          node: [10,20,30,40,5,6,7,8]
          ring_group_actor_id_offset: 1
```

Verifying the result

You can verify that the changes have been applied in the `CMS` section of the cluster's Embedded UI (available on port 8765): go to the `Tablets` tab and check the replicas of the metadata subsystem tablets to confirm that the configuration has been picked up.

State Storage Move

Article under development

> **Warning**
>
> This article is dedicated to YDB clusters that use configuration V2. This configuration method is currently experimental and is only available for YDB versions starting from v25.1. For production use, we recommend choosing configuration V1 — it is the main method and is officially supported for all YDB clusters.

If you need to decommission a YDB cluster node that contains part of State Storage, you need to move it to another node.

> **Warning**
>
> Errors in metadata subsystem configuration (including the `domains_config` section) or an incorrect sequence of changes can lead to YDB cluster unavailability.

When using Configuration V2, State Storage configuration is performed automatically. To make changes to the configuration, you need to perform the following actions: stop automatic State Storage configuration management, get the current State Storage configuration, change the current configuration in the desired way and specify it explicitly as the target, then apply it, ensure it is applied, perform node decommissioning, remove explicit State Storage configuration specification and enable automatic State Storage configuration.

As an example, consider a YDB cluster where node 1 is part of State Storage, and node 10 is not, and we will assume that the goal of changing the State Storage configuration is to decommission node 1.

1. Stop automatic State Storage configuration management
2. Get the current State Storage configuration
3. Change the current configuration in the desired way and specify it explicitly as the target

   Suppose the following State Storage configuration is obtained:

   ```
   ...
     state_storage:
     - ring:
         node: [1, 2, 3, 4, 5, 6, 7, 8, 9]
         nto_select: 9
       ssid: 1
   ...
   ```

   A static node is configured and running on the host with `node_id:1`, which is a part of State Storage. Suppose we need to decommission this host.

   To replace `node_id:1`, we use another host with a static node deployed on it with `node_id:10`.

   To move State Storage from node `node_id:1` to `node_id:10`, in the configuration file `config.yaml` change the `node` host list, replacing the identifier of the node to be removed with the identifier of the node to be added:

   ```
   ...
     state_storage:
     - ring:
         node: [10, 2, 3, 4, 5, 6, 7, 8, 9]
         nto_select: 9
       ssid: 1
   ...
   ```

4. Apply the configuration.
5. Ensure the configuration is applied.
   Note that the full application of new State Storage nodes after reconfiguration occurs with a delay of at least 15 seconds.
6. Perform node decommissioning.
7. Remove explicit State Storage configuration specification.
8. Enable automatic State Storage configuration.

Static Group Move

Article under development

> **Warning**
>
> This article is dedicated to YDB clusters that use configuration V2. This configuration method is currently experimental and is only available for YDB versions starting from v25.1. For production use, we recommend choosing configuration V1 — it is the main method and is officially supported for all YDB clusters.

When using Configuration V2, static group management is performed automatically and the SelfHeal mechanism will perform reconfiguration when one static group node fails.

When manual static group configuration management is needed, disable automatic static group management, get the current static group configuration, make changes and apply the modified configuration as the target static group configuration. Then remove the target static group configuration from the configuration file and enable automatic static group configuration management.

> **Warning**
>
> Errors in metadata subsystem configuration (including the `domains_config` section) or an incorrect sequence of changes can lead to YDB cluster unavailability.

As an example, consider a YDB cluster where a static node is configured and running on the host with `node_id:1`. This node serves part of the static group.

Static group configuration fragment:

```
...
groups:
  ...
  rings:
    ...
    fail_domains:
    - vdisk_locations:
      - node_id: 1
        path: /dev/vda
        pdisk_category: SSD
    ...
...
```

To replace `node_id:1`, use another host with a static node deployed on it with `node_id:10`.

To move part of the static group from host `node_id:1` to `node_id:10`:

1. Disable automatic static group management.
2. Get the current static group configuration.
3. Make changes and apply the modified configuration as the target static group configuration.

   In the configuration file `config.yaml`, change the `node_id` value, replacing the identifier of the host being removed with the identifier of the host being added:

   ```
   ...
   groups:
     ...
     rings:
       ...
       fail_domains:
       - vdisk_locations:
         - node_id: 10
           path: /dev/vda
           pdisk_category: SSD
       ...
   ...
   ```

   Change the `path` and disk `pdisk_category` if they differ on the host with `node_id: 10`.
4. Go to the YDB UI monitoring page and ensure that the static group VDisk appeared on the target physical disk and is replicating. For more details, see Monitoring static groups.
5. Remove the target static group configuration from the configuration file and enable automatic static group configuration management.

Replacing Node FQDN

This procedure describes how to replace the FQDN (Fully Qualified Domain Name) of a YDB cluster node without downtime.

Prerequisites

> **Note**
>
> A YDB cluster is fault-tolerant. Temporary node shutdown does not lead to cluster unavailability. For more details, see YDB Cluster Topology.

> **Warning**
>
> Errors in metadata subsystem configuration (including the `domains_config` section) or an incorrect sequence of changes can lead to YDB cluster unavailability.

Procedure Overview

The FQDN replacement process involves:

1. **Preparation**: Verify cluster health and prepare a new node configuration
2. **Node shutdown**: Gracefully stop the node to be replaced
3. **Configuration update**: Update the cluster configuration with a new FQDN
4. **Node restart**: Start the node with new FQDN
5. **Verification**: Confirm successful FQDN change

Step-by-Step Instructions

Step 1: Verify Cluster Health

Before starting the replacement, ensure the cluster is healthy:

```
ydb monitoring healthcheck
```

Step 2: Prepare New Node Configuration

1. Update DNS records to point the new FQDN to the same IP address
2. Update TLS certificates if they include hostname verification
3. Prepare updated configuration files with the new FQDN

Step 3: Stop the Target Node

Gracefully stop the node that needs FQDN replacement:

```
# For systemd-managed nodes
sudo systemctl stop ydbd-storage

# For manually started nodes
kill -TERM <ydbd_pid>
```

Step 4: Update Cluster Configuration

Update the cluster configuration to reflect the new FQDN:

```
# Example configuration update
hosts:
  - host: new-hostname.example.com  # Updated FQDN
    host_config_id: 1
    port: 19001
    location:
      unit: "1"
      data_center: "DC1"
      rack: "1"
```

Step 5: Apply Configuration Changes

Apply the updated configuration to the cluster:

```
ydb admin config replace --config-file updated-config.yaml
```

Step 6: Start Node with New FQDN

Start the node using the new FQDN:

```
# Update hostname if necessary
sudo hostnamectl set-hostname new-hostname.example.com

# Start the node
sudo systemctl start ydbd-storage
```

Step 7: Verify the Change

Confirm the FQDN change was successful:

```
# Check node status
ydb monitoring healthcheck

# Verify node registration
ydb admin config fetch | grep new-hostname
```

Troubleshooting

Common Issues

1. **DNS resolution problems**: Ensure new FQDN resolves correctly
2. **Certificate validation errors**: Update certificates if they include hostname verification
3. **Node registration failures**: Check network connectivity and firewall rules

Recovery Procedures

If the FQDN replacement fails:

1. Revert DNS changes to the original FQDN
2. Restore the original configuration
3. Restart the node with the original settings
4. Investigate and resolve the underlying issue

Best Practices

1. **Test in staging**: Always test FQDN replacement in a non-production environment first
2. **Backup configurations**: Keep backups of working configurations before making changes
3. **Monitor during change**: Watch cluster health metrics during the replacement process
4. **Document changes**: Maintain records of FQDN changes for future reference
5. **Coordinate with the team**: Ensure all team members are aware of the planned change

Database Node Authentication and Authorization

Database node authentication in a YDB cluster ensures verification of database node authenticity when making service calls to other nodes via the gRPC protocol. Node authorization ensures verification and provision of necessary permissions when processing service calls, including operations for registering starting nodes in the cluster and accessing configuration. Using database node authentication and authorization is recommended for all YDB clusters, as it helps avoid situations of unauthorized data access through the inclusion of attacker-controlled nodes in the cluster.

Database node authentication and authorization is performed in the following order:

1. The starting database node opens a gRPC connection to one of the cluster storage nodes specified in the `--node-broker` command-line option. The connection uses the TLS protocol, and the starting node's certificate is used as the client certificate in the connection settings.
2. The storage node and database node perform mutual authenticity verification using the TLS protocol: the certificate trust chain is verified and the hostname correspondence to the "Subject Name" field value of the certificate is checked.
3. The storage node verifies that the "Subject" field of the certificate meets the requirements established by settings in the static configuration.
4. Upon successful completion of the above checks, the connection from the database node is considered authenticated, and it is assigned an access subject identifier — SID, determined by the settings.
5. The database node uses the established gRPC connection to register as part of the cluster using the corresponding service call. During registration, the database node transmits its network address intended for interaction with other cluster nodes.
6. The storage node verifies that the SID assigned to the gRPC connection is in the list of allowed ones. Upon successful completion of this check, the storage node registers the database node in the cluster, mapping the received network address to the node identifier.
7. The database node joins the cluster by connecting through its network address and specifying the node identifier received during registration. Attempts to join the cluster by nodes with unknown network addresses or identifiers are blocked by other nodes.

The following describes the settings necessary to enable database node authentication and authorization.

Configuration Prerequisites

1. The deployed YDB cluster must have gRPC traffic encryption configured using the TLS protocol.
2. When preparing node certificates for a cluster where database node authentication and authorization is planned to be used, it is necessary to ensure uniform rules for filling the "Subject" field of certificates, allowing identification of certificates issued for cluster nodes. More information is provided in the certificate verification rules configuration documentation.

   > **Note**
   >
   > The proposed example script for generating self-signed YDB node certificates fills the "Subject" field with the value `O=YDB` for all node certificates. The configuration examples provided below are prepared for certificates with exactly this "Subject" field content.

3. The command-line parameters for starting database nodes must include options that specify paths to certificate files from trusted certificate authorities, a node certificate, and a node key. The list of additional options is provided in the table below.

Command-line Option	Description
`--grpc-ca`	Path to the certificate file `ca.crt` from a trusted certificate authority.
`--grpc-cert`	Path to the node certificate file `node.crt`.
`--grpc-key`	Path to the node private key file `node.key`.

   Example command for starting a database node with options specifying paths to TLS keys and certificates for the gRPC protocol:

   ```
   /opt/ydb/bin/ydbd server --config-dir /opt/ydb/cfg --tenant /Root/testdb \
       --grpcs-port 2136 --grpc-ca /opt/ydb/certs/ca.crt \
       --grpc-cert /opt/ydb/certs/node.crt --grpc-key /opt/ydb/certs/node.key \
       --ic-port 19002 --ca /opt/ydb/certs/ca.crt \
       --mon-port 8766 --mon-cert /opt/ydb/certs/web.pem \
       --node-broker grpcs://<ydb1>:2135 \
       --node-broker grpcs://<ydb2>:2135 \
       --node-broker grpcs://<ydb3>:2135
   ```

Enabling Database Node Authentication and Authorization

> **Note**
>
> Starting from version 25.2, properly configured database node authentication is mandatory.
>
> Node registration in the cluster is impossible without successful authentication.

To enable mandatory database node authorization, the following configuration blocks must be added to the cluster configuration file:

1. At the root level of the configuration, add a `client_certificate_authorization` block specifying requirements for filling the "Subject" field of trusted certificates of connecting nodes, for example:

   ```
   client_certificate_authorization:
     request_client_certificate: true
     client_certificate_definitions:
       - member_groups: ["registerNode@cert"]
         subject_terms:
         - short_name: "O"
           values: ["YDB"]
   ```

If necessary, add other certificate checks according to the documentation.

After the certificate is verified and its "Subject" field is confirmed to comply with the requirements established in the `subject_terms` block, the connection will be assigned access subjects from the `member_groups` parameter. To distinguish such access subjects, the `@cert` suffix is added to their names.

2. In the `security_config` section of the cluster configuration, add the `register_dynamic_node_allowed_sids` element with a list of access subjects allowed to register database nodes. Example:

```
security_config:
  enforce_user_token_requirement: true
  ...
  register_dynamic_node_allowed_sids:
  - "registerNode@cert"
```

For more information on configuring cluster authentication parameters, see Configuring Administrative and Other Privileges.

3. In the `auth_config` section of the cluster configuration, add the `node_registration_token` as an empty string. Example:

```
auth_config:
  ...
  node_registration_token: ""
```

4. Update the static configuration files on all cluster nodes manually or using the Ansible playbook.
5. Perform a rolling restart of cluster storage nodes using ydbops or using the Ansible playbook.
6. Perform a rolling restart of cluster database nodes using ydbops or using the Ansible playbook.

Static group SelfHeal

> **Warning**
>
> This article is dedicated to YDB clusters that use configuration V2. This configuration method is currently experimental and is only available for YDB versions starting from v25.1. For production use, we recommend choosing configuration V1 — it is the main method and is officially supported for all YDB clusters.

When using configuration V2, SelfHeal can automatically move a static group VDisk from faulty PDisks and restore the group's fault tolerance.

The general SelfHeal mechanism detects a faulty PDisk and initiates VDisk relocation. For dynamic groups, the Blob Storage Controller changes the configuration, while distributed configuration changes the static group configuration.

To allow distributed configuration to change the static group automatically, enable the `automatic_static_group_management` parameter. This parameter is disabled by default.

Enabling and disabling static group SelfHeal

Static group SelfHeal requires the following components to be enabled:

- distributed configuration V2 — `self_management_config.enabled: true`;
- the general SelfHeal mechanism, which is enabled by default.

To enable or disable automatic static group management:

1. Fetch the current cluster configuration using the ydb admin cluster config fetch command:

   ```
   ydb [global options...] admin cluster config fetch > config.yaml
   ```

2. Set the `automatic_static_group_management` parameter in the `config.yaml` configuration file:

   ```
   config:
     self_management_config:
       enabled: true
       automatic_static_group_management: true
   ```

   `true` enables automatic static group management, while `false` disables it.

3. Apply the updated configuration using the ydb admin cluster config replace command:

   ```
   ydb [global options...] admin cluster config replace -f config.yaml
   ```

Restricting the set of target nodes

By default, SelfHeal can move a static group VDisk to any suitable node. To restrict the set of target nodes, specify their IDs in the `static_group_self_heal_allowed_nodes` parameter:

```
config:
  self_management_config:
    enabled: true
    automatic_static_group_management: true
    static_group_self_heal_allowed_nodes:
    - 1
    - 2
    - 3
```

An empty list means that no restrictions apply. The allowed nodes must have suitable PDisks and enough free space to move the VDisk without violating the failure model.

Cluster Configuration Migration

YDB supports two configuration management mechanisms: V1 and V2 (experimental, available from version 25.1). Key differences between them are described in the article Comparing configurations V1 and V2.

Tip

New YDB clusters are recommended to be deployed using configuration V2. If a cluster was deployed using configuration V1, it will still use it after updating to YDB version 25.1 or higher. After such an update, it is recommended to plan and perform migration to V2, because support for V1 will be discontinued in future versions of YDB. For the instructions on how to determine the configuration version of the cluster, see Checking Configuration Version.

Depending on the current state of your cluster, you can perform migration:

- **To configuration V2:** If your cluster is managed by configuration V1, you can switch to the experimental configuration V2 mechanism.
- **To configuration V1:** If unexpected problems arose after switching to configuration V2, or you need to roll back the YDB version below 25.1, you can perform reverse migration to manual configuration management (V1).

Problems during migration?

If unexpected problems arise when using migration instructions (especially when rolling back to V1), it is recommended to report them immediately as a GitHub issue, providing maximum context and diagnostics for reproduction.

Before performing migration, make sure to determine which configuration version is currently used in your cluster using the version check instructions.

Migration to Configuration V2

> **Warning**
>
> This article is dedicated to YDB clusters that use configuration V2. This configuration method is currently experimental and is only available for YDB versions starting from v25.1. For production use, we recommend choosing configuration V1 — it is the main method and is officially supported for all YDB clusters.

This document contains instructions for migrating from configuration V1 to configuration V2.

In configuration V1, there are two different mechanisms for applying configuration files:

- static configuration manages storage nodes of the YDB cluster and requires manual placement of files on each cluster node;
- dynamic configuration manages database nodes of the YDB cluster and is loaded into the cluster centrally using YDB CLI commands.

In configuration V2, this process is unified: a single configuration file is loaded into the system through YDB CLI commands, automatically delivered to all cluster nodes.

The State Storage and static group components of the YDB cluster are key to the cluster's correct operation. When working with configuration V1, these components are configured manually by specifying the `domains_config` and `blob_storage_config` sections in the configuration file.
In configuration V2, automatic configuration of these components is possible without specifying the corresponding sections in the configuration file.

Initial State

Migration to configuration V2 can be performed if the following conditions are met:

1. The YDB cluster has been updated to version 25.1 and higher.
2. The YDB cluster is configured with a configuration V1 file `config.yaml`, located in the node file system and connected through the `ydbd --yaml-config` argument.
3. The cluster configuration file contains the `domains_config` and `blob_storage_config` sections for configuring State Storage and static group respectively.

Checking the Current Configuration Version

Before starting the migration, ensure that your cluster is running on configuration V1. You can find out the current configuration version on nodes using several methods described in the article Checking Configuration Version.

You should continue following this instruction only if the nodes are running on configuration version V1. If all nodes already have version V2 enabled, migration is not required.

Instructions for Migration to Configuration V2

To migrate the YDB cluster to configuration V2, you need to perform the following steps:

1. Check for the presence of a dynamic configuration file in the cluster. To do this, run the ydb admin cluster config fetch command:

   ```
   ydb -e grpc://<node.ydb.tech>:2135 admin cluster config fetch > config.yaml
   ```

   If no such configuration exists in the cluster, the command will output the message:

   ```
   No config returned.
   ```

   If a file is found, you should use it and skip the next step of this instruction. Before proceeding, make sure that the file contains the settings that were stored in the static configuration file on storage nodes in configuration V1.
2. If there is no dynamic configuration file in the cluster, run the dynamic configuration file generation command ydb admin cluster config generate. The file will be generated based on the static configuration file located on the cluster nodes.

   ```
   ydb -e grpc://<node.ydb.tech>:2135 admin cluster config generate > config.yaml
   ```

3. Check that the `config.yaml` file obtained in step 1 or 2 contains the storage node settings from the current static configuration V1. If the file was obtained from an existing dynamic configuration, these settings may be missing because in V1 dynamic configuration might not contain storage node parameters.

   The goal of this step is to get a single combined configuration file that contains both parameters from the dynamic configuration and settings from the static configuration.

   Move all sections and parameters that describe storage nodes, disks, State Storage, static group, and channel profiles from the static configuration file to the `config` section of the `config.yaml` file. In particular, check for the following sections and parameters if they are set in the static configuration V1:

   - `hosts`;
   - `host_configs`;
   - `domains_config`;
   - `blob_storage_config`;
   - `channel_profile_config`;
   - other related storage configuration parameters, such as `static_erasure`.
4. Add the following field to the checked and updated `config.yaml` file:

   ```
   feature_flags:
       ...
       switch_to_config_v2: true
   ```

More details

Enabling this flag means that the DS Controller tablet, not the Console tablet, is now responsible for configuration storage and operations. This switches the main cluster configuration management mechanism.

5. Place the `config.yaml` file on all cluster nodes, replacing the previous configuration file with it.
6. Create a directory for the YDB node to work with configuration on each node. If running multiple cluster nodes on one host, create separate directories for each node. Initialize the directory by running the ydb admin node config init command on each node. In the `--from-config` parameter, specify the path to the `config.yaml` file placed on the nodes earlier.

```
sudo mkdir -p /opt/ydb/config-dir
sudo chown -R ydb:ydb /opt/ydb/config-dir
ydb admin node config init --config-dir /opt/ydb/config-dir --from-config /opt/ydb/cfg/config.yaml
```

More details

In the future, the system will independently save the current configuration in the specified directories.

7. Restart all cluster nodes using the rolling-restart procedure, adding the `ydbd --config-dir` option when starting the node with the path to the directory specified, and removing the `ydbd --yaml-config` option.

Manual

When starting manually, add the `--config-dir` option to the `ydbd server` command, without specifying the `--yaml-config` option:

```
ydbd server --config-dir /opt/ydb/config-dir
```

Using systemd

When using systemd, add the `--config-dir` option to the `ydbd server` command in the systemd configuration file, and also remove the `--yaml-config` option:

```
ExecStart=/opt/ydb/bin/ydbd server --config-dir /opt/ydb/config-dir
```

After updating the systemd file, run the following command to apply the changes:

```
sudo systemctl daemon-reload
```

8. Load the previously obtained configuration file `config.yaml` into the system using the ydb admin cluster config replace command:

```
ydb -e grpc://<node.ydb.tech>:2135 admin cluster config replace -f config.yaml
```

The command will request confirmation to perform the operation `This command may damage your cluster, do you want to continue? [y/N]`, in response to this request you need to agree and enter `y`.

More details

After executing the command, the configuration file will be loaded into the internal storage of the DS Controller tablet and saved in the directories specified in the `--config-dir` option on each node. From this moment, any configuration change on existing nodes is performed using special commands of the YDB CLI. Also, when starting a node, the current configuration will be automatically loaded from the configuration directory.

9. Get the current cluster configuration using ydb admin cluster config fetch:

```
ydb -e grpc://<node.ydb.tech>:2135 admin cluster config fetch > config.yaml
```

The `config.yaml` file should match the configuration files distributed across the cluster nodes, except for the `metadata.version` field, which should be one unit higher compared to the version on the cluster nodes.

10. Add the following block to `config.yaml` in the `config` section:

```
self_management_config:
  enabled: true
```

More details

This section is responsible for enabling the distributed configuration mechanism in the cluster. Configuration storage and any operations on it will be performed through this mechanism.

11. Load the updated configuration file into the cluster using ydb admin cluster config replace:

```
ydb -e grpc://<node.ydb.tech>:2135 admin cluster config replace -f config.yaml
```

12. Restart all storage nodes of the cluster using the rolling restart procedure.
13. If there is a `config.domains_config.security_config` section in the `config.yaml` file, move it one level up — to the `config` section.
14. Remove the `config.blob_storage_config` and `config.domains_config` sections from the `config.yaml` file.

15. Load the updated configuration file into the cluster:

```
ydb -e grpc://<node.ydb.tech>:2135 admin cluster config replace -f config.yaml
```

More details

After loading the configuration, the YDB cluster will be switched to automatic configuration management mode for State Storage and static group using the distributed configuration mechanism.

You can verify the successful completion of the migration by checking the configuration version on the cluster nodes using one of the methods described in the article Checking Configuration Version. On all cluster nodes, the `Configuration version` should be equal to `v2`.

Result

As a result of the performed actions, the cluster will be migrated to configuration V2 mode. Unified configuration management is performed using special commands of the YDB CLI, static group and State Storage are managed automatically by the system.

Migration to Configuration V1

This document contains instructions for migrating from configuration V2 to configuration V1.

> **Note**
>
> This guide is intended for emergency situations when unexpected problems arise after migrating to configuration V2 and a rollback to configuration V1 is required, for example, for subsequent rollback to a YDB version below v25-1. This procedure is not required in normal operation mode.

Initial State

Migration to configuration V1 is only possible if the cluster uses configuration V2. This can be achieved:

- as a result of migration to configuration V2
- during initial deployment of the cluster

You can determine the current configuration version on nodes using several methods described in the article Checking Configuration Version. Before starting the migration, ensure that the cluster is running on configuration V2.

Instructions for Migration to Configuration V1

To migrate the YDB cluster to configuration V1, you need to perform the following steps:

1. Get the current cluster configuration using the ydb admin cluster config fetch command:

   ```
   ydb -e grpc://<node.ydb.tech>:2135 admin cluster config fetch --v2-internal-state > config.yaml
   ```

   More details

   The `--v2-internal-state` argument specifies that the full cluster configuration will be retrieved, including State Storage and static group configuration parameters.

2. Modify the `config.yaml` configuration file by changing the `self_management_config.enabled` parameter value from `true` to `false`:

   ```
   self_management_config:
     enabled: false
   ```

   More details

   This section is responsible for managing the distributed configuration mechanism. Setting `enabled: false` disables this mechanism. Further management of State Storage and static group configuration will be performed manually through the `domains_config` and `blob_storage_config` sections respectively in the configuration file (these sections were obtained in the previous step when using the `--full` flag).

3. Load the updated configuration file into the cluster using ydb admin cluster config replace:

   ```
   ydb -e grpc://<node.ydb.tech>:2135 admin cluster config replace -f config.yaml
   ```

4. Restart all cluster nodes using the rolling restart procedure.

   More details

   After restarting the nodes, the cluster will be switched to manual State Storage and static group management mode, but will still use a unified configuration file delivered through the BSController tablet. Node configuration at startup will still be read from the directory specified in the `ydbd --config-dir` option and saved there.

5. Get the current cluster configuration using `ydb admin cluster config fetch`:

   ```
   ydb -e grpc://<node.ydb.tech>:2135 admin cluster config fetch > config.yaml
   ```

   More details

   The obtained configuration will not contain the `domains_config` and `blob_storage_config` sections, as they are managed manually and should not be part of the dynamic configuration.

6. Place the obtained `config.yaml` file (this will be your static configuration V1) in the file system of each cluster node.
7. Restart all cluster nodes using the rolling-restart procedure, specifying the path to the static configuration file through the `ydbd --yaml-config` option and removing the `ydbd --config-dir` option:

   Manual

   When starting manually, add the `--yaml-config` option to the `ydbd server` command, without specifying the `--config-dir` option:

   ```
   ydbd server --yaml-config /opt/ydb/cfg/config.yaml
   ```

   Using systemd

   When using systemd, add the `--yaml-config` option to the `ydbd server` command in the systemd configuration file, and also remove the `--config-dir` option:

```
ExecStart=/opt/ydb/bin/ydbd server --yaml-config /opt/ydb/config/config.yaml
```

After updating the systemd file, run the following command to apply the changes:

```
sudo systemctl daemon-reload
```

You can verify the successful completion of the migration by checking the configuration version on the cluster nodes using one of the methods described in the article Checking Configuration Version. All cluster nodes should use configuration `v1`.

Result

As a result of the performed actions, the cluster will be migrated to configuration V1 mode. The configuration consists of two parts: static and dynamic, static group and State Storage management is performed manually.

Checking Configuration Version

There are two main ways to check which configuration mechanism version (V1 or V2) the nodes of your YDB cluster are using:

1. YDB UI
2. Cluster metrics

With YDB UI

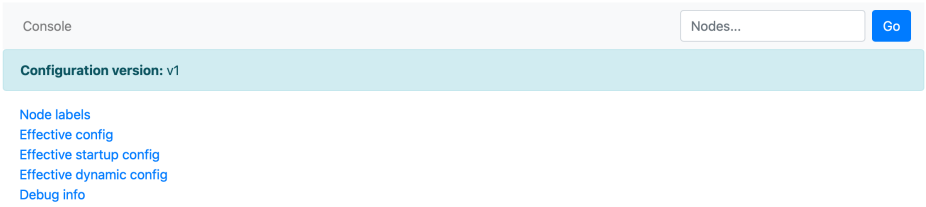
This method can be used if metrics collection from the YDB cluster to the monitoring system is not configured. You can check the configuration version for a specific node or switch between nodes in YDB UI:

1. Open the `configs_dispatcher` actor page for any cluster node in your browser:

   ```
   http://<endpoint>:8765/actors/configs_dispatcher
   ```

   where `<endpoint>` is the address of any YDB cluster node.
2. In the upper part of the opened page, find the `Configuration version` field. It shows the configuration version (`v1` or `v2`) used by this node.

   This is how the page of a node using configuration V1 looks:

3. To check other nodes, use the `Nodes...` search field in the upper right corner of the page to switch between nodes.

With Cluster Metrics

This method is convenient when there are a large number of nodes in the YDB cluster. If you have configured metrics collection from the YDB cluster to the monitoring system, perform the following actions:

1. Find the dashboard displaying cluster metrics.
2. Go to the `config` sensor group and the `configs_dispatcher` subsystem.
3. Pay attention to the `ConfigurationV1` and `ConfigurationV2` sensors. The values of these sensors show the number of cluster nodes running with configuration V1 and V2 respectively.

For example, if `ConfigurationV1 > 0` , it means there are nodes in the cluster that use configuration V1. If `ConfigurationV1 = 0` and `ConfigurationV2` equals the total number of nodes, it means all nodes use configuration V2.

Comparing YDB Cluster Configurations: V1 and V2

In YDB, there are two main approaches to cluster configuration management: V1 and V2. Starting from YDB version 25.1, configuration V2 is supported, which unifies YDB cluster management, allows working with configuration entirely through the YDB CLI, and also automates the most complex aspects of configuration (managing static group and State Storage). Configuration V2 is currently experimental and is not recommended for use in production environments.

> **Tip**
>
> New YDB clusters are recommended to be deployed using configuration V2. If a cluster was deployed using configuration V1, it will still use it after updating to YDB version 25.1 or higher. After such an update, it is recommended to plan and perform migration to V2, because support for V1 will be discontinued in future versions of YDB. For the instructions on how to determine the configuration version of the cluster, see Checking Configuration Version.

This article describes the key differences between these two approaches.

Characteristic	Configuration V1	Configuration V2
Configuration structure	Separate: static and dynamic.	**Unified** configuration.
File management	Static: manual file placement on each node. Dynamic: centralized upload via CLI.	Unified: centralized upload via CLI, automatic delivery to all nodes.
Delivery and application mechanism	Static: read and applied from a local file at startup. Dynamic: through `Console` tablet.	Fully automatic via the distributed configuration mechanism.
State Storage and static group management	**Manual**: mandatory `domains_config` and `blob_storage_config` sections in static configuration.	**Automatic**: managed by the distributed configuration system.
Recommended for YDB versions	Production use for all YDB versions.	Experimental use on clusters version 25.1 and above.

Configuration V1

Configuration V1 of a YDB cluster consists of two parts:

- **Static configuration:** manages key node parameters, including State Storage and static group configuration (`domains_config` and `blob_storage_config` sections respectively). Requires manual placement of the same configuration file on each cluster node. The path to the configuration is specified when starting the node using the `--yaml-config` option.
- **Dynamic configuration:** manages other cluster parameters. Loaded centrally using the `ydb admin config replace` command and distributed to database nodes.

If your cluster is running on configuration V1, it is recommended to perform migration to configuration V2.

Configuration V2

Starting from YDB version 25.1, configuration V2 is supported. Key features:

- **Unified configuration file:** the entire cluster configuration is stored and managed as a single entity.
- **Centralized management:** configuration is loaded to the cluster using the `ydb admin cluster config replace` command and automatically delivered to all nodes by the YDB cluster itself through the distributed configuration mechanism.
- **Early validation:** the configuration file is validated before delivering it to cluster nodes, not when restarting the server process.
- **Automatic State Storage and static group management:** V2 supports automatic configuration, which allows skipping these sections in the configuration file.
- **Storage on nodes:** the current configuration is automatically saved by each node in a special directory (specified by the `--config-dir` option when starting `ydbd`) and used during subsequent restarts.

Using configuration V2 is recommended for all YDB clusters version 25.1 and above.

YDB Deployment Options

This section of YDB documentation describes methods for deploying YDB clusters using various infrastructure management tools, as well as instructions for further work with them depending on the chosen method.

- **Ansible**: for deployments on bare metal and virtual machines.
- **Kubernetes**: for containerized deployments.
- **Manual**: for manual deployment.

Regardless of the chosen infrastructure management method, it is recommended to familiarize yourself with YDB system requirements before getting started.

Working With YDB Using Ansible

This section of YDB documentation contains a collection of articles intended for DevOps engineers managing YDB clusters using Ansible. This is the recommended approach to running production YDB clusters directly on virtual machines or bare metal. It is recommended to use Kubernetes instead of Ansible for containerized environments.

The key articles to get started with this section:

- Deploying a YDB cluster with Ansible
- Deploy Infrastructure for YDB Cluster using Terraform
- Restarting YDB Clusters deployed with Ansible
- Observability:
  - Logging on Clusters Deployed with Ansible

Deploying a YDB cluster with Ansible

This guide describes the process of deploying a YDB cluster on a group of servers using Ansible. This approach is recommended for environments with physical servers or virtual machines.

Key materials:

- Deployment preparation
- Deploying a cluster using configuration V1
- Deploying a cluster using configuration V2

Deployment preparation

Prerequisites

Choose a topology for installation

Before installing, choose a suitable YDB cluster topology — it determines the number of servers and disks required:

To get started with YDB, we recommend the `mirror-3dc-3-nodes` topology. It requires only 3 servers and 9 disks for user data — the simplest and fastest way to create a pilot cluster.

For large-scale deployments, choose one of the following options:

- `mirror-3-dc` — for clusters spread across multiple datacenters. Requires at least 9 servers and 9 disks for user data.
- `block-4-2` — for a cluster in a single datacenter. Requires at least 8 servers and 8 disks for user data.

Each server must have at least one dedicated disk for user data. We also recommend adding a separate small disk for the operating system. For more on topology and redundancy options, see this article.

You can expand the cluster later without stopping the cluster or interrupting user access to data.

Server configuration

Note

Recommended server requirements:

- 16 CPU cores (based on 8 CPUs used by the storage node and 8 CPUs by the dynamic node).
- 16 GB RAM (recommended minimum).
- Additional SSD disks for data, at least 120 GB each.
- SSH access.
- Network connectivity between machines in the cluster.
- OS: Ubuntu 18+, Debian 9+.
- Internet access for updating repositories and downloading required packages.

See YDB System Requirements and Recommendations for more details.

When using virtual machines from a public cloud provider, we recommend following Deploy Infrastructure for YDB Cluster using Terraform.

Software setup

To work with the project on your local (staging or installation) machine, you need:

- Python 3 version 3.10+
- Ansible core version 2.11 through 2.18
- A working directory on a machine with SSH access to all cluster servers

Tip

We recommend keeping the working directory and all files created during this guide in a version control system repository, for example Git. If multiple DevOps engineers will work with the cluster being deployed, they should collaborate in a shared repository.

If Ansible is already installed, you can go to the Ansible project setup section. If Ansible is not installed yet, install it using one of the following methods:

Installing Ansible globally

Using Ubuntu 22.04 LTS as an example:

- Update the apt package list with `sudo apt-get update`.
- Upgrade packages with `sudo apt-get upgrade`.
- Install the `software-properties-common` package to manage your distribution's software sources — `sudo apt install software-properties-common`.
- Add a new PPA to apt — `sudo add-apt-repository --yes --update ppa:ansible/ansible`.
- Install Ansible — `sudo apt-get install ansible-core` (ensure the installed version is no higher than 2.18 and no lower than 2.11; installing the `ansible` package alone will result in an outdated, unsuitable version).
- Check the Ansible core version — `ansible --version`.

Installing Ansible in a Python virtual environment

Using Ubuntu 22.04 LTS as an example:

- Update the list of available deb packages — `sudo apt-get update`.
- Install the `python3-venv` package for managing Python virtual environments — `sudo apt-get install python3-venv`.
- Create a directory where the virtual environment will be created. For example, `mkdir venv-ansible`.
- Create a Python virtual environment — `python3 -m venv venv-ansible`, where `venv-ansible` is the path to the directory created in the previous step.
- Activate the virtual environment — `source venv-ansible/bin/activate`. All further Ansible steps are performed inside the virtual environment. You can leave it with the `deactivate` command.
- Install the recommended Ansible version with `pip3 install "ansible-core>=2.11,<2.19"` (ensure the installed version is no higher than 2.18 and no lower than 2.11). Check the installed Ansible version — `ansible --version`.

For more information and other installation options, see the Ansible installation guide.

Ansible project setup

Installing YDB Ansible playbooks

Via requirements.yaml

```
cat <<EOF > requirements.yaml
roles: []
collections:
  - name: git+https://github.com/ydb-platform/ydb-ansible
    type: git
    version: latest
EOF
ansible-galaxy install -r requirements.yaml
```

One-time

```
ansible-galaxy collection install git+https://github.com/ydb-platform/ydb-ansible.git,latest
```

After completing the preparation steps, you can proceed to deploying the system. Choose the instruction that matches your configuration:

- Deploying a cluster using configuration V1
- Deploying a cluster using configuration V2

Deploying a cluster using configuration V1

Prepare the environment

Before deploying the system, complete the preparation steps. See the Deployment preparation document.

Create a working directory

```
mkdir deployment
cd deployment
mkdir inventory
mkdir files
```

Create the Ansible configuration file

Create `ansible.cfg` with Ansible configuration suitable for your target deployment environment. See the Ansible configuration reference for details. This guide assumes the `./inventory` subdirectory of the working directory is set up for inventory files.

Example starter ansible.cfg

> **Note**
>
> Using the `StrictHostKeyChecking=no` parameter in `ssh_args` makes automation easier but reduces SSH connection security (disables host key verification). For production environments, we recommend omitting this argument and configuring trusted keys manually. Use this parameter only for test and temporary installations.

```
[defaults]
conditional_bare_variables = False
force_handlers = True
gathering = explicit
interpreter_python = /usr/bin/python3
inventory = ./inventory
pipelining = True
private_role_vars = True
timeout = 5
verbosity = 1
log_path = ./ydb.log
vault_password_file = ./ansible_vault_password_file

[ssh_connection]
retries = 5
timeout = 60
ssh_args = -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -o ControlMaster=auto -o ControlPe
rsist=60s -o ControlPath=/tmp/ssh-%h-%p-%r -o ServerAliveCountMax=3 -o ServerAliveInterval=10
```

Create the main inventory file

Create the file `inventory/50-inventory.yaml` and fill it according to the chosen topology (see topology selection). Examples for each supported topology are in the tabs below — choose the one that fits.

mirror-3-dc-3nodes

```
all:
  children:
    ydb:
      # Servers
      hosts:
        static-node-1.ydb-cluster.com:
        static-node-2.ydb-cluster.com:
        static-node-3.ydb-cluster.com:

      vars:
        # Ansible
        ansible_user: username
        ansible_ssh_private_key_file: "/path/to/your/id_rsa"

        # System
        system_timezone: UTC
        system_ntp_servers: [time.cloudflare.com, time.google.com, ntp.ripe.net, pool.ntp.org]

        # Nodes
        ydb_config: "{{ ansible_config_file | dirname }}/files/config.yaml"
        ydb_version: "system_version"

        # Storage
        ydb_cores_static: 8
        ydb_disks:
          - name: /dev/vdb
            label: ydb_disk_1
          - name: /dev/vdc
            label: ydb_disk_2
          - name: /dev/vdd
            label: ydb_disk_3
        ydb_allow_format_drives: true
        ydb_skip_data_loss_confirmation_prompt: false
```

```
        ydb_pool_kind: ssd
        ydb_database_groups: 8
        ydb_cores_dynamic: 8
        ydb_dynnodes:
          - { instance: 'a', offset: 1 }
          - { instance: 'b', offset: 2 }
        ydb_brokers:
          - static-node-1.ydb-cluster.com
          - static-node-2.ydb-cluster.com
          - static-node-3.ydb-cluster.com

        # Database
        ydb_user: root
        ydb_domain: Root
        ydb_dbname: database

        # Authorization settings
        ydb_enforce_user_token_requirement: true
        ydb_request_client_certificate: true
```

mirror-3-dc-9-nodes

```
all:
  children:
    ydb:
      # Servers
      hosts:
        static-node-1.ydb-cluster.com:
        static-node-2.ydb-cluster.com:
        static-node-3.ydb-cluster.com:
        static-node-4.ydb-cluster.com:
        static-node-5.ydb-cluster.com:
        static-node-6.ydb-cluster.com:
        static-node-7.ydb-cluster.com:
        static-node-8.ydb-cluster.com:
        static-node-9.ydb-cluster.com:

      vars:
        # Ansible
        ansible_user: username
        ansible_ssh_private_key_file: "/path/to/your/id_rsa"

        # System
        system_timezone: UTC
        system_ntp_servers: [time.cloudflare.com, time.google.com, ntp.ripe.net, pool.ntp.org]

        # Nodes
        ydb_config: "{{ ansible_config_file | dirname }}/files/config.yaml"
        ydb_version: "system_version"

        # Storage
        ydb_cores_static: 8
        ydb_disks:
          - name: /dev/vdb
            label: ydb_disk_1
        ydb_allow_format_drives: true
        ydb_skip_data_loss_confirmation_prompt: false
        ydb_pool_kind: ssd
        ydb_database_groups: 8
        ydb_cores_dynamic: 8
        ydb_dynnodes:
          - { instance: 'a', offset: 1 }
          - { instance: 'b', offset: 2 }
        ydb_brokers:
          - static-node-1.ydb-cluster.com
          - static-node-2.ydb-cluster.com
          - static-node-3.ydb-cluster.com

        # Database
        ydb_user: root
        ydb_domain: Root
        ydb_dbname: database

        # Authorization settings
        ydb_enforce_user_token_requirement: true
        ydb_request_client_certificate: true
```

block-4-2

```
all:
  children:
    ydb:
      # Servers
      hosts:
        static-node-1.ydb-cluster.com:
        static-node-2.ydb-cluster.com:
        static-node-3.ydb-cluster.com:
        static-node-4.ydb-cluster.com:
        static-node-5.ydb-cluster.com:
        static-node-6.ydb-cluster.com:
```

```
        static-node-7.ydb-cluster.com:
        static-node-8.ydb-cluster.com:

      vars:
        # Ansible
        ansible_user: username
        ansible_ssh_private_key_file: "/path/to/your/id_rsa"

        # System
        system_timezone: UTC
        system_ntp_servers: [time.cloudflare.com, time.google.com, ntp.ripe.net, pool.ntp.org]

        # Nodes
        ydb_config: "{{ ansible_config_file | dirname }}/files/config.yaml"
        ydb_version: "system_version"

        # Storage
        ydb_cores_static: 8
        ydb_disks:
          - name: /dev/vdb
            label: ydb_disk_1
        ydb_allow_format_drives: true
        ydb_skip_data_loss_confirmation_prompt: false
        ydb_pool_kind: ssd
        ydb_database_groups: 7
        ydb_cores_dynamic: 8
        ydb_dynnodes:
          - { instance: 'a', offset: 1 }
          - { instance: 'b', offset: 2 }
        ydb_brokers:
          - static-node-1.ydb-cluster.com
          - static-node-2.ydb-cluster.com
          - static-node-3.ydb-cluster.com

        # Database
        ydb_user: root
        ydb_domain: Root
        ydb_dbname: database

        # Authorization settings
        ydb_enforce_user_token_requirement: true
        ydb_request_client_certificate: true
```

Required settings to adapt for your environment in the chosen template:

1. **Server hostnames.** Replace `static-node-*.ydb-cluster.com` in `all.children.ydb.hosts` and `all.children.vars.ydb_brokers` with the actual FQDNs.
2. **SSH access configuration.** Specify the `ansible_user` and the path to the private key `ansible_ssh_private_key_file` that Ansible will use to connect to your servers.
3. **Filesystem paths to block devices.** In the `all.children.ydb.vars.ydb_disks` section, the template assumes `/dev/vda` is for the operating system and the following disks, such as `/dev/vdb`, are for the YDB storage layer. Disk labels are created by the playbooks automatically, and their names can be arbitrary.
4. **System version.** In the `ydb_version` parameter, specify the YDB version to install. The list of available versions is on the downloads page.

Recommended settings to adapt for your environment:

- `ydb_domain`. This will be the first path component for all scheme objects in the cluster. For example, you can put your company name, cluster region, and so on there.
- `ydb_dbname`. This will be the second path component for all scheme objects in the database. For example, you can put the use case or project name there.

Additional settings

There are several options to specify which YDB executables you want to use for the cluster:

- `ydb_version`: automatically download one of the official YDB releases by version number. For example, `23.4.11`.
- `ydb_archive`: a local filesystem path to a YDB distribution archive downloaded or otherwise prepared in advance.

Installing a connector may be required for using federated queries. The playbook can deploy fq-connector-go on hosts with dynamic nodes. Use the following settings:

- `ydb_install_fq_connector` — set to `true` to install the connector.
- Choose one of the available options for deploying fq-connector-go executables:
  - `ydb_fq_connector_version`: automatically download one of the fq-connector-go official releases by version number. For example, `v0.7.1`.
  - `ydb_fq_connector_git_version`: automatically compile the fq-connector-go executable from source code downloaded from the official GitHub repository. The setting value is a branch, tag, or commit name. For example, `main`.
  - `ydb_fq_connector_archive`: a local filesystem path to a fq-connector-go distribution archive downloaded or otherwise prepared in advance.
  - `ydb_fq_connector_binary`: local filesystem paths to the fq-connector-go executable, downloaded or otherwise prepared in advance.
- `ydb_tls_dir` — specify a local path to a folder with TLS certificates prepared in advance. It must contain the `ca.crt` file and subdirectories whose names match the node hostnames, each containing certificates for that node. If not specified, self-signed TLS certificates will be generated automatically for the entire YDB cluster.
- `ydb_brokers` — list the FQDNs of the broker nodes. For example:

```
ydb_brokers:
    - static-node-1.ydb-cluster.com
    - static-node-2.ydb-cluster.com
    - static-node-3.ydb-cluster.com
```

The optimal value for the `ydb_database_groups` setting in the `vars` section depends on the available disks. Assuming only one database in the cluster, use the following logic:

- For production deployments, use disks with capacity over 800 GB and high IOPS, then choose the value for this setting based on cluster topology:
  - For `block-4-2`, set `ydb_database_groups` to 95% of the total number of disks, rounded down.
  - For `mirror-3-dc`, set `ydb_database_groups` to 84% of the total number of disks, rounded down.
- For testing YDB on small disks, set `ydb_database_groups` to 1 regardless of cluster topology.

The values of the `system_timezone` and `system_ntp_servers` variables depend on the infrastructure where the YDB cluster is deployed. By default, `system_ntp_servers` includes a set of NTP servers without regard to the geographic location of the infrastructure. We strongly recommend using a local NTP server for on-premise infrastructure and the following NTP servers for cloud providers:

AWS

- `system_timezone`: USA/<region_name>
- `system_ntp_servers`: [169.254.169.123, time.aws.com] Learn more about AWS NTP server settings.

Azure

- You can read about how time synchronization is configured on Azure virtual machines in this article.

Alibaba

- The specifics of connecting to NTP servers in Alibaba are described in this article.

Yandex Cloud

- `system_timezone`: Europe/Moscow
- `system_ntp_servers`: [0.ru.pool.ntp.org, 1.ru.pool.ntp.org, ntp0.NL.net, ntp2.vniiftri.ru, ntp.ix.ru, ntps1-1.cs.tu-berlin.de] Learn more about Yandex Cloud NTP server settings.

Change the root user password

Create the file `ansible_vault_password_file` with the following content:

```
password
```

This file contains the password that Ansible will use to encrypt and decrypt sensitive data automatically, for example user password files. This way passwords are not stored in plain text in the repository. For more on how Ansible Vault works, see the official documentation.

Next, set the password for the initial user specified in the `ydb_user` setting (default `root`). This user will have full access rights in the cluster initially; you can change this later if needed. Create `inventory/99-inventory-vault.yaml` with the following content (replace `<password>` with the actual password):

```
all:
  children:
    ydb:
      vars:
        ydb_password: <password>
```

Encrypt this file with the command `ansible-vault encrypt inventory/99-inventory-vault.yaml`.

Prepare the YDB configuration file

Create the file `files/config.yaml` and fill it according to the chosen topology (see topology selection). Examples for each supported topology are in the tabs below — choose the one that fits.

mirror-3-dc-3nodes

```
storage_config_generation: 0
static_erasure: mirror-3-dc
host_configs:
- drive:
  - path: /dev/disk/by-partlabel/ydb_disk_1
    type: SSD
  - path: /dev/disk/by-partlabel/ydb_disk_2
    type: SSD
  - path: /dev/disk/by-partlabel/ydb_disk_3
    type: SSD
  host_config_id: 1
hosts:
- host: static-node-1.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 1
    data_center: 'zone-a'
    rack: '1'
```

```
- host: static-node-2.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 2
    data_center: 'zone-b'
    rack: '2'
- host: static-node-3.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 3
    data_center: 'zone-d'
    rack: '3'
domains_config:
  domain:
  - name: Root
    storage_pool_types:
    - kind: ssd
      pool_config:
        box_id: 1
        erasure_species: mirror-3-dc
        kind: ssd
        geometry:
          realm_level_begin: 10
          realm_level_end: 20
          domain_level_begin: 10
          domain_level_end: 256
        pdisk_filter:
        - property:
          - type: SSD
        vdisk_kind: Default
  state_storage:
  - ring:
      node: [1, 2, 3]
      nto_select: 3
    ssid: 1
  security_config:
    enforce_user_token_requirement: true
    monitoring_allowed_sids:
    - "root"
    - "ADMINS"
    - "DATABASE-ADMINS"
    administration_allowed_sids:
    - "root"
    - "ADMINS"
    - "DATABASE-ADMINS"
    viewer_allowed_sids:
    - "root"
    - "ADMINS"
    - "DATABASE-ADMINS"
    register_dynamic_node_allowed_sids:
    - databaseNodes@cert
    - root@builtin
blob_storage_config:
  service_set:
    groups:
    - erasure_species: mirror-3-dc
      rings:
      - fail_domains:
        - vdisk_locations:
          - node_id: static-node-1.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
        - vdisk_locations:
          - node_id: static-node-1.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_2
        - vdisk_locations:
          - node_id: static-node-1.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_3
      - fail_domains:
        - vdisk_locations:
          - node_id: static-node-2.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
        - vdisk_locations:
          - node_id: static-node-2.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_2
        - vdisk_locations:
          - node_id: static-node-2.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_3
      - fail_domains:
        - vdisk_locations:
          - node_id: static-node-3.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
        - vdisk_locations:
          - node_id: static-node-3.ydb-cluster.com
            pdisk_category: SSD
```

```
            path: /dev/disk/by-partlabel/ydb_disk_2
        - vdisk_locations:
          - node_id: static-node-3.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_3
channel_profile_config:
  profile:
  - channel:
    - erasure_species: mirror-3-dc
      pdisk_category: 1   # 0=ROT, 1=SSD, 2=NVME
      storage_pool_kind: ssd
    - erasure_species: mirror-3-dc
      pdisk_category: 1
      storage_pool_kind: ssd
    - erasure_species: mirror-3-dc
      pdisk_category: 1
      storage_pool_kind: ssd
    profile_id: 0
interconnect_config:
    start_tcp: true
    encryption_mode: OPTIONAL
    path_to_certificate_file: "/opt/ydb/certs/node.crt"
    path_to_private_key_file: "/opt/ydb/certs/node.key"
    path_to_ca_file: "/opt/ydb/certs/ca.crt"
grpc_config:
    cert: "/opt/ydb/certs/node.crt"
    key: "/opt/ydb/certs/node.key"
    ca: "/opt/ydb/certs/ca.crt"
    services_enabled:
    - legacy
    - discovery
auth_config:
  path_to_root_ca: /opt/ydb/certs/ca.crt
client_certificate_authorization:
  request_client_certificate: true
  client_certificate_definitions:
      - member_groups: ["databaseNodes@cert"]
        subject_terms:
        - short_name: "O"
          values: ["YDB"]
query_service_config:
  generic:
    connector:
      endpoint:
        host: localhost
        port: 19102
      use_ssl: false
    default_settings:
      - name: DateTimeFormat
        value: string
      - name: UsePredicatePushdown
        value: "true"
feature_flags:
  enable_external_data_sources: true
  enable_script_execution_operations: true
```

mirror-3-dc-9-nodes

```
storage_config_generation: 0
static_erasure: mirror-3-dc
host_configs:
- drive:
  - path: /dev/disk/by-partlabel/ydb_disk_1
    type: SSD
  host_config_id: 1
hosts:
- host: static-node-1.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 1
    data_center: 'zone-a'
    rack: '1'
- host: static-node-2.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 2
    data_center: 'zone-a'
    rack: '2'
- host: static-node-3.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 3
    data_center: 'zone-a'
    rack: '3'
- host: static-node-4.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 4
    data_center: 'zone-b'
    rack: '4'
- host: static-node-5.ydb-cluster.com
```

```
  host_config_id: 1
  walle_location:
    body: 5
    data_center: 'zone-b'
    rack: '5'
- host: static-node-6.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 6
    data_center: 'zone-b'
    rack: '6'
- host: static-node-7.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 7
    data_center: 'zone-d'
    rack: '7'
- host: static-node-8.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 8
    data_center: 'zone-d'
    rack: '8'
- host: static-node-9.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 9
    data_center: 'zone-d'
    rack: '9'
domains_config:
  domain:
  - name: Root
    storage_pool_types:
    - kind: ssd
      pool_config:
        box_id: 1
        erasure_species: mirror-3-dc
        kind: ssd
        pdisk_filter:
        - property:
          - type: SSD
        vdisk_kind: Default
  state_storage:
  - ring:
      node: [1, 2, 3, 4, 5, 6, 7, 8, 9]
      nto_select: 9
    ssid: 1
  security_config:
    enforce_user_token_requirement: true
    monitoring_allowed_sids:
    - "root"
    - "ADMINS"
    - "DATABASE-ADMINS"
    administration_allowed_sids:
    - "root"
    - "ADMINS"
    - "DATABASE-ADMINS"
    viewer_allowed_sids:
    - "root"
    - "ADMINS"
    - "DATABASE-ADMINS"
    register_dynamic_node_allowed_sids:
    - databaseNodes@cert
    - root@builtin
blob_storage_config:
  service_set:
    groups:
    - erasure_species: mirror-3-dc
      rings:
      - fail_domains:
        - vdisk_locations:
          - node_id: static-node-1.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
        - vdisk_locations:
          - node_id: static-node-2.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
        - vdisk_locations:
          - node_id: static-node-3.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
      - fail_domains:
        - vdisk_locations:
          - node_id: static-node-4.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
        - vdisk_locations:
          - node_id: static-node-5.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
```

```
        - vdisk_locations:
          - node_id: static-node-6.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
      - fail_domains:
        - vdisk_locations:
          - node_id: static-node-7.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
        - vdisk_locations:
          - node_id: static-node-8.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
        - vdisk_locations:
          - node_id: static-node-9.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
channel_profile_config:
  profile:
  - channel:
    - erasure_species: mirror-3-dc
      pdisk_category: 1   # 0=ROT, 1=SSD, 2=NVME
      storage_pool_kind: ssd
    - erasure_species: mirror-3-dc
      pdisk_category: 1
      storage_pool_kind: ssd
    - erasure_species: mirror-3-dc
      pdisk_category: 1
      storage_pool_kind: ssd
    profile_id: 0
interconnect_config:
    start_tcp: true
    encryption_mode: OPTIONAL
    path_to_certificate_file: "/opt/ydb/certs/node.crt"
    path_to_private_key_file: "/opt/ydb/certs/node.key"
    path_to_ca_file: "/opt/ydb/certs/ca.crt"
grpc_config:
    cert: "/opt/ydb/certs/node.crt"
    key: "/opt/ydb/certs/node.key"
    ca: "/opt/ydb/certs/ca.crt"
    services_enabled:
    - legacy
auth_config:
  path_to_root_ca: /opt/ydb/certs/ca.crt
client_certificate_authorization:
  request_client_certificate: true
  client_certificate_definitions:
      - member_groups: ["databaseNodes@cert"]
        subject_terms:
        - short_name: "O"
          values: ["YDB"]
query_service_config:
  generic:
    connector:
      endpoint:
        host: localhost
        port: 19102
      use_ssl: false
    default_settings:
      - name: DateTimeFormat
        value: string
      - name: UsePredicatePushdown
        value: "true"
feature_flags:
  enable_external_data_sources: true
  enable_script_execution_operations: true
```

block-4-2

```
storage_config_generation: 0
static_erasure: block-4-2
host_configs:
- drive:
  - path: /dev/disk/by-partlabel/ydb_disk_1
    type: SSD
  host_config_id: 1
hosts:
- host: static-node-1.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 1
    data_center: 'zone-a'
    rack: '1'
- host: static-node-2.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 2
    data_center: 'zone-a'
    rack: '2'
- host: static-node-3.ydb-cluster.com
  host_config_id: 1
```

```
  walle_location:
    body: 3
    data_center: 'zone-a'
    rack: '3'
- host: static-node-4.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 4
    data_center: 'zone-a'
    rack: '4'
- host: static-node-5.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 5
    data_center: 'zone-a'
    rack: '5'
- host: static-node-6.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 6
    data_center: 'zone-a'
    rack: '6'
- host: static-node-7.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 7
    data_center: 'zone-a'
    rack: '7'
- host: static-node-8.ydb-cluster.com
  host_config_id: 1
  walle_location:
    body: 8
    data_center: 'zone-a'
    rack: '8'
domains_config:
  domain:
  - name: Root
    storage_pool_types:
    - kind: ssd
      pool_config:
        box_id: 1
        erasure_species: block-4-2
        kind: ssd
        pdisk_filter:
        - property:
          - type: SSD
        vdisk_kind: Default
  state_storage:
  - ring:
      node:
        - 1
        - 2
        - 3
        - 4
        - 5
        - 6
        - 7
        - 8
      nto_select: 8
    ssid: 1
  security_config:
    enforce_user_token_requirement: true
    monitoring_allowed_sids:
    - "root"
    - "ADMINS"
    - "DATABASE-ADMINS"
    administration_allowed_sids:
    - "root"
    - "ADMINS"
    - "DATABASE-ADMINS"
    viewer_allowed_sids:
    - "root"
    - "ADMINS"
    - "DATABASE-ADMINS"
    register_dynamic_node_allowed_sids:
    - databaseNodes@cert
    - root@builtin
blob_storage_config:
  service_set:
    groups:
    - erasure_species: block-4-2
      rings:
      - fail_domains:
        - vdisk_locations:
          - node_id: static-node-1.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
        - vdisk_locations:
          - node_id: static-node-2.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
```

```
        - vdisk_locations:
          - node_id: static-node-3.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
        - vdisk_locations:
          - node_id: static-node-4.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
        - vdisk_locations:
          - node_id: static-node-5.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
        - vdisk_locations:
          - node_id: static-node-6.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
        - vdisk_locations:
          - node_id: static-node-7.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
        - vdisk_locations:
          - node_id: static-node-8.ydb-cluster.com
            pdisk_category: SSD
            path: /dev/disk/by-partlabel/ydb_disk_1
channel_profile_config:
  profile:
  - channel:
    - erasure_species: block-4-2
      pdisk_category: 1   # 0=ROT, 1=SSD, 2=NVME
      storage_pool_kind: ssd
    - erasure_species: block-4-2
      pdisk_category: 1
      storage_pool_kind: ssd
    - erasure_species: block-4-2
      pdisk_category: 1
      storage_pool_kind: ssd
    profile_id: 0
interconnect_config:
    start_tcp: true
    encryption_mode: OPTIONAL
    path_to_certificate_file: "/opt/ydb/certs/node.crt"
    path_to_private_key_file: "/opt/ydb/certs/node.key"
    path_to_ca_file: "/opt/ydb/certs/ca.crt"
grpc_config:
    cert: "/opt/ydb/certs/node.crt"
    key: "/opt/ydb/certs/node.key"
    ca: "/opt/ydb/certs/ca.crt"
    services_enabled:
    - legacy
    - discovery
auth_config:
  path_to_root_ca: /opt/ydb/certs/ca.crt
client_certificate_authorization:
  request_client_certificate: true
  client_certificate_definitions:
      - member_groups: ["databaseNodes@cert"]
        subject_terms:
        - short_name: "O"
          values: ["YDB"]
query_service_config:
  generic:
    connector:
      endpoint:
        host: localhost
        port: 19102
      use_ssl: false
    default_settings:
      - name: DateTimeFormat
        value: string
      - name: UsePredicatePushdown
        value: "true"
feature_flags:
  enable_external_data_sources: true
  enable_script_execution_operations: true
```

To speed up and simplify the initial YDB deployment, the configuration file already contains most of the cluster setup. You only need to replace the placeholder host FQDNs with the actual ones in the `hosts` and `blob_storage_config` sections.

- The `hosts` section:

```
...
hosts:
  - host: static-node-1.ydb-cluster.com # VM FQDN
    host_config_id: 1
    walle_location:
      body: 1
      data_center: 'zone-a'
      rack: '1'
...
```

- The `blob_storage_config` section:

```
...
- fail_domains:
  - vdisk_locations:
    - node_id: static-node-1.ydb-cluster.com # VM FQDN
      pdisk_category: SSD
      path: /dev/disk/by-partlabel/ydb_disk_1
...
```

Leave the rest of the configuration file sections and settings unchanged.

Deploy the YDB cluster

After completing all the preparation steps above, the actual initial cluster deployment is running the following command from the working directory:

```
ansible-playbook ydb_platform.ydb.initial_setup
```

Shortly after it starts, you will need to confirm full wipe of the configured disks. Completion can then take tens of minutes depending on the environment and settings. This playbook performs roughly the same steps as in the manual YDB cluster deployment instructions.

Check cluster state

On the last step, the playbook runs several test queries using real temporary tables to verify correct operation. On success, you will see status ok, failed=0, and test query results (3 and 6) for each server when the playbook output is verbose enough.

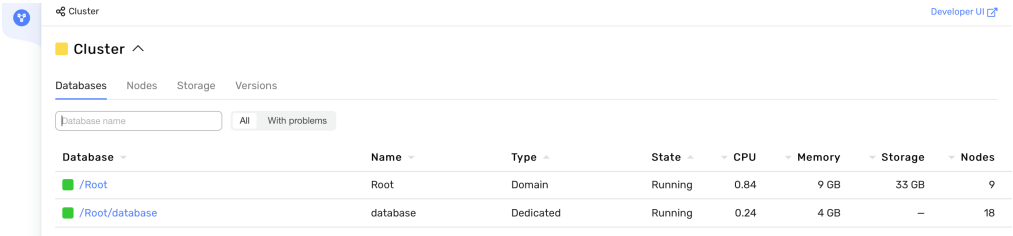
Example output

```
...

TASK [ydb_platform.ydb.ydbd_dynamic : run test queries] ************************************************
*********************************************************************************************************
ok: [static-node-1.ydb-cluster.com] => (item={'instance': 'a'}) => {"ansible_loop_var": "item", "change
d": false, "item": {"instance": "a"}, "msg": "all test queries were successful, details: {\"count\":3,\"s
um\":6}\n"}
ok: [static-node-1.ydb-cluster.com] => (item={'instance': 'b'}) => {"ansible_loop_var": "item", "change
d": false, "item": {"instance": "b"}, "msg": "all test queries were successful, details: {\"count\":3,\"s
um\":6}\n"}
ok: [static-node-2.ydb-cluster.com] => (item={'instance': 'a'}) => {"ansible_loop_var": "item", "change
d": false, "item": {"instance": "a"}, "msg": "all test queries were successful, details: {\"count\":3,\"s
um\":6}\n"}
ok: [static-node-2.ydb-cluster.com] => (item={'instance': 'b'}) => {"ansible_loop_var": "item", "change
d": false, "item": {"instance": "b"}, "msg": "all test queries were successful, details: {\"count\":3,\"s
um\":6}\n"}
ok: [static-node-3.ydb-cluster.com] => (item={'instance': 'a'}) => {"ansible_loop_var": "item", "change
d": false, "item": {"instance": "a"}, "msg": "all test queries were successful, details: {\"count\":3,\"s
um\":6}\n"}
ok: [static-node-3.ydb-cluster.com] => (item={'instance': 'b'}) => {"ansible_loop_var": "item", "change
d": false, "item": {"instance": "b"}, "msg": "all test queries were successful, details: {\"count\":3,\"s
um\":6}\n"}
PLAY RECAP *********************************************************************************************
*********************************************************************************************************
static-node-1.ydb-cluster.com : ok=167  changed=80   unreachable=0    failed=0    skipped=167  rescued=0
ignored=0
static-node-2.ydb-cluster.com : ok=136  changed=69   unreachable=0    failed=0    skipped=113  rescued=0
ignored=0
static-node-3.ydb-cluster.com : ok=136  changed=69   unreachable=0    failed=0    skipped=113  rescued=0
ignored=0
```

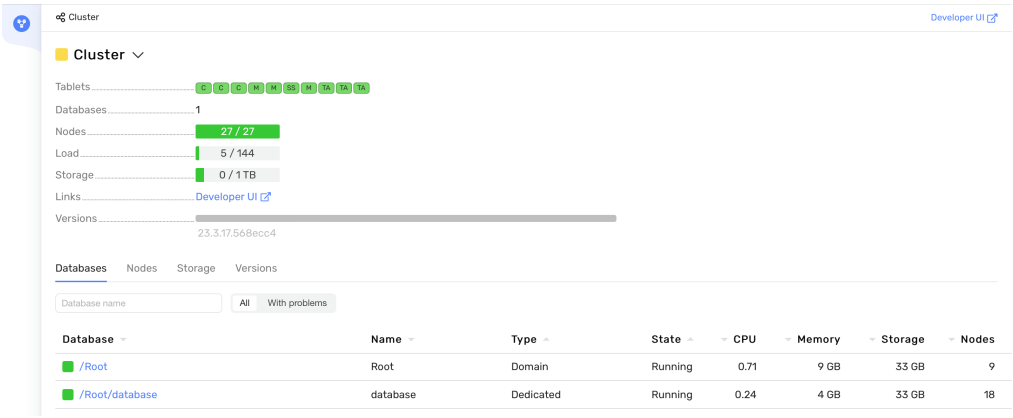
Running the `ydb_platform.ydb.initial_setup` playbook creates a YDB cluster. It will contain a domain named from the `ydb_domain` setting (default `Root`), a database named from the `ydb_dbname` setting (default `database`), and an initial user named from the `ydb_user` setting (default `root`).

Additional steps

The easiest way to explore the newly deployed cluster is YDB UI, which runs on port 8765 on each server. If you do not have direct browser access to that port, set up SSH tunneling by running `ssh -L 8765:localhost:8765 -i <private-key> <user>@<any-ydb-server-hostname>` on your local machine (add more options if needed). After the connection is established, open localhost:8765 in your browser. The browser may ask you to accept a security exception. Example:



After the YDB cluster is created, check its state on this YDB UI page: http://localhost:8765/monitoring/cluster/tenants. It might look like this:

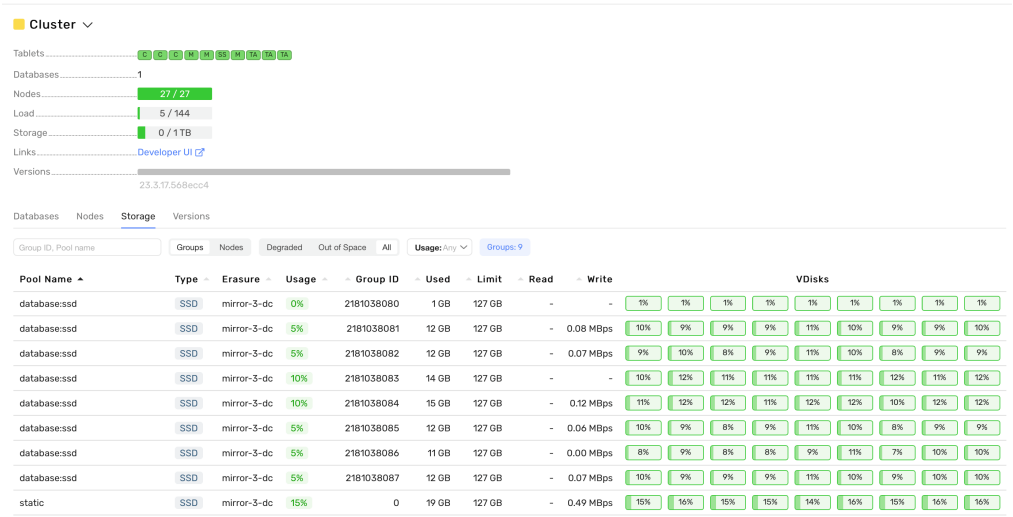


This section shows the following YDB cluster parameters:

- `Tablets` — list of running tablets. All tablet state indicators should be green.
- `Nodes` — number and state of storage and database nodes running in the cluster. The node state indicator should be green, and the number of created and running nodes should match (e.g., 18/18 for a nine-node cluster with one database node per server).

The `Load` (RAM used) and `Storage` (disk space used) indicators should also be green.

You can check the storage group state in the `storage` section — http://localhost:8765/monitoring/cluster/storage:



The `VDisks` indicators should be green, and the `state` status (in the tooltip when hovering over the Vdisk indicator) should be `Ok`. For more on cluster state indicators and monitoring, see YDB Monitoring.

Cluster testing

You can test the cluster using the built-in load tests in YDB CLI. Install YDB CLI and create a profile with connection parameters, replacing the placeholders:

```
ydb \
  config profile create <profile-name> \
  -d /<ydb-domain>/<ydb-database> \
  -e grpcs://<any-ydb-cluster-hostname>:2135 \
  --ca-file $(pwd)/files/TLS/certs/ca.crt \
  --user root \
  --password-file <path-to-a-file-with-password>
```

Command parameters and their meaning:

- `config profile create` — creates a connection profile. Specify the profile name. For more on creating and changing profiles, see Creating and updating profiles.
- `-e` — endpoint, a string in the form `protocol://host:port`. You can specify the FQDN of any cluster node and omit the port. Port 2135 is used by default.
- `--ca-file` — path to the root certificate for database connections over `grpcs`.
- `--user` — user for database connection.
- `--password-file` — path to the password file. Omit to enter the password manually.

To verify the profile was created, use `ydb config profile list`. Activate the profile with `ydb config profile activate <profile-name>`. To confirm it is active, run `ydb config profile list` again — the active profile will show `(active)`.

To run a YQL query, use `ydb sql -s 'SELECT 1;'`, which returns the result of `SELECT 1` in table form in the terminal. After checking the connection, create a test table with:
`ydb workload kv init --init-upserts 1000 --cols 4`. This creates the test table `kv_test` with 4 columns and 1000 rows. To verify the table and data, run `ydb sql -s 'select * from kv_test limit 10;'`.

The terminal will show a table of 10 rows. You can then run cluster performance tests. Key-Value load describes workload types (`upsert`, `insert`, `select`, `read-rows`, `mixed`) and their parameters. Example for the `upsert` workload with `--print-`

`timestamp` and default parameters: `ydb workload kv run upsert --print-timestamp`:

```
Window Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)        Timestamp
1          727 0       0       11      27      71      116     2024-02-14T12:56:39Z
2          882 0       0       10      21      29      38      2024-02-14T12:56:40Z
3          848 0       0       10      22      30      105     2024-02-14T12:56:41Z
4          901 0       0       9       20      27      42      2024-02-14T12:56:42Z
5          879 0       0       10      22      31      59      2024-02-14T12:56:43Z
...
```

When you are done, remove the `kv_test` table with `ydb workload kv clean`. For more on test table options and tests, see Key-Value load.

See also

- Additional Ansible configuration examples
- Restarting YDB Clusters deployed with Ansible
- Updating Configuration of YDB Clusters Deployed with Ansible
- Updating YDB Version on Clusters Deployed with Ansible

Deploying a cluster using configuration V2

Alert

This article is about YDB clusters that use **configuration V2**. This configuration method is still experimental and is only available for YDB versions starting with v25.1. For production we recommend configuration V1, which is the main, officially supported configuration for all YDB clusters.

Prepare the environment

Before deploying the system, complete the preparation steps. See the Deployment preparation document.

Create a working directory

```
mkdir deployment
cd deployment
mkdir -p inventory/group_vars/ydb
mkdir files
```

Create the Ansible configuration file

Create `ansible.cfg` with Ansible configuration suitable for your target deployment environment. See the Ansible configuration reference for details. This guide assumes the `./inventory` subdirectory of the working directory is set up for inventory files.

Example starter ansible.cfg

Note

Using the `StrictHostKeyChecking=no` parameter in `ssh_args` makes automation easier but reduces SSH connection security (disables host key verification). For production environments, we recommend omitting this argument and configuring trusted keys manually. Use this parameter only for test and temporary installations.

```
[defaults]
conditional_bare_variables = False
force_handlers = True
forks = 300
gathering = explicit
host_key_checking = False
interpreter_python = /usr/bin/python3
inventory = ./inventory
module_name = shell
pipelining = True
private_role_vars = True
retry_files_enabled = False
timeout = 5
vault_password_file = ./ansible_vault_password_file
verbosity = 1
log_path = ./ydb.log

[ssh_connection]
retries = 5
ssh_args = -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -o ControlMaster=auto -o ControlPe
rsist=60s -o ControlPath=/tmp/ssh-%h-%p-%r -o ServerAliveCountMax=3 -o ServerAliveInterval=10
```

Create the main inventory file

Create the file `inventory/group_vars/ydb/all.yaml` and fill it.

mirror-3-dc-3nodes

```
# Ansible
ansible_user: username
ansible_ssh_private_key_file: "/path/to/your/id_rsa"

# System
system_timezone: UTC
system_ntp_servers: [time.cloudflare.com, time.google.com, ntp.ripe.net, pool.ntp.org]

# Database
ydb_user: root
ydb_dbname: db

# Storage
ydb_disks:
  - name: /dev/vdb
    label: ydb_disk_1
  - name: /dev/vdc
    label: ydb_disk_2
  - name: /dev/vdd
    label: ydb_disk_3
ydb_pool_kind: ssd
ydbops_local: true
ydb_cores_dynamic: 2
```

```
ydb_dynnodes:
  - {"instance": "a", offset: 1}
ydb_cores_static:  2

# YDB
ydb_version: "system_version"
ydb_archive: "{{ ansible_config_file | dirname }}/files/ydbd.tar.gz"

# Additional parameters
ydb_allow_format_drives: true
ydb_skip_data_loss_confirmation_prompt: false
```

Required settings to adapt for your environment in the chosen template:

1. **SSH access configuration.** Specify the `ansible_user` and the path to the private key `ansible_ssh_private_key_file` that Ansible will use to connect to your servers.
2. **Filesystem paths to block devices.** In the `ydb_disks` section, the template assumes `/dev/vda` is for the operating system and the following disks, such as `/dev/vdb`, are for the YDB storage layer. Disk labels are created by the playbooks automatically, and their names can be arbitrary.
3. **System version.** In the `ydb_version` parameter, specify the YDB version to install. The list of available versions is on the downloads page.

Recommended settings to adapt for your environment:

- `ydb_domain`. This will be the first path component for all scheme objects in the cluster. For example, you can put your company name, cluster region, and so on there.
- `ydb_dbname`. This will be the second path component for all scheme objects in the database. For example, you can put the use case or project name there.

Additional settings

There are several options to specify which YDB executables you want to use for the cluster:

- `ydb_version`: automatically download one of the official YDB releases by version number. For example, `23.4.11`.
- `ydb_archive`: a local filesystem path to a YDB distribution archive downloaded or otherwise prepared in advance.

Installing a connector may be required for using federated queries. The playbook can deploy fq-connector-go on hosts with dynamic nodes. Use the following settings:

- `ydb_install_fq_connector` — set to `true` to install the connector.
- Choose one of the available options for deploying fq-connector-go executables:
  - `ydb_fq_connector_version`: automatically download one of the fq-connector-go official releases by version number. For example, `v0.7.1`.
  - `ydb_fq_connector_git_version`: automatically compile the fq-connector-go executable from source code downloaded from the official GitHub repository. The setting value is a branch, tag, or commit name. For example, `main`.
  - `ydb_fq_connector_archive`: a local filesystem path to a fq-connector-go distribution archive downloaded or otherwise prepared in advance.
  - `ydb_fq_connector_binary`: local filesystem paths to the fq-connector-go executable, downloaded or otherwise prepared in advance.
- `ydb_tls_dir` — specify a local path to a folder with TLS certificates prepared in advance. It must contain the `ca.crt` file and subdirectories whose names match the node hostnames, each containing certificates for that node. If not specified, self-signed TLS certificates will be generated automatically for the entire YDB cluster.
- `ydb_brokers` — list the FQDNs of the broker nodes. For example:

```
ydb_brokers:
    - static-node-1.ydb-cluster.com
    - static-node-2.ydb-cluster.com
    - static-node-3.ydb-cluster.com
```

The optimal value for the `ydb_database_groups` setting in the `vars` section depends on the available disks. Assuming only one database in the cluster, use the following logic:

- For production deployments, use disks with capacity over 800 GB and high IOPS, then choose the value for this setting based on cluster topology:
  - For `block-4-2`, set `ydb_database_groups` to 95% of the total number of disks, rounded down.
  - For `mirror-3-dc`, set `ydb_database_groups` to 84% of the total number of disks, rounded down.
- For testing YDB on small disks, set `ydb_database_groups` to 1 regardless of cluster topology.

The values of the `system_timezone` and `system_ntp_servers` variables depend on the infrastructure where the YDB cluster is deployed. By default, `system_ntp_servers` includes a set of NTP servers without regard to the geographic location of the infrastructure. We strongly recommend using a local NTP server for on-premise infrastructure and the following NTP servers for cloud providers:

AWS

- `system_timezone`: USA/<region_name>
- `system_ntp_servers`: [169.254.169.123, time.aws.com] Learn more about AWS NTP server settings.

Azure

- You can read about how time synchronization is configured on Azure virtual machines in this article.

Alibaba

- The specifics of connecting to NTP servers in Alibaba are described in this article.

Yandex Cloud

- `system_timezone`: Europe/Moscow
- `system_ntp_servers`: [0.ru.pool.ntp.org, 1.ru.pool.ntp.org, ntp0.NL.net, ntp2.vniiftri.ru, ntp.ix.ru, ntps1-1.cs.tu-berlin.de]
  Learn more about Yandex Cloud NTP server settings.

Change the root user password

Create the file `ansible_vault_password_file` with the following content:

```
password
```

This file contains the password that Ansible will use to encrypt and decrypt sensitive data automatically, for example user password files. This way passwords are not stored in plain text in the repository. For more on how Ansible Vault works, see the official documentation.

Next, set the password for the initial user specified in the `ydb_user` setting (default `root`). This user will have full access rights in the cluster initially; you can change this later if needed. Create `inventory/group_vars/ydb/vault.yaml` with the following content (replace `<password>` with the actual password):

```
ydb_password: <password>
```

Encrypt this file with the command `ansible-vault encrypt inventory/group_vars/ydb/vault.yaml`.

Prepare the YDB configuration file

Create the file `files/config.yaml` and fill it.

mirror-3-dc-3nodes

```
metadata:
  kind: MainConfig
  cluster: ""
  version: 0
config:
  yaml_config_enabled: true
  erasure: mirror-3-dc
  fail_domain_type: disk
  self_management_config:
    enabled: true
  default_disk_type: SSD
  host_configs:
  - host_config_id: 1
    drive:
    - path: /dev/disk/by-partlabel/ydb_disk_1
      type: SSD
    - path: /dev/disk/by-partlabel/ydb_disk_2
      type: SSD
    - path: /dev/disk/by-partlabel/ydb_disk_3
      type: SSD
  hosts:
  - host: ydb-node-zone-a.local
    host_config_id: 1
    location:
      body: 1
      data_center: 'zone-a'
      rack: '1'
  - host: ydb-node-zone-b.local
    host_config_id: 1
    location:
      body: 2
      data_center: 'zone-b'
      rack: '1'
  - host: ydb-node-zone-c.local
    host_config_id: 1
    location:
      body: 3
      data_center: 'zone-c'
      rack: '1'
  actor_system_config:
    use_auto_config: true
    cpu_count: 1
  interconnect_config:
    start_tcp: true
    encryption_mode: OPTIONAL
    path_to_certificate_file: "/opt/ydb/certs/node.crt"
    path_to_private_key_file: "/opt/ydb/certs/node.key"
    path_to_ca_file: "/opt/ydb/certs/ca.crt"
  grpc_config:
    cert: "/opt/ydb/certs/node.crt"
    key: "/opt/ydb/certs/node.key"
    ca: "/opt/ydb/certs/ca.crt"
    services_enabled:
    - legacy
  security_config:
    enforce_user_token_requirement: true
  client_certificate_authorization:
```

```
    request_client_certificate: true
    client_certificate_definitions:
    - member_groups: ["databaseNodes@cert"]
      subject_terms:
      - short_name: "O"
        values: ["YDB"]
  domains_config:
    security_config:
      monitoring_allowed_sids:
      - "root"
      - "ADMINS"
      - "DATABASE-ADMINS"
      administration_allowed_sids:
      - "root"
      - "ADMINS"
      - "DATABASE-ADMINS"
      viewer_allowed_sids:
      - "root"
      - "ADMINS"
      - "DATABASE-ADMINS"
      register_dynamic_node_allowed_sids:
      - databaseNodes@cert
      - root@builtin
```

To speed up and simplify the initial YDB deployment, the configuration file already contains most of the cluster setup. You only need to replace the placeholder host FQDNs in the `hosts` section and the disk paths in the `host_configs` section with the actual values.

- The `hosts` section:

```
...
hosts:
  - host: ydb-node-zone-a.local
  host_config_id: 1
  location:
    body: 1
    data_center: 'zone-a'
    rack: '1'
...
```

- The `host_configs` section:

```
...
host_configs:
- host_config_id: 1
  drive:
  - path: /dev/disk/by-partlabel/ydb_disk_1
    type: SSD
  - path: /dev/disk/by-partlabel/ydb_disk_2
    type: SSD
  - path: /dev/disk/by-partlabel/ydb_disk_3
    type: SSD
...
```

Leave the rest of the configuration file sections and settings unchanged.

Create the file `inventory/ydb_inventory.yaml` with the following content:

```
plugin: ydb_platform.ydb.ydb_inventory
ydb_config: "files/config.yaml"
```

Deploy the YDB cluster

After completing all the preparation steps above, the actual initial cluster deployment is running the following command from the working directory:

```
ansible-playbook ydb_platform.ydb.initial_setup
```

Shortly after it starts, you will need to confirm full wipe of the configured disks. Completion can then take tens of minutes depending on the environment and settings. This playbook performs roughly the same steps as in the manual YDB cluster deployment instructions.

Check cluster state

On the last step, the playbook runs several test queries using real temporary tables to verify correct operation. On success, you will see status ok, failed=0, and test query results (3 and 6) for each server when the playbook output is verbose enough.

Example output

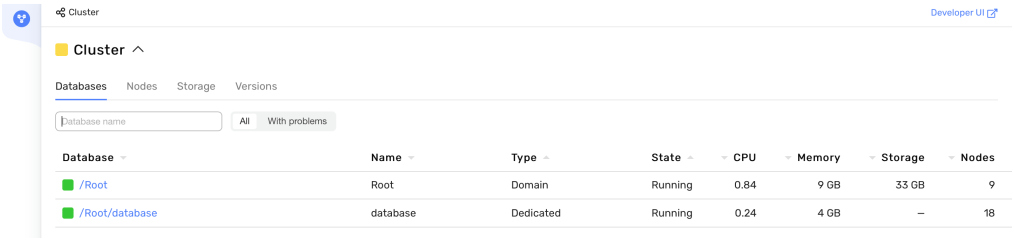
```
...

TASK [ydb_platform.ydb.ydbd_dynamic : run test queries] ************************************************
*********************************************************************************************
ok: [static-node-1.ydb-cluster.com] => (item={'instance': 'a'}) => {"ansible_loop_var": "item", "change
d": false, "item": {"instance": "a"}, "msg": "all test queries were successful, details: {\"count\":3,\"s
um\":6}\n"}
ok: [static-node-1.ydb-cluster.com] => (item={'instance': 'b'}) => {"ansible_loop_var": "item", "change
d": false, "item": {"instance": "b"}, "msg": "all test queries were successful, details: {\"count\":3,\"s
```

```
um\":6}\n"}
ok: [static-node-2.ydb-cluster.com] => (item={'instance': 'a'}) => {"ansible_loop_var": "item", "change
d": false, "item": {"instance": "a"}, "msg": "all test queries were successful, details: {\"count\":3,\"s
um\":6}\n"}
ok: [static-node-2.ydb-cluster.com] => (item={'instance': 'b'}) => {"ansible_loop_var": "item", "change
d": false, "item": {"instance": "b"}, "msg": "all test queries were successful, details: {\"count\":3,\"s
um\":6}\n"}
ok: [static-node-3.ydb-cluster.com] => (item={'instance': 'a'}) => {"ansible_loop_var": "item", "change
d": false, "item": {"instance": "a"}, "msg": "all test queries were successful, details: {\"count\":3,\"s
um\":6}\n"}
ok: [static-node-3.ydb-cluster.com] => (item={'instance': 'b'}) => {"ansible_loop_var": "item", "change
d": false, "item": {"instance": "b"}, "msg": "all test queries were successful, details: {\"count\":3,\"s
um\":6}\n"}
PLAY RECAP *********************************************************************************************
*********************************************************************************************
static-node-1.ydb-cluster.com : ok=167  changed=80   unreachable=0    failed=0    skipped=167  rescued=0
ignored=0
static-node-2.ydb-cluster.com : ok=136  changed=69   unreachable=0    failed=0    skipped=113  rescued=0
ignored=0
static-node-3.ydb-cluster.com : ok=136  changed=69   unreachable=0    failed=0    skipped=113  rescued=0
ignored=0
```

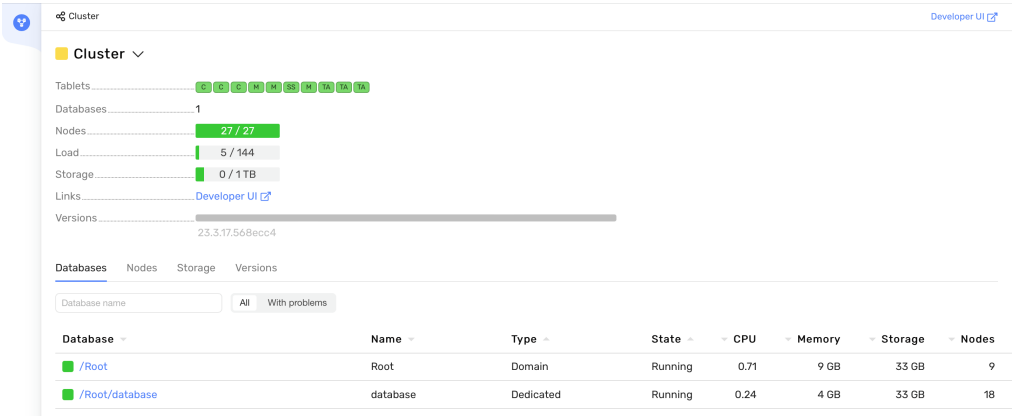
Running the `ydb_platform.ydb.initial_setup` playbook creates a YDB cluster. It will contain a domain named from the `ydb_domain` setting (default `Root`), a database named from the `ydb_dbname` setting (default `db`), and an initial user named from the `ydb_user` setting (default `root`).

Additional steps

The easiest way to explore the newly deployed cluster is YDB UI, which runs on port 8765 on each server. If you do not have direct browser access to that port, set up SSH tunneling by running `ssh -L 8765:localhost:8765 -i <private-key> <user>@<any-ydb-server-hostname>` on your local machine (add more options if needed). After the connection is established, open localhost:8765 in your browser. The browser may ask you to accept a security exception. Example:



After the YDB cluster is created, check its state on this YDB UI page: http://localhost:8765/monitoring/cluster/tenants. It might look like this:

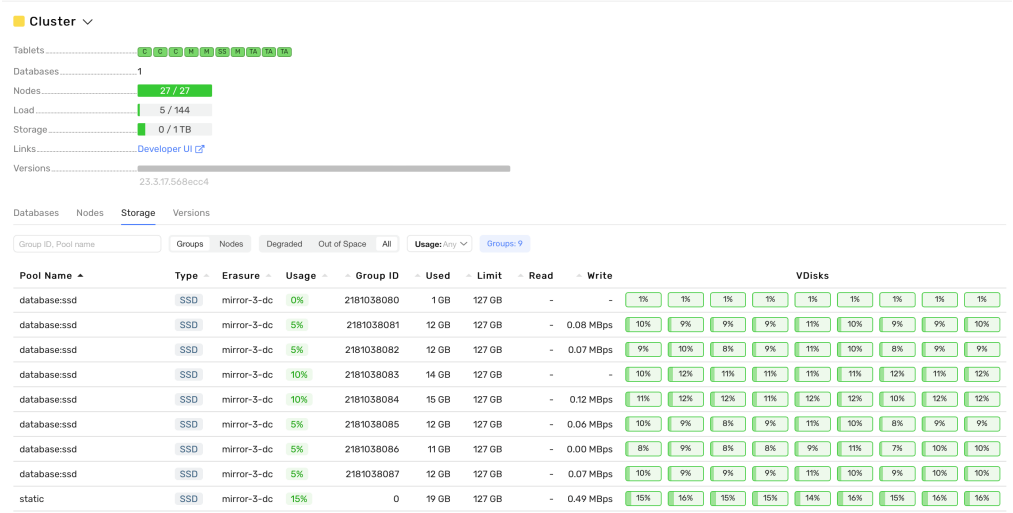


This section shows the following YDB cluster parameters:

- `Tablets` — list of running tablets. All tablet state indicators should be green.
- `Nodes` — number and state of storage and database nodes running in the cluster. The node state indicator should be green, and the number of created and running nodes should match (e.g., 18/18 for a nine-node cluster with one database node per server).

The `Load` (RAM used) and `Storage` (disk space used) indicators should also be green.

You can check the storage group state in the `storage` section — http://localhost:8765/monitoring/cluster/storage:



The `VDisks` indicators should be green, and the `state` status (in the tooltip when hovering over the Vdisk indicator) should be `Ok`. For more on cluster state indicators and monitoring, see YDB Monitoring.

Cluster testing

You can test the cluster using the built-in load tests in YDB CLI. Install YDB CLI and create a profile with connection parameters, replacing the placeholders:

```
ydb \
  config profile create <profile-name> \
  -d /<ydb-domain>/<ydb-database> \
  -e grpcs://<any-ydb-cluster-hostname>:2135 \
  --ca-file $(pwd)/files/TLS/certs/ca.crt \
  --user root \
  --password-file <path-to-a-file-with-password>
```

Command parameters and their meaning:

- `config profile create` — creates a connection profile. Specify the profile name. For more on creating and changing profiles, see Creating and updating profiles.
- `-e` — endpoint, a string in the form `protocol://host:port`. You can specify the FQDN of any cluster node and omit the port. Port 2135 is used by default.
- `--ca-file` — path to the root certificate for database connections over `grpcs`.
- `--user` — user for database connection.
- `--password-file` — path to the password file. Omit to enter the password manually.

To verify the profile was created, use `ydb config profile list`. Activate the profile with `ydb config profile activate <profile-name>`. To confirm it is active, run `ydb config profile list` again — the active profile will show `(active)`.

To run a YQL query, use `ydb sql -s 'SELECT 1;'`, which returns the result of `SELECT 1` in table form in the terminal. After checking the connection, create a test table with:
`ydb workload kv init --init-upserts 1000 --cols 4`. This creates the test table `kv_test` with 4 columns and 1000 rows. To verify the table and data, run `ydb sql -s 'select * from kv_test limit 10;'`.

The terminal will show a table of 10 rows. You can then run cluster performance tests. Key-Value load describes workload types (`upsert`, `insert`, `select`, `read-rows`, `mixed`) and their parameters. Example for the `upsert` workload with `--print-timestamp` and default parameters: `ydb workload kv run upsert --print-timestamp`:

```
Window Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)        Timestamp
1          727 0       0       11      27      71      116     2024-02-14T12:56:39Z
2          882 0       0       10      21      29      38      2024-02-14T12:56:40Z
3          848 0       0       10      22      30      105     2024-02-14T12:56:41Z
4          901 0       0       9       20      27      42      2024-02-14T12:56:42Z
5          879 0       0       10      22      31      59      2024-02-14T12:56:43Z
...
```

When you are done, remove the `kv_test` table with `ydb workload kv clean`. For more on test table options and tests, see Key-Value load.

See also

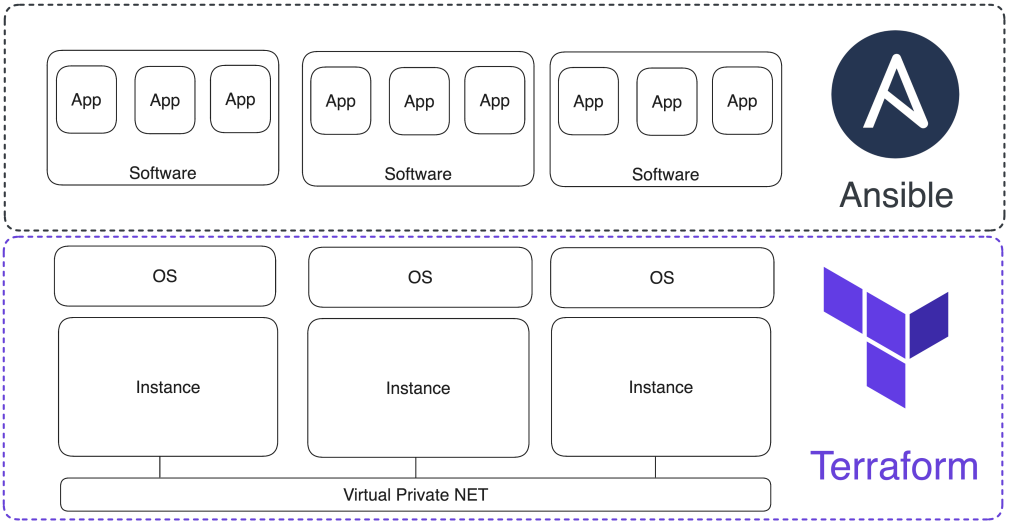
- Additional Ansible configuration examples
- Restarting YDB Clusters deployed with Ansible
- Updating Configuration of YDB Clusters Deployed with Ansible
- Updating YDB Version on Clusters Deployed with Ansible

Deploy Infrastructure for YDB Cluster using Terraform

You can deploy a YDB cluster for production use in three recommended ways: using Ansible, Kubernetes or manually. While the Kubernetes option is almost self-sufficient, the Ansible and manual options require SSH access to properly configured servers or virtual machines.

This article describes how to create and configure the necessary set of virtual machines in various cloud providers for a YDB cluster, using Terraform.

Terraform is an open-source infrastructure management software based on the "Infrastructure as Code" model. The same approach is used in Ansible, a configuration management system. Terraform and Ansible work at different levels: Terraform manages the infrastructure, and Ansible configures the environments on virtual machines (VM).



The configuration for setting up the VM environment is described in YAML format, and the infrastructure code is written in HCL (Terraform configuration language). The basic logical unit of recording in HCL is a "block". A block consists of a keyword identifying its type, name, and the block's body inside curly brackets. For example, this is what a virtual server control block in AWS might look like:

```
resource "aws_instance" "ydb-vm" {
  count                  = var.instance_count
  ami                    = "ami-008fe2fc65df48dac"
  instance_type          = "t2.micro"
  key_name               = var.req_key_pair
  vpc_security_group_ids = [var.input_security_group_id]
  subnet_id              = element(var.input_subnet_ids, count.index % length(var.input_subnet_ids))

  tags = {
    Name                 = "ydb-node-${count.index +1}"
    Username             = "ubuntu"
  }
}
```

Blocks can be independent, refer to each other, and thus be dependent, or they can also be nested inside each other.

Main block types:

- `resource` – a block for initializing an infrastructure resource (VM, network, subnet, disk, DNS zone, etc.).
- `provider` – a block for initializing the provider, API versions, and authentication data.
- `variable` – a variable either with a default value or empty for storing data entered by the user or passed by other blocks.
- `output` – outputs data to the terminal and saves it in a variable.
- `data` – a variable for requesting data from external cloud resources not presented in the created infrastructure.
- `module` – a logical grouping of resources that can be reused several times within the same or different projects.
- `terraform` – a block for configuring the behavior of Terraform itself, including the version of Terraform and used providers, as well as the backend settings, which are used for storing Terraform's state.

Blocks are written in files with the `.tf` extension and are logically grouped in directories, which in Terraform terminology are called modules. A module usually consists of the following files:

- `main.tf` – the main file where the infrastructure code is located. There can be several files containing infrastructure code.
- `variables.tf` – local variables of the module, which receive data from other modules or have default values.
- `outputs.tf` – variables that contain the results of the resource's operation (VM IP addresses, network/subnet IDs, etc.).

Modules are connected to the project in the root file `main.tf` as follows:

```
module "vpc" {
  source                     = "./modules/vpc"
  subnets_count              = var.subnets_count
  subnets_availability_zones = var.availability_zones
}
```

In the example, the `vpc` module is connected (the module name is assigned when connecting). The required parameter is `source`, a path to the directory where the module is located. `subnets_count` and `subnets_availability_zones` are variables inside the `vpc` module that take values from the global level variables `var.subnets_count`, `var.availability_zones`.

Modules, just like blocks, are placed one after another in the root `main.tf` file of the project. The main advantage of the modular approach to project organization is the ability to manage logically related sets of resources easily. Therefore, our repository with ready-made Terraform scenarios is organized as follows:

```
.
├── README.md
├── README_RU.md
├── aws
│   ├── README.md
│   ├── README_RU.md
│   ├── main.tf
│   ├── modules
│   │   ├── dns
│   │   ├── eip
│   │   ├── instance
│   │   ├── key_pair
│   │   ├── security
│   │   └── vpc
│   └── variables.tf
├── azure
│   ├── README.md
│   ├── README_RU.md
│   ├── main.tf
│   ├── modules
│   │   ├── dns
│   │   ├── resource_group
│   │   ├── security
│   │   ├── vm
│   │   └── vpc
│   └── variables.tf
├── ...
```

The subdirectories contain readme files, a file `variables.td` with local module variables and a central file `main.tf`, which includes modules from the `modules` subdirectory. The set of modules depends on the cloud provider. Basic modules, functionally the same for all providers, have the same names:

- `vpc` – cloud network and subnet management module.
- `dns` – DNS zone and DNS records management module.
- `security` – security group management module.
- `instance` – VM control module.

To use ready-made Terraform scripts from the repository, you need to download the repository with the command `git clone https://github.com/ydb-platform/ydb-terraform.git`, make changes to the Terraform configuration file `~/.terraformrc`, set the current values of global script variables and download the CLI of the cloud provider where the infrastructure will be created.

If you plan to use multiple providers, you can add the following code to `~/.terraformrc`, which will set the download paths for all providers described below:

```
provider_installation {
  network_mirror {
    url     = "https://terraform-mirror.yandexcloud.net/"
    include = ["registry.terraform.io/*/*"]
  }
  direct {
    exclude = ["registry.terraform.io/*/*"]
    exclude = ["terraform.storage.ydb.tech/*/*"]
  }
```

If you already use Terraform providers provided in the official repository, they will continue to work.

Deployment Overview

The following are step-by-step instructions for creating infrastructure in AWS, Azure, GCP, or Yandex Cloud. By default, example Terraform scenarios deploy the same type of infrastructure:

- VMs in three availability zones.
- Cloud network, public and private subnets (per subnet per availability zone).
- Private DNS zone.
- Security groups allowing ICMP and traffic on ports: 22, 65535, 19001, 8765, and 2135.

Most cluster parameters are adjustable (number of VMs, size and type of connected disks, number of networks, DNS zone domain name, etc.), but please note that the defaults are minimum recommended values, so changing them downwards may cause issues.

Create Infrastructure in AWS to Deploy a YDB Cluster

Create an account in AWS and add enough balance to run 9 VMs. Using the calculator, you can estimate the approximate cost of maintaining infrastructure depending on the region and other circumstances.

Create a user and connection key in AWS Cloud to run the AWS CLI:

1. The user is created in the Security credentials → Access management → Users → Create User section.
2. The next step is to assign rights to the user. Select `AmazonEC2FullAccess`.
3. After creating a user, go to its page, open the `Security credentials` tab, and click the **Create access key** button in the **Access keys** section.
4. Select `Command Line Interface` from the proposed options.
5. Next, create a tag for the key and click the **Create access key** button.
6. Copy the values of the `Access key` and `Secret access key` fields.

Install AWS CLI and run the `aws configure` command. Enter the values of the `Access key` and `Secret access key` fields saved earlier. Edit the `~/.aws/credentials` and `~/.aws/config` files as follows:

1. Add `[AWS_def_reg]` to `~/.aws/config` before `region = ...`.
2. Add `[AWS]` before the connection key secret information.

Go to the `aws` directory in the downloaded repository and edit the following variables in the `variable.tf` file:

1. `aws_region` – the region in which the infrastructure will be deployed.
2. `aws_profile` – security profile name from the file `~/.aws/credentials`.
3. `availability_zones` – list of region availability zones. It is formed from the name of the region and the serial letter. For example, for the `us-west-2` region, the list of availability zones will look like this: `["us-west-2a", "us-west-2b", "us-west-2c"]`.

Now, being in the `aws` subdirectory, you can run the following sequence of commands to install the provider, initialize modules, and create the infrastructure:

1. `terraform init` – installing the provider and initializing modules.
2. `terraform plan` – creating a plan for future infrastructure.
3. `terraform apply` – create resources in the cloud.

Next, use the commands `terraform plan`, `terraform apply`, and `terraform destroy` (destruction of the created infrastructure) to apply further changes as necessary.

Create Infrastructure in Azure to Deploy a YDB Cluster

Create an account in Azure and top up your account account with the amount, sufficient to operate 9 VMs. You can estimate the approximate cost of maintaining infrastructure depending on the region and other circumstances using calculator.

Authentication to the Azure Provider for Terraform goes through the CLI:

1. You can download, install, and configure the Azure CLI by following these instructions.
2. Log in using the Azure CLI interactively with the `az login` command.
3. The easiest way to create a pair of SSH keys (Linux, macOS) is to use the `ssh-keygen` command.

After logging into Azure and generating SSH keys, you need to change the default value of the following variables in the root file `variables.tf`:

1. `auth_location` —the name of the region where the infrastructure will be deployed. The command `az account list-locations | grep "displayName"` can obtain a list of available regions depending on the subscription.
2. `ssh_key_path` – path to the public part of the generated SSH key.

Now, being in the `azure` subdirectory, you can run the following sequence of commands to install the provider, initialize modules, and create the infrastructure:

1. `terraform init` – installing the provider and initializing modules.
2. `terraform plan` – creating a plan for future infrastructure.
3. `terraform apply` – create resources in the cloud.

Next, use the commands `terraform plan`, `terraform apply`, and `terraform destroy` (destruction of the created infrastructure) to apply further changes as necessary.

Creating Infrastructure in Google Cloud Platform to Deploy a YDB Cluster

Register in the Google Cloud console and create a project. Activate your payment account and top it up with funds to launch nine VMs. You can estimate the approximate cost in calculator.

Set up GCP CLI:

1. Activate Compute Engine API and Cloud DNS API.
2. Download and install GCP CLI by following these instructions.
3. Go to the `.../google-cloud-sdk/bin` subdirectory and run the `./gcloud compute regions list` command to get a list of available regions.
4. Run the command `./gcloud auth application-default login` to configure the connection profile.

Go to the `gcp` subdirectory (located in the downloaded repository), and in the `variables.tf` file set the current values for the following variables:

1. `project` – the project's name that was set in the Google Cloud cloud console.
2. `region` – the region where the infrastructure will be deployed.
3. `zones` – list of availability zones in which subnets and VMs will be created.

Now, being in the `gcp` subdirectory, you can run the following sequence of commands to install the provider, initialize modules, and create the infrastructure:

1. `terraform init` – installing the provider and initializing modules.
2. `terraform plan` – creating a plan for future infrastructure.
3. `terraform apply` – create resources in the cloud.

Next, use the commands `terraform plan`, `terraform apply`, and `terraform destroy` (destruction of the created infrastructure) to apply further changes as necessary.

Creating an Infrastructure in Yandex Cloud to Deploy a YDB Cluster

To create infrastructure in Yandex Cloud using Terraform, you need:

1. Prepare the cloud for work:
   - Register in Yandex Cloud.
   - Connect payment account.

    - Make sure that there are enough funds to create nine VMs.
2. Install and configure Yandex Cloud CLI:
    - Download Yandex Cloud CLI.
    - Create profile
3. Create service account using the CLI.
4. Generate Authorized key in JSON format for connecting Terraform to the cloud using the CLI: `yc iam key create --service-account-name <acc name> --output <file name> --folder-id <cloud folder id>`. Information about the created key will be displayed in the terminal:

    ```
    id: ajenap572v8e1l...
    service_account_id: aje90em65r69...
    created_at: "2024-09-03T15:34:57.495126296Z"
    key_algorithm: RSA_2048
    ```

    The authorized key will be created in the directory where the command was executed.
5. Configure Yandex Cloud Terraform provider.
6. Download this repository with the command `git clone https://github.com/ydb-platform/ydb-terraform.git`.
7. Go to the `yandex_cloud` directory in the downloaded repository and make changes to the following variables in the `variables.tf` file:
    - `key_path` – path to the SA key generated using the CLI.
    - `cloud_id` – cloud ID. You can get a list of available clouds with the command `yc resource-manager cloud list`.
    - `folder_id` – Cloud folder ID. Can be obtained with the command `yc resource-manager folder list`.

Now, being in the `yandex_cloud` subdirectory, you can run the following sequence of commands to install the provider, initialize modules, and create the infrastructure:

1. `terraform init` – installing the provider and initializing modules.
2. `terraform plan` – creating a plan for future infrastructure.
3. `terraform apply` – create resources in the cloud.

Next, use the commands `terraform plan`, `terraform apply`, and `terraform destroy` (destruction of the created infrastructure) to apply further changes as necessary.

Restarting YDB Clusters deployed with Ansible

YDB clusters provide strong availability guarantees; thus, the cluster's fault tolerance model needs to be considered during any maintenance, including cluster restarts. There are two kinds of nodes that might need to be restarted:

- Database nodes (also known as dynamic) are stateless; thus, the primary consideration is having enough of them running to handle each database's load. A basic rolling restart with a little delay is usually sufficient for dynamic nodes.
- Storage nodes (also known as static) are stateful and responsible for safely persisting data. Thus, they require special handling to ensure data availability. Each YDB cluster has a dedicated component that keeps track of all outages and maintenance and can tell if it is currently safe to stop or restart a particular node. Thus, asking for its permission for each operation is essential, and a complete restart of storage nodes often takes a while.

Restart via Ansible Playbook

ydb-ansible repository contains a playbook called `ydb_platform.ydb.restart` that can be used to restart a YDB cluster. Run it from the same directory used for the initial deployment.

Restart All Nodes

By default, the `ydb_platform.ydb.restart` restarts all cluster nodes. Static nodes go first, then dynamic nodes. The command to run it:

```
ansible-playbook ydb_platform.ydb.restart
```

Filter by Node Type

Tasks in the `ydb_platform.ydb.restart` playbook are tagged with node types, so you can use Ansible's tags functionality to filter nodes by their kind.

These two commands are equivalent and will restart all storage nodes:

```
ansible-playbook ydb_platform.ydb.restart --tags storage
ansible-playbook ydb_platform.ydb.restart --tags static
```

These two commands are equivalent and will restart all database nodes:

```
ansible-playbook ydb_platform.ydb.restart --tags database
ansible-playbook ydb_platform.ydb.restart --tags dynamic
```

Filter by Hostname

To restart a specific host or subset of hosts, use the `--limit` argument:

```
ansible-playbook ydb_platform.ydb.restart --limit='<hostname>'
ansible-playbook ydb_platform.ydb.restart --limit='<hostname-1,hostname-2>'
```

It can be used together with tags, too:

```
ansible-playbook ydb_platform.ydb.restart --tags database --limit='<hostname>'
```

Restart Nodes Manually

The ydbops tool properly implements various YDB cluster manipulations, including restarts. The `ydb_platform.ydb.restart` playbook explained above uses it behind the scenes, but it can be used manually, too.

There are more guidelines and information on how this works in the Maintenance without downtime article.

Updating Configuration of YDB Clusters Deployed with Ansible

During initial deployment, the Ansible playbook used the provided config file to create the initial cluster configuration. Technically, it generates two variants of the original config file and deploys them to all hosts via Ansible's mechanism for cross-server file copy. This article explains which options are available to change the cluster's configuration after the initial deployment.

Update Configuration via Ansible Playbook

ydb-ansible repository contains a playbook called `ydb_platform.ydb.update_config` that can be used to update YDB cluster's configuration. Go to the same directory used for the initial deployment, edit `files/config.yaml` as needed, and then run this playbook:

```
ansible-playbook ydb_platform.ydb.update_config
```

The playbook deploys the new version of the config files and then performs a rolling restart.

Filter by Node Type

Tasks in the `ydb_platform.ydb.update_config` playbook are tagged with node types, so you can use Ansible's tags functionality to filter nodes by their kind.

These two commands are equivalent and will change the configuration of all storage nodes:

```
ansible-playbook ydb_platform.ydb.update_config --tags storage
ansible-playbook ydb_platform.ydb.update_config --tags static
```

These two commands are equivalent and will change the configuration of all database nodes:

```
ansible-playbook ydb_platform.ydb.update_config --tags database
ansible-playbook ydb_platform.ydb.update_config --tags dynamic
```

Skip Restart

There's a `no_restart` tag to only deploy the config files and skip the cluster restart. This might be useful if the cluster will be restarted later manually or as part of some other maintenance tasks. Example:

```
ansible-playbook ydb_platform.ydb.update_config --tags no_restart
```

Updating YDB Version on Clusters Deployed with Ansible

During the initial deployment, the Ansible playbook provides several options for selecting the YDB server executable (`ydbd`). This article explains the available options for changing the cluster's version after the initial deployment.

> **Warning**
>
> YDB has specific rules regarding version compatibility. It is essential to refer to the version compatibility guide and changelog to correctly choose a new version to upgrade to and prepare for any nuances.

Update Executables via Ansible Playbook

The ydb-ansible repository contains a playbook called `ydb_platform.ydb.update_executable` that can be used to upgrade or downgrade a YDB cluster to another version. Navigate to the same directory used for the initial deployment, edit `inventory/50-inventory.yaml` to specify the target YDB version to install (typically, via the `ydb_version` or `ydb_git_version` variables), and then run this playbook:

```
ansible-playbook ydb_platform.ydb.update_executable
```

The playbook obtains a new binary and then deploys it to the cluster via Ansible's cross-server copying. After that, it performs a rolling restart.

Filter by Node Type

Tasks in the `ydb_platform.ydb.update_executable` playbook are tagged with node types, so you can use Ansible's tags functionality to filter nodes by their kind.

These two commands are equivalent and will change the configuration of all storage nodes:

```
ansible-playbook ydb_platform.ydb.update_executable --tags storage
ansible-playbook ydb_platform.ydb.update_executable --tags static
```

These two commands are equivalent and will change the configuration of all database nodes:

```
ansible-playbook ydb_platform.ydb.update_executable --tags database
ansible-playbook ydb_platform.ydb.update_executable --tags dynamic
```

Skip Restart

There's a `no_restart` tag to only deploy the executable files and skip the cluster restart. This might be useful if the cluster will be restarted later manually or as part of some other maintenance tasks. Example:

```
ansible-playbook ydb_platform.ydb.update_executable --tags no_restart
```

Ensuring Observability of a YDB Cluster Deployed with Ansible

This section of YDB documentation covers various observability-related topics specific to clusters deployed with Ansible.

- Logging on Clusters Deployed with Ansible

Additionally, there's a separate section with Reference on YDB observability that contains articles that do not depend on a chosen cluster deployment method.

Logging on Clusters Deployed with Ansible

During initial deployment, the Ansible playbook sets up several systemd units that run YDB nodes. Typically, there are multiple YDB nodes per physical server or virtual machine, each having its own log. There are two main ways to view logs of such cluster: via an Ansible playbook or via `ssh`.

View Logs via Ansible Playbook

ydb-ansible repository contains a playbook called `ydb_platform.ydb.logs` that can be used to show logs of all YDB nodes in a cluster. The playbook gathers logs from nodes and outputs them to `stdout`, which allows to pipe them for further processing, for example with commands like `grep` or `awk`.

All Logs of All Nodes

By default, the `ydb_platform.ydb.logs` playbook fetches logs of all YDB nodes. The command to do it:

```
ansible-playbook ydb_platform.ydb.logs
```

Filter by Node Type

There are two main node types in a YDB cluster:

- Storage (also known as static)
- Database (also known as dynamic)

Tasks in the `ydb_platform.ydb.logs` playbook are tagged with node types, so you can use Ansible's tags functionality to filter logs by node type.

These two commands are equivalent and will output the storage node logs:

```
ansible-playbook ydb_platform.ydb.logs --tags storage
ansible-playbook ydb_platform.ydb.logs --tags static
```

These two commands are equivalent, too, and will output the database node logs:

```
ansible-playbook ydb_platform.ydb.logs --tags database
ansible-playbook ydb_platform.ydb.logs --tags dynamic
```

Filter by hostname

To show logs of a specific host or subset of hosts, use the `--limit` argument:

```
ansible-playbook ydb_platform.ydb.logs --limit='<hostname>'
ansible-playbook ydb_platform.ydb.logs --limit='<hostname-1,hosntname-2>'
```

It can be used together with tags, too:

```
ansible-playbook ydb_platform.ydb.logs --tags database --limit='<hostname>'
```

View Logs via SSH

To manually access YDB cluster logs via `ssh`, perform the following steps:

1. Construct a `ssh` command to access the server that runs a YDB node you need logs for. The basic version would look like `ssh -i <path-to-ssh-key> <username>@<hostname>`. Take values for these placeholders from the `inventory/50-inventory.yaml` you used for deployment:
   - `<path-to-ssh-key>` is `children.ydb.ansible_ssh_private_key_file`
   - `<username>` is `children.ydb.ansible_user`
   - `<hostname>` is one of `children.ydb.hosts`
2. Choose which systemd unit's logs you need. You can skip this step if you already know the unit name. After logging in to the server using the `ssh` command constructed in the previous step, obtain the list of YDB-related systemd units using `systemctl list-units | grep ydb`. There'll likely be one storage node and multiple database nodes.

   Example output

   ```
   $ systemctl list-units | grep ydb
   ydb-transparent-hugepages.service                                          loaded active     exit
   ed    Configure Transparent Huge Pages (THP)
   ydbd-database-a.service                                                    loaded active     runn
   ing   YDB dynamic node / database / a
   ydbd-database-b.service                                                    loaded active     runn
   ing   YDB dynamic node / database / b
   ydbd-storage.service                                                       loaded active     runn
   ing   YDB storage node
   ```

3. Take the systemd unit name from the previous step and use it in the following command `journalctl -u <systemd-unit>` to actually show logs. You can specify `-u` multiple times to show logs of multiple units or use any other arguments from `man journalctl` to adjust the output.

Working With YDB Using Kubernetes

This section of YDB documentation contains a collection of articles intended for DevOps Engineers deploying YDB clusters using Kubernetes. This is the recommended approach to running production YDB clusters in containerized environments. For running YDB clusters on virtual machines or bare metal, use Ansible instead.

The key articles to get started with this section:

- Initial deployment

Getting Started with YDB in Kubernetes

Deploying YDB in Kubernetes is a simple way to set up and run a YDB cluster. Kubernetes allows to use an universal approach to managing your application in any cloud service provider. This guide provides instructions on how to deploy YDB in AWS EKS or Yandex Managed Service for Kubernetes.

Prerequisites

YDB is delivered as a Helm chart that is a package with templates of Kubernetes structures. For more information about Helm, see the documentation. The YDB chart can be deployed in the following environment:

1. A Kubernetes cluster with version 1.20 or higher. It needs to support Dynamic Volume Provisioning. Follow the instructions below if you don't have a suitable cluster yet.
2. The `kubectl` command line tool is installed and Kubernetes cluster access is configured.
3. The Helm package manager with a version higher than 3.1.0 is installed.

For YDB to work efficiently, we recommend using physical (not virtual) disks larger than 800 GB as block devices.

The minimum disk size is 80 GB, otherwise the YDB node won't be able to use the device. Correct and uninterrupted operation with minimum-size disks is not guaranteed. We recommend using such disks exclusively for informational purposes.

> **Warning**
>
> Configurations with disks less than 800 GB or any types of storage system virtualization cannot be used for production services or system performance testing.
>
> We don't recommend storing YDB data on disks shared with other processes (for example, the operating system).

Creating a Kubernetes Cluster

Skip this section if you have already configured a suitable Kubernetes cluster.

AWS EKS

1. Configure `awscli` and `eksctl` to work with AWS resources according to the documentation.
2. Configure `kubectl` to work with a Kubernetes cluster.
3. Run the following command:

```
eksctl create cluster \
  --name ydb \
  --nodegroup-name standard-workers \
  --node-type c5a.2xlarge \
  --nodes 3 \
  --nodes-min 1 \
  --nodes-max 4
```

This command will create a Kubernetes cluster named `ydb`. The `--node-type` flag indicates that the cluster is deployed using `c5a.2xlarge` (8vCPUs, 16 GiB RAM) instances. This meets minimal guidelines for running YDB.

It takes 10 to 15 minutes on average to create a Kubernetes cluster. Wait for the process to complete before proceeding to the next step of YDB deployment. The `kubectl` configuration will be automatically updated to work with the cluster after it is created.

Yandex Managed Service for Kubernetes

Follow the instructions in the Yandex Managed Service for Kubernetes quick start guide.

Overview of YDB Helm chart

The Helm chart installs YDB Kubernetes Operator to the Kubernetes cluster. It is a controller that follows the Operator design pattern. It implements the logic required for deploying and managing YDB components.

A YDB cluster consists of two kinds of nodes:

- **Storage nodes** (Storage resource) provide the data persistence layer.
- **Dynamic nodes** (Database resource) implement data access and processing.

Create both resources with the desired parameters to deploy a YDB cluster in Kubernetes. The schema for these resources is hosted on GitHub.

After the chart data is processed by the controller, the following resources are created:

- StatefulSet: A workload controller that assigns stable network IDs and disk resources to each container.
- Service: An object that is used to access the created databases from applications.
- ConfigMap: An object that is used to store the cluster configuration.

See the operator's source code on GitHub. The Helm chart is in the deploy folder.
YDB containers are deployed using `cr.yandex/yc/ydb` images. Currently, they are only available as prebuilt artifacts.

Environment Preparation

1. Add the YDB repository to Helm:

Run the command:

```
helm repo add ydb https://charts.ydb.tech/
```

`ydb`: The repository alias.
`https://charts.ydb.tech/`: The YDB repository URL.

Output:

```
"ydb" has been added to your repositories
```

2. Update the Helm chart index:

Run the command:

```
helm repo update
```

Output:

```
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "ydb" chart repository
Update Complete. ⎈Happy Helming!⎈
```

Deploying a YDB Cluster

Install the YDB Kubernetes operator

Use `helm` to deploy the YDB Kubernetes operator to the cluster:

```
helm install ydb-operator ydb/ydb-operator
```

- `ydb-operator`: The installation name.
- `ydb/ydb-operator`: The name of the chart in the repository you have added earlier.

Result:

```
NAME: ydb-operator
LAST DEPLOYED: Thu Aug 12 19:32:28 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Deploy Storage Nodes

YDB supports a number of storage topologies. YDB Kubernetes operator comes with a few sample configuration files for the most common topologies. This guide uses them as-is, but feel free to adjust them as needed or implement a new configuration file from scratch.

Apply the manifest for creating storage nodes:

block-4-2

```
kubectl apply -f https://raw.githubusercontent.com/ydb-platform/ydb-kubernetes-operator/master/samples/storag
e-block-4-2.yaml
```

This will create 8 YDB storage nodes that persist data using erasure coding. This takes only 50% of additional storage space to provide fault-tolerance.

mirror-3-dc

```
kubectl apply -f https://raw.githubusercontent.com/ydb-platform/ydb-kubernetes-operator/master/samples/storag
e-mirror-3dc.yaml
```

This will create 9 YDB storage nodes that store data with replication factor 3.

This command creates a `StatefulSet` object that describes a set of YDB containers with stable network IDs and disks assigned to them, as well as `Service` and `ConfigMap` objects that are required for the cluster to work.

YDB storage nodes take a while to initialize. You can check the initialization progress with `kubectl get storages.ydb.tech` or `kubectl describe storages.ydb.tech`. Wait until the status of the Storage resource changes to `Ready`.

> **Warning**
>
> The cluster configuration is static. The controller won't process any changes when the manifest is reapplied. You can only update cluster parameters such as version or disk size by creating a new cluster.

Create a database and dynamic nodes

YDB database is a logical entity that is served by a set of dynamic nodes. A sample manifest that comes with YDB Kubernetes operator creates a database named `database-sample` with 3 dynamic nodes. As with storage nodes, feel free to adjust the configuration as needed.

Apply the manifest for creating a database and dynamic nodes:

```
kubectl apply -f https://raw.githubusercontent.com/ydb-platform/ydb-kubernetes-operator/master/samples/databa
se.yaml
```

> **Note**
>
> The value referenced by `.spec.storageClusterRef.name` key must match the name of the `Storage` resource with storage nodes.

A `StatefulSet` object that describes a set of dynamic nodes is created after processing the manifest. The created database will be accessible from inside the Kubernetes cluster by the `database-sample` hostname or the `database-sample.<namespace>.svc.cluster.local` FQDN, where `namespace` indicates the namespace that was used for the installation. You can connect the database via port 2135.

View the status of the created resource:

```
kubectl describe database.ydb.tech
```

Result:

```
Name:         database-sample
Namespace:    default
Labels:       <none>
Annotations:  <none>
API Version:  ydb.tech/v1alpha1
Kind:         Database
...
  Storage Endpoint:  grpc://storage-sample-grpc.default.svc.cluster.local:2135
Status:
  State:  Ready
Events:
  Type     Reason              Age    From          Message
  ----     ------              ----   ----          -------
  Normal   Provisioning        8m10s  ydb-operator  Resource sync is in progress
  Normal   Provisioning        8m9s   ydb-operator  Resource sync complete
  Normal   TenantInitialized   8m9s   ydb-operator  Tenant /root/database-sample created
```

`State: Ready` means that the database is ready to be used.

Test Cluster Operation

Check how YDB works:

1. Check that all nodes are in the `Running` status:

   ```
   kubectl get pods
   ```

   Result:

   ```
   NAME                READY   STATUS    RESTARTS   AGE
   database-sample-0   1/1     Running   0          1m
   database-sample-1   1/1     Running   0          1m
   database-sample-2   1/1     Running   0          1m
   database-sample-3   1/1     Running   0          1m
   database-sample-4   1/1     Running   0          1m
   database-sample-5   1/1     Running   0          1m
   storage-sample-0    1/1     Running   0          1m
   storage-sample-1    1/1     Running   0          1m
   storage-sample-2    1/1     Running   0          1m
   storage-sample-3    1/1     Running   0          1m
   storage-sample-4    1/1     Running   0          1m
   storage-sample-5    1/1     Running   0          1m
   storage-sample-6    1/1     Running   0          1m
   storage-sample-7    1/1     Running   0          1m
   storage-sample-8    1/1     Running   0          1m
   ```

2. Start a new pod with YDB CLI:

   ```
   kubectl run -it --image=cr.yandex/crptqonuodf51kdj7a7d/ydb:24.4.4.2 --rm ydb-cli bash
   ```

3. Query the YDB database (you can get the endpoint from the output of `kubectl describe database.ydb.tech`):

   ```
   /opt/ydb/bin/ydb \
     --endpoint grpc://database-sample-grpc:2135 \
     --database /Root/database-sample \
     sql -s 'SELECT 2 + 2;'
   ```

   - `--endpoint`: The database endpoint.
   - `--database`: The name of the created database.
   - `--query`: The query text.

   Result:

   ```
   ┌─────────┐
   | column0 |
   ├─────────┤
   ```

```
| 4       |
└─────────┘
```

Further Steps

After you have tested that the created YDB cluster operates fine you can continue using it as you see fit. For example, if you just want to continue experimenting, you can use it to follow the YQL tutorial.

Below are a few more things to consider.

Monitoring

YDB provides standard mechanisms for collecting logs and metrics. Logging is done to standard `stdout` and `stderr` streams and can be redirected using popular solutions. For example, you can use a combination of Fluentd and Elastic Stack.

To collect metrics, `ydb-controller` provides resources like `ServiceMonitor`. They can be handled using kube-prometheus-stack.

Tuning Allocated Resources

You can limit resource consumption for each YDB pod. If you leave the limit values empty, a pod can use the entire CPU time and VM RAM. This may cause undesirable effects. We recommend that you always specify the resource limits explicitly.

To learn more about resource allocation and limits, see the Kubernetes documentation.

Release the Unused Resources

If you no longer need the created YDB cluster, delete it by following these steps:

1. To delete a YDB database and its dynamic nodes, just delete the respective `Database` resource:

   ```
   kubectl delete database.ydb.tech database-sample
   ```

2. To delete YDB storage nodes, run the following commands:

   ```
   kubectl delete storage.ydb.tech storage-sample
   kubectl delete pvc -l app.kubernetes.io/name=ydb
   ```

3. To remove the YDB Kubernetes operator, delete it with Helm:

   ```
   helm delete ydb-operator
   ```

   - `ydb-operator`: The name of the release that the controller was installed under.

Manual YDB Cluster Management Overview

This section provides information about deploying, configuring, maintaining, monitoring, and performing diagnostics of multi-node YDB clusters.

Main resources:

- Deploying YDB Cluster Manually
- Cluster Disk Subsystem Management Overview
- Setting Up YDB Cluster Monitoring
- Logging in YDB
- Backup and recovery
- Using YDB UI
- Cluster System Views

Deploying YDB Cluster Manually

This document describes how to deploy a multi-tenant YDB cluster with distributed architecture on multiple bare-metal or virtual servers.

Main guides:

- Deployment Preparation
- Deploying a Cluster Using Configuration V1
- Deploying a Cluster Using Configuration V2

Deployment Preparation

Getting Started

Prerequisites

Review the system requirements and the cluster topology.

Make sure you have SSH access to all servers. This is required to install artifacts and run the YDB executable.

The network configuration must allow TCP connections on the following ports (these are defaults, but you can change them by settings):

- 22: SSH service
- 2135, 2136: GRPC for client-cluster interaction.
- 19001, 19002: Interconnect for intra-cluster node interaction
- 8765, 8766: HTTP interface of YDB UI.

Distinct ports are necessary for gRPC, Interconnect and HTTP interface of each dynamic node when hosting multiple dynamic nodes on a single server.

Make sure that the system clocks running on all the cluster's servers are synced by `ntpd` or `chrony`. We recommend using the same time source for all servers in the cluster to maintain consistent leap seconds processing.

If the Linux flavor run on the cluster servers uses `syslogd` for logging, set up log file rotation using `logrotate` or similar tools. YDB services can generate substantial amounts of system logs, particularly when you elevate the logging level for diagnostic purposes. That's why it's important to enable system log file rotation to prevent the `/var` file system overflow.

Select the servers and disks to be used for storing data:

- Use the `block-4-2` fault tolerance model for cluster deployment in one availability zone (AZ). Use at least eight servers to safely survive the loss of two servers.
- Use the `mirror-3-dc` fault tolerance model for cluster deployment in three availability zones (AZ). To survive the loss of one AZ and one server in another AZ, use at least nine servers. Make sure that the number of servers running in each AZ is the same.

> **Note**
>
> Run each static node (data node) on a separate server. Both static and dynamic nodes can run together on the same server. A server can also run multiple dynamic nodes if it has enough computing power.

For more information about hardware requirements, see YDB System Requirements and Recommendations.

Preparing TLS Keys and Certificates

The TLS protocol provides traffic protection and authentication for YDB server nodes. Before you install your cluster, determine which servers it will host, establish the node naming convention, come up with node names, and prepare your TLS keys and certificates.

You can use existing certificates or generate new ones. Prepare the following files with TLS keys and certificates in the PEM format:

- `ca.crt`: CA-issued certificate used to sign the other TLS certificates (these files are the same on all the cluster nodes).
- `node.key`: Secret TLS keys for each cluster node (one key per cluster server).
- `node.crt`: TLS certificates for each cluster node (each certificate corresponds to a key).
- `web.pem`: Concatenation of the node secret key, node certificate, and the CA certificate needed for the monitoring HTTP interface (a separate file is used for each server in the cluster).

Your organization should define the parameters required for certificate generation in its policy. The following parameters are commonly used for generating certificates and keys for YDB:

- 2048-bit or 4096-bit RSA keys
- Certificate signing algorithm: SHA-256 with RSA encryption
- Validity period of node certificates: at least 1 year
- CA certificate validity period: at least 3 years.

Make sure that the CA certificate is appropriately labeled, with the CA property enabled along with the "Digital Signature, Non Repudiation, Key Encipherment, Certificate Sign" usage types.

For node certificates, it's key that the actual host name (or names) match the values in the "Subject Alternative Name" field. Enable both the regular usage types ("Digital Signature, Key Encipherment") and advanced usage types ("TLS Web Server Authentication, TLS Web Client Authentication") for the certificates. Node certificates must support both server authentication and client authentication (the `extendedKeyUsage = serverAuth,clientAuth` option in the OpenSSL settings).

For batch generation or update of YDB cluster certificates by OpenSSL, you can use the sample script from the YDB GitHub repository. Using the script, you can streamline preparation for installation, automatically generating all the key files and certificate files for all your cluster nodes in a single step.

Create a System User and a Group to Run YDB

On each server that will be running YDB, execute the command below:

```
sudo groupadd ydb
sudo useradd ydb -g ydb
```

To ensure that YDB can access block disks, add the user that will run YDB processes, to the `disk` group:

```
sudo usermod -aG disk ydb
```

Configure File Descriptor Limits

For proper operation of YDB, especially when using spilling in multi-node clusters, it is recommended to increase the limit of simultaneously open file descriptors.

To change the file descriptor limit, add the following lines to the `/etc/security/limits.conf` file:

```
ydb soft nofile 10000
ydb hard nofile 10000
```

Where `ydb` is the username under which `ydbd` runs.

After changing the file, you need to reboot the system or log in again to apply the new limits.

> **Note**
>
> For more information about spilling configuration and its relationship with file descriptors, see the Spilling Configuration section.

Install YDB Software on Each Server

1. Download and unpack an archive with the `ydbd` executable and the libraries required for YDB to run:

```
mkdir ydbd-stable-linux-amd64
curl -L https://binaries.ydb.tech/ydbd-stable-linux-amd64.tar.gz | tar -xz --strip-component=1 -C ydbd-stable
-linux-amd64
```

1. Create directories for YDB software:

```
sudo mkdir -p /opt/ydb /opt/ydb/cfg
```

1. Copy the executable and libraries to the appropriate directories:

```
sudo cp -iR ydbd-stable-linux-amd64/bin /opt/ydb/
sudo cp -iR ydbd-stable-linux-amd64/lib /opt/ydb/
```

1. Set the owner of files and folders:

```
sudo chown -R root:bin /opt/ydb
```

Prepare and Clear Disks on Each Server

For YDB to work efficiently, we recommend using physical (not virtual) disks larger than 800 GB as block devices.

The minimum disk size is 80 GB, otherwise the YDB node won't be able to use the device. Correct and uninterrupted operation with minimum-size disks is not guaranteed. We recommend using such disks exclusively for informational purposes.

> **Warning**
>
> Configurations with disks less than 800 GB or any types of storage system virtualization cannot be used for production services or system performance testing.
>
> We don't recommend storing YDB data on disks shared with other processes (for example, the operating system).

To get a list of available block devices on the server, you can use the `lsblk` command. Example output:

```
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
loop0    7:0    0  63.3M  1 loop /snap/core20/1822
...
vda    252:0    0    40G  0 disk
├─vda1 252:1    0     1M  0 part
└─vda2 252:2    0    40G  0 part /
vdb    252:16   0   186G  0 disk
└─vdb1 252:17   0   186G  0 part
```

The names of block devices depend on the operating system settings provided by the base image or manually configured. Typically, device names consist of up to three parts:

- A fixed prefix or a prefix indicating the device type
- A device sequential identifier (which can be a letter or a number)
- A partition sequential identifier on the given device (usually a number)

1. Create partitions on the selected disks:

> **Alert**
>
> The next operation will delete all partitions on the specified disk. Make sure that you specified a disk that contains no external data.

```
DISK=/dev/nvme0n1
sudo parted ${DISK} mklabel gpt -s
sudo parted -a optimal ${DISK} mkpart primary 0% 100%
sudo parted ${DISK} name 1 ydb_disk_ssd_01
sudo partx --u ${DISK}
```

Execute the command `ls -l /dev/disk/by-partlabel/` to ensure that a disk with the label `/dev/disk/by-partlabel/ydb_disk_ssd_01` has appeared in the system.

If you plan to use more than one disk on each server, replace `ydb_disk_ssd_01` with a unique label for each one. Disk labels should be unique within each server. They are used in configuration files, see the following guides.

To streamline the next setup step, it makes sense to use the same disk labels on cluster servers having the same disk configuration.

2. Clear the disk by this command built-in the `ydbd` executable:

> **Warning**
>
> After executing this command, data on the disk will be erased.

```
sudo LD_LIBRARY_PATH=/opt/ydb/lib /opt/ydb/bin/ydbd admin bs disk obliterate /dev/disk/by-partlabel/ydb_disk_ssd_01
```

Perform this operation for each disk to be used for YDB data storage.

After completing the preparation steps, proceed to deployment. Select the guide that matches your configuration:

- Deploying a cluster using configuration V1

Deploying a Cluster Using Configuration V1

Prepare the Environment

Before deploying the system, complete the preparatory steps. Review the Deployment Preparation document.

Prepare Configuration Files

Prepare the YDB configuration file according to your chosen topology (see cluster topology). Examples for each supported topology are provided below in tabs — select and use the one that fits your case.

mirror-3-dc-3nodes

```
static_erasure: mirror-3-dc
host_configs:
  - drive:
      - path: /dev/disk/by-partlabel/ydb_disk_ssd_01
        type: SSD
      - path: /dev/disk/by-partlabel/ydb_disk_ssd_02
        type: SSD
      - path: /dev/disk/by-partlabel/ydb_disk_ssd_03
        type: SSD
    host_config_id: 1
  hosts:
  - host: static-node-1.ydb-cluster.com
    host_config_id: 1
    walle_location:
      body: 1
      data_center: "zone-a"
      rack: "1"
  - host: static-node-2.ydb-cluster.com
    host_config_id: 1
    walle_location:
      body: 2
      data_center: "zone-b"
      rack: "2"
  - host: static-node-3.ydb-cluster.com
    host_config_id: 1
    walle_location:
      body: 3
      data_center: "zone-d"
      rack: "3"
  domains_config:
  security_config:
  enforce_user_token_requirement: true
  default_users:
    - name: "root"
      password: ""
  default_access:
    - "+(F):root"
  domain:
  - name: Root
  storage_pool_types:
  - kind: ssd
  pool_config:
  box_id: 1
  erasure_species: mirror-3-dc
  kind: ssd
  geometry:
    realm_level_begin: 10
    realm_level_end: 20
    domain_level_begin: 10
    domain_level_end: 256
  pdisk_filter:
    - property:
        - type: SSD
  vdisk_kind: Default
 state_storage:
  - ring:
      node: [1, 2, 3]
      nto_select: 3
      ssid: 1
  table_service_config:
  sql_version: 1
  actor_system_config:
  executor:
    - name: System
      threads: 2
      type: BASIC
    - name: User
      threads: 3
      type: BASIC
    - name: Batch
      threads: 2
      type: BASIC
    - name: IO
      threads: 1
      time_per_mailbox_micro_secs: 100
      type: IO
    - name: IC
      spin_threshold: 10
```

```
      threads: 1
      time_per_mailbox_micro_secs: 100
      type: BASIC
  scheduler:
    progress_threshold: 10000
    resolution: 256
    spin_threshold: 0
 blob_storage_config:
  service_set:
    groups:
      - erasure_species: mirror-3-dc
        rings:
          - fail_domains:
              - vdisk_locations:
                  - node_id: static-node-1.ydb-cluster.com
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
                  - node_id: static-node-1.ydb-cluster.com
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_02
                  - node_id: static-node-1.ydb-cluster.com
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_03
              - vdisk_locations:
                  - node_id: static-node-2.ydb-cluster.com
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
                  - node_id: static-node-2.ydb-cluster.com
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_02
                  - node_id: static-node-2.ydb-cluster.com
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_03
              - vdisk_locations:
                  - node_id: static-node-3.ydb-cluster.com
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
                  - node_id: static-node-3.ydb-cluster.com
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_02
                  - node_id: static-node-3.ydb-cluster.com
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_03
  channel_profile_config:
  profile:
    - channel:
        - erasure_species: mirror-3-dc
          pdisk_category: 1
          storage_pool_kind: ssd
        - erasure_species: mirror-3-dc
          pdisk_category: 1
          storage_pool_kind: ssd
        - erasure_species: mirror-3-dc
          pdisk_category: 1
          storage_pool_kind: ssd
      profile_id: 0
  interconnect_config:
  start_tcp: true
  encryption_mode: OPTIONAL
  path_to_certificate_file: "/opt/ydb/certs/node.crt"
  path_to_private_key_file: "/opt/ydb/certs/node.key"
  path_to_ca_file: "/opt/ydb/certs/ca.crt"
 grpc_config:
  cert: "/opt/ydb/certs/node.crt"
  key: "/opt/ydb/certs/node.key"
  ca: "/opt/ydb/certs/ca.crt"
  services_enabled:
    - legacy
  client_certificate_authorization:
    request_client_certificate: true
    client_certificate_definitions:
      - member_groups: ["registerNode@cert"]
        subject_terms:
          - short_name: "O"
            values: ["YDB"]
```

mirror-3-dc-9nodes

```
static_erasure: mirror-3-dc
host_configs:
  - drive:
      - path: /dev/disk/by-partlabel/ydb_disk_ssd_01
        type: SSD
      - path: /dev/disk/by-partlabel/ydb_disk_ssd_02
        type: SSD
    host_config_id: 1
hosts:
  - host: ydb-node-zone-a-1.local
    host_config_id: 1
    walle_location:
      body: 1
```

```
      data_center: 'zone-a'
      rack: '1'
  - host: ydb-node-zone-a-2.local
    host_config_id: 1
    walle_location:
      body: 2
      data_center: 'zone-a'
      rack: '2'
  - host: ydb-node-zone-a-3.local
    host_config_id: 1
    walle_location:
      body: 3
      data_center: 'zone-a'
      rack: '3'
  - host: ydb-node-zone-b-1.local
    host_config_id: 1
    walle_location:
      body: 4
      data_center: 'zone-b'
      rack: '4'
  - host: ydb-node-zone-b-2.local
    host_config_id: 1
    walle_location:
      body: 5
      data_center: 'zone-b'
      rack: '5'
  - host: ydb-node-zone-b-3.local
    host_config_id: 1
    walle_location:
      body: 6
      data_center: 'zone-b'
      rack: '6'
  - host: ydb-node-zone-d-1.local
    host_config_id: 1
    walle_location:
      body: 7
      data_center: 'zone-d'
      rack: '7'
  - host: ydb-node-zone-d-2.local
    host_config_id: 1
    walle_location:
      body: 8
      data_center: 'zone-d'
      rack: '8'
  - host: ydb-node-zone-d-3.local
    host_config_id: 1
    walle_location:
      body: 9
      data_center: 'zone-d'
      rack: '9'
domains_config:
  security_config:
  enforce_user_token_requirement: true
  default_users:
    - name: "root"
      password: ""
  default_access:
    - "+(F):root"
domain:
  - name: Root
storage_pool_types:
  - kind: ssd
pool_config:
  box_id: 1
  erasure_species: mirror-3-dc
  kind: ssd
  pdisk_filter:
    - property:
        - type: SSD
  vdisk_kind: Default
state_storage:
  - ring:
      node: [1, 2, 3, 4, 5, 6, 7, 8, 9]
      nto_select: 9
      ssid: 1
table_service_config:
  sql_version: 1
actor_system_config:
  executor:
    - name: System
      threads: 2
      type: BASIC
    - name: User
      threads: 3
      type: BASIC
    - name: Batch
      threads: 2
      type: BASIC
    - name: IO
      threads: 1
      time_per_mailbox_micro_secs: 100
```

```
      type: IO
    - name: IC
      spin_threshold: 10
      threads: 1
      time_per_mailbox_micro_secs: 100
      type: BASIC
  scheduler:
    progress_threshold: 10000
    resolution: 256
    spin_threshold: 0
blob_storage_config:
  service_set:
    groups:
      - erasure_species: mirror-3-dc
        rings:
          - fail_domains:
              - vdisk_locations:
                  - node_id: "ydb-node-zone-a-1.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-a-2.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-a-3.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-b-1.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-b-2.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-b-3.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-d-1.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-d-2.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-d-3.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
channel_profile_config:
  profile:
    - channel:
        - erasure_species: mirror-3-dc
          pdisk_category: 1
          storage_pool_kind: ssd
        - erasure_species: mirror-3-dc
          pdisk_category: 1
          storage_pool_kind: ssd
        - erasure_species: mirror-3-dc
          pdisk_category: 1
          storage_pool_kind: ssd
      profile_id: 0
interconnect_config:
  start_tcp: true
  encryption_mode: OPTIONAL
  path_to_certificate_file: "/opt/ydb/certs/node.crt"
  path_to_private_key_file: "/opt/ydb/certs/node.key"
  path_to_ca_file: "/opt/ydb/certs/ca.crt"
grpc_config:
  cert: "/opt/ydb/certs/node.crt"
  key: "/opt/ydb/certs/node.key"
  ca: "/opt/ydb/certs/ca.crt"
  services_enabled:
    - legacy
  client_certificate_authorization:
    request_client_certificate: true
    client_certificate_definitions:
      - member_groups: ["registerNode@cert"]
        subject_terms:
          - short_name: "O"
            values: ["YDB"]
```

block-4-2

```
static_erasure: block-4-2
host_configs:
  - drive:
      - path: /dev/disk/by-partlabel/ydb_disk_ssd_01
```

```
        type: SSD
      - path: /dev/disk/by-partlabel/ydb_disk_ssd_02
        type: SSD
    host_config_id: 1
hosts:
  - host: ydb-node-zone-a-1.local
    host_config_id: 1
    walle_location:
      body: 1
      data_center: 'zone-a'
      rack: '1'
  - host: ydb-node-zone-a-2.local
    host_config_id: 1
    walle_location:
      body: 2
      data_center: 'zone-a'
      rack: '2'
  - host: ydb-node-zone-a-3.local
    host_config_id: 1
    walle_location:
      body: 3
      data_center: 'zone-a'
      rack: '3'
  - host: ydb-node-zone-a-4.local
    host_config_id: 1
    walle_location:
      body: 4
      data_center: 'zone-a'
      rack: '4'
  - host: ydb-node-zone-a-5.local
    host_config_id: 1
    walle_location:
      body: 5
      data_center: 'zone-a'
      rack: '5'
  - host: ydb-node-zone-a-6.local
    host_config_id: 1
    walle_location:
      body: 6
      data_center: 'zone-a'
      rack: '6'
  - host: ydb-node-zone-a-7.local
    host_config_id: 1
    walle_location:
      body: 7
      data_center: 'zone-a'
      rack: '7'
  - host: ydb-node-zone-a-8.local
    host_config_id: 1
    walle_location:
      body: 8
      data_center: 'zone-a'
      rack: '8'
domains_config:
  security_config:
  enforce_user_token_requirement: true
  default_users:
    - name: "root"
      password: ""
  default_access:
    - "+(F):root"
domain:
  - name: Root
storage_pool_types:
  - kind: ssd
pool_config:
  box_id: 1
  erasure_species: block-4-2
  kind: ssd
  pdisk_filter:
    - property:
        - type: SSD
  vdisk_kind: Default
state_storage:
  - ring:
      node: [1, 2, 3, 4, 5, 6, 7, 8]
      nto_select: 5
      ssid: 1
table_service_config:
  sql_version: 1
actor_system_config:
  executor:
    - name: System
      threads: 2
      type: BASIC
    - name: User
      threads: 3
      type: BASIC
    - name: Batch
      threads: 2
      type: BASIC
```

```
    - name: IO
      threads: 1
      time_per_mailbox_micro_secs: 100
      type: IO
    - name: IC
      spin_threshold: 10
      threads: 1
      time_per_mailbox_micro_secs: 100
      type: BASIC
  scheduler:
    progress_threshold: 10000
    resolution: 256
    spin_threshold: 0
blob_storage_config:
  service_set:
    groups:
      - erasure_species: block-4-2
        rings:
          - fail_domains:
              - vdisk_locations:
                  - node_id: "ydb-node-zone-a-1.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-a-2.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-a-3.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-a-4.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-a-5.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-a-6.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-a-7.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
              - vdisk_locations:
                  - node_id: "ydb-node-zone-a-8.local"
                    pdisk_category: SSD
                    path: /dev/disk/by-partlabel/ydb_disk_ssd_01
channel_profile_config:
  profile:
    - channel:
        - erasure_species: block-4-2
          pdisk_category: 1
          storage_pool_kind: ssd
        - erasure_species: block-4-2
          pdisk_category: 1
          storage_pool_kind: ssd
        - erasure_species: block-4-2
          pdisk_category: 1
          storage_pool_kind: ssd
      profile_id: 0
interconnect_config:
  start_tcp: true
  encryption_mode: OPTIONAL
  path_to_certificate_file: "/opt/ydb/certs/node.crt"
  path_to_private_key_file: "/opt/ydb/certs/node.key"
  path_to_ca_file: "/opt/ydb/certs/ca.crt"
grpc_config:
  cert: "/opt/ydb/certs/node.crt"
  key: "/opt/ydb/certs/node.key"
  ca: "/opt/ydb/certs/ca.crt"
  services_enabled:
    - legacy
  client_certificate_authorization:
    request_client_certificate: true
    client_certificate_definitions:
      - member_groups: ["registerNode@cert"]
        subject_terms:
          - short_name: "O"
            values: ["YDB"]
```

To speed up and simplify the initial YDB deployment, the configuration file already contains most cluster setup settings. Replace the default host FQDNs with your actual ones in the `hosts` and `blob_storage_config` sections.

- `hosts` section:

```
...
hosts:
  - host: static-node-1.ydb-cluster.com  # VM FQDN
    host_config_id: 1
    walle_location:
      body: 1
      data_center: 'zone-a'
      rack: '1'
...
```

- `blob_storage_config` section:

```
...
- fail_domains:
  - vdisk_locations:
    - node_id: static-node-1.ydb-cluster.com  # VM FQDN
      pdisk_category: SSD
      path: /dev/disk/by-partlabel/ydb_disk_1
...
```

Leave all other configuration sections and settings unchanged.

Save the YDB configuration file as `/opt/ydb/cfg/config.yaml` on each cluster server.

For more detailed information about creating the configuration file, see YDB Cluster Configuration.

Copy TLS Keys and Certificates to Each Server

Copy the prepared TLS keys and certificates to a protected directory on each YDB cluster node. Below are sample commands to create a protected directory and copy the key and certificate files.

```
sudo mkdir -p /opt/ydb/certs
sudo cp -v ca.crt /opt/ydb/certs/
sudo cp -v node.crt /opt/ydb/certs/
sudo cp -v node.key /opt/ydb/certs/
sudo cp -v web.pem /opt/ydb/certs/
sudo chown -R ydb:ydb /opt/ydb/certs
sudo chmod 700 /opt/ydb/certs
```

Start Static Nodes

Manually

Run the YDB data storage service on each static cluster node:

```
sudo su - ydb
cd /opt/ydb
export LD_LIBRARY_PATH=/opt/ydb/lib
/opt/ydb/bin/ydbd server --log-level 3 --syslog --tcp --yaml-config  /opt/ydb/cfg/config.yaml \
    --grpcs-port 2135 --ic-port 19001 --mon-port 8765 --mon-cert /opt/ydb/certs/web.pem --node static &
```

Using systemd

On each server that will host a static cluster node, create a systemd configuration file `/etc/systemd/system/ydbd-storage.service` using the template below. You can also download the sample file from the repository.

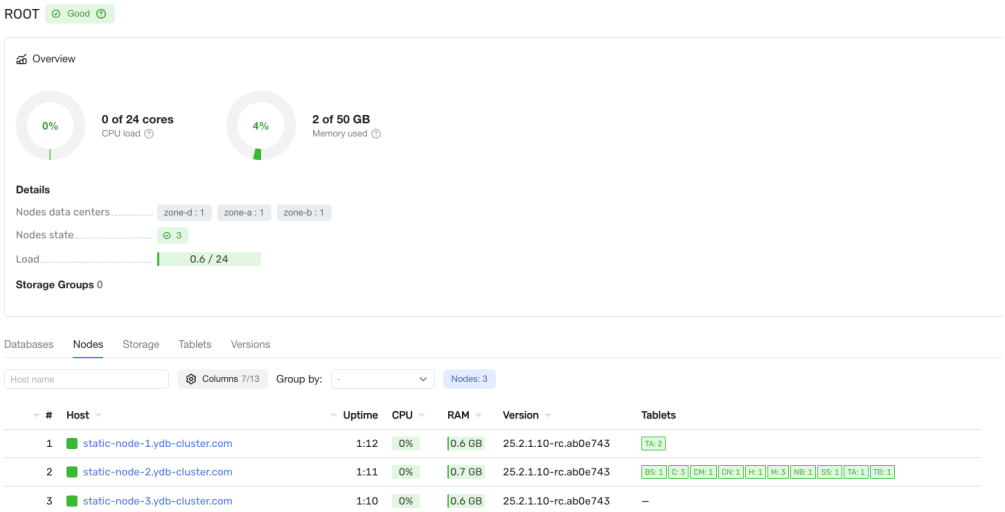
```
[Unit]
Description=YDB storage node
After=network-online.target rc-local.service
Wants=network-online.target
StartLimitInterval=10
StartLimitBurst=15

[Service]
Restart=always
RestartSec=1
User=ydb
PermissionsStartOnly=true
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=ydbd
SyslogFacility=daemon
SyslogLevel=err
Environment=LD_LIBRARY_PATH=/opt/ydb/lib
ExecStart=/opt/ydb/bin/ydbd server --log-level 3 --syslog --tcp \
    --yaml-config  /opt/ydb/cfg/config.yaml \
    --grpcs-port 2135 --ic-port 19001 --mon-port 8765 \
    --mon-cert /opt/ydb/certs/web.pem --node static
LimitNOFILE=65536
LimitCORE=0
LimitMEMLOCK=3221225472

[Install]
WantedBy=multi-user.target
```

Run the service on each static YDB node:

```
sudo systemctl start ydbd-storage
```

After starting the static nodes, verify they are running via YDB UI:

1. Open `https://<node.ydb.tech>:8765` in your browser, where `<node.ydb.tech>` is the FQDN of the server running any static node;
2. Go to the **Nodes** tab;
3. Ensure all 3 static nodes are listed.



Initialize the Cluster

The cluster initialization operation configures the set of static nodes listed in the cluster configuration file for storing YDB data.

To initialize the cluster, you need the Certificate Authority `ca.crt` file; its path must be specified when running the commands. Before running the commands, copy `ca.crt` to the server where you will execute them.

On one of the storage servers in the cluster, run the following commands:

First, obtain an authentication token for request authorization. Run the command below:

```
/opt/ydb/bin/ydb --ca-file ca.crt -e grpcs://`hostname -f`:2135 -d /Root --user root --no-password auth get-t
oken -f > auth_token
```

Initialize the cluster using the obtained token:

```
export LD_LIBRARY_PATH=/opt/ydb/lib
/opt/ydb/bin/ydbd --ca-file ca.crt -s grpcs://`hostname -f`:2135 -f auth_token \
    admin blobstorage config init --yaml-file  /opt/ydb/cfg/config.yaml
echo $?
```

On successful cluster initialization, the cluster initialization command exit code should be zero.

Create a Database

To work with row-oriented or column-oriented tables, you need to create at least one database and run the process or processes serving it (dynamic nodes).

To run the administrative database creation command, you need the Certificate Authority `ca.crt` file, same as for cluster initialization above.

When creating the database, you set the initial number of storage groups, which determines the available I/O throughput and maximum storage capacity. The number of storage groups can be increased after database creation if needed.

On one of the storage servers in the cluster, run the following commands:

```
export LD_LIBRARY_PATH=/opt/ydb/lib
/opt/ydb/bin/ydbd --ca-file ca.crt -s grpcs://`hostname -f`:2135 -f auth_token \
    admin database /Root/testdb create ssd:8
echo $?
```

On successful database creation, the command exit code should be zero.

The command example above uses the following parameters:

- `/Root` — name of the root domain, automatically generated during cluster initialization;
- `testdb` — name of the database to create;
- `ssd:8` — defines the storage pool for the database and the number of groups in it. The pool name (`ssd`) must match the disk type specified in the cluster configuration (for example, in `default_disk_type`) and is case-insensitive. The number after the colon is the number of storage groups to allocate.

Run Dynamic Nodes

Manually

Run the YDB dynamic node for the `/Root/testdb` database:

```
sudo su - ydb
cd /opt/ydb
export LD_LIBRARY_PATH=/opt/ydb/lib
/opt/ydb/bin/ydbd server --grpcs-port 2136 --grpc-ca /opt/ydb/certs/ca.crt \
    --ic-port 19002 --ca /opt/ydb/certs/ca.crt \
    --mon-port 8766 --mon-cert /opt/ydb/certs/web.pem \
    --yaml-config  /opt/ydb/cfg/config.yaml \
    --tenant /Root/testdb \
    --grpc-cert /opt/ydb/certs/node.crt \
    --grpc-key /opt/ydb/certs/node.key \
    --node-broker grpcs://<ydb-static-node1>:2135 \
    --node-broker grpcs://<ydb-static-node2>:2135 \
    --node-broker grpcs://<ydb-static-node3>:2135 &
```

In the command example above, `<ydb-static-node1>`, `<ydb-static-node2>`, `<ydb-static-node3>` are the FQDNs of any three servers running the cluster's static nodes.

Using systemd

Create a systemd configuration file `/etc/systemd/system/ydbd-testdb.service` using the template below. You can also download the sample file from the repository.

```
[Unit]
Description=YDB testdb dynamic node
After=network-online.target rc-local.service
Wants=network-online.target
StartLimitInterval=10
StartLimitBurst=15

[Service]
Restart=always
RestartSec=1
User=ydb
PermissionsStartOnly=true
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=ydbd
SyslogFacility=daemon
SyslogLevel=err
Environment=LD_LIBRARY_PATH=/opt/ydb/lib
ExecStart=/opt/ydb/bin/ydbd server \
    --grpcs-port 2136 --grpc-ca /opt/ydb/certs/ca.crt \
    --ic-port 19002 --ca /opt/ydb/certs/ca.crt \
    --mon-port 8766 --mon-cert /opt/ydb/certs/web.pem \
    --yaml-config  /opt/ydb/cfg/config.yaml \
    --tenant /Root/testdb \
    --grpc-cert /opt/ydb/certs/node.crt \
    --grpc-key /opt/ydb/certs/node.key \
    --node-broker grpcs://<ydb-static-node1>:2135 \
    --node-broker grpcs://<ydb-static-node2>:2135 \
    --node-broker grpcs://<ydb-static-node3>:2135
LimitNOFILE=65536
LimitCORE=0
LimitMEMLOCK=32212254720

[Install]
WantedBy=multi-user.target
```

In the command example above, `<ydb-static-node1>`, `<ydb-static-node2>`, `<ydb-static-node3>` are the FQDNs of any three servers running the cluster's static nodes.

Run the YDB dynamic node for the `/Root/testdb` database:

```
sudo systemctl start ydbd-testdb
```

Run additional dynamic nodes on other servers for database scaling and fault tolerance.

Account Setup

1. Install the YDB CLI as described in the documentation.
2. Set the password for the `root` account using the token obtained earlier:

   ```
   ydb --ca-file ca.crt -e grpcs://<node.ydb.tech>:2136 -d /Root/testdb --token-file auth_token \
       yql -s 'ALTER USER root PASSWORD "passw0rd"'
   ```

   Replace `passw0rd` with the desired password. Save the password in a separate file. Subsequent commands as the `root` user will use the password passed with the `--password-file <path_to_user_password>` option. You can also save the password in a connection profile, as described in the YDB CLI documentation.
3. Create additional accounts:

```
ydb --ca-file ca.crt -e grpcs://<node.ydb.tech>:2136 -d /Root/testdb --user root --password-file <path_t
o_root_pass_file> \
    yql -s 'CREATE USER user1 PASSWORD "passw0rd"'
```

4. Set account permissions by adding them to built-in groups:

```
ydb --ca-file ca.crt -e grpcs://<node.ydb.tech>:2136 -d /Root/testdb --user root --password-file <path_t
o_root_pass_file> \
    yql -s 'ALTER GROUP `ADMINS` ADD USER user1'
```

In the command examples above, `<node.ydb.tech>` is the FQDN of the server running any dynamic node serving the `/Root/testdb` database. When connecting via SSH to a YDB dynamic node, you can use `grpcs://$(hostname -f):2136` to get the FQDN.

When running account creation and group assignment commands, the YDB CLI client will prompt for the `root` user password. To avoid repeated password entry, create a connection profile as described in the YDB CLI documentation.

Test the Created Database

1. Install the YDB CLI as described in the documentation.
2. Create a test row-oriented table (`test_row_table`) or column-oriented table (`test_column_table`):

Creating a row-oriented table

```
ydb --ca-file ca.crt -e grpcs://<node.ydb.tech>:2136 -d /Root/testdb --user root \
    yql -s 'CREATE TABLE `testdir/test_row_table` (id Uint64, title Utf8, PRIMARY KEY (id));'
```

Creating a column-oriented table

```
ydb --ca-file ca.crt -e grpcs://<node.ydb.tech>:2136 -d /Root/testdb --user root \
    yql -s 'CREATE TABLE `testdir/test_column_table` (id Uint64 NOT NULL, title Utf8, PRIMARY KEY (id)) WITH
(STORE = COLUMN);'
```

Where `<node.ydb.tech>` is the FQDN of the server running the dynamic node serving the `/Root/testdb` database.

Checking Access to YDB UI

To check access to YDB UI, open `https://<node.ydb.tech>:8765` in your browser, where `<node.ydb.tech>` is the FQDN of the server running any static YDB node.

Configure your browser to trust the Certificate Authority that issued certificates for the YDB cluster. Otherwise, you will see a warning about an untrusted certificate.

If authentication is enabled in the cluster, the browser will prompt for login and password. After entering valid credentials, the YDB UI welcome page will appear. The available features and user interface are described in Using YDB UI.

Note

A common way to provide access to YDB UI is to set up a fault-tolerant HTTP balancer using `haproxy` , `nginx` , or similar software. HTTP balancer configuration details are beyond the scope of the standard YDB installation guide.

Installing YDB in Unprotected Mode

Warning

We do not recommend using the unprotected YDB mode for production or application development.

The installation procedure above assumes YDB deployment in the standard protected mode.

The unprotected YDB mode is intended for test scenarios, primarily related to YDB software development and testing. In unprotected mode:

- Traffic between cluster nodes and between applications and the cluster uses unencrypted connections;
- User authentication is not used (enabling authentication without traffic encryption makes no sense, since the login and password would be transmitted over the network in plain text).

To install YDB for operation in unprotected mode, follow the procedure above with the following exceptions:

1. When preparing for installation, you do not need to generate TLS certificates and keys or copy certificates and keys to the cluster nodes.
2. Remove the `security_config` , `interconnect_config` , and `grpc_config` sections from the cluster node configuration files.
3. Use simplified commands to run static and dynamic cluster nodes: omit options specifying certificate and key file names; use the `grpc` protocol instead of `grpcs` when specifying connection endpoints.
4. Skip the authentication token step before cluster initialization and database creation, as it is not needed in unprotected mode.
5. The cluster initialization command has the following format:

```
export LD_LIBRARY_PATH=/opt/ydb/lib
/opt/ydb/bin/ydbd admin blobstorage config init --yaml-file  /opt/ydb/cfg/config.yaml
echo $?
```

6. The database creation command has the following format:

```
export LD_LIBRARY_PATH=/opt/ydb/lib
/opt/ydb/bin/ydbd admin database /Root/testdb create ssd:1
```

7. When accessing the database from the YDB CLI and applications, use grpc instead of grpcs and do not use authentication.

Deploying a Cluster Using Configuration V2

> **Alert**
>
> This article covers YDB clusters that use **configuration V2**. This configuration method is experimental and is only available for YDB versions starting with v25.1. For production use, we recommend choosing configuration V1 — it is the main and officially supported configuration for all YDB clusters.

Prepare the Environment

Before deploying the system, complete the preparatory steps. Review the Deployment Preparation document.

Prepare Configuration Files

Prepare the YDB configuration file:

```
metadata:
  kind: MainConfig
  cluster: ""
  version: 0
config:
  erasure: mirror-3-dc
  fail_domain_type: disk
  self_management_config:
    enabled: true
  default_disk_type: SSD
  host_configs:
  - host_config_id: 1
    drive:
    - path: /dev/disk/by-partlabel/ydb_disk_ssd_01
      type: SSD
    - path: /dev/disk/by-partlabel/ydb_disk_ssd_02
      type: SSD
    - path: /dev/disk/by-partlabel/ydb_disk_ssd_03
      type: SSD
  hosts:
  - host: ydb-node-zone-a.local
    host_config_id: 1
    location:
      body: 1
      data_center: 'zone-a'
      rack: '1'
  - host: ydb-node-zone-b.local
    host_config_id: 1
    location:
      body: 2
      data_center: 'zone-b'
      rack: '2'
  - host: ydb-node-zone-c.local
    host_config_id: 1
    location:
      body: 3
      data_center: 'zone-c'
      rack: '3'
  actor_system_config:
    use_auto_config: true
    cpu_count: 8
  interconnect_config:
    start_tcp: true
    encryption_mode: OPTIONAL
    path_to_certificate_file: "/opt/ydb/certs/node.crt"
    path_to_private_key_file: "/opt/ydb/certs/node.key"
    path_to_ca_file: "/opt/ydb/certs/ca.crt"
  grpc_config:
    cert: "/opt/ydb/certs/node.crt"
    key: "/opt/ydb/certs/node.key"
    ca: "/opt/ydb/certs/ca.crt"
    services_enabled:
    - legacy
  security_config:
    enforce_user_token_requirement: true
    monitoring_allowed_sids:
    - "root"
    - "ADMINS"
    - "DATABASE-ADMINS"
    administration_allowed_sids:
    - "root"
    - "ADMINS"
    - "DATABASE-ADMINS"
    viewer_allowed_sids:
    - "root"
    - "ADMINS"
    - "DATABASE-ADMINS"
    register_dynamic_node_allowed_sids:
    - databaseNodes@cert
    - root@builtin
    bootstrap_allowed_sids:
    - "root"
    - "ADMINS"
```

```
    - "DATABASE-ADMINS"
  client_certificate_authorization:
    request_client_certificate: true
    client_certificate_definitions:
        - member_groups: ["ADMINS"]
          subject_terms:
          - short_name: "O"
            values: ["YDB"]
```

To speed up and simplify the initial deployment of YDB, the configuration file already contains most of the cluster setup settings. You only need to replace the default host FQDNs in the `hosts` section and disk paths in the `host_configs` section with your actual values.

- The `hosts` section:

```
...
hosts:
- host: ydb-node-zone-a.local
  host_config_id: 1
  location:
    body: 1
    data_center: 'zone-a'
    rack: '1'
...
```

- The `host_configs` section:

```
...
host_configs:
- host_config_id: 1
  drive:
  - path: /dev/disk/by-partlabel/ydb_disk_ssd_01
    type: SSD
  - path: /dev/disk/by-partlabel/ydb_disk_ssd_02
    type: SSD
  - path: /dev/disk/by-partlabel/ydb_disk_ssd_03
    type: SSD
...
```

Leave the other sections and configuration file settings unchanged.

Save the YDB configuration file as `/tmp/config.yaml` on each cluster server.

For more information on creating the configuration file, see YDB Cluster Configuration.

Copy TLS Keys and Certificates to Each Server

Copy the prepared TLS keys and certificates to a secure directory on each YDB cluster node. Below is an example of commands to create a secure directory and copy the key and certificate files.

```
sudo mkdir -p /opt/ydb/certs
sudo cp -v ca.crt /opt/ydb/certs/
sudo cp -v node.crt /opt/ydb/certs/
sudo cp -v node.key /opt/ydb/certs/
sudo cp -v web.pem /opt/ydb/certs/
sudo chown -R ydb:ydb /opt/ydb/certs
sudo chmod 700 /opt/ydb/certs
```

Prepare Configuration on Static Cluster Nodes

Create an empty `opt/ydb/cfg` directory on each machine for the cluster to work with the configuration. If running multiple cluster nodes on a single machine, create separate directories for each node.

```
sudo mkdir -p /opt/ydb/cfg
sudo chown -R ydb:ydb /opt/ydb/cfg
```

Run a special command on each machine to initialize this directory with the configuration file.

```
sudo /opt/ydb/bin/ydb admin node config init --config-dir /opt/ydb/cfg --from-config /tmp/config.yaml
```

After running this command, the source file `/tmp/config.yaml` is no longer used and can be removed.

Start Static Nodes

Manually

Start the YDB storage service on each static cluster node:

```
sudo su - ydb
cd /opt/ydb
export LD_LIBRARY_PATH=/opt/ydb/lib
/opt/ydb/bin/ydbd server --log-level 3 --syslog --tcp --yaml-config  /opt/ydb/cfg/config.yaml \
    --grpcs-port 2135 --ic-port 19001 --mon-port 8765 --mon-cert /opt/ydb/certs/web.pem --node static
```

Using systemd

Create the systemd configuration file `/etc/systemd/system/ydbd-storage.service` on each server that will host a static cluster node, using the template below. You can also download the template from the repository.

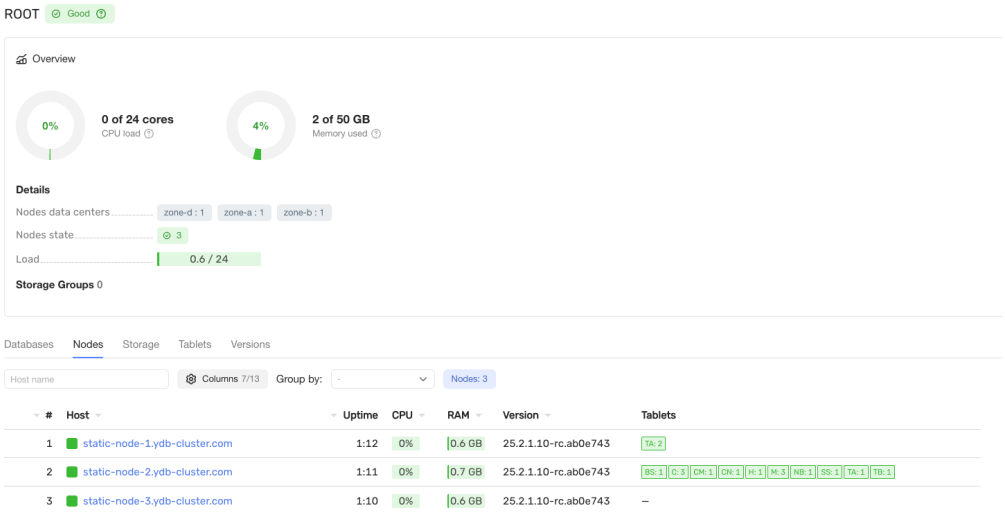
```
[Unit]
Description=YDB storage node
After=network-online.target rc-local.service
Wants=network-online.target
StartLimitInterval=10
StartLimitBurst=15

[Service]
Restart=always
RestartSec=1
User=ydb
PermissionsStartOnly=true
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=ydbd
SyslogFacility=daemon
SyslogLevel=err
Environment=LD_LIBRARY_PATH=/opt/ydb/lib
ExecStart=/opt/ydb/bin/ydbd server --log-level 3 --syslog --tcp \
    --yaml-config  /opt/ydb/cfg/config.yaml \
    --grpcs-port 2135 --ic-port 19001 --mon-port 8765 \
    --mon-cert /opt/ydb/certs/web.pem --node static
LimitNOFILE=65536
LimitCORE=0
LimitMEMLOCK=3221225472

[Install]
WantedBy=multi-user.target
```

Start the service on each YDB static node:

```
sudo systemctl start ydbd-storage
```

After starting the static nodes, verify they are working via YDB UI:

1. Open `https://<node.ydb.tech>:8765` in your browser, where `<node.ydb.tech>` is the FQDN of the server running any static node;
2. Go to the **Nodes** tab;
3. Make sure all 3 static nodes are displayed in the list.



Initialize the Cluster

The cluster initialization operation configures the set of static nodes listed in the cluster configuration file for YDB data storage.

You will need the Certificate Authority (CA) certificate file `ca.crt` for cluster initialization; its path must be specified when running the relevant commands. Before running these commands, copy the `ca.crt` file to the server where they will be executed.

On one of the storage servers in the cluster, run the commands:

Initialize the cluster using the obtained token

```
export LD_LIBRARY_PATH=/opt/ydb/lib
/opt/ydb/bin/ydb --ca-file ca.crt \
    --client-cert-file node.crt \
    --client-cert-key-file node.key \
    -e grpcs://`hostname -f`:2135 \
    admin cluster bootstrap --uuid <string>
echo $?
```

After initializing the cluster, you must obtain an authentication token before running further administrative commands.

```
/opt/ydb/bin/ydb -e grpcs://`hostname -f`:2135 -d /Root --ca-file ca.crt \
--user root --no-password auth get-token --force > auth_token
```

If the cluster initialization completes successfully, the command exit code should be zero.

Create a Database

To work with row or column tables, you need to create at least one database and start the process or processes that serve it (dynamic nodes).

You will need the Certificate Authority (CA) certificate file `ca.crt` to run the administrative command for creating a database, similar to the cluster initialization procedure described above.

When creating a database, you set the initial number of storage groups, which determines the available I/O throughput and maximum storage capacity. The number of storage groups can be increased after database creation if needed.

On one of the storage servers in the cluster, run the commands:

```
export LD_LIBRARY_PATH=/opt/ydb/lib
/opt/ydb/bin/ydbd --ca-file ca.crt -s grpcs://`hostname -f`:2135 -f auth_token \
    admin database /Root/testdb create ssd:8
echo $?
```

If the database is created successfully, the command exit code should be zero.

The example commands above use the following parameters:

- `/Root` — the root domain name, generated automatically during cluster initialization;
- `testdb` — the name of the database being created;
- `ssd:8` — specifies the storage pool for the database and the number of groups in it. The pool name (`ssd`) must match the disk type specified in the cluster configuration (for example, in `default_disk_type`) and is case-insensitive. The number after the colon is the number of allocated storage groups.

Start Dynamic Nodes

Manually

Start the YDB dynamic node for the `/Root/testdb` database:

```
sudo su - ydb
cd /opt/ydb
export LD_LIBRARY_PATH=/opt/ydb/lib
/opt/ydb/bin/ydbd server --grpcs-port 2136 --grpc-ca /opt/ydb/certs/ca.crt \
    --ic-port 19002 --ca /opt/ydb/certs/ca.crt \
    --mon-port 8766 --mon-cert /opt/ydb/certs/web.pem \
    --yaml-config  /opt/ydb/cfg/config.yaml \
    --tenant /Root/testdb \
    --grpc-cert /opt/ydb/certs/node.crt \
    --grpc-key /opt/ydb/certs/node.key \
    --node-broker grpcs://<ydb-static-node1>:2135 \
    --node-broker grpcs://<ydb-static-node2>:2135 \
    --node-broker grpcs://<ydb-static-node3>:2135
```

In the example above, `<ydb-static-node1>`, `<ydb-static-node2>`, and `<ydb-static-node3>` are the FQDNs of any three servers running static cluster nodes.

Using systemd

Create the systemd configuration file `/etc/systemd/system/ydbd-testdb.service` using the template below. You can also download the template from the repository.

```
[Unit]
Description=YDB testdb dynamic node
After=network-online.target rc-local.service
Wants=network-online.target
StartLimitInterval=10
StartLimitBurst=15

[Service]
Restart=always
RestartSec=1
User=ydb
PermissionsStartOnly=true
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=ydbd
SyslogFacility=daemon
SyslogLevel=err
Environment=LD_LIBRARY_PATH=/opt/ydb/lib
ExecStart=/opt/ydb/bin/ydbd server \
    --grpcs-port 2136 --grpc-ca /opt/ydb/certs/ca.crt \
    --ic-port 19002 --ca /opt/ydb/certs/ca.crt \
    --mon-port 8766 --mon-cert /opt/ydb/certs/web.pem \
    --yaml-config  /opt/ydb/cfg/config.yaml \
    --tenant /Root/testdb \
    --grpc-cert /opt/ydb/certs/node.crt \
    --grpc-key /opt/ydb/certs/node.key \
```

```
    --node-broker grpcs://<ydb-static-node1>:2135 \
    --node-broker grpcs://<ydb-static-node2>:2135 \
    --node-broker grpcs://<ydb-static-node3>:2135
LimitNOFILE=65536
LimitCORE=0
LimitMEMLOCK=32212254720

[Install]
WantedBy=multi-user.target
```

In the example above, `<ydb-static-node1>`, `<ydb-static-node2>`, and `<ydb-static-node3>` are the FQDNs of any three servers running static cluster nodes.

Start the YDB dynamic node for the `/Root/testdb` database:

```
sudo systemctl start ydbd-testdb
```

Start additional dynamic nodes on other servers to scale and ensure database fault tolerance.

Configure User Accounts

1. Set a password for the `root` account using the token obtained earlier:

   ```
   ydb --ca-file ca.crt -e grpcs://<node.ydb.tech>:2136 -d /Root/testdb --token-file auth_token \
       yql -s 'ALTER USER root PASSWORD "passw0rd"'
   ```

   Replace `passw0rd` with your desired password. Save the password in a separate file. Subsequent commands as the `root` user will use the password passed via the `--password-file <path_to_user_password>` option. You can also save the password in a connection profile, as described in the YDB CLI documentation.
2. Create additional user accounts:

   ```
   ydb --ca-file ca.crt -e grpcs://<node.ydb.tech>:2136 -d /Root/testdb --user root --password-file <path_t
   o_root_pass_file> \
       yql -s 'CREATE USER user1 PASSWORD "passw0rd"'
   ```
3. Set account permissions by adding them to built-in groups:

   ```
   ydb --ca-file ca.crt -e grpcs://<node.ydb.tech>:2136 -d /Root/testdb --user root --password-file <path_t
   o_root_pass_file> \
       yql -s 'ALTER GROUP `ADMINS` ADD USER user1'
   ```

In the examples above, `<node.ydb.tech>` is the FQDN of the server running any dynamic node that serves the `/Root/testdb` database. When connecting via SSH to the YDB dynamic node, you can use `grpcs://$(hostname -f):2136` to get the FQDN.

When running user account creation and group assignment commands, the YDB CLI client will prompt for the `root` user password. To avoid entering the password multiple times, create a connection profile as described in the YDB CLI documentation.

Test Working with the Created Database

1. Install YDB CLI as described in the documentation.
2. Create a test row table (`test_row_table`) or column table (`test_column_table`):

Creating a row table

```
ydb --ca-file ca.crt -e grpcs://<node.ydb.tech>:2136 -d /Root/testdb --user root \
    yql -s 'CREATE TABLE `testdir/test_row_table` (id Uint64, title Utf8, PRIMARY KEY (id));'
```

Creating a column table

```
ydb --ca-file ca.crt -e grpcs://<node.ydb.tech>:2136 -d /Root/testdb --user root \
    yql -s 'CREATE TABLE `testdir/test_column_table` (id Uint64 NOT NULL, title Utf8, PRIMARY KEY (id)) WITH
(STORE = COLUMN);'
```

Where `<node.ydb.tech>` is the FQDN of the server running the dynamic node that serves the `/Root/testdb` database.

Verifying Access to YDB UI

To verify access to YDB UI, open `https://<node.ydb.tech>:8765` in a web browser, where `<node.ydb.tech>` is the FQDN of the server running any YDB static node.

The web browser must be configured to trust the Certificate Authority that issued the certificates for the YDB cluster; otherwise, a warning about using an untrusted certificate will be displayed.

If authentication is enabled in the cluster, the web browser will display a login and password prompt. After entering valid authentication credentials, the YDB UI home page should appear. A description of the available features and user interface is provided in Using YDB UI.

> **Note**
>
> Typically, to provide access to YDB UI, a fault-tolerant HTTP load balancer is configured using software such as `haproxy`, `nginx`, or similar. HTTP load balancer configuration details are beyond the scope of the standard YDB installation instructions.

Installing YDB in Insecure Mode

> **Warning**
>
> We do not recommend using YDB in insecure mode for either production or application development.

The installation procedure described above deploys YDB in the standard secure mode.

The insecure mode of YDB is intended for test tasks, primarily related to YDB software development and testing. In insecure mode:

- traffic between cluster nodes, as well as between applications and the cluster, uses unencrypted connections;
- user authentication is not used (enabling authentication without traffic encryption would be pointless, since login and password would be transmitted over the network in plain text).

To install YDB for operation in insecure mode, follow the procedure described above with the following exceptions:

1. No TLS certificates and keys need to be created during preparation, and certificates and keys are not copied to cluster nodes.
2. The `security_config`, `interconnect_config`, and `grpc_config` sections are removed from cluster node configuration files.
3. Use simplified commands for starting static and dynamic cluster nodes: omit options with certificate and key file names, and use the `grpc` protocol instead of `grpcs` when specifying connection endpoints.
4. Skip the authentication token step before cluster initialization and database creation, as it is not needed in insecure mode.
5. Run the cluster initialization command in the following form:

   ```
   export LD_LIBRARY_PATH=/opt/ydb/lib
   ydb admin cluster bootstrap --uuid <string>
   echo $?
   ```

6. Run the database creation command in the following form:

   ```
   export LD_LIBRARY_PATH=/opt/ydb/lib
   /opt/ydb/bin/ydbd admin database /Root/testdb create ssd:1
   ```

7. When accessing the database from YDB CLI and applications, use the grpc protocol instead of grpcs, and do not use authentication.

Updating configuration of manually deployed YDB clusters

When manually deploying a YDB cluster, configuration management is performed through YDB CLI. This article covers methods for changing cluster configuration after initial deployment.

Basic configuration operations

Getting current configuration

To get the current cluster configuration, use the command:

```
ydb -e grpcs://<endpoint>:2135 admin cluster config fetch > config.yaml
```

Use the address of any cluster node as `<endpoint>`.

Applying new configuration

To upload updated configuration to the cluster, use the following command:

```
ydb -e grpcs://<endpoint>:2135 admin cluster config replace -f config.yaml
```

Some configuration parameters are applied on the fly after executing the command, however some require performing the cluster restart procedure.

Cluster maintenance

The ydbops utility uses CMS to perform cluster maintenance without losing availability. You can also use CMS directly through the gRPC API.

Rolling restart

To perform a rolling restart of the entire cluster, you can use the command:

```
$ ydbops restart
```

By default, the `strong` availability mode will be used, minimizing the risk of losing availability. You can override it using the `--availability-mode` parameter.

The `ydbops` utility will automatically create a maintenance task to restart the entire cluster using the specified availability mode. As it progresses, `ydbops` will update the maintenance task and obtain exclusive locks on nodes in CMS until all nodes are restarted.

Take a host out for maintenance

To take a host out for maintenance, follow these steps:

1. Create a maintenance task using the command:

   ```
   $ ydbops maintenance create --hosts=<fqdn> --duration=<seconds>
   ```

   This command will create a maintenance task that will take an exclusive lock on the host with the fully qualified domain name `<fqdn>` for `<seconds>` seconds.
2. After creating the task, you need to update its state until the lock is taken, using the command:

   ```
   $ ydbops maintenance refresh --task-id=<id>
   ```

   This command will update the task with identifier `<id>` and attempt to take the required lock. When you receive a `PERFORMED` response, you can proceed to the next step.
3. Perform host maintenance while the lock is held.
4. After completing the work, you need to release the lock on the host using the command:

   ```
   $ ydbops maintenance complete --task-id=<id> --hosts=<fqdn>
   ```

Updating YDB

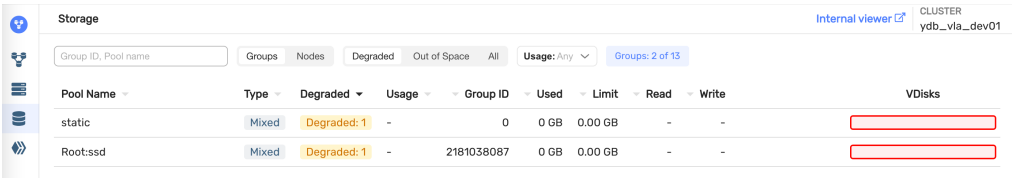
YDB is a distributed system that supports rolling restart without downtime or performance degradation.

Update Procedure

The basic scenario is updating the executable file and restarting each node one by one:

1. Updating and restarting storage nodes.
2. Updating and restarting dynamic nodes.

The shutdown and startup process is described on the Safe restart and shutdown of nodes page.
You must update YDB nodes one by one and monitor the cluster status after each step in YDB Monitoring: make sure the `Storage` tab has no pools in the `Degraded` status (as shown in the example below). Otherwise, stop the update process.



Version Compatibility

All minor versions within a major version are compatible for updates. Major versions are consecutively compatible. To update to the next major version, you must first update to the latest minor release of the current major version. For example:

- `X.Y.* → X.Y.*`: Update is possible, all minor versions within a single major version are compatible.
- `X.Y.Z` (the latest available version in `X.Y.*`) → `X.Y+1.*` : Update is possible, major versions are consistent.
- `X.Y.* → X.Y+2.*`: Update is impossible, major versions are inconsistent.
- `X.Y.* → X.Y-2.*`: Update is impossible, major versions are inconsistent.

A list of available versions can be found on the download page. The YDB release policy is described in more details in the Release management article of the YDB development documentation.

> **Warning**
>
> Also, in any case, you cannot roll back more than 2 major versions relative to the version that was deployed at least once. This is because such an old version may not know how to work with data on the disks that the newer version persisted.

Examples of Version Compatibility

- `v.22.2.5 → v.22.2.47`: Update is possible.
- `v.22.2.47 → v.22.3.21`: Update is possible.
- `v.22.2.40 → v.22.3.21`: Update is impossible, first upgrade to the latest minor version (v.22.2.47).
- `v.22.2.47 → v.22.4.5`: Update is impossible, upgrade to the next major version first (v.22.3.*).

Checking Update Results

You can check the updated node versions on the `Nodes` page in Monitoring.

Database node authentication and authorization

Node authentication in the YDB cluster ensures that database nodes are authenticated when making service requests to other nodes via the gRPC protocol. Node authorization ensures that the privileges required by the service requests are checked and granted during request processing. These service calls include database node registration within the cluster and access to dynamic configuration information. The use of node authorization is recommended for all YDB clusters, as it helps prevent unauthorized access to data by adding nodes to the cluster.

Database node authentication and authorization are performed in the following order:

1. The database node being started opens a gRPC connection to one of the cluster storage nodes specified in the `--node-broker` command-line option. The connection uses the TLS protocol, and the certificate of the running node is used as the client certificate for the connection.
2. The storage node and the database node perform mutual authentication checks using the TLS protocol: the certificate trust chain is checked, and the hostname is matched against the value of the "Subject Name" field of the certificate.
3. The storage node checks the "Subject" field of the certificate for compliance with the requirements set up through settings in the static configuration.
4. If the above checks are successful, the connection from the database node is considered authenticated, and it is assigned a security identifier - SID, which is determined by the settings.
5. The database node uses the established gRPC connection to register with the cluster through the corresponding service request. When registering, the database node sends its network address intended to be used for communication with other cluster nodes.
6. The storage node checks whether the SID assigned to the gRPC connection is in the list of acceptable ones. If this check is successful, the storage node registers the database node within the cluster, saving the association between the network address of the registered node and its identifier.
7. The database node joins the cluster by connecting via its network address and providing the node ID it received during registration. Attempts to join the cluster by nodes with unknown network addresses or IDs are blocked by other nodes.

Below are the steps required to enable the node authentication and authorization feature.

Configuration prerequisites

1. The deployed YDB cluster must have gRPC traffic encryption configured to use the TLS protocol.
2. When preparing node certificates for a cluster where you plan to use the node authorization feature, uniform rules must be used for populating the "Subject" field of the certificates. This allows the identification of certificates issued for the cluster nodes. For more information, see the certificate verification rules documentation.

   > **Note**
   >
   > The proposed example script generates self-signed certificates for YDB nodes and ensures that the "Subject" field is populated with the value `O=YDB` for all node certificates. The configuration examples provided below are prepared for certificates with this specific "Subject" field configuration, but feel free to use your real organization name instead.

3. The command-line parameters for starting database nodes must include options that specify the paths to the trusted CA certificate, the node certificate, and the node key files. The required additional command-line parameters are:

Command-line option	Description
`--grpc-ca`	Path to the trusted certification authority file `ca.crt`
`--grpc-cert`	Path to the node certificate file `node.crt`
`--grpc-key`	Path to the node secret key file `node.key`

   Below is an example of the complete command to start the database node, including the extra options for gRPC TLS key and certificate files:

   ```
   /opt/ydb/bin/ydbd server --yaml-config  /opt/ydb/cfg/config.yaml --tenant /Root/testdb \
       --grpcs-port 2136 --grpc-ca /opt/ydb/certs/ca.crt \
       --grpc-cert /opt/ydb/certs/node.crt --grpc-key /opt/ydb/certs/node.key \
       --ic-port 19002 --ca /opt/ydb/certs/ca.crt \
       --mon-port 8766 --mon-cert /opt/ydb/certs/web.pem \
       --node-broker grpcs://<ydb1>:2135 \
       --node-broker grpcs://<ydb2>:2135 \
       --node-broker grpcs://<ydb3>:2135
   ```

Enabling database node authentication and authorization

To enable mandatory database node authorization, add the following configuration blocks to the static cluster configuration file:

1. At the root level, add the `client_certificate_authorization` block to define the requirements for the "Subject" field of trusted node certificates. For example:

   ```
   client_certificate_authorization:
     request_client_certificate: true
     client_certificate_definitions:
       - member_groups: ["registerNode@cert"]
         subject_terms:
         - short_name: "O"
           values: ["YDB"]
   ```

   Add other certificate validation settings as defined in the documentation, if required.

   If the certificate is successfully verified and the components of the "Subject" field comply with the requirements defined in the `subject_terms` sub-block, the connection will be assigned the access subjects listed in the `member_groups` parameter. To distinguish these subjects from other user groups and accounts, their names typically have the `@cert` suffix.

2. Add the `register_dynamic_node_allowed_sids` element to the cluster authentication settings `security_config` block, and list the subjects permitted for database node registration. Example:

```
domains_config:
    ...
    security_config:
        enforce_user_token_requirement: true
        monitoring_allowed_sids:
        ...
        administration_allowed_sids:
        ...
        viewer_allowed_sids:
        ...
        register_dynamic_node_allowed_sids:
        - "registerNode@cert"
```

   For more detailed information on configuring cluster authentication parameters, see the relevant documentation section.

3. In the `auth_config` section of the cluster configuration, add the `node_registration_token` as an empty string. Example:

```
auth_config:
  ...
  node_registration_token: ""
```

4. Deploy the static configuration files on all cluster nodes either manually, or using the Ansible playbook action.
5. Perform the rolling restart of storage nodes by using ydbops or Ansible playbook action.
6. Perform the rolling restart of database nodes through ydbops or Ansible playbooks.

Managing a cluster in bridge mode

> **Feature of Yandex Enterprise Database**
>
> This functionality is available only in the Yandex Enterprise Database. In the open-source version of YDB it is absent.

Below are typical operations for a cluster in bridge mode using the corresponding YDB CLI commands.

View current state

Shows the current state of each pile configured on the YDB cluster.

```
ydb admin cluster bridge list
```

Example output:

```
pile-a: PRIMARY
pile-b: SYNCHRONIZED
```

Planned `PRIMARY` change (switchover)

If planned maintenance is scheduled in the foreseeable future in the data center or on the equipment where the current `PRIMARY` pile is running, it is recommended to switch the cluster to use another pile as `PRIMARY` in advance. Select another pile in the `SYNCHRONIZED` state to switch it to the `PRIMARY` state with the following command:

```
ydb admin cluster bridge switchover --new-primary <pile>
```

The switchover is performed smoothly: roles go through `PRIMARY/PROMOTED` and end in the `SYNCHRONIZED/PRIMARY` state.

Planned pile disconnection (takedown)

If planned maintenance will make one of the pile unavailable, it must be taken out of the cluster before starting using the following command:

```
ydb admin cluster bridge takedown --pile <pile>
# if disconnecting the current PRIMARY:
ydb admin cluster bridge takedown --pile <current-primary> --new-primary <synchronized-pile>
```

When the operation is performed, the pile is transitioned to `SUSPENDED`, then to `DISCONNECTED`; further cluster operations are performed without the disconnected pile.

If disconnecting the current `PRIMARY` and it was not possible to change it in advance, these operations can be combined by specifying the new `PRIMARY` in the `--new-primary` argument, which must be in the `SYNCHRONIZED` state.

```
ydb admin cluster bridge takedown --pile <pile>
# if disconnecting the current PRIMARY:
ydb admin cluster bridge takedown --pile <current-primary> --new-primary <synchronized-pile>
```

> **Warning**
>
> Before starting planned maintenance, always verify using the list command that the pile disconnection operation has completed successfully and all pile are in the expected state.

Emergency disconnection of unavailable pile (failover)

Since synchronous replication operates between pile, when one of them unexpectedly fails, cluster operation stops by default, and a decision must be made whether to continue cluster operation without this pile. This decision can be made by a person (for example, an on-call DevOps engineer) or by automation external to the YDB cluster.

If the decision is to continue cluster operation, run the following command:

```
ydb admin cluster bridge failover --pile <unavailable-pile>
```

If the current `PRIMARY` is unavailable, you must add the `--new-primary` parameter with the name of a pile in the `SYNCHRONIZED` state. If the parameter is not specified or is specified incorrectly, the command will fail with an error without any changes to the cluster.

```
ydb admin cluster bridge failover --pile <unavailable-pile>
# if the current PRIMARY is unavailable:
ydb admin cluster bridge failover --pile <unavailable-primary> --new-primary <synchronized-pile>
```

The unavailable pile will be transitioned to the `DISCONNECTED` state, and when a new `PRIMARY` is specified, that role will be switched. If other pile are in states other than `SYNCHRONIZED`, emergency disconnection can also be performed. Valid transitions depend on the current state pair and are shown on the state diagram and in the transition table.

> **Warning**
>
> If mandatory authentication is enabled on the cluster, login and password authentication cannot be used while the cluster is in an emergency state — use client certificate authentication instead. For details, see Authentication specifics when the cluster is in an emergency state.

Return pile to the cluster (rejoin)

After planned maintenance is complete or the causes of the failure have been resolved, previously disconnected pile must be explicitly brought back into operation with the following command:

```
ydb admin cluster bridge rejoin --pile <pile>
```

Immediately after the operation starts, the pile transitions to the `NOT_SYNCHRONIZED` state and a background data synchronization process starts; when synchronization completes, the pile automatically becomes `SYNCHRONIZED`. After waiting for this state, you can switch the `PRIMARY` role to this pile if needed.

Authentication specifics when the cluster is in an emergency state

Bridge cluster management commands require cluster administrator privileges. If mandatory authentication is enabled (`enforce_user_token_requirement: true`), all authentication methods are available during normal cluster operation, including login and password authentication.

However, when a pile fails, the cluster suspends serving requests until failover is performed. In this state, login and password authentication cannot be used: validating credentials requires access to SchemeShard, which is unavailable while the cluster is in an emergency state. A login attempt fails with the following error:

```
SchemeShard is unreachable
```

The only authentication method available while the cluster is in an emergency state is client certificate authentication. The client certificate is verified locally by the node that received the request, so this check does not depend on the availability of the cluster as a whole.

Client certificate authentication must be configured on the cluster in advance:

1. The `client_certificate_authorization` section of the cluster configuration must assign an administrative group SID to connections with a trusted client certificate:

   ```
   client_certificate_authorization:
     request_client_certificate: true
     client_certificate_definitions:
       - member_groups: ["ADMINS"]
         subject_terms:
         - short_name: "O"
           values: ["YDB"]
   ```

2. This SID must be included in the `administration_allowed_sids` list of the `security_config` section:

   ```
   security_config:
     enforce_user_token_requirement: true
     administration_allowed_sids:
     - "root"
     - "ADMINS"
   ```

If the cluster is deployed following the deployment guide, these settings are already included in the configuration, and the `node.crt` and `node.key` files from the `~/CA/certs/` directory on any cluster node can be used as the client certificate.

To authenticate with a client certificate, specify it in the YDB CLI global options `--client-cert-file` and `--client-cert-key-file`. Connect to a node of a healthy pile. Failover example:

```
ydb -e grpcs://<node.ydb.tech>:2135 \
    --ca-file ca.crt \
    --client-cert-file node.crt \
    --client-cert-key-file node.key \
    admin cluster bridge failover --pile <unavailable-pile> --new-primary <synchronized-pile>
```

where:

- `<node.ydb.tech>` — FQDN of a node in a healthy pile;
- `ca.crt` — certificate of the cluster's trusted certificate authority;
- `node.crt`, `node.key` — client certificate and its private key that satisfy the requirements of the `client_certificate_authorization` section.

> **Warning**
>
> Verify that client certificate authentication works in advance, before an emergency occurs. If client certificate authentication was not configured before the emergency, bringing the cluster out of the emergency state becomes significantly more complicated and requires manual intervention on the cluster nodes.

Cluster Disk Subsystem Management Overview

Managing a cluster's disk subsystem includes the following actions:

- Configuration changes:
  - Cluster Expansion.
  - Adding storage groups.
- Maintenance:
  - Replacing Node FQDN.
  - Safe restart and shutdown of nodes.
  - Enabling/disabling Scrubbing.
  - SelfHeal.
  - Data Center Decommissioning.
  - Decommissioning groups using virtual groups.
  - Moving VDisks.
  - Blob Depot.
  - Group Decommissioning.
- Troubleshooting:
  - Staying within the failure model.
  - Disk load balancing.
  - Freeing up space on physical devices.

Adding storage groups

As the amount of stored data grows, you may need to add disks to your YDB cluster. You can add disks either to existing nodes or along with new nodes. To make the resources of new disks available to the database, add storage groups.

To add new storage groups, use YDB DSTool.

View a list of cluster storage pools:

```
ydb-dstool -e <bs_endpoint> pool list
```

Result example:

```
┌──────────────┬──────────────────┬─────────────────┬──────┬──────────────┬──────────────┐
| BoxId:PoolId | PoolName         | ErasureSpecies | Kind | Groups_TOTAL | VDisks_TOTAL |
├──────────────┼──────────────────┼─────────────────┼──────┼──────────────┼──────────────┤
| [1:1]        | /Root/testdb:ROT | mirror-3-dc    | ROT  | 1            | 9            |
└──────────────┴──────────────────┴─────────────────┴──────┴──────────────┴──────────────┘
```

The command below adds 10 groups to the `/Root/testdb:ROT` pool:

```
ydb-dstool -e <bs_endpoint> group add --pool-name /Root/testdb:ROT --groups 10
```

If successful, the command returns a zero `exit status`. Or else, it returns a non-zero exit status and outputs an error message to `stderr`.

To check if groups can be added without actually adding them, use the `--dry-run` global parameter. The command below checks if 100 groups can be added to the `/Root/testdb:ROT` pool:

```
ydb-dstool --dry-run -e <bs_endpoint> group add --pool-name /Root/testdb:ROT --groups 100
```

The `--dry-run` parameter lets you estimate the maximum number of groups that you can add to the pool.

Safe restart and shutdown of nodes

Stopping/restarting a YDB process on a node

To make sure that the process is stoppable, follow these steps.

1. Access the node via SSH.
2. Execute the command

   ```
   ydbd cms request restart host {node_id} --user {user} --duration 60 --dry --reason 'some-reason'
   ```

   If the process is stoppable, you'll see `ALLOW`.
3. Stop the process

   ```
   sudo service ydbd stop
   ```

4. Restart the process if needed

   ```
   sudo service ydbd start
   ```

Replacing equipment

Before replacing equipment, make sure that the YDB process is stoppable.
If the replacement is going to take a long time, first move all the VDisks from this node and wait until replication is complete.
After replication is complete, you can safely shut down the node.

To make sure that disabling the dynamic node doesn't affect query handling, drain the tablets from this node first.

Go to the Hive web-viewer page.
Click "View Nodes" to see a list of all nodes.

Before disabling the node, first disable the transfer of tablets through the Active button, then click Drain, and wait for all the tablets to be moved away.

Enabling/disabling Scrubbing

Scrubbing is a process that reads data, checks its integrity, and restores it if needed. The process is run by default. The interval between completing a scrub and starting the next one is 1 month. You can change the interval using YDB DSTool. The process checks data that was accessed before the previous scrub. Scrubbing is started and stopped for the entire YDB cluster. Scrubbing is performed in the background without overloading the system.

To set a 48-hour interval, run the command:

```
ydb-dstool -e <bs_endpoint> cluster set --scrub-periodicity 48h
```

You can also set the maximum number of cluster disks to be scrubbed at a time. For example, to only scrub one disk at a time, run the command:

```
ydb-dstool -e <bs_endpoint> cluster set --max-scrubbed-disks-at-once
```

To stop cluster scrubbing, run the command:

```
ydb-dstool -e <bs_endpoint> cluster set --scrub-periodicity disable
```

Data Center Decommissioning

> **Note**
>
> Use this procedure only to decommission an entire data center.

Decommissioning is the procedure for moving a VDisk from a PDisk that needs to be decommissioned.

Data is moved to PDisk clusters where there is enough free space to create new slots. Moving is performed only when there is a possibility of moving with at least partial preservation of the failure model. It may happen that the system can't strictly follow the failure model during decommissioning. However, it does its best to ensure fault-tolerance as fully as under normal operation. For example, when encoding `mirror-3-dc`, a situation may arise when a group is located in four rather than three data centers.

Decommissioning is done asynchronously, meaning that data is not moved immediately while handling the command. If the failure model allows, SelfHeal moves slots one by one in the background to completely release the specified PDisks.

Managing Decommission

To manage the state of decommissioning, set the `DecommitStatus` parameter for the appropriate PDisk. The parameter can take the following values:

- `DECOMMIT_NONE`: The disk is not being decommissioned and is running normally, according to its condition.
- `DECOMMIT_PENDING`: Disk decommissioning is scheduled. Data is not transferred from the disk. However, slots for new groups won't be created and the slots of the previously created groups won't be moved.
- `DECOMMIT_IMMINENT`: Disk decommissioning is required. Data is transferred in the background to disks that have the `DECOMMIT_NONE` status and satisfy the failure model.

The `DECOMMIT_PENDING` and `DECOMMIT_IMMINENT` values shouldn't be removed under normal decommissioning, since the equipment is removed from the cluster by running the `DefineBox` command.

To cancel decommissioning, just change the disk status to `DECOMMIT_NONE`. In this case, BS_CONTROLLER won't take any additional actions: the previously moved VDisks remain where they are. To return them, you can use commands to move slots point by point, depending on the specific situation.

By managing the `DECOMMIT_PENDING` and `DECOMMIT_IMMINENT` states, you can perform cluster decommissioning in parts.

For example, you need to move equipment from data center-1 (DC-1) to data center-2 (DC-2):

1. The DC-2 hosts buffer equipment to transfer the first chunk of data to.
2. Switch the status of all DC-1 disks to `DECOMMIT_PENDING` at once so that no data can be moved inside the DC-1. Set this status for every disk in the data center simultaneously: applying `DECOMMIT_PENDING` to only a subset of disks in the data center may cause violation of the storage group failure model.
3. Switch the status of all DC-1 disks to `DECOMMIT_IMMINENT` on the equipment that is equivalent to the buffer one.
4. Wait until all the disks in the `DECOMMIT_IMMINENT` status are released.
5. Move the released equipment from the DC-1 to the DC-2 and switch the status of its disks to `DECOMMIT_NONE`.

Repeat the above steps for the next set of equipment in the DC-1 until all the equipment is moved.

To set the desired state of disk decommissioning, use the YDB DSTool utility. The command below sets the `DECOMMIT_IMMINENT` status for the disk with the ID `1000` on the node with the ID `1`:

```
ydb-dstool -e  <bs_endpoint> pdisk set --decommit-status DECOMMIT_IMMINENT --pdisk-ids "[1:1000]"
```

Decommissioning groups using virtual groups

Alert

The decommission operation is extremely dangerous and can lead to irreversible data loss. Before performing a decommission, **be sure** to create a backup of all cluster data.

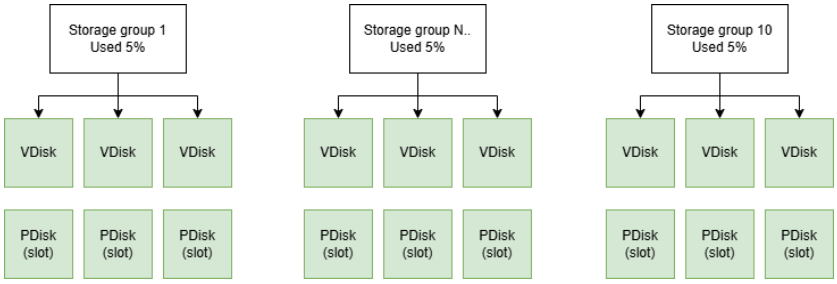
Alert

The ability to roll back a storage group decommission is under development. At the moment, the operation is irreversible. It is recommended to use decommissioning only in exceptional cases when it is necessary to free up resources once.

How it works

Physical storage groups are a limited resource in a cluster: their number is determined by the cluster size, and they cannot be deleted as long as they store data. This can lead to a situation where there are not enough groups to create a new database or expand an existing one, and space on physical disks cannot be freed because old groups contain data.

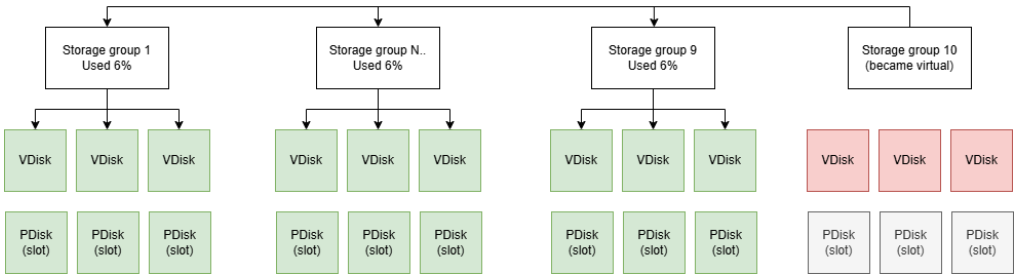
Consider an example: a database uses 10 storage groups. Each of these groups contains some amount of user data, but they are only 5% full. Throughput is also not fully utilized. As a result, all groups are formally occupied, and when trying to create a new database, a new storage group cannot be allocated — there are no more free slots on PDisks. At the same time, existing groups are used inefficiently.



In such a situation, the task arises to free up some of the storage groups (decommission them) from the current database in order to return them to the pool of available resources for subsequent use.

To solve this problem, a storage group decommission mechanism is used. As part of this mechanism, one new virtual storage group is created for the original physical group, which replaces this physical group. After performing the operation, the system moves to the following state:

- The original physical storage group is replaced by a virtual one.
- Data is copied from the source group to the remaining physical groups of the cluster (increasing their usage).
- The virtual group stores data in the remaining physical groups of the cluster.
- Resources used by the source group are freed (VDisks are deleted, PDisk slots are freed).



Decommissioning process stages

Note

Access to data is preserved during decommissioning — users can read and write data as usual. All data movements occur in the background in parallel with the main application work. It is worth noting that during decommissioning, an increased load is still created on the storage system.

In the following, by the source physical group we will mean the group over which the decommissioning process is taking place.

Initialization of a virtual storage group and blocking writes to the source physical group

What happens:

- A virtual group is created.
- Writing new data to the source physical group is blocked.
- All new writes start going to the virtual group.

Status: `DecommitStatus: PENDING` → `IN_PROGRESS`

Time: Usually fractions of a second

Metadata copying

What happens:

- Metadata of all blobs (information about what data exists, but not the data itself) is copied from the source physical group to the remaining physical storage groups

Status: `DecommitStatus: IN_PROGRESS`

Time: Usually seconds

Data copying

What happens:

- Data is read from the source physical group.
- Written to the remaining physical groups of the cluster through the virtual group.
- Data is distributed across several groups for balancing.

Status: `DecommitStatus: IN_PROGRESS`

Time: Depends on the volume of data (can be hours for large groups)

Freeing resources of the source physical group

What happens:

- All data is copied.
- The source physical group and its VDisks are deleted.
- PDisk resources are freed.
- The group is marked as decommissioned.

Status: `DecommitStatus: IN_PROGRESS` → `DONE`

Time: Minutes (VDisk deletion)

Prerequisites

The group decommission instructions work if the following conditions are met:

1. A deployed YDB cluster. Deployment instructions are available in the Initial Deployment section.
2. The `ydb-dstool` utility is installed. Installation instructions are available in the YDB DSTool installation documentation.
3. An authentication token generated for cluster access:

```
ydb -e grpc://<cluster-endpoint> -d <db_path> --user <user> auth get-token -f > ~/ydb_token
```

Replace `<cluster-endpoint>` with your cluster address, `<db_path>` with the database path, and `<user>` with the username for authorization.

> **Note**
>
> Decommissioning can be performed even on an empty cluster; the presence of data is not required. However, if there is data in the cluster, this will allow checking its consistency after decommissioning. If necessary, you can load test data using the `ydb workload tpcc` utility:
>
> ```
> ydb workload tpcc --path tpcc/10wh init -w 100
> ydb workload tpcc --path tpcc/10wh import -w 100
> ```
>
> And run the TPC-C test before and after decommissioning:
>
> ```
> ydb workload tpcc --path tpcc/10wh run -w 100
> ```
>
> More about working with test load in the ydb workload tpcc documentation.

Starting decommissioning

Selecting a group for decommissioning

Before starting decommissioning, you must select the group (group ID) you want to decommission. You can choose any group, but it is recommended to choose a less busy group — this will help complete the decommissioning process faster. This can be done through the cluster web interface or using the CLI:

UI (Web interface)

1. Open the cluster management web interface `https://<cluster-endpoint>:8765`.
2. Open the `Storage` tab.
3. Select the group you need and save the `Group ID`.

CLI

1. Run the command to get a list of storage groups

```
ydb-dstool -e <cluster-endpoint> \
  --ca-file <path-to-ca-cert> \
  --token-file ~/ydb_token \
  group list
```

2. In the command output, find the group you need and save its `Group ID`.

Command to start

To start decommissioning, run the `ydb-dstool` command with the `group decommit` parameter; it will automatically create a virtual group and start the decommissioning process:

```
ydb-dstool -e grpcs://<cluster-endpoint>:2135 \
  --ca-file <path-to-ca-cert> \
  --token-file ~/ydb_token \
  group decommit \
  --group-ids <GROUP_ID> \
  --database /Root/<database-name> \
  --wait
```

Command parameters

- `-e <endpoint>` — cluster address (e.g., `grpcs://static-node-1.ydb-cluster.com:2135`);
- `--ca-file <path>` — path to the CA certificate file for TLS connection;
- `--token-file <path>` — path to the file with the authentication token;
- `--group-ids <GROUP_ID>` — ID of the group to decommission (multiple groups can be specified separated by spaces);
- `--database=<DB>` — path to the database or domain where decommissioning is performed (e.g., `/Root/database`);
- `--wait` — wait for decommissioning to start; if a start error occurs, the error is displayed on the screen, and decommissioning is cancelled automatically (only when this option is specified).

Command example

```
ydb-dstool -e grpcs://static-node-1.ydb-cluster.com:2135 \
  --ca-file /home/ubuntu/ansible-11-nov-244/3-nodes-mirror-3-dc/files/TLS/certs/ca.crt \
  --token-file ~/ydb_token \
  group decommit \
  --group-ids 2181038081 \
  --database=/Root/database \
  --wait
```

Checking decommissioning status

After starting decommissioning, you can check its status using the command:

```
ydb-dstool -e grpcs://<cluster-endpoint>:2135 \
  --ca-file <path-to-ca-cert> \
  --token-file ~/ydb_token \
  group list --columns GroupId PoolName \
  ErasureSpecies OperatingStatus VDisks_TOTAL BlobDepotId DecommitStatus
```

In the command output for groups for which the decommissioning procedure was started, an additional `DecommitStatus` field appears, which can take one of the following values:

- `NONE` — decommissioning for the specified group is not performed;
- `PENDING` — group decommissioning is expected but not yet running (a virtual group is being created);
- `IN_PROGRESS` — group decommissioning is being performed (all writes already go to the virtual group, reads go to the virtual group and the old group);
- `DONE` — decommissioning is fully completed (the physical group has turned into a virtual one).

Command output example

```
┌─────────────┬─────────────────────┬─────────────────┬─────────────────┬─────────────────┬─────────────────────┬────
──────────────┐
│ GroupId     │ PoolName            │ ErasureSpecies │ OperatingStatus │ VDisks_TOTAL │ BlobDepotId        │ Dec
ommitStatus │
├─────────────┼─────────────────────┼─────────────────┼─────────────────┼─────────────────┼─────────────────────┼────
──────────────┤
│ 2181038080 │ /Root/database:ssd │ mirror-3-dc    │ FULL            │ 9            │ 0                  │ NON
E           │
│ 2181038081 │ /Root/database:ssd │ mirror-3-dc    │ FULL            │ 0            │ 72075186224038889 │ DON
E           │
│ 2181038082 │ /Root/database:ssd │ mirror-3-dc    │ FULL            │ 9            │ 0                  │ NON
E           │
│ 2181038083 │ /Root/database:ssd │ mirror-3-dc    │ FULL            │ 9            │ 0                  │ NON
E           │
│ 2181038084 │ /Root/database:ssd │ mirror-3-dc    │ FULL            │ 9            │ 0                  │ NON
E           │
│ 2181038085 │ /Root/database:ssd │ mirror-3-dc    │ FULL            │ 9            │ 0                  │ NON
E           │
│ 2181038086 │ /Root/database:ssd │ mirror-3-dc    │ FULL            │ 9            │ 0                  │ NON
E           │
│ 2181038087 │ /Root/database:ssd │ mirror-3-dc    │ FULL            │ 9            │ 0                  │ NON
E           │
└─────────────┴─────────────────────┴─────────────────┴─────────────────┴─────────────────┴─────────────────────┴────
──────────────┘
```

Note that in the above output example for group `2181038081`, the number of VDisks has changed (it became 0), the virtual group ID appeared in the `BlobDepotId` field, and the decommissioning status (`DecommitStatus`) changed to `DONE`. This reflects the successful completion of the decommissioning procedure for this group.

Moving VDisks

Sometimes you may need to free up a block store volume to replace equipment. Or a VDisk may be in active use, affecting the performance of other VDisks running on the same PDisk. In cases like this, VDisks need to be moved.

Move a VDisk from a block store volume

Get a list of VDisk IDs using YDB DSTool:

```
ydb-dstool -e <bs_endpoint> vdisk list --format tsv --columns VDiskId --no-header
```

To move a VDisk from a block store volume, run the following commands on the cluster node:

```
ydb-dstool -e <bs_endpoint> vdisk evict --vdisk-ids VDISK_ID1 ... VDISK_IDN
ydbd admin bs config invoke --proto 'Command { ReassignGroupDisk { GroupId: <Storage group ID> GroupGeneratio
n: <Storage group generation> FailRealmIdx: <FailRealm> FailDomainIdx: <FailDomain> VDiskIdx: <Slot number> }
}'
```

- `VDISK_ID1 ... VDISK_IDN`: The list of VDisk IDs like `[GroupId:GroupGeneration:FailRealmIdx:FailDomainIdx:VDiskIdx]`. The IDs are separated by a space.
- `GroupId`: The ID of the storage group.
- `GroupGeneration`: Storage group generation.
- `FailRealmIdx`: Fail realm number.
- `FailDomainIdx`: Fail domain number.
- `VDiskIdx`: Slot number.

Move VDisks from a broken/missing block store volume

If SelfHeal is disabled or fails to move VDisks automatically, you'll have to run this operation manually:

1. Go to monitoring and make sure that the VDisk has actually failed.
2. Get the appropriate `[NodeId:PDiskId]` using YDB DSTool:

   ```
   ydb-dstool -e <bs_endpoint> vdisk list | fgrep VDISK_ID
   ```

3. Move the VDisk:

   ```
   ydb-dstool -e <bs_endpoint> pdisk set --status BROKEN --pdisk-ids "[NodeId:PDiskId]"
   ```

Enable the VDisk back after reassignment

To enable the VDisk back after reassignment:

1. Go to monitoring and make sure that the VDisk is actually operable.
2. Get the appropriate `[NodeId:PDiskId]` using YDB DSTool:

   ```
   ydb-dstool -e <bs_endpoint> pdisk list
   ```

3. Enable the PDisk back:

   ```
   ydb-dstool -e <bs_endpoint> pdisk set --status ACTIVE --pdisk-ids "[NodeId:PDiskId]"
   ```

Staying within the failure model

One VDisk in the storage group failed

With SelfHeal enabled, this situation is considered normal. SelfHeal will move the VDisk over the time specified in the settings, then data replication will start on a different PDisk.

If SelfHeal is disabled, you'll have to manually move the VDisk. Before moving it, make sure that **only one** VDisk in the storage group has failed.
Then follow the instructions.

More than one VDisk in the same storage group have failed without going beyond the failure model

In this kind of failure, no data is lost, the system maintains operability, and read and write queries are executed successfully. Performance might degrade because of the load handover from the failed disks to the operable ones.

If multiple VDisks have failed in the group, SelfHeal stops moving VDisks. If the maximum number of failed VDisks for the failure model has been reached, recover at least one VDisk before you start moving the VDisks. You may also need to be more careful when moving VDisks one by one.

The number of failed VDisks has exceeded the failure model

The availability and operability of the system might be lost. Make sure to revive at least one VDisk without losing the data stored on it.

Disk load balancing

YDB supports two methods for disk load balancing:

- Distribute the load evenly across groups.
- Distribute VDisks evenly across block store volumes.

Distribute the load evenly across groups

At the bottom of the Hive web-viewer page, there is a button named "Reassign Groups".

Distribute VDisks evenly across block store volumes

As a result of some operations, such as decommissioning, VDisks can be distributed across block store volumes unevenly. You can distribute them more evenly in one of the following ways:

- Move VDisks one by one from overloaded block store volumes.
- Use YDB DSTool. The command below moves a VDisk from an overloaded block store volume to a less loaded one:

  ```
  ydb-dstool -e <bs_endpoint> cluster balance
  ```

  The command moves a single VDisk per run.

Changing the number of slots for VDisks on PDisks

To add storage groups, redefine the host config by increasing the number of slots on PDisks.

Before that, you need to get the config to be changed. You can do this with the following command:

```
Command {
  TReadHostConfig{
    HostConfigId: <host-config-id>
  }
}
```

```
ydbd -s <endpoint> admin bs config invoke --proto-file ReadHostConfig.txt
```

Insert the obtained config into the protobuf below and edit the `PDiskConfig/ExpectedSlotCount` field value in it.

```
Command {
  TDefineHostConfig {
    <host config>
  }
}
```

```
ydbd -s <endpoint> admin bs config invoke --proto-file DefineHostConfig.txt
```

Freeing up space on physical devices

When the disk space is used up, the database may start responding to all queries with an error. To keep the database healthy, we recommend deleting a part of the data or adding block store volumes to extend the cluster.

Below are instructions that can help you add or free up disk space.

Defragment a VDisk

When working with the DB, internal VDisk fragmentation is done. You can find out the percentage of fragmentation on the VDisk monitoring page. We do not recommend that you perform defragmentation of VDisks that are fragmented by 20% or less.

According to the failure model, the cluster survives the loss of two VDisks in the same group without data loss. If all VDisks in the group are up and there are no VDisks with the error or replication status, then deleting data from one VDisk will result in the VDisk recovering it in a compact format. Please keep in mind that data storage redundancy will be decreased until automatic data replication is complete.

During data replication, the load on all the group's VDisks increases, and response times may deteriorate.

1. View the fragmentation coefficient on the VDisk page in the viewer.

   If its value is more than 20%, defragmentation can help free up VDisk space.
2. Check the status of the group that hosts the VDisk. There should be no VDisks that are unavailable or in the error or replication status in the group.

   You can view the status of the group in the viewer.
3. Run the wipe command for the VDisk.

   All data stored on a VDisk will be permanently deleted, whereupon the VDisk will begin restoring the data by reading them from the other VDisks in the group.

   ```
   ydbd admin blobstorage group reconfigure wipe --domain <Domain number> --node <Node ID> --pdisk <pdisk-id> --vslot <Slot number>
   ```

   You can view the details for the command in the viewer.

If the block store volume is running out of space, you can apply defragmentation to the entire block store volume.

1. Check the health of the groups in the cluster. There shouldn't be any problem groups on the same node with the problem block store volume.
2. Log in via SSH to the node hosting this block store volume
3. Check if you can restart the process.
4. Stop the process

   ```
   sudo systemctl stop ydbd
   ```
5. Format the block store volume

   ```
   sudo ydbd admin blobstorage disk obliterate <path to the store volume part label>
   ```
6. Run the process

   ```
   sudo systemctl start ydbd
   ```

Moving individual VDisks from full block store volumes

If defragmentation doesn't help free up space on the block store volume, you can move individual VDisks.

Replacing a node's FQDN

Sometimes, a node's FQDN changes, but the node itself remains in the system under a different name. Simply changing the node name in the hosts section will not work because `BS_CONTROLLER` internally stores the resource bindings to the `FQDN:IcPort` pairs, where IcPort is the Interconnect port number on which the node operates.

Replacement procedure

1. Determine the NodeId of the node to be replaced.
2. Prepare the DefineBox command that describes the cluster resources, in which an element `EnforcedNodeId: <NodeId>` will be added for the resources of the node to be replaced.
3. Execute this command.
4. Replace the FQDN in the hosts list in `cluster.yaml`.
5. Perform a rolling restart.
6. Remove the EnforcedNodeId field from DefineBox and replace the Fqdn with the new node name.
7. Execute DefineBox with the new values.

Example

Suppose a cluster consisting of three nodes:

`config.yaml`:

```
- host: host1.my.sub.net
  node_id: 1
  location: {unit: 12345, data_center: MYDC, rack: r1}
- host: host2.my.sub.net
  node_id: 2
  location: {unit: 23456, data_center: MYDC, rack: r2}
- host: host3.my.sub.net
  node_id: 3
  location: {unit: 34567, data_center: MYDC, rack: r3}
```

DefineBox looks like this:

```
DefineBox {
    BoxId: 1
    Host { Key { Fqdn: "host1.my.sub.net" IcPort: 19001 } HostConfigId: 1 }
    Host { Key { Fqdn: "host2.my.sub.net" IcPort: 19001 } HostConfigId: 1 }
    Host { Key { Fqdn: "host3.my.sub.net" IcPort: 19001 } HostConfigId: 1 }
}
```

Suppose we want to rename host1.my.sub.net to host4.my.sub.net. First, we create a DefineBox as follows:

```
DefineBox {
    BoxId: 1
    Host { Key { Fqdn: "host1.my.sub.net" IcPort: 19001 } HostConfigId: 1 EnforcedNodeId: 1 }
    Host { Key { Fqdn: "host2.my.sub.net" IcPort: 19001 } HostConfigId: 1 }
    Host { Key { Fqdn: "host3.my.sub.net" IcPort: 19001 } HostConfigId: 1 }
}
```

Then modify `config.yaml`:

```
- host: host4.my.sub.net
  node_id: 1
  location: {unit: 12345, data_center: MYDC, rack: r1}
- host: host2.my.sub.net
  node_id: 2
  location: {unit: 23456, data_center: MYDC, rack: r2}
- host: host3.my.sub.net
  node_id: 3
  location: {unit: 34567, data_center: MYDC, rack: r3}
```

Next, perform a rolling restart of the cluster.

Finally, perform the second adjusted DefineBox:

```
DefineBox {
    BoxId: 1
    Host { Key { Fqdn: "host4.my.sub.net" IcPort: 19001 } HostConfigId: 1 }
    Host { Key { Fqdn: "host2.my.sub.net" IcPort: 19001 } HostConfigId: 1 }
    Host { Key { Fqdn: "host3.my.sub.net" IcPort: 19001 } HostConfigId: 1 }
}
```

Blob Depot

Blob depot extends the functionality of the storage subsystem by adding virtual group functionality.

A virtual group, like a physical group, is a fault tolerance unit of the storage subsystem in the cluster. However, a virtual group stores its data in other groups (unlike a physical group, which stores data on VDisks).

This method of data storage allows for more flexible use of the YDB storage subsystem, in particular:

- Using more "heavy" tablets in conditions where the size of one physical group is limited.
- Providing tablets with a wider write bandwidth by balancing writes across all groups on which the blob depot is running.
- Ensuring transparent data migration between different groups for client tablets.

Blob depot can also be used for group decommissioning, that is, to remove a VDisk within a physical group while preserving all data that was written to this group. In this usage scenario, data from the physical group is transparently transferred to the virtual group for the client, then all VDisks of the physical group are removed to free up the resources they occupy.

Virtual Group Mode

In virtual group mode, blob depot allows combining several groups into a single space for storing large amounts of data. Load balancing by occupied space is provided, as well as increased throughput by distributing writes across different groups. Background (completely transparent to the client) data transfer is also possible.

Virtual groups are also created within Storage Pools, like physical groups, but for virtual groups it is recommended to create a separate pool with the command `dstool pool create virtual`. These pools can be specified in Hive for creating other tablets. However, to avoid latency degradation, channels 0 and 1 of tablets are recommended to be placed on physical groups, and only data channels should be placed on virtual groups with blob depot.

How to Launch

A virtual group is created through BS_CONTROLLER by passing a special command. The virtual group creation command is idempotent, so to avoid creating extra blob depots, each virtual group is assigned a name. The name must be unique within the entire cluster. In case of repeated command execution, an error will be returned with the `Already: true` field filled and indicating the number of the previously created virtual group.

```
dstool -e ... --direct group virtual create --name vg1 vg2 --hive-id=72057594037968897 --storage-pool-name=/R
oot:virtual --log-channel-sp=/Root:ssd --data-channel-sp=/Root:ssd*8
```

Command line parameters:

- `--name` — unique name for the virtual group (or several virtual groups with similar parameters)
- `--hive-id=N` — number of the Hive tablet that will manage this blob depot; you must specify the Hive of the tenant within which the blob depot is launched
- `--storage-pool-name=POOL_NAME` — name of the Storage Pool within which the blob depot needs to be created
- `--storage-pool-id=BOX:POOL` — alternative to `--storage-pool-name`, where you can specify an explicit numeric pool identifier
- `--log-channel-sp=POOL_NAME` — name of the pool where channel 0 of the blob depot tablet will be placed
- `--snapshot-channel-sp=POOL_NAME` — name of the pool where channel 0 of the blob depot tablet will be placed; if not specified, the value from --log-channel-sp is used
- `--data-channel-sp=POOL_NAME[*COUNT]` — name of the pool where data channels are placed; if the COUNT parameter is specified (after the asterisk), COUNT data channels are created in the specified pool; it is recommended to create a large number of data channels for blob depot in virtual group mode (64..250) to most efficiently use storage
- `--wait wait` — for blob depot creation to complete; if this option is not specified, the command terminates immediately after responding to the blob depot creation request, without waiting for the creation and launch of the tablets themselves

How to Check that the Virtual Group is Running

You can view the result of virtual group creation in the following ways:

- via the BS_CONTROLLER monitoring page
- via the `dstool group list --virtual-groups-only` command

In both cases, creation should be controlled through the `VirtualGroupName` field, which should match what was passed in the `--name` parameter. If the `dstool group virtual create` command completed successfully, the virtual group unconditionally appears in the group list, but the `VirtualGroupState` field can take one of the following values:

- `NEW` — group is waiting for initialization (tablet creation through Hive, its configuration and launch is in progress)
- `WORKING` — group is created and working, ready to execute user requests
- `CREATE_FAILED` — an error occurred during group creation, the text description of which can be seen in the ErrorReason field

```
$ dstool --cluster=$CLUSTER --direct group list --virtual-groups-only
┌──────────────┬──────────────┬──────────────┬──────────────┬──────────────┬──────────────┬──────────────┬──
────────────────┬──────────────┬──────────────────┬──────────────┬──────────────┐
| GroupId    | BoxId:PoolId | PoolName      | Generation | ErasureSpecies | OperatingStatus | VDisks_TOTAL |
VirtualGroupState | VirtualGroupName | BlobDepotId       | ErrorReason | DecommitStatus |
├──────────────┼──────────────┼──────────────┼──────────────┼──────────────┼──────────────┼──────────────┼──
────────────────┼──────────────┼──────────────────┼──────────────┼──────────────┤
| 4261412864 | [1:2]        | /Root:virtual | 0          | none           | DISINTEGRATED   | 0            |
WORKING           | vg1              | 72075186224037888 |             | NONE           |
| 4261412865 | [1:2]        | /Root:virtual | 0          | none           | DISINTEGRATED   | 0            |
WORKING           | vg2              | 72075186224037890 |             | NONE           |
| 4261412866 | [1:2]        | /Root:virtual | 0          | none           | DISINTEGRATED   | 0            |
WORKING           | vg3              | 72075186224037889 |             | NONE           |
| 4261412867 | [1:2]        | /Root:virtual | 0          | none           | DISINTEGRATED   | 0            |
WORKING           | vg4              | 72075186224037891 |             | NONE           |
└──────────────┴──────────────┴──────────────┴──────────────┴──────────────┴──────────────┴──────────────┴──
────────────────┴──────────────┴──────────────────┴──────────────┴──────────────┘
```

Architecture

Blob depot is a tablet that, in addition to two system channels (0 and 1), also contains a set of additional channels where the actual data written to the storage. Client data is written to these additional channels.

Blob depot as a tablet can be launched on any cluster node.

When blob depot operates in virtual group mode, agents (BlobDepotAgent) are used to access it. These are actors that perform functions similar to DS proxy — they are launched on each node that uses a virtual group with blob depot. These same actors convert storage requests into commands for blob depot and provide data exchange with it.

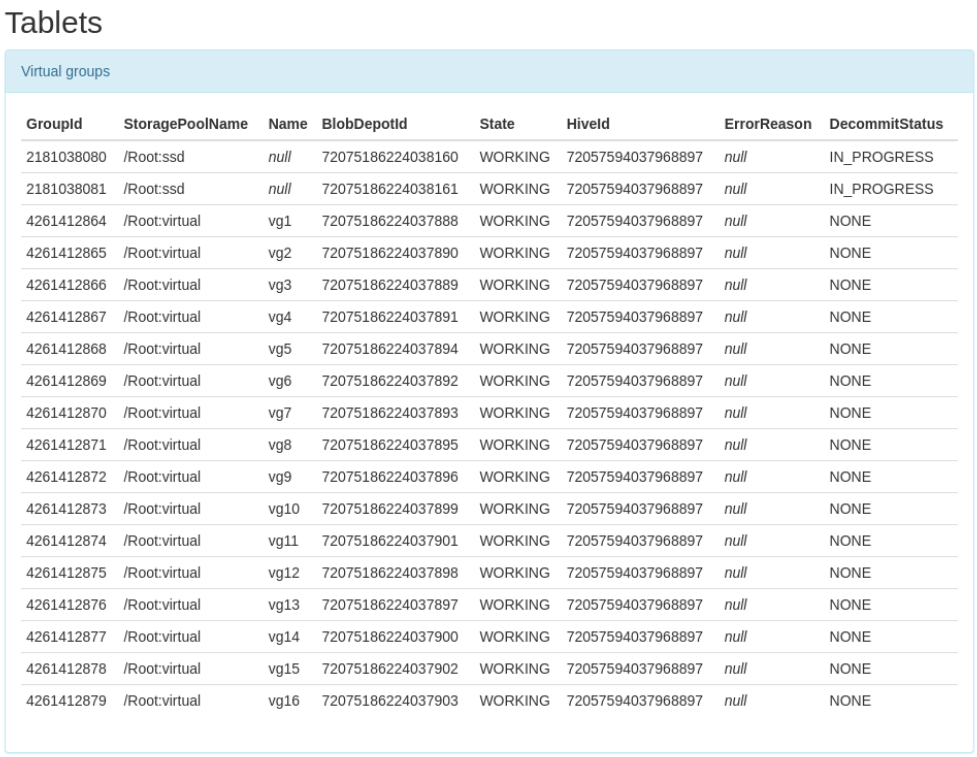
Diagnostic Mechanisms

The following mechanisms are provided for diagnosing blob depot operability:

- BS_CONTROLLER monitoring page
- blob depot monitoring page
- internal viewer
- event log
- charts

BS_CONTROLLER Monitoring Page

On the BS_CONTROLLER monitoring page there is a special Virtual groups tab, which shows all groups that use blob depot:



Tablets

Virtual groups

GroupId	StoragePoolName	Name	BlobDepotId	State	HiveId	ErrorReason	DecommitStatus
2181038080	/Root:ssd	*null*	72075186224038160	WORKING	72057594037968897	*null*	IN_PROGRESS
2181038081	/Root:ssd	*null*	72075186224038161	WORKING	72057594037968897	*null*	IN_PROGRESS
4261412864	/Root:virtual	vg1	72075186224037888	WORKING	72057594037968897	*null*	NONE
4261412865	/Root:virtual	vg2	72075186224037890	WORKING	72057594037968897	*null*	NONE
4261412866	/Root:virtual	vg3	72075186224037889	WORKING	72057594037968897	*null*	NONE
4261412867	/Root:virtual	vg4	72075186224037891	WORKING	72057594037968897	*null*	NONE
4261412868	/Root:virtual	vg5	72075186224037894	WORKING	72057594037968897	*null*	NONE
4261412869	/Root:virtual	vg6	72075186224037892	WORKING	72057594037968897	*null*	NONE
4261412870	/Root:virtual	vg7	72075186224037893	WORKING	72057594037968897	*null*	NONE
4261412871	/Root:virtual	vg8	72075186224037895	WORKING	72057594037968897	*null*	NONE
4261412872	/Root:virtual	vg9	72075186224037896	WORKING	72057594037968897	*null*	NONE
4261412873	/Root:virtual	vg10	72075186224037899	WORKING	72057594037968897	*null*	NONE
4261412874	/Root:virtual	vg11	72075186224037901	WORKING	72057594037968897	*null*	NONE
4261412875	/Root:virtual	vg12	72075186224037898	WORKING	72057594037968897	*null*	NONE
4261412876	/Root:virtual	vg13	72075186224037897	WORKING	72057594037968897	*null*	NONE
4261412877	/Root:virtual	vg14	72075186224037900	WORKING	72057594037968897	*null*	NONE
4261412878	/Root:virtual	vg15	72075186224037902	WORKING	72057594037968897	*null*	NONE
4261412879	/Root:virtual	vg16	72075186224037903	WORKING	72057594037968897	*null*	NONE

The table provides the following columns:

Field	Description
GroupId	Group number.
StoragePoolName	Name of the pool where the group is located.
Name	Virtual group name; it is unique for the entire cluster. For decommissioned groups this will be null.
BlobDepotId	Number of the blob depot tablet responsible for serving this group.
State	Blob depot state; can be NEW, WORKING, CREATED_FAILED.
HiveId	Number of the Hive tablet within which the specified blob depot was created.
ErrorReason	For CREATE_FAILED state contains a text description of the creation error reason.
DecommitStatus	Group decommission state; can be NONE, PENDING, IN_PROGRESS, DONE.

Blob Depot Monitoring Page

The blob depot monitoring page presents the main tablet operation parameters, which are grouped by tabs available via the "Contained data" link:

- data
- refcount
- trash
- barriers
- blocks
- storage

In addition, the main page provides brief information about the blob depot state:

Contained data

Stats

Parameter	Value
Data, bytes	21.83 GiB
Trash in flight, bytes	0 B
Trash pending, bytes	0 B
Data in GroupId# 2181038088, bytes	826.47 MiB
Data in GroupId# 2181038089, bytes	816.17 MiB
Data in GroupId# 2181038090, bytes	736.13 MiB
Data in GroupId# 2181038092, bytes	797.28 MiB
Data in GroupId# 2181038093, bytes	622.74 MiB
Data in GroupId# 2181038094, bytes	734.44 MiB
Data in GroupId# 2181038095, bytes	727.78 MiB
Data in GroupId# 2181038096, bytes	843.88 MiB
Data in GroupId# 2181038098, bytes	724.19 MiB
Data in GroupId# 2181038099, bytes	757.32 MiB
Data in GroupId# 2181038100, bytes	725.58 MiB
Data in GroupId# 2181038101, bytes	673.08 MiB
Data in GroupId# 2181038102, bytes	715.34 MiB
Data in GroupId# 2181038104, bytes	748.80 MiB
Data in GroupId# 2181038105, bytes	710.01 MiB
Data in GroupId# 2181038106, bytes	770.66 MiB
Data in GroupId# 2181038107, bytes	705.25 MiB
Data in GroupId# 2181038108, bytes	898.86 MiB
Data in GroupId# 2181038110, bytes	725.41 MiB
Data in GroupId# 2181038111, bytes	788.05 MiB
Data in GroupId# 2181038112, bytes	639.93 MiB
Data in GroupId# 2181038113, bytes	720.35 MiB
Data in GroupId# 2181038114, bytes	657.61 MiB

This table provides the following data:

- Data, bytes — amount of saved data bytes (TotalStoredDataSize).
- Trash in flight, bytes — amount of bytes of unnecessary data waiting for transactions to complete to become garbage (InFlightTrashSize).
- Trash pending, bytes — amount of garbage bytes that have not yet been passed to garbage collection (TotalStoredTrashSize).
- Data in GroupId# XXX, bytes — amount of data bytes in group XXX (both useful data and not yet collected garbage).

Data

Main

Parameter	Value
Loaded	true
Last assimilated blob id	
Data size, number of keys	284398
RefCount size, number of blobs	284398
Total stored data size, bytes	21.83 GiB
Keys made certain, number of keys	0

Uncertainty resolver

Parameter	Value
Keys queried	0
Gets issued	0
Keys resolved	0
Keys unresolved	0
Keys dropped	498590
Keys being processed	0
Blobs in flight	0

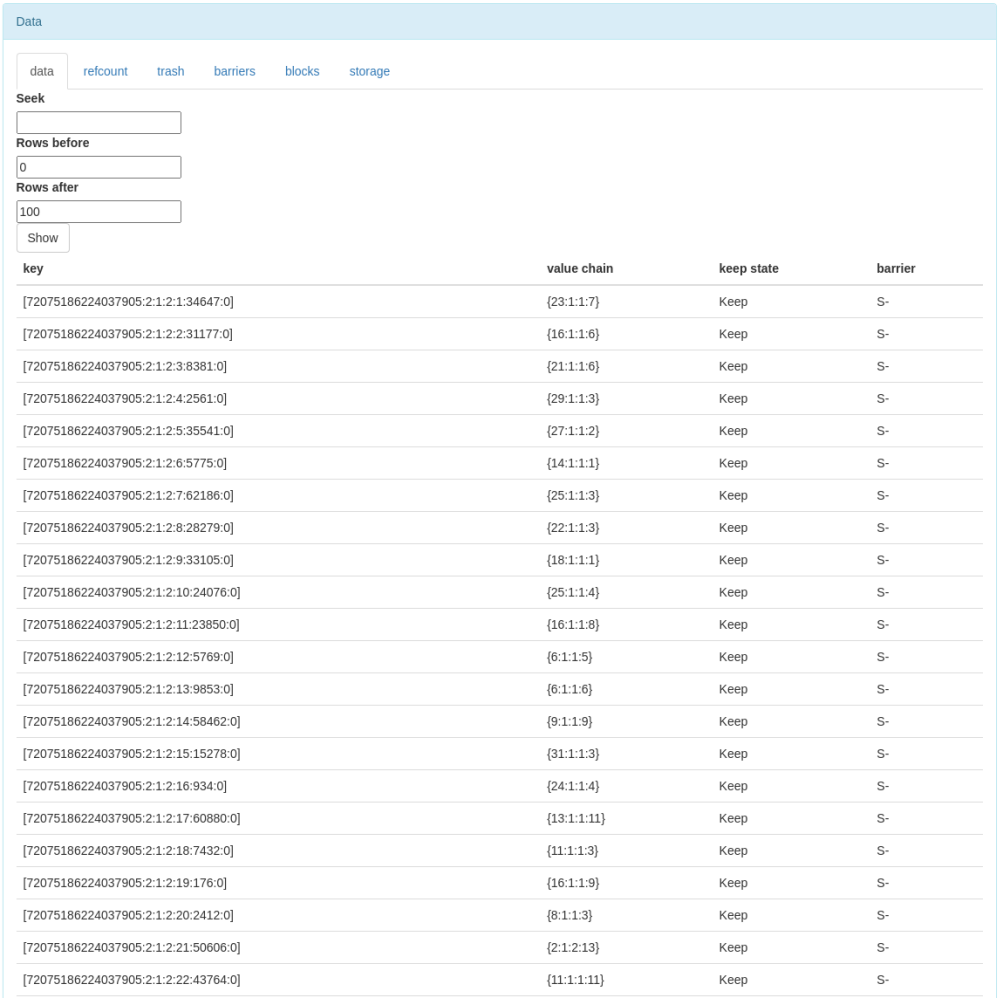
Parameter purposes are as follows:

- Loaded — boolean value showing whether all metadata from the tablet's local database is loaded into memory.
- Last assimilated blob id — BlobId of the last read blob (metadata copying during decommission).
- Data size, number of keys — number of saved data keys.

- RefCount size, number of blobs — number of unique data blobs that the blob depot stores in its namespace.
- Total stored data size, bytes — similar to "Data, bytes" from the table above.
- Keys made certain, number of keys — number of unwritten keys that were then confirmed by reading.

The "Uncertainty resolver" section relates to the component that works with data written but not confirmed to the blob depot.

data



key	value chain	keep state	barrier
[72075186224037905:2:1:2:1:34647:0]	{23:1:1:7}	Keep	S-
[72075186224037905:2:1:2:2:31177:0]	{16:1:1:6}	Keep	S-
[72075186224037905:2:1:2:3:8381:0]	{21:1:1:6}	Keep	S-
[72075186224037905:2:1:2:4:2561:0]	{29:1:1:3}	Keep	S-
[72075186224037905:2:1:2:5:35541:0]	{27:1:1:2}	Keep	S-
[72075186224037905:2:1:2:6:5775:0]	{14:1:1:1}	Keep	S-
[72075186224037905:2:1:2:7:62186:0]	{25:1:1:3}	Keep	S-
[72075186224037905:2:1:2:8:28279:0]	{22:1:1:3}	Keep	S-
[72075186224037905:2:1:2:9:33105:0]	{18:1:1:1}	Keep	S-
[72075186224037905:2:1:2:10:24076:0]	{25:1:1:4}	Keep	S-
[72075186224037905:2:1:2:11:23850:0]	{16:1:1:8}	Keep	S-
[72075186224037905:2:1:2:12:5769:0]	{6:1:1:5}	Keep	S-
[72075186224037905:2:1:2:13:9853:0]	{6:1:1:6}	Keep	S-
[72075186224037905:2:1:2:14:58462:0]	{9:1:1:9}	Keep	S-
[72075186224037905:2:1:2:15:15278:0]	{31:1:1:3}	Keep	S-
[72075186224037905:2:1:2:16:934:0]	{24:1:1:4}	Keep	S-
[72075186224037905:2:1:2:17:60880:0]	{13:1:1:11}	Keep	S-
[72075186224037905:2:1:2:18:7432:0]	{11:1:1:3}	Keep	S-
[72075186224037905:2:1:2:19:176:0]	{16:1:1:9}	Keep	S-
[72075186224037905:2:1:2:20:2412:0]	{8:1:1:3}	Keep	S-
[72075186224037905:2:1:2:21:50606:0]	{2:1:2:13}	Keep	S-
[72075186224037905:2:1:2:22:43764:0]	{11:1:1:11}	Keep	S-

The data table contains the following columns:

- key — key identifier (BlobId in the client namespace)
- value chain — key value formed by concatenating blob fragments from the blob depot namespace (this field lists these blobs)
- keep state — keep flag values for this blob from the client's perspective (Default, Keep, DoNotKeep)
- barrier — field showing which barrier this blob falls under (S — under soft barrier, H — under hard barrier; actually H never happens because blobs are synchronously deleted from the table when a hard barrier is set)

Given the potentially large table size, only part of it is shown on the monitoring page. To search for a specific blob, you can fill in the "seek" field by entering the BlobId of the desired blob, then specify the number of rows of interest before and after this blob and click the "Show" button.

refcount

Data

data | refcount | trash | barriers | blocks | storage

blob id	refcount
[72075186224037894:1:116:7:33:30016:0]	1
[72075186224037894:1:110:26:1057:28880:0]	1
[72075186224037894:1:361:2:705:39373:0]	1
[72075186224037894:1:148:30:2097:5588:0]	1
[72075186224037894:1:503:20:81:54961:0]	1
[72075186224037894:1:442:16:817:48656:0]	1
[72075186224037894:1:162:8:705:17513:0]	1
[72075186224037894:1:284:7:577:63292:0]	1
[72075186224037894:1:462:13:337:4355:0]	1
[72075186224037894:1:401:9:1073:6248:0]	1
[72075186224037894:1:120:26:17:54953:0]	1
[72075186224037894:1:444:19:145:13229:0]	1
[72075186224037894:1:488:4:161:55863:0]	1
[72075186224037894:1:584:12:209:47719:0]	1
[72075186224037894:1:260:19:81:51993:0]	1
[72075186224037894:1:228:6:897:58958:0]	1
[72075186224037894:1:367:24:17:28839:0]	1
[72075186224037894:1:332:11:577:52762:0]	1
[72075186224037894:1:492:23:145:58541:0]	1
[72075186224037894:1:168:30:17:56918:0]	1
[72075186224037894:1:527:27:929:50666:0]	1
[72075186224037894:1:477:7:225:60133:0]	1
[72075186224037894:1:573:15:273:64158:0]	1
[72075186224037894:1:479:23:881:34206:0]	1
[72075186224037894:1:100:7:465:17313:0]	1
[72075186224037894:1:599:6:97:75:0]	1
[72075186224037894:1:400:12:97:29934:0]	1

The refcount table contains two columns: "blob id" and "refcount". Blob id is the identifier of the stored blob written on behalf of the blob depot to storage. Refcount is the number of references to this blob from the data table (from the value chain column).

The TotalStoredDataSize metric is formed from the sum of sizes of all blobs in this table, each of which is counted exactly once, without considering the refcount field.

trash

Data

data | refcount | trash | barriers | blocks | storage

group id	blob id	in flight
2181038119	[72075186224037894:1:602:28:81:75:0]	
2181038119	[72075186224037894:1:602:28:129:75:0]	
2181038119	[72075186224037894:1:602:28:385:71:0]	
2181038119	[72075186224037894:1:602:28:401:75:0]	
2181038119	[72075186224037894:1:602:28:417:75:0]	
2181038119	[72075186224037894:1:602:28:865:75:0]	
2181038119	[72075186224037894:1:602:28:881:75:0]	
2181038106	[72075186224037894:1:597:17:81:75:0]	
2181038106	[72075186224037894:1:597:17:97:75:0]	
2181038106	[72075186224037894:1:597:17:433:75:0]	
2181038100	[72075186224037894:1:603:12:49:75:0]	
2181038099	[72075186224037894:1:607:11:113:75:0]	
2181038099	[72075186224037894:1:607:11:401:75:0]	
2181038099	[72075186224037894:1:607:11:449:75:0]	
2181038118	[72075186224037894:1:617:27:625:75:0]	
2181038120	[72075186224037894:1:607:29:113:75:0]	
2181038123	[72075186224037894:1:583:31:193:75:0]	
2181038123	[72075186224037894:1:583:31:449:77:0]	
2181038093	[72075186224037894:1:615:6:1025:71:0]	
2181038108	[72075186224037894:1:609:19:209:71:0]	
2181038108	[72075186224037894:1:609:19:945:75:0]	
2181038092	[72075186224037894:1:612:5:113:75:0]	
2181038092	[72075186224037894:1:612:5:337:75:0]	
2181038092	[72075186224037894:1:612:5:353:75:0]	
2181038092	[72075186224037894:1:612:5:369:75:0]	
2181038092	[72075186224037894:1:612:5:385:75:0]	
2181038116	[72075186224037894:1:618:25:225:75:0]	

The table contains three columns: "group id", "blob id" and "in flight". Group id is the number of the group where the no longer needed blob is stored. Blob id is the identifier of the blob itself. In flight is a sign that the blob is still going through a transaction, only after which it can be passed to the garbage collector.

The TotalStoredTrashSize and InFlightTrashSize metrics are formed from this table by summing blob sizes without the in flight flag and with it, respectively.

barriers

Data

data | refcount | trash | barriers | blocks | storage

tablet id	channel	soft	hard
72075186224038155	0	2:157 => 2:39350	0:0 => 0:0
72075186224037909	2	2:1 => 1:4294967295	2:0 => 1:4294967295
72075186224038059	2	2:1 => 1:4294967295	2:0 => 1:4294967295
72075186224038050	1	2:160 => 2:40178	0:0 => 0:0
72075186224038021	2	2:1 => 1:4294967295	2:0 => 1:4294967295
72075186224038005	2	2:1 => 1:4294967295	2:0 => 1:4294967295
72075186224038144	1	2:157 => 2:39352	0:0 => 0:0
72075186224038149	2	2:1 => 1:4294967295	2:0 => 1:4294967295
72075186224038091	0	2:157 => 2:39371	0:0 => 0:0
72075186224038000	1	2:157 => 2:39371	0:0 => 0:0
72075186224038085	2	2:1 => 1:4294967295	2:0 => 1:4294967295
72075186224038011	0	3:122 => 3:30448	0:0 => 0:0
72075186224038128	1	2:157 => 2:39372	0:0 => 0:0
72075186224038027	0	2:151 => 2:37872	0:0 => 0:0
72075186224037931	0	2:154 => 2:38678	0:0 => 0:0
72075186224038139	0	2:157 => 2:39374	0:0 => 0:0
72075186224038133	2	2:1 => 1:4294967295	2:0 => 1:4294967295
72075186224038016	1	2:158 => 2:39689	0:0 => 0:0
72075186224037925	2	2:1 => 1:4294967295	2:0 => 1:4294967295
72075186224037915	0	2:155 => 2:38870	0:0 => 0:0
72075186224038080	1	2:155 => 2:38889	0:0 => 0:0
72075186224037905	2	4:1 => 3:4294967295	4:0 => 2:0
72075186224038072	0	2:156 => 2:39187	0:0 => 0:0
72075186224037920	1	2:158 => 2:39692	0:0 => 0:0

The barriers table contains information about client barriers that were passed to the blob depot. It consists of columns "tablet id" (tablet number), "channel" (channel number for which the barrier is written), as well as barrier values: "soft" and "hard". The value has the format `gen:counter => collect_gen:collect_step`, where gen is the tablet generation number in which this barrier was set, counter is the sequential number of the garbage collection command, `collect_gen:collect_step` is the barrier value (all blobs whose generation and step within the generation are less than or equal to the specified barrier are deleted).

blocks

Data

data refcount trash barriers blocks storage

tablet id	blocked generation
72075186224038011	2
72075186224038128	1
72075186224037925	1
72075186224038072	1
72075186224038016	1
72075186224038133	1
72075186224037931	1
72075186224038050	1
72075186224038021	1
72075186224037905	3
72075186224038080	1
72075186224038139	1
72075186224037909	1
72075186224038085	1
72075186224038027	1
72075186224038144	1
72075186224038000	1
72075186224038059	1
72075186224037915	1
72075186224038149	1
72075186224038091	1
72075186224038005	1
72075186224037920	1
72075186224038155	1

The blocks table contains a list of client tablet locks and consists of columns "tablet id" (tablet number) and "blocked generation" (generation number of this tablet in which nothing can be written anymore).

storage

Data

data refcount trash barriers blocks storage

group id	bytes stored in current generation	bytes stored total	status flag	free space share
2181038088	972.28 MiB	972.28 MiB	{ Valid }	95.0 %
2181038089	966.29 MiB	966.29 MiB	{ Valid }	92.3 %
2181038090	858.26 MiB	858.26 MiB	{ Valid }	94.8 %
2181038092	951.95 MiB	951.95 MiB	{ Valid }	94.7 %
2181038093	793.19 MiB	793.19 MiB	{ Valid }	92.2 %
2181038094	874.82 MiB	874.82 MiB	{ Valid }	95.0 %
2181038095	847.88 MiB	847.88 MiB	{ Valid }	94.5 %
2181038096	986.85 MiB	986.85 MiB	{ Valid }	92.5 %
2181038098	882.24 MiB	882.24 MiB	{ Valid }	94.7 %
2181038099	919.22 MiB	919.22 MiB	{ Valid }	94.6 %
2181038100	915.49 MiB	915.49 MiB	{ Valid }	92.2 %
2181038101	807.56 MiB	807.56 MiB	{ Valid }	94.5 %
2181038102	877.60 MiB	877.60 MiB	{ Valid }	94.8 %
2181038104	869.11 MiB	869.11 MiB	{ Valid }	92.4 %
2181038105	884.52 MiB	884.52 MiB	{ Valid }	94.6 %
2181038106	898.87 MiB	898.87 MiB	{ Valid }	94.6 %
2181038107	845.60 MiB	845.60 MiB	{ Valid }	92.5 %
2181038108	1.00 GiB	1.00 GiB	{ Valid }	94.8 %
2181038110	871.55 MiB	871.55 MiB	{ Valid }	94.7 %
2181038111	919.71 MiB	919.71 MiB	{ Valid }	92.4 %
2181038112	773.51 MiB	773.51 MiB	{ Valid }	95.0 %
2181038113	833.96 MiB	833.96 MiB	{ Valid }	94.5 %
2181038114	780.37 MiB	780.37 MiB	{ Valid }	92.6 %
2181038116	986.28 MiB	986.28 MiB	{ Valid }	94.7 %
2181038117	852.59 MiB	852.59 MiB	{ Valid }	94.6 %

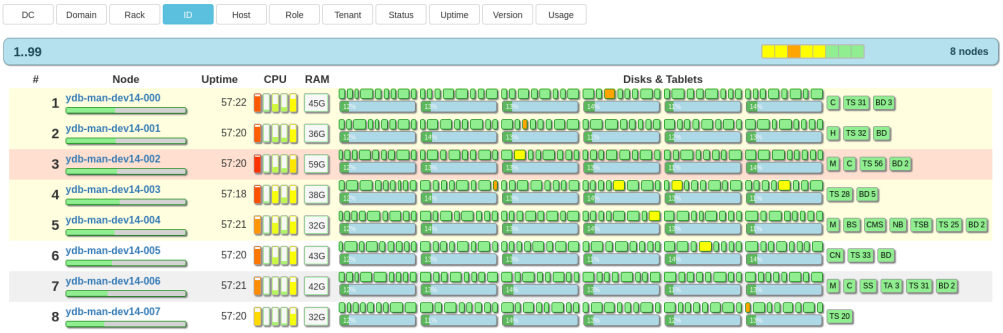
The storage table shows statistics on stored data for each group where the blob depot stores data. This table contains the following columns:

- group id — number of the group where data is stored
- bytes stored in current generation — volume of data written to this group in the current tablet generation (only useful data is counted, excluding garbage)
- bytes stored total — volume of all data saved by this blob depot to the specified group
- status flag — color status flags of the group
- free space share — group occupancy indicator (value 0 corresponds to a group completely filled by space, 1 — completely free)

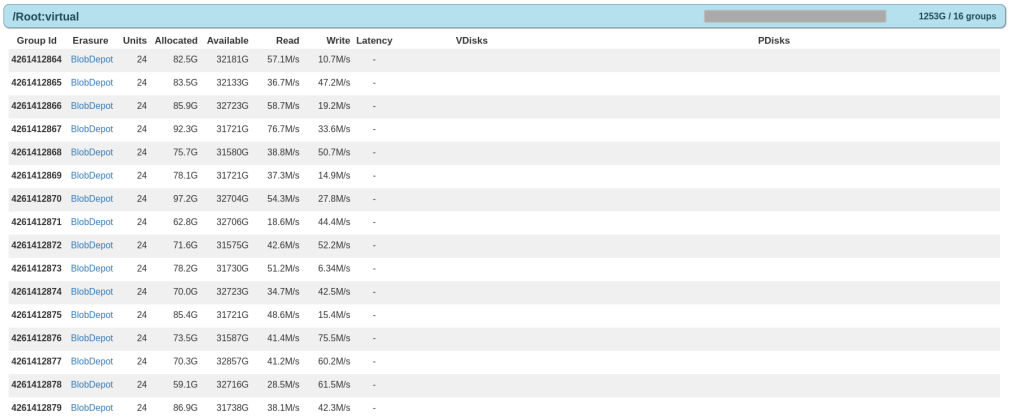
Internal Viewer

On the Internal viewer monitoring page shown below, blob depots can be seen in the Storage sections and as BD tablets.

In the Nodes section, BD tablets running on different system nodes are visible:



In the Storage section, you can see virtual groups that work through blob depot. They can be distinguished by the link with the text BlobDepot in the Erasure column. The link in this column leads to the tablet monitoring page. Otherwise, virtual groups are displayed the same way, except that they have no PDisk and VDisk. However, decommissioned groups will look the same as virtual ones, but have PDisk and VDisk until decommission is complete.



Event Log

The blob depot tablet writes events to the log with the following component names:

- `BLOB_DEPOT` — blob depot tablet component.
- `BLOB_DEPOT_AGENT` — blob depot agent component.
- `BLOB_DEPOT_TRACE` — special component for debug tracing of all data-related events.

BLOB_DEPOT and BLOB_DEPOT_AGENT are output as structured records that have fields allowing identification of the blob depot and the group it serves. For BLOB_DEPOT this is the Id field and it has the format {TabletId:GroupId}:Generation, where TabletId is the blob depot tablet number, GroupId is the group number it serves, Generation is the generation in which the running blob depot writes messages to the log. For BLOB_DEPOT_AGENT this field is called AgentId and has the format {TabletId:GroupId}.

At DEBUG level, most occurring events will be logged, both on the tablet side and on the agent side. This mode is used for debugging and is not recommended in production environments due to the large number of generated events.

Charts

Each blob depot tablet provides the following charts:

Chart	Type	Description
TotalStoredDataSize	simple	Amount of saved user data net (if there are multiple references to one blob, it is counted once).
TotalStoredTrashSize	simple	Amount of bytes in garbage data that is no longer needed but has not yet been passed to garbage collection.
InFlightTrashSize	simple	Amount of garbage bytes that are still waiting for write confirmation to the local database (they cannot even start being collected yet).
BytesToDecommit	simple	Amount of data bytes remaining to be decommissioned (if this blob depot is operating in group decommission mode).
Puts/Incoming	cumulative	Rate of incoming write requests (in pieces per unit time).

Puts/Ok	cumulative	Number of successfully executed write requests.
Puts/Error	cumulative	Number of write requests completed with an error.
Decommit/GetBytes	cumulative	Data read rate during decommission.
Decommit/PutOkBytes	cumulative	Data write rate during decommission (only successfully executed writes are counted).

Group Decommissioning

Physical groups are a valuable resource in the cluster: groups can be created, but they cannot be deleted without deleting the database that uses them, since there is no mechanism for guaranteed eviction of tablet data from the group. At the same time, the number of physical groups is determined by the cluster size, and groups cannot be moved from one tenant's pool to another tenant's pool due to the use of different encryption keys for different tenants.

This can lead to a situation where there are not enough resources to create a new group to expand an existing database or create a new database, and it is also impossible to delete an old group to free up resources, since it may contain data.

To solve this problem, you can create a virtual group with channels on top of the remaining groups in the pool, copy data from the physical group to it, and then free up the resources occupied by the physical group. This task is solved by the group decommissioning process.

Group decommissioning allows removing redundant VDisks from PDisks while preserving the data of this group. This mode is implemented by creating a blob depot that starts serving the decommissioned group instead of DS proxy. In parallel, the blob depot copies data from the physical decommissioned group. As soon as all data is copied, the physical VDisks are deleted and resources are freed, while all data from the decommissioned group is distributed across other groups.

The decommissioning process is completely transparent to tablets and users and consists of several stages:

1. Creating a blob depot tablet and distributing the group configuration to block writes to the physical group disks.
2. Copying lock metadata from the physical group. After this moment, the decommissioned group becomes available for work. Before the lock copying moment, working with the group is impossible. However, this process takes a very short time, so it is practically invisible to the client. Requests arriving at this moment are queued and wait for the stage to complete.
3. Copying barrier metadata from the physical group.
4. Copying blob metadata from the physical group.
5. Copying blob data from the physical group.
6. Deleting VDisks of the physical group.

It is worth noting once again that from the moment of blocking writes to the physical group until the moment of reading all locks, work with the group is suspended. The suspension time under normal operation is fractions of a second.

How to Launch

To start decommissioning, a BS_CONTROLLER command is executed, in which you need to specify the list of groups to be decommissioned, as well as the number of the Hive tablet that will manage the blob depots of the decommissioned groups. You can also specify a list of pools where the blob depot will store its data. If this list is not specified, BS_CONTROLLER automatically selects the same pools where the decommissioned groups are located for data storage, and the number of data channels is made equal to the number of physical groups in these pools (but no more than 250).

```
dstool -e ... --direct group decommit --group-ids 2181038080 --database=/Root/db1 --wait
```

Command line parameters:

- `--wait` — wait for the decommission to start; if a startup error occurs, the error is displayed on the screen and the decommission is canceled automatically (only when this option is specified).
- `--group-ids` GROUP_ID — GROUP_ID list of groups for which decommissioning can be performed.
- `--database=DB` — specify the tenant in which decommissioning should be done (or the domain, if decommission is performed for groups within the domain).
- `--log-channel-sp=POOL_NAME` — name of the pool where channel 0 of the blob depot tablet will be placed.
- `--snapshot-channel-sp=POOL_NAME` — name of the pool where channel 1 of the blob depot tablet will be placed; if not specified, the value from `--log-channel-sp` is used.
- `--data-channel-sp=POOL_NAME[*COUNT]` — name of the pool where data channels are placed; if the `COUNT` parameter is specified (after the asterisk), `COUNT` data channels are created in the specified pool.

If neither `--log-channel-sp` , nor `--snapshot-channel-sp` , nor `--data-channel-sp` are specified, then the storage pool to which the decommissioned group belongs is automatically found, and the zero and first channels of the blob depot are created in it, as well as N data channels, where N is the number of remaining physical groups in this pool.

How to Check that Everything is Running

You can view the decommissioning result similarly to creating virtual groups. For decommissioned groups, an additional DecommitStatus field appears, which can take one of the following values:

- `NONE` — decommissioning is not performed for the specified group
- `PENDING` — group decommissioning is expected but not yet performed (blob depot is being created)
- `IN_PROGRESS` — group decommissioning is in progress (all writes already go to the blob depot, reads go to the blob depot and the old group)
- `DONE` — decommissioning is completely finished

```
$ dstool --cluster=$CLUSTER --direct group list --virtual-groups-only
┌─────────────┬──────────────┬────────────────┬──────────────┬─────────────────┬──────────────────┬─────────────────┬──────────────
──────────────┬──────────────────┬──────────────┬──────────────────┐
| GroupId     | BoxId:PoolId | PoolName       | Generation | ErasureSpecies | OperatingStatus | VDisks_TOTAL |
VirtualGroupState | VirtualGroupName | BlobDepotId       | ErrorReason | DecommitStatus |
├─────────────┼──────────────┼────────────────┼──────────────┼─────────────────┼──────────────────┼─────────────────┼──────────────
──────────────┼──────────────────┼──────────────┼──────────────────┤
| 2181038080 | [1:1]        | /Root:ssd      | 2          | block-4-2       | FULL             | 8               |
WORKING           |                  | 72075186224038160 |             | IN_PROGRESS  |
| 2181038081 | [1:1]        | /Root:ssd      | 2          | block-4-2       | FULL             | 8               |
WORKING           |                  | 72075186224038161 |             | IN_PROGRESS  |
| 4261412864 | [1:2]        | /Root:virtual  | 0          | none            | DISINTEGRATED    | 0               |
WORKING           | vg1              | 72075186224037888 |             | NONE         |
| 4261412865 | [1:2]        | /Root:virtual  | 0          | none            | DISINTEGRATED    | 0               |
WORKING           | vg2              | 72075186224037890 |             | NONE         |
| 4261412866 | [1:2]        | /Root:virtual  | 0          | none            | DISINTEGRATED    | 0               |
WORKING           | vg3              | 72075186224037889 |             | NONE         |
```

```
│ 4261412867 │ [1:2]        │ /Root:virtual │ 0          │ none           │ DISINTEGRATED   │ 0            │
WORKING           │ vg4              │ 72075186224037891 │             │ NONE           │
└──────────────┴──────────────┴──────────────┴──────────────┴──────────────┴──────────────┴──────────────┴─
────────────────┴────────────────┴────────────────┴──────────────┴──────────────┘
```

How to Assess Progress

To assess the time and progress of decommissioning, charts are provided that allow you to understand:

- whether decommissioning is in progress (Decommit/GetBytes)
- whether data writing is happening (Decommit/PutOkBytes)
- how much data remains to be decommissioned (BytesToDecommit)

If everything is executed successfully, the Decommit/GetBytes rate approximately corresponds to Decommit/PutOkBytes. Minor discrepancies are acceptable due to the fact that decommissioned data may become outdated and be deleted by the tablet that stores data in it.

To estimate the remaining decommissioning time, it is sufficient to divide BytesToDecommit by the average Decommit/PutOkBytes rate.

Deploying YDB with Federated Query functionality

Warning

This functionality is in the "Experimental" mode.

General Installation Scheme

YDB can perform federated queries to external sources, for example, object storages or relational DBMS, without the need to move the data from external sources directly into YDB. This section describes the changes that are required in the configuration of YDB and the surrounding infrastructure to enable federated queries.

Note

A special microservice called connector must be deployed to access some of data sources. Check the list of supported sources to determine if you need to install a connector.

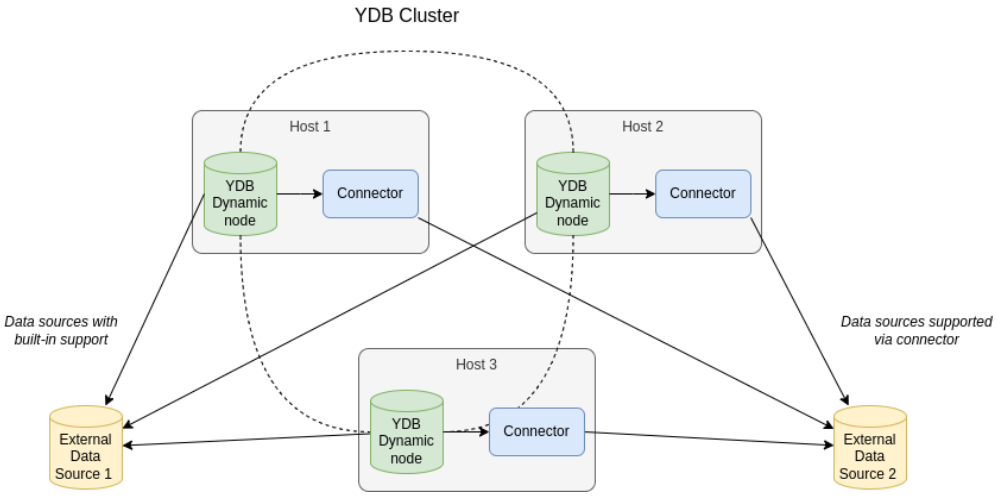
The YDB cluster and external data sources in a production installation should be deployed on different physical or virtual servers, including clouds. If access to a specific source requires a connector, it should be deployed on the same servers as the dynamic nodes of YDB. In other words, each `ydbd` process running in dynamic node mode should have one local connector process.

The following requirements must be met:

- The external data source must be accessible over the network to queries from YDB database nodes or from the connector, if present.
- The connector must be accessible over the network from YDB database nodes.

Tip

The easiest way to make the connector accessible from YDB nodes is to run them on the same set of hosts.



Note

Currently, we do not support deploying the connector in Kubernetes, but we plan to add it shortly.

Step-by-Step Guide

1. Follow the steps in the dynamic node YDB deployment guide up to and including preparing the configuration files.
2. If a connector must be deployed to access the desired source, do so according to the instructions.
3. If a connector needs to be deployed to access your desired source, add the `generic` subsection to the `query_service_config` section of the YDB configuration file as shown below. Specify the network address of the connector in the `connector.endpoint.host` and `connector.endpoint.port` fields (default values are `localhost` and `2130`). When co-locating the connector and the YDB dynamic node on the same server, encrypted connections between them are *not required*. If necessary, you can enable encryption by setting `connector.use_ssl` to `true` and specifying the path to the CA certificate that is used to sign the connector's TLS keys in `connector.ssl_ca_crt`:

```
query_service_config:
    generic:
        connector:
            endpoint:
                host: localhost                # hostname where the connector is deployed
                port: 2130                     # port number for the connector's listening socket
            use_ssl: false                     # flag to enable encrypted connections
            ssl_ca_crt: "/opt/ydb/certs/ca.crt" # (optional) path to the CA certificate
        default_settings:
            - name: DateTimeFormat
              value: string
            - name: UsePredicatePushdown
              value: "true"
```

4. Add the following `feature_flags` section to the YDB configuration file:

```
feature_flags:
    enable_external_data_sources: true
    enable_script_execution_operations: true
```

5. Continue deploying YDB database nodes. See the instructions.

Deploying Connectors to External Data Sources

> **Warning**
>
> This functionality is in "Experimental" mode.

Connectors are special microservices providing YDB with a universal abstraction for accessing external data sources. Connectors act as extension points for the YDB federated query processing system. This guide will discuss the specifics of deploying connectors in an on-premise environment.

fq-connector-go

The `fq-connector-go` connector is implemented in Go; its source code is hosted on GitHub. It provides access to the following data sources:

- ClickHouse
- Greenplum
- Microsoft SQL Server
- MySQL
- PostgreSQL
- YDB

The connector can be installed using a binary distribution or a Docker image.

Running from a Binary Distribution

Use binary distributions to install the connector on a physical or virtual Linux server without container virtualization.

1. On the releases page of the connector, select the latest release and download the archive for your platform and architecture. The following command downloads version `v0.2.4` of the connector for the Linux platform and `amd64` architecture:

   ```
   mkdir /tmp/connector && cd /tmp/connector
   wget https://github.com/ydb-platform/fq-connector-go/releases/download/v0.2.4/fq-connector-go-v0.2.4-lin
   ux-amd64.tar.gz
   tar -xzf fq-connector-go-v0.2.4-linux-amd64.tar.gz
   ```

2. If YDB nodes have not yet been deployed on the server, create directories for storing executable and configuration files:

   ```
   sudo mkdir -p /opt/ydb/bin /opt/ydb/cfg
   ```

3. Place the extracted executable and configuration files of the connector into the newly created directories:

   ```
   sudo cp fq-connector-go /opt/ydb/bin
   sudo cp fq-connector-go.yaml /opt/ydb/cfg
   ```

4. In the recommended usage mode, the connector is deployed on the same servers as the dynamic nodes of YDB, so encryption of network connections between them *is not required*. However, if you need to enable encryption, prepare a pair of TLS keys and specify the paths to the public and private keys in the `connector_server.tls.cert` and `connector_server.tls.key` fields of the `fq-connector-go.yaml` configuration file:

   ```
   connector_server:
     # ...
     tls:
       cert: "/opt/ydb/certs/fq-connector-go.crt"
       key: "/opt/ydb/certs/fq-connector-go.key"
   ```

5. If external data sources use TLS, the connector will need a root or intermediate Certificate Authority (CA) certificate that signed the sources' certificates to establish encrypted connections. Linux servers usually have some CA root certificates pre-installed. For Ubuntu OS, the list of supported CAs can be displayed with the following command:

   ```
   awk -v cmd='openssl x509 -noout -subject' '/BEGIN/{close(cmd)};{print | cmd}' < /etc/ssl/certs/ca-certif
   icates.crt
   ```

   If the server lacks the required CA certificate, copy it to a special system directory and update the certificates list:

   ```
   sudo cp root_ca.crt /usr/local/share/ca-certificates/
   sudo update-ca-certificates
   ```

6. You can start the service manually or using `systemd`.

   Manually

   Start the service from the console with the following command:

   ```
   /opt/ydb/bin/fq-connector-go server -c /opt/ydb/cfg/fq-connector-go.yaml
   ```

   Using systemd

   Along with the binary distribution, fq-connector-go includes a sample configuration file (unit) for the `systemd` initialization system. Copy the unit to the `/etc/systemd/system` directory, enable, and start the service:

```
cd /tmp/connector
sudo cp fq-connector-go.service /etc/systemd/system/
sudo systemctl enable fq-connector-go.service
sudo systemctl start fq-connector-go.service
```

If successful, the service should enter the `active (running)` state. Check it with the following command:

```
sudo systemctl status fq-connector-go
● fq-connector-go.service - YDB FQ Connector Go
    Loaded: loaded (/etc/systemd/system/fq-connector-go.service; enabled; vendor preset: enabled)
    Active: active (running) since Thu 2024-02-29 17:51:42 MSK; 2s ago
```

Service logs can be read using the command:

```
sudo journalctl -u fq-connector-go.service
```

Running in Docker

1. To run the connector, use the official Docker image. It already contains the service's configuration file. Start the service with default settings using the following command:

   ```
   docker run -d \
       --name=fq-connector-go \
       -p 2130:2130 \
       ghcr.io/ydb-platform/fq-connector-go:latest
   ```

   A listening socket of the GRPC service connector will start on port 2130 of your host's public network interface. Subsequently, the YDB server must connect to this network address.
2. If configuration changes are needed, prepare the configuration file based on the sample and mount it to the container:

   ```
   docker run -d \
       --name=fq-connector-go \
       -p 2130:2130 \
       -v /path/to/config.yaml:/opt/ydb/cfg/fq-connector-go.yaml
       ghcr.io/ydb-platform/fq-connector-go:latest
   ```

3. In the recommended usage mode, the connector is deployed on the same servers as the dynamic nodes of YDB, so encryption of network connections between them *is not required*. However, if you need to enable encryption between YDB and the connector, prepare a pair of TLS keys and specify the paths to the public and private keys in the `connector_server.tls.cert` and `connector_server.tls.key` fields of the configuration file:

   ```
   connector_server:
     # ...
     tls:
       cert: "/opt/ydb/certs/fq-connector-go.crt"
       key: "/opt/ydb/certs/fq-connector-go.key"
   ```

   When starting the container, mount the directory with the TLS key pair inside it so that they are accessible to the `fq-connector-go` process at the paths specified in the configuration file:

   ```
   docker run -d \
       --name=fq-connector-go \
       -p 2130:2130 \
       -v /path/to/config.yaml:/opt/ydb/cfg/fq-connector-go.yaml
       -v /path/to/keys/:/opt/ydb/certs/
       ghcr.io/ydb-platform/fq-connector-go:latest
   ```

4. If external data sources use TLS, the connector will need a root or intermediate Certificate Authority (CA) certificate that signed the sources' certificates to establish encrypted connections. The Docker image for the connector is based on the Alpine Linux distribution image, which already contains some CA certificates. Check for the required CA in the pre-installed list with the following command:

   ```
   docker run -it --rm ghcr.io/ydb-platform/fq-connector-go sh
   # then in the console inside the container:
   apk add openssl
   awk -v cmd='openssl x509 -noout -subject' ' /BEGIN/{close(cmd)};{print | cmd}' < /etc/ssl/certs/ca-certi
   ficates.crt
   ```

   If the source TLS keys are issued by a CA that is not included in the trusted list, add the CA certificate to the system paths of the container with the connector. For example, build a custom Docker image based on the existing one. Prepare the following `Dockerfile`:

   ```
   FROM ghcr.io/ydb-platform/fq-connector-go:latest

   USER root

   RUN apk --no-cache add ca-certificates openssl
   COPY root_ca.crt /usr/local/share/ca-certificates
   RUN update-ca-certificates
   ```

Place the `Dockerfile` and the CA root certificate in one folder, navigate to it, and build the image with the following command:

```
docker build -t fq-connector-go_custom_ca .
```

The new `fq-connector-go_custom_ca` image can be used to deploy the service using the above commands.

Configuration

A current example of the `fq-connector-go` service configuration file can be found in the repository.

Parameter	Description
`connector_server`	Required section. Contains the settings of the main GPRC server that accesses the data.
`connector_server.endpoint.host`	Hostname or IP address on which the service's listening socket runs.
`connector_server.endpoint.port`	Port number on which the service's listening socket runs.
`connector_server.tls`	Optional section. Filled if TLS connections are required for the main GRPC service `fq-connector-go`. By default, the service runs without TLS.
`connector_server.tls.key`	Full path to the private encryption key.
`connector_server.tls.cert`	Full path to the public encryption key.
`logger`	Optional section. It contains logging settings.
`logger.log_level`	Logging level. Valid values: `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`. Default value: `INFO`.
`logger.enable_sql_query_logging`	For data sources supporting SQL, query logging is enabled. Valid values: `true`, `false`. **IMPORTANT**: Enabling this option may result in printing confidential user data in the logs. Default value: `false`.
`paging`	Optional section. It contains settings for the algorithm of splitting the data stream extracted from the source into Arrow blocks. For each request, a queue of blocks prepared for sending to YDB is created in the connector. Arrow block allocation contributes significantly to the memory consumption of the `fq-connector-go` process. The minimum memory required for the connector's operation can be roughly estimated by the formula , where is the number of concurrent requests, is the `paging.bytes_per_page` parameter, and is the `paging.prefetch_queue_capacity` parameter.
`paging.bytes_per_page`	Maximum number of bytes in one block. Recommended values range from 4 to 8 MiB, and the maximum is 48 MiB. Default value: 4 MiB.
`paging.prefetch_queue_capacity`	Number of pre-read data blocks stored in the connector's address space until YDB requests the next data block. In some scenarios, larger values of this setting can increase throughput but will also lead to higher memory consumption by the process. Recommended values - at least 2. Default value: 2.

YDB Enterprise Manager

YDB Enterprise Manager (hereinafter referred to as YDB EM) is a tool for centralized management of YDB clusters through a web interface and API.

> **Note**
>
> Deploying YDB clusters is possible without YDB EM — using Ansible, Kubernetes, or manually. YDB EM provides a convenient web interface on top of existing clusters.

Purpose

YDB EM connects to existing YDB clusters and provides a graphical interface and API for the following tasks:

- centralized access to databases and YDB clusters from a single interface;
- managing dynamic nodes of a cluster — starting, stopping, and scaling;
- managing databases — creating, deleting, and modifying parameters;
- monitoring cluster and node status;
- managing resources (CPU, RAM) allocated to dynamic nodes;
- advisor — diagnostics and recommendations for resolving the most common issues;
- advanced SQL editor for running queries against databases;
- AI assistant for working with YDB.

> **Warning**
>
> YDB EM does not deploy YDB clusters. Before using YDB EM, the cluster must be deployed using one of the deployment options.

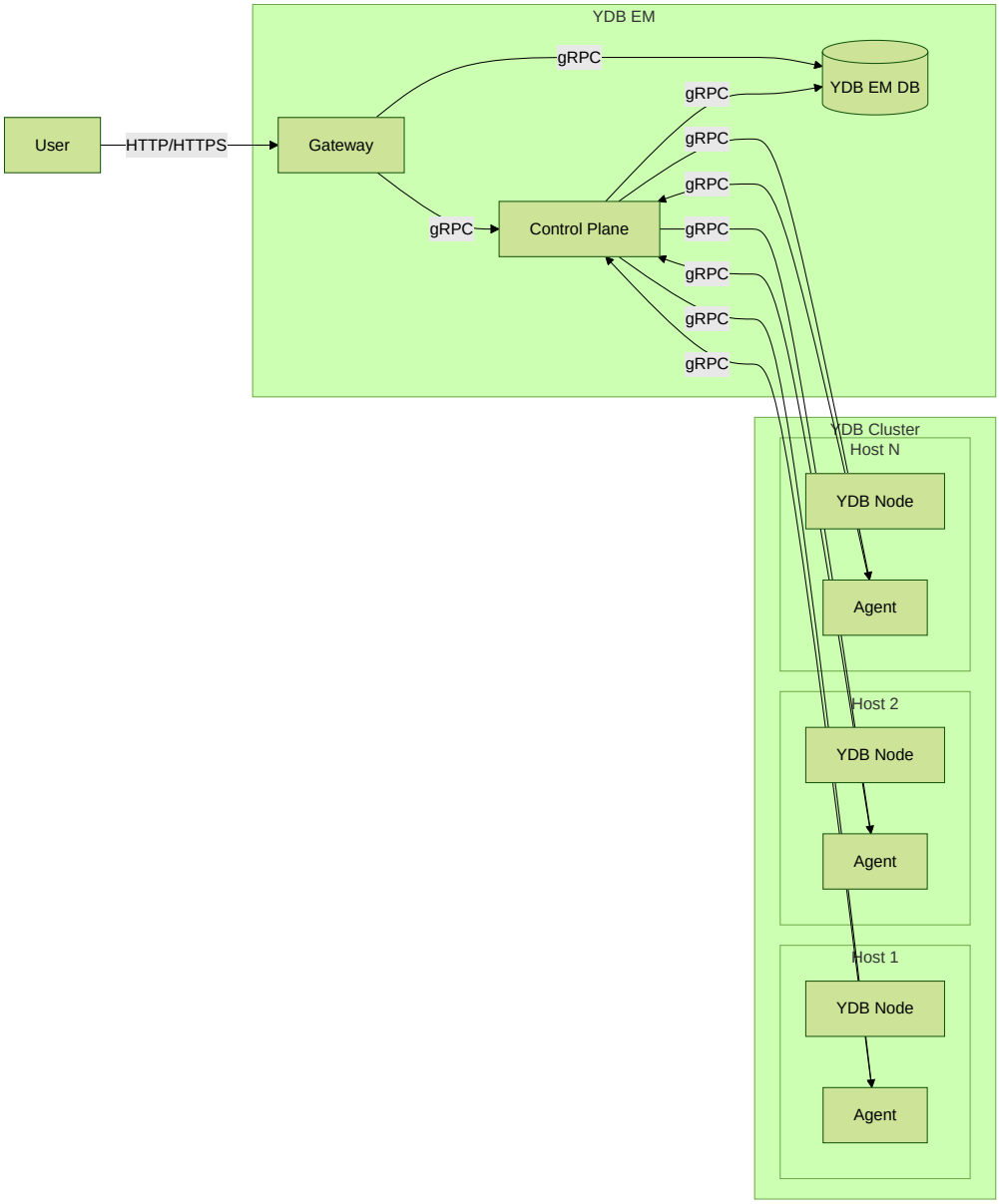
Architecture

YDB EM consists of three components:

- **Gateway** — web interface and API backend. Accepts requests from users (via a browser or API) and communicates with the Control Plane and the YDB EM database.
- **Control Plane (CP)** — coordinates cluster management. Receives commands from Gateway, stores configuration in the YDB EM database, and dispatches tasks to agents.
- **Agent** — runs on each YDB cluster host where dynamic nodes operate. The agent executes Control Plane commands: starts and stops YDB node processes, monitors their state, and reports information about available host resources.

To store its own metadata (cluster configuration, node state), YDB EM uses a YDB database — it can reside in the same cluster that EM manages.

Interaction diagram



The user interacts with Gateway through a browser or API. Gateway forwards requests to the Control Plane, which coordinates the work of agents on cluster hosts. Agents manage YDB node processes and report host status.

Key topics

- Initial deployment of YDB EM
- Configuring S3 backups for YDB EM

Initial deployment of YDB EM

This guide describes the process of initial deployment of YDB Enterprise Manager (hereinafter referred to as YDB EM) using Ansible. Upon completion, you will have a running YDB EM instance connected to an existing YDB cluster.

Before you start

Requirements

To deploy YDB EM, you need:

1. **A running YDB cluster**. YDB EM connects to an existing cluster to manage its resources; EM cannot operate without a cluster. A service database for storing YDB EM metadata will be automatically created in this cluster. For more information on deploying a cluster, see YDB Deployment Options.
2. **YDB database connection parameters** for the database that will be used to store YDB EM metadata:
   - Endpoint (for example, `grpcs://ydb-node01.ru-central1.internal:2135`).
   - Username and password for obtaining an authorization token. The user must have permissions to modify the schema, write, and read data. If the YDB cluster is configured for LDAP authentication, use an account from the corresponding directory.
3. **A host for Gateway and Control Plane components** — a single server (physical or virtual) where the YDB EM server components will be installed. This can be any of the YDB cluster hosts or a separate machine.
4. **Ansible** on the control machine (from which the installation will be performed). ansible-core 2.14–2.18 is recommended; for more details, see Deploying a YDB cluster with Ansible.
5. **SSH access** from the control machine to all servers where YDB EM components will be installed (the Gateway/CP host and cluster hosts for agents).

Network requirements

The network configuration must allow TCP connections on the following ports (default values):

Source	Destination	Port	Protocol	Purpose
User	Gateway	8789	HTTP/HTTPS	Web interface and API
Gateway, Agent	Control Plane	8787	gRPC	Management commands
Gateway, Control Plane	YDB	2135	gRPC	Connection to YDB EM DB

Downloading the package

Download the YDB EM package to the control machine (from which the installation via Ansible will be performed). Check the latest release information for the current version number and archive link:

Version	Link
`<VERSION>`	`https://binaries.ydbem.website.yandexcloud.net/builds/<VERSION>/ydb-em-<VERSION>-stable-linux-amd64.tar.xz`

Extract the downloaded archive into a working directory (replace `<VERSION>` with the actual version):

```
tar -xf ydb-em-<VERSION>-stable-linux-amd64.tar.xz
```

Package contents:

File	Description
`bin/ydb-em-gateway`	Gateway binary
`bin/ydb-em-cp`	Control Plane binary
`bin/ydb-em-agent`	Agent binary
`collections/ydb_platform-ydb-*.tar.gz`	Ansible collection for YDB
`collections/ydb_platform-ydb_em-*.tar.gz`	Ansible collection for YDB EM
`examples.tar.gz`	Ansible configuration templates (inventory and playbooks)
`install.sh`	Automatic installation script

Installing Ansible collections

Install the Ansible collection for YDB if it is not already installed:

```
ansible-galaxy collection install collections/ydb_platform-ydb-*.tar.gz
```

Install the Ansible collection for YDB EM:

```
ansible-galaxy collection install collections/ydb_platform-ydb_em-*.tar.gz
```

Note

Ansible collection versions are included in the YDB EM package. Use the files from the `collections/` directory of your package.

Preparing the configuration

Extracting configuration templates

The YDB EM package includes ready-made Ansible configuration templates — inventory files, playbooks, and directories for placing binaries and certificates. Extract this archive:

```
tar -xf examples.tar.gz
```

This will create the `examples/` directory, which will be used as the working directory for the installation.

Placing files

1. Copy the binary files from the `bin/` directory of the package to the `examples/files/` directory:

   ```
   cp bin/ydb-em-gateway bin/ydb-em-cp bin/ydb-em-agent examples/files/
   ```

2. Place TLS certificates in the `examples/files/certs/` directory. Use the same certificates as for the YDB cluster nodes (CA certificate, node certificates, and keys):

   ```
   cp /path/to/your/certs/* examples/files/certs/
   ```

Configuring the Ansible inventory

Ansible uses inventory files to describe target servers and their parameters. The YDB EM configuration is split into several files:

- `examples/inventory/50-inventory.yaml` — common parameters: SSH connection, YDB connection, Control Plane settings.
- `examples/inventory/90-inventory.yaml` — server descriptions (which hosts are used for which components).
- `examples/inventory/99-inventory-vault.yaml` — sensitive data (passwords).

Configuring hosts

Open the file `examples/inventory/90-inventory.yaml` and configure two host groups.

Group `ydb_em` — hosts for YDB EM server components (Gateway and Control Plane). A single host is sufficient — it can be any of the YDB cluster hosts or a separate server:

```
ydb_em:
  hosts:
    ydb-node01.ru-central1.internal:
```

Group `ydbd_dynamic` — YDB cluster hosts where the Agent will be installed. An Agent is needed on every host that runs YDB dynamic nodes — YDB EM manages node processes through these agents. Typically, all cluster hosts should be included:

```
ydb:
  children:
    ydbd_dynamic:
      hosts:
        ydb-node01.ru-central1.internal:
            ydb_em_agent_cpu: 4
            ydb_em_agent_memory: 8
            location: db-dc-1
        ydb-node02.ru-central1.internal:
            ydb_em_agent_cpu: 4
            ydb_em_agent_memory: 16
            location: db-dc-2
        ydb-node03.ru-central1.internal:
            location: db-dc-3
    ydb_em:
      hosts:
        ydb-node01.ru-central1.internal:
```

For each host in the `ydbd_dynamic` group, you can specify additional parameters. If parameters are not specified, default values will be used. Resource settings can be adjusted later through the YDB EM web interface.

Parameter	Description
`ydb_em_agent_cpu`	Number of CPUs available for nodes on the host
`ydb_em_agent_memory`	Amount of RAM (in gigabytes) available for nodes on the host
`ydb_em_agent_name`	Host name used by the agent
`location`	Host location (availability zone)

> **Warning**
>
> YDB binaries must be installed on all hosts in the `ydbd_dynamic` group before deploying YDB EM. For more details, see Deploying a YDB cluster with Ansible.

Configuring SSH connection

In the file `examples/inventory/50-inventory.yaml`, verify the SSH connection parameters from the control machine to the target servers:

```
# Remote user with sudo privileges
ansible_user: ansible

# Settings for connecting through a Bastion/Jump host (JUMP_IP)
# ansible_ssh_common_args: "-o ProxyJump=ansible@{{ lookup('env','JUMP_IP') }} -o UserKnownHostsFile=/dev/nul
l -o StrictHostKeyChecking=no"

# Common SSH settings
ansible_ssh_common_args: "-o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no"

# Private key for connection
ansible_ssh_private_key_file: "~/.ssh/id_ed25519"
```

Warning

The `UserKnownHostsFile=/dev/null` and `StrictHostKeyChecking=no` parameters disable SSH host key verification, making the connection vulnerable to man-in-the-middle (MITM) attacks. This configuration is only acceptable for test environments. In production environments, it is recommended to use host key verification by specifying the path to a real `known_hosts` file:

```
ansible_ssh_common_args: "-o StrictHostKeyChecking=yes"
```

Note

If a Bastion/Jump host is used to access the servers, uncomment and configure the corresponding `ansible_ssh_common_args` line.

Configuring the YDB database connection

In the file `examples/inventory/50-inventory.yaml`, specify the connection parameters for the YDB database that will be used to store YDB EM metadata:

```
# YDB connection endpoint
ydb_em_db_connection_endpoint: "grpcs://ydb-node01.ru-central1.internal:2135"
# Root domain
ydb_em_db_connection_db_root: "/Root"
# Database name
ydb_em_db_connection_db: "db"
# YDB user
ydb_user: root
```

The YDB user password is set in the file `examples/inventory/99-inventory-vault.yaml`:

```
all:
  children:
    ydb:
      vars:
        ydb_password: <password>
```

Tip

To protect sensitive data, it is recommended to encrypt the `99-inventory-vault.yaml` file using Ansible Vault:

```
ansible-vault encrypt examples/inventory/99-inventory-vault.yaml
```

Configuring the Control Plane connection

In the file `examples/inventory/50-inventory.yaml`, specify the YDB EM Control Plane connection endpoint:

```
# The endpoint must point to a host from the ydb_em group
ydb_em_cp_connection_endpoint: "grpcs://ydb-node01.ru-central1.internal:8787"
```

This parameter is used in the configuration of **Gateway** and **Agent** for connecting to the **Control Plane**.

Note

For high availability, you can specify a load balancer address (for example, an L3/L4 load balancer or DNS-based balancing) instead of a direct host address.

Running the installation

After completing the configuration, navigate to the `examples` directory and run the installation using one of the following methods:

Automatic installation

Run the automatic installation script:

```
cd examples
./install.sh
```

The script will automatically perform all the necessary steps to deploy YDB EM components.

Manual Ansible run

You can also run the installation manually by executing the Ansible playbook:

```
ansible-playbook ydb_platform.ydb_em.initial_setup
```

> **Note**
>
> If the `99-inventory-vault.yaml` file is encrypted with Ansible Vault, add the `--ask-vault-pass` flag:
>
> ```
> ansible-playbook ydb_platform.ydb_em.initial_setup --ask-vault-pass
> ```

Verifying the installation

After the installation completes successfully, open the YDB EM web interface in a browser:

```
https://<FQDN of the host from the ydb_em group>:8789/ui/clusters
```

For example:

```
https://ydb-node01.ru-central1.internal:8789/ui/clusters
```

The web interface should display a page with a list of YDB clusters.

Troubleshooting

Cannot connect to the web interface

1. Make sure port `8789` is accessible from your machine:

   ```
   curl -k https://<FQDN>:8789/ui/clusters
   ```

2. Check the status of the YDB EM Gateway service on the target host:

   ```
   sudo systemctl status ydb-em-gateway
   ```

3. Check the service logs:

   ```
   sudo journalctl -u ydb-em-gateway -n 100
   ```

Agent cannot connect to Control Plane

1. Make sure port `8787` is accessible between the cluster nodes and the Control Plane host.
2. Check the status of the YDB EM Agent service on the node:

   ```
   sudo systemctl status ydb-em-agent
   ```

3. Verify the correctness of TLS certificates and the Control Plane endpoint in the agent configuration.

YDB database connection errors

1. Make sure the endpoint, database name, username, and password are specified correctly.
2. Check network accessibility of port `2135` on the YDB node from the YDB EM host.
3. Check the YDB EM Control Plane logs:

   ```
   sudo journalctl -u ydb-em-cp -n 100
   ```

Configuring S3 backups for YDB EM

This guide describes how to configure database backups to an S3-compatible storage through YDB Enterprise Manager (YDB EM). Backups are configured in the Control Plane configuration file. In the examples below, the path to this file is shown as `<ydb-em-cp-config.yaml>`.

S3 backup configuration consists of three parts:

- `backup_targets` — where to store backups;
- `secret_key` — which master key is used to encrypt S3 access keys;
- `locations[].default_backup_config` — when to run backups and how long to retain them.

> **Note**
>
> This article describes automated backup configuration through YDB EM. For general information about backup options in YDB, see Backup concepts and Backup and recovery.

Configuring S3 storage

Add the `backup_targets` block to the Control Plane configuration:

```
backup_targets:
  - target_id: "target-em"
    tags:
      locations:
        - "em"
    settings:
      s3:
        endpoint: s3.example.net
        bucket: ydb-em-backups
        access_key: "<encrypted access key>"
        secret_key: "<encrypted secret key>"
        compression: "zstd"
```

Parameter	Description
`target_id`	Unique backup target ID.
`tags.locations`	List of `location_id` values this target applies to. A database is backed up to this target if its `location_id` is included in the list. For YDB EM, this is usually `em`, matching `locations[].database_location_id` and `meta_location_id`.
`settings.s3.endpoint`	S3-compatible storage endpoint.
`settings.s3.bucket`	Bucket name for backups.
`settings.s3.scheme`	Connection protocol: `1` for HTTP, `2` for HTTPS. If omitted, HTTPS is used.
`settings.s3.access_key`	Encrypted S3 access key.
`settings.s3.secret_key`	Encrypted S3 secret key.
`settings.s3.compression`	Compression algorithm for exported data. The default is `zstd`. Remove this parameter if compression is not required.

A **location** is a logical group of databases in YDB EM that each database belongs to through its `location_id`. Typically, a location corresponds to a single placement zone of databases — for example, a data center or availability zone where they run — but a location can also be used for logical separation, such as by environment (`prod` , `test`). For example, if databases are deployed across two data centers, you can assign them different locations and send backups from each data center to its own S3 storage. A typical YDB EM installation uses a single location with the value `em`, which matches the `locations[].database_location_id` and `meta_location_id` fields in the Control Plane configuration.

`tags.locations` is useful when one Control Plane manages databases from several locations and different locations must use different backup storage targets. The worker selects a target by the database `location_id`: if the database `location_id` is included in `tags.locations`, this target is used for backups. For example, this lets you send backups from different zones or environments to different S3 buckets.

> **Note**
>
> Instead of binding a target to `location_id`, you can use `explicit_backup_targets` with `by_database_path` or `by_cloud_id` rules if a specific database or cloud must be assigned to a specific target.

Configuring the master key

At the top level of the configuration file, specify the path to the master key file:

```
secret_key: configs/em/secret_key
```

This master key is used to encrypt and decrypt `settings.s3.access_key` and `settings.s3.secret_key` values. The master key file must be available to Control Plane processes at the path specified in the `secret_key` parameter.

> **Warning**
>
> Do not store `access_key` and `secret_key` in the configuration as plaintext. YDB EM expects encrypted values and decrypts them at runtime.

Encrypting S3 access keys

Encrypt the S3 access keys with the same master key specified in the `secret_key` parameter. Use the `ydb-em-cp` Control Plane binary, which is included in the YDB EM package as `bin/ydb-em-cp` and is placed on the Control Plane host during initial deployment:

```
ydb-em-cp admin crypto encrypt --body '<plaintext access key>' --cfg-file <ydb-em-cp-config.yaml>
ydb-em-cp admin crypto encrypt --body '<plaintext secret key>' --cfg-file <ydb-em-cp-config.yaml>
```

Copy the resulting encrypted strings to `settings.s3.access_key` and `settings.s3.secret_key`.

To check that a value can be decrypted with the same master key, run:

```
ydb-em-cp admin crypto decrypt --body '<encrypted value>' --cfg-file <ydb-em-cp-config.yaml>
```

Configuring schedule and retention

The target defines where backups are stored. The schedule and retention period are configured separately in `locations[].default_backup_config`. This is the default backup configuration for databases in the specified location. If an individual backup configuration is set for a specific database in YDB EM, it can differ from these defaults.

```
locations:
  - database_location_id: em
    default_backup_config:
      backup_settings:
        - name: daily
          type: SYSTEM
          backup_schedule:
            daily_backup_schedule:
              execute_time:
                hours: 20
          backup_time_to_live: "604800s"
```

Parameter	Description
`backup_settings[].name`	Backup setting name.
`backup_settings[].type`	Backup setting type. Use `SYSTEM` for a system schedule.
`backup_schedule.daily_backup_schedule.execute_time.hours`	Hour of the daily backup start time in UTC.
`backup_time_to_live`	Backup retention period in seconds. For example, `604800s` means 7 days.

Applying the configuration

To apply the settings:

1. Place the updated configuration file in the Control Plane working directory. For example, for an installation under `/opt/ydb-em`, this can be `/opt/ydb-em/ydb-em-cp/cfg/config.yaml`.
2. Make sure the master key file specified in `secret_key` is available to Control Plane processes.
3. Check that `settings.s3.access_key` and `settings.s3.secret_key` are encrypted with the same master key.
4. Restart the Control Plane service:

   ```
   sudo systemctl restart ydb-em-cp
   ```

After restart, Control Plane resolves the target by `location_id` according to the schedule, configures the S3 storage, and starts data export to the specified bucket.

Checklist

Before starting backups, check that:

- `backup_targets[].tags.locations` contains the `location_id` values of the databases that require backups;
- `endpoint`, `bucket`, and `scheme` point to the correct S3 storage;
- `access_key` and `secret_key` are encrypted with `ydb-em-cp admin crypto encrypt`;
- the top-level `secret_key` parameter points to the correct master key file;
- `locations[].default_backup_config` defines the schedule and retention period;
- the updated configuration file is placed in the Control Plane working directory;
- the Control Plane service is restarted after the configuration change.

Observability overview

This section contains descriptions of tools for monitoring the YDB cluster.

Main subsections:

- Setting Up YDB Cluster Monitoring
- Logging in YDB
- Minidumps (built-in Google Breakpad)
- Tracing in YDB
- Cluster System Views
- Alerting
- Diagnostics of an isolated cluster

Setting Up YDB Cluster Monitoring

This page explains how to set up monitoring for a YDB cluster.

YDB provides numerous system state metrics. Instant metric values can be viewed in the web interface:

```
http://<ydb-server-address>:<ydb-port>/counters/
```

where:

- `<ydb-server-address>` – YDB server address.

  For a local single-node YDB cluster started using the Quick start instructions, use the address `localhost`.
- `<ydb-port>` – YDB port. Default value: 8765.

Related metrics are grouped into subgroups (for example `counters auth`). To view metric values for a specific subgroup only, navigate to a URL of the following form:

```
http://<ydb-server-address>:<ydb-port>/counters/counters=<servicename>/
```

- `<servicename>` — metric subgroup name.

For example, server hardware resource utilization data is available at the following URL:

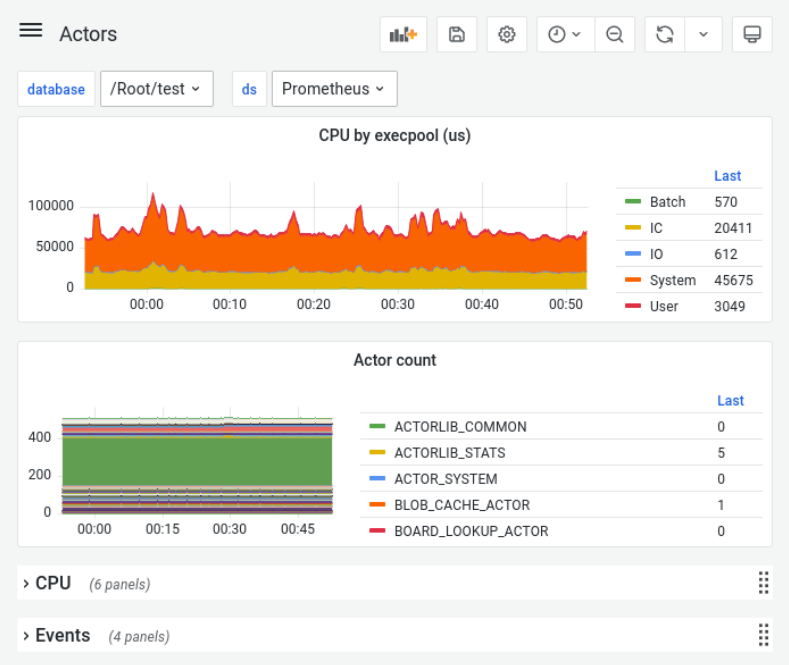
```
http://<ydb-server-address>:<ydb-port>/counters/counters=utils
```

To collect metric values, you can use the popular open-source tool Prometheus or any other system that supports this format. YDB metric values in Prometheus format are available at URLs of the following form:

```
http://<ydb-server-address>:<ydb-port>/counters/counters=<servicename>/prometheus
```

- `<servicename>` — metric subgroup name.

Data can be visualized using any system that supports the Prometheus format, such as Grafana, Zabbix or Amazon CloudWatch:



Setting Up Monitoring with Prometheus and Grafana

To set up YDB cluster monitoring using Prometheus and Grafana:

1. Install Prometheus.
2. Edit the Prometheus configuration files:
   2.1. In the `targets` section of `ydbd-storage.yml`, specify the addresses of all YDB cluster servers and the ports of storage nodes running on the servers.

   ```
   - labels:
       container: ydb-static
     targets:
     - "ydb-s1.example.com:8765"
     - "ydb-s2.example.com:8765"
     - "ydb-s3.example.com:8765"
   ```

   For a local single-node YDB cluster, specify one address in the targets section:

   ```
   - labels:
       container: ydb-static
   ```

```
  targets:
  - "localhost:8765"
```

2.2. In the `targets` section of `ydbd-database.yml`, specify the addresses of all YDB cluster servers and the ports of all database nodes running on the servers.

```
- labels:
    container: ydb-dynamic
  targets:
  - "ydb-s1.example.com:31002"
  - "ydb-s1.example.com:31012"
  - "ydb-s1.example.com:31022"
  - "ydb-s2.example.com:31002"
  - "ydb-s2.example.com:31012"
  - "ydb-s2.example.com:31022"
  - "ydb-s3.example.com:31002"
  - "ydb-s3.example.com:31012"
  - "ydb-s3.example.com:31022"
```

For a local single-node YDB cluster, specify one address in the targets section:

```
- labels:
    container: ydb-dynamic
  targets:
  - "localhost:8765"
```

2.3. If necessary, in the `tls_config` section of `prometheus_ydb.yml`, specify the Certificate Authority (CA) certificate that signed the other TLS certificates of the YDB cluster:

```
scheme: https
tls_config:
    ca_file: '<ydb-ca-file>'
```

3. Start Prometheus using `prometheus_ydb.yml` as a configuration file.
4. Install and start Grafana.
5. Create a data source with type `prometheus` in Grafana and connect it to the running Prometheus instance.
6. Upload YDB dashboards to Grafana.

   You can upload dashboards using the Grafana UI Import tool or run the script. Note that the script uses basic authentication in Grafana. For other cases, modify the script.

   See the Grafana dashboards reference.

Logging in YDB

Each YDB component writes messages of different levels to logs. These can be used to detect critical problems or understand the causes of issues.

Logging Setup

Logging configuration for individual components can be done in YDB UI.

Currently, there are two options for starting YDB logging: manually and using systemd.

Manually

For convenience, YDB provides standard mechanisms for collecting logs and metrics.
Logging is performed to standard `stdout` and `stderr` channels and can be redirected using popular solutions.

Using Systemd

By default, logs are written to `journald` and can be retrieved using `journalctl -u ydbd-storage`. For database nodes, change the systemd unit name appropriately.

Minidumps (built-in Google Breakpad)

The `ydbd` nodes in YDB can collect **minidumps** when the process crashes, using the built-in Google Breakpad. A minidump is a compact snapshot of the process state (stack, registers, list of loaded modules) at the moment of a crash caused by an unhandled signal (`SIGSEGV` , `SIGABRT` , `SIGFPE` , and others).

Unlike a system core dump, a minidump:

- takes up significantly less space;
- is collected by its own signal handler and does not depend on the host's `ulimit` and `core_pattern` settings;
- can be automatically passed to an external script for post-processing (uploading, symbolization, alerting).

> **Note**
>
> Minidump collection works on Linux only. On other platforms (for example, macOS) `ydbd` runs, but the built-in Breakpad is not part of the build and minidumps are not collected.

Enabling minidump collection

By default, minidump collection is disabled. To enable it, specify the directory where dumps will be written. This can be done in two ways: through environment variables or through `ydbd server` command-line options. Command-line options take precedence over environment variables.

If the directory is not set by either method, the built-in Breakpad is not activated and does not install signal handlers.

> **Note**
>
> The specified directory must exist before the process starts; YDB does not create it automatically. If the directory is missing at crash time, the minidump will not be written.

Environment variables

Variable	Purpose
`BREAKPAD_MINIDUMPS_PATH`	directory for writing minidumps; setting this variable enables minidump collection at process startup
`BREAKPAD_MINIDUMPS_SCRIPT`	path to the minidump post-processing script

Command-line options

The options are added to the `ydbd server` command:

Option	Purpose
`--breakpad-minidumps-path PATH`	directory for writing minidumps
`--breakpad-minidumps-script SCRIPT`	path to the minidump post-processing script

Startup example:

```
ydbd server \
    --breakpad-minidumps-path /var/log/ydb/minidumps \
    --breakpad-minidumps-script /usr/local/bin/process-minidump.sh \
    ...
```

Post-processing script

If a post-processing script is set, it is run immediately after the minidump is written. The script is called with the following arguments:

Argument	Value
`$1`	`true` or `false` — whether the minidump was written successfully
`$2`	path to the minidump file (empty string if writing failed)

The `ydbd` process environment is not passed to the script, so any parameters it needs must be hardcoded in the script itself or derived from the arguments.

> **Warning**
>
> The script is launched from the crashed process's signal handler via `fork` / `exec` . The signal handler blocks until the script exits, so the script should complete quickly. Heavy processing (uploading the dump, symbolization, alerting) should be delegated to background processes that the script spawns before exiting.

Example of a script that only moves the dump to an archive directory:

```
#!/bin/sh
# $1 — succeeded (true|false), $2 — path to the minidump
if [ "$1" = "true" ]; then
    mv "$2" /var/log/ydb/minidumps/archive/
fi
```

Analyzing minidumps

A minidump does not contain symbols: to obtain a human-readable stack, you need the symbols of the same `ydbd` build artifact. The standard Google Breakpad tools (`minidump_stackwalk` and `.sym` files) produced from the symbols of the corresponding build are used for the analysis.

See also

- Logging in YDB
- Setting Up YDB Cluster Monitoring
- Google Breakpad

Cluster System Views

For internal introspection of cluster state, users can query special service views (system views). These views are available from the cluster root directory and use the system path prefix `.sys`.

Cloud database users typically do not have access to cluster system views, as the cloud team is responsible for their maintenance and timely diagnostics.

In the descriptions of available fields below, the **Key** column contains the primary key field index of the corresponding view.

> **Note**
>
> Similar system views exist for what happens inside a specific database; they are described in a separate article for developers.

Access control

The ability to flexibly configure the access rights to system views allows for precise separation of access to service information and user data.

Typical access control scenarios

- **Access only to system views:**
  If your infrastructure has users or services that need to monitor the state of the cluster or database — for example, database administrators — but do not require access to user data, it is recommended to grant these users read-only permissions for the `.sys` directory.
- **Access only to data:**
  When you need to hide information about service and system objects from individual users or groups (for example, analysts who do not need access to internal diagnostics), it is sufficient to grant read-only permissions for the corresponding directory containing user data. In this case, there is no need to grant permissions for the `.sys` directory or the database root — this ensures that users will not see and will not be able to query system views.

You can also assign different permissions to individual system views — both for specific users and for groups. This approach enables flexible implementation of the principle of least privilege and control over different levels of access to service information for various roles and tasks.

Distributed Storage

Information about the distributed storage operation is contained in several interconnected views, each responsible for describing its own entity:

- PDisk
- VSlot
- Group
- Storage Pool

Additionally, there is a separate view that shows statistics on the usage of group numbers in different storage pools and the growth capabilities of these pools.

ds_pdisks

Field	Type	Key	Value
NodeId	Uint32	0	Identifier of the node on which PDisk is running
PDiskId	Uint32	1	PDisk identifier (unique within the node)
Type	String		Media type (`ROT`, `SSD`, `NVME`)
Kind	Uint64		User-defined numeric identifier needed to group disks with the same media type into different subgroups
Path	String		Path to the block device inside the machine
Guid	Uint64		Unique identifier randomly generated when adding a disk to the system, designed to prevent data loss in case disks are swapped
BoxId	Uint64		Identifier of the Box that includes this PDisk
SharedWithOs	Bool		Presence of the "SharedWithOs" label, set manually when creating PDisk. Can be used for filtering disks when creating new groups.
ReadCentric	Bool		Presence of the "ReadCentric" label, set manually when creating PDisk. Can be used for filtering disks when creating new groups.
AvailableSize	Uint64		Number of bytes available for allocation on PDisk
TotalSize	Uint64		Total number of bytes on PDisk
Status	String		PDisk operating mode that affects its participation in group allocation (`ACTIVE`, `INACTIVE`, `BROKEN`, `FAULTY`, `TO_BE_REMOVED`)
StatusChangeTimestamp	Timestamp		Time when `Status` last changed; if `NULL`, `Status` has not changed since PDisk creation
ExpectedSlotCount	Uint32		Maximum number of VSlots that can be created on this PDisk
NumActiveSlots	Uint32		Number of currently occupied VSlots

DecommitStatus	String		Status of PDisk decommissioning (`DECOMMIT_NONE` , `DECOMMIT_PENDING` , `DECOMMIT_IMMINENT` , `DECOMMIT_REJECTED`)

ds_vslots

Field	Type	Key	Value
NodeId	Uint32	0	Identifier of the node on which VSlot is running
PDiskId	Uint32	1	PDisk identifier within the node on which VSlot is running
VSlotId	Uint32	2	VSlot identifier within PDisk
GroupId	Uint32		Storage group number that includes this VSlot
GroupGeneration	Uint32		Storage group configuration generation that includes this VSlot
FailRealm	Uint32		Relative fail realm number of VSlot within the storage group
FailDomain	Uint32		Relative fail domain number of VSlot within the fail realm
VDisk	Uint32		Relative VSlot number within the fail domain
AllocatedSize	Uint64		Number of bytes that VSlot occupies on PDisk
AvailableSize	Uint64		Number of bytes available for allocation to this VSlot
Status	String		State of the running VDisk in this VSlot (`INIT_PENDING` , `REPLICATING` , `READY` , `ERROR`)
Kind	String		Preset VDisk operating mode setting (`Default` , `Log` , ...)

Note that the tuple `(NodeId, PDiskId)` forms a foreign key to the `ds_pdisks` view, and `(GroupId)` to the `ds_groups` view.

ds_groups

Field	Type	Key	Value
GroupId	Uint32	0	Storage group number in the cluster
Generation	Uint32		Storage group configuration generation
ErasureSpecies	String		Redundancy encoding mode for the group (`block-4-2` , `mirror-3-dc` , `mirror-3of4` , ...)
BoxId	Uint64		Identifier of the Box in which this group was created
StoragePoolId	Uint64		Storage pool identifier within the Box where this group operates
EncryptionMode	Uint32		Presence of data encryption in the group and encryption algorithm if enabled
LifeCyclePhase	Uint32		Presence of an expired encryption key if encryption is enabled
AllocatedSize	Uint64		Amount of allocated data bytes in the group (converted to user bytes, i.e., before redundancy)
AvailableSize	Uint64		Amount of user data bytes available for allocation (also before redundancy)
SeenOperational	Bool		Boolean flag showing whether the group was in operational state after its creation
PutTabletLogLatency	Interval		90th percentile of `PutTabletLog` request execution time
PutUserDataLatency	Interval		90th percentile of `PutUserData` request execution time
GetFastLatency	Interval		90th percentile of `GetFast` request execution time
OperatingStatus	String		Group status based on latest VDisk reports only (`UNKNOWN` , `FULL` , `PARTIAL` , `DEGRADED` , `DISINTEGRATED`)
ExpectedStatus	String		Status based not only on operational report, but on PDisk status and plans too (`UNKNOWN` , `FULL` , `PARTIAL` , `DEGRADED` , `DISINTEGRATED`)

In this view, the tuple `(BoxId, StoragePoolId)` forms a foreign key to the `ds_storage_pools` view.

ds_storage_pools

Field	Type	Key	Value
BoxId	Uint64	0	Identifier of the Box that includes this storage pool
StoragePoolId	Uint64	1	Storage pool identifier within the Box
Name	String		User-defined storage pool name (used for linking tablets and storage pools)
Generation	Uint64		Storage pool configuration generation (number of changes)
ErasureSpecies	String		Redundancy encoding mode for all groups within this storage pool
VDiskKind	String		Preset operating mode setting for all VDisks in this storage pool
Kind	String		User-defined string description of pool purpose, can also be used for filtering

NumGroups	Uint32		Number of groups within this storage pool
EncryptionMode	Uint32		Data encryption setting for all groups (similar to `ds_groups.EncryptionMode`)
SchemeshardId	Uint64		SchemeShard identifier of the schema object to which this storage pool belongs (currently always `NULL`)
PathId	Uint64		Schema object node identifier within the specified SchemeShard to which this storage pool belongs

ds_storage_stats

Unlike other views showing physical entities, `ds_storage_stats` shows aggregated storage information.

Field	Type	Key	Value
BoxId	Uint64	0	Box identifier for which statistics are calculated
PDiskFilter	String	1	String description of filters selecting PDisk for group creation (e.g., by media type)
ErasureSpecies	String	2	Redundancy encoding mode for which statistics are collected
CurrentGroupsCreated	Uint32		Number of created groups with specified characteristics
CurrentAllocatedSize	Uint64		Total occupied space across all groups included in `CurrentGroupsCreated`
CurrentAvailableSize	Uint64		Total space available for allocation across all groups included in `CurrentGroupsCreated`
AvailableGroupsToCreate	Uint32		Number of groups with specified characteristics that can be created considering reserve needs
AvailableSizeToCreate	Uint64		Number of available bytes that would result from creating all groups from `AvailableGroupsToCreate`

Note that `AvailableGroupsToCreate` shows the maximum number of groups that can be created if no other types of groups are created. Thus, when expanding one storage pool, the `AvailableGroupsToCreate` numbers in several statistics rows may change.

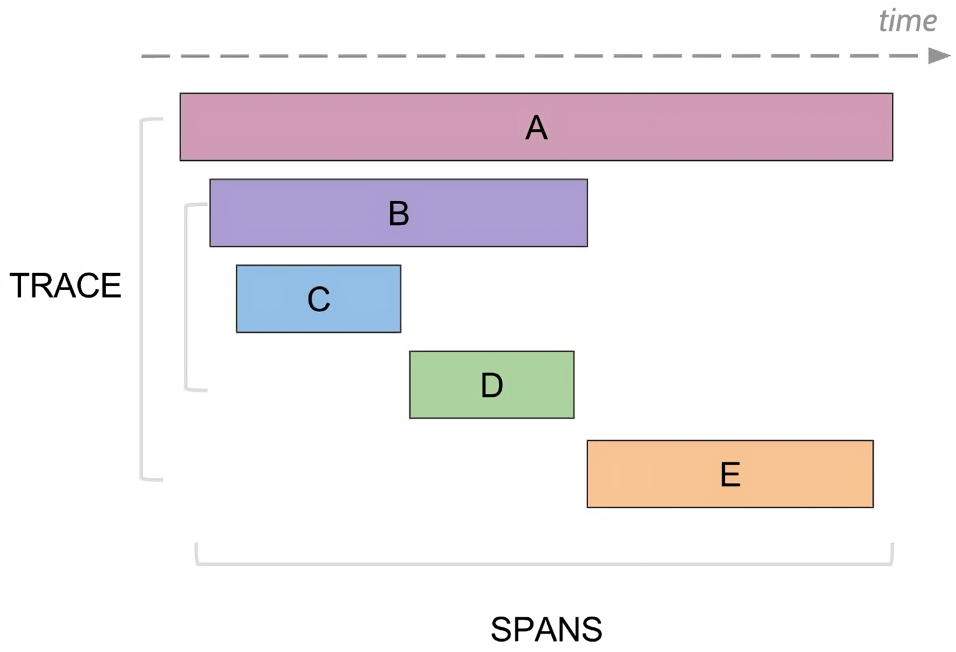
Note

Accessing system views is more of an analytical workload. Frequent access to them in large databases will significantly consume system resources. The recommended load is no more than 1-2 RPS.

Tracing in YDB

> **Note**
>
> The OpenTelemetry website describes the concept of tracing in detail in the Observability Primer article.

Tracing is a tool that allows you to view the detailed path of a request through a distributed system. A set of spans describes the path of a single request (trace). A span is a time segment usually associated with the execution time of a specific operation (e.g., writing information to disk or executing a transaction). Spans form a tree, often with the subtree of a span as its detail, but this is not always the case.



To aggregate disparate spans into traces, they are sent to a *collector*. This service aggregates and stores received spans for subsequent trace analysis. YDB does not include this service; the administrator must set it up independently. Typically, Jaeger is used as a collector.

Minimal configuration

To enable tracing in YDB, add the following section to the configuration:

```
tracing_config:
  backend:
    opentelemetry:
      collector_url: grpc://example.com:4317
      service_name: ydb
  external_throttling:
    - max_traces_per_minute: 10
```

Here, the `collector_url` field sets the URL of an OTLP-compatible span collector. More details on the backend section can be found in the relevant section.

With this configuration, no requests are sampled, and no more than ten requests per minute with an external trace-id are traced by each cluster node.

Section descriptions

Backend

Example section

```
tracing_config:
  # ...
  backend:
    opentelemetry:
      collector_url: grpc://example.com:4317
      service_name: ydb
```

Description

This section describes the span collector. Currently, the only option is `opentelemetry`. Spans are pushed from the cluster node to the collector, requiring the collector to be OTLP compatible.

In the `opentelemetry` section:

- `collector_url` — the URL of the span collector. The scheme can be either `grpc://` for an insecure connection or `grpcs://` for a TLS connection.
- `service_name` — the name under which all spans will be marked.

Both parameters are mandatory.

Uploader

Example section

```
tracing_config:
  # ...
  uploader:
    max_exported_spans_per_second: 30
    max_spans_in_batch: 100
    max_bytes_in_batch: 10485760 # 10 MiB
    max_export_requests_inflight: 3
    max_batch_accumulation_milliseconds: 5000
    span_export_timeout_seconds: 120
```

Description

The uploader is a cluster node component responsible for sending spans to the collector. To avoid overloading the span collector, the uploader will not send more than `max_exported_spans_per_second` spans per second on average.

For optimization, the uploader sends spans in batches. Each batch contains no more than `max_spans_in_batch` spans with a total serialized size of no more than `max_bytes_in_batch` bytes. Each batch accumulates for no more than `max_batch_accumulation_milliseconds` milliseconds. Batches can be sent in parallel, with the maximum number of simultaneously sent batches controlled by the `max_export_requests_inflight` parameter. If more than `span_export_timeout_seconds` seconds have passed since the uploader received the span, the uploader may delete it to send fresher spans.

Default values:

- `max_exported_spans_per_second = inf` (no limits)
- `max_spans_in_batch = 150`
- `max_bytes_in_batch = 20000000`
- `max_batch_accumulation_milliseconds = 1000`
- `span_export_timeout_seconds = inf` (no spans are deleted by the uploader)
- `max_export_requests_inflight = 1`

The `uploader` section may be completely absent, in which case each parameter will use its default value.

> **Note**
>
> The uploader is a node-local component. Therefore, the described limits apply to each node separately, not to the entire cluster.

External throttling

Example section

```
tracing_config:
  # ...
  external_throttling:
    - scope:
        database: /Root/db1
      max_traces_per_minute: 60
      max_traces_burst: 3
```

Description

YDB supports the transmission of external trace-ids to build a coherent request trace. The method for transmitting an external trace-id is described on the Passing external trace-id to YDB page. To avoid overloading the collector, YDB has a mechanism to limit the number of externally traced requests. The limits are described in this section and are a sequence of rules. Each rule contains:

- `scope` – a set of selectors for filtering the request.
- `max_traces_per_minute` – the maximum average number of requests per minute traced by this rule. A positive integer is expected.
- `max_traces_burst` – the maximum burst of externally traced requests. A non-negative integer is expected.

The only mandatory parameter is `max_traces_per_minute`.

A detailed description of these options is provided in the Rule semantics section.

The `external_throttling` section is not mandatory; if it is absent, all trace-ids in requests are **ignored** (no external traces are continued).

This section can be modified without restarting the node using the dynamic configuration mechanism.

Sampling

Example section

```
tracing_config:
  # ...
  sampling:
    - fraction: 0.01
      level: 10
      max_traces_per_minute: 5
      max_traces_burst: 2
    - scope:
```

```
        request_types:
          - KeyValue.ExecuteTransaction
          - KeyValue.Read
      fraction: 0.1
      level: 15
      max_traces_per_minute: 5
      max_traces_burst: 2
```

Description

For diagnosing system issues, looking at a sample request trace can be useful regardless of whether users trace their requests or not. For this purpose, YDB has a request sampling mechanism. For a sampled request, a random trace-id is generated. This section controls request sampling in a format similar to `external_throttling`. Each rule has two additional fields:

- `fraction` – the fraction of requests sampled by this rule. A floating-point number between 0 and 1 is expected.
- `level` — the detail level of the trace. An integer from 0 to 15 is expected. This parameter is described in more detail in the Detail levels section.

Both fields are mandatory.

The `sampling` section is not mandatory; no requests will be sampled if it is absent.

This section can be modified without restarting the node using the dynamic configuration mechanism.

Rule semantics

Selectors

Each rule includes an optional `scope` field with a set of selectors that determine which requests the rule applies to. Currently, the supported selectors are:

- `request_types`

  Accepts a list of request types. A request matches this selector if its type is in the list.

Possible values

- KeyValue.CreateVolume
- KeyValue.DropVolume
- KeyValue.AlterVolume
- KeyValue.DescribeVolume
- KeyValue.ListLocalPartitions
- KeyValue.AcquireLock
- KeyValue.ExecuteTransaction
- KeyValue.Read
- KeyValue.ReadRange
- KeyValue.ListRange
- KeyValue.GetStorageChannelStatus
- Table.CreateSession
- Table.KeepAlive
- Table.AlterTable
- Table.CreateTable
- Table.DropTable
- Table.DescribeTable
- Table.CopyTable
- Table.CopyTables
- Table.RenameTables
- Table.ExplainDataQuery
- Table.ExecuteSchemeQuery
- Table.BeginTransaction
- Table.DescribeTableOptions
- Table.DeleteSession
- Table.CommitTransaction
- Table.RollbackTransaction
- Table.PrepareDataQuery
- Table.ExecuteDataQuery
- Table.BulkUpsert
- Table.StreamExecuteScanQuery
- Table.StreamReadTable
- Table.ReadRows
- Query.ExecuteQuery
- Query.ExecuteScript
- Query.FetchScriptResults
- Query.CreateSession
- Query.DeleteSession
- Query.AttachSession
- Query.BeginTransaction
- Query.CommitTransaction
- Query.RollbackTransaction
- Discovery.WhoAmI
- Discovery.NodeRegistration
- Discovery.ListEndpoints

Note

Tracing is supported not only for the request types listed above. This list includes request types that are supported by the `request_types` selector.

Warning

Note that the QueryService API is experimental and may change in the future.

- `database`

  Filters requests to the specified database.

A request matches a rule if it matches all selectors. `scope` can be absent, which is equivalent to an empty set of selectors, and all requests will fall under this rule.

Rate limiting

The `max_traces_per_minute` and `max_traces_burst` parameters limit the number of requests. In the case of sampling, they limit the number of requests sampled by this rule. In the case of external throttling, they limit the number of external traces that enter the system.

A variation of the leaky bucket is used for rate limiting with a bucket size equal to `max_traces_burst + 1`. For example, if `max_traces_per_minute = 60` and `max_traces_burst = 0`, then with a flow of 10,000 requests per minute, one request will be traced every second. If `max_traces_burst = 20`, then with the same request flow, the first 21 requests will be traced, and then one request per second will be traced.

Warning

The limits on the number of traced requests are local to the cluster node. For example, if each cluster node has a rule specifying `max_traces_per_minute = 1`, then no more than one request per minute will be traced **from each cluster node** by this rule.

Detail levels

As with logs, diagnosing most system issues does not require the most detailed trace. Therefore, in YDB, each span has its own level described by an integer from 0 to 15 inclusive. Each rule in the `sampling` section must include the detail level of the generated trace (`level`); spans with a level less than or equal to `level` will be included in it.

The YDB architecture section describes the system's division into 5 layers:

Layer	Components
1	gRPC Proxies
2	Query Processor
3	Distributes Transactions
4	Tablet, System tablet
5	Distributed Storage

Each layer has seven detail levels:

Level	Value
`Off`	No tracing
`TopLevel`	Lowest detail, no more than two spans per request to the component
`Basic`	Spans of main component operations
`Detailed`	Highest detail applicable for diagnosing problems in production
`Diagnostic`	Detailed debugging information for developers
`Trace`	Very detailed debugging information

The table below shows the distribution of system layer detail levels by trace detail levels:

Trace detail level	gRPC Proxies	Query Processor	Distributed Transactions	Tablets	Distributed Storage
0	`TopLevel`	`Off`	`Off`	`Off`	`Off`
1	`TopLevel`	**`TopLevel`**	`Off`	`Off`	`Off`
2	`TopLevel`	`TopLevel`	**`TopLevel`**	`Off`	`Off`
3	`TopLevel`	`TopLevel`	`TopLevel`	**`TopLevel`**	`Off`
4	`TopLevel`	`TopLevel`	`TopLevel`	`TopLevel`	**`TopLevel`**
5	**`Basic`**	`TopLevel`	`TopLevel`	`TopLevel`	`TopLevel`

6	`Basic`	**`Basic`**	`TopLevel`	`TopLevel`	`TopLevel`
7	`Basic`	`Basic`	**`Basic`**	`TopLevel`	`TopLevel`
8	`Basic`	`Basic`	`Basic`	**`Basic`**	`TopLevel`
9	`Basic`	`Basic`	`Basic`	`Basic`	**`Basic`**
10	**`Detailed`**	**`Detailed`**	`Basic`	`Basic`	`Basic`
11	`Detailed`	`Detailed`	**`Detailed`**	`Basic`	`Basic`
12	`Detailed`	`Detailed`	`Detailed`	**`Detailed`**	`Basic`
13	`Detailed`	`Detailed`	`Detailed`	`Detailed`	**`Detailed`**
14	**`Diagnostic`**	**`Diagnostic`**	**`Diagnostic`**	**`Diagnostic`**	**`Diagnostic`**
15	**`Trace`**	**`Trace`**	**`Trace`**	**`Trace`**	**`Trace`**

Rules

External throttling

The semantics of each rule are as follows: it allocates a quota for the number of requests in this category. For example, if the `external_throttling` section looks like this:

```
tracing_config:
  external_throttling:
    - max_traces_per_minute: 60
    - scope:
        request_types:
          - KeyValue.ReadRange
      max_traces_per_minute: 20
```

With a sufficient flow of requests with an external trace-id, at least 60 requests per minute and at least 20 `KeyValue.ReadRange` type requests per minute will be traced. A total of up to 80 requests per minute will be traced.

The algorithm is as follows: for a request with an external trace-id, the rules that apply to this request are determined. The request consumes the quota of all rules that still have it. The request is not traced only if none of the rules have any quota left.

Sampling

The semantics of the rule for sampling are similar: with a sufficiently low flow of requests in this category, at least a `fraction` of the requests with at least `level` detail will be sampled.

The algorithm is similar: the set of rules that apply to this request is determined for a request without an external trace-id (either due to its initial absence or due to a previous decision not to trace this request). The request consumes the quota of all rules that still have it and that have randomly "decided" to sample it. It is not sampled if no rule decides to sample the request (all rules that "decided" to sample the request have no quota left). Otherwise, the detail level is determined as the maximum among the rules into whose quota the request fell.

For example, with the following `sampling` configuration:

```
tracing_config:
  sampling:
    - scope:
        database: /Root/db1
      fraction: 0.5
      level: 5
      max_traces_per_minute: 100
    - scope:
        database: /Root/db1
      fraction: 0.01
      level: 15
      max_traces_per_minute: 5
```

With a sufficiently low flow of requests to the `/Root/db1` database, the following will be sampled:

- 1% of requests with a detail level of 15
- 49.5% of requests with a detail level of 5

With a sufficiently high flow of requests to the `/Root/db1` database, the following will be sampled:

- 5 requests per minute with a detail level of 15
- between 95 and 100 requests per minute with a detail level of 5

Alerting

Alerting is a mechanism for automatic notifications that informs administrators or DevOps engineers about events that require their attention or intervention.

Use it to monitor critical database parameters in real time (such as overall system health, CPU utilization, authentication errors, and disk space usage) and to respond quickly to failures, errors, or load above acceptable thresholds.

This page describes configurable alerting rules and how they work in YDB.

Alerting in YDB with Prometheus Alerting Rules

Prometheus Alerting Rules are a set of rules written in YAML that define conditions for generating alerts. The rules use PromQL (Prometheus Query Language) and let you detect issues automatically from collected metrics. For example, you can configure a rule that fires when CPU load exceeds 70% or when disk usage is above 60%.

Before You Begin

- Set up a Prometheus server. For installation, see the Prometheus documentation.
- Get familiar with recording rules.
- Configure notifications so that alerts are delivered to your preferred channels. For configuration, see the Alertmanager documentation.

Key rule parameters:

```
- alert: YDBAuthTicketErrors  # Alert name
  expr: auth_TicketsErrors > 2  # Condition that triggers the alert
  for: 1m  # Time the condition must hold before the alert fires
  ...
```

- **Where are they stored?** Rules are usually defined in separate files (for example, `rules.yml`) and loaded into the Prometheus configuration.
- **How are they evaluated?** The Prometheus server periodically (by default every 1–2 minutes) evaluates the `expr` expressions in the rules. If the condition is true for the duration specified in `for`, an alert is generated.

Alert states:

- **Pending:** The alert condition (`expr`) is satisfied (the expression evaluates to `true`), but the `for` duration has not elapsed yet.
- **Firing:** The condition has remained true for the full `for` duration; the alert is active.
- **Resolved:** The condition no longer holds; the alert is closed.

For details on the system and rule structure, see the official Prometheus documentation.

Recommended Alerting Rules for Every Cluster

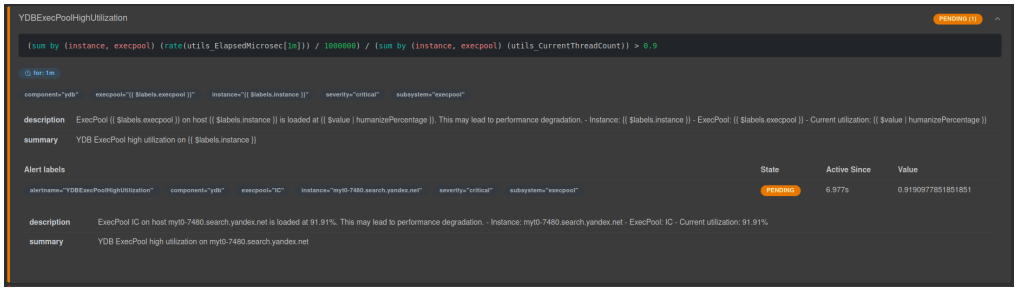
YDB ExecPool High Utilization

Description: This rule monitors resource pool utilization in YDB. It fires when utilization exceeds 90%, which can lead to performance degradation and higher query latency.
What to do: Analyze the load. The alert message includes the name of the pool where the threshold is exceeded. For diagnosing CPU-related issues, see CPU bottleneck.

> **Note**
>
> If you change the alert configuration, update the `for` parameter together with the `utils_ElapsedMicrosec` window so that their values stay consistent.

```
- alert: YDBExecPoolHighUtilization
  expr: |
    (
      sum by (instance, execpool) (
        rate(utils_ElapsedMicrosec[1m])
      ) / 1000000
    ) / (
      sum by(instance, execpool) (
        utils_CurrentThreadCount
      )
    ) > 0.9
  for: 1m
  labels:
    severity: critical
    component: ydb
    subsystem: execpool
    instance: "{{ $labels.instance }}"
    execpool: "{{ $labels.execpool }}"
  annotations:
    summary: "YDB ExecPool high utilization on {{ $labels.instance }}"
    description: |
      ExecPool {{ $labels.execpool }} on host {{ $labels.instance }}
      is loaded at {{ $value | humanizePercentage }}.
      This may lead to performance degradation.
      - Instance: {{ $labels.instance }}
      - ExecPool: {{ $labels.execpool }}
      - Current utilization: {{ $value | humanizePercentage }}
```

Example of a firing alert:

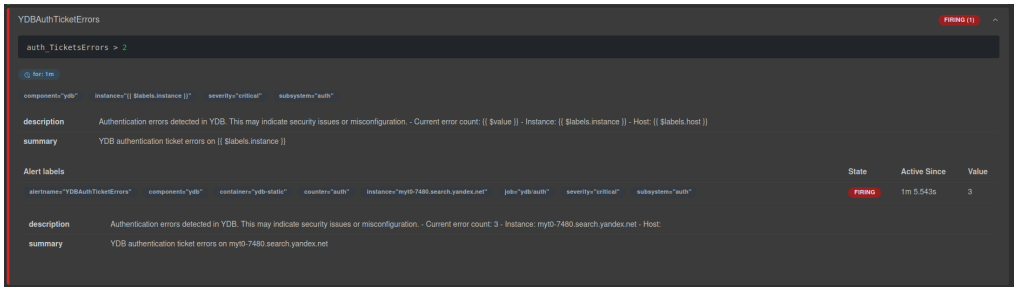


YDB Authentication Errors

Description: This rule tracks authentication errors in YDB. It fires when more than 2 errors occur within the `for` period. This may indicate incorrect credentials, security misconfiguration, or a system configuration issue.
What to do: Search Logging in YDB for authentication errors and identify the cause.

```
- alert: YDBAuthTicketErrors
  expr: auth_TicketsErrors > 2
  for: 1m
  labels:
    severity: critical
    component: ydb
    subsystem: auth
    instance: "{{ $labels.instance }}"
  annotations:
    summary: "YDB authentication ticket errors on {{ $labels.instance }}"
    description: |
      Authentication errors detected in YDB.
      This may indicate security issues or misconfiguration.
      - Current error count: {{ $value }}
      - Instance: {{ $labels.instance }}
      - Host: {{ $labels.host }}
```

Example of a firing alert:



YDB Storage Usage

Description: This rule monitors disk space usage. It fires when storage utilization reaches critical levels. Configure two thresholds: a warning at 80% usage and a critical alert at 90%.
What to do: Determine why the database has grown beyond the expected size. Remove unneeded data (for example, old logs) or increase the database size limit if appropriate. For diagnosing disk space issues, see Disk space.

Warning Storage Usage (80%)

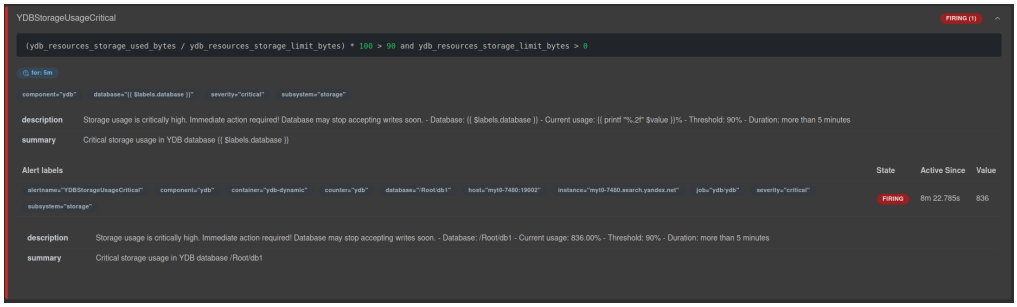
```
- alert: YDBStorageUsageWarning
  expr: |
    (ydb_resources_storage_used_bytes / ydb_resources_storage_limit_bytes) * 100 > 80
    and ydb_resources_storage_limit_bytes > 0
  for: 5m
  labels:
    severity: warning
    component: ydb
    subsystem: storage
    database: "{{ $labels.database }}"
  annotations:
    summary: "High storage usage in YDB database {{ $labels.database }}"
    description: |
      Storage usage is above warning threshold.
      Consider cleaning up old data or increasing storage capacity.
      - Database: {{ $labels.database }}
      - Current usage: {{ printf "%.2f" $value }}%
      - Threshold: 80%
      - Duration: more than 5 minutes
```

Critical Storage Usage (90%)

```
- alert: YDBStorageUsageCritical
  expr: |
    (ydb_resources_storage_used_bytes / ydb_resources_storage_limit_bytes) * 100 > 90
    and ydb_resources_storage_limit_bytes > 0
  for: 5m
  labels:
    severity: critical
```

```
    component: ydb
    subsystem: storage
    database: "{{ $labels.database }}"
  annotations:
    summary: "Critical storage usage in YDB database {{ $labels.database }}"
    description: |
      Storage usage is critically high. Immediate action required!
      Database may stop accepting writes soon.
      - Database: {{ $labels.database }}
      - Current usage: {{ printf "%.2f" $value }}%
      - Threshold: 90%
      - Duration: more than 5 minutes
```

Example of a firing alert:



Full Configuration File

A full configuration file with all of the rules above is available on GitHub.

Configuration Recommendations

- **Thresholds:** Adjust the thresholds in the `expr` parameter to match your SLAs and load profile.
- **Pending duration:** Increase the `for` parameter for less critical alerts to reduce false positives.

Diagnostics of an isolated cluster

If a cluster is hidden behind network barriers, direct monitoring of its state is impossible. In such cases, to identify problems in the cluster's internal environment, you can use the mechanism of saving its state to a file.

Collecting diagnostic information

To collect diagnostic information, use the command:

```
ydb [global options...] admin cluster diagnostics collect \
  --duration 200 \
  --period 15 \
  --output diagnostics.tar
```

Command parameters

- `--duration` — the number of seconds during which cluster information will be collected.
- `--period` — the interval in seconds between metric collections.
- `--output` — the path and file name in `.tar` format, where all collected cluster information will be written.
- `--no-sanitize` — allows you to get a full report by disabling the sanitization of user table names, columns, and queries (which are stripped by default).

The command writes the internal state of cluster nodes to the specified file over a given period. Every `period` seconds, metrics are collected from each cluster node and written to the file.

Archive contents

The resulting file is an archive and contains:

- Metrics collected from cluster nodes.
- Cluster information.

All data is written in JSON format in a human-readable form.

Cluster information

Cluster information includes:

- Health Check report.
- Queries to the cluster partition tables `/Root/.sys` and their responses.
- Current configuration of cluster subsystems.
- Current state of cluster nodes.

Backup and recovery

YDB is initially designed for data safety in case of hardware failures: redundancy options are available for different numbers of availability zones, racks, hosts, disks, and other components (see cluster operating modes). This reduces the risk of data loss due to hardware failures.

However, data can be lost or corrupted **logically**: errors or malicious actions can lead to mass deletion/corruption of data by operations legitimate for the DBMS. In such cases, cluster fault tolerance does not replace a **separate copy** of data outside the cluster.

Backup is used to protect against such scenarios and allows restoring data from a backup. It is recommended to store copies on a separate medium or in cloud object storage (for example, via export to files or export to S3).

YDB provides several solutions for performing backup and recovery. For conceptual information and comparison of backup methods, see backup concepts.

- Backup to files and recovery using the YDB CLI.
- Backup to S3-compatible storage and recovery using the YDB CLI.
- Backup of system tablets to the local file system of cluster hosts.

YDB CLI

Files

The following commands are used for backup to files:

- `ydb admin cluster dump` — for backing up cluster metadata
- `ydb admin database dump` — for backing up the database
- `ydb tools dump` — for backing up individual schema objects or directories

Learn more about these commands in Exporting data to the file system.

The following commands are used for restoration from a file backup:

- `ydb admin cluster restore` — for restoring cluster metadata from a backup
- `ydb admin database restore` — for restoring the database from a backup
- `ydb tools restore` — for restoring individual schema objects or directories from a backup

Learn more about these commands in Importing data from the file system.

S3-compatible storage

To perform backup to an S3-compatible storage (for example, AWS S3), the `ydb export s3` command is used. Follow the link to the YDB CLI reference for information about this command.

To perform restoration from a backup created in an S3-compatible storage, the `ydb import s3` command is used. Follow the link to the YDB CLI reference for information about this command.

NFS

To perform backup to NFS on hosts where YDB is running, the `ydb export nfs` command is used. Follow the link to the YDB CLI reference for information about this command.

To perform restoration from a backup created in NFS on hosts where YDB is running, the `ydb import nfs` command is used. Follow the link to the YDB CLI reference for information about this command.

For more details on configuring NFS for backup and recovery, see the recipe Backup and recovery via NFS.

> **Note**
>
> The speed of backup and restore operations to/from S3-compatible storage or NFS is tuned to minimize the impact on user load. To control the operation speed, configure limits for the corresponding queue of the resource broker.

> **Note**
>
> When performing an export, a directory named `export_*` is created in the root directory of the database, where `*` is the numeric part of the export identifier. This directory contains tables that hold a consistent snapshot of the exported data at the time the export started. After the backup operation completes successfully, the `export_*` directory along with its contents is deleted.

Backup collections

Backup collections provide incremental backup and point-in-time recovery for production workloads. For conceptual information and architecture details, see Backup collections.

Backup collections are recommended for production environments with a regular backup schedule and large datasets where incremental changes are significantly smaller than the total volume. For simpler scenarios (one-time migrations, development environments, small databases), consider using export/import or dump/restore.

For step-by-step instructions on configuring and using backup collections, see:

- Backup collections — architecture, concepts, and limitations
- Recipes for backup collections — typical scenarios and examples

Backup of system tablets

The system tablet backup mechanism provides incremental copying of cluster metadata to the local file system of the cluster hosts.

For conceptual information and how it works, see Backup concepts.

For step-by-step instructions on enabling and restoring, see Recipes for system tablet backup.

YDB for Application Developers / Software Engineers

This section of YDB documentation covers everything you need to know to develop applications interacting with YDB.

Main resources:

- Getting started with YDB as an Application Developer / Software Engineer
- Example applications working with YDB
- Generate code from YQL (sqlc-ydb)
- YQL Tutorial - Overview
- Choosing a primary key for:
  - Row-oriented tables
  - Column-oriented tables
- Secondary indexes
- Uploading data to YDB
- Paginated output
- Using timeouts
- System database views
- Change Data Capture
- Custom attributes in tables
- Reference:
  - YQL - Overview
  - Reference for YDB SDK
  - YDB CLI
  - Kafka API

If you're interested in developing YDB core or satellite projects, refer to the documentation for contributors.

Getting started with YDB as an Application Developer / Software Engineer

First of all, you'll need to obtain access to a YDB cluster. Follow the quickstart instructions to get a basic local instance. Later on, you can work with your DevOps team to build a production-ready cluster or leverage one of the cloud service providers that offer a managed YDB service.

The second step is designing a data schema for an application you will build from scratch or adapt the schema of an existing application if you're migrating from another database management system.

In parallel with designing the schema, you need to set up your development environment for interaction with YDB. There are a few main aspects to it, explored below.

Choosing API

Choose the YDB API you want to use; there are several options:

- The recommended way for mainstream programming languages is using a YDB SDK. They provide high-level APIs and implement best practices on working with YDB. YDB SDKs are available for several popular languages and strive for feature parity, but not all are feature-complete. Refer to the SDK feature comparison table to check if the SDK for the programming language you had in mind will fit your needs or to choose a programming language with better feature coverage if you're flexible.
- If you are interested in YDB topics feature, it is worth noting that they also provide Kafka-compatible API. Follow that link if this use case is relevant.
- As a last resort, YDB's native API is based on the gRPC protocol, which has an ecosystem around it, including code generation of clients. YDB's gRPC specs are hosted on GitHub and you could leverage them in your application. The generated clients are low-level and will require extra work to handle aspects like retries and timeouts properly, so go this route only if other options above aren't possible and you know what you're doing.

Install prerequisites

Choose the specific programming language you'll be using. Install the respective YDB SDK.

Additionally, you'd want to set up at least one of the available ways to run ad-hoc queries for debugging purposes. Choose at least one according to your preferences:

- YDB CLI
- YDB UI
- Any SQL IDE that supports JDBC

Start coding

For YDB SDK route

- Go through YQL tutorial to get familiar with YDB's SQL dialect.
- Explore example applications to see how working with SDK's looks like.
- Check out SDK recipies for typical SDK use cases, which you can refer to later.
- Learn how to handle YDB SDK errors.
- Leverage your IDE capabilities to navigate the SDK code.

Testing

To write tests on applications working with YDB:

- For functional tests, you can mock YDB's responses using a suitable testing framework for your chosen programming language.
- For integrational tests, you can launch a single-node YDB instance in your CI/CD environment with either a Docker image or executable similarly to how it is done in the Quickstart article. It is recommended to test against a few versions of YDB: the one you have in production to check for issues you can encounter when updating your application and the newer versions to identify the potential issues of upgrading YDB early on.
- For performance tests, you'd want to use a cluster deployed according to instructions for production use as single-node won't yield realistic results. You can run ydb workload if you want to see how your YDB cluster performs in generic scenarios even before writing any application code. Then you can use the source code of the library behind this tool as an example of how to write your own performance tests with YDB. It'd be great if you could contribute an anonymized version of your workload to upstream so it can be included in performance testing of YDB itself.

What's next

The above should be enough to start developing applications that interact with YDB. Along the way use YQL and YDB SDK reference documentation and other resources in this documentation section.

In case of any issues, feel free to discuss them in YDB Discord.

Streaming queries

Practical aspects of working with streaming queries:

- Local and external topics — accessing topics of the current and another database in YQL queries
- Typical patterns — minimal examples for a quick start
- Writing to tables — how streaming queries allow writing data to YDB tables in real time.
- Data enrichment — ways to enrich data in a stream using reference data.
- Data formats for reading/writing topics — supported data formats when working with topics, examples of their use.
- Data delivery guarantees — level of guarantees, observed anomalies in window aggregation, and recommendations.
- Checkpoints — a mechanism for saving the state of stream processing to ensure fault tolerance and recovery capability.
- Watermarks — a mechanism for tracking time progress in a data stream.

See also

- Recipes for working with streaming queries
- Streaming queries overview

Typical streaming query patterns

This section contains minimal examples of streaming queries for the most common scenarios. First, a basic pattern for reading data from a topic is described, followed by variants of full data processing: data processing and writing results to a topic in JSON format, to a topic as a string, and to a table. Each example can be used as a starting point for your own tasks.

The examples below use local and external topics. Notation:

- `ext_source` — a pre-created `external data source`
- `input_topic` — the topic from which data is read
- `output_topic` — the topic where results are written
- `output_table` — the YDB table where results are written

Reading data from a topic

Reading structured messages is done using `SELECT ... FROM ... WITH (FORMAT, SCHEMA)`. The `WITH` block specifies the input data format and schema — which fields are expected in each message and their types.

The following fragment is used inside CREATE STREAMING QUERY in the `DO BEGIN ... END DO` block:

```
SELECT
    Id,
    Name
FROM
    topic_name  -- local topic; for external: ext_source.topic_name
WITH (
    FORMAT = json_each_row,
    SCHEMA = (
        Id Uint64 NOT NULL,
        Name Utf8 NOT NULL
    )
);
```

For more details on formats: Data formats for reading/writing from topics.

This pattern is used in all subsequent examples.

Writing to a topic (JSON)

The query reads events from the input topic, forms a JSON object from individual fields, and writes the result to the output topic. The `AsStruct` function creates a structure from the specified fields, `Yson::From` converts it to Yson, `Yson::SerializeJson` serializes it to a JSON string, and `ToBytes` converts it to the `String` type, which is required for writing to the topic.

```
CREATE STREAMING QUERY write_json_example AS
DO BEGIN

INSERT INTO ext_source.output_topic -- or local topic output_topic
SELECT
    -- Building JSON from individual fields
    ToBytes(Unwrap(Yson::SerializeJson(Yson::From(
        AsStruct(Id AS id, Name AS name)
    ))))
FROM
    input_topic -- or external topic ext_source.input_topic
WITH (
    FORMAT = json_each_row,  -- Input data format
    SCHEMA = (               -- Input data schema
        Id Uint64 NOT NULL,
        Name Utf8 NOT NULL
    )
);

END DO
```

For more details on functions:

- AsStruct
- Yson::From
- Yson::SerializeJson
- Unwrap
- ToBytes.

Writing to a topic (string)

The query reads events from the input topic and writes one field as a string to the output topic. To write strings to a topic, `SELECT` must return a single column of type `String` or `Utf8`.

```
CREATE STREAMING QUERY write_utf8_example AS
DO BEGIN

INSERT INTO output_topic -- or external topic ext_source.output_topic
SELECT
    Name
FROM
    ext_source.input_topic -- or local topic input_topic
WITH (
    FORMAT = json_each_row,  -- Input data format
    SCHEMA = (               -- Input data schema
```

```
        Id Uint64 NOT NULL,
        Name Utf8 NOT NULL
    )
);

END DO
```

For more details on write formats: Formats for writing data.

Writing to a table

The query reads events from the topic and writes them to the `output_table` table. The table must be created in advance with a schema matching the selected columns.

> **Warning**
>
> Writing to tables in streaming queries is supported **only in UPSERT mode**. The `INSERT INTO` operation is not supported, because during reprocessing of events (the at-least-once guarantee) it would lead to duplicate rows. With `UPSERT`, if a row with such a primary key already exists, it will be updated; otherwise, a new row will be inserted, and `INSERT INTO` will fail with an error.

```
CREATE STREAMING QUERY write_table_example AS
DO BEGIN

-- Writing to table (only UPSERT, INSERT not supported)
UPSERT INTO output_table
SELECT
    Id,
    Name
FROM
    ext_source.input_topic -- or local topic input_topic
WITH (
    FORMAT = json_each_row,  -- Input data format
    SCHEMA = (               -- Input data schema
        Id Uint64 NOT NULL,
        Name Utf8 NOT NULL
    )
);

END DO
```

For more details: Writing to tables.

See also

- Local and external topics
- CREATE STREAMING QUERY
- Quick start: reading and writing to topics

Writing to tables

Writing to tables lets you save the results of a streaming query for later analysis with regular SQL queries. For example, you can aggregate events from a stream and save the results to a table.

For writing, UPSERT INTO is used — inserting a new row or updating an existing one by primary key. The UPSERT operation is idempotent by primary key: rewriting the same row results in an update, not duplication. This is important because streaming queries provide the at-least-once guarantee — when recovering from a checkpoint, some events may be processed again.

Alert

Not supported:

- The INSERT INTO command — use UPSERT INTO. INSERT INTO is not used because reprocessing events (at-least-once guarantee) would lead to duplicate rows.
- Writing to YDB tables located in external databases. In the current version, writing is only possible to local tables.

Example

The query reads events from a topic and writes them to the `output_table` table. The `Ts` field is converted from a string to the `Timestamp` type using `CAST`, and Unwrap removes the optionality of the result.

```
CREATE STREAMING QUERY query_with_table_write AS
DO BEGIN

-- Reading from a topic and writing to a table
UPSERT INTO
    output_table
SELECT
    -- Converting a string to Timestamp
    Unwrap(CAST(Ts AS Timestamp)) AS Ts,
    Country,
    Count
FROM
    -- Reading events from a topic
    input_topic
WITH (
    -- Data format in a topic
    FORMAT = json_each_row,
    -- Data schema
    SCHEMA = (
        Ts String NOT NULL,
        Count Uint64 NOT NULL,
        Country Utf8 NOT NULL
    )
);

END DO
```

Limitations

In a single streaming query:

- You cannot use the same table for stream enrichment using `JOIN` and for writing the query result.
- You cannot write to the same table multiple times.

Data enrichment

Data enrichment is adding additional information from a reference to events in a stream. For example, an event contains only an identifier, and the reference allows adding a name or other attributes to it. You can use data from a local table or from an S3 object storage as a reference.

In streaming queries, the reference is connected using the `JOIN` construct. The stream must be on the left, the reference on the right.

> **Warning**
>
> The reference is fully loaded into memory when the query starts. If the data in the reference has changed, to get the current version of the reference, you need to restart the query — delete it using DROP STREAMING QUERY and recreate it using CREATE STREAMING QUERY.

Data enrichment from local and external topics is possible.

In the examples below:

- `ext_source` — a pre-created external data source for topics in another YDB database
- `input_topic` and `output_topic` — topics in the current or external YDB database

Streaming queries for data enrichment

The queries in the examples below read events from the input topic, attach the service name from the reference by `ServiceId` to each event, and write the result to the output topic.

For more details on the functions used in the queries:

- TableRow
- Yson::From
- Yson::SerializeJson
- Unwrap
- ToBytes.

Enrichment from a local table

In this example, the reference is stored in the table `services_dict` in the current database.

Create a streaming query that performs enrichment:

```
CREATE STREAMING QUERY query_with_table_join AS
DO BEGIN

-- Reading events from input topic
$topic_data = SELECT
    *
FROM
    ext_source.input_topic -- or local topic input_topic
WITH (
    FORMAT = json_each_row,
    SCHEMA = (
        Time String NOT NULL,
        ServiceId Uint32 NOT NULL,
        Message String NOT NULL
    )
);

-- Joining reference to stream by ServiceId
$joined_data = SELECT
    s.Name AS Name,
    t.*
FROM
    $topic_data AS t
LEFT JOIN
    services_dict AS s
ON
    t.ServiceId = s.ServiceId;

-- Writing to the output topic (JSON)
INSERT INTO
    output_topic -- or external topic ext_source.output_topic
SELECT
    ToBytes(Unwrap(Yson::SerializeJson(Yson::From(TableRow()))))
FROM
    $joined_data;

END DO
```

Enrichment from S3

The reference is stored in S3 and connected via an external data source.

Create an additional external data source to read the reference from S3:

```
-- S3 data source for reading reference
CREATE EXTERNAL DATA SOURCE s3_source WITH (
    SOURCE_TYPE = "ObjectStorage",
    LOCATION = "<s3_endpoint>",
```

```
    AUTH_METHOD = "NONE"
);
```

Where:

- `<s3_endpoint>` — the S3 storage URL, for example `https://storage.yandexcloud.net/<bucket>/` for Yandex Cloud.

Create a streaming query that performs enrichment:

```
CREATE STREAMING QUERY query_with_join AS
DO BEGIN

-- Reading events from input topic
$topic_data = SELECT
    *
FROM
    input_topic -- or external topic ext_source.input_topic
WITH (
    FORMAT = json_each_row,
    SCHEMA = (
        Time String NOT NULL,
        ServiceId Uint32 NOT NULL,
        Message String NOT NULL
    )
);

-- Reading service reference from S3
$s3_data = SELECT
    *
FROM
    s3_source.`file.csv`
WITH (
    FORMAT = csv_with_names,
    SCHEMA = (
        ServiceId Uint32,
        Name Utf8
    )
);

-- Joining reference to stream by ServiceId
$joined_data = SELECT
    s.Name AS Name,
    t.*
FROM
    $topic_data AS t
LEFT JOIN
    $s3_data AS s
ON
    t.ServiceId = s.ServiceId;

-- Write result to output topic in JSON format
INSERT INTO
    ext_source.output_topic -- or local topic output_topic
SELECT
    ToBytes(Unwrap(Yson::SerializeJson(Yson::From(TableRow()))))
FROM
    $joined_data;

END DO
```

For more details on data formats (`json_each_row`, `csv_with_names`, etc.): Data formats for reading/writing from topics.

Data formats for reading/writing from topics

This section describes the data formats of streaming queries supported in YDB when reading from topics, and the list of supported YQL types for each data format.

Supported data formats

YDB supports built-in data formats (predefined by default) and custom formats (configured by the user for specific tasks).

Below is the list of data formats built into YDB:

- `csv_with_names`
- `tsv_with_names`
- `json_list`
- `json_each_row`
- `json_as_string`
- `parquet`
- `raw`

Examples of parsing data in custom formats.

In the examples on this page, `input_topic` and `output_topic` are topics in the current database, and `output_table` is a YDB table. For topics in another database, see local and external topics.

Formats for writing data

When writing to a topic, `SELECT` must return a single column of type `String`, `Utf8`, `Json`, or `Yson`. The column cannot be `Optional`.

Writing a single column:

```
CREATE STREAMING QUERY write_string_example AS
DO BEGIN

    INSERT INTO output_topic
    SELECT
        CAST(Data AS String)
    FROM
        input_topic
    WITH (
        FORMAT = raw,
        SCHEMA = (
            Data String
        )
    );

END DO
```

To write multiple columns, serialize them into JSON:

```
CREATE STREAMING QUERY write_json_example AS
DO BEGIN

    INSERT INTO output_topic
    SELECT
        ToBytes(Unwrap(Yson::SerializeJson(Yson::From(TableRow()))))
    FROM
        input_topic
    WITH (
        FORMAT = json_each_row,
        SCHEMA = (
            Id Uint64 NOT NULL,
            Name Utf8 NOT NULL
        )
    );

END DO
```

For more details on functions: TableRow, Yson::From, Yson::SerializeJson, Unwrap, ToBytes.

If a single streaming query writes to multiple topics, all output topics must be different.

Formats for reading data

csv_with_names format

This format is based on the CSV format. Data is placed in columns separated by commas, with column names in the first row.

Example data (in a single message):

```
Year,Manufacturer,Model,Price
1997,Man_1,Model_1,3000.00
1999,Man_2,Model_2,4900.00
```

Example query:

```
CREATE STREAMING QUERY csv_example AS
DO BEGIN
```

```
    UPSERT INTO output_table
    SELECT
        *
    FROM
        input_topic
    WITH (
        FORMAT = csv_with_names,
        SCHEMA = (
            Year Int32 NOT NULL,
            Manufacturer String NOT NULL,
            Model String NOT NULL,
            Price Double NOT NULL
        )
    );

END DO
```

tsv_with_names format

This format is based on the TSV format. Data is placed in columns separated by tab characters (code `0x9`), with column names in the first row.

Example data (in a single message):

```
Year    Manufacturer    Model   Price
1997    Man_1   Model_1    3000.00
1999    Man_2   Model_2    4900.00
```

Example query:

```
CREATE STREAMING QUERY tsv_example AS
DO BEGIN

    UPSERT INTO output_table
    SELECT
        *
    FROM
        input_topic
    WITH (
        FORMAT = tsv_with_names,
        SCHEMA = (
            Year Int32 NOT NULL,
            Manufacturer String NOT NULL,
            Model String NOT NULL,
            Price Double NOT NULL
        )
    );

END DO
```

json_list format

This format is based on the JSON representation of data. In this format, each message must be a JSON array of objects.

Example of valid data (as a list of JSON objects):

```
[
    { "Year": 1997, "Manufacturer": "Man_1", "Model": "Model_1", "Price": 3000.0 },
    { "Year": 1999, "Manufacturer": "Man_2", "Model": "Model_2", "Price": 4900.00 }
]
```

Example of invalid data (each message contains a separate JSON object, but these objects are not combined into a list):

```
{ "Year": 1997, "Manufacturer": "Man_1", "Model": "Model_1", "Price": 3000.0 }
{ "Year": 1999, "Manufacturer": "Man_2", "Model": "Model_2", "Price": 4900.00 }
```

Example query:

```
CREATE STREAMING QUERY json_list_example AS
DO BEGIN

    UPSERT INTO output_table
    SELECT
        *
    FROM
        input_topic
    WITH (
        FORMAT = json_list,
        SCHEMA = (
            Year Int32 NOT NULL,
            Manufacturer String NOT NULL,
            Model String NOT NULL,
            Price Double NOT NULL
        )
    );
```

```
END DO
```

json_each_row format

This format is based on the JSON representation of data. In this format, each message must be a JSON object. This format is used when transferring data through streaming systems, such as Apache Kafka or YDB Topics.
Multiple separate JSONs in a single message are not supported; a JSON list is also not supported.

Example of valid data (in a single message):

```
{ "Year": 1997, "Manufacturer": "Man_1", "Model": "Model_1", "Price": 3000.0 }
```

Example of invalid data:

```
{ "Year": 1997, "Manufacturer": "Man_1", "Model": "Model_1", "Price": 3000.0 }
{ "Year": 1999, "Manufacturer": "Man_2", "Model": "Model_2", "Price": 4900.00 }
```

Example query:

```
CREATE STREAMING QUERY json_each_row_example AS
DO BEGIN

    UPSERT INTO output_table
    SELECT
        *
    FROM
        input_topic
    WITH (
        FORMAT = json_each_row,
        SCHEMA = (
            Year Int32 NOT NULL,
            Manufacturer Utf8 NOT NULL,
            Model Utf8 NOT NULL,
            Price Double NOT NULL
        )
    );

END DO
```

json_as_string format

This format is based on the JSON representation of data.

In this format, each message must contain:

- An object in valid JSON representation on each separate line of the file.
- Objects in valid JSON representation combined into a list.

The `json_as_string` format does not split the input JSON document into fields, but represents each message as a single JSON object (or a single line). This format is convenient when the list of fields is not the same across all rows and can vary.

Example of valid data:

```
{ "Year": 1997, "Attrs": { "Manufacturer": "Man_1", "Model": "Model_1" }, "Price": 3000.0 }
{ "Year": 1999, "Attrs": { "Manufacturer": "Man_2", "Model": "Model_2" }, "Price": 4900.00 }
```

In this format, the schema of the read data must consist of only one column with one of the allowed data types; see below for details.

Example query:

```
CREATE STREAMING QUERY json_as_string_example AS
DO BEGIN

    INSERT INTO output_topic
    SELECT
        *
    FROM
        input_topic
    WITH (
        FORMAT = json_as_string,
        SCHEMA = (
            Data Json
        )
    );

END DO
```

parquet format

This format allows reading the contents of messages in Apache Parquet format.

Example query:

```
CREATE STREAMING QUERY parquet_example AS
DO BEGIN
```

```
    UPSERT INTO output_table
    SELECT
        *
    FROM
        input_topic
    WITH (
        FORMAT = parquet,
        SCHEMA = (
            Year Int32 NOT NULL,
            Manufacturer Utf8 NOT NULL,
            Model Utf8 NOT NULL,
            Price Double NOT NULL
        )
    );

END DO
```

raw format

This format allows reading the contents of messages as-is, in "raw" form. Data read in this way can be processed using YQL. Default schema: `SCHEMA(Data String)`.

Example query:

```
CREATE STREAMING QUERY raw_example AS
DO BEGIN

    INSERT INTO output_topic
    SELECT
        *
    FROM
        input_topic
    WITH (
        FORMAT = raw,
        SCHEMA = (
            Data String
        )
    );

END DO
```

Supported data types

Table of all supported types in the query schema:

Type	csv_with_names	tsv_with_names	json_list	json_each_row	json_as_string	parquet	raw
`Int8`, `Int16`, `Int32`, `Int64`, `Uint8`, `Uint16`, `Uint32`, `Uint64`, `Float`, `Double`	✓	✓	✓	✓	✓	✓	
`Bool`	✓	✓	✓	✓		✓	
`DyNumber`							
`String`, `Utf8`	✓	✓	✓	✓	✓	✓	✓
`Json`					✓		✓
`JsonDocument`							
`Yson`							
`Uuid`				✓			
`Date`, `Datetime`, `Timestamp`, `TzDate`, `TzDateTime`, `TzTimestamp`							
`Interval`							
`Date32`, `Datetime64`, `Timestamp64`, `Interval64`, `TzDate32`, `TzDateTime64`, `TzTimestamp64`							
`Optional<T>`	✓	✓	✓	✓	✓	✓	✓

Examples of parsing data in custom formats

Parsing JSON with built-in functions

Example data:

```
{"key": 1997, "value": "42"}
```

Example query:

```
CREATE STREAMING QUERY json_builtins_example AS
DO BEGIN

    INSERT INTO output_topic
    SELECT
        JSON_VALUE(Data, "$.key") AS Key,
        JSON_VALUE(Data, "$.value") AS Value
    FROM
        input_topic
    WITH (
        FORMAT = raw,
        SCHEMA = (
            Data Json
        )
    );

END DO
```

For more details about functions: JSON_VALUE.

Parsing JSON with the Yson library

Example data (Change Data Capture format):

```
{"update":{"volume":10,"product":"bWlsaw=="},"key":[6],"ts":[1765192622420,1844674407370955161 5]}
```

Example query:

```
CREATE STREAMING QUERY json_yson_example AS
DO BEGIN

    $input = SELECT
        Yson::ConvertTo(
            Data, Struct<
                update: Struct<volume: Uint64>,
                key: List<Uint64>,
                ts: List<Uint64>
            >
        )
    FROM
        input_topic
    WITH (
        FORMAT = json_as_string,
        SCHEMA = (
            Data Json
        )
    );

    INSERT INTO output_topic
    SELECT
        ts[0] AS Ts,
        update.volume AS Volume
    FROM
        $input
    FLATTEN COLUMNS;

END DO
```

For more information about functions:

- Yson::ConvertTo
- FLATTEN COLUMNS.

Parsing DSV (TSKV)

Example data:

```
name=Elena  uid=95792365232151958
name=Denis  uid=78086244452810046
name=Mikhail    uid=70609792906901286
```

Example query:

```
CREATE STREAMING QUERY dsv_example AS
DO BEGIN
```

```
    $input = SELECT
        Dsv::Parse(Data, "\t") AS Data
    FROM
        input_topic
    FLATTEN LIST BY (
        String::SplitToList(Data, "\n", TRUE AS SkipEmpty) AS Data
    );

    INSERT INTO output_topic
    SELECT
        DictLookup(Data, "name") AS Name,
        DictLookup(Data, "uid") AS Uid
    FROM
        $input;

END DO
```

For more information about functions:

- String::SplitToList
- DictLookup
- FLATTEN LIST BY.

Data delivery guarantees

Delivery guarantees determine how many times each event from the input topic will be processed by a streaming query. Understanding the system's guarantees is critical when designing data processing pipelines.

> **Note**
>
> We are constantly working on developing streaming processing mechanisms. In future versions, the guarantees provided will be improved.

Data processing guarantees (dataplane):

- at-least-once — for all query types, each event is processed at least once.

Anomalies when modifying queries (control plane):

- Event loss when recreating a query — when recreating a query via DROP + CREATE, some events that arrived between deletion and creation will be skipped.
- Partial first aggregation window — when a query starts, the first aggregation window contains incomplete data.

Checkpoints and recovery

YDB periodically saves a checkpoint — a snapshot of the query state containing:

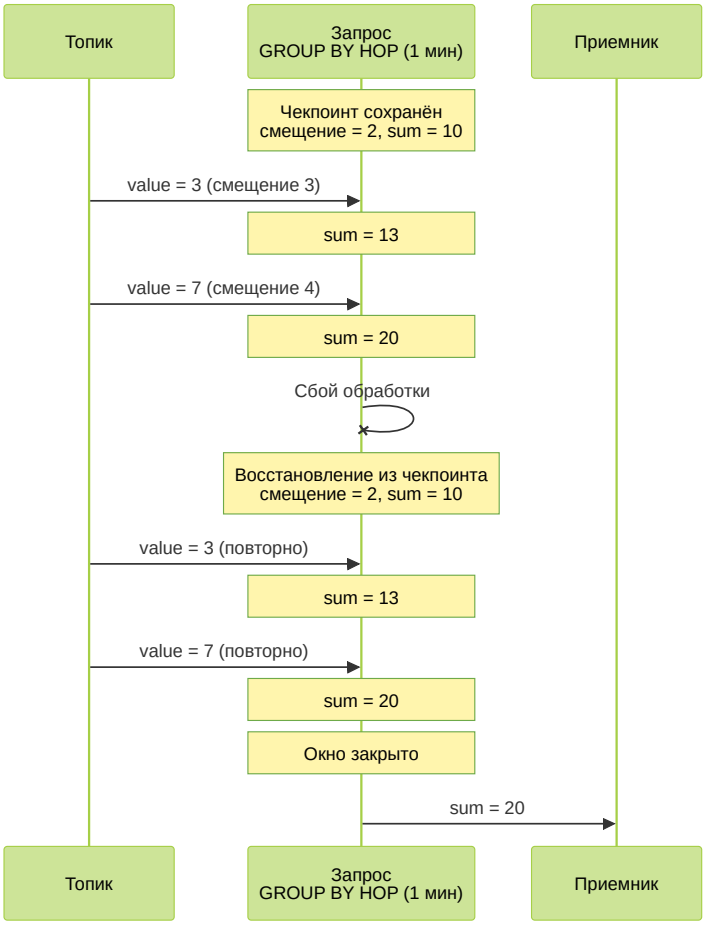
- offsets in input topics — positions up to which events have been read and processed;
- aggregation states — intermediate results of operations, for example accumulated values in GROUP BY HOP.

YDB stores read offsets in its own checkpoints, rather than relying on the offsets of a consumer in an external system.

During recovery, the query rolls back to the last checkpoint: it resumes reading from the saved offsets and restores the aggregation states. Events that arrived between the checkpoint and the failure will be reprocessed. For more details on the checkpoint mechanism, see the Checkpoints section.

Data processing guarantees (dataplane) — at-least-once

If a failure occurs during stream processing (compute node restart, network outage, timeout), YDB automatically restores the query from the last checkpoint. The at-least-once guarantee is provided for all types of streaming queries — each event will be processed at least once. The query resumes reading from the saved offset and resends the processing results. This applies to all types of queries: queries without aggregation (filtering, enrichment, transformation) and queries with window aggregation.



When writing the result to a table via UPSERT, reprocessing does not lead to duplication: UPSERT updates the existing row by primary key. Data is not lost, and duplicates do not accumulate.

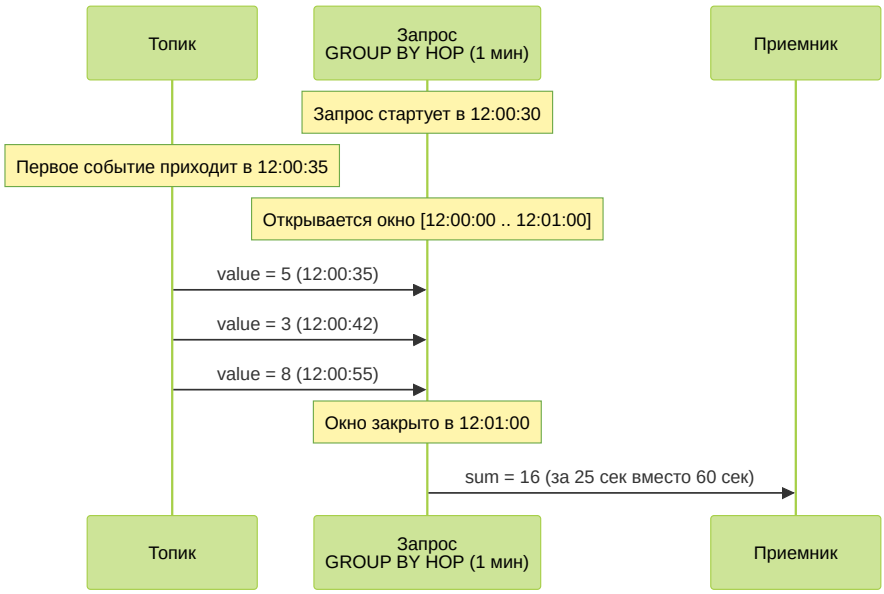
When writing the result to an output topic, reprocessing leads to duplicates: the same events will be written to the topic more than once. The consumer of the output topic must account for this and, if necessary, perform deduplication on its own.

Guarantees when modifying a query (control plane)

Currently, changing the query text without stopping it is not supported. To update a query, a combination of DROP and CREATE commands is used; in this case, the `at-least-once` guarantee is not met: some events may be skipped. The scenarios where this occurs are described below.

Partial results of the first window when starting a query

Time windows (GROUP BY HOP) calculate their boundaries based on absolute (wall-clock) time. Window boundaries are aligned to multiples of intervals from the start of the epoch: for example, with a 1-minute window, boundaries always occur at 12:00:00, 12:01:00, 12:02:00, etc., regardless of when the query was started. If the query starts at 12:00:30, it falls into the already running window [12:00:00 .. 12:01:00], but data only starts arriving at 12:00:30. As a result, the aggregate of the first window is computed from 30 seconds of data instead of a full minute.

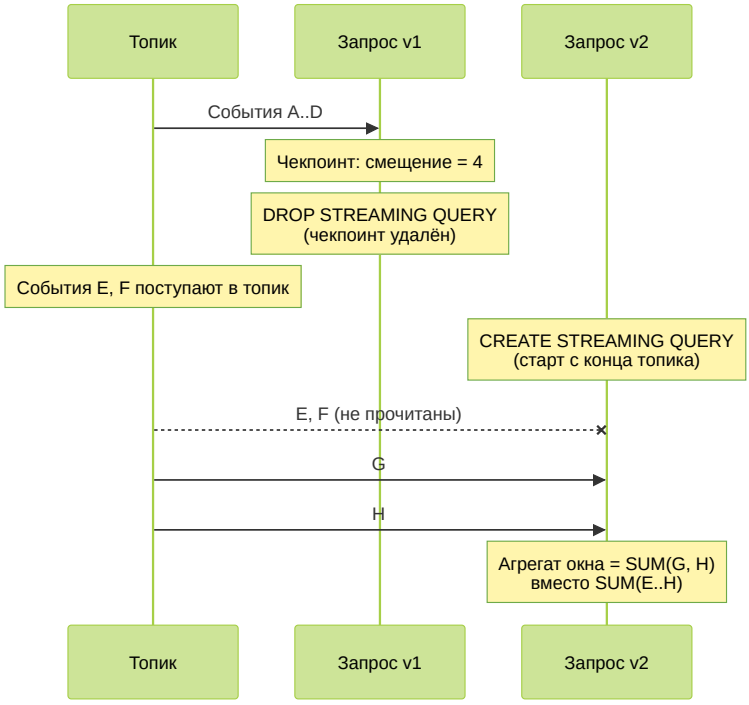


This is expected behavior on the first start — all subsequent windows will receive data for the full interval, which is important to consider when recreating a query.

Event loss when recreating a query

To change the query text, the combination of commands DROP + CREATE is used. When `DROP`, the checkpoint is deleted along with the query, since YDB uses internal storage of read offsets from the source, these offsets are deleted along with the query. The new query does not have a saved position and starts reading from the end of the topic. All events that arrived in the topic between the deletion of the old query and the start of the new one will not be read.

A similar situation occurs if the data pointed to by the offset in the checkpoint has already been deleted from the topic by TTL.



For queries with window aggregation, the first windows after recreation will contain data gaps and understated aggregates.

See also

- Streaming queries — general description of streaming queries.
- Checkpoints — checkpoint mechanism that ensures recovery after failures.
- Writing to tables — writing to tables and UPSERT idempotence.

Checkpoints

A **checkpoint** is persisted state of a running streaming query, used to recover processing after failures. YDB periodically checkpoints all running streaming queries.

Checkpoint contents

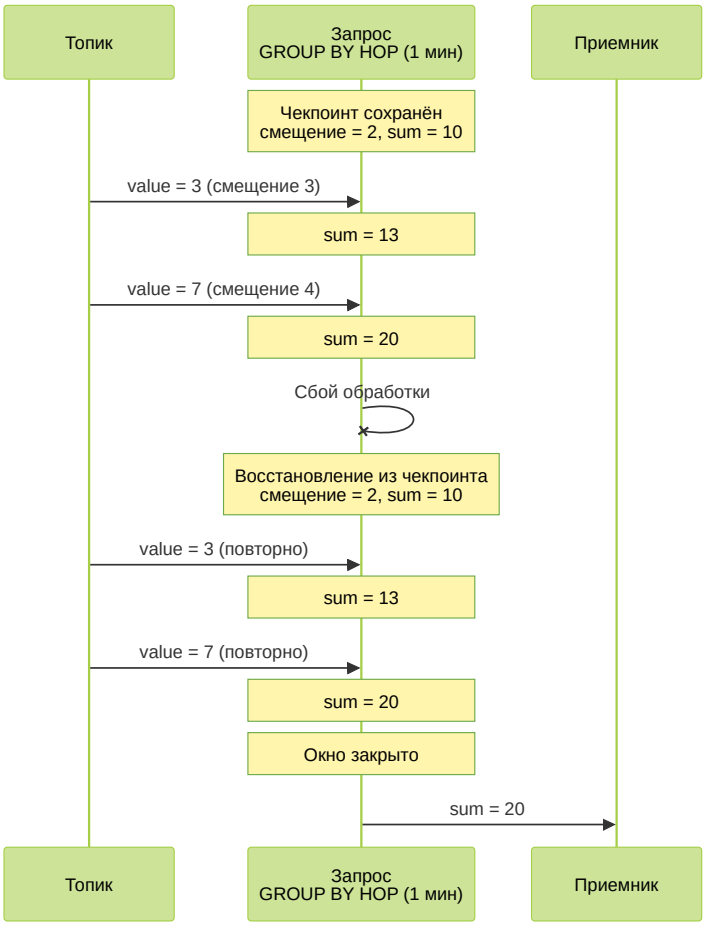
A checkpoint contains:

- Offsets in input topics — positions up to which events were read and processed;
- Aggregation state — intermediate results such as accumulators for GROUP BY HOP.

YDB stores read offsets in its own checkpoints and does not rely on external consumer offsets. When a query is removed (DROP STREAMING QUERY), offsets are removed with the checkpoint — external systems are not aware how far the query read in the topic.

Recovery after failure

When processing fails (compute node restart, network interruption, timeout), the query restarts automatically and restores state from the latest checkpoint: it resumes reading from saved offsets and restores aggregation state.

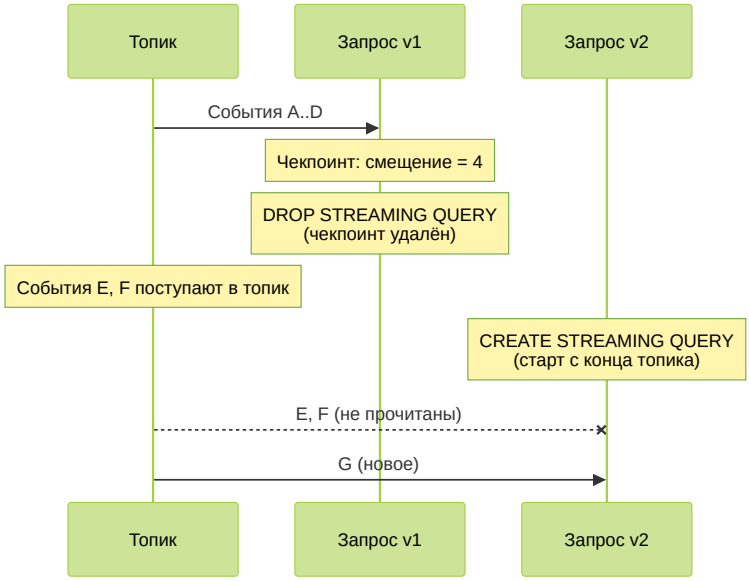


Events that arrived between the last checkpoint and the failure are processed again. That provides at-least-once delivery — each event is processed at least once.

Saving and selecting checkpoints for recovery is automatic. Old checkpoints are removed after a new one is saved successfully.

Checkpoint deleted when recreating a query

When you delete a query (DROP STREAMING QUERY), its checkpoint is deleted with it. Because offsets live only in the checkpoint, a new query (CREATE STREAMING QUERY) has no saved position and starts reading from the end of the topic. Events that arrived between deleting the old query and starting the new one are not read.



The same happens if data referenced by an offset in the checkpoint has already been removed from the topic due to TTL.

For how this affects delivery guarantees, see Event loss when recreating a query.

Disabling checkpoints

To reduce overhead, you can disable checkpointing with pragma `ydb.DisableCheckpoints`.

> **Warning**
>
> With checkpoints disabled there are no consistency guarantees across user or internal restarts. Use only for debugging.

```
CREATE STREAMING QUERY query_without_checkpoints AS
DO BEGIN

PRAGMA ydb.DisableCheckpoints = "TRUE";

INSERT INTO
    output_topic
SELECT
    *
FROM
    input_topic;

END DO
```

See also

- Data delivery guarantees — delivery guarantees and anomalies.
- Streaming queries — streaming queries overview.

Watermarks

Watermark — a monotonically increasing lower bound on event times in the stream (more about the concept: Watermarks). This section describes the configuration of watermarks in streaming queries YDB.

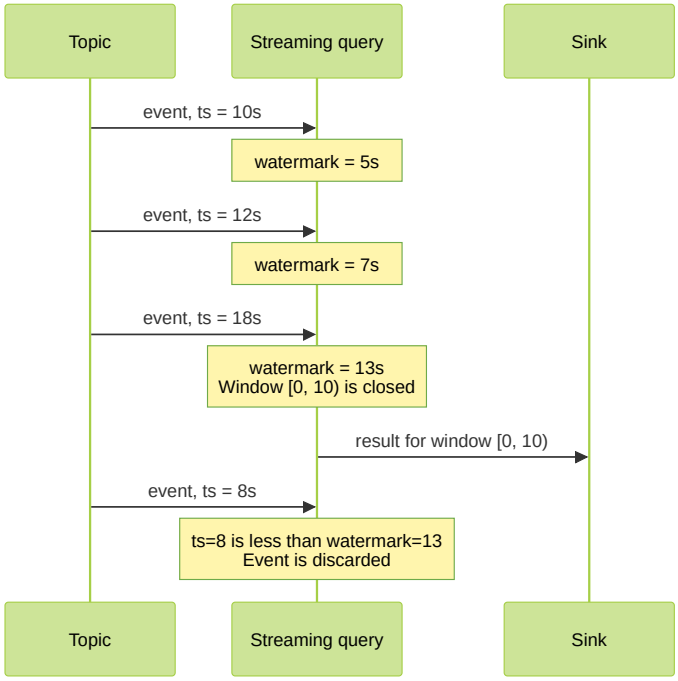
Event time

In stream processing, each event has a timestamp that the system uses to track the progress of time in the stream. In the current implementation, the only source of event time is the write time of the event to the topic, available via the system column `__ydb_write_time`.

> **Note**
>
> Support for arbitrary expressions to extract time from event data (for example, the `event.created_at` field) is planned in future versions.

Usage

Watermark is used by operations that depend on the progress of event time in the stream. In YDB, such operations include window aggregation HoppingWindow — it defines sliding time windows that group events. When a watermark is received, `HoppingWindow` closes all windows that are fully covered by this value.



Watermark computation

When the system receives an event, it updates the watermark — **advancing** it forward along the time axis. The watermark is calculated as `maximum observed event time − delay`, where `delay` is the lag value specified in the `WATERMARK` expression (for example, `Interval("PT5S")` in `WATERMARK = __ydb_write_time - Interval("PT5S")`).

Events in a stream may arrive out of chronological order: an event with timestamp 10:00:03 may be processed after an event with timestamp 10:00:05. Reasons: clock skew in a distributed system, network delays, uneven load on topic partitions.

The `delay` parameter sets the allowable time margin for events arriving with a delay. For example, with `delay` of 5 seconds, an event with time 00:00:48 will be accepted even if events with time 00:00:50 have already arrived: the watermark has not yet reached 00:00:48. If the same event arrives later, when the watermark has already advanced past 00:00:48, it will be considered late and discarded.

On the trade-off between accuracy and result delivery latency: Accuracy vs. Latency Trade-off.

Idle partitions

If the input topic contains multiple partitions, each of them advances the watermark independently. The overall query watermark does not advance ahead of the slowest partition: windows are not closed until at least one partition has reached the corresponding point in time.

If one of the partitions stops receiving data, its watermark stops advancing. Such a partition is called idle. As long as the idle partition is taken into account when computing the overall watermark, it also stops advancing, and results are not emitted despite data arriving from other partitions.

To avoid this blocking, an idle partition is excluded from the common watermark calculation after a configurable timeout period (the `WATERMARK_IDLE_TIMEOUT` parameter; for details, see the Configuration section).

Configuration

Watermarks are enabled and configured in the WITH section when reading from a topic.

Configuration parameters:

- `WATERMARK` — expression for calculating the watermark. Currently, only the write time to a topic with a constant delay is supported. Format: `__ydb_write_time - Interval("<delay>")`, where `<delay>` is specified in ISO 8601 format.
- `WATERMARK_GRANULARITY` — watermark generation frequency. The smaller it is, the higher the CPU consumption by the query and the lower the response latency. Only relevant for streaming queries. Specified in ISO 8601 format. Default value — 1

second.

- `WATERMARK_IDLE_TIMEOUT` — period after which an idle partition will be excluded from the combined watermark calculation. Only relevant for streaming queries. Specified in ISO 8601 format. Default value — 5 seconds.

> **Warning**
>
> When using HoppingWindow, the first parameter (time extractor) and the time source in the WATERMARK expression must match. In the current implementation, both must use `__ydb_write_time`.

Example

Below is an example of a streaming query with watermark and window aggregation. The query reads events from a topic, filters them by the `pass` field, and aggregates the `payload` values in windows of 10 seconds with a 5-second step. The watermark is configured with a 5-second lag.

Input data

```
{"pass": 1, "payload": "a"} // record time: 1970-01-01T00:00:40Z
{"pass": 1, "payload": "b"} // record time: 1970-01-01T00:00:42Z
{"pass": 0, "payload": "c"} // record time: 1970-01-01T00:00:50Z
{"pass": 1, "payload": "d"} // record time: 1970-01-01T00:00:40Z
```

Query

```
CREATE STREAMING QUERY example AS
DO BEGIN
    $input = (
        SELECT
            t.*,
            __ydb_write_time AS ts
        FROM
            Input
        WITH (
            FORMAT = json_each_row,
            SCHEMA = (
                pass Int64,
                payload String
            ),
            WATERMARK = __ydb_write_time - Interval("PT5S")
        ) AS t
    );

    $output = (
        SELECT
            AGGREGATE_LIST(payload) AS result,
            CAST(HOP_END() AS String) AS ts
        FROM
            $input
        WHERE pass > 0
        GROUP BY
            HoppingWindow(ts, "PT5S", "PT10S")
    );

    INSERT INTO Output
    SELECT
        ToBytes(Unwrap(Yson2::SerializeJson(Yson::From(TableRow()))))
    FROM $output;
END DO;
```

Where:

- `CREATE STREAMING QUERY` — creates a named streaming query.
- `__ydb_write_time` is a system column containing the time the event was written to the topic.
- `FORMAT = json_each_row` — data format in a topic, each line contains a separate JSON object.
- `WATERMARK = __ydb_write_time - Interval("PT5S")` is a watermark with a 5-second lag. `Interval("PT5S")` sets the interval in ISO 8601 format.
- `AGGREGATE_LIST` — aggregate function that collects values into a list.
- `HOP_END()` returns the timestamp of the end of the current window.
- `HoppingWindow(ts, "PT5S", "PT10S")` — window function with a step of 5 seconds and a window size of 10 seconds.

Result

```
{"result": ["a", "b"], "ts": "1970-01-01T00:00:45.000000Z"}
```

Explanation

1. The first event (`"a"`, record time 40s) passes the filter (`pass > 0`) and enters windows `[35; 45)` and `[40; 50)`. Watermark advances to 35s and does not close any window.
2. The second event (`"b"`, record time 42s) similarly falls into windows `[35; 45)` and `[40; 50)`. The watermark advances to 37s.
3. The third event (`"c"`, record time 50s) is discarded by the filter (`pass = 0`). Despite this, the event still advances the watermark to 45s. The watermark closes window `[35; 45)` — result `["a", "b"]` is output.

4. The fourth event (`"d"`, record time 40s) does not advance the watermark: it is already at the 45s mark. The event passes the filter but is discarded as late — its record time (40s) is less than the current watermark (45s). Although window `[40; 50)` is still open, the watermark has already promised that no events with time < 45s will arrive, so `d` is not counted in any of its windows.

See also

- GROUP BY ... HoppingWindow — a window function that uses watermarks.
- WITH — the WITH clause for configuring topic read parameters.
- Data delivery guarantees — data delivery guarantees.
- Checkpoints — checkpoint mechanism.

Example applications working with YDB

This section outlines the implementation of example applications, all designed to perform similar functions, using the YDB SDKs across various programming languages. Each app is developed to demonstrate how a respective SDK can be utilized in a specific language.

- C++
- C# (.NET)
- Go
- Java
- JavaScript
- Python
- Rust

> **Note**
>
> Refer to YDB SDK reference documentation for more details.

A test app performs the following steps:

Initializing a database connection

To interact with YDB, create instances of the driver, client, and session:

- The YDB driver facilitates interaction between the app and YDB nodes at the transport layer. It must be initialized before creating a client or session and must persist throughout the YDB access lifecycle.
- The YDB client operates on top of the YDB driver and enables the handling of entities and transactions.
- The YDB session, which is part of the YDB client context, contains information about executed transactions and prepared queries.

C++ | C# (.NET) | Go | Java | JavaScript | PHP | Python | Rust

Creating tables

Create tables to be used in operations on a test app. This step results in the creation of database tables for the series directory data model:

- `Series`
- `Seasons`
- `Episodes`

After the tables are created, a method for retrieving information about data schema objects is called, and the result of its execution is displayed.

C++ | C# (.NET) | Go | Java | JavaScript | PHP | Python | Rust

Adding data

Add data to the created tables using the `UPSERT` statement in YQL. A data update request is sent to the server as a single request with transaction auto-commit mode enabled.

C++ | C# (.NET) | Go | Java | JavaScript | PHP | Python | Rust

Retrieving data

Retrieve data using a `SELECT` statement in YQL. Handle the retrieved data selection in the app.

C++ | C# (.NET) | Go | Java | JavaScript | PHP | Python | Rust

Parameterized queries

Query data using parameters. This query execution method is preferable because it allows the server to reuse the query execution plan for subsequent calls and protects against vulnerabilities such as SQL injection.

C++ | C# (.NET) | Go | Java | JavaScript | PHP | Python | Rust

Multistep transactions

Multiple statements can be executed within a single multistep transaction. Client-side code can run between query steps. Using a transaction ensures that queries executed in its context are consistent with each other.

C++ | C# (.NET) | Go | Java | JavaScript | PHP | Python

Managing transactions

Transactions are managed through TCL `Begin` and `Commit` calls.

In most cases, instead of explicitly using `Begin` and `Commit` calls, it's better to use transaction control parameters in execute calls. This allows to avoid additional requests to YDB server and thus run queries more efficiently.

C++ | C# (.NET) | Go | Java | JavaScript | PHP | Python | Rust

Error handling

For more information about error handling, see Error handling in the API.

Example app in C++

This page contains a detailed description of the code of a test app that is available as part of the YDB C++SDK.

Initializing a database connection

To interact with YDB, create instances of the driver, client, and session:

- The YDB driver facilitates interaction between the app and YDB nodes at the transport layer. It must be initialized before creating a client or session and must persist throughout the YDB access lifecycle.
- The YDB client operates on top of the YDB driver and enables the handling of entities and transactions.
- The YDB session, which is part of the YDB client context, contains information about executed transactions and prepared queries.

App code snippet for driver initialization:

```
    auto connectionParams = TConnectionsParams()
        .SetEndpoint(endpoint)
        .SetDatabase(database)
        .SetAuthToken(GetEnv("YDB_TOKEN"));

    TDriver driver(connectionParams);
```

App code snippet for creating a client:

```
    TClient client(driver);
```

Creating tables

Create tables to be used in operations on a test app. This step results in the creation of database tables for the series directory data model:

- `Series`
- `Seasons`
- `Episodes`

After the tables are created, a method for retrieving information about data schema objects is called, and the result of its execution is displayed.

```
    //! Creates sample tables with the ExecuteQuery method
    ThrowOnError(client.RetryQuerySync([](TSession session) {
        auto query = Sprintf(R"(
            CREATE TABLE series (
                series_id Uint64,
                title Utf8,
                series_info Utf8,
                release_date Uint64,
                PRIMARY KEY (series_id)
            );
        )");
        return session.ExecuteQuery(query, TTxControl::NoTx()).GetValueSync();
    }));
```

Adding data

Add data to the created tables using the `UPSERT` statement in YQL. A data update request is sent to the server as a single request with transaction auto-commit mode enabled.

Code snippet for data insert/update:

```
//! Shows basic usage of mutating operations.
void UpsertSimple(TQueryClient client) {
    ThrowOnError(client.RetryQuerySync([](TSession session) {
        auto query = Sprintf(R"(
            UPSERT INTO episodes (series_id, season_id, episode_id, title) VALUES
                (2, 6, 1, "TBD");
        )");

        return session.ExecuteQuery(query,
            TTxControl::BeginTx(TTxSettings::SerializableRW()).CommitTx()).GetValueSync();
    }));
}
```

`PRAGMA TablePathPrefix` adds a specified prefix to the table paths. It uses standard filesystem path concatenation, meaning it supports parent folder referencing and does not require a trailing slash. For example:

```
PRAGMA TablePathPrefix = "/cluster/database";
SELECT * FROM episodes;
```

For more information about `PRAGMA` in YQL, refer to the YQL documentation.

Retrieving data

Retrieve data using a `SELECT` statement in YQL. Handle the retrieved data selection in the app.

To execute YQL queries, use the `ExecuteQuery` method.
The SDK lets you explicitly control the execution of transactions and configure the transaction execution mode using the `TTxControl` class.

In the code snippet below, the transaction settings are defined using the `TTxControl::BeginTx` method. With `TTxSettings`, set the `SerializableRW` transaction execution mode. When all the queries in the transaction are completed, the transaction is automatically completed by explicitly setting `CommitTx()`. The `query` described using the YQL syntax is passed to the `ExecuteQuery` method for execution.

```
void SelectSimple(TQueryClient client) {
    TMaybe<TResultSet> resultSet;
    ThrowOnError(client.RetryQuerySync([&resultSet](TSession session) {
        auto query = Sprintf(R"(
            SELECT series_id, title, CAST(release_date AS Date) AS release_date
            FROM series
            WHERE series_id = 1;
        )");

        auto txControl =
            // Begin a new transaction with SerializableRW mode
            TTxControl::BeginTx(TTxSettings::SerializableRW())
            // Commit the transaction at the end of the query
            .CommitTx();

        auto result = session.ExecuteQuery(query, txControl).GetValueSync();
        if (!result.IsSuccess()) {
            return result;
        }
        resultSet = result.GetResultSet(0);
        return result;
    }));
```

Processing execution results

The `TResultSetParser` class is used for processing query execution results.

The code snippet below shows how to process query results using the `parser` object:

```
    TResultSetParser parser(*resultSet);
    while (parser.TryNextRow()) {
        Cout << "> SelectSimple:" << Endl << "Series"
            << ", Id: " << parser.ColumnParser("series_id").GetOptionalUint64()
            << ", Title: " << parser.ColumnParser("title").GetOptionalUtf8()
            << ", Release date: " << parser.ColumnParser("release_date").GetOptionalDate()->FormatLocalTime
("%Y-%m-%d")
            << Endl;
    }
}
```

The given code snippet prints the following text to the console at startup:

```
> SelectSimple:
series, Id: 1, title: IT Crowd, Release date: 2006-02-03
```

Parameterized queries

Query data using parameters. This query execution method is preferable because it allows the server to reuse the query execution plan for subsequent calls and protects against vulnerabilities such as SQL injection.

The code snippet shows the use of parameterized queries and the `TParamsBuilder` to generate parameters and pass them to the `ExecuteQuery` method:

```
void SelectWithParams(TQueryClient client) {
    TMaybe<TResultSet> resultSet;
    ThrowOnError(client.RetryQuerySync([&resultSet](TSession session) {
        ui64 seriesId = 2;
        ui64 seasonId = 3;
        auto query = Sprintf(R"(
            DECLARE $seriesId AS Uint64;
            DECLARE $seasonId AS Uint64;

            SELECT sa.title AS season_title, sr.title AS series_title
            FROM seasons AS sa
            INNER JOIN series AS sr
            ON sa.series_id = sr.series_id
            WHERE sa.series_id = $seriesId AND sa.season_id = $seasonId;
        )");

        auto params = TParamsBuilder()
            .AddParam("$seriesId")
                .Uint64(seriesId)
                .Build()
            .AddParam("$seasonId")
                .Uint64(seasonId)
                .Build()
            .Build();

        auto result = session.ExecuteQuery(
```

```
            query,
            TTxControl::BeginTx(TTxSettings::SerializableRW()).CommitTx(),
            params).GetValueSync();

        if (!result.IsSuccess()) {
            return result;
        }
        resultSet = result.GetResultSet(0);
        return result;
    }));

    TResultSetParser parser(*resultSet);
    if (parser.TryNextRow()) {
        Cout << "> SelectWithParams:" << Endl << "Season"
            << ", Title: " << parser.ColumnParser("season_title").GetOptionalUtf8()
            << ", Series title: " << parser.ColumnParser("series_title").GetOptionalUtf8()
            << Endl;
    }
}
```

The given code snippet prints the following text to the console at startup:

```
> SelectWithParams:
Season, title: Season 3, series title: Silicon Valley
```

Stream queries

Making a stream query that results in a data stream. Streaming lets you read an unlimited number of rows and amount of data.

> **Warning**
>
> Do not use the `StreamExecuteQuery` method without wrapping the call with `RetryQuery` or `RetryQuerySync`.

```
void StreamQuerySelect(TQueryClient client) {
    Cout << "> StreamQuery:" << Endl;

    ThrowOnError(client.RetryQuerySync([](TQueryClient client) -> TStatus {
        auto query = Sprintf(R"(
            DECLARE $series AS List<UInt64>;

            SELECT series_id, season_id, title, CAST(first_aired AS Date) AS first_aired
            FROM seasons
            WHERE series_id IN $series
            ORDER BY season_id;
        )");

        auto paramsBuilder = TParamsBuilder();
        auto& listParams = paramsBuilder
                                    .AddParam("$series")
                                    .BeginList();

        for (auto x : {1, 10}) {
            listParams.AddListItem().Uint64(x);
        }

        auto parameters = listParams
                                .EndList()
                                .Build()
                                .Build();

        // Executes stream query
        auto resultStreamQuery = client.StreamExecuteQuery(query, TTxControl::NoTx(), parameters).GetValueSyn
c();

        if (!resultStreamQuery.IsSuccess()) {
            return resultStreamQuery;
        }

        // Iterates over results
        bool eos = false;

        while (!eos) {
            auto streamPart = resultStreamQuery.ReadNext().ExtractValueSync();

            if (!streamPart.IsSuccess()) {
                eos = true;
                if (!streamPart.EOS()) {
                    return streamPart;
                }
                continue;
            }

            // It is possible for lines to be duplicated in the output stream due to an external retrier
            if (streamPart.HasResultSet()) {
                auto rs = streamPart.ExtractResultSet();
                TResultSetParser parser(rs);
                while (parser.TryNextRow()) {
                    Cout << "Season"
```

```
                            << ", SeriesId: " << parser.ColumnParser("series_id").GetOptionalUint64()
                            << ", SeasonId: " << parser.ColumnParser("season_id").GetOptionalUint64()
                            << ", Title: " << parser.ColumnParser("title").GetOptionalUtf8()
                            << ", Air date: " << parser.ColumnParser("first_aired").GetOptionalDate()->Format
LocalTime("%Y-%m-%d")
                            << Endl;
                }
            }
        }
        return TStatus(EStatus::SUCCESS, NYql::TIssues());
    }));

}
```

The given code snippet prints the following text to the console at startup (there may be duplicate lines in the output stream due to an external `RetryQuerySync`):

```
> StreamQuery:
Season, SeriesId: 1, SeasonId: 1, Title: Season 1, Air date: 2006-02-03
Season, SeriesId: 1, SeasonId: 2, Title: Season 2, Air date: 2007-08-24
Season, SeriesId: 1, SeasonId: 3, Title: Season 3, Air date: 2008-11-21
Season, SeriesId: 1, SeasonId: 4, Title: Season 4, Air date: 2010-06-25
```

Multistep transactions

Multiple statements can be executed within a single multistep transaction. Client-side code can run between query steps. Using a transaction ensures that queries executed in its context are consistent with each other.

The first step is to prepare and execute the first query:

```
//! Shows usage of transactions consisting of multiple data queries with client logic between them.
void MultiStep(TQueryClient client) {
    TMaybe<TResultSet> resultSet;
    ThrowOnError(client.RetryQuerySync([&resultSet](TSession session) {
        ui64 seriesId = 2;
        ui64 seasonId = 5;
        auto query1 = Sprintf(R"(
            DECLARE $seriesId AS Uint64;
            DECLARE $seasonId AS Uint64;

            SELECT first_aired AS from_date FROM seasons
            WHERE series_id = $seriesId AND season_id = $seasonId;
        )");

        auto params1 = TParamsBuilder()
            .AddParam("$seriesId")
                .Uint64(seriesId)
                .Build()
            .AddParam("$seasonId")
                .Uint64(seasonId)
                .Build()
            .Build();

        // Execute the first query to retrieve the required values for the client.
        // Transaction control settings do not set the CommitTx flag, allowing the transaction to remain acti
ve
        // after query execution.
        auto result = session.ExecuteQuery(
            query1,
            TTxControl::BeginTx(TTxSettings::SerializableRW()),
            params1);

        auto resultValue = result.GetValueSync();

        if (!resultValue.IsSuccess()) {
            return resultValue;
        }
```

A transaction identifier needs to be obtained to continue working within the current transaction:

```
        // Get the active transaction id
        auto txId = resultValue.GetTransaction()->GetId();

        // Processing the request result
        TResultSetParser parser(resultValue.GetResultSet(0));
        parser.TryNextRow();
        auto date = parser.ColumnParser("from_date").GetOptionalUint64();

        // Perform some client logic on returned values
        auto userFunc = [] (const TInstant fromDate) {
            return fromDate + TDuration::Days(15);
        };

        TInstant fromDate = TInstant::Days(*date);
        TInstant toDate = userFunc(fromDate);
```

The next step is to create the next query that uses the results of code execution on the client side:

```
        // Construct next query based on the results of client logic
        auto query2 = Sprintf(R"(
            DECLARE $seriesId AS Uint64;
            DECLARE $fromDate AS Uint64;
            DECLARE $toDate AS Uint64;

            SELECT season_id, episode_id, title, air_date FROM episodes
            WHERE series_id = $seriesId AND air_date >= $fromDate AND air_date <= $toDate;
        )");

        auto params2 = TParamsBuilder()
            .AddParam("$seriesId")
                .Uint64(seriesId)
                .Build()
            .AddParam("$fromDate")
                .Uint64(fromDate.Days())
                .Build()
            .AddParam("$toDate")
                .Uint64(toDate.Days())
                .Build()
            .Build();

        // Execute the second query.
        // The transaction control settings continue the active transaction (tx)
        // and commit it at the end of the second query execution.
        auto result2 = session.ExecuteQuery(
            query2,
            TTxControl::Tx(txId).CommitTx(),
            params2).GetValueSync();

        if (!result2.IsSuccess()) {
            return result2;
        }
        resultSet = result2.GetResultSet(0);
        return result2;
    })); // The end of the retried lambda

    TResultSetParser parser(*resultSet);
    Cout << "> MultiStep:" << Endl;
    while (parser.TryNextRow()) {
        auto airDate = TInstant::Days(*parser.ColumnParser("air_date").GetOptionalUint64());

        Cout << "Episode " << parser.ColumnParser("episode_id").GetOptionalUint64()
            << ", Season: " << parser.ColumnParser("season_id").GetOptionalUint64()
            << ", Title: " << parser.ColumnParser("title").GetOptionalUtf8()
            << ", Air date: " << airDate.FormatLocalTime("%a %b %d, %Y")
            << Endl;
    }
}
```

The given code snippets output the following text to the console at startup:

```
> MultiStep:
Episode 1, Season: 5, title: Grow Fast or Die Slow, Air date: Sun Mar 25, 2018
Episode 2, Season: 5, title: Reorientation, Air date: Sun Apr 01, 2018
Episode 3, Season: 5, title: Chief Operating Officer, Air date: Sun Apr 08, 2018
```

Managing transactions

Transactions are managed through TCL `Begin` and `Commit` calls.

In most cases, instead of explicitly using `Begin` and `Commit` calls, it's better to use transaction control parameters in execute calls. This allows to avoid additional requests to YDB server and thus run queries more efficiently.

Code snippet for `BeginTransaction` and `tx.Commit()` calls:

```
void ExplicitTcl(TQueryClient client) {
    // Demonstrate the use of explicit Begin and Commit transaction control calls.
    // In most cases, it's preferable to use transaction control settings within ExecuteDataQuery calls inste
ad,
    // as this avoids additional hops to the YDB cluster and allows for more efficient query execution.
    ThrowOnError(client.RetryQuerySync([](TQueryClient client) -> TStatus {
        auto airDate = TInstant::Now();
        auto session = client.GetSession().GetValueSync().GetSession();
        auto beginResult = session.BeginTransaction(TTxSettings::SerializableRW()).GetValueSync();
        if (!beginResult.IsSuccess()) {
            return beginResult;
        }

        // Get newly created transaction id
        auto tx = beginResult.GetTransaction();

        auto query = Sprintf(R"(
            DECLARE $airDate AS Date;

            UPDATE episodes SET air_date = CAST($airDate AS Uint16) WHERE title = "TBD";
        )");

        auto params = TParamsBuilder()
```

```
            .AddParam("$airDate")
                .Date(airDate)
                .Build()
            .Build();

        // Execute query.
        // Transaction control settings continues active transaction (tx)
        auto updateResult = session.ExecuteQuery(query,
            TTxControl::Tx(tx.GetId()),
            params).GetValueSync();

        if (!updateResult.IsSuccess()) {
            return updateResult;
        }
        // Commit active transaction (tx)
        return tx.Commit().GetValueSync();
    }));
}
```

Example app in C# (.NET)

This page contains a detailed description of the code of a test app that uses the YDB C# (.NET) SDK.

Initializing a database connection

To interact with YDB, create instances of the driver, client, and session:

- The YDB driver facilitates interaction between the app and YDB nodes at the transport layer. It must be initialized before creating a client or session and must persist throughout the YDB access lifecycle.
- The YDB client operates on top of the YDB driver and enables the handling of entities and transactions.
- The YDB session, which is part of the YDB client context, contains information about executed transactions and prepared queries.

App code snippet for connecting to the database:

```
using Ydb.Sdk.Ado;

await using var dataSource = new YdbDataSource("Host=localhost;Port=2136;Database=/local");
await using var connection = await dataSource.OpenConnectionAsync();
```

Creating tables

Create tables to be used in operations on a test app. This step results in the creation of database tables for the series directory data model:

- `Series`
- `Seasons`
- `Episodes`

After the tables are created, a method for retrieving information about data schema objects is called, and the result of its execution is displayed.

To create tables, use `YdbCommand` with a DDL (Data Definition Language) YQL query:

```
await using var command = new YdbCommand(connection)
{
    CommandText = @"
        CREATE TABLE series (
            series_id Uint64 NOT NULL,
            title Utf8,
            series_info Utf8,
            release_date Date,
            PRIMARY KEY (series_id)
        );

        CREATE TABLE seasons (
            series_id Uint64,
            season_id Uint64,
            title Utf8,
            first_aired Date,
            last_aired Date,
            PRIMARY KEY (series_id, season_id)
        );

        CREATE TABLE episodes (
            series_id Uint64,
            season_id Uint64,
            episode_id Uint64,
            title Utf8,
            air_date Date,
            PRIMARY KEY (series_id, season_id, episode_id)
        );"
};
await command.ExecuteNonQueryAsync();
```

Adding data

Add data to the created tables using the `UPSERT` statement in YQL. A data update request is sent to the server as a single request with transaction auto-commit mode enabled.

Code snippet for data insert/update:

```
await using var command = new YdbCommand(@"
    UPSERT INTO series (series_id, title, release_date) VALUES
        ($id, $title, $release_date);
    ", connection);
command.Parameters.Add(new YdbParameter("$id", YdbDbType.Uint64, 1UL));
command.Parameters.Add(new YdbParameter("$title", YdbDbType.Text, "NewTitle"));
command.Parameters.Add(new YdbParameter("$release_date", YdbDbType.Date, DateTime.UtcNow));
await command.ExecuteNonQueryAsync();
```

Retrieving data

Retrieve data using a `SELECT` statement in YQL. Handle the retrieved data selection in the app.

To read data with a YQL query, use the `ExecuteReaderAsync` method. Query parameters are passed through the `Parameters` collection of the `YdbCommand` object:

```
await using var command = new YdbCommand(@"
    SELECT
        series_id,
        title,
        release_date
    FROM series
    WHERE series_id = $id;
    ", connection);
command.Parameters.Add(new YdbParameter("$id", YdbDbType.Uint64, id));
await using var reader = await command.ExecuteReaderAsync();
```

Processing execution results

The query result is processed via `DbDataReader`. Example of processing the result:

```
while (await reader.ReadAsync())
{
    Console.WriteLine($"> Series, " +
        $"series_id: {reader.GetUint64(0)}, " +
        $"title: {reader.GetString(1)}, " +
        $"release_date: {reader.GetDateTime(2)}");
}
```

For sequential row reading from another query:

```
await using var command = new YdbCommand(
    "SELECT title FROM seasons ORDER BY series_id, season_id;", connection);
await using var reader = await command.ExecuteReaderAsync();
while (await reader.ReadAsync())
{
    Console.WriteLine(reader.GetString(0));
}
```

Example app in Go

This page provides a detailed description of the code for a test app that uses the YDB Go SDK.

Downloading and starting

The instructions below assume that Git and Go are installed. Make sure to install the YDB Go SDK.

Create a working directory and use it to run the following command from the command line to clone the GitHub repository:

```
git clone https://github.com/ydb-platform/ydb-go-sdk.git
```

Next, from the same working directory, run the following command to start the test app:

Local Docker

To connect to a locally deployed YDB database according to the Docker use case, run the following command in the default configuration:

```
( export YDB_ANONYMOUS_CREDENTIALS=1 && cd ydb-go-sdk/examples && \
go run ./basic/native/query -ydb="grpc://localhost:2136/local" )
```

Any database

To run the example against any available YDB database, the endpoint and the database path need to be provide.

If authentication is enabled for the database, the authentication mode needs to be chosen and credentials (a token or a username/password pair) need to be provided.

Run the command as follows:

```
( export <auth_mode_var>="<auth_mode_value>" && cd ydb-go-sdk/examples && \
go run ./basic -ydb="<endpoint>?database=<database>" )
```

where

- `<endpoint>`: The endpoint.
- `<database>`: The database path.
- `<auth_mode_var>`: The environment variable that determines the authentication mode.
- `<auth_mode_value>` is the authentication parameter value for the selected mode.

For example:

```
( export YDB_ACCESS_TOKEN_CREDENTIALS="t1.9euelZqOnJuJlc..." && cd ydb-go-sdk/examples && \
go run ./basic -ydb="grpcs://ydb.example.com:2135/somepath/somelocation" )
```

Initializing a database connection

To interact with YDB, create instances of the driver, client, and session:

- The YDB driver facilitates interaction between the app and YDB nodes at the transport layer. It must be initialized before creating a client or session and must persist throughout the YDB access lifecycle.
- The YDB client operates on top of the YDB driver and enables the handling of entities and transactions.
- The YDB session, which is part of the YDB client context, contains information about executed transactions and prepared queries.

To work with YDB in `Go`, import the `ydb-go-sdk` driver package:

```
import (
 "context"
 "log"
 "path"

 "github.com/ydb-platform/ydb-go-sdk/v3"
 "github.com/ydb-platform/ydb-go-sdk/v3/query"
)
```

To interact with YDB, it is necessary to create a YDB driver:

```
db, err := ydb.Open(context.Background(), "grpc://localhost:2136/local")
if err != nil {
  // handle connection error
}

// You should close the driver when your application finishes its work (for example, when exiting the progra
m).
defer db.Close(context.Background())
```

The `ydb.Open` method returns a driver instance if successful. The driver performs several services, such as YDB cluster discovery and client-side load balancing.

The `ydb.Open` method takes two mandatory arguments:

- a context
- a YDB connection string

There are also many connection options available that let you override the default settings.

By default, anonymous authentication is used. To connect to the YDB cluster using a token, use the following syntax:

```
db, err := ydb.Open(context.Background(), clusterEndpoint,
 ydb.WithAccessTokenCredentials(token),
)
```

You can see the full list of auth providers in the ydb-go-sdk documentation and on the recipes page.

It is necessary to close the driver at the end of work to clean up resources.

```
defer db.Close(ctx)
```

The `db` struct is the entry point for working with YDB. To query tables, use the `db.Query()` query service:

YQL queries are executed within special objects called `query.Session`. Sessions store the execution context of queries (for example, transactions) and provide server-side load balancing among the YDB cluster nodes.

The query service client provides an API for executing queries:

- `db.Query().Do(ctx, op)` creates sessions in the background and automatically retries the provided `op func(ctx context.Context, s query.Session) error` operation if necessary. As soon as a session is ready, it is passed to the callback.
- `db.Query().DoTx(ctx, op)` automatically handles the transaction lifecycle. It provides a prepared transaction object, `query.TxActor`, to the user-defined function `op func(ctx context.Context, tx query.TxActor)` error. If the operation returns without an error (nil), the transaction commits automatically. If the operation returns an error, the transaction rolls back automatically.
- `db.Query().Exec` runs a single query that returns **no result**, with automatic retry logic on failure. This method returns nil if the execution is successful or an error otherwise.
- `db.Query().Query` executes a single query containing one or more statements that return a result. It automatically handles retries. Upon successful execution, it returns a fully materialized result (`query.Result`). All result rows are loaded into memory and available for immediate iteration. For queries returning large datasets, this may lead to an out of memory problem.
- `db.Query().QueryResultSet` executes a query that contains exactly one statement returning results (it may contain other auxiliary statements that return no results, such as `UPSERT`). Like `db.Query().Query`, it automatically retries failed operations and returns a fully materialized result set (`query.ResultSet`). Queries that return large datasets may cause an OOM error.
- `db.Query().QueryRow` runs queries expected to return exactly one row. It also automatically retries failed operations. On success, it returns a `query.Row` instance.

Creating tables

Create tables to be used in operations on a test app. This step results in the creation of database tables for the series directory data model:

- `Series`
- `Seasons`
- `Episodes`

After the tables are created, a method for retrieving information about data schema objects is called, and the result of its execution is displayed.

Example of a query with no returned result (table creation):

```
import "github.com/ydb-platform/ydb-go-sdk/v3/query"

err = db.Query().Exec(ctx, `
 CREATE TABLE IF NOT EXISTS series (
  series_id Bytes,
  title Text,
  series_info Text,
  release_date Date,
  comment Text,

  PRIMARY KEY(series_id)
 )`, query.WithTxControl(query.NoTx()),
)
if err != nil {
  // handle query execution error
}
```

Retrieving data

Retrieve data using a `SELECT` statement in YQL. Handle the retrieved data selection in the app.

To execute YQL queries and fetch results, use `query.Session` methods: `query.Session.Query`, `query.Session.QueryResultSet`, or `query.Session.QueryRow`.

The YDB SDK supports explicit transaction control via the `query.TxControl` structure:

```
readTx := query.TxControl(
 query.BeginTx(
  query.WithSnapshotReadOnly(),
 ),
 query.CommitTx(),
)
row, err := db.Query().QueryRow(ctx,`
 DECLARE $seriesID AS Uint64;
 SELECT
```

```
   series_id,
   title,
   release_date
 FROM
   series
 WHERE
   series_id = $seriesID;`,
 query.WithParameters(
  ydb.ParamsBuilder().Param("$seriesID").Uint64(1).Build(),
 ),
 query.WithTxControl(readTx),
)
if err != nil {
  // handle query execution error
}
```

You can extract row data (`query.Row`) using the following methods:

- `query.Row.ScanStruct` — scans row data into a struct based on struct field tags that match column names.
- `query.Row.ScanNamed` — scans data into variables using explicitly defined column-variable pairs.
- `query.Row.Scan` — scans data directly by column order into the provided variables.

ScanStruct

```
var info struct {
 SeriesID    string    `sql:"series_id"`
 Title       string    `sql:"title"`
 ReleaseDate time.Time `sql:"release_date"`
}
err = row.ScanStruct(&info)
if err != nil {
  // handle query execution error
}
```

ScanNamed

```
var seriesID, title string
var releaseDate time.Time
err = row.ScanNamed(query.Named("series_id", &seriesID), query.Named("title", &title), query.Named("release_d
ate", &releaseDate))
if err != nil {
  // handle query execution error
}
```

Scan

```
var seriesID, title string
var releaseDate time.Time
err = row.Scan(&seriesID, &title, &releaseDate)
if err != nil {
  // handle query execution error
}
```

Warning

If the expected query result is very large, avoid loading all data into memory using helper methods like `query.Client.Query` or `query.Client.QueryResultSet`. These methods return fully materialized results, storing all rows from the server in local client memory. Large result sets can cause an OOM problem.

Instead, use the `query.TxActor.Query` or `query.TxActor.QueryResultSet` methods on a transaction or session. These methods return iterators over results without fully materializing them upfront. The `query.Session` object is accessible via the `query.Client.Do` method, which handles automatic retries. Keep in mind that the read operation can be interrupted at any time, restarting the entire query process. Therefore, the user function passed to `Do` may run multiple times.

```
err = db.Query().Do(ctx,
 func(ctx context.Context, s query.Session) error {
  rows, err := s.QueryResultSet(ctx,`
   SELECT series_id, season_id, title, first_aired
   FROM seasons`,
  )
  if err != nil {
   return err
  }
  defer rows.Close(ctx)
  for row, err := range rows.Rows(ctx) {
   if err != nil {
    return err
   }
   var info struct {
    SeriesID    string    `sql:"series_id"`
    SeasonID    string    `sql:"season_id"`
    Title       string    `sql:"title"`
    FirstAired  time.Time `sql:"first_aired"`
   }
```

```
    err = row.ScanStruct(&info)
    if err != nil {
     return err
    }
    fmt.Printf("%+v\n", info)
   }
   return nil
  },
  query.WithIdempotent(),
 )
 if err != nil {
   // handle query execution error
 }
```

Example app in Java

This page contains a detailed description of the code of a test app that is available as part of the YDB Java SDK Examples.

Downloading SDK Examples and running the example

The following execution scenario is based on Git and Maven.

Create a working directory and use it to run from the command line the command to clone the GitHub repository:

```
git clone https://github.com/ydb-platform/ydb-java-examples
```

Then build the SDK Examples

```
mvn package -f ./ydb-java-examples
```

Next, from the same working directory, run the following command to start the test app:

Local Docker

To connect to a locally deployed YDB database according to the Docker use case, run the following command in the default configuration:

```
YDB_ANONYMOUS_CREDENTIALS=1 java -jar ydb-java-examples/query-example/target/ydb-query-example.jar grpc://loc
alhost:2136/local
```

Any database

To run the example against any available YDB database, the endpoint and the database path need to be provide.

If authentication is enabled for the database, the authentication mode needs to be chosen and credentials (a token or a username/password pair) need to be provided.

Run the command as follows:

```
<auth_mode_var>="<auth_mode_value>" java -jar ydb-java-examples/query-example/target/ydb-query-example.jar gr
pcs://<endpoint>:<port>/<database>
```

where

- `<endpoint>`: The endpoint.
- `<database>`: The database path.
- `<auth_mode_var>`: The environment variable that determines the authentication mode.
- `<auth_mode_value>` is the authentication parameter value for the selected mode.

For example:

```
YDB_ACCESS_TOKEN_CREDENTIALS="..." java -jar ydb-java-examples/query-example/target/ydb-query-example.jar grp
cs://ydb.example.com:2135/somepath/somelocation
```

Initializing a database connection

To interact with YDB, create instances of the driver, client, and session:

- The YDB driver facilitates interaction between the app and YDB nodes at the transport layer. It must be initialized before creating a client or session and must persist throughout the YDB access lifecycle.
- The YDB client operates on top of the YDB driver and enables the handling of entities and transactions.
- The YDB session, which is part of the YDB client context, contains information about executed transactions and prepared queries.

Main driver initialization parameters

- A connection string containing details about an endpoint and database. This is the only parameter that is required.
- Authentication provider. Unless explicitly specified, an anonymous connection is used.
- Session pool settings

App code snippet for driver initialization:

```
this.transport = GrpcTransport.forConnectionString(connectionString)
        .withAuthProvider(CloudAuthHelper.getAuthProviderFromEnviron())
        .build();
this.queryClient = QueryClient.newClient(transport).build();
```

It is also [recommended] (../../../recipes/ydb-sdk/retry.md) to use the `SessionRetryContext` helper class for execution of operations with the YDB: it ensures proper retries in case the database becomes partially unavailable. Sample code to initialize the retry context:

```
this.retryCtx = SessionRetryContext.create(queryClient).build();
```

Creating tables

Create tables to be used in operations on a test app. This step results in the creation of database tables for the series directory data model:

- `Series`

- `Seasons`
- `Episodes`

After the tables are created, a method for retrieving information about data schema objects is called, and the result of its execution is displayed.

To create tables, use the `TxMode.NONE` transaction mode, which allows the execution of DDL queries:

```
private void createTables() {
    retryCtx.supplyResult(session -> session.createQuery(""
            + "CREATE TABLE series ("
            + "  series_id UInt64,"
            + "  title Text,"
            + "  series_info Text,"
            + "  release_date Date,"
            + "  PRIMARY KEY(series_id)"
            + ")", TxMode.NONE).execute()
    ).join().getStatus().expectSuccess("Can't create table series");

    retryCtx.supplyResult(session -> session.createQuery(""
            + "CREATE TABLE seasons ("
            + "  series_id UInt64,"
            + "  season_id UInt64,"
            + "  title Text,"
            + "  first_aired Date,"
            + "  last_aired Date,"
            + "  PRIMARY KEY(series_id, season_id)"
            + ")", TxMode.NONE).execute()
    ).join().getStatus().expectSuccess("Can't create table seasons");

    retryCtx.supplyResult(session -> session.createQuery(""
            + "CREATE TABLE episodes ("
            + "  series_id UInt64,"
            + "  season_id UInt64,"
            + "  episode_id UInt64,"
            + "  title Text,"
            + "  air_date Date,"
            + "  PRIMARY KEY(series_id, season_id, episode_id)"
            + ")", TxMode.NONE).execute()
    ).join().getStatus().expectSuccess("Can't create table episodes");
}
```

Adding data

Add data to the created tables using the `UPSERT` statement in YQL. A data update request is sent to the server as a single request with transaction auto-commit mode enabled.

To execute YQL queries, use the `QuerySession.createQuery()` method. It creates a new `QueryStream` object, which allows to execute a query and subscribe for receiving response data from the server. Because write requests don't expect any results, the `QueryStream.execute()` method is used without parameters; it just executes the request and waits for the stream to complete. Code snippet demonstrating this logic:

```
private void upsertSimple() {
    String query
            = "UPSERT INTO episodes (series_id, season_id, episode_id, title) "
            + "VALUES (2, 6, 1, \"TBD\");";

    // Executes data query with specified transaction control settings.
    retryCtx.supplyResult(session -> session.createQuery(query, TxMode.SERIALIZABLE_RW).execute())
        .join().getValue();
```

Retrieving data

Retrieve data using a `SELECT` statement in YQL. Handle the retrieved data selection in the app.

Direct usage of the `QueryStream` class to obtain results may not always be convenient, as it involves receiving data from the server asynchronously in the callback of the `QueryStream.execute()` method. If the number of expected rows in a result is not too large, it is more practical to use the `QueryReader` helper from the SDK, which first reads all response parts from the stream and provides them all to the user in an ordered form.

```
private void selectSimple() {
    String query
            = "SELECT series_id, title, release_date "
            + "FROM series WHERE series_id = 1;";

    // Executes data query with specified transaction control settings.
    QueryReader result = retryCtx.supplyResult(
            session -> QueryReader.readFrom(session.createQuery(query, TxMode.SERIALIZABLE_RW))
    ).join().getValue();

    logger.info("--[ SelectSimple ]--");

    ResultSetReader rs = result.getResultSet(0);
    while (rs.next()) {
        logger.info("read series with id {}, title {} and release_date {}",
                rs.getColumn("series_id").getUint64(),
                rs.getColumn("title").getText(),
                rs.getColumn("release_date").getDate()
        );
```

```
    }
}
```

As a result of the query, an object of the `QueryReader` class is generated. It may contain several sets obtained using the `getResultSet( <index> )` method. Since there was only one `SELECT` statement in the query, the result contains only one selection indexed as `0`. The given code snippet prints the following text to the console at startup:

```
12:06:36.548 INFO  App - --[ SelectSimple ]--
12:06:36.559 INFO  App - read series with id 1, title IT Crowd and release_date 2006-02-03
```

Parameterized queries

Query data using parameters. This query execution method is preferable because it allows the server to reuse the query execution plan for subsequent calls and protects against vulnerabilities such as SQL injection.

The code snippet below demonstrates how to use parameterized queries and the `Params` class to construct parameters and pass them to the `QuerySession.createQuery` method.

```
private void selectWithParams(long seriesID, long seasonID) {
    String query
            = "DECLARE $seriesId AS Uint64; "
            + "DECLARE $seasonId AS Uint64; "
            + "SELECT sa.title AS season_title, sr.title AS series_title "
            + "FROM seasons AS sa INNER JOIN series AS sr ON sa.series_id = sr.series_id "
            + "WHERE sa.series_id = $seriesId AND sa.season_id = $seasonId";

    // Begin new transaction with SerializableRW mode
    TxControl txControl = TxControl.serializableRw().setCommitTx(true);

    // Type of parameter values should be exactly the same as in DECLARE statements.
    Params params = Params.of(
            "$seriesId", PrimitiveValue.newUint64(seriesID),
            "$seasonId", PrimitiveValue.newUint64(seasonID)
    );

    DataQueryResult result = retryCtx.supplyResult(session -> session.executeDataQuery(query, txControl, para
ms))
            .join().getValue();

    logger.info("--[ SelectWithParams ] -- ");

    ResultSetReader rs = result.getResultSet(0);
    while (rs.next()) {
        logger.info("read season with title {} for series {}",
                rs.getColumn("season_title").getText(),
                rs.getColumn("series_title").getText()
        );
    }
}
```

Streaming data reads

If the expected row count in the response is large, asynchronous reading is the preferred way to process them. In this case, the `SessionRetryContext` is still used for retries because processing response parts can be interrupted at any moment, requiring the entire execution process to restart.

```
private void asyncSelectRead(long seriesID, long seasonID) {
    String query
            = "DECLARE $seriesId AS Uint64; "
            + "DECLARE $seasonId AS Uint64; "
            + "SELECT ep.title AS episode_title, sa.title AS season_title, sr.title AS series_title "
            + "FROM episodes AS ep "
            + "JOIN seasons AS sa ON sa.season_id = ep.season_id "
            + "JOIN series AS sr ON sr.series_id = sa.series_id "
            + "WHERE sa.series_id = $seriesId AND sa.season_id = $seasonId;";

    // Type of parameter values should be exactly the same as in DECLARE statements.
    Params params = Params.of(
            "$seriesId", PrimitiveValue.newUint64(seriesID),
            "$seasonId", PrimitiveValue.newUint64(seasonID)
    );

    logger.info("--[ ExecuteAsyncQueryWithParams ]--");
    retryCtx.supplyResult(session -> {
        QueryStream asyncQuery = session.createQuery(query, TxMode.SNAPSHOT_RO, params);
        return asyncQuery.execute(part -> {
            ResultSetReader rs = part.getResultSetReader();
            logger.info("read {} rows of result set {}", rs.getRowCount(), part.getResultSetIndex());
            while (rs.next()) {
                logger.info("read episode {} of {} for {}",
                        rs.getColumn("episode_title").getText(),
                        rs.getColumn("season_title").getText(),
                        rs.getColumn("series_title").getText()
                );
            }
        });
    }).join().getStatus().expectSuccess("execute query problem");
}
```

Multistep transactions

Multiple statements can be executed within a single multistep transaction. Client-side code can run between query steps. Using a transaction ensures that queries executed in its context are consistent with each other.

To ensure interoperability between the transactions and the retry context, each transaction must wholly execute inside the callback passed to `SessionRetryContext`. The callback must return after the entire transaction is completed.

Code template for running complex transactions inside `SessionRetryContext`

```
private void multiStepTransaction(long seriesID, long seasonID) {
    retryCtx.supplyStatus(session -> {
        QueryTransaction transaction = session.createNewTransaction(TxMode.SNAPSHOT_RO);

        //...

        return CompletableFuture.completedFuture(Status.SUCCESS);
    }).join().expectSuccess("multistep transaction problem");
}
```

The first step is to prepare and execute the first query:

```
    String query1
            = "DECLARE $seriesId AS Uint64; "
            + "DECLARE $seasonId AS Uint64; "
            + "SELECT MIN(first_aired) AS from_date FROM seasons "
            + "WHERE series_id = $seriesId AND season_id = $seasonId;";

    // Execute the first query to start a new transaction
    QueryReader res1 = QueryReader.readFrom(transaction.createQuery(query1, Params.of(
            "$seriesId", PrimitiveValue.newUint64(seriesID),
            "$seasonId", PrimitiveValue.newUint64(seasonID)
    ))).join().getValue();
```

After that, we can process the resulting data on the client side:

```
    // Perform some client logic on returned values
    ResultSetReader resultSet = res1.getResultSet(0);
    if (!resultSet.next()) {
        throw new RuntimeException("not found first_aired");
    }
    LocalDate fromDate = resultSet.getColumn("from_date").getDate();
    LocalDate toDate = fromDate.plusDays(15);
```

And get the current `transaction id` to continue processing within the same transaction:

```
    // Get active transaction id
    logger.info("started new transaction {}", transaction.getId());
```

The next step is to create the next query that uses the results of code execution on the client side:

```
    // Construct next query based on the results of client logic
    String query2
            = "DECLARE $seriesId AS Uint64;"
            + "DECLARE $fromDate AS Date;"
            + "DECLARE $toDate AS Date;"
            + "SELECT season_id, episode_id, title, air_date FROM episodes "
            + "WHERE series_id = $seriesId AND air_date >= $fromDate AND air_date <= $toDate;";

    // Execute the second query and commit
    QueryReader res2 = QueryReader.readFrom(transaction.createQueryWithCommit(query2, Params.of(
        "$seriesId", PrimitiveValue.newUint64(seriesID),
        "$fromDate", PrimitiveValue.newDate(fromDate),
        "$toDate", PrimitiveValue.newDate(toDate)
    ))).join().getValue();

    logger.info("--[ MultiStep ]--");
    ResultSetReader rs = res2.getResultSet(0);
    while (rs.next()) {
        logger.info("read episode {} with air date {}",
                rs.getColumn("title").getText(),
                rs.getColumn("air_date").getDate()
        );
    }
```

The given code snippets output the following text to the console at startup:

```
12:06:36.850 INFO  App - --[ MultiStep ]--
12:06:36.851 INFO  App - read episode Grow Fast or Die Slow with air date 2018-03-25
12:06:36.851 INFO  App - read episode Reorientation with air date 2018-04-01
12:06:36.851 INFO  App - read episode Chief Operating Officer with air date 2018-04-08
```

Managing transactions

Transactions are managed through TCL `Begin` and `Commit` calls.

In most cases, instead of explicitly using `Begin` and `Commit` calls, it's better to use transaction control parameters in execute calls. This allows to avoid additional requests to YDB server and thus run queries more efficiently.

Code snippet for `beginTransaction()` and `transaction.commit()` calls:

```
private void tclTransaction() {
    retryCtx.supplyResult(session -> {
        QueryTransaction transaction = session.beginTransaction(TxMode.SERIALIZABLE_RW)
            .join().getValue();

        String query
                = "DECLARE $airDate AS Date; "
                + "UPDATE episodes SET air_date = $airDate WHERE title = \"TBD\";";

        Params params = Params.of("$airDate", PrimitiveValue.newDate(Instant.now()));

        // Execute data query.
        // Transaction control settings continues active transaction (tx)
        QueryReader reader = QueryReader.readFrom(transaction.createQuery(query, params))
            .join().getValue();

        logger.info("get transaction {}", transaction.getId());

        // Commit active transaction (tx)
        return transaction.commit();
    }).join().getStatus().expectSuccess("tcl transaction problem");
}
```

Example app in JavaScript

This page provides a detailed description of the example application code that uses the YDB JavaScript SDK.

Usage examples are available in the ydb-js-sdk repository, and additional real-world scenario examples can be found in the ydb-js-examples repository.

Initializing a database connection

To interact with YDB, create instances of the driver, client, and session:

- The YDB driver facilitates interaction between the app and YDB nodes at the transport layer. It must be initialized before creating a client or session and must persist throughout the YDB access lifecycle.
- The YDB client operates on top of the YDB driver and enables the handling of entities and transactions.
- The YDB session, which is part of the YDB client context, contains information about executed transactions and prepared queries.

To work with YDB, you need to create a driver instance and a query client.

Installing the required packages:

```
npm install @ydbjs/core @ydbjs/query
```

App code snippet for driver initialization:

Using connectionString

```
import { Driver } from '@ydbjs/core'
import { query } from '@ydbjs/query'

const connectionString = 'grpc://localhost:2136/local'
const driver = new Driver(connectionString)
await driver.ready()

const sql = query(driver)
```

Using authentication

```
import { Driver } from '@ydbjs/core'
import { query } from '@ydbjs/query'
import { AnonymousCredentialsProvider } from '@ydbjs/auth'

const connectionString = 'grpc://localhost:2136/local'
const driver = new Driver(connectionString, {
  credentialsProvider: new AnonymousCredentialsProvider(),
})
await driver.ready()

const sql = query(driver)
```

Creating tables

Create tables to be used in operations on a test app. This step results in the creation of database tables for the series directory data model:

- `Series`
- `Seasons`
- `Episodes`

After the tables are created, a method for retrieving information about data schema objects is called, and the result of its execution is displayed.

```
await sql`
    CREATE TABLE series (
        series_id Uint64,
        title Text,
        series_info Text,
        release_date Date,
        PRIMARY KEY (series_id)
    )
`

await sql`
    CREATE TABLE seasons (
        series_id Uint64,
        season_id Uint64,
        title Text,
        first_aired Date,
        last_aired Date,
        PRIMARY KEY (series_id, season_id)
    )
`

await sql`
    CREATE TABLE episodes (
        series_id Uint64,
        season_id Uint64,
        episode_id Uint64,
```

```
        title Text,
        air_date Date,
        PRIMARY KEY (series_id, season_id, episode_id)
    )
`
```

Adding data

Add data to the created tables using the `UPSERT` statement in YQL. A data update request is sent to the server as a single request with transaction auto-commit mode enabled.

Code snippet for data insert/update:

```
await sql`
    UPSERT INTO episodes (series_id, season_id, episode_id, title)
    VALUES (2, 6, 1, "TBD")
`
```

For inserting data using parameters:

```
import { Uint64, Text, Date as YdbDate } from '@ydbjs/value/primitive'

const data = [
  {
    series_id: new Uint64(1n),
    title: new Text('IT Crowd'),
    series_info: new Text('British sitcom'),
    release_date: new YdbDate(new Date('2006-02-03')),
  },
]

await sql`INSERT INTO series SELECT * FROM AS_TABLE(${data})`
```

Retrieving data

Retrieve data using a `SELECT` statement in YQL. Handle the retrieved data selection in the app.

The tagged template syntax is used for executing YQL queries. The result is an array of result sets (YDB supports multiple result sets per query).

```
const resultSets = await sql`
    SELECT series_id, title, release_date
    FROM series
    WHERE series_id = 1
`

// resultSets[0] contains the first result set
const [firstResultSet] = resultSets
console.log(firstResultSet)
// [ { series_id: 1n, title: 'IT Crowd', release_date: 2006-02-03T00:00:00.000Z } ]
```

For queries with multiple result sets:

```
type Result = [[{ id: bigint }], [{ count: bigint }]]
const [rows, [{ count }]] = await sql<Result>`
    SELECT series_id as id FROM series;
    SELECT COUNT(*) as count FROM series;
`
```

Parameterized queries

Query data using parameters. This query execution method is preferable because it allows the server to reuse the query execution plan for subsequent calls and protects against vulnerabilities such as SQL injection.

The SDK automatically binds parameters through interpolation in template strings. Native JavaScript types, YDB value classes, arrays, and objects are all supported.

```
const seriesId = 1n
const title = 'IT Crowd'

const resultSets = await sql`
    SELECT series_id, title, release_date
    FROM series
    WHERE series_id = ${seriesId} AND title = ${title}
`
```

For named parameters and custom types:

```
import { Uint64 } from '@ydbjs/value/primitive'

const id = new Uint64(1n)
const resultSets = await sql`SELECT * FROM series WHERE series_id = $id`.parameter('id', id)
```

Queries are executed with streaming data transfer by default. For working with large volumes of data, use standard queries:

```
const resultSets = await sql`
    SELECT series_id, season_id, title, first_aired
    FROM seasons
    WHERE series_id IN (1, 2)
    ORDER BY season_id
`

for (const row of resultSets[0]) {
  console.log(`Season ${row.season_id}: ${row.title}`)
}
```

Managing transactions

Transactions are managed through TCL `Begin` and `Commit` calls.

In most cases, instead of explicitly using `Begin` and `Commit` calls, it's better to use transaction control parameters in execute calls. This allows to avoid additional requests to YDB server and thus run queries more efficiently.

To execute queries within a transaction, use the `sql.begin()` or `sql.transaction()` methods:

begin() — serializable read-write

```
const result = await sql.begin(async (tx) => {
  await tx`
      UPDATE episodes
      SET air_date = CurrentUtcDate()
      WHERE series_id = 2 AND season_id = 6 AND episode_id = 1
  `
  return await tx`SELECT * FROM episodes WHERE series_id = 2`
})
```

begin() with isolation settings

```
await sql.begin({ isolation: 'snapshotReadOnly', idempotent: true }, async (tx) => {
  return await tx`SELECT COUNT(*) FROM series`
})
```

Error handling

For more information about error handling, see Error handling in the API.

The `YDBError` class is used for error handling:

```
import { YDBError } from '@ydbjs/error'

try {
  await sql`SELECT * FROM non_existent_table`
} catch (error) {
  if (error instanceof YDBError) {
    console.error('YDB Error:', error.message)
  }
}
```

Additional features

Query options

```
import { StatsMode } from '@ydbjs/api/query'

await sql`SELECT * FROM series`
  .isolation('onlineReadOnly', { allowInconsistentReads: true })
  .idempotent(true)
  .timeout(5000)
  .withStats(StatsMode.FULL)
```

Dynamic identifiers

For dynamic table and column names, use the `identifier` method:

```
const tableName = 'series'
await sql`SELECT * FROM ${sql.identifier(tableName)}`
```

Closing the driver

Always close the driver when finished:

```
driver.close()
```

Example app in PHP

This page contains a detailed description of the code of a test app that is available as part of the YDB PHP SDK.

Initializing a database connection

To interact with YDB, create instances of the driver, client, and session:

- The YDB driver facilitates interaction between the app and YDB nodes at the transport layer. It must be initialized before creating a client or session and must persist throughout the YDB access lifecycle.
- The YDB client operates on top of the YDB driver and enables the handling of entities and transactions.
- The YDB session, which is part of the YDB client context, contains information about executed transactions and prepared queries.

App code snippet for driver initialization:

```
<?php

use YdbPlatform\Ydb\Ydb;

$config = [
    // Database path
    'database'    => '/ru-central1/b1glxxxxxxxxxxxxxxx/etn0xxxxxxxxxxxxxxx',

    // Database endpoint
    'endpoint'    => 'ydb.serverless.yandexcloud.net:2135',

    // Auto discovery (dedicated server only)
    'discovery'   => false,

    // IAM config
    'iam_config'  => [
        // 'root_cert_file' => './CA.pem',  Root CA file (uncomment for dedicated server only)
    ],

    'credentials' => new AccessTokenAuthentication('AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA') // use from ref
erence/ydb-sdk/auth
];

$ydb = new Ydb($config);
```

Creating tables

Create tables to be used in operations on a test app. This step results in the creation of database tables for the series directory data model:

- `Series`
- `Seasons`
- `Episodes`

After the tables are created, a method for retrieving information about data schema objects is called, and the result of its execution is displayed.

To create tables, use the `session->createTable()` method:

```
protected function createTabels()
{
    $this->ydb->table()->retrySession(function (Session $session) {

        $session->createTable(
            'series',
            YdbTable::make()
                ->addColumn('series_id', 'UINT64')
                ->addColumn('title', 'UTF8')
                ->addColumn('series_info', 'UTF8')
                ->addColumn('release_date', 'UINT64')
                ->primaryKey('series_id')
        );

    }, true);

    $this->print('Table `series` has been created.');

    $this->ydb->table()->retrySession(function (Session $session) {

        $session->createTable(
            'seasons',
            YdbTable::make()
                ->addColumn('series_id', 'UINT64')
                ->addColumn('season_id', 'UINT64')
                ->addColumn('title', 'UTF8')
                ->addColumn('first_aired', 'UINT64')
                ->addColumn('last_aired', 'UINT64')
                ->primaryKey(['series_id', 'season_id'])
        );

    }, true);

    $this->print('Table `seasons` has been created.');
```

```
    $this->ydb->table()->retrySession(function (Session $session) {

        $session->createTable(
            'episodes',
            YdbTable::make()
                ->addColumn('series_id', 'UINT64')
                ->addColumn('season_id', 'UINT64')
                ->addColumn('episode_id', 'UINT64')
                ->addColumn('title', 'UTF8')
                ->addColumn('air_date', 'UINT64')
                ->primaryKey(['series_id', 'season_id', 'episode_id'])
        );

    }, true);

    $this->print('Table `episodes` has been created.');
}
```

You can use the `session->describeTable()` method to output information about the table structure and make sure that it was properly created:

```
protected function describeTable($table)
{
    $data = $ydb->table()->retrySession(function (Session $session) use ($table) {

        return $session->describeTable($table);

    }, true);

    $columns = [];

    foreach ($data['columns'] as $column) {
        if (isset($column['type']['optionalType']['item']['typeId'])) {
            $columns[] = [
                'Name' => $column['name'],
                'Type' => $column['type']['optionalType']['item']['typeId'],
            ];
        }
    }

    print('Table `' . $table . '`');
    print_r($columns);
    print('');
    print('Primary key: ' . implode(', ', (array)$data['primaryKey']));
}
```

Adding data

Add data to the created tables using the `UPSERT` statement in YQL. A data update request is sent to the server as a single request with transaction auto-commit mode enabled.

Code snippet for data insert/update:

```
protected function upsertSimple()
{
    $ydb->table()->retryTransaction(function (Session $session) {
        $session->query('
        DECLARE $series_id AS Uint64;
        DECLARE $season_id AS Uint64;
        DECLARE $episode_id AS Uint64;
        DECLARE $title AS Utf8;
            UPSERT INTO episodes (series_id, season_id, episode_id, title)
            VALUES ($series_id, $season_id, $episode_id, $title);', [
                '$series_id' => (new Uint64Type(2))->toTypedValue(),
                '$season_id' => (new Uint64Type(6))->toTypedValue(),
                '$episode_id' => (new Uint64Type(1))->toTypedValue(),
                '$title' => (new Utf8Type('TBD'))->toTypedValue(),
            ]);
    }, true);

    print('Finished.');
}
```

Retrieving data

Retrieve data using a `SELECT` statement in YQL. Handle the retrieved data selection in the app.

To execute YQL queries, use the `session->query()` method.

```
$result = $ydb->table()->retryTransaction(function (Session $session) {
        return $session->query('
        DECLARE $seriesID AS Uint64;
        $format = DateTime::Format("%Y-%m-%d");
        SELECT
            series_id,
            title,
            $format(DateTime::FromSeconds(CAST(release_date AS Uint32))) AS release_date
        FROM series
        WHERE series_id = $seriesID;', [
```

```
            '$seriesID' => (new Uint64Type(1))->toTypedValue()
        ]);
}, true, $params);

print_r($result->rows());
```

Parameterized queries

Query data using parameters. This query execution method is preferable because it allows the server to reuse the query execution plan for subsequent calls and protects against vulnerabilities such as SQL injection.

Here's a code sample that shows how to use prepared queries.

```
protected function selectPrepared($series_id, $season_id, $episode_id)
{
    $result = $ydb->table()->retryTransaction(function (Session $session) use ($series_id, $season_id, $episo
de_id) {

        $prepared_query = $session->prepare('
        DECLARE $series_id AS Uint64;
        DECLARE $season_id AS Uint64;
        DECLARE $episode_id AS Uint64;

        $format = DateTime::Format("%Y-%m-%d");
        SELECT
            title AS `Episode title`,
            $format(DateTime::FromSeconds(CAST(air_date AS Uint32))) AS `Air date`
        FROM episodes
        WHERE series_id = $series_id AND season_id = $season_id AND episode_id = $episode_id;');

        return $prepared_query->execute(compact(
            'series_id',
            'season_id',
            'episode_id'
        ));
    },true);

    $this->print($result->rows());
}
```

Example app in Python

This page contains a detailed description of the code of a test app that is available as part of the YDB Python SDK.

Downloading and starting

The following execution scenario is based on git and Python3. Be sure to install the YDB Python SDK.

Create a working directory and use it to run from the command line the command to clone the GitHub repository and install the necessary Python packages:

```
git clone https://github.com/ydb-platform/ydb-python-sdk.git
python3 -m pip install iso8601
```

Next, from the same working directory, run the following command to start the test app:

Local Docker

To connect to a locally deployed YDB database according to the Docker use case, run the following command in the default configuration:

```
YDB_ANONYMOUS_CREDENTIALS=1 \
python3 ydb-python-sdk/examples/basic_example_v1/ -e grpc://localhost:2136 -d /local
```

Any database

To run the example against any available YDB database, the endpoint and the database path need to be provide.

If authentication is enabled for the database, the authentication mode needs to be chosen and credentials (a token or a username/password pair) need to be provided.

Run the command as follows:

```
<auth_mode_var>="<auth_mode_value>" \
python3 ydb-python-sdk/examples/basic_example_v1/ -e <endpoint> -d <database>
```

where

- `<endpoint>`: The endpoint.
- `<database>`: The database path.
- `<auth_mode_var>`: The environment variable that determines the authentication mode.
- `<auth_mode_value>` is the authentication parameter value for the selected mode.

For example:

```
YDB_ACCESS_TOKEN_CREDENTIALS="t1.9euelZqOnJuJlc..." \
python3 ydb-python-sdk/examples/basic_example_v1/ -e grpcs://ydb.example.com:2135 -d /path/db )
```

Initializing a database connection

To interact with YDB, create instances of the driver, client, and session:

- The YDB driver facilitates interaction between the app and YDB nodes at the transport layer. It must be initialized before creating a client or session and must persist throughout the YDB access lifecycle.
- The YDB client operates on top of the YDB driver and enables the handling of entities and transactions.
- The YDB session, which is part of the YDB client context, contains information about executed transactions and prepared queries.

App code snippet for driver initialization:

Synchronous

```
def run(endpoint, database):
    driver_config = ydb.DriverConfig(
        endpoint, database, credentials=ydb.credentials_from_env_variables(),
        root_certificates=ydb.load_ydb_root_certificate(),
    )
    with ydb.Driver(driver_config) as driver:
        try:
            driver.wait(timeout=5)
        except TimeoutError:
            print("Connect failed to YDB")
            print("Last reported errors by discovery:")
            print(driver.discovery_debug_details())
            exit(1)
```

Asynchronous

```
async def run(endpoint, database):
    driver_config = ydb.DriverConfig(
        endpoint, database, credentials=ydb.credentials_from_env_variables(),
        root_certificates=ydb.load_ydb_root_certificate(),
    )
    async with ydb.aio.Driver(driver_config) as driver:
        try:
            await driver.wait(timeout=5)
```

```
        except TimeoutError:
            print("Connect failed to YDB")
            print("Last reported errors by discovery:")
            print(driver.discovery_debug_details())
            exit(1)
```

App code snippet for session pool initialization:

Synchronous

```
with ydb.QuerySessionPool(driver) as pool:
    pass  # operations with pool here
```

Asynchronous

```
async with ydb.aio.QuerySessionPool(driver) as pool:
    pass  # operations with pool here
```

Executing queries

YDB Python SDK supports queries described by YQL syntax.
There are two primary methods for executing queries, each with different properties and use cases:

- `pool.execute_with_retries`:
  - Buffers the entire result set in client memory.
  - Automatically retries execution in case of retriable issues.
  - Does not allow specifying a transaction execution mode.
  - Recommended for one-off queries that are expected to produce small result sets.
- `tx.execute`:
  - Returns an iterator over the query results, allowing processing of results that may not fit into client memory.
  - Retries must be handled manually via `pool.retry_operation_sync`.
  - Allows specifying a transaction execution mode.
  - Recommended for scenarios where `pool.execute_with_retries` is insufficient.

Creating tables

Create tables to be used in operations on a test app. This step results in the creation of database tables for the series directory data model:

- `Series`
- `Seasons`
- `Episodes`

After the tables are created, a method for retrieving information about data schema objects is called, and the result of its execution is displayed.

To execute `CREATE TABLE` queries, use the `pool.execute_with_retries()` method:

Synchronous

```
def create_tables(pool: ydb.QuerySessionPool):
    print("\nCreating table series...")
    pool.execute_with_retries(
        """
        CREATE TABLE `series` (
            `series_id` Int64,
            `title` Utf8,
            `series_info` Utf8,
            `release_date` Date,
            PRIMARY KEY (`series_id`)
        )
        """
    )

    print("\nCreating table seasons...")
    pool.execute_with_retries(
        """
        CREATE TABLE `seasons` (
            `series_id` Int64,
            `season_id` Int64,
            `title` Utf8,
            `first_aired` Date,
            `last_aired` Date,
            PRIMARY KEY (`series_id`, `season_id`)
        )
        """
    )

    print("\nCreating table episodes...")
    pool.execute_with_retries(
        """
        CREATE TABLE `episodes` (
            `series_id` Int64,
            `season_id` Int64,
            `episode_id` Int64,
            `title` Utf8,
```

```
            `air_date` Date,
            PRIMARY KEY (`series_id`, `season_id`, `episode_id`)
        )
        """
    )
```

Asynchronous

```
async def create_tables(pool: ydb.aio.QuerySessionPool):
    print("\nCreating table series...")
    await pool.execute_with_retries(
        """
        CREATE TABLE `series` (
            `series_id` Int64,
            `title` Utf8,
            `series_info` Utf8,
            `release_date` Date,
            PRIMARY KEY (`series_id`)
        )
        """
    )

    print("\nCreating table seasons...")
    await pool.execute_with_retries(
        """
        CREATE TABLE `seasons` (
            `series_id` Int64,
            `season_id` Int64,
            `title` Utf8,
            `first_aired` Date,
            `last_aired` Date,
            PRIMARY KEY (`series_id`, `season_id`)
        )
        """
    )

    print("\nCreating table episodes...")
    await pool.execute_with_retries(
        """
        CREATE TABLE `episodes` (
            `series_id` Int64,
            `season_id` Int64,
            `episode_id` Int64,
            `title` Utf8,
            `air_date` Date,
            PRIMARY KEY (`series_id`, `season_id`, `episode_id`)
        )
        """
    )
```

Adding data

Add data to the created tables using the `UPSERT` statement in YQL. A data update request is sent to the server as a single request with transaction auto-commit mode enabled.

Code snippet for data insert/update:

Synchronous

```
def upsert_simple(pool: ydb.QuerySessionPool):
    print("\nPerforming UPSERT into episodes...")
    pool.execute_with_retries(
        """
        UPSERT INTO episodes (series_id, season_id, episode_id, title) VALUES (2, 6, 1, "TBD");
        """
    )
```

Asynchronous

```
async def upsert_simple(pool: ydb.aio.QuerySessionPool):
    print("\nPerforming UPSERT into episodes...")
    await pool.execute_with_retries(
        """
        UPSERT INTO episodes (series_id, season_id, episode_id, title) VALUES (2, 6, 1, "TBD");
        """
    )
```

Retrieving data

Retrieve data using a `SELECT` statement in YQL. Handle the retrieved data selection in the app.

To execute YQL queries, the `pool.execute_with_retries()` method is often sufficient.

Synchronous

```
def select_simple(pool: ydb.QuerySessionPool):
    print("\nCheck series table...")
```

```
    result_sets = pool.execute_with_retries(
        """
        SELECT
            series_id,
            title,
            release_date
        FROM series
        WHERE series_id = 1;
        """,
    )
    first_set = result_sets[0]
    for row in first_set.rows:
        print(
            "series, id: ",
            row.series_id,
            ", title: ",
            row.title,
            ", release date: ",
            row.release_date,
        )
    return first_set
```

Asynchronous

```
async def select_simple(pool: ydb.aio.QuerySessionPool):
    print("\nCheck series table...")
    result_sets = await pool.execute_with_retries(
        """
        SELECT
            series_id,
            title,
            release_date
        FROM series
        WHERE series_id = 1;
        """,
    )
    first_set = result_sets[0]
    for row in first_set.rows:
        print(
            "series, id: ",
            row.series_id,
            ", title: ",
            row.title,
            ", release date: ",
            row.release_date,
        )
    return first_set
```

As the result of executing the query, a list of `result_set` is returned, iterating on which the text is output to the console:

```
> SelectSimple:
series, Id: 1, title: IT Crowd, Release date: 2006-02-03
```

Parameterized queries

For parameterized query execution, `pool.execute_with_retries()` and `tx.execute()` behave similarly. To execute parameterized queries, you need to pass a dictionary with parameters to one of these functions, where each key is the parameter name, and the value can be one of the following:

1. A value of a basic Python type
2. A tuple containing the value and its type
3. A special type, `ydb.TypedValue(value=value, value_type=value_type)`

If you specify a value without an explicit type, the conversion takes place according to the following rules:

Python type	YDB type
`int`	`ydb.PrimitiveType.Int64`
`float`	`ydb.PrimitiveType.Double`
`str`	`ydb.PrimitiveType.Utf8`
`bytes`	`ydb.PrimitiveType.String`
`bool`	`ydb.PrimitiveType.Bool`
`list`	`ydb.ListType`
`dict`	`ydb.DictType`

Warning

Automatic conversion of lists and dictionaries is possible only if the structures are homogeneous. The type of nested values will be determined recursively according to the rules explained above. In case of using heterogeneous structures, requests will raise `TypeError`.

A code snippet demonstrating the parameterized query execution:

Synchronous

```
def select_with_parameters(pool: ydb.QuerySessionPool, series_id, season_id, episode_id):
    result_sets = pool.execute_with_retries(
        """
        DECLARE $seriesId AS Int64;
        DECLARE $seasonId AS Int64;
        DECLARE $episodeId AS Int64;

        SELECT
            title,
            air_date
        FROM episodes
        WHERE series_id = $seriesId AND season_id = $seasonId AND episode_id = $episodeId;
        """,
        {
            "$seriesId": series_id,  # data type could be defined implicitly
            "$seasonId": (season_id, ydb.PrimitiveType.Int64),  # could be defined via a tuple
            "$episodeId": ydb.TypedValue(episode_id, ydb.PrimitiveType.Int64),  # could be defined via a spec
ial class
        },
    )

    print("\n> select_with_parameters:")
    first_set = result_sets[0]
    for row in first_set.rows:
        print("episode title:", row.title, ", air date:", row.air_date)

    return first_set
```

Asynchronous

```
async def select_with_parameters(pool: ydb.aio.QuerySessionPool, series_id, season_id, episode_id):
    result_sets = await pool.execute_with_retries(
        """
        DECLARE $seriesId AS Int64;
        DECLARE $seasonId AS Int64;
        DECLARE $episodeId AS Int64;

        SELECT
            title,
            air_date
        FROM episodes
        WHERE series_id = $seriesId AND season_id = $seasonId AND episode_id = $episodeId;
        """,
        {
            "$seriesId": series_id,  # could be defined implicitly
            "$seasonId": (season_id, ydb.PrimitiveType.Int64),  # could be defined via a tuple
            "$episodeId": ydb.TypedValue(episode_id, ydb.PrimitiveType.Int64),  # could be defined via a spec
ial class
        },
    )

    print("\n> select_with_parameters:")
    first_set = result_sets[0]
    for row in first_set.rows:
        print("episode title:", row.title, ", air date:", row.air_date)

    return first_set
```

The code snippet above outputs the following text to the console:

```
> select_prepared_transaction:
('episode title:', u'To Build a Better Beta', ', air date:', '2016-06-05')
```

Managing transactions

Transactions are managed through TCL `Begin` and `Commit` calls.

In most cases, instead of explicitly using `Begin` and `Commit` calls, it's better to use transaction control parameters in execute calls. This allows to avoid additional requests to YDB server and thus run queries more efficiently.

The `session.transaction().execute()` method can also be used to execute YQL queries. Unlike `pool.execute_with_retries`, this method allows explicit control of transaction execution by configuring the desired transaction mode using the `TxControl` class.

Available transaction modes:

- `ydb.QuerySerializableReadWrite()` (default);
- `ydb.QueryOnlineReadOnly(allow_inconsistent_reads=False)`;
- `ydb.QuerySnapshotReadOnly()`;
- `ydb.QueryStaleReadOnly()`.

For more information about transaction modes, see Transaction Modes.

The result of executing `tx.execute()` is an iterator. This iterator allows you to read result rows without loading the entire result set into memory. However, the iterator must be read to the end after each request to correctly maintain the transaction state on the YDB

server side. If this is not done, write queries could not be applied on the YDB server side. For convenience, the result of the `tx.execute()` function can be used as a context manager that automatically iterates to the end upon exit.

Synchronous

```
with tx.execute(query) as _:
    pass
```

Asynchronous

```
async with await tx.execute(query) as _:
    pass
```

The code snippet below demonstrates the explicit use of `transaction().begin()` and `tx.commit()`:

Synchronous

```
def explicit_transaction_control(pool: ydb.QuerySessionPool, series_id, season_id, episode_id):
    def callee(session: ydb.QuerySession):
        query = """
        DECLARE $seriesId AS Int64;
        DECLARE $seasonId AS Int64;
        DECLARE $episodeId AS Int64;

        UPDATE episodes
        SET air_date = CurrentUtcDate()
        WHERE series_id = $seriesId AND season_id = $seasonId AND episode_id = $episodeId;
        """

        # Get newly created transaction id
        tx = session.transaction().begin()

        # Execute data query.
        # Transaction control settings continues active transaction (tx)
        with tx.execute(
            query,
            {
                "$seriesId": (series_id, ydb.PrimitiveType.Int64),
                "$seasonId": (season_id, ydb.PrimitiveType.Int64),
                "$episodeId": (episode_id, ydb.PrimitiveType.Int64),
            },
        ) as _:
            pass

        print("\n> explicit TCL call")

        # Commit active transaction(tx)
        tx.commit()

    return pool.retry_operation_sync(callee)
```

Asynchronous

```
async def explicit_transaction_control(
    pool: ydb.aio.QuerySessionPool, series_id, season_id, episode_id
):
    async def callee(session: ydb.aio.QuerySession):
        query = """
        DECLARE $seriesId AS Int64;
        DECLARE $seasonId AS Int64;
        DECLARE $episodeId AS Int64;

        UPDATE episodes
        SET air_date = CurrentUtcDate()
        WHERE series_id = $seriesId AND season_id = $seasonId AND episode_id = $episodeId;
        """

        # Get newly created transaction id
        tx = await session.transaction().begin()

        # Execute data query.
        # Transaction control settings continues active transaction (tx)
        async with await tx.execute(
            query,
            {
                "$seriesId": (series_id, ydb.PrimitiveType.Int64),
                "$seasonId": (season_id, ydb.PrimitiveType.Int64),
                "$episodeId": (episode_id, ydb.PrimitiveType.Int64),
            },
        ) as _:
            pass

        print("\n> explicit TCL call")

        # Commit active transaction(tx)
        await tx.commit()
```

```
    return await pool.retry_operation_async(callee)
```

However, a transaction can be opened implicitly with the first request and can be committed automatically by setting the `commit_tx=True` flag in arguments. Implicit transaction management is preferable because it requires fewer server calls.

Iterating over query results

If a `SELECT` query is expected to return a potentially large number of rows, it is recommended to use the `tx.execute` method instead of `pool.execute_with_retries` to avoid excessive memory consumption on the client side. Instead of buffering the entire result set into memory, `tx.execute` returns an iterator for each top-level `SELECT` statement in the query.

Example of a `SELECT` with unlimited data and implicit transaction control:

Synchronous

```
def huge_select(pool: ydb.QuerySessionPool):
    def callee(session: ydb.QuerySession):
        query = """SELECT * from episodes;"""

        with session.transaction(ydb.QuerySnapshotReadOnly()).execute(
            query,
            commit_tx=True,
        ) as result_sets:
            print("\n> Huge SELECT call")
            for result_set in result_sets:
                for row in result_set.rows:
                    print("episode title:", row.title, ", air date:", row.air_date)

    return pool.retry_operation_sync(callee)
```

Asynchronous

```
async def huge_select(pool: ydb.aio.QuerySessionPool):
    async def callee(session: ydb.aio.QuerySession):
        query = """SELECT * from episodes;"""

        async with await session.transaction(ydb.QuerySnapshotReadOnly()).execute(
            query,
            commit_tx=True,
        ) as result_sets:
            print("\n> Huge SELECT call")
            async for result_set in result_sets:
                for row in result_set.rows:
                    print("episode title:", row.title, ", air date:", row.air_date)

    return await pool.retry_operation_async(callee)
```

Example app in Rust

This page provides a detailed description of the code for a test app that uses the YDB Rust SDK.

Downloading and starting

Git and Rust 1.85+ are required. For SDK installation, see Installing the YDB SDK.

Clone the repository:

```
git clone https://github.com/ydb-platform/ydb-rs-sdk.git
```

Start the example:

Local Docker

To connect to a locally deployed YDB database according to the Docker use case, run the following command in the default configuration:

```
export YDB_CONNECTION_STRING=grpc://localhost:2136/local
cd ydb-rs-sdk/ydb
cargo run --example basic
```

Any database

To run the example against any available YDB database, provide the endpoint and the database path.

If authentication is enabled, choose an authentication mode and set the corresponding environment variables.

```
export <auth_mode_var>="<auth_mode_value>"
export YDB_CONNECTION_STRING="<endpoint>/<database>"
cd ydb-rs-sdk/ydb
cargo run --example basic
```

where

- `<endpoint>`: The endpoint.
- `<database>`: The database path.
- `<auth_mode_var>`: The environment variable for the authentication mode.
- `<auth_mode_value>`: The credential value for the selected mode.

For example:

```
export YDB_ACCESS_TOKEN_CREDENTIALS="t1.9euelZqOnJuJlc..."
export YDB_CONNECTION_STRING="grpcs://ydb.example.com:2135/somepath/somelocation"
cd ydb-rs-sdk/ydb
cargo run --example basic
```

Initializing a database connection

To interact with YDB, create instances of the driver, client, and session:

- The YDB driver facilitates interaction between the app and YDB nodes at the transport layer. It must be initialized before creating a client or session and must persist throughout the YDB access lifecycle.
- The YDB client operates on top of the YDB driver and enables the handling of entities and transactions.
- The YDB session, which is part of the YDB client context, contains information about executed transactions and prepared queries.

Import the crate and open a client:

```
use ydb::{ClientBuilder, YdbResult};

#[tokio::main]
async fn main() -> YdbResult<()> {
    let connection_string = std::env::var("YDB_CONNECTION_STRING")
        .unwrap_or_else(|_| "grpc://localhost:2136/local".to_string());

    let client = ClientBuilder::new_from_connection_string(connection_string)?.client()?;
    client.wait().await?;

    let mut qc = client.query_client().clone_with_idempotent_operations(true);
    // ...
    Ok(())
}
```

`ClientBuilder::new_from_connection_string` accepts a YDB connection string (`grpc://host:port/database`).
`client.wait()` waits until the driver discovers cluster endpoints. `query_client()` is the entry point for the Query Service API.

By default, anonymous authentication is used for local Docker. For token auth, use `ClientBuilder::with_credentials` — see authentication recipes.

Query Service client

To run a single transactional SQL statement, use awaitable builders on `QueryClient`:

- `qc.exec(yql)` — statement with no result set (DDL, DML).

- `qc.query_row(yql)` — exactly one row.
- `qc.query(yql).await?` — streaming `QueryStream` for large results.

Automatic retries are enabled via `clone_with_idempotent_operations(true)` for idempotent reads and DDL.

Creating tables

Create tables to be used in operations on a test app. This step results in the creation of database tables for the series directory data model:

- `Series`
- `Seasons`
- `Episodes`

After the tables are created, a method for retrieving information about data schema objects is called, and the result of its execution is displayed.

Table creation (no explicit transaction control — implicit session, server-side DDL):

```
qc.exec(format!(
    "CREATE TABLE IF NOT EXISTS `{}` (
        series_id Bytes,
        title Text,
        series_info Text,
        release_date Date,
        comment Text,
        PRIMARY KEY(series_id)
    )",
    "native/query/series"
))
.await?;
```

Adding data

Add data to the created tables using the `UPSERT` statement in YQL. A data update request is sent to the server as a single request with transaction auto-commit mode enabled.

Bulk load with `AS_TABLE` and typed parameters:

```
use ydb::{Value, ydb_struct};

let rows: Vec<Value> = /* ... */;
let list = Value::list_from(example_row, rows)?;
qc.exec("UPSERT INTO ... FROM AS_TABLE($seriesData);")
    .param("$seriesData", list)
    .await?;
```

Retrieving data

Retrieve data using a `SELECT` statement in YQL. Handle the retrieved data selection in the app.

Read materialized result with snapshot read-only isolation:

```
use ydb::QueryTxMode;

let mut result = qc
    .query("SELECT series_id, title, release_date FROM `native/query/series`")
    .with_tx_mode(QueryTxMode::SnapshotReadOnly)
    .idempotent(true)
    .await?;

while let Some(result_set) = result.next_result_set().await? {
    for mut row in result_set {
        // extract columns from row
    }
}
result.close().await?;
```

Parameterized queries

Query data using parameters. This query execution method is preferable because it allows the server to reuse the query execution plan for subsequent calls and protects against vulnerabilities such as SQL injection.

Per-call parameters use `.param(name, value)` or the `ydb_params!` macro:

```
use ydb::ydb_params;

// one parameter at a time
qc.exec("UPSERT INTO `native/query/series` (series_id, title) VALUES ($id, $title)")
    .param("$id", b"series-1".to_vec())
    .param("$title", "Example title")
    .await?;

// several parameters via macro
qc.exec("UPSERT INTO `native/query/series` (series_id, title) VALUES ($id, $title)")
    .params(ydb_params!(
        "$id" => b"series-2".to_vec(),
        "$title" => "Another title",
```

```
    ))
    .await?;
```

Managing transactions

Transactions are managed through TCL `Begin` and `Commit` calls.

In most cases, instead of explicitly using `Begin` and `Commit` calls, it's better to use transaction control parameters in execute calls. This allows to avoid additional requests to YDB server and thus run queries more efficiently.

The Rust SDK does not expose explicit `Begin` / `Commit` to application code. Use `retry_transaction` with `QueryTransactionOptions` for interactive transactions. For a single SQL statement, set isolation with `.with_tx_mode(...)`.

Single statement in snapshot read-only mode:

```
use ydb::QueryTxMode;

let mut row = qc
    .query_row("SELECT title FROM `native/query/series` WHERE series_id = $id")
    .param("$id", b"series-1".to_vec())
    .with_tx_mode(QueryTxMode::SnapshotReadOnly)
    .idempotent(true)
    .await?;
```

Interactive transaction with multiple operations:

```
use ydb::{QueryTransactionOptions, QueryTxMode};

let mut qc = qc.clone_with_transaction_options(
    QueryTransactionOptions::new().with_mode(QueryTxMode::SerializableReadWrite),
);

let title: String = qc
    .retry_transaction(async |tx| {
        let mut row = tx
            .query_row("SELECT title FROM `native/query/series` WHERE series_id = $id")
            .param("$id", b"series-1".to_vec())
            .await?;
        Ok(row.remove_field_by_name("title")?.try_into()?)
    })
    .await?;
```

By default, such calls use implicit transaction control — the server infers isolation from the SQL statement.

Generate code from YQL (sqlc-ydb)

sqlc-ydb generates typed code for YDB from a schema and YQL queries. You write SQL; the tool creates methods that bind parameters and read results through the selected SDK or driver. It is an independent project following the familiar sqlc workflow.

This guide takes a schema and two queries through code generation and execution in an application.

Installation

On Linux and macOS, run the commands below. Generation needs no SDK or database connection. The installer selects the latest stable release, verifies SHA256 and places the executable in `~/.local/bin`. If PATH setup is needed, follow the installer's instructions before running `sqlc-ydb version`.

```
curl -fsSL https://raw.githubusercontent.com/ydb-platform/sqlc-ydb/main/install.sh | bash
sqlc-ydb version
```

For Windows, download the appropriate archive from the latest stable release. See the installation guide for other installation methods and version selection.

Schema and queries

Create an empty project directory and save this schema, shared by all languages, as `schema.sql`:

```
CREATE TABLE authors (
    id Uint64 NOT NULL,
    name Utf8 NOT NULL,
    PRIMARY KEY (id)
);
```

Save these two queries as `queries.sql`. `UpsertAuthor` inserts an author or updates an existing author's name; `GetAuthor` reads by primary key. Parameter names start with `$`; their types are inferred from the schema, so no `DECLARE` is needed here.

```
-- name: UpsertAuthor :exec
UPSERT INTO authors (id, name) VALUES ($author_id, $author_name);

-- name: GetAuthor :one
SELECT id, name FROM authors WHERE id = $author_id;
```

The `-- name:` comment sets the generated method name. `:exec` executes a statement without result rows; `:one` retrieves one row. Every example below uses these same input files.

1. Generator configuration in sqlc.yaml

Save the configuration from the selected tab as `sqlc.yaml`. Input paths and the `out` directory are resolved relative to this file. To list the languages and frameworks available in your installed version, run:

```
sqlc-ydb init --help
```

Choose a language and runtime from that output, then request help for the generator's options:

```
sqlc-ydb init --language <language> --runtime <runtime> --help
```

Replace `<language>` and `<runtime>` with values from the help output. The command describes each option's purpose, type, allowed values, and default. Omitting `--runtime` selects the language's default runtime. Calls with `--help` do not create files.

The tabs below illustrate configuration examples. For the current list of supported languages, runtimes, and options, consult the help provided by your installed sqlc-ydb version.

C++

YDB SDK

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      cpp:
        out: "cpp/ydb"
        namespace: "authors"
        runtime: "ydb"
```

userver

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      cpp:
        out: "cpp/userver"
```

```
        namespace: "authors"
        runtime: "userver"
```

Go

YDB SDK

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      go:
        out: "go/native"
        package: "authors"
        sql_package: "ydb"
```

database/sql

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      go:
        out: "go/sql"
        package: "authors"
        sql_package: "database/sql"
```

Java

YDB SDK

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      java:
        out: "java/ydb"
        package: "authors"
        runtime: "ydb"
```

JDBC

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      java:
        out: "java/jdbc"
        package: "authors"
        runtime: "jdbc"
```

jOOQ

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      java:
        out: "java/jooq"
        package: "authors"
        runtime: "jooq"
```

Python

YDB SDK

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      python:
```

```
        out: "python/ydb"
        runtime: "ydb"
```

DB-API

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      python:
        out: "python/dbapi"
        runtime: "dbapi"
```

SQLAlchemy

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      python:
        out: "python/sqlalchemy"
        runtime: "sqlalchemy"
```

C#

ADO.NET

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      csharp:
        out: "csharp/adonet"
        namespace: "Authors"
        runtime: "adonet"
```

Dapper

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      csharp:
        out: "csharp/dapper"
        namespace: "Authors"
        runtime: "dapper"
```

TypeScript

YDB SDK

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      typescript:
        out: "typescript"
        runtime: "ydb"
```

Rust

YDB SDK

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      rust:
        out: "rust"
        runtime: "ydb"
```

PHP

YDB SDK

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      php:
        out: "php"
        namespace: "Authors"
        runtime: "ydb"
```

Kotlin

YDB SDK

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      kotlin:
        out: "kotlin/ydb"
        package: "authors"
        runtime: "ydb"
```

JDBC

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      kotlin:
        out: "kotlin/jdbc"
        package: "authors"
        runtime: "jdbc"
```

Exposed

```
version: "2"
sql:
  - engine: ydb
    schema: schema.sql
    queries: queries.sql
    gen:
      kotlin:
        out: "kotlin/exposed"
        package: "authors"
        runtime: "exposed"
```

2. Code generation

Run these commands next to `sqlc.yaml`. `compile` checks the inputs, `generate` writes code, and `diff` checks that saved files match generation. Add `diff` to CI: it exits with code 1 when files differ.

```
sqlc-ydb compile
sqlc-ydb generate
sqlc-ydb diff
```

The following excerpts show actual generated row types or read methods for the shared query. Imports and enclosing class declarations are omitted. Use the complete files in `out`; do not edit generated code manually.

C++

YDB SDK

```
struct GetAuthorRow final {
    std::uint64_t id;
    std::string name;
};

    void UpsertAuthor(std::uint64_t author_id, const std::string& author_name) const;
    std::optional<GetAuthorRow> GetAuthor(std::uint64_t author_id) const;
```

userver

```
struct GetAuthorRow final {
    std::uint64_t id;
```

```
    ::userver::ydb::Utf8 name;
};

    void UpsertAuthor(std::uint64_t author_id, const ::userver::ydb::Utf8& author_name) const;
    std::optional<GetAuthorRow> GetAuthor(std::uint64_t author_id) const;
```

Go

YDB SDK

```
// -- name: GetAuthor :one
func (q *Queries) GetAuthor(ctx context.Context, arg uint64, opts ...query.ExecuteOption) (GetAuthorRow, erro
r) {
    parameters := ydb.ParamsBuilder()
    parameters = parameters.Param("$author_id").Uint64(arg)

    callOptions := append([]query.ExecuteOption(nil), opts...)
    callOptions = append(callOptions, query.WithParameters(parameters.Build()))

    result, err := q.db.QueryRow(ctx, ""+
        "SELECT id, name FROM authors WHERE id = $author_id;",
        callOptions...,
    )
    if err != nil {
        return GetAuthorRow{}, err
    }

    var row GetAuthorRow
    if err := result.ScanNamed(
        query.Named("id", &row.ID),
        query.Named("name", &row.Name),
    ); err != nil {
        return GetAuthorRow{}, err
    }

    return row, nil
}
```

database/sql

```
// -- name: GetAuthor :one
func (q *Queries) GetAuthor(ctx context.Context, arg uint64) (GetAuthorRow, error) {
    var row GetAuthorRow
    err := q.db.QueryRowContext(ctx, ""+
        "SELECT id, name FROM authors WHERE id = $author_id;",
        sql.Named("author_id", arg),
    ).Scan(
        &row.ID,
        &row.Name,
    )

    return row, err
}
```

Java

YDB SDK

```
public record GetAuthorRow(long id, String name) {}

    // -- name: GetAuthor :one
    public java.util.Optional<GetAuthorRow> getAuthor(long authorId) {
        var _params = Params.create();
        _params.put("$author_id", PrimitiveValue.newUint64(authorId));
        var _query = QueryReader.readFrom(
                client.createQuery("""
                    SELECT id, name FROM authors WHERE id = $author_id;\
                    """, _params)).join().getValue();
        if (_query.getResultSetCount() != 1) throw new IllegalStateException("Expected one result set");
        var _rows = _query.getResultSet(0);
        if (!_rows.next()) return java.util.Optional.empty();
        long _value0 = _rows.getColumn(0).getUint64();
        String _value1 = _rows.getColumn(1).getText();
        return java.util.Optional.of(new GetAuthorRow(_value0, _value1));
    }
```

JDBC

```
public record GetAuthorRow(long id, String name) {}

    // -- name: GetAuthor :one
    public java.util.Optional<GetAuthorRow> getAuthor(long authorId) throws java.sql.SQLException {
        try (var _prepared = client.prepareStatement("""
            SELECT id, name FROM authors WHERE id = ?;\
            """)) {
            _prepared.setObject(1, PrimitiveValue.newUint64(authorId));
            try (var _rows = _prepared.executeQuery()) {
                if (!_rows.next()) return java.util.Optional.empty();
```

```
                long _value0 = _rows.getLong(1);
                String _value1 = _rows.getString(2);
                return java.util.Optional.of(new GetAuthorRow(_value0, _value1));
            }
        }
    }
```

jOOQ

```
public record GetAuthorRow(ULong id, String name) {}

    // -- name: GetAuthor :one
    public Optional<GetAuthorRow> getAuthor(ULong authorId) {
        return dsl.select(AUTHORS.ID, AUTHORS.NAME)
                .from(AUTHORS)
                .where(AUTHORS.ID.eq(val(authorId, YdbTypes.UINT64)))
                .fetchOptional(mapping(GetAuthorRow::new));
    }
```

Python

YDB SDK

```
@dataclass
class Authors:
    id: int
    name: str
```

DB-API

```
@dataclass
class Authors:
    id: int
    name: str
```

SQLAlchemy

```
@dataclass
class Authors:
    id: int
    name: str
```

C#

ADO.NET

```
public sealed record GetAuthorRow(
    ulong ID,
    string Name
);

    // -- name: GetAuthor :one
    public async Task<GetAuthorRow> GetAuthorAsync(ulong authorId, CancellationToken cancellationToken = defa
ult)
    {
        await using var command = new YdbCommand(
            "SELECT id, name FROM authors WHERE id = $author_id;", _connection) { Transaction = _transaction
};
        command.Parameters.Add(new YdbParameter("$author_id", DbType.UInt64, authorId));
        await using var reader = await command.ExecuteReaderAsync(cancellationToken).ConfigureAwait(false);
        if (!await reader.ReadAsync(cancellationToken).ConfigureAwait(false))
        {
            throw new InvalidOperationException("query returned no rows");
        }
        return GetAuthorRowFrom(reader);
    }

    private static GetAuthorRow GetAuthorRowFrom(DbDataReader reader) => new(
        reader.GetFieldValue<ulong>(0),
        reader.GetFieldValue<string>(1)
    );
```

Dapper

```
public sealed record GetAuthorRow(
    ulong ID,
    string Name
);

    // -- name: GetAuthor :one
    public async Task<GetAuthorRow> GetAuthorAsync(ulong authorId, CancellationToken cancellationToken = defa
ult, int? commandTimeout = null)
    {
        var parameters = new YdbParameters(
            new YdbParameter("$author_id", DbType.UInt64, authorId)
```

```
        );

        var command = new CommandDefinition(
            commandText: """
            SELECT id, name FROM authors WHERE id = $author_id;
            """,
            parameters: parameters,
            transaction: _transaction,
            commandTimeout: commandTimeout,
            cancellationToken: cancellationToken);

        return await _connection.QueryFirstAsync<GetAuthorRow>(command).ConfigureAwait(false);
    }
```

TypeScript

YDB SDK

```
// -- name: GetAuthor :one
  async getAuthor(authorId: bigint, configure?: ConfigureQuery): Promise<GetAuthorRow | null> {
    const stmt = this.#sql<[GetAuthorRow]>`SELECT id, name FROM authors WHERE id = $author_id;`
      .parameter("author_id", new Uint64(authorId));
    configure?.(stmt);
    const [rows] = await stmt;

    return rows[0] ?? null;
  }
```

Rust

YDB SDK

```
// -- name: GetAuthor :one
    #[builder(on(String, into))]
    pub async fn author(&mut self, author_id: u64) -> ydb::YdbResult<GetAuthorRow> {
        let mut row = self
            .client
            .query_row(r"SELECT id, name FROM authors WHERE id = $author_id;")
            .param("$author_id", author_id)
            .await?;
        Ok(GetAuthorRow {
            id: row.remove_field(0)?.try_into()?,
            name: row.remove_field(1)?.try_into()?,
        })
    }
```

PHP

YDB SDK

```
final class GetAuthorRow
{
    public function __construct(
        public readonly string $id,
        public readonly string $name
    ) {}
}
```

Kotlin

YDB SDK

```
data class GetAuthorRow(
    val id: Long,
    val name: String
)

    // -- name: GetAuthor :one
    fun getAuthor(authorId: Long): GetAuthorRow? {
        val _params = Params.create()
        _params.put("\$author_id", PrimitiveValue.newUint64(authorId))
        val _query = if (transaction != null) {
            QueryReader.readFrom(transaction.createQuery(
                "SELECT id, name FROM authors WHERE id = \$author_id;", _params)).join().getValue()
        } else {
            client!!.supplyResult { _session ->
                QueryReader.readFrom(_session.createQuery(
                "SELECT id, name FROM authors WHERE id = \$author_id;", TxMode.SERIALIZABLE_RW, _params))
            }.join().getValue()
        }
        kotlin.check(_query.getResultSetCount() == 1) { "Expected one result set" }
        val _rows = _query.getResultSet(0)
        if (!_rows.next()) return null
        val _value0: Long = _rows.getColumn(0).getUint64()
        val _value1: String = _rows.getColumn(1).getText()
        return GetAuthorRow(_value0, _value1)
    }
```

JDBC

```
data class GetAuthorRow(
    val id: Long,
    val name: String
)

    // -- name: GetAuthor :one
    fun getAuthor(authorId: Long): GetAuthorRow? {
        client.prepareStatement(
            "SELECT id, name FROM authors WHERE id = ?;").use { _prepared ->
            _prepared.setObject(1, PrimitiveValue.newUint64(authorId))
            _prepared.executeQuery().use { _rows ->
                if (!_rows.next()) return null
                val _value0: Long = _rows.getLong(1)
                val _value1: String = _rows.getString(2)
                return GetAuthorRow(_value0, _value1)
            }
        }
    }
```

Exposed

```
data class GetAuthorRow(
    val id: Long,
    val name: String
)

    // -- name: GetAuthor :one
    fun getAuthor(authorId: Long): GetAuthorRow? {
        val _connection = client.connection.connection as java.sql.Connection
        _connection.prepareStatement(
            "SELECT id, name FROM authors WHERE id = ?;").use { _prepared ->
            _prepared.setObject(1, PrimitiveValue.newUint64(authorId))
            _prepared.executeQuery().use { _rows ->
                if (!_rows.next()) return null
                val _value0: Long = _rows.getLong(1)
                val _value1: String = _rows.getString(2)
                return GetAuthorRow(_value0, _value1)
            }
        }
    }
```

Application setup

The generator does not create database tables. Apply `schema.sql` using your migration tool or the YDB CLI, then add a row for the single-query example:

```
UPSERT INTO authors (id, name) VALUES (42, "Alice");
```

Add the selected SDK/driver dependencies and configure authentication. The fragments below assume initialized connections or clients and imported generated code. See the project examples for complete projects with dependencies and connection setup.

3. Individual query execution

Call the generated method for author 42:

C++

YDB SDK

```
authors::Queries queries{client};
auto author = queries.GetAuthor(42);
```

`client` is a configured `NYdb::NQuery::TQueryClient`; C++20 and the YDB C++ SDK are required. Standalone calls use SDK retries; `RetryQuerySync` retries the entire transaction. `Queries` does not finish a supplied transaction: the example commits or rolls it back explicitly. Complete example.

userver

```
authors::Queries queries{table_client};
auto author = queries.GetAuthor(42);
```

`table_client` is a `userver::ydb::TableClient` obtained from the YDB component; run this code in a userver coroutine. `RetryTx` retries the whole callback and finishes the transaction according to `TxAction`; return `kRollback` to discard changes. `Utf8` is userver's string type for YQL Utf8. Complete example.

Go

YDB SDK

```
q := authors.New(driver.Query())
author, err := q.GetAuthor(ctx, 42)
if err != nil {
```

```
    return err
}
fmt.Println(author.Name)
```

`driver` is an initialized `*ydb.Driver`; `ctx` is a context. Import the generated package as `authors` and `github.com/ydb-platform/ydb-go-sdk/v3/query` for the transaction. These fragments belong in a function returning `error`.

database/sql

```
q := authors.New(connection)
author, err := q.GetAuthor(ctx, 42)
if err != nil {
    return err
}
fmt.Println(author.Name)
```

`connection` is an initialized `*sql.DB` using the YDB driver; `ctx` is a context. Import the generated package as `authors`. These fragments belong in a function returning `error`; retry the complete transaction block when needed.

Java

YDB SDK

```
GetAuthorRow author = retry.supplyResult(session -> {
    var tx = session.createNewTransaction(TxMode.SERIALIZABLE_RW);
    try {
        var row = new Queries(tx).getAuthor(42L).orElseThrow();
        return CompletableFuture.completedFuture(tx.commit().join().map(info -> row));
    } finally {
        if (tx.isActive()) {
            tx.rollback().join().expectSuccess();
        }
    }
}).join().getValue();
```

`retry` is an application-configured `SessionRetryContext`; import `TxMode`, `CompletableFuture`, and the generated `Queries` and `GetAuthorRow`. Even one Java native call uses an open transaction. `retry` reruns the whole function with a new session; the caller completes the transaction.

JDBC

```
GetAuthorRow author = new Queries(connection).getAuthor(42L).orElseThrow();
```

`connection` is a `java.sql.Connection` initially in autocommit mode, with no other active transaction. Import the generated `Queries` and `GetAuthorRow`. The transaction code restores autocommit after commit or rollback and propagates exceptions. The application can retry the whole transaction when appropriate.

jOOQ

```
GetAuthorRow author = new Queries(dsl).getAuthor(ULong.valueOf(42)).orElseThrow();
```

`dsl` is a `YdbDSLContext`; import `YDB`, `ULong`, and the generated `Queries` and `GetAuthorRow`. The single call uses an autocommit connection. `transactionResult` commits on success and rolls back on an exception; `YDB.using(configuration)` uses the current transaction connection. The application handles retries around the whole transaction.

Python

YDB SDK

```
import ydb
from python.ydb.queries import Querier

queries = Querier(pool, retry_settings=ydb.RetrySettings(idempotent=True))
author = queries.get_author(42)
```

`pool` is an initialized `ydb.QuerySessionPool`. `retry_tx_sync` retries the whole transaction and commits after the callback; the callback must have no external side effects. The repeated UPSERT in this example sets fixed values. Runnable example.

DB-API

```
from python.dbapi.queries import Querier

author = Querier(connection).get_author(42)
```

`connection` is an open `ydb_dbapi` connection with no active transaction. The driver defaults to AUTOCOMMIT: select SERIALIZABLE before `begin()` to share a transaction. The caller manages retries of the whole transaction. Runnable example.

SQLAlchemy

```
from python.sqlalchemy.queries import Querier

with engine.connect() as connection:
    author = Querier(connection).get_author(42)
```

`engine` is an initialized SQLAlchemy Engine using the `yql+ydb` dialect. Set SERIALIZABLE before `begin()`: a transaction block alone does not override the driver's AUTOCOMMIT. The context commits or rolls back; the caller manages retries of the whole transaction. Runnable example.

C#

ADO.NET

```
var author = await dataSource.ExecuteAsync(
    (connection, ct) => new Authors.Queries(connection).GetAuthorAsync(42UL, ct),
    cancellationToken);
```

Requires .NET 8, Ydb.Sdk, a configured `YdbDataSource dataSource`, and a `CancellationToken cancellationToken`. `ExecuteAsync` retries the whole callback according to the data source policy. For multiple calls in one transaction, use `ExecuteInTransactionAsync`: the provider manages commit, rollback and whole-transaction retries. Complete example. `GetAuthorAsync` throws `InvalidOperationException` when no row is found.

Dapper

```
var author = await dataSource.ExecuteAsync(
    (connection, ct) => new Authors.Queries(connection).GetAuthorAsync(42UL, ct),
    cancellationToken);
```

Requires .NET 8, Ydb.Sdk and Dapper, a configured `YdbDataSource dataSource`, and a `CancellationToken cancellationToken`. `ExecuteAsync` retries the whole callback according to the data source policy. For multiple calls in one transaction, use `ExecuteInTransactionAsync`: the provider manages commit, rollback and whole-transaction retries. Complete example. `GetAuthorAsync` throws `InvalidOperationException` when no row is found.

TypeScript

YDB SDK

```
import { Queries } from "./typescript/queries.js";

const queries = new Queries(sql);
const author = await queries.getAuthor(42n);
console.log(author?.name);
```

`sql` is the `query(driver)` function from `@ydbjs/query` for an initialized driver. `transaction` commits on callback success and rolls back on failure.

Rust

YDB SDK

```
use generated::queries::Queries;

let mut queries = Queries::new(&mut query_client);
let author = queries.author().author_id(42).call().await?;
println!("{}", author.name);
```

`query_client` is a mutable `ydb::QueryClient`. Expose the files in `rust/` as the `generated` module and add the `ydb` and `bon` dependencies. The SDK commits on callback success and retries the whole callback.

PHP

YDB SDK

```
use Authors\Queries;

$queries = new Queries($table, idempotent: true);
$author = $queries->getAuthor('42');
```

`$table` is an initialized `YdbPlatform\Ydb\Table`; generated classes are available through autoload. Pass `Uint64` as a decimal string. For this SELECT, safe retries are explicitly enabled with `idempotent: true`. Runnable example.

Kotlin

YDB SDK

```
val author = requireNotNull(Queries(retry).getAuthor(42L))
```

`retry` is an application-configured `SessionRetryContext`; import `TxMode`, `CompletableFuture`, and the generated `Queries`. The single call uses the retry context; the composed operation passes one `QueryTransaction` to the helpers. The outer `retry` retries the whole transaction.

JDBC

```
val author = requireNotNull(Queries(connection).getAuthor(42L))
```

`connection` is a `java.sql.Connection` initially in autocommit mode, with no other active transaction; import the generated `Queries`. The transaction code restores autocommit after commit or rollback and propagates exceptions. The application can retry the whole transaction when appropriate.

Exposed

```
val author = ydbTransaction(database) {
    requireNotNull(Queries(this).getAuthor(42L))
}
```

`database` is a configured Exposed `Database` with the YDB dialect registered; import `tech.ydb.exposed.dialect.ydbTransaction` and the generated `Queries`. `ydbTransaction` supplies one `JdbcTransaction`, commits on success, and rolls back on an exception. Retry policy applies to the whole block.

4. Multiple query execution in one transaction

In each example, two generated methods execute two YQL queries in one transaction: `UpsertAuthor` writes the name, then `GetAuthor` reads it. The application controls transaction boundaries and retries; retry the whole block when needed. Keep external side effects out of callbacks that the SDK may retry.

C++

YDB SDK

```
std::optional<authors::GetAuthorRow> author;
const auto status = client.RetryQuerySync([&](NYdb::NQuery::TSession session) -> NYdb::TStatus {
    auto started = session.BeginTransaction(
        NYdb::NQuery::TTxSettings::SerializableRW()).GetValueSync();
    if (!started.IsSuccess()) {
        return started;
    }
    auto transaction = started.GetTransaction();
    try {
        authors::Queries queries{transaction};
        queries.UpsertAuthor(42, "Alice");
        author = queries.GetAuthor(42);
    } catch (const NYdb::NStatusHelpers::TYdbErrorException& error) {
        transaction.Rollback().GetValueSync();
        return error.GetStatus();
    } catch (...) {
        transaction.Rollback().GetValueSync();
        throw;
    }
    return transaction.Commit().GetValueSync();
});
NYdb::NStatusHelpers::ThrowOnError(status);
```

userver

```
std::optional<authors::GetAuthorRow> author;
table_client.RetryTx("upsert-and-read-author", {}, [&](::userver::ydb::TxActor& transaction) {
    authors::Queries queries{transaction};
    queries.UpsertAuthor(42, ::userver::ydb::Utf8{"Alice"});
    author = queries.GetAuthor(42);
    return ::userver::ydb::TxAction::kCommit;
});
```

Go

YDB SDK

```
return driver.Query().DoTx(ctx, func(ctx context.Context, tx query.TxActor) error {
    q := authors.New(tx)
    if err := q.UpsertAuthor(ctx, authors.UpsertAuthorParams{
        AuthorID: 42, AuthorName: "Alice",
    }); err != nil {
        return err
    }
    _, err := q.GetAuthor(ctx, 42)
    return err
})
```

database/sql

```
tx, err := connection.BeginTx(ctx, nil)
if err != nil {
    return err
}
defer tx.Rollback()

q := authors.New(tx)
if err := q.UpsertAuthor(ctx, authors.UpsertAuthorParams{
    AuthorID: 42, AuthorName: "Alice",
}); err != nil {
    return err
}
if _, err := q.GetAuthor(ctx, 42); err != nil {
    return err
```

```
}
return tx.Commit()
```

Java

YDB SDK

```
GetAuthorRow author = retry.supplyResult(session -> {
    var tx = session.createNewTransaction(TxMode.SERIALIZABLE_RW);
    try {
        var queries = new Queries(tx);
        queries.upsertAuthor(42L, "Alice");
        var row = queries.getAuthor(42L).orElseThrow();
        return CompletableFuture.completedFuture(tx.commit().join().map(info -> row));
    } finally {
        if (tx.isActive()) {
            tx.rollback().join().expectSuccess();
        }
    }
}).join().getValue();
```

JDBC

```
connection.setAutoCommit(false);
try {
    var queries = new Queries(connection);
    queries.upsertAuthor(42L, "Alice");
    GetAuthorRow author = queries.getAuthor(42L).orElseThrow();
    connection.commit();
} catch (Exception error) {
    connection.rollback();
    throw error;
} finally {
    connection.setAutoCommit(true);
}
```

jOOQ

```
GetAuthorRow author = dsl.transactionResult(configuration -> {
    var queries = new Queries(YDB.using(configuration));
    var id = ULong.valueOf(42);
    queries.upsertAuthor(id, "Alice");
    return queries.getAuthor(id).orElseThrow();
});
```

Python

YDB SDK

```
import ydb
from python.ydb.queries import Querier


def update_author(tx):
    queries = Querier(tx)
    queries.upsert_author(42, "Alice")
    return queries.get_author(42)


author = pool.retry_tx_sync(
    update_author,
    tx_mode=ydb.QuerySerializableReadWrite(),
    retry_settings=ydb.RetrySettings(idempotent=True),
)
```

DB-API

```
import ydb_dbapi
from python.dbapi.queries import Querier

connection.set_isolation_level(ydb_dbapi.IsolationLevel.SERIALIZABLE)
connection.begin()
try:
    queries = Querier(connection)
    queries.upsert_author(42, "Alice")
    author = queries.get_author(42)
    connection.commit()
except Exception:
    connection.rollback()
    raise
```

SQLAlchemy

```
from python.sqlalchemy.queries import Querier
```

```
with engine.connect().execution_options(isolation_level="SERIALIZABLE") as connection:
    with connection.begin():
        queries = Querier(connection)
        queries.upsert_author(42, "Alice")
        author = queries.get_author(42)
```

C#

ADO.NET

```
var author = await dataSource.ExecuteInTransactionAsync(async (connection, ct) =>
{
    var queries = new Authors.Queries(connection);
    await queries.UpsertAuthorAsync(new Authors.UpsertAuthorParams(42UL, "Alice"), ct);
    return await queries.GetAuthorAsync(42UL, ct);
}, cancellationToken: cancellationToken);
```

Dapper

```
var author = await dataSource.ExecuteInTransactionAsync(async (connection, ct) =>
{
    var queries = new Authors.Queries(connection);
    await queries.UpsertAuthorAsync(new Authors.UpsertAuthorParams(42UL, "Alice"), ct);
    return await queries.GetAuthorAsync(42UL, ct);
}, cancellationToken: cancellationToken);
```

TypeScript

YDB SDK

```
import { Queries } from "./typescript/queries.js";
import type { ConfigureQuery } from "./typescript/queries.js";

const author = await sql.transaction(async (tx, signal) => {
  const queries = new Queries(tx);
  const configure: ConfigureQuery = (stmt) => { stmt.signal(signal); };
  await queries.upsertAuthor({ authorId: 42n, authorName: "Alice" }, configure);
  return queries.getAuthor(42n, configure);
});
console.log(author?.name);
```

Rust

YDB SDK

```
use generated::queries::Queries;

let author = query_client.retry_tx(ydb::closure!(async |tx| {
    let mut queries = Queries::new(tx);
    queries.upsert_author()
        .author_id(42)
        .author_name("Alice")
        .call().await?;
    Ok(queries.author().author_id(42).call().await?)
})).await?;
println!("{}", author.name);
```

PHP

YDB SDK

```
use Authors\Queries;
use Authors\UpsertAuthorParams;

$session = $table->session();
$txId = $session->beginTransaction();
$queries = (new Queries($table))->withTx($session, $txId);

$queries->upsertAuthor(new UpsertAuthorParams('42', 'Alice'));
$author = $queries->getAuthor('42');

$session->commitTransaction();
```

Both queries execute in the same transaction. `$table` is an initialized `YdbPlatform\Ydb\Table`, and generated classes are available through autoload. Error handling is omitted: on failure, the application must call `$session->rollbackTransaction()`. The bound helper does not retry individual queries; retries must repeat the entire transaction. After commit or rollback, the session returns to the pool; acquire a session again and create a new bound helper for the next transaction. PHP contract.

Kotlin

YDB SDK

```
val author = retry.supplyResult { session ->
    val tx = session.createNewTransaction(TxMode.SERIALIZABLE_RW)
    try {
        val queries = Queries(tx)
```

```
        queries.upsertAuthor(42L, "Alice")
        val row = requireNotNull(queries.getAuthor(42L))
        CompletableFuture.completedFuture(tx.commit().join().map { row })
    } finally {
        if (tx.isActive) {
            tx.rollback().join().expectSuccess()
        }
    }
}.join().getValue()
```

JDBC

```
connection.autoCommit = false
try {
    val queries = Queries(connection)
    queries.upsertAuthor(42L, "Alice")
    val author = requireNotNull(queries.getAuthor(42L))
    connection.commit()
} catch (error: Exception) {
    connection.rollback()
    throw error
} finally {
    connection.autoCommit = true
}
```

Exposed

```
val author = ydbTransaction(database) {
    val queries = Queries(this)
    queries.upsertAuthor(42L, "Alice")
    requireNotNull(queries.getAuthor(42L))
}
```

The current set of supported commands, configuration options, and YQL constructs is maintained in the sqlc-ydb compatibility contract. If a query is unsupported or generated code is awkward to use, open an issue with a minimal schema, query, configuration and the output of `sqlc-ydb version --verbose`.

YQL Tutorial - Overview

From this tutorial, you will learn how to perform basic operations with data in YDB and get familiar with the YQL syntax. A detailed description of this syntax is also available in the YQL reference documentation.

The tutorial consists of 15 steps:

1. Creating a table
2. Adding data to a table
3. Selecting data from all columns
4. Selecting data from specific columns
5. Sorting and filtering
6. Data aggregation
7. Additional selection criteria
8. Joining tables with JOIN
9. Inserting and updating data with REPLACE
10. Inserting and updating data with UPSERT
11. Inserting data with INSERT
12. Updating data with UPDATE
13. Deleting data
14. Adding and deleting columns
15. Deleting a table

Creating a table

Create the tables and set the data schema for them using the statement CREATE TABLE.

> **Note**
>
> Keywords are case-insensitive and written in capital letters for clarity only.

```
CREATE TABLE series         -- series is the table name.
(                           -- Must be unique within the folder.
    series_id Uint64,
    title Utf8,
    series_info Utf8,
    release_date Uint64,
    PRIMARY KEY (series_id) -- The primary key is a column or
                            -- combination of columns that uniquely identifies
                            -- each table row (contains only
                            -- non-repeating values). A table can have
                            -- only one primary key. For every table
                            -- in YDB, the primary key is required.
);

CREATE TABLE seasons
(
    series_id Uint64,
    season_id Uint64,
    title Utf8,
    first_aired Uint64,
    last_aired Uint64,
    PRIMARY KEY (series_id, season_id)
);

CREATE TABLE episodes
(
    series_id Uint64,
    season_id Uint64,
    episode_id Uint64,
    title Utf8,
    air_date Uint64,
    PRIMARY KEY (series_id, season_id, episode_id)
);

COMMIT;
```

Adding data to a table

Populate the created tables with data using the REPLACE INTO statement.

```
REPLACE INTO series (series_id, title, release_date, series_info)

VALUES

    -- By default, numeric literals have type Int32
    -- if the value is within the range.
    -- Otherwise, they automatically expand to Int64.
    (
        1,
        "IT Crowd",
        CAST(Date("2006-02-03") AS Uint64),   -- CAST converts one datatype into another.
                                              -- You can convert a string
                                              -- literal into a primitive literal.
                                              -- The Date() function converts a string
                                              -- literal in ISO 8601 format into a date.

        "The IT Crowd is a British sitcom produced by Channel 4, written by Graham Linehan, produced by Ash A
talla and starring Chris O'Dowd, Richard Ayoade, Katherine Parkinson, and Matt Berry."),
    (
        2,
        "Silicon Valley",
        CAST(Date("2014-04-06") AS Uint64),
        "Silicon Valley is an American comedy television series created by Mike Judge, John Altschuler and Da
ve Krinsky. The series focuses on five young men who founded a startup company in Silicon Valley."
    )
    ;

REPLACE INTO seasons (series_id, season_id, title, first_aired, last_aired)
VALUES
    (1, 1, "Season 1", CAST(Date("2006-02-03") AS Uint64), CAST(Date("2006-03-03") AS Uint64)),
    (1, 2, "Season 2", CAST(Date("2007-08-24") AS Uint64), CAST(Date("2007-09-28") AS Uint64)),
    (1, 3, "Season 3", CAST(Date("2008-11-21") AS Uint64), CAST(Date("2008-12-26") AS Uint64)),
    (1, 4, "Season 4", CAST(Date("2010-06-25") AS Uint64), CAST(Date("2010-07-30") AS Uint64)),
    (2, 1, "Season 1", CAST(Date("2014-04-06") AS Uint64), CAST(Date("2014-06-01") AS Uint64)),
    (2, 2, "Season 2", CAST(Date("2015-04-12") AS Uint64), CAST(Date("2015-06-14") AS Uint64)),
    (2, 3, "Season 3", CAST(Date("2016-04-24") AS Uint64), CAST(Date("2016-06-26") AS Uint64)),
    (2, 4, "Season 4", CAST(Date("2017-04-23") AS Uint64), CAST(Date("2017-06-25") AS Uint64)),
    (2, 5, "Season 5", CAST(Date("2018-03-25") AS Uint64), CAST(Date("2018-05-13") AS Uint64))
;

REPLACE INTO episodes (series_id, season_id, episode_id, title, air_date)
VALUES
    (1, 1, 1, "Yesterday's Jam", CAST(Date("2006-02-03") AS Uint64)),
    (1, 1, 2, "Calamity Jen", CAST(Date("2006-02-03") AS Uint64)),
    (1, 1, 3, "Fifty-Fifty", CAST(Date("2006-02-10") AS Uint64)),
    (1, 1, 4, "The Red Door", CAST(Date("2006-02-17") AS Uint64)),
    (1, 1, 5, "The Haunting of Bill Crouse", CAST(Date("2006-02-24") AS Uint64)),
    (1, 1, 6, "Aunt Irma Visits", CAST(Date("2006-03-03") AS Uint64)),
    (1, 2, 1, "The Work Outing", CAST(Date("2006-08-24") AS Uint64)),
    (1, 2, 2, "Return of the Golden Child", CAST(Date("2007-08-31") AS Uint64)),
    (1, 2, 3, "Moss and the German", CAST(Date("2007-09-07") AS Uint64)),
    (1, 2, 4, "The Dinner Party", CAST(Date("2007-09-14") AS Uint64)),
    (1, 2, 5, "Smoke and Mirrors", CAST(Date("2007-09-21") AS Uint64)),
    (1, 2, 6, "Men Without Women", CAST(Date("2007-09-28") AS Uint64)),
    (1, 3, 1, "From Hell", CAST(Date("2008-11-21") AS Uint64)),
    (1, 3, 2, "Are We Not Men?", CAST(Date("2008-11-28") AS Uint64)),
    (1, 3, 3, "Tramps Like Us", CAST(Date("2008-12-05") AS Uint64)),
    (1, 3, 4, "The Speech", CAST(Date("2008-12-12") AS Uint64)),
    (1, 3, 5, "Friendface", CAST(Date("2008-12-19") AS Uint64)),
    (1, 3, 6, "Calendar Geeks", CAST(Date("2008-12-26") AS Uint64)),
    (1, 4, 1, "Jen The Fredo", CAST(Date("2010-06-25") AS Uint64)),
    (1, 4, 2, "The Final Countdown", CAST(Date("2010-07-02") AS Uint64)),
    (1, 4, 3, "Something Happened", CAST(Date("2010-07-09") AS Uint64)),
    (1, 4, 4, "Italian For Beginners", CAST(Date("2010-07-16") AS Uint64)),
    (1, 4, 5, "Bad Boys", CAST(Date("2010-07-23") AS Uint64)),
    (1, 4, 6, "Reynholm vs Reynholm", CAST(Date("2010-07-30") AS Uint64)),
    (2, 1, 1, "Minimum Viable Product", CAST(Date("2014-04-06") AS Uint64)),
    (2, 1, 2, "The Cap Table", CAST(Date("2014-04-13") AS Uint64)),
    (2, 1, 3, "Articles of Incorporation", CAST(Date("2014-04-20") AS Uint64)),
    (2, 1, 4, "Fiduciary Duties", CAST(Date("2014-04-27") AS Uint64)),
    (2, 1, 5, "Signaling Risk", CAST(Date("2014-05-04") AS Uint64)),
    (2, 1, 6, "Third Party Insourcing", CAST(Date("2014-05-11") AS Uint64)),
    (2, 1, 7, "Proof of Concept", CAST(Date("2014-05-18") AS Uint64)),
    (2, 1, 8, "Optimal Tip-to-Tip Efficiency", CAST(Date("2014-06-01") AS Uint64)),
    (2, 2, 1, "Sand Hill Shuffle", CAST(Date("2015-04-12") AS Uint64)),
    (2, 2, 2, "Runaway Devaluation", CAST(Date("2015-04-19") AS Uint64)),
    (2, 2, 3, "Bad Money", CAST(Date("2015-04-26") AS Uint64)),
    (2, 2, 4, "The Lady", CAST(Date("2015-05-03") AS Uint64)),
    (2, 2, 5, "Server Space", CAST(Date("2015-05-10") AS Uint64)),
    (2, 2, 6, "Homicide", CAST(Date("2015-05-17") AS Uint64)),
    (2, 2, 7, "Adult Content", CAST(Date("2015-05-24") AS Uint64)),
    (2, 2, 8, "White Hat/Black Hat", CAST(Date("2015-05-31") AS Uint64)),
    (2, 2, 9, "Binding Arbitration", CAST(Date("2015-06-07") AS Uint64)),
    (2, 2, 10, "Two Days of the Condor", CAST(Date("2015-06-14") AS Uint64)),
    (2, 3, 1, "Founder Friendly", CAST(Date("2016-04-24") AS Uint64)),
    (2, 3, 2, "Two in the Box", CAST(Date("2016-05-01") AS Uint64)),
    (2, 3, 3, "Meinertzhagen's Haversack", CAST(Date("2016-05-08") AS Uint64)),
```

```
    (2, 3, 4, "Maleant Data Systems Solutions", CAST(Date("2016-05-15") AS Uint64)),
    (2, 3, 5, "The Empty Chair", CAST(Date("2016-05-22") AS Uint64)),
    (2, 3, 6, "Bachmanity Insanity", CAST(Date("2016-05-29") AS Uint64)),
    (2, 3, 7, "To Build a Better Beta", CAST(Date("2016-06-05") AS Uint64)),
    (2, 3, 8, "Bachman's Earnings Over-Ride", CAST(Date("2016-06-12") AS Uint64)),
    (2, 3, 9, "Daily Active Users", CAST(Date("2016-06-19") AS Uint64)),
    (2, 3, 10, "The Uptick", CAST(Date("2016-06-26") AS Uint64)),
    (2, 4, 1, "Success Failure", CAST(Date("2017-04-23") AS Uint64)),
    (2, 4, 2, "Terms of Service", CAST(Date("2017-04-30") AS Uint64)),
    (2, 4, 3, "Intellectual Property", CAST(Date("2017-05-07") AS Uint64)),
    (2, 4, 4, "Teambuilding Exercise", CAST(Date("2017-05-14") AS Uint64)),
    (2, 4, 5, "The Blood Boy", CAST(Date("2017-05-21") AS Uint64)),
    (2, 4, 6, "Customer Service", CAST(Date("2017-05-28") AS Uint64)),
    (2, 4, 7, "The Patent Troll", CAST(Date("2017-06-04") AS Uint64)),
    (2, 4, 8, "The Keenan Vortex", CAST(Date("2017-06-11") AS Uint64)),
    (2, 4, 9, "Hooli-Con", CAST(Date("2017-06-18") AS Uint64)),
    (2, 4, 10, "Server Error", CAST(Date("2017-06-25") AS Uint64)),
    (2, 5, 1, "Grow Fast or Die Slow", CAST(Date("2018-03-25") AS Uint64)),
    (2, 5, 2, "Reorientation", CAST(Date("2018-04-01") AS Uint64)),
    (2, 5, 3, "Chief Operating Officer", CAST(Date("2018-04-08") AS Uint64)),
    (2, 5, 4, "Tech Evangelist", CAST(Date("2018-04-15") AS Uint64)),
    (2, 5, 5, "Facial Recognition", CAST(Date("2018-04-22") AS Uint64)),
    (2, 5, 6, "Artificial Emotional Intelligence", CAST(Date("2018-04-29") AS Uint64)),
    (2, 5, 7, "Initial Coin Offering", CAST(Date("2018-05-06") AS Uint64)),
    (2, 5, 8, "Fifty-One Percent", CAST(Date("2018-05-13") AS Uint64));

COMMIT;
```

Selecting data from all columns

Select all columns from the table using SELECT:

> **Note**
>
> We assume that you already created tables in step Creating a table and populated them with data in step Adding data to a table.

```
SELECT         -- Data selection operator.

    *          -- Select all columns from the table.

FROM episodes; -- The table to select the data from.

COMMIT;
```

Selecting data from specific columns

Select the data from the columns `series_id`, `release_date`, and `title`. At the same time, rename `title` to `series_title` and cast the type of `release_date` from `Uint32` to `Date`.

> **Note**
>
> We assume that you already created tables in step Creating a table and populated them with data in step Adding data to a table.

```
SELECT
    series_id,             -- The names of columns (series_id, release_date, title)
                           -- are separated by commas.

    title AS series_title, -- You can use AS to rename columns
                           -- or give a name to an arbitrary expression

    CAST(release_date AS Date) AS release_date

FROM series;

COMMIT;
```

Sorting and filtering

Select the first three episodes from every season of "IT Crowd", except the first season.

> **Note**
>
> We assume that you already created tables in step Creating a table and populated them with data in step Adding data to a table.

```
SELECT
   series_id,
   season_id,
   episode_id,
   CAST(air_date AS Date) AS air_date,
   title

FROM episodes
WHERE
   series_id = 1      -- List of conditions to build the result
   AND season_id > 1  -- Logical AND is used for complex conditions

ORDER BY              -- Sorting the results.
   series_id,         -- ORDER BY sorts the values by one or multiple
   season_id,         -- columns. Columns are separated by commas.
   episode_id

LIMIT 3               -- LIMIT N after ORDER BY means
                      -- "get top N" or "get bottom N" results,
;                     -- depending on sort order.

COMMIT;
```

Data aggregation

Find out the number of unique episodes within every season of every series.

> **Note**
>
> We assume that you already created tables in step Creating a table and populated them with data in step Adding data to a table.

```
SELECT
    series_id,
    season_id,
    COUNT(*) AS cnt  -- Aggregation function COUNT returns the number of rows
                     -- output by the query.
                     -- Asterisk (*) specifies that COUNT
                     -- counts the total number of rows in the table.
                     -- COUNT(*) returns the number of rows in
                     -- the specified table, preserving the duplicate rows.
                     -- It counts each row separately.
                     -- The result includes rows that contain null values.
FROM episodes

GROUP BY
    series_id,       -- The query result will follow the listed order of columns.
    season_id        -- Multiple columns are separated by a comma.
                     -- Other columns can be listed after a SELECT only if
                     -- they are passed to an aggregate function.
ORDER BY
    series_id,
    season_id
;

COMMIT;
```

Additional selection criteria

Select all the episode names of the first season of each series and sort them by name.

> **Note**
>
> We assume that you already created tables in step Creating a table and populated them with data in step Adding data to a table.

```
SELECT
    series_title,               -- series_title is defined below in GROUP BY

    String::JoinFromList(       -- calling a C++ UDF,
                                -- see below

        AGGREGATE_LIST(title),  -- an aggregate function that
                                -- returns all the passed values as a list

        ", "                    -- String::JoinFromList concatenates
                                -- items of a given list (the first argument)
                                -- to a string using the separator (the second argument)
    ) AS episode_titles
FROM episodes
WHERE series_id IN (1,2)        -- IN defines the set of values in the WHERE clause,
                                -- to be included into the result.
                                -- Syntax:
                                -- test_expression (NOT) IN
                                -- ( subquery | expression ` ,...n ` )
                                -- If the value of test_expression is equal
                                -- to any value returned by subquery or is equal to
                                -- any expression from the comma-separated list,
                                -- the result value is TRUE. Otherwise, it's FALSE.
                                -- using NOT IN negates the result of subquery
                                -- or expression.
                                -- Warning: using null values together with
                                -- IN or NOT IN may lead to undesirable outcomes.
AND season_id = 1
GROUP BY
    CASE                        -- CASE evaluates a list of conditions and
                                -- returns one of multiple possible resulting
                                -- expressions. CASE can be used in any
                                -- statement or with any clause
                                -- that supports a given statement. For example, you can use CASE in
                                -- statements such as SELECT, UPDATE, and DELETE,
                                -- and in clauses such as IN, WHERE, and ORDER BY.
        WHEN series_id = 1
        THEN "IT Crowd"
        ELSE "Other series"
    END AS series_title         -- GROUP BY can be performed on
                                -- an arbitrary expression.
                                -- The result is available in a SELECT
                                -- via the alias specified with AS.
;

COMMIT;
```

Joining tables with JOIN

Merge the columns of the source tables `seasons` and `series`, then output all the seasons of the IT Crowd series to the resulting table using the JOIN operator.

> **Note**
>
> We assume that you already created tables in step Creating a table and populated them with data in step Adding data to a table.

```
SELECT
    sa.title AS season_title,    -- sa and sr are "join names",
    sr.title AS series_title,    -- table aliases declared below using AS.
    sr.series_id,                -- They are used to avoid
    sa.season_id                 -- ambiguity in the column names used.

FROM
    seasons AS sa
INNER JOIN
    series AS sr
ON sa.series_id = sr.series_id
WHERE sa.series_id = 1
ORDER BY                         -- Sorting of the results.
    sr.series_id,
    sa.season_id                 -- ORDER BY sorts the values by one column
;                                -- or multiple columns.
                                 -- Columns are separated by commas.

COMMIT;
```

Inserting and updating data with REPLACE

Add data to the table using REPLACE INTO:

> **Note**
>
> We assume that you already created tables in step Creating a table and populated them with data in step Adding data to a table.

```
REPLACE INTO episodes
(
    series_id,
    season_id,
    episode_id,
    title,
    air_date
)
VALUES
(
    2,
    5,
    12,
    "Test Episode !!!",
    CAST(Date("2018-08-27") AS Uint64)
)
;

-- COMMIT is called so that the next SELECT operation
-- can see the changes made by the previous transaction.
COMMIT;

-- View result:
SELECT * FROM episodes WHERE series_id = 2 AND season_id = 5;

COMMIT;
```

Inserting and updating data with UPSERT

Add data to the table using UPSERT INTO:

> **Note**
>
> We assume that you already created tables in step Creating a table and populated them with data in step Adding data to a table.

```
UPSERT INTO episodes
(
    series_id,
    season_id,
    episode_id,
    title,
    air_date
)
VALUES
(
    2,
    5,
    13,
    "Test Episode",
    CAST(Date("2018-08-27") AS Uint64)
)
;

COMMIT;

-- View result:
SELECT * FROM episodes WHERE series_id = 2 AND season_id = 5;

COMMIT;
```

Inserting data with INSERT

Add data to the table using INSERT INTO:

> **Note**
>
> We assume that you already created tables in step Creating a table and populated them with data in step Adding data to a table.

```
INSERT INTO episodes
(
    series_id,
    season_id,
    episode_id,
    title,
    air_date
)
VALUES
(
    2,
    5,
    21,
    "Test 21",
    CAST(Date("2018-08-27") AS Uint64)
),                                          -- Rows are separated by commas.
(
    2,
    5,
    22,
    "Test 22",
    CAST(Date("2018-08-27") AS Uint64)
)
;

COMMIT;

-- View result:
SELECT * FROM episodes WHERE series_id = 2 AND season_id = 5;

COMMIT;
```

Updating data with UPDATE

Update data in the table using the UPDATE operator:

> **Note**
>
> We assume that you already created tables in step Creating a table and populated them with data in step Adding data to a table.

```
UPDATE episodes
SET title="test Episode 2"
WHERE
    series_id = 2
    AND season_id = 5
    AND episode_id = 12
;

COMMIT;

-- View result:
SELECT * FROM episodes WHERE series_id = 2 AND season_id = 5;

-- YDB doesn't see changes that take place at the start of the transaction,
-- which is why it first performs a read. You can't UPDATE or DELETE a table
-- already changed within the current transaction. UPDATE ON and
-- DELETE ON let you read, update, and delete multiple rows from one table
-- within a single transaction.

$to_update = (
    SELECT series_id,
           season_id,
           episode_id,
           Utf8("Yesterday's Jam UPDATED") AS title
    FROM episodes
    WHERE series_id = 1 AND season_id = 1 AND episode_id = 1
);

SELECT * FROM episodes WHERE series_id = 1 AND season_id = 1;

UPDATE episodes ON
SELECT * FROM $to_update;

COMMIT;

-- View result:
SELECT * FROM episodes WHERE series_id = 1 AND season_id = 1;

COMMIT;
```

Deleting data

Delete data from the table using DELETE.

> **Note**
>
> We assume that you already created tables in step Creating a table and populated them with data in step Adding data to a table.

```
DELETE
FROM episodes
WHERE
    series_id = 2
    AND season_id = 5
    AND episode_id = 12
;

COMMIT;

-- View result:
SELECT * FROM episodes WHERE series_id = 2 AND season_id = 5;

-- YDB doesn't see changes that take place at the start of the transaction,
-- which is why it first performs a read. It is impossible to execute UPDATE or DELETE on
-- if the table was changed within the current transaction. UPDATE ON and
-- DELETE ON let you read, update, and delete multiple rows from one table
-- within a single transaction.

$to_delete = (
    SELECT series_id, season_id, episode_id
    FROM episodes
    WHERE series_id = 1 AND season_id = 1 AND episode_id = 2
);

SELECT * FROM episodes WHERE series_id = 1 AND season_id = 1;

DELETE FROM episodes ON
SELECT * FROM $to_delete;

COMMIT;

-- View result:
SELECT * FROM episodes WHERE series_id = 1 AND season_id = 1;

COMMIT;
```

Adding and deleting columns

Add a new column to the table and then delete it.

> **Note**
>
> We assume that you already created tables in step Creating a table and populated them with data in step Adding data to a table.

Adding a column

Add a non-key column to the existing table:

```
ALTER TABLE episodes ADD COLUMN viewers Uint64;
```

Deleting a column

Delete the column you added from the table:

```
ALTER TABLE episodes DROP COLUMN viewers;
```

Deleting a table

Delete the created tables using the DROP TABLE statement.

```
DROP TABLE episodes;
DROP TABLE seasons;
DROP TABLE series;
```

Choosing a primary key

The recommendations for choosing a proper primary key for a table depend on its type:

- Row-oriented tables
- Column-oriented tables

Selecting a primary key for maximum performance

The way columns are selected for a table's primary key defines YDB's ability to scale load and improve performance.

General recommendations for choosing a primary key:

- Avoid situations where the main load falls on one partition of a table. The more evenly load is distributed across partitions, the better the performance.
- Reduce the number of partitions that can be affected in a single request. Moreover, if the request affects no more than one partition, it is performed using a special simplified protocol. This significantly increases the speed and saves the resources.

All YDB tables are sorted by primary key in ascending order. In a table with a monotonically increasing primary key, this will result in new data being added at the end of a table. As YDB splits table data into partitions based on key ranges, inserts are always processed by the same server that is responsible for the "last" partition. Concentrating the load on a single server results in slow data uploading and inefficient use of a distributed system.

You get the same behaviour if the monotonically increasing value is the **only** primary-key column or the **first component of a composite primary key**: partition ranges are driven by the key prefix, so monotonicity at the beginning of the key produces the same hot-spot pattern. Auto-increment types (`SERIAL`), when used as the sole or leading component of `PRIMARY KEY`, behave in the same way.

As an example, let's take logging of user events to a table with the `( timestamp, userid, userevent, PRIMARY KEY (timestamp, userid) )` schema.

The values in the `timestamp` column increase monotonically resulting in all new records being added at the end of a table, and the final partition, which is responsible for this range of keys, handles all the table inserts. This makes scaling insert loads impossible and performance will be limited by the single process servicing this partition and won't increase as new servers are added to a cluster.

YDB supports further automatic partition splitting upon a threshold size or load being reached. However, in this situation, once it splits off, the new partition will again begin handling all the inserts, and the situation will recur.

Techniques that let you evenly distribute load across table partitions

Changing the sequence of key components

Writing data to a table with the `( timestamp, userid, userevent, PRIMARY KEY (timestamp, userid) )` schema results in an uneven load on table partitions due to a monotonically increasing primary key. Changing the sequence of key components so that the monotonically increasing part isn't the first component can help distribute the load more evenly. If you redefine a table's primary key as `PRIMARY KEY (userid, timestamp)`, the DB writes will distribute more evenly across the partitions provided there is a sufficient number of users generating events.

Using a hash of key column values as a primary key

To obtain a more even distribution of operations across a table's partitions and reduce the size of internal data structures, make the primary key "prefix" (initial part) values more varied. To do this, make the primary key include the value of a hash of the entire primary key or a part of the primary key.

For instance, the schema of this table with the schema `( timestamp, userid, userevent, PRIMARY KEY (userid, timestamp) )` might be made to include an additional field computed as a hash: `userhash = HASH(userid)`. This would change the table schema as follows:

```
( userhash, userid, timestamp, userevent, PRIMARY KEY (userhash, userid, timestamp) )
```

If you select the hash function properly, rows will be distributed fairly evenly throughout the entire key space, which will result in a more even load on the system. At the same time, the fact that the key includes `userid, timestamp` after `userhash` keeps the data local and sorted by time for a specific user.

The `userhash` field in the example above must be computed by the application and specified explicitly both for inserting new records into the table and for data access by primary key.

Another common special case of the same idea is monotonically increasing identifiers (`SERIAL` , a sequence, or a sequential numeric id)—not only the timestamp from the log example above.

Monotonically increasing identifier (SERIAL, sequence, auto increment)

Don't use a monotonically increasing value as the sole primary-key column or as the **first** component of a composite primary key. Typical examples are an auto-increment id (`SERIAL`), a value from a sequence, or a sequential numeric id.

Row-oriented tables in YDB are partitioned by ranges of the primary key, so rows with a monotonic key keep landing in the last partition and create a hot spot: most write load concentrates on a small set of partitions while others are underused.

The same issue applies to a composite key when the **first** component increases monotonically—for example, `PRIMARY KEY (order_id, customer_id)`.

YDB supports auto-increment types (`SERIAL`), but using them as the only or leading primary-key column isn't recommended for insert-heavy workloads.

Prefer a random identifier (such as UUID) or a composite business key without a monotonic prefix whenever possible. If you must keep a monotonically increasing id in the key, use a composite primary key where the **first** column is a hash of the id and the **second** is the id itself. That tends to yield **more even** distribution of rows across partitions while preserving uniqueness (a hash does not guarantee a perfectly balanced split).

Poor example

```
CREATE TABLE orders (
    order_id Uint64,
    customer_id Uint64,
    amount Double,
    PRIMARY KEY (order_id)
);

INSERT INTO orders (order_id, customer_id, amount) VALUES
    (1001, 10, 999.99),
```

```
    (1002, 11, 1499.00),
    (1003, 12, 499.50);

SELECT *
FROM orders
WHERE order_id = 1001;
```

Good example

```
CREATE TABLE orders (
    order_hash Uint64,
    order_id Uint64,
    customer_id Uint64,
    amount Double,
    PRIMARY KEY (order_hash, order_id)
);

INSERT INTO orders (order_hash, order_id, customer_id, amount) VALUES
    (Digest::NumericHash(1001), 1001, 10, 999.99),
    (Digest::NumericHash(1002), 1002, 11, 1499.00),
    (Digest::NumericHash(1003), 1003, 12, 499.50);

SELECT *
FROM orders
WHERE order_hash = Digest::NumericHash(1001)
    AND order_id = 1001;
```

Compute `order_hash` on write and on keyed reads—the same rule as for `userhash` in the example above.

Reducing the number of partitions affected by a single query

Let's assume that the main scenario for working with table data is to read all events by a specific `userid`. Then, when you use the `( timestamp, userid, userevent, PRIMARY KEY (timestamp, userid) )` table schema, each read affects all the partitions of the table. Moreover, each partition is fully scanned, since the rows related to a specific `userid` are located in an order that isn't known in advance. Changing the sequence of `( timestamp, userid, userevent, PRIMARY KEY (userid, timestamp) )` key components causes all rows related to a specific `userid` to follow each other. This row distribution will be useful for reading data by `userid` and will reduce load.

NULL value in a key column

In YDB, all columns, including key ones, may contain a NULL value. Using NULL as values in key columns isn't recommended. According to the SQL standard (ISO/IEC 9075), you can't compare NULL with other values. Therefore, the use of concise SQL statements with simple comparison operators may lead, for example, to skipping rows containing NULL during filtering.

Row size limit

To achieve high performance, we don't recommend writing rows larger than 8 MB and key columns larger than 2 KB to the DB.

Choosing keys for maximum column-oriented table performance

Unlike row-oriented YDB tables, column-oriented tables are partitioned by designated **partitioning keys**. Within each partition, data is distributed based on the table's primary key.

Partitioning key

The partitioning key must be a non-empty subset of the primary key columns. The hash of the partition key determines the partition to which the row belongs. The partition key should be chosen to ensure that data is evenly distributed across partitions. This is typically achieved by including high-cardinality columns, such as high-resolution timestamps (`Timestamp` data type), in the partition key. Using a partitioning key with low cardinality can lead to an uneven distribution of data across partitions, causing some partitions to become overloaded. Overloaded partitions may result in suboptimal query performance and/or limit the maximum rate of data insertion.

Column-oriented tables do not support automatic repartitioning at the moment. That's why it's important to specify a realistic number of partitions at table creation. You can evaluate the number of partitions you need based on the expected data amounts you are going to add to the table. The average insert throughput for a partition is 1 MB/s. The throughput is mostly affected by the selected primary keys (the need to sort data inside the partition when inserting data). We do not recommend setting up more than 128 partitions for small data streams.

Primary key

The primary key determines how the data will be stored inside the partition. That's why, when selecting a primary key, you need to keep in mind both the effectiveness of reading data from the partition and the effectiveness of inserting data into the partition. The optimum insert use case is to write data to the beginning or end of the table, making rare local updates of previously inserted data. For example, an effective use case would be to store application logs by timestamps, adding records to the end of the partition using the current time in the primary key.

Example

When your data stream is 1 GB per second, an analytical table with 1,000 partitions is an optimal choice. Nevertheless, it is not advisable to create tables with an excessive number of partitions: this could raise resource consumption in the cluster and negatively impact the query rate.

Secondary indexes

Indexes are auxiliary structures within databases that help find data by certain criteria without having to search an entire database and retrieve sorted samples without actually sorting, which would require processing the entire dataset.

Data in a YDB table is always sorted by the primary key. That means that retrieving any entry from the table with specified field values comprising the primary key always takes the minimum fixed time, regardless of the total number of table entries. Indexing by the primary key makes it possible to retrieve any consecutive range of entries in ascending or descending order of the primary key. Execution time for this operation depends only on the number of retrieved entries rather than on the total number of table records.

To use a similar feature with any field or combination of fields, additional indexes called **secondary indexes** can be created for them.

In transactional systems, indexes are used to limit or avoid performance degradation and increase of query cost as your data grows.

This article describes the main operations with secondary indexes and gives references to detailed information on each operation. For more information about various types of secondary indexes and their specifics, see Secondary indexes in the Concepts section.

Creating secondary indexes

A secondary index is a data schema object that can be defined when creating a table with the `CREATE TABLE` YQL command or added to it later with the `ALTER TABLE` YQL command.

The `table index add` command is supported in the YDB CLI.

Since an index contains its own data derived from table data, when creating an index on an existing table with data, an operation is performed to initially build an index. This may take a long time. This operation is executed in the background and you can keep working with the table while it's in progress. However, you can't use the new index until its build is completed.

An index can only be used in the order of the fields included in it. If an index contains two fields, such as `a` and `b`, you can effectively use it for queries such as:

- `WHERE a = $var1 AND b = $var2`.
- `WHERE a = $var1`.
- `WHERE a > $var1` and other comparison operators.
- `WHERE a = $var1 AND b > $var2` and any other comparison operators in which the first field must be checked for equality.

This index can't be used in the following queries:

- `WHERE b = $var1`.
- `WHERE a > $var1 AND b > $var2`, which is equivalent to `WHERE a > $var1` in terms of applying the index.
- `WHERE b > $var1`.

To effectively limit the result of a query using the `LIMIT` command, you must use one-way sorting in the `ORDER BY` command:

- `WHERE a = $var1 AND b > $var2 ORDER BY a, b DESC LIMIT 1`, all index rows that match the filter conditions will be read;
- `WHERE a = $var1 AND b > $var2 ORDER BY a DESC, b DESC LIMIT 1`, only one row will be read.

Considering the above, there's no use in pre-indexing all possible combinations of table columns to speed up the execution of any query. An index is always a compromise between the lookup and write speed and the storage space occupied by the data. Indexes are created for specific queries and search criteria made by an app in the database.

Using secondary indexes when selecting data

For a table to be accessed by a secondary index, its name must be explicitly specified in the `VIEW` section after the table name as described in the article about the YQL `SELECT` statement. For example, a query to retrieve orders from the `orders` table by the specified customer ID (`id_customer`) looks like this:

```
DECLARE $customer_id AS Uint64;

SELECT *
FROM   orders VIEW idx_customer AS o
WHERE  o.id_customer = $customer_id
```

Where `idx_customer` is the name of the secondary index on the `orders` table with the `id_customer` field specified first.

If no `VIEW` section is specified, making a query like this results in a full scan of the `orders` table.

In transactional applications, such information queries are executed with paginated data output. This eliminates an increase in the cost and time of query execution if the number of entries that meet the filtering conditions grows. The described approach to writing paginated queries using the primary key can also be applied to columns that are part of a secondary index.

An experimental capability to automatically select a secondary index for use in a query is also available. The selection algorithm is currently rule-based and uses only the query text to automatically select a secondary index.

Automatic use of indexes when selecting data

Explicitly specifying the `VIEW` section takes priority over the optimizer's decision to use secondary indexes. That is, the query

```
SELECT * FROM `Table` VIEW Index
```

is guaranteed to perform selection using the `Index` index.

To explicitly specify reading using the primary key, use the following construct:

```
SELECT * FROM `Table` VIEW PRIMARY KEY
```

Criteria for selecting a secondary index

The index used for reading is selected during query optimization when determining the row ranges that need to be read (predicate pushdown). Indexes, like the main table, are a set of rows ordered by a set of key columns.

Selection between reading using an index and reading using the primary key is based on the following metrics:

1. Need for additional reads from the main table. If the index contains all columns required for the query, no additional reads are needed.
2. Length of the point prefix of the predicate for the corresponding table's key. That is, the predicate constrains a set of columns that are the first components of the key with point conditions: `=`, `IN`, `IS NULL`. Priority is given to indexes for which all indexed columns are fixed, or to the main table if the entire primary key is point-fixed.
3. Number of columns used in the read range bounds. In the following query to the `Table` table with primary key (Key1, Key2, Key3)

```
SELECT * FROM `Table` WHERE (Key1, Key2, Key3) < ($param1, $param2, $param3) AND (Key1, Key2) > ($param4, $pa
ram5)
```

reading will be performed in the range `(($param4, $param5), ($param1, $param2, $param3))`, and thus the number of used columns will be 3. Similarly, preference is given to indexes for which all indexed columns are used.

Read methods are ranked against each other according to criterion 2, with ties broken by criterion 3, and criterion 1 is additionally taken into account.

Examples of automatic index selection

```
CREATE TABLE `Table` (
     Key Int32,
     SubKey1 Int32,
     SubKey2 String,
     Value1 String,
     Value2 String,
     PRIMARY KEY (Key, SubKey1, SubKey2),
     INDEX Index12 GLOBAL ON (SubKey1, SubKey2),
     INDEX Index21 GLOBAL ON (SubKey2, Value1),
     INDEX Index212 GLOBAL ON (SubKey2) COVER (Value2)
);
```

`SELECT * FROM Table WHERE SubKey1 = $p1 and SubKey2 > $p2` — `Index12` will be used. The range expression is `(($p1; $p2), ($p1)]`. The point prefix length for `Index12` is 1; for other indexes, it is 0.

`SELECT * FROM Table WHERE Key = $p1 and SubKey1 = $p2 And SubKey2 = $p2` — no index will be used. When selecting a scan of the main table, all 3 columns are used: `[Key, Fk1, Fk2]`, and the point prefix length is 3.

`SELECT * FROM Table WHERE Key = $p1 and SubKey2 = $p2` — secondary indexes will not be used. When selecting any secondary index, 1 column is used, and the point prefix is also at most 1 for any index choice.

`SELECT * FROM Table WHERE Key >= $p1 and SubKey1 = $p2 And SubKey2 = $p3` — `Index12` should be selected, because when it is selected, the resulting range `[[Fk1; Fk2; Key], [Fk1; Fk2])` yields a point prefix length of 2, and 3 columns are used.

`SELECT * FROM Table WHERE Key = 2 and SubKey2 = 3` — secondary indexes should not be used. When reading by `PK` or using any of the secondary indexes, the point prefix consists of at most one column. Also, at most one column is used.

`SELECT * FROM Table WHERE SubKey1 > 2` — `Index12` should be selected. Only when using `Index12` will the read range be non-trivial.

`SELECT * FROM Table WHERE SubKey2 = 2` — either `Index21` or `Index212` may be selected. When using the indexes mentioned above, the point prefix length will be 1. The number of used columns is also maximized when selecting `Index21` and `Index212`.

`SELECT Value2 FROM Table WHERE SubKey2 = 2` — `Index212` should be selected. When using `Index21` and `Index212`, the point prefix length will be 1, but when using `Index212`, no read from the main table is required.

`SELECT * FROM Table WHERE SubKey2 > 2` — `Index21` or `Index212` will be used, because the read range is non-trivial only when using them.

`SELECT * FROM Table WHERE SubKey1 = 2` — `Index12` will be used, because when using it, the point prefix length will be 1, whereas in other cases it is 0.

Checking the cost of queries

Any query made in a transactional application should be checked in terms of the number of I/O operations it performed in the database and how much CPU was used to run it. You should also make sure these indicators don't continuously grow as the database volume grows. YDB returns statistics required for the analysis after running each query.

If you use the YDB CLI, select the `--stats` option to enable printing statistics after executing the `yql` command. All YDB SDKs also contain structures with statistics returned after running a query. If you make a query in the UI, you'll see a tab with statistics next to the results tab.

Updating data using a secondary index

The `UPDATE`, `UPSERT`, and `REPLACE` YQL record modification statements don't permit specifying a secondary index to perform a search for data, so an attempt to make an `UPDATE ... WHERE indexed_field = $value` will result in a full scan of the table. To avoid this, you can first run `SELECT` by index to get the primary key value and then run `UPDATE` by the primary key. You can also use `UPDATE ON`.

To update data in the `table1` table, run the query:

```
$to_update = (
    SELECT pk_field, $f1 AS field1, $f2 AS field2, ...
    FROM   table1 VIEW idx_field3
    WHERE  field3 = $f3)

UPDATE table1 ON SELECT * FROM $to_update
```

Note

Currently, data updating is possible only using a synchronous secondary index. This limitation exists because data modification is permitted only in Serializable transactions, and accessing asynchronous indices would violate the guarantees of this transaction mode.

Deleting data using a secondary index

To delete data by secondary index, use `SELECT` with a predicate by secondary index and then call `DELETE ON`.

To delete all data about series with zero views from the `series` table, run the query:

```
DELETE FROM series ON
SELECT series_id
FROM series VIEW views_index
WHERE views = 0;
```

Note

Currently, deleting data is possible only using a synchronous secondary index. This is because data removal is permitted only in Serializable transactions, and accessing asynchronous indices would violate the guarantees of this transaction mode.

Atomic replacement of a secondary index

You can atomically replace a secondary index. This can be useful if you want your index to become covering. This operation is totally transparent for your running applications: when you replace the index, the compiled queries are invalidated.

To replace an existing index atomically, use the YDB CLI command ydb table index rename with the `--replace` parameter.

Performance of data writes to tables with secondary indexes

You need additional data structures to enable secondary indexes. Support for these structures makes table data update operations more costly.

During synchronous index updates, a transaction is only committed after all the necessary data is written in both a table and synchronous indexes. As a result, it takes longer to execute it and makes it necessary to use distributed transactions even if adding or updating entries in a single partition.

Indexes that are updated asynchronously let you use single-shard transactions. However, they only guarantee eventual consistency and still put a load on the database.

Vector Indexes

Vector indexes are a specialized type of secondary index that enable efficient vector search in multidimensional spaces. While traditional secondary indexes optimize searching by equality or range, vector indexes allow similarity searching based on similarity or distance functions.

Data in a YDB table is stored and sorted by the primary key, ensuring efficient searching by exact match and range scanning. Vector indexes provide similar efficiency for nearest neighbor searches in vector spaces.

Types of Vector Indexes

A vector index can be global or global filtered. Indexes of any of these types can also be covering and include a copy of additional column data from the main table.

Global Vector Index

A global vector index on the `embedding` column enables fast approximate nearest neighbor search across the entire table:

```
ALTER TABLE my_table
  ADD INDEX my_index
  GLOBAL USING vector_kmeans_tree
  ON (embedding)
  COVER (embedding, data)
  WITH (distance=cosine, vector_type="float", vector_dimension=512, overlap_clusters=3);
```

Example search query using this index:

```
PRAGMA ydb.KMeansTreeSearchTopSize = "10";

DECLARE $query_vector AS string;

$query_vector = Knn::ToBinaryStringFloat([1.0, 1.2, 0, ...]);

SELECT user, data
FROM my_table VIEW my_index
ORDER BY Knn::CosineSimilarity(embedding, $query_vector) DESC
LIMIT 10;
```

Note that:

- Both the `embedding` column and the `$query_vector` parameter must be of string type and contain an array of numbers in the simple binary format.
- It is more efficient to pass the parameter from the SDK as a string by serializing the numbers on the application side (examples). Alternatively, the value can be passed from the SDK as a vector of numbers and converted from a list using `Knn::ToBinaryString*` functions, but this is slower.
- The `COVER (embedding, data)` clause is optional and is used to create a covering index. This helps further speed up the search.
- Vector index search is always approximate — its results differ from a full-scan search.
- Increasing the `PRAGMA KMeansTreeSearchTopSize` parameter improves search quality (recall) at the cost of speed. The parameter sets the number of index clusters nearest to the query that are scanned. The default value is 1 (minimum quality, maximum speed).
- The `overlap_clusters=3` parameter significantly improves future search quality during indexing by specifying the number of clusters each vector is added to, but increases the index size.
- The `vector_type` and `vector_dimension` parameters can be omitted if the table is not empty — they will be autodetected from existing rows.

Filtered Vector Index

A filtered vector index enables searching for nearest neighbors within each category defined by unique values of additional columns.

To create such an index, specify multiple index columns. The last column must be the vector column; the others (category columns) can be of any type:

```
ALTER TABLE my_table
  ADD INDEX my_index
  GLOBAL USING vector_kmeans_tree
  ON (user, embedding)
  COVER (embedding, data)
  WITH (distance=cosine, vector_type="float", vector_dimension=512);
```

Search queries using this filtered index must include conditions on the `user` column:

```
PRAGMA ydb.KMeansTreeSearchTopSize = "10";

DECLARE $query_vector AS string;

$query_vector = Knn::ToBinaryStringFloat([1.0, 1.2, 0, ...]);

SELECT user, data
FROM my_table VIEW my_index
WHERE user = 'john'
ORDER BY Knn::CosineSimilarity(embedding, $query_vector) DESC
LIMIT 10;
```

Indexing and search parameters work the same as for a global index.

Covering Vector Index

A covering vector index stores a copy of additional column data to avoid reading from the main table and further speed up the search.

Note that by default the index does not contain a copy of the vector column (in the example — `embedding`), so if it is not explicitly added to the list of covered columns, reading from the main table cannot be avoided, since vectors are always used for exact result sorting in the final search step.

```
ALTER TABLE my_table
  ADD INDEX my_index
  GLOBAL USING vector_kmeans_tree
  ON (embedding)
  COVER (embedding, data)
  WITH (distance=cosine);
```

Distance Functions

The following similarity or distance functions are supported:

- `distance=cosine` or `similarity=cosine` — cosine distance, corresponds to `ORDER BY Knn::CosineDistance(...) ASC` or `ORDER BY Knn::CosineSimilarity(...) DESC`.
- `distance=manhattan` — Manhattan distance (L1 metric), corresponds to `ORDER BY Knn::ManhattanDistance(...) ASC`.
- `distance=euclidean` — Euclidean distance (L2 metric), corresponds to `ORDER BY Knn::EuclideanDistance(...) ASC`.
- `similarity=inner_product` — inner product, corresponds to `ORDER BY Knn::InnerProductSimilarity(...) DESC`.

Full Vector Index Syntax

Creating a vector index:

- During table creation: CREATE TABLE.
- Adding to an existing table: ALTER TABLE.

Full syntax for queries using a vector index:

- VIEW VECTOR INDEX.

Search Algorithm

The current implementation offers one type of index: `vector_kmeans_tree`.

Vector Index Type `vector_kmeans_tree`

The `vector_kmeans_tree` index implements hierarchical data clustering. The structure of the index includes:

1. Hierarchical clustering:
   - the index builds multiple levels of k-means clusters;
   - at each level, vectors are distributed across a predefined number of clusters raised to the power of the level;
   - the first level clusters the entire dataset;
   - subsequent levels recursively cluster the contents of each parent cluster.
2. Search process:
   - search proceeds recursively from the first level to the subsequent ones;
   - during queries, the index analyzes only the most promising clusters;
   - such search space pruning avoids complete enumeration of all vectors.
3. Parameters:
   - `levels`: number of levels in the tree, defining search depth (recommended 1-3);
   - `clusters`: number of clusters in k-means, defining search width (recommended 64-512).
   - `overlap_clusters`: number of leaf-level clusters each vector is added to (recommended 3).

Internally, a vector index consists of index tables named `indexImpl*Table`. In selection queries using the vector index, these tables appear in query statistics. For more on the structure of the vector index, see the dedicated article Vector index type: vector_kmeans_tree.

Overlapping Clusters

A vector index in YDB can add each vector to multiple clusters to improve search recall and speed:

```
ALTER TABLE my_table
  ADD INDEX my_index
  GLOBAL USING vector_kmeans_tree
  ON (embedding)
  WITH (distance=cosine, overlap_clusters=3);
```

In this example, each vector will be added to 3 nearest clusters instead of 1.

The `overlap_clusters` parameter is recommended for nearly all use cases, especially for vector indexes with `levels > 1`, as it significantly improves search recall even with small `PRAGMA KMeansTreeSearchTopSize` values (for example, 3).

This way, you can reduce the PRAGMA value and significantly speed up the search while maintaining the same recall.

Partitioning of Index Tables

The most heavily loaded table in a vector index is `indexImplLevelTable`, the cluster structure table. Every search query reads this table, so load on its partitions may limit query performance.

To improve performance, you can enable auto-partitioning by load:

```
ALTER TABLE `my_table/my_index/indexImplLevelTable`
SET AUTO_PARTITIONING_BY_LOAD ENABLED;
```

Or by size:

```
ALTER TABLE `my_table/my_index/indexImplLevelTable`
SET AUTO_PARTITIONING_PARTITION_SIZE_MB 100;
```

The same settings can be applied to other index tables (`indexImplPostingTable` and `indexImplPrefixTable`), but the Level table is the most loaded while being small in size, so auto-partitioning settings are most relevant for it.

Using Index Table Replicas

Another way to speed up search is to use table replicas. To do this:

1. Create a covering index so that only index tables are involved in search queries.
2. Enable replicas on all index tables:

   ```
   ALTER TABLE `my_table/my_index/indexImplLevelTable` SET READ_REPLICAS_SETTINGS 'PER_AZ:3';
   ALTER TABLE `my_table/my_index/indexImplPostingTable` SET READ_REPLICAS_SETTINGS 'PER_AZ:3';
   ```

   And, for a filtered index, also:

   ```
   ALTER TABLE `my_table/my_index/indexImplPrefixTable` SET READ_REPLICAS_SETTINGS 'PER_AZ:3';
   ```

3. Use the Stale Read-Only query mode.

Updating Vector Indexes

Updating vector indexes has the following limitations:

Clusters are not recalculated during update

When a table with a vector index is updated, its internal structure — a tree of clusters (groups of similar vectors) — is not recalculated. New or modified records are simply assigned to existing clusters.

Over time, this can lead to index degradation, resulting in:

1. Reduced completeness — the index may return fewer relevant results because clusters no longer reflect the actual data distribution.
2. Reduced performance — unbalanced clusters (for example, one cluster containing too many records) can slow down search queries and, in the worst case, lead to full table scans.

The extent of degradation depends on the nature of the updates:

- If the index was built on a representative sample (e.g., a random 50% of the data) and the remaining records are added later, the index structure remains mostly relevant, and degradation is minimal.
- If entire groups of similar vectors were absent from the initial dataset, the clustering may fail to partition the space effectively, leading to a significant drop in result relevance.

A particularly problematic corner case arises when a vector index is created on an empty table. In this scenario, the index consists of a single cluster, and all new records are placed within it. As a result, searches using such an index are equivalent to full table scans.

To prevent degradation:

- Avoid creating a vector index on an empty table.
- If a large volume of new data has been added, build a new index and atomically replace the old index with the updated one.

Update inconsistency during index build

Vector indexes do not support consistent updates during build. That is, a vector index is not updated when data in the main table is modified until the index build is finished.

This means that if you want a vector index to remain 100% consistent, you have to pause table updates while it is being built.

Updates are not blocked automatically because vector index search is approximate by nature, and in many cases temporary inconsistency during the build is acceptable.

Recipes for Working with Vector Indexes

To get started with vector indexes, you can use the following recipes:

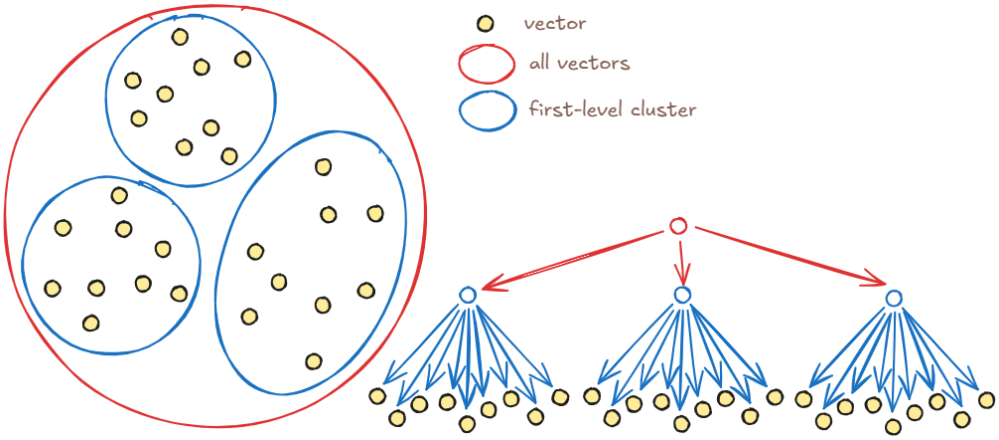
- YDB CLI & YQL

Vector index type: vector_kmeans_tree

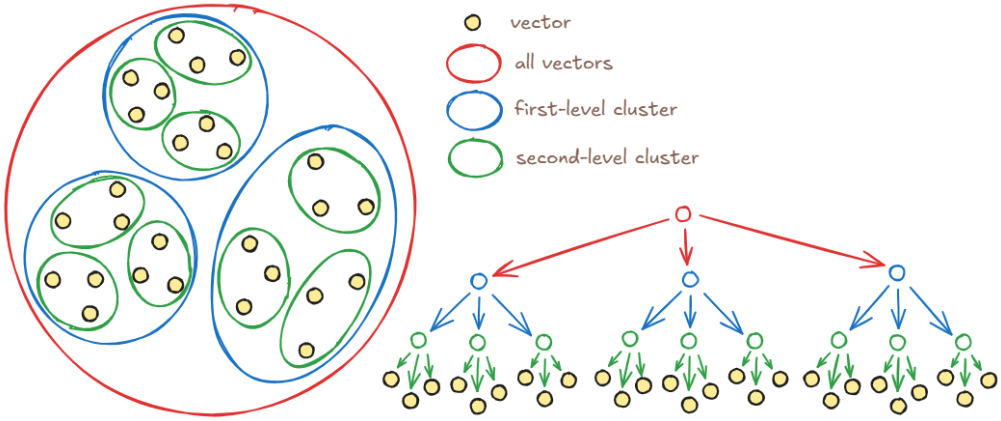
Vector indexes are specialized data structures that enable efficient vector search in multidimensional spaces. They are described in the overview; this article details the `vector_kmeans_tree` vector index type.

Creating the vector index

The `vector_kmeans_tree` algorithm recursively partitions all vectors to be searched into clusters using K-means. First, vectors are split into the number of clusters specified when creating the index via the vector index parameter `clusters`. For example, first-level clusters might look like this:



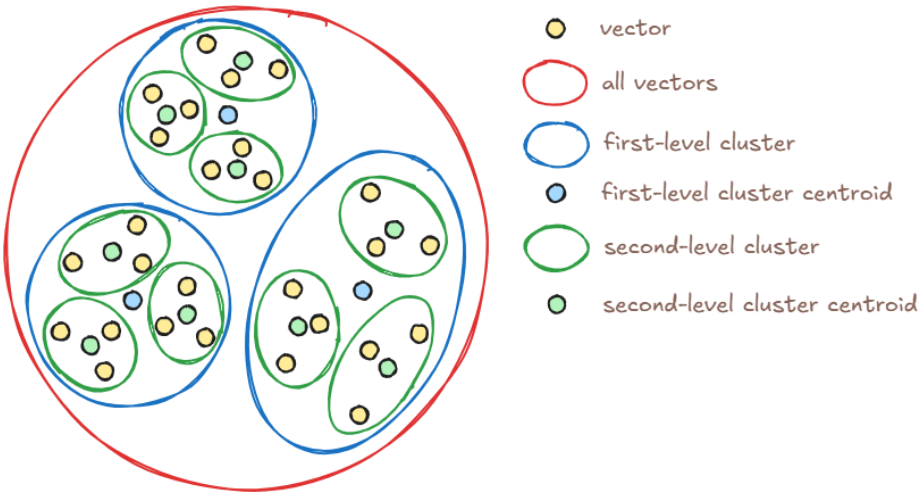
Each cluster is then split into the same number of clusters. Because each first-level cluster contains second-level clusters, a cluster tree is formed. The number of levels in the tree is set by the vector index parameter `levels`. For example, second-level clusters look like this:



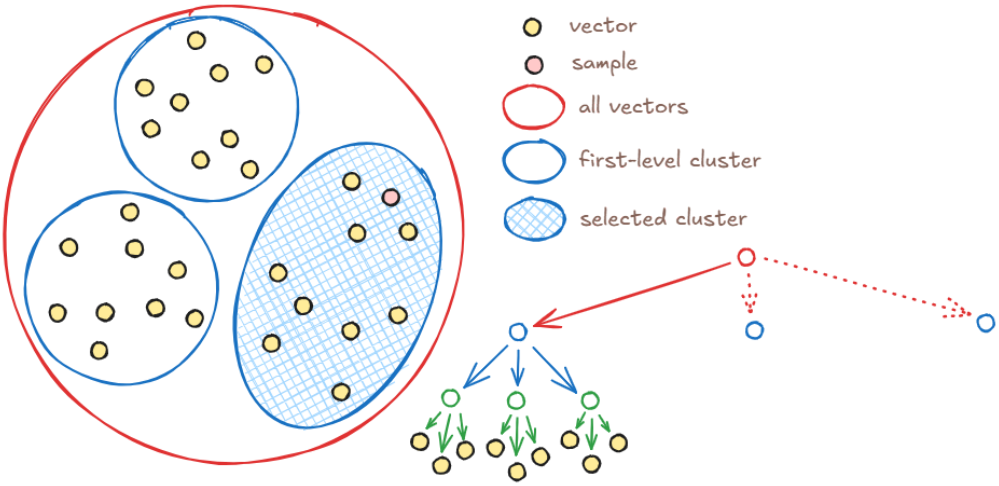
The vector index tree will have three first-level clusters, each containing three second-level clusters, each of which in turn contains three vectors.

Search in the vector index

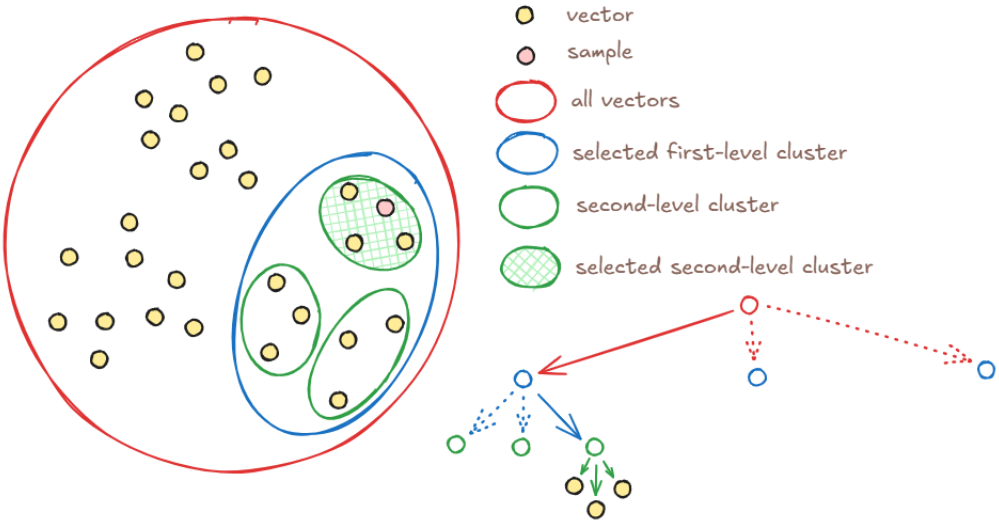
For search in a `vector_kmeans_tree` vector index, one of several clusters is chosen at each level; that cluster contains vectors similar to the query vector. To choose the right cluster, a centroid — the arithmetic mean of all vectors in the cluster — is computed and stored for each cluster.



After choosing a cluster at one level, YDB moves to the next level and considers the clusters that belong to the cluster chosen at the previous level. YDB compares distances between cluster centroids and the query vector and selects the cluster with the smallest distance. Approximate search assumes that if a cluster's centroid is closest to the query, the vectors in that cluster are also closest to the query.



Each first-level cluster is split into second-level clusters, and so on. The algorithm “descends” the tree step by step, each time selecting a smaller cluster until it reaches a leaf cluster that holds vectors instead of centroids. YDB scans these vectors and returns the requested number of nearest neighbors. By limiting the number of clusters at each level, YDB keeps search complexity at `O(1)`.



If a leaf cluster has fewer vectors than requested, YDB returns all vectors found. To improve recall in such cases, use the `KMeansTreeSearchTopSize` parameter described below.

Index parameters

YDB lets you configure the number of levels in the vector index with the `levels` parameter and the number of clusters per level with the `clusters` parameter. Choose these values based on the expected number of vectors and cluster characteristics: network latency between nodes, network throughput, and CPU performance.

Clusters in the vector index tree are stored in distributed storage. On each step down the tree, YDB reads from distributed storage and receives binary data with the list of vectors over the network. The time for this operation depends on datacenter hardware, network load, and the amount of data transferred between clusters.

Choosing one cluster at a level requires scanning all centroid vectors and computing the distance to the query. You can estimate scan speed on your hardware using this query.

For small datasets, set `levels` to 1. For billions of vectors, you may need to increase it to 4 and tune `clusters` for your hardware.

The optimal number of clusters largely depends on server performance — how many vectors they can scan per second. So after choosing `levels` from expected data volume and network latency, tune `clusters` using performance tests.

For most hardware configurations, 20 to 50 clusters is recommended; this range is chosen for typical 256-dimensional vectors and balances performance and search recall.

Also aim for a configuration where the leaf clusters of the vector index contain at most 512 vectors, so that scanning them stays within milliseconds.

For example, if you expect 10 million vectors, set `levels` to 3 and `clusters` to 50. The vector index will look like this:

- Level 1: 50 clusters of 200,000 vectors each;
- Level 2: each level-1 cluster split into 50 clusters of 4,000 vectors each;
- Level 3: each level-2 cluster split into 50 clusters of 80 vectors each.

Search recall tuning

If a cluster’s centroid is closest to the query, the vectors in that cluster are usually closer to the query than vectors in other clusters. For complex data topologies this may not hold, so YDB allows scanning vectors in more than one cluster per level.

The number of clusters participating in search is set by the `KMeansTreeSearchTopSize` parameter. By default it is 1: YDB picks one nearest cluster at the first level, then requests its child clusters at the second level over the network, picks one nearest again, and so on. This parameter is not set at index creation time but for the search.

If you set a larger value, e.g. 3, the system will select three nearest clusters at each level instead of one. Moving to the next level then requires fetching the child vectors of all three selected clusters over the network, from which the next three are chosen. This improves recall but increases network traffic and the number of vectors scanned.

Covering index

By default, the leaf cluster stores not the vectors themselves but only a list of keys of those vectors in the table. Such a list is called a Posting Table — a form of inverted index. Storing references to vectors instead of vectors minimizes the size of the vector index.

However, after finding the leaf cluster, YDB must run a `SELECT` against the table to fetch the vectors (the exact number depends on the size of the leaf cluster).

Fetching vectors from the leaf cluster can be slow if `clusters` is too large or the table index does not include the vector column. In such cases you can speed up search by increasing index size: if you make the index covering, vectors are stored in the index. Data is still read over the network from distributed storage, but that is faster than a full query to the indexed table.

A covering index can include not only vectors but any other columns from the indexed table. Then `SELECT`s on those columns will be faster at the cost of storing an extra copy of their data in the index.

A covering index can be made faster by enabling read from replicas.

Vector index with filtering

Often you need to filter by additional attributes or metadata. A vector index with filtering speeds up queries with WHERE conditions by returning similar objects only among those that match the filter. With such an index, nearest-neighbor search is restricted to rows that satisfy the filter, which keeps performance high on large datasets.

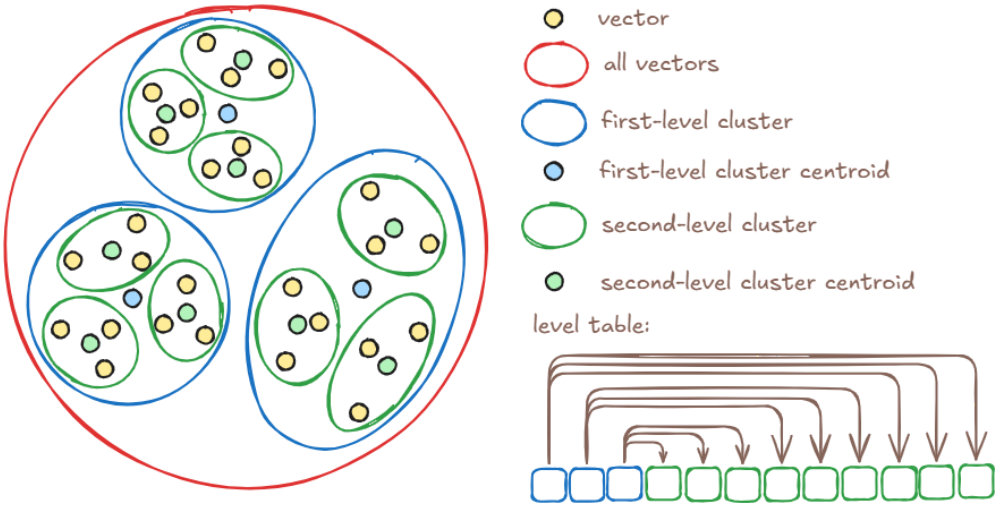
Fast, accurate filtered search is supported at the index structure level. The filtered column must be explicitly listed in the SQL index creation statement.

Filtering is built into the vector index because:

- If you filter before search, you get a set of rows in which to find nearest vectors. A normal vector index can only search over all vectors in the table, not an arbitrary subset. To find nearest vectors, the DB would have to scan all remaining rows and compute distance to the query for each.
- If you filter after search, you may get fewer results than needed and recall drops. You would have to repeat the search with a higher limit, which is also slow. In YDB, filtering is integrated into the vector index structure and algorithm, so rows that do not match the filter are excluded from consideration immediately.

Index internals

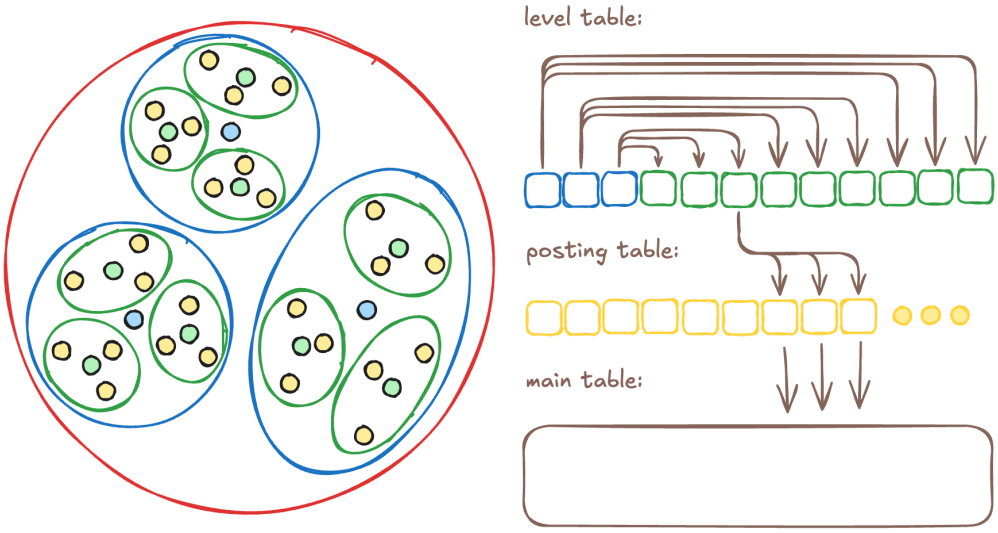
The index is implemented with two extra tables: the level table (`indexImplLevelTable`) and the posting table (`indexImplPostingTable`). The level table stores centroids for the cluster tree. YDB queries this table when moving from level to level during vector search. For example, if the vector index is built over 27 vectors with `levels=2` and `clusters=3`, the level table looks like this:



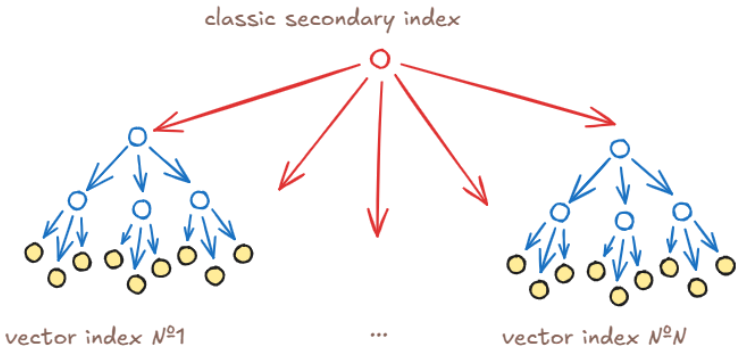
For each centroid in the level table, the parent cluster at the level above is also stored. YDB uses this to request only the centroids that belong to clusters selected at the previous level when moving down.

To avoid re-reading centroids from distributed storage on every query, the level table can be cached in memory on each node — see `resource_manager.kqp_level_cache_max_size_bytes` in `table_service_config`.

The posting table links the leaf-level centroids in the level table to the vectors in the source table. If a covering index is used, vectors and extra columns specified at index creation are also stored in this table:



With a filtered index, a regular secondary index on the filtered columns is added in front of the vector index. A separate vector index is built for each distinct value in the secondary index. The secondary index is stored in an extra relational table, the prefix table (`indexImplPrefixTable`), and all vector index trees remain in the level table:



Structure of table `indexImplLevelTable`

The primary key of this table includes the columns `__ydb_parent` and `__ydb_id`.

Column	Data type	Description
`__ydb_parent`	`Uint64`	Parent cluster identifier
`__ydb_id`	`Uint64`	Current cluster identifier
`__ydb_centroid`	`String`	Centroid vector of the current cluster

Structure of table `indexImplPostingTable`

The primary key of this table includes the column `__ydb_parent` and the columns that form the primary key of the indexed table (`id` in the example below).

Column	Data type	Description
`__ydb_parent`	`Uint64`	Identifier of the cluster containing the current vector
`id`	`Uint64`	Primary key of the indexed table that holds the vectors. Column names and types are copied from the indexed table.

Example structure of `indexImplPostingTable` with a covering index

With a covering index, the specified columns from the indexed table are copied into `indexImplPostingTable`. The primary key is the same as for `indexImplPostingTable` without covering. If the covering index includes the `embedding` column, the table looks like this:

Column	Data type	Description
`__ydb_parent`	`Uint64`	Identifier of the cluster containing the current vector
`id`	`Uint64`	Primary key of the indexed table that holds the vectors. Column names and types are copied from the indexed table.
`embedding`	`String`	Copy of the vector from the indexed table

Example structure of `indexImplPrefixTable` for a filtered index

With a filtered index, column names and types are copied from the indexed table into `indexImplPrefixTable`. If the index is filtered by the `category_id` column, the table looks like this:

Column	Data type	Description
`category_id`	`Uint64`	Column names and types are copied from the indexed table
`__ydb_id`	`Uint64`	Root cluster identifier of the vector index

Fulltext Indexes

Fulltext indexes are a specialized type of secondary index that enable efficient text search within table columns. While traditional secondary indexes optimize searching by equality or range, fulltext indexes allow searching by words, phrases, and (with n-grams) by substrings.

For the general idea of fulltext search, see Fulltext search.

Characteristics of Fulltext Indexes

Fulltext indexes in YDB are built by tokenizing text and creating an inverted index. This enables:

- fast filtering with FulltextMatch
- relevance ranking (BM25) with FulltextScore when using fulltext_relevance
- case normalization, stemming, and n-gram matching via index filters

The current implementation supports two indexes:

- fulltext_plain — basic fulltext index
- fulltext_relevance — fulltext index with BM25 statistics for relevance scoring

Additionally, a fulltext index can be **covering** (via `COVER`), meaning it includes a copy of extra columns from the base table.

Types of Fulltext Indexes

YDB supports two types of fulltext indexes, differing in the statistics they store:

- fulltext_plain — stores only the inverted index. Supports filtering via FulltextMatch, but does not support relevance ranking.
- fulltext_relevance — additionally stores term frequency statistics (TF-IDF / BM25) required by FulltextScore.

Indexes of either type can also be filtered to scope search to a specific logical partition.

Basic fulltext index (`fulltext_plain`)

Use `fulltext_plain` when you only need to check whether terms are present in the text, without relevance ranking. This index is more compact than `fulltext_relevance` and is suitable for most filtering tasks.

Example: create a global fulltext index on the `body` column:

```
ALTER TABLE articles
  ADD INDEX ft_index
  GLOBAL USING fulltext_plain
  ON (body)
  WITH (tokenizer=standard, use_filter_lowercase=true);
```

Here `tokenizer=standard` splits text into words on whitespace and punctuation, and `use_filter_lowercase=true` normalizes all tokens to lowercase, making the search case-insensitive.

Example query:

```
SELECT id, title
FROM articles VIEW ft_index
WHERE FulltextMatch(body, "search terms")
LIMIT 20;
```

Fulltext index for ranking (`fulltext_relevance`)

`fulltext_relevance` stores the inverted index along with term frequency statistics (BM25), which allows FulltextScore to compute a relevance score for each document. Use this type when you need not just to find documents containing certain words, but also to rank them by how well they match the query.

Example index:

```
ALTER TABLE articles
  ADD INDEX ft_index
  GLOBAL USING fulltext_relevance
  ON (body)
  WITH (tokenizer=standard, use_filter_lowercase=true);
```

Example ranking query:

```
SELECT id, title, FulltextScore(body, "search terms") AS relevance
FROM articles VIEW ft_index
WHERE FulltextScore(body, "search terms") > 0
ORDER BY relevance DESC
LIMIT 10;
```

Substring search (n-grams)

If you need substring search, create the index with n-grams. Two types of n-grams are available:

- **Regular n-grams** (`use_filter_ngram`) — split words into all possible substrings of specified length, allowing matches anywhere within a word. For example, the word "search" will be split into "sea", "ear", "arc", "rch", etc.
- **Edge n-grams** (`use_filter_edge_ngram`) — create substrings only from the beginning of words, which is ideal for autocomplete functionality. For example, the word "search" will be split into "se", "sea", "sear", "searc", "search".

When using n-grams, the following becomes available:

- FulltextMatch(..., "Wildcard" AS Mode) — patterns with `%` and `_` (similar to `LIKE`)

- `LIKE` / `ILIKE` predicates over the indexed text column — YDB automatically uses the n-gram index when accessed via `VIEW IndexName`

Example index with n-grams:

```
ALTER TABLE articles
  ADD INDEX ngram_index
  GLOBAL USING fulltext_plain
  ON (body)
  WITH (
    tokenizer=standard,
    use_filter_lowercase=true,
    use_filter_ngram=true,
    filter_ngram_min_length=3,
    filter_ngram_max_length=5
  );
```

Example query with `FulltextMatch`:

```
SELECT id, title
FROM articles VIEW ngram_index
WHERE FulltextMatch(body, "%learn%", "Wildcard" AS Mode)
LIMIT 20;
```

Example query with `LIKE`:

```
SELECT id, title
FROM articles VIEW ngram_index
WHERE body LIKE "%learn%ing%"
LIMIT 20;
```

A `LIKE` / `ILIKE` query uses the same logic as `FulltextMatch(body, ..., "Wildcard" AS Mode)` and accesses the same n-gram index.

Filtered fulltext index

A filtered fulltext index enables fulltext search within each logical partition defined by filter columns. To create such an index, specify one or more filter columns before the text column in the `ON` clause. The last column must be the text column; the others can be of any comparable type:

```
ALTER TABLE articles
  ADD INDEX ft_index
  GLOBAL USING fulltext_plain
  ON (user_id, body)
  WITH (tokenizer=standard, use_filter_lowercase=true);
```

Search queries using a filtered index must include an equality predicate on every filter column:

```
SELECT id, title
FROM articles VIEW ft_index
WHERE user_id = 42 AND FulltextMatch(body, "search terms")
LIMIT 20;
```

Multiple filter columns are supported. The equality predicates may appear in any order in `WHERE`; YDB reorders them internally to match the index column order.

Primary key types

Inside the inverted index, every indexed document is identified by a numeric document id (`doc_id`). How YDB derives the `doc_id` depends on the base table's primary key:

- **Single integer primary key** (`Uint64`, `Int64`, `Uint32`, or `Int32`) — the primary key value is used directly as the `doc_id`. This is the most compact form, and no additional structures are created.
- **Any other primary key** (for example, `Utf8`, `String`, other non-integer types, or a composite key of several columns) — YDB maintains a system column `__ydb_row_id` of type `Uint64` and uses it as the `doc_id`.

When you create a fulltext index on a table whose primary key is not a single integer column, YDB automatically:

- adds the `__ydb_row_id` column to the table — existing rows are backfilled while the index is being built;
- generates a `__ydb_row_id` value for every row where the column is omitted from `INSERT` / `UPSERT`;
- creates a unique secondary index named `__ydb_unique_row_id` over `__ydb_row_id`. At query time this index maps a matched `__ydb_row_id` back to the table's primary key before the row is read from the main table.

If the table already has more than one fulltext index, they all **reuse** the same `__ydb_row_id` column and `__ydb_unique_row_id` index — these structures are created only once per table.

Warning

The `__ydb_row_id` column and the `__ydb_unique_row_id` index are managed by YDB:

- Let YDB populate `__ydb_row_id` — omit it from `INSERT` / `UPSERT` and it is filled in automatically. User modification of the `__ydb_row_id` column is forbidden — any attempts to set or change the value of this column will be rejected.
- The `__ydb_unique_row_id` index cannot be dropped while any fulltext index depends on it. Drop the dependent fulltext index(es) first.

Full syntax for fulltext indexes

Creating a fulltext index:

- during table creation: CREATE TABLE
- adding to an existing table: ALTER TABLE

Full syntax for querying a fulltext index:

- VIEW (Fulltext index)

Functions and expressions for fulltext search:

- Fulltext built-in functions
- LIKE / ILIKE with a fulltext index

Note

The optimizer doesn't select a fulltext index automatically, so you must specify it explicitly using `VIEW IndexName`.

If the `VIEW` expression is not used, `FulltextMatch` / `FulltextScore` queries will fail.

This limitation may be removed in future versions of YDB.

Updating Fulltext Indexes

Fulltext indexes are maintained automatically on data modifications. Tables with fulltext indexes support:

- `INSERT`
- `UPSERT`
- `REPLACE`
- `UPDATE`
- `DELETE`

Dropping Fulltext Indexes

```
ALTER TABLE articles DROP INDEX ft_index;
```

Limitations

- Tables with a non-integer or composite primary key get an auto-managed `__ydb_row_id` column and `__ydb_unique_row_id` unique index (see Primary key types).
- `BulkUpsert` isn't supported for tables with fulltext indexes.
- Fulltext index access must be specified explicitly using `VIEW IndexName`.
- Only one text column can be indexed (per fulltext index). Use `COVER` for additional columns.
- `FulltextMatch` / `FulltextScore` can't be used with `OR` or `NOT`. Combining them with other predicates via `AND` is supported.
- A single read through `VIEW` supports only one fulltext predicate: multiple `FulltextScore` calls are not supported, and mixing `FulltextMatch` and `FulltextScore` in the same `WHERE` is not supported.
- For relevance access, you must include `FulltextScore(...) > 0` in `WHERE` (otherwise the query fails).
- Filtered fulltext indexes: every filter column needs an equality predicate in `WHERE`.
- Filtered fulltext indexes: filter columns must not be primary key columns.

Hybrid search

Hybrid search combines fulltext search and vector search into a single ranked result: each document is scored by both its text relevance and its embedding similarity, and the two rankings are fused into one. This brings together the precision of lexical matching and the recall of semantic similarity, and is a common building block for the retrieval stage of Retrieval-Augmented Generation (RAG).

For the general idea of hybrid search, see Hybrid search concept.

Hybrid search is not a separate index type. It reuses two existing indexes on the same table:

- a fulltext_relevance index over the text column — provides the BM25 relevance signal;
- a vector_kmeans_tree index over the embedding column — provides the nearest-neighbor (semantic) signal.

Preparing the indexes

Create a table with a text column and an embedding column, then add both indexes.

```
CREATE TABLE documents (
    id Uint64,
    text Utf8,
    embedding String,
    PRIMARY KEY (id)
);
```

Add a `fulltext_relevance` index over the text column (relevance scoring requires this type, not `fulltext_plain`):

```
ALTER TABLE documents
  ADD INDEX ft_idx
  GLOBAL USING fulltext_relevance
  ON (text)
  WITH (tokenizer=standard, use_filter_lowercase=true);
```

Add a `vector_kmeans_tree` index over the embedding column:

```
ALTER TABLE documents
  ADD INDEX vec_idx
  GLOBAL USING vector_kmeans_tree
  ON (embedding)
  WITH (distance=cosine);
```

For details on each index type, see Fulltext Indexes and Vector Indexes.

Running a hybrid query

A hybrid query is a regular `SELECT` over the base table (without `VIEW`) whose `ORDER BY` key is a single `HybridRank` call. `HybridRank` takes one scoring expression per branch: a FullTextScore for the text branch and a Knn distance or similarity for the vector branch.

```
PRAGMA ydb.KMeansTreeSearchTopSize = "10";

$queryText = "machine learning";
$queryVector = Knn::ToBinaryStringFloat([0.1, 0.2, 0.3, 0.4]);

SELECT id, text
FROM documents
ORDER BY HybridRank(
    FullTextScore(text, $queryText),
    Knn::CosineDistance(embedding, $queryVector))
LIMIT 10;
```

Both inputs come from the same user query: `$queryText` is the search text (matched lexically), and `$queryVector` is its embedding computed by the application (matched semantically). YDB cannot compute embeddings itself, so the vector is passed in — here it is built from a literal via Knn::ToBinaryStringFloat, but in an application it is the output of an embedding model.

YDB resolves each branch to its index automatically by the scored column (`text` → the fulltext relevance index, `embedding` → the vector index), retrieves a candidate pool from each, and fuses the two rankings. The result is the top `LIMIT` documents by the fused score.

Note that, unlike fulltext and vector search, a hybrid query does **not** use `VIEW IndexName`: the indexes are selected from the `HybridRank` arguments, and the query reads through the base table.

`PRAGMA ydb.KMeansTreeSearchTopSize` controls the recall of the vector branch — see KMeansTreeSearchTopSize. As with plain vector search, it should be set explicitly.

Tuning the fusion

The fusion method and its parameters are passed as named arguments to `HybridRank`. The most common ones:

- `Mode` — `"rrf"` (default, Reciprocal Rank Fusion) or `"linear"` (weighted sum of normalized scores);
- `Weights` — a per-branch weight tuple, one value per scoring argument, to bias the ranking toward one signal;
- `K` — the RRF constant (default `60.0`);
- `Indexes` / `Limits` — explicit per-branch index names and candidate-pool sizes.

For the built-in modes, the per-branch contribution is fixed by the formula. For full control over the fusion, pass a custom lambda instead of `Mode`:

- `RankLambda` — a lambda that receives each document's per-branch ranks (1-based position within each branch) and returns the fused score;
- `ScoreLambda` — a lambda that receives each document's per-branch raw scores (the fulltext relevance or vector distance/similarity) and returns the fused score.

A custom lambda replaces the built-in fusion, so it cannot be combined with `Mode`, `Weights`, `K`, or `Normalize` — fold any weights and constants into the lambda body. For a detailed description and examples, see Hybrid search query syntax (HybridRank).

For example, to weight the vector branch twice as much as the text branch under RRF:

```
$queryText = "machine learning";
$queryVector = Knn::ToBinaryStringFloat([0.1, 0.2, 0.3, 0.4]);

SELECT id, text
FROM documents
ORDER BY HybridRank(
    FullTextScore(text, $queryText),
    Knn::CosineDistance(embedding, $queryVector),
    (1, 2) AS Weights)
LIMIT 10;
```

For the full list of parameters and their semantics, see Hybrid search (HybridRank).

Limitations

- The table must have both a ready `fulltext_relevance` index and a non-prefixed `vector_kmeans_tree` index over the respective columns; otherwise the query fails with a clear message.
- Prefixed vector indexes are not supported yet.
- If more than one fulltext (or vector) index matches a branch's column, the branch is ambiguous and must be disambiguated with an explicit `AS Indexes` override.
- `LIMIT` must be a literal, because it sizes the per-branch candidate pools. To use a parameterized `LIMIT`, pass explicit `AS Limits`.
- `HybridRank(...)` must be the entire `ORDER BY` key — it cannot be negated, wrapped, or combined with other sort keys.
- A custom fusion lambda (`RankLambda` or `ScoreLambda`) replaces the built-in fusion and cannot be combined with `Mode`, `Weights`, `K`, or `Normalize`. At most one of `RankLambda` or `ScoreLambda` may be specified.

JSON indexes

JSON indexes are a type of secondary index implemented on top of an inverted index that speeds up filtering table rows by conditions imposed on the contents of columns of type `Json` and `JsonDocument`. The index is used if the `WHERE` predicate uses the functions JSON_EXISTS and JSON_VALUE with JsonPath expressions. Unlike traditional secondary indexes optimized for equality or range searches on individual table columns, a JSON index works with arbitrary paths within a JSON document.

For a general description of JSON search and the structure of an inverted index on JSON document paths, see the Searching JSON document contents section.

Characteristics of JSON indexes

JSON indexes in YDB allow:

- Quickly filter rows by JSON_EXISTS and JSON_VALUE with JsonPath expressions
- combine indexed conditions with `AND` and `OR` operators
- use query parameter values passed by the application when processing checked predicates.

A JSON index is a global synchronous index — its data is always consistent with the base table.

When executing a query, a JSON index can be applied:

- explicitly — via the `<table_name> VIEW <index_name>` operator
- automatically by the optimizer, if the predicate matches the formal rules.

Syntax of JSON indexes

Creating a JSON index:

- when creating a table: INDEX (CREATE TABLE)
- Adding to an existing table: ALTER TABLE.

Deleting JSON indexes is done via ALTER TABLE:

```
ALTER TABLE documents DROP INDEX json_idx
```

Query syntax with explicit JSON index specification:

- VIEW (JSON index).

Functions and expressions for working with JSON in predicates:

- JSON functions — `JSON_EXISTS` and `JSON_VALUE`
- JsonPath — a query language for accessing values inside JSON.

Ready-made use cases are collected in the JSON document search recipes section.

Updating JSON indexes

JSON indexes are automatically maintained when data is modified and are updated synchronously together with the main table. Tables with JSON indexes support:

- `INSERT`
- `UPSERT`
- `REPLACE`
- `UPDATE`
- `DELETE`

Batch operations (`BATCH UPDATE` and `BATCH DELETE`) are not supported for tables with JSON indexes. When attempting to execute such a query on a table for which a JSON index has been created, the query will be rejected, a corresponding error will be returned to the application, and the data will remain unchanged.

Additionally, tables with JSON indexes do not support:

- Bulk data loading via a `BulkUpsert` call — the requested operation will be rejected with a corresponding error message.
- automatic deletion of rows by TTL — errors are returned when attempting to create a table with both a TTL policy and a JSON index, as well as when trying to retrieve such a combination of properties using the `ALTER TABLE` commands.

Supported predicates

For execution via JSON indexes, only expressions based on the `JSON_EXISTS` and `JSON_VALUE` functions in the `WHERE` block, combined by the `AND` / `OR` operators according to the rules below, are supported.

JSON_EXISTS

Checking the existence of a path or value inside a JsonPath filter.

Allowed:

```
-- Document root (value not NULL)
WHERE JSON_EXISTS(doc, '$')

-- Key chain; array indexes are 'transparent'
WHERE JSON_EXISTS(doc, '$.user.name')
WHERE JSON_EXISTS(doc, '$.items[*].sku')
WHERE JSON_EXISTS(doc, '$.items[0 to last].active')

-- Filter ? (...) — predicates inside the filter are allowed
WHERE JSON_EXISTS(doc, '$.items ? (@.price == 100)')
WHERE JSON_EXISTS(doc, '$.items ? (@.qty >= 1 && @.qty <= 10)')
WHERE JSON_EXISTS(doc, '$.items ? (@.tag == $t)' PASSING "sale" AS t)
```

```
-- JsonPath methods (path is indexed up to the method; exact check is performed by post-filter)
WHERE JSON_EXISTS(doc, '$.value.type()')
WHERE JSON_EXISTS(doc, '$.arr.size()')

-- Combinations on one column
WHERE JSON_EXISTS(doc, '$.a') AND JSON_EXISTS(doc, '$.b')
WHERE JSON_EXISTS(doc, '$.a') OR JSON_EXISTS(doc, '$.b')
```

Prohibited (error when using the `VIEW` statement or index auto-selection failure):

```
-- Comparison predicates at the top level of the path (outside ? (...))
WHERE JSON_EXISTS(doc, '$.key == 10')
WHERE JSON_EXISTS(doc, 'exists($.key)')
WHERE JSON_EXISTS(doc, '$.key starts with "a"')

-- Negation in JsonPath
WHERE JSON_EXISTS(doc, '!($.key == 10)')

-- ON ERROR TRUE
WHERE JSON_EXISTS(doc, '$.key' TRUE ON ERROR)

-- Path without context operator ($) — the passed document is not used
WHERE JSON_EXISTS(doc, '1')
```

Note

The `JSON_EXISTS` function returns `true` for any non-empty JsonPath result. The `$.key == 10` predicate specified at the top level would give "path existence" even when the comparison is false, which does not match the expected semantics. Comparisons should be moved into `JSON_VALUE` calls or into a filter of the form `? (...)`.

JSON_VALUE

Extracting a scalar value with a required `RETURNING <type>`.

To compare a value using `JSON_VALUE`, you must always specify `RETURNING` with the required type. By default, `JSON_VALUE` returns type `Utf8`, which leads to incorrect comparison during query execution — values of different types are compared as strings:

```
$tmp = Json(@@["1", 1]@@);
SELECT JSON_VALUE($tmp, '$[0]') == "1"; -- true: correct, string compared with string
SELECT JSON_VALUE($tmp, '$[1]') == "1"; -- true: incorrect, number compared with string
```

Supported types for the `RETURNING` section: `Int8` … `Int64`, `Uint8` … `Uint64`, `Float`, `Double`, `Bytes` (`String`), `Text` (`Utf8`), `Bool`.

Examples of predicates applied via a JSON index:

```
-- Equality (path + value are included in the index)
WHERE JSON_VALUE(doc, '$.user.age' RETURNING Int32) = 25
WHERE JSON_VALUE(doc, '$.flag' RETURNING Bool) = true
WHERE JSON_VALUE(doc, '$.name' RETURNING Utf8) = "Alice"u

-- Implicit comparison with true for Bool
WHERE JSON_VALUE(doc, '$.active' RETURNING Bool)

-- Parameters
WHERE JSON_VALUE(doc, '$.user.id' RETURNING Int64) = $id
WHERE JSON_VALUE(doc, '$.tag' RETURNING Utf8) = $tag

-- Comparisons (only the path is used in the index, comparison is performed by post-filter)
WHERE JSON_VALUE(doc, '$.score' RETURNING Int64) > 0
WHERE JSON_VALUE(doc, '$.score' RETURNING Int64) != 100
WHERE JSON_VALUE(doc, '$.score' RETURNING Int64) BETWEEN 1 AND 10
WHERE JSON_VALUE(doc, '$.score' RETURNING Int64) NOT BETWEEN 0 AND 5

-- IN: list of literals
WHERE JSON_VALUE(doc, '$.status' RETURNING Utf8) IN ("open"u, "pending"u)

-- IN: specified parameter of type List<Utf8>
WHERE JSON_VALUE(doc, '$.status' RETURNING Utf8) IN $status_list

-- PASSING for JsonPath variables
WHERE JSON_VALUE(doc, '$.x ? (@.y == $v)' RETURNING Int64 PASSING 42 AS v) = 10

-- JsonPath predicates inside the path.
-- Unlike JSON_EXISTS, predicates at the top level are allowed.
WHERE JSON_VALUE(doc, '$.user ? (@.role == "admin")' RETURNING Utf8) = "ok"u
WHERE JSON_VALUE(doc, '$.code starts with "A"' RETURNING String) != ""
WHERE JSON_VALUE(doc, 'exists($.meta)' RETURNING Bool)

-- AND / OR combinations on one column
WHERE JSON_VALUE(doc, '$.a' RETURNING Int32) = 1
   OR JSON_VALUE(doc, '$.b' RETURNING Int32) = 2
WHERE JSON_EXISTS(doc, '$.a') AND JSON_VALUE(doc, '$.a' RETURNING Int32) = 10
```

Examples of predicates that cannot be applied via a JSON index:

```
-- JSON_VALUE call without RETURNING
WHERE JSON_VALUE(doc, '$.key') = "x"

-- DEFAULT with ON EMPTY / ON ERROR (except NULL)
WHERE JSON_VALUE(doc, '$.k' RETURNING Utf8 DEFAULT "x" ON ERROR) = "y"

-- Unsupported data type in RETURNING
WHERE JSON_VALUE(doc, '$.ts' RETURNING Timestamp) = ...

-- RETURNING Bool with comparison operators
WHERE JSON_VALUE(doc, '$.flag' RETURNING Bool) >= true

-- IS NULL / IS NOT NULL — semantically contradict the 'path existence' index
WHERE JSON_VALUE(doc, '$.k' RETURNING Utf8) IS NULL

-- Comparison of two JSON_VALUE from different columns
WHERE JSON_VALUE(doc1, '$.k' RETURNING Utf8) = JSON_VALUE(doc2, '$.k' RETURNING Utf8)

-- Nested JSON_* in arguments
WHERE JSON_VALUE(JSON_QUERY(doc, '$.a'), '$.b' RETURNING Utf8) = "x"
```

Note

To check 'value equals `false`' or 'value equals `null`', use a JsonPath filter inside `JSON_EXISTS`, for example `JSON_EXISTS(doc, '$.k ? (@ == false)')` or `JSON_EXISTS(doc, '$.k ? (@ == null)')`, not `JSON_VALUE(...) IS NULL`.

Limitations

- JSON indexes are supported only for row tables.
- The table's primary key must consist of a single column of an integer type (`Uint64`, `Uint32`, `Int64`, or `Int32`). This is a temporary limitation that will be removed in future development.
- A single JSON index indexes exactly one column of type `Json` or `JsonDocument`.
- The `COVER` expression is not supported for JSON indexes.
- A number of data modification operations and mechanisms are not supported for tables with JSON indexes.
- The parameter type of a read query from the index cannot be wrapped in `Optional<T>` — optional parameters are not supported.
- Equality comparison with an integer literal whose absolute value exceeds 2^{53} is not accelerated by the value index (such numbers do not fit into the numeric type used in `Json` and `JsonDocument`) and is reduced to a path existence check.
- Casting floating-point literals (`Float`, `Double`) to integer types during comparison is not performed — such comparison is not accelerated by the index.

Recipes

Ready-made scenarios for working with a JSON index:

- JSON index – quick start — quick start.
- Catalog with nested attributes — product catalog with nested attributes.
- Parameterized queries and JsonPath variables — parameterized queries and JsonPath variables.
- Checking field type and path existence — field type check and path existence.

Related materials

- JSON functions — `JSON_EXISTS`, `JSON_VALUE`, `JSON_QUERY`, JsonPath syntax.
- Secondary indexes — general information about global indexes and `VIEW`.
- Full-text indexes — a related mechanism built on top of an inverted index of words and phrases.
- INDEX (CREATE TABLE) and VIEW (JSON index) — syntax reference.

Local indexes

Local indexes are auxiliary structures stored together with table data and applied while reading from storage. Unlike global indexes, they do not materialize a separate index table and are not selected in queries with `VIEW`.

This section contains practical materials on local index types:

- Bloom skip indexes — Bloom indexes for accelerating selective queries by skipping data fragments.
- min_max index — min_max index that skips data fragments based on saved value ranges.

Bloom skip indexes

Bloom skip indexes are local auxiliary structures based on a Bloom filter that speed up selective queries by skipping data fragments that cannot contain the requested value. Unlike global secondary indexes, they act as read-time filters on the main table and reduce the amount of data that must be actually read.

Types

YDB supports two types:

- `bloom_filter`: a filter over exact values of the indexed column; use for equality and `IN` (see when to use which).
- `bloom_ngram_filter`: a filter over n-grams of a string column (`String`, `Utf8`); use for substring and `LIKE` pattern search on column-oriented tables.

Local bloom skip indexes

The `bloom_filter` type is supported on both row-oriented (OLTP) and column-oriented (OLAP) tables, but the underlying implementation differs:

- On row-oriented tables, the filter is built as a prefix bloom filter over a left prefix of the primary key. The indexed columns must be a contiguous leading subset of the primary key columns. This accelerates point lookups and range scans that constrain the leading key columns. Use ALTER TABLE ... ADD INDEX to create a prefix bloom filter and ALTER TABLE ... DROP INDEX to remove it.
- On column-oriented tables, the filter is built over the indexed column values in each data fragment (portion). It is applied during analytical scans to skip fragments that cannot contain the requested value.

`bloom_ngram_filter` is supported only on column-oriented tables.

Parameters and defaults

Full list of `WITH (...)` parameters and defaults:

- `bloom_filter`
  - `false_positive_probability`: Target false-positive rate of the filter: the fraction of fragments that are not skipped even though the requested value is not there (range `(0, 1)`). A lower value reduces extra reads but increases index size.
    - If omitted, the default is `0.0001` for row-oriented tables and `0.1` for column-oriented tables. The stricter default on row-oriented tables matches OLTP point lookups; on column-oriented tables it reflects analytical scans, where a larger trade-off between index size and fragment skipping is acceptable.
  - Indexed columns: YQL types that support equality comparison, except `Yson`, `Json`, and `JsonDocument` (see comparison operators). One column in a column-oriented table; multiple columns in a row-oriented table. On row-oriented tables, the indexed columns must form a left prefix of the primary key.
- `bloom_ngram_filter` (`String` and `Utf8` columns; column-oriented tables only)
  - `ngram_size`: N-gram length, an integer from `3` to `8` (default `3`).
  - `false_positive_probability`: Target false-positive rate (range `(0, 1)`; default `0.1`).
  - `case_sensitive`: Whether n-grams respect character case: `true` or `false` (default `true`).

Creation syntax: CREATE TABLE, ALTER TABLE ADD INDEX.

To change parameters after creation, use ALTER INDEX.

Examples

Create a table with a `bloom_filter` index:

```
CREATE TABLE events (
    id Uint64,
    resource_id Utf8,
    message Utf8,
    PRIMARY KEY (id),
    INDEX idx_bloom LOCAL USING bloom_filter
        ON (resource_id)
        WITH (false_positive_probability = 0.01)
);
```

Add a `bloom_ngram_filter` index to an existing table:

```
ALTER TABLE events
  ADD INDEX idx_ngram LOCAL USING bloom_ngram_filter
  ON (message)
  WITH (
    ngram_size = 3,
    false_positive_probability = 0.01,
    case_sensitive = true
  );
```

Create a row-oriented (OLTP) table with prefix bloom filters on primary-key prefixes. On row-oriented tables the indexed columns must be a left prefix of the primary key:

```
CREATE TABLE orders (
    customer_id Utf8,
    order_id Utf8,
    amount Decimal(10,2),
    PRIMARY KEY (customer_id, order_id),
    -- Prefix bloom filter on the first key column
    INDEX idx_customer LOCAL USING bloom_filter
        ON (customer_id)
        WITH (false_positive_probability = 0.001),
```

```
    -- Prefix bloom filter on the full primary key
    INDEX idx_full_key LOCAL USING bloom_filter
        ON (customer_id, order_id)
);
```

Alter index parameters:

```
ALTER TABLE events ALTER INDEX idx_ngram SET (
    ngram_size = 4,
    false_positive_probability = 0.005,
    case_sensitive = false
);
```

Parameter tuning

If queries still read too much data after deploying the index, try lowering `false_positive_probability`. If the index uses too much space, try raising it.

Use `ALTER INDEX` to change values, as in the example above.

Tips:

- Start with `false_positive_probability = 0.01`, then adjust based on real read metrics and index size.
- For `bloom_ngram_filter`, `ngram_size` typically starts at `3`; increasing it can make filtering stricter for longer substrings.
- Change one parameter at a time and compare results under the same workload.

Behavior and limitations

- The index is always local (`LOCAL`); there is no global variant.
- Queries do not use the `VIEW <index>` syntax (unlike, for example, fulltext indexes).
- The filter is applied on read only to data fragments where an index block was already stored during a write or portion merge, other fragments are not skipped by this index until merge.
- On row-oriented tables, `bloom_filter` is implemented as a **prefix bloom filter**: the indexed columns must form a left prefix of the primary key. The filter is built over that key prefix and accelerates point lookups and range scans that constrain the prefix. Multiple bloom indexes over the same prefix length are not allowed.

Limitations

- `COVER (...)` and extra index columns are not supported.
- Column-oriented tables allow only one indexed column. Row-oriented tables allow multiple indexed columns.
- `bloom_ngram_filter` is not supported on row-oriented tables.
- On row-oriented tables, the indexed columns of a `bloom_filter` must be a left prefix of the primary key. Non-prefix column sets are rejected.
- On row-oriented tables, two `bloom_filter` indexes cannot share the same prefix length (the same set of leading primary-key columns).

Additional Materials

- Secondary indexes
- YQL reference: CREATE TABLE
- YQL reference: SELECT
- YQL reference: ALTER TABLE
- Quickstart

min_max index

min_max index is a local index that speeds up scanning queries with a highly selective filter by skipping fragments. Unlike global secondary indexes, it acts as a read filter for the base table and reduces the amount of data that actually needs to be read.

For each indexed data fragment, the min_max index stores the minimum and maximum value of one column. During query execution, YDB evaluates the predicate on these two values. If the evaluation results show that the predicate will filter out all tuples of the fragment, the fragment is skipped.

Examples

Creation syntax: CREATE TABLE, ALTER TABLE ADD INDEX.

Creating a columnar table with a min_max index:

```
CREATE TABLE events (
    id Uint64,
    created_at Timestamp,
    level Int32,
    resource_id Utf8,
    PRIMARY KEY (id),
    INDEX idx_created_at LOCAL USING min_max
        ON (created_at),
    INDEX idx_level LOCAL USING min_max
        ON (level)
)
WITH (
    STORE = COLUMN
);
```

Adding a min_max index to an existing columnar table:

```
ALTER TABLE events
  ADD INDEX idx_resource_id LOCAL USING min_max
  ON (resource_id);
```

Queries with range predicates can use the index to skip non-matching fragments:

```
SELECT id, resource_id
FROM events
WHERE created_at BETWEEN Timestamp("2024-01-01T00:00:00.000000Z")
                     AND Timestamp("2024-01-02T00:00:00.000000Z");
```

When to use

The min_max index is useful when the values of the indexed column change little between rows that are adjacent in the primary key order: for example, timestamps, monotonically increasing identifiers, or other values correlated with the primary key.

The min_max index can also be useful when the filter predicate selects a very small fraction of the data (on the order of one row per million). For example, when querying a service log table and keeping only records with the `ERROR` level: if the service writes one error per 1,000,000 records, the min_max index will likely significantly reduce the read volume.

Features and limitations

- The index is always local (`LOCAL`); there is no global variant.
- Queries do not use the `VIEW <index>` syntax (unlike, for example, full-text indexes).
- The filter is applied during reads only to data fragments for which the minimum and maximum values of the indexed column have already been computed and stored with the table data at write time or merge. For other fragments, no skip by this index is performed.

Limitations

- Supported only for columnar tables.
- `ON (...)` must specify exactly one column.
- `COVER (...)` and additional data columns are not supported.
- Specific parameters of `WITH (...)` are not supported.
- `ALTER INDEX` is not supported for the min_max index.
- Columns of types `Json` and `JsonDocument` are not supported.

Additional materials

- Secondary indexes
- Local indexes
- YQL reference: CREATE TABLE
- YQL reference: SELECT
- YQL reference: ALTER TABLE
- Quick start

Query execution optimization

Materials on analyzing and tuning query execution:

- Using query plans for query optimization — logical and execution plans, finding bottlenecks.
- Optimizer hints — influencing the cost-based optimizer via the `PRAGMA ydb.OptimizerHints` pragma.
- Parameterized queries and recompilation — why identical SQL structure recompiles when values are embedded in the query text.

Query plan optimization

It's very useful to to analyze execution plans for queries in order to detect and eliminate the causes of possible inefficiencies. YDB provides two types of query plans: logical plan and execution plan. Logical plan is better suited for analyzing complex queries with a large number of JOIN operators. Execution plan is more detailed: it additionally shows the stages of the distributed plan and connectors between them, which makes it more convenient for analyzing simple OLTP queries.

Logical Query Plan

You can get the logical plan via YDB CLI.

This plan allows you to compare the query optimizer's predictions with the execution statistics. If the predictions differ significantly from the actual data at the execution stage, this may indicate that the optimizer has not built the most efficient plan for the current query. In this case, you can use optimizer hints to create a more efficient plan.

Query Execution Plan

To illustrate how to work with the execution plan, consider the following OLTP query that searches for a series by name:

```
SELECT season_id, episode_id
  FROM episodes
  WHERE title = 'The Work Outing'
```

Schema of the `episodes` table:

Id	Key	Name	Type	NotNull
1	🔑	series_id	Uint64	
2	🔑	season_id	Uint64	
3	🔑	episode_id	Uint64	
5		air_date	Uint64	
4		title	Utf8	

Let's build a query execution plan for this query. You can do this via either UI or YDB CLI:

YDB CLI

You can build a query plan via YDB CLI using the following command:

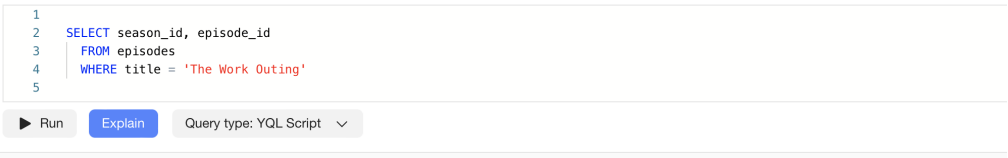
```
ydb -p <profile_name> table query explain \
  -q "SELECT season_id, episode_id
  FROM episodes
  WHERE title = 'The Work Outing'"
```

Result:

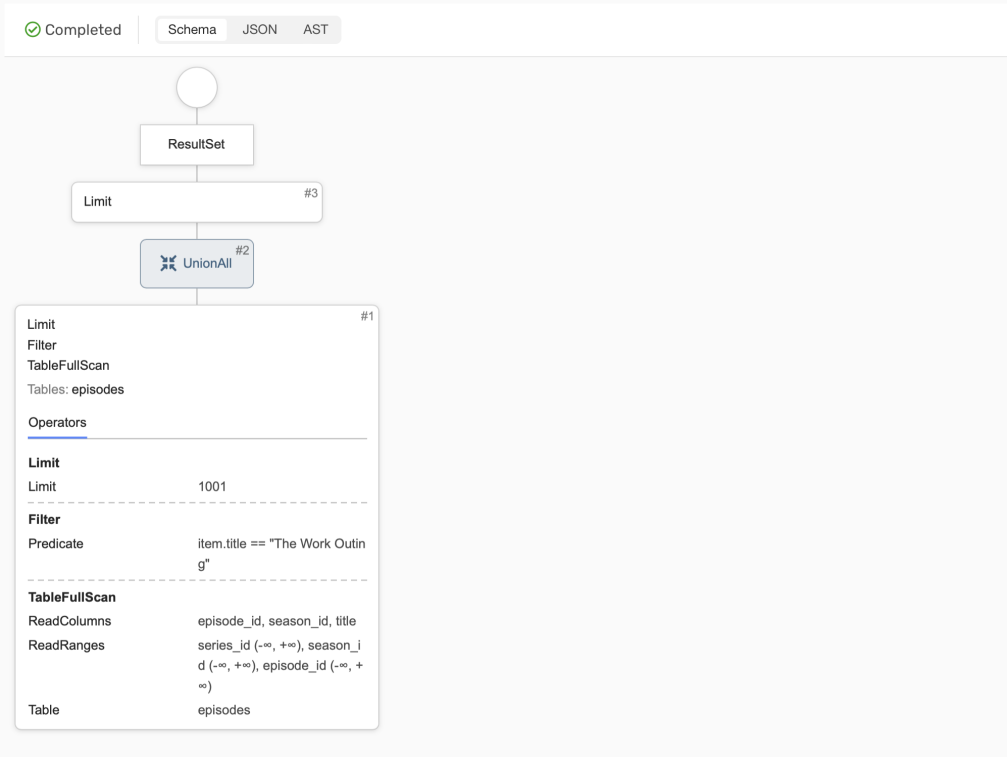
```
Query Plan:
ResultSet
└──Limit (Limit: 1001)
   └──<UnionAll>
      └──Limit (Limit: 1001)
      └──Filter (Predicate: item.title == "The Work Outing")
      └──TableFullScan (ReadRanges: ["series_id (-∞, +∞)","season_id (-∞, +∞)","episode_id (-∞, +∞)"], ReadCo
lumns: ["episode_id","season_id","title"], Table: episodes)
         Tables: ["episodes"]
```

YDB UI

You can also build a query plan via YDB UI. You need to navigate to the database page, go to the `Query` section, type the query text, and click on `Explain`:



Result:



Both plan representations contain the result being returned to the client at the root, table operations at the leaves, and data transformations at the intermediate nodes. It is important to pay attention to the node containing the table reading operation. In this case, it is a `TableFullScan` for the `episodes` table. Full table scans consume time and resources proportional to the size of the table, so it is advisable to avoid them whenever possible in tables that tend to grow over time or are simply large.

One typical approach to avoid full scans is using a secondary index. In this case, it makes sense to add a secondary index for the column `title` using the following query:

```
ALTER TABLE episodes
  ADD INDEX title_index GLOBAL ON (title)
```

Please note that this example uses synchronous secondary index. Building an index in YDB is an asynchronous operation. Even if the index creation query is successful, it is advisable to wait for some time because the index may not be ready for use immediately. You can manage asynchronous operations through the CLI.

Let's build the query plan using the secondary index `title_index`. Secondary indexes to be used need to be explicitly specified in the `VIEW` clause.

YDB CLI

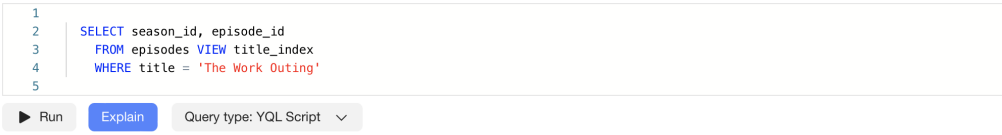
Command:

```
ydb -p <profile_name> table query explain \
  -q "SELECT season_id, episode_id
  FROM episodes VIEW title_index
  WHERE title = 'The Work Outing'"
```

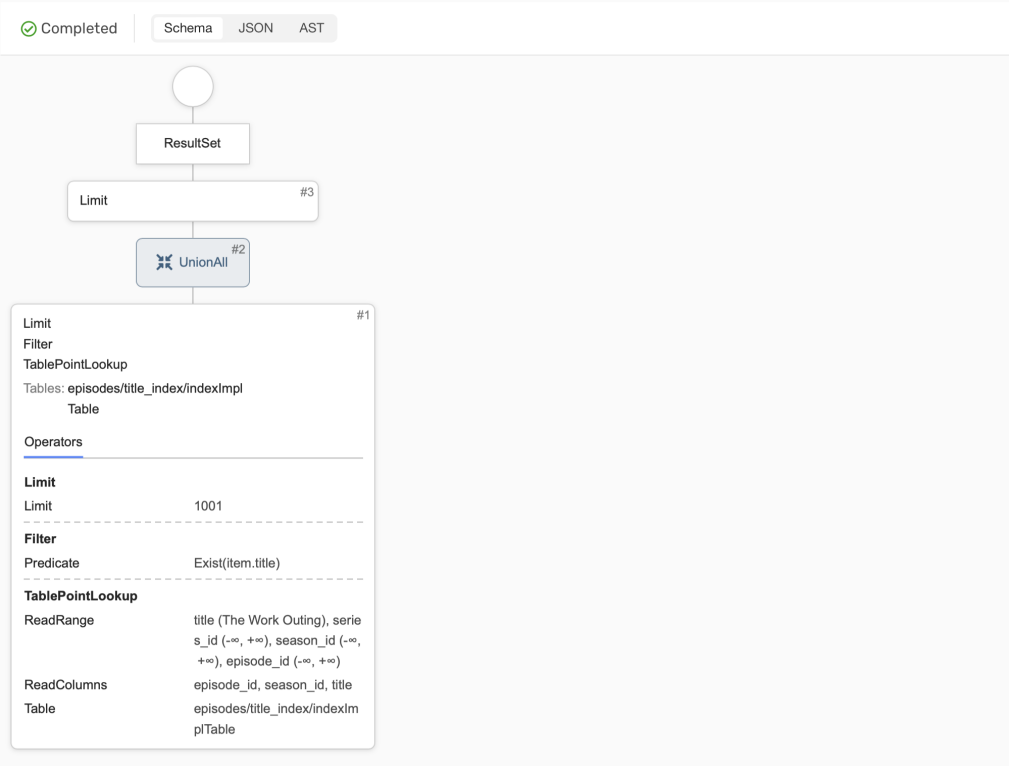
Result:

```
Query Plan:
ResultSet
└──Limit (Limit: 1001)
   └──<UnionAll>
      └──Limit (Limit: 1001)
      └──Filter (Predicate: Exist(item.title))
      └──TablePointLookup (ReadRange: ["title (The Work Outing)","series_id (-∞, +∞)","season_id (-∞, +∞)","episode_id (-∞, +∞)"], ReadLimit: 1001, ReadColumns: ["episode_id","season_id","title"], Table: episodes/title_index/indexImplTable)
         Tables: ["episodes/title_index/indexImplTable"]
```

YDB UI



Result:



The secondary index allowed the query to be executed without fully scanning the main table. Instead of a `TableFullScan,` we received a `TablePointLookup` —reading the index table by key. We no longer need to read the main table because all necessary columns are contained in the index table.

Optimizer Hints

Optimizer hints allow you to influence the behavior of the cost-based optimizer when planning the execution of SQL queries. YDB supports four types of hints for managing joins and statistics.

Usage

Hints are specified via the `PRAGMA ydb.OptimizerHints` pragma at the beginning of the SQL query.
If the optimizer was unable to apply at least one of the specified hints to the query, the user will be notified via a warning.

Syntax

Hints are specified as a string containing an array of expressions of one of four types:

```
JoinType(TableList JoinType)
Rows(TableList Op Value)
Bytes(TableList Op Value)
JoinOrder(JoinTree)

where:
TableList - list of table names or aliases from the query
Op - operation:
- `#` - set absolute value
- `*` - multiply by value
- `/` - divide by value
- `+` - add value
- `-` - subtract value
- `Number` - absolute numeric value
Value - numeric value
JoinTree - representation of binary tree using parentheses, for example: (R S) (T U)
```

For example, the following query uses 3 cardinality (number of records) hints `Rows`, a total join order hint `JoinOrder`, and a join algorithm selection hint `JoinType`:

```
PRAGMA ydb.OptimizerHints =
'
    Rows(R # 10e8)
    Rows(T # 1)
    Rows(R T # 1)
    JoinOrder( (R S) (T U) )
    JoinType(T U Broadcast)
';

SELECT * FROM
    R   INNER JOIN  S   on  R.id = S.id
        INNER JOIN  T   on  R.id = T.id
        INNER JOIN  U   on  T.id = U.id;
```

Requirements for CBO (Cost Based Optimizer)

> **Note**
>
> All hints (`Rows`, `Bytes`, `JoinOrder`) work only with **enabled** cost based optimizer, except `JoinType` - it can be specified even if CBO is disabled.

Hint types

JoinType - Join algorithm

Allows you to force the join algorithm for specific tables.

There are currently three types of join algorithms supported in YDB:

- BroadcastJoin - is a join type where one of the data sets is small enough to be copied (broadcasted) to all required nodes in the cluster. This allows each node to perform the join locally, without transmitting the data over the network.

> **Note**
>
> If the join order is not fixed by a separate hint, the optimizer will build both variants of the plan: where the left and right input of the join is broadcasted. If the join order is fixed by a hint, the right side of the join will be broadcasted.

- ShuffleJoin is a type of join in which the data is divided (shuffled) by the join key so that records with the same key are processed on the same processing node. After such a redistribution of data, each node performs a local join of the tables. The results are combined into one common data set.
- LookupJoin - for each row of one input, a query is made to the table or index of the other input, currently supported only for row tables.

> **Note**
>
> If the join order is not fixed by a separate hint, the optimizer will build both variants of the plan: where queries are made to the left and right inputs of the join. If the join order is fixed by a hint, queries will be made to the right side of the join.

Syntax

```
JoinType(t1 t2 ... tn Broadcast | Shuffle | Lookup)
```

Parameters

- `t1 t2 ... tn` - tables involved in the join
- Algorithm:
- `Broadcast` - select the BroadcastJoin algorithm
- `Shuffle` - select the ShuffleJoin algorithm
- `Lookup` - select the LookupJoin algorithm

How it works

If the query plan contains a join operator that joins only those tables listed in the hint, the optimizer will choose the specified join algorithm if it is applicable (for example, you cannot apply the LookupJoin algorithm to column-oriented tables). If the algorithm cannot be applied, the user will be notified via a warning.

Examples

```
-- Use Broadcast to join tables nation, region
JoinType(nation region Broadcast)

-- Use HashJoin for a join whose subtree will contain only tables customers, orders, products
JoinType(customers orders products Shuffle)

JoinType(nation region Lookup)
```

Apply join algorithm hints to the following query:

```
PRAGMA ydb.OptimizerHints =
'
JoinType(R S Shuffle)
JoinType(R S T Broadcast)
JoinType(R S T U Shuffle)
JoinType(R S T U V Broadcast)
';

SELECT * FROM
R INNER JOIN S on R.id = S.id
INNER JOIN T on R.id = T.id
INNER JOIN U on T.id = U.id
INNER JOIN V on U.id = V.id;
```

You can view the query plan using the CLI command:

```
 ydb -p <profile_name> sql --explain -f query.sql
```

```
┌──────────────────────────────────────────────────────────────────────────────────────────────┐
│ Operation                                                                                    │
├──────────────────────────────────────────────────────────────────────────────────────────────┤
│ ┌─> ResultSet                                                                                │
│ └─┬─> InnerJoin (MapJoin) (U.id = V.id)                                                      │
│   ├─┬─> InnerJoin (Grace) (T.id = U.id)                                                      │
│   │ ├─┬─> HashShuffle (KeyColumns: ["T.id"], HashFunc: "HashV2")                             │
│   │ │ └─┬─> InnerJoin (MapJoin) (R.id = T.id)                                                │
│   │ │   ├─┬─> InnerJoin (Grace) (R.id = S.id)                                              │
│   │ │   │ ├─┬─> HashShuffle (KeyColumns: ["id"], HashFunc: "HashV2")                       │
│   │ │   │ │ └──> TableFullScan (Table: R, ReadColumns: ["id (-∞, +∞)","payload1","ts"])   │
│   │ │   │ └─┬─> HashShuffle (KeyColumns: ["id"], HashFunc: "HashV2")                       │
│   │ │   │   └──> TableFullScan (Table: S, ReadColumns: ["id (-∞, +∞)","payload2"])        │
│   │ │   └──> TableFullScan (Table: T, ReadColumns: ["id (-∞, +∞)","payload3"])            │
│   │ └─┬─> HashShuffle (KeyColumns: ["id"], HashFunc: "HashV2")                           │
│   │   └──> TableFullScan (Table: U, ReadColumns: ["id (-∞, +∞)","payload4"])              │
│   └──> TableFullScan (Table: V, ReadColumns: ["id (-∞, +∞)","payload5"])                  │
└──────────────────────────────────────────────────────────────────────────────────────────────┘
```

Since the optimizer may change the join order during query optimization, the join hint should reflect the exact list of tables that are being joined.
For example, in this query, the join order is assumed to be R with S, then T, and finally U. Specifying a different join algorithm may change the join order in the plan, and some join hints will not be applied. In this case, you can add an additional join order hint.

2. Rows - Cardinality Hints

Allows you to change the expected number of rows (optimizer estimate) for a join result or individual tables.

How it works

The optimizer will change its estimate of the cardinality (number of rows) for a join operation that joins only listed tables.

Syntax

```
Rows(t1 t2 ... tn (*|/|+|-|#) Number)
```

Parameters

- `t1 t2 ... tn` - tables
- Operation:
- `*` - multiply by value
- `/` - divide by value
- `+` - add value
- `-` - subtract value
- `#` - replace value
- `Number` - numeric value

Examples

```
-- Multiply expected row count by 2 for join in subtree which contains only tables users orders yandex
Rows(users orders yandex * 2.0)

-- Replace cardinality of table products by 1.3e6
Rows(products # 1.3e6)

-- Reduce expected row count by 228 times
Rows(filtered_table / 228)

-- Add 5000 rows to the expected result
Rows(table1 table2 + 5000)
```

Let's run the query without cardinality hints, and then see how the hints change the query plan.

```
SELECT * FROM
R INNER JOIN S on R.id = S.id
INNER JOIN T on R.id = T.id;
```

Without hints, the optimizer builds the following plan:

```
┌────────┬────────┬────────┬──────────────────────────────────────────────────────────────────────────────────────┐
| E-Cost | E-Rows | E-Size | Operation                                                                            |
├────────┼────────┼────────┼──────────────────────────────────────────────────────────────────────────────────────┤
|        |        |        | ┌> ResultSet                                                                         |
| 114    | 10     | 300    | └─┬> InnerJoin (MapJoin) (S.id = R.id)                                                |
| 57     | 10     | 200    |   ├─┬> InnerJoin (MapJoin) (S.id = T.id)                                              |
| 0      | 10     | 100    |   | ├──> TableFullScan (Table: S, ReadColumns: ["id (-∞, +∞)","payload2"])           |
| 0      | 10     | 100    |   | └──> TableFullScan (Table: T, ReadColumns: ["id (-∞, +∞)","payload3"])           |
| 0      | 10     | 100    |   └──> TableFullScan (Table: R, ReadColumns: ["id (-∞, +∞)","payload1","ts"])        |
└────────┴────────┴────────┴──────────────────────────────────────────────────────────────────────────────────────┘
```

If we specify the following hints:

```
PRAGMA ydb.OptimizerHints =
'
    Rows(R # 10e8)
    Rows(T # 1)
    Rows(S # 10e8)
    Rows(R T # 1)
    Rows(R S # 10e8)
';
SELECT * FROM
    R   INNER JOIN  S   on  R.id = S.id
        INNER JOIN  T   on  R.id = T.id;
```

The resulting plan will look like:

```
┌────────────┬────────┬────────┬──────────────────────────────────────────────────────────────────────────────────────
──┐
| E-Cost     | E-Rows | E-Size | Operation
|
├────────────┼────────┼────────┼──────────────────────────────────────────────────────────────────────────────────────
──┤
|            |        |        | ┌> ResultSet
|
| 3.000e+09  | 1      | 100    | └─┬> InnerJoin (MapJoin) (S.id = R.id)
|
| 0          | 1e+09  | 100    |   ├──> TableFullScan (Table: S, ReadColumns: ["id (-∞, +∞)","payload2"])
|
| 1.500e+09  | 1      | 100    |   └─┬> InnerJoin (MapJoin) (R.id = T.id)
|
| 0          | 1e+09  | 100    |     ├──> TableFullScan (Table: R, ReadColumns: ["id (-∞, +∞)","payload1","t
s"]) |
| 0          | 10     | 100    |     └──> TableFullScan (Table: T, ReadColumns: ["id (-∞, +∞)","payload3"])
|
└────────────┴────────┴────────┴──────────────────────────────────────────────────────────────────────────────────────
──┘
```

Also a warning will be issued:

```
Warning: Unapplied hint: Rows(R S # 10e8)
```

Here you can see that after applying the cardinality hints of the base tables, the order of the joins changed, and one of the hints could not be applied, since there is no such join in the plan.

3. Bytes - Data Size Hints

Allows you to change the expected data size in bytes for a join or individual tables.

Syntax

```
Bytes(t1 t2 ... tn (*|/|+|-|#) Number)
```

Parameters are similar to Rows, but apply to the size of the data in bytes

Examples

```
-- Multiply the expected data size by 1.5
Bytes(large_table * 1.5)

-- Replace the size of the data for the connection with 1GB
Bytes(table1 table2 # 1073741824)

-- Reduce the expected size by 2
Bytes(compressed_table / 2)

-- Add 100MB to the expected size
Bytes(temp_table + 104857600)
```

4. JoinOrder - Join order

Allows you to fix a specific subtree of joins in the overall join tree.

Syntax

```
JoinOrder((t1 t2) (t3 (t4 ...)))
```

Parameters

- Nested parentheses structure defines the join order
- `(t1 t2)` means that t1 and t2 must be joined first
- You can set any nesting depth

How it works

The optimizer will only consider plans that have the specified partial or full join order.

Examples

```
-- Force users to join orders first, then products
JoinOrder((users orders) products)

-- More complex join order
JoinOrder(((customers orders) products) shipping)

-- Join grouping
JoinOrder((table1 table2) (table3 table4))

-- Multi-level structure
JoinOrder((users (orders products)) (addresses phones))
```

Let's apply the join order hint to the following query:

```
SELECT * FROM
R INNER JOIN S on R.id = S.id
INNER JOIN T on R.id = T.id;
```

The query plan without specifying hints is displayed as follows:

```
┌─────────┬────────┬────────┬──────────────────────────────────────────────────────────────────────────────────┐
│ E-Cost  │ E-Rows │ E-Size │ Operation                                                                        │
├─────────┼────────┼────────┼──────────────────────────────────────────────────────────────────────────────────┤
│         │        │        │ ┌> ResultSet                                                                     │
│ 114     │ 10     │ 300    │ └─┬> InnerJoin (MapJoin) (S.id = R.id)                                           │
│ 57      │ 10     │ 200    │   ├─┬> InnerJoin (MapJoin) (S.id = T.id)                                         │
│ 0       │ 10     │ 100    │   │ ├──> TableFullScan (Table: S, ReadColumns: ["id (-∞, +∞)","payload2"])     │
│ 0       │ 10     │ 100    │   │ └──> TableFullScan (Table: T, ReadColumns: ["id (-∞, +∞)","payload3"])     │
│ 0       │ 10     │ 100    │   └──> TableFullScan (Table: R, ReadColumns: ["id (-∞, +∞)","payload1","ts"]) │
└─────────┴────────┴────────┴──────────────────────────────────────────────────────────────────────────────────┘
```

Here you can see that the order of connections has changed to the one specified in the hint.

Combining hints

You can use several types of hints simultaneously within one pragma:

```
PRAGMA ydb.OptimizerHints =
'
    JoinType(users orders Broadcast)
    Rows(users orders * 0.5)
    JoinOrder((users orders) products)
    Bytes(products # 1073741824)
';
```

Parameterized queries and recompilation

Query compilation takes time and resources, so YDB provides a compile cache. The compilation result is stored in the cache on a cluster node and reused only when the query **text matches exactly**. If your application builds YQL with string concatenation or formatting, each new set of values produces a **different text**, and the server recompiles it even when the SQL structure is the same.

Consequences:

- higher latency at the compilation stage;
- increased CPU load on nodes;
- with many unique query texts, reuse of the limited per-node cache gets worse.

A **parameterized query** helps avoid extra compilation: the YQL text stays fixed, and input values are passed separately via named parameters (for example, `$userId`). Below, both approaches are compared on the same example.

Embedding values in the query text

The application inserts values directly into the query text:

```
SELECT id, name FROM users WHERE id = 123 AND status = "active";
SELECT id, name FROM users WHERE id = 456 AND status = "inactive";
```

The queries differ only in values, but the texts are different — for the server these are two different queries, and each one is compiled separately.

Passing values as parameters

The application passes values separately from the text; the query text does not change:

```
DECLARE $userId AS Uint64;
DECLARE $status AS Utf8;

SELECT id, name FROM users WHERE id = $userId AND status = $status;
```

The server handles calls to such a query differently:

1. **First call** — the server compiles the query and stores the result in the cache on the node.
2. **Later calls** — if the text is already in the cache, the ready result is reused: only `$userId` and `$status` change; recompilation is not required.

Only values can be passed as parameters: you cannot parameterize a table name or sort order; changing them changes the query text.

See Query compile cache for cache contents, and Top queries for compilation time when queries run (the `CompileDuration` field).

Cache size and settings, the `KeepInCache` flag, `DECLARE` syntax, and passing parameters from code are described in Parameterized queries in the YDB SDK reference.

See also

- Using query plans for query optimization
- Query compile cache
- Parameterized queries (SDK reference)
- Executing parameterized queries (CLI reference)
- Application examples
- Query execution optimization overview

Uploading data to YDB

This section provides recommendations on efficiently uploading data to YDB.

There are anti-patterns and non-optimal settings for uploading data. They don't guarantee acceptable data uploading performance. To accelerate data uploads, consider the following recommendations:

- Shard a table when creating it. This lets you effectively use the system bandwidth as soon as you start uploading data.
  - By default, a new table consists of a single shard. YDB supports automatic table sharding by data volume. This means that a table shard is divided into two shards when it reaches a certain size.
  - The acceptable size for splitting a table shard is 2 GB. As the number of shards grows, the data upload bandwidth increases, but it remains low for some time at first. Therefore, when uploading a large amount of data for the first time, we recommend initially creating a table with the desired number of shards. You can calculate the number of shards based on 1 GB of data per shard in a resulting set.
- Insert multiple rows in each transaction to reduce the overhead of the transactions themselves.
  - Each transaction in YDB has some overhead. It is recommended to make transactions that insert multiple rows to reduce the total overhead. Good performance indicators terminate a transaction when it reaches 1 MB of data or 100,000 rows.
  - When uploading data, avoid transactions that insert a single row.
- Within each transaction, insert rows from the primary key-sorted set to minimize the number of shards affected by each transaction.
  - In YDB, transactions that span multiple shards have a higher overhead compared to transactions that involve exactly one shard. Moreover, this overhead increases with the growing number of table shards involved in the transaction.
  - We recommend selecting rows to be inserted in a particular transaction so that they're located in a small number of shards, ideally, in one.
- If you need to push data to multiple tables, we recommend pushing data to a single table within a single query.
- If you need to push data to a table with a synchronous secondary index, we recommend that you first push data to a table and, when done, build a secondary index.
- You should avoid writing data sequentially in ascending or descending order of the primary key. Writing data to a table with a monotonically increasing key causes all new data to be written to the end of the table since all tables in YDB are sorted by ascending primary key. As YDB splits table data into shards based on key ranges, inserts are always processed by the same server responsible for the "last" shard. Concentrating the load on a single server will result in slow data uploading and inefficient use of a distributed system.
- Some use cases require writing the initial data (often large amounts) to a table before enabling OLTP workloads. In this case, transactionality at the level of individual queries is not required, and you can use `BulkUpsert` calls in the CLI, SDK and API. Since no transactionality is used, this approach has a much lower overhead than YQL queries. In case of a successful response to the query, the `BulkUpsert` method guarantees that all data added within this query is committed.

> **Warning**
>
> The `BulkUpsert` method isn't supported for tables with synchronous secondary indexes.

We recommend the following algorithm for efficiently uploading data to YDB:

1. Create a table with the desired number of shards based on 1 GB of data per shard.
2. Sort the source data set by the expected primary key.
3. Partition the resulting data set by the number of shards in the table. Each part will contain a set of consecutive rows.
4. Upload the resulting parts to the table shards concurrently.
5. Make a `COMMIT` after every 100,000 rows or 1 MB of data.

Paginated output

This section provides recommendations for organizing paginated data output.

To organize paginated output, we recommend selecting data sorted by primary key sequentially, limiting the number of rows with the LIMIT keyword.

Note

`$lastCity, $lastNumber`: Primary key values obtained from the previous query.

A query demonstrating the recommended way to organize paginated output:

```
--  Table `schools`:
-- ┌─────────┬──────────┬─────┐
-- | Name    | Type     | Key |
-- ├─────────┼──────────┼─────┤
-- | city    | Utf8?    | K0  |
-- | number  | Uint32?  | K1  |
-- | address | Utf8?    |     |
-- └─────────┴──────────┴─────┘

DECLARE $limit AS Uint64;
DECLARE $lastCity AS Utf8;
DECLARE $lastNumber AS Uint32;

SELECT * FROM schools
WHERE (city, number) > ($lastCity, $lastNumber)
ORDER BY city, number
LIMIT $limit;
```

In the query example shown above, the `WHERE` clause uses a tuple comparison to select the next set of rows. Tuples are compared element by element from left to right, so the order of the fields in the tuple must match the order of the fields in the primary key to avoid table full scan.

NULL value in key column

In YDB, all columns, including key ones, may have a NULL value. Despite this, using `NULL` as key column values is highly discouraged, since the SQL standard doesn't allow `NULL` to be compared. As a result, concise SQL statements with simple comparison operators won't work correctly. Instead, you'll have to use cumbersome statements with `IS NULL` / `IS NOT NULL` expressions.

Examples of paginated output implementation

- C++
- Java
- Python
- Go

Using timeouts

This section describes available timeouts and provides examples of their usage in various programming languages.

Prerequisites for using timeouts

The timeout mechanism in YDB is designed to:

- Make sure the query execution time doesn't exceed a certain interval after which its result is not interesting for further use.
- Detect network connectivity issues.

Both of these use cases are important for ensuring the fault tolerance of the entire system. Let's take a closer look at timeouts.

Operation timeout

The `operation_timeout` value shows the time during which the query result is interesting to the user. If the operation fails during this time, the server returns an error with the `Timeout` code and tries to terminate the query, but its cancellation is not guaranteed. So the query that the user was returned the `Timeout` error for can be both successfully executed on the server and canceled.

Timeout for canceling an operation

The `cancel_after` value shows the time after which the server will start canceling the query, if it can be canceled. If canceled, the server returns the `Cancelled` error code.

Transport timeout

The client must set a transport timeout for each query. This value lets you determine the amount of time that the client is ready to wait for a response from the server. If the server doesn't respond during this time, the client will get a transport error with the `DeadlineExceeded` code. Be sure to set such a client timeout value that won't trigger transport timeouts under the normal operation of the application and network.

Using timeouts

We recommend that you always set an operation timeout and transport timeout. The value of the transport timeout should be 50-100 milliseconds more than that of the operation timeout, that way there is some time left for the client to get a server error with the `Timeout` code.

Timeout usage example:

Python

```python
import ydb

def execute_in_tx(session, query):
  settings = ydb.BaseRequestSettings()
  settings = settings.with_timeout(0.5)  # transport timeout
  settings = settings.with_operation_timeout(0.4)  # operation timeout
  settings = settings.with_cancel_after(0.4)  # cancel after timeout
  session.transaction().execute(
      query,
      commit_tx=True,
      settings=settings,
  )
```

C++

```cpp
#include <ydb/public/sdk/cpp/client/ydb.h>
#include <ydb/public/sdk/cpp/client/ydb_table.h>
#include <ydb/public/sdk/cpp/client/ydb_value.h>

using namespace NYdb;
using namespace NYdb::NTable;

TAsyncStatus ExecuteInTx(TSession& session, TString query, TParams params) {
  return session.ExecuteDataQuery(
      query
      , TTxControl::BeginTx(TTxSettings::SerializableRW()).CommitTx()
      , TExecDataQuerySettings()
      .OperationTimeout(TDuration::MilliSeconds(300))  // operation timeout
      .ClientTimeout(TDuration::MilliSeconds(400))   // transport timeout
      .CancelAfter(TDuration::MilliSeconds(300)));  // cancel after timeout
}
```

Go

```go
import (
  "context"
  "time"

  ydb "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/table"
)

func executeInTx(ctx context.Context, s table.Session, query string) {
ctx, cancel := context.WithTimeout(ctx, time.Millisecond*500) // transport timeout
defer cancel()
ctx = ydb.WithOperationTimeout(ctx, time.Millisecond*400)     // operation timeout
ctx = ydb.WithOperationCancelAfter(ctx, time.Millisecond*400) // cancel after timeout
tx := table.TxControl(table.BeginTx(table.WithSerializableReadWrite()), table.CommitTx())
```

```
_, res, err := s.Execute(ctx, tx, query, table.NewQueryParameters())
}
```

System database views

To obtain system information about the database state, you can refer to system views. They are available from the root of the database tree and use the system path prefix `.sys`.

> **Note**
>
> Frequent access to system views leads to additional load on the database, especially in the case of a large database size. Exceeding a frequency of 1 query per second is not recommended.

Partitions

The following system view stores detailed information about partitions of database tables:

- `partition_stats` — contains information about instantaneous metrics and cumulative counters of operations. The former include, for example, CPU load data or the number of running transactions. The latter include the total number of rows read.

It is intended, for example, for identifying unevenly loaded partitions or displaying the size of data in them.

Instantaneous metrics (`NodeId`, `AccessTime`, `CPUCores`, etc.) contain instantaneous values.
Cumulative (non-instantaneous) metrics (`RowReads`, `RowUpdates`, `LocksAcquired`, etc.) store accumulated values since the last start of the tablet (`StartTime`) serving the partition.

View structure:

Column	Description	Data type	Instantaneous/Cumulative
`OwnerId`	Table SchemeShard ID. Key: `0`.	`Uint64`	Instant
`PathId`	Path identifier in SchemeShard. Key: `1`.	`Uint64`	Instant
`PartIdx`	Partition sequence number. Key: `2`.	`Uint64`	Instantaneous
`FollowerId`	Identifier of the replica of the partition tablet. A value of 0 means the leader. Key: `3`.	`Uint32`	Instantaneous
`DataSize`	Approximate data size of the partition in bytes.	`Uint64`	Instantaneous
`RowCount`	Approximate number of rows.	`Uint64`	Instant
`IndexSize`	Partition index size in bytes.	`Uint64`	Instantaneous
`CPUCores`	Instantaneous load value on the partition (fraction of CPU core time spent by the partition actor).	`Double`	Instant
`TabletId`	Partition tablet identifier.	`Uint64`	Instantaneous
`Path`	Full path to the table.	`Utf8`	Momentary
`NodeId`	Node ID on which the partition is currently served.	`Uint32`	Instantaneous
`StartTime`	Last start time of the partition tablet.	`Timestamp`	Instantaneous
`AccessTime`	Last read moment from the partition.	`Timestamp`	Instantaneous
`UpdateTime`	Last write moment to the partition.	`Timestamp`	Instant
`RowReads`	Number of reads by key.	`Uint64`	Cumulative
`RowUpdates`	Number of written rows.	`Uint64`	Cumulative
`RowDeletes`	Number of deleted rows.	`Uint64`	Cumulative
`RangeReads`	Number of reads by key range.	`Uint64`	Cumulative
`RangeReadRows`	Number of rows read in ranges.	`Uint64`	Cumulative
`InFlightTxCount`	Number of running transactions.	`Uint64`	Instant
`ImmediateTxCompleted`	Number of completed single-shard transactions.	`Uint32`	Cumulative
`CoordinatedTxCompleted`	Number of completed distributed transactions.	`Uint64`	Cumulative
`TxRejectedByOverload`	Number of transactions cancelled due to high load.	`Uint64`	Cumulative
`TxRejectedByOutOfStorage`	Number of transactions cancelled due to insufficient storage space.	`Uint64`	Cumulative
`TxCompleteLag`	Transaction execution delay (how far transactions lag behind the scheduled time).	`Interval`	Instantaneous
`LastTtlRunTime`	Last start time of partition cleanup by TTL	`Timestamp`	Instant
`LastTtlRowsProcessed`	Number of partition rows checked during the last TTL cleanup	`Uint64`	Instant

`LastTtlRowsErased`	Number of partition rows deleted during the last TTL cleanup	`Uint64`	Instant
`LocksAcquired`	Number of acquired locks.	`Uint64`	Cumulative
`LocksWholeShard`	Number of set whole-shard locks.	`Uint64`	Cumulative
`LocksBroken`	Number of broken locks.	`Uint64`	Cumulative

Query examples

Top 5 most loaded partitions among all database tables:

```
SELECT
    Path,
    PartIdx,
    CPUCores
FROM `.sys/partition_stats`
ORDER BY CPUCores DESC
LIMIT 5
```

List of database tables with sizes and current load:

```
SELECT
    Path,
    COUNT(*) as Partitions,
    SUM(RowCount) as Rows,
    SUM(DataSize) as Size,
    SUM(CPUCores) as CPU
FROM `.sys/partition_stats`
GROUP BY Path
```

List of database tables with the highest number of broken locks:

```
SELECT
    Path,
    COUNT(*) as Partitions,
    SUM(LocksBroken) as TotalLocksBroken
FROM `.sys/partition_stats`
GROUP BY Path
ORDER BY TotalLocksBroken DESC
```

Top queries

The following system views store data for analyzing user queries.

Maximum total query execution time:

- `top_queries_by_duration_one_minute` — data is divided into minute intervals, contains data for the last 6 hours.
- `top_queries_by_duration_one_hour`: data is split into hourly intervals, contains data for the last 2 weeks.

The largest number of bytes read from the table:

- `top_queries_by_read_bytes_one_minute` — data is split into minute intervals, contains data for the last 6 hours.
- `top_queries_by_read_bytes_one_hour` — data is split into hour intervals, contains data for the last 2 weeks.

Maximum CPU time spent:

- `top_queries_by_cpu_time_one_minute` — data is split into minute intervals, contains data for the last 6 hours.
- `top_queries_by_cpu_time_one_hour` — data is split into hour intervals, contains data for the last 2 weeks.

Queries with the same text are combined, and the query with the maximum value of the corresponding metric is included in the output. Each time interval (minute or hour) contains the TOP-5 queries executed in that time interval.

Fields providing information about CPU time spent (... `CPUTime`) are expressed in microseconds.

Query text is limited to 10 kilobytes.

All views have the same structure:

Column	Description
`IntervalEnd`	End time of the minute or hour interval for which statistics were collected. Type: `Timestamp`. Key: `0`.
`Rank`	Query rank in the top. Type: `Uint32`. Key: `1`.
`QueryText`	Query text. Type: `Utf8`.
`Duration`	Total query execution time. Type: `Interval`.
`EndTime`	End time of query execution. Type: `Timestamp`.

`Type`	Query type ("data", "scan", "script"). Type: `String`.
`ReadRows`	Number of rows read. Type: `Uint64`.
`ReadBytes`	Number of bytes read. Type: `Uint64`.
`UpdateRows`	Number of rows written. Type: `Uint64`.
`UpdateBytes`	Number of bytes written. Type: `Uint64`.
`DeleteRows`	Number of rows deleted. Type: `Uint64`.
`DeleteBytes`	Number of bytes deleted. Type: `Uint64`.
`Partitions`	Number of table partitions involved in query execution. Type: `Uint64`.
`UserSID`	User security ID. Type: `String`.
`ParametersSize`	Size of query parameters in bytes. Type: `Uint64`.
`CompileDuration`	Query compilation duration. Type: `Interval`.
`FromQueryCache`	Whether the prepared query cache was used. Type: `Bool`.
`CPUTime`	Total CPU time used for query execution (microseconds). Type: `Uint64`.
`ShardCount`	Number of shards involved in query execution. Type: `Uint64`.
`SumShardCPUTime`	Total CPU time spent in shards. Type: `Uint64`.
`MinShardCPUTime`	Minimum CPU time spent in shards. Type: `Uint64`.
`MaxShardCPUTime`	Maximum CPU time spent in shards. Type: `Uint64`.
`ComputeNodesCount`	Number of compute nodes involved in query execution. Type: `Uint64`.
`SumComputeCPUTime`	Total CPU time spent in compute nodes. Type: `Uint64`.
`MinComputeCPUTime`	Minimum CPU time spent in compute nodes. Type: `Uint64`.
`MaxComputeCPUTime`	Maximum CPU time spent in compute nodes. Type: `Uint64`.
`CompileCPUTime`	CPU time spent on query compilation. Type: `Uint64`.
`ProcessCPUTime`	CPU time spent on general query processing. Type: `Uint64`.

Query examples

Top queries by execution time. The query is run against the `.sys/top_queries_by_duration_one_minute` view:

```
PRAGMA AnsiInForEmptyOrNullableItemsCollections;
$last = (
    SELECT
        MAX(IntervalEnd)
    FROM `.sys/top_queries_by_duration_one_minute`
);
SELECT
    IntervalEnd,
    Rank,
    QueryText,
    Duration
FROM `.sys/top_queries_by_duration_one_minute`
WHERE IntervalEnd IN $last
```

Queries that read the most bytes. The query is run against the `.sys/top_queries_by_read_bytes_one_minute` view:

```
SELECT
    IntervalEnd,
    QueryText,
    ReadBytes,
    ReadRows,
    Partitions
FROM `.sys/top_queries_by_read_bytes_one_minute`
WHERE Rank = 1
```

Detailed query information

The following system view contains detailed query information:

- `query_metrics_one_minute` — data is split into minute intervals, contains up to 256 queries for the last 6 hours.

Each row of the view contains information about multiple queries with the same text that occurred during the interval. The view fields contain the minimum, maximum, and total values for each tracked query characteristic. Within the interval, queries are sorted in descending order of total CPU time spent.

Limitations:

- query text is limited to 10 kilobytes.
- statistics may be incomplete if the database is under heavy load.

View structure:

Column	Description
`IntervalEnd`	End time of the minute interval for which statistics were collected Type: `Timestamp`. Key: `0`.
`Rank`	Query rank within the interval (by SumCPUTime field). Type: `Uint32`. Key: `1`.
`QueryText`	Query text. Type: `Utf8`.
`Count`	Number of query runs. Type: `Uint64`.
`SumDuration`	Total query duration. Type: `Interval`.
`MinDuration`	Minimum query duration. Type: `Interval`.
`MaxDuration`	Maximum query duration. Type: `Interval`.
`SumCPUTime`	Total CPU time spent. Type: `Uint64`.
`MinCPUTime`	Minimum CPU time spent. Type: `Uint64`.
`MaxCPUTime`	Maximum CPU time spent. Type: `Uint64`.
`SumReadRows`	Total number of rows read. Type: `Uint64`.
`MinReadRows`	Minimum number of rows read. Type: `Uint64`.
`MaxReadRows`	Maximum number of rows read. Type: `Uint64`.
`SumReadBytes`	Total number of bytes read. Type: `Uint64`.
`MinReadBytes`	Minimum number of bytes read. Type: `Uint64`.
`MaxReadBytes`	Maximum number of bytes read. Type: `Uint64`.
`SumUpdateRows`	Total number of rows written. Type: `Uint64`.
`MinUpdateRows`	Minimum number of rows written. Type: `Uint64`.
`MaxUpdateRows`	Maximum number of rows written. Type: `Uint64`.
`SumUpdateBytes`	Total number of bytes written. Type: `Uint64`.

`MinUpdateBytes`	Minimum number of bytes written. Type: `Uint64`.
`MaxUpdateBytes`	Maximum number of bytes written. Type: `Uint64`.
`SumDeleteRows`	Total number of rows deleted. Type: `Uint64`.
`MinDeleteRows`	Minimum number of rows deleted. Type: `Uint64`.
`MaxDeleteRows`	Maximum number of rows deleted. Type: `Uint64`.
`LocksBrokenAsBreaker`	Number of locks broken by this query. Type: `Uint64`.
`LocksBrokenAsVictim`	Number of locks of this query that were broken. Type: `Uint64`.

Note

This statistic does not include:

- Locks broken due to schema changes, TTL, asynchronous replication
- Locks broken due to partition splitting or merging
- Locks broken during tablet restart
- Locks broken when committing interactive transactions (when COMMIT is executed as a separate query)

Query examples

Top 10 queries over the last 6 hours by total number of rows written in a minute interval:

```
SELECT
    SumUpdateRows,
    Count,
    QueryText,
    IntervalEnd
FROM `.sys/query_metrics_one_minute`
ORDER BY SumUpdateRows DESC LIMIT 10
```

Recent queries that read the most bytes per minute:

```
SELECT
    IntervalEnd,
    SumReadBytes,
    MinReadBytes,
    SumReadBytes / Count as AvgReadBytes,
    MaxReadBytes,
    QueryText
FROM `.sys/query_metrics_one_minute`
WHERE SumReadBytes > 0
ORDER BY IntervalEnd DESC, SumReadBytes DESC
LIMIT 100
```

Detailed information about session queries

The following system view contains detailed information about current sessions and the queries being executed in them.

- `.sys/query_sessions` — contains information about sessions and the queries being executed in them.

Column	Description
`SessionId`	Unique session identifier. Type: `Utf8`. Key: `0`.
`NodeId`	Identifier of the node where the session is running. Type: `Uint32`.
`State`	Current session state. Type: `Utf8`.
`Query`	Text of the last or currently executing query. Type: `Utf8`.
`QueryCount`	Number of queries executed within this session. Type: `Uint32`.
`ClientAddress`	Network address of the client that initiated the session. Type: `Utf8`.
`ClientPID`	Process ID (PID) of the client application. Type: `Utf8`.

`ClientUserAgent`	Client software information (User-Agent). Type: `Utf8`.
`ClientSdkBuildInfo`	Client SDK build information. Type: `Utf8`.
`ApplicationName`	Application name specified by the client when connecting. Type: `Utf8`.
`SessionStartAt`	Session start (creation) time. Type: `Timestamp`.
`QueryStartAt`	Start time of the current query execution. Type: `Timestamp`.
`StateChangeAt`	Time of the last session state change. Type: `Timestamp`.
`UserSID`	Security ID of the user owning the session. Type: `Utf8`.
`WmPoolId`	Identifier of the Workload Manager pool in which the session query is executed. Type: `Utf8`.
`WmState`	Query state in Workload Manager. Type: `Utf8`.
`WmEnterTime`	Time when the query transitioned to PENDING or DELAYED status. Type: `Timestamp`.
`WmExitTime`	Time when the query was submitted for execution. Type: `Timestamp`.

Possible values of the `WmState` field:

- `NONE` - Not being processed.
- `PENDING` - Being processed (in classification/routing).
- `DELAYED` - In queue.
- `EXITED` - Submitted for execution.

Possible values of the `State` field:

- `IDLE` - Session is waiting for a query.
- `EXECUTING` - Query is being executed.

Query examples

View all active sessions:

```
SELECT * FROM `.sys/query_sessions`
```

Top 20 longest running queries:

```
SELECT
    Query,
    SessionId,
    NodeId,
    QueryStartAt
FROM `.sys/query_sessions`
WHERE State = 'EXECUTING'
ORDER BY QueryStartAt ASC
LIMIT 20
```

Search for application sessions filtered by Workload Manager pool:

```
SELECT
    SessionId,
    Query,
    State,
    WmState,
    ClientAddress
FROM `.sys/query_sessions`
WHERE ApplicationName = 'my_analytics_app'
  AND WmPoolId = 'heavy_queries'
```

Query compilation cache

The following system view contains information about queries stored in the compilation cache on all cluster nodes:

- `compile_cache_queries` — contains information about queries in the compilation cache of all cluster nodes.

> **Warning**
>
> The `compile_cache_queries` system view is not available in serverless mode.

Each cluster node has its own cache of compiled queries. This cache is used by all sessions running on that node: if during query execution its text is already in the cache, no additional compilation is required.

The system view `compile_cache_queries` provides data on the cache state on all or selected cluster nodes. When querying this view, requests are sent to each node unless a restriction on `NodeId` is specified in the `WHERE` condition. The response returns current information about the cache contents for each node, and the resulting table combines the received data.

View structure:

Column	Description
`NodeId`	Node ID where the query is stored in the cache. Type: `Uint32`. Key: `0`.
`QueryId`	Unique query ID in the node cache. Type: `Utf8`. Key: `1`.
`Query`	Query text. If the query exceeds 10 KB, it is truncated. Type: `Utf8`.
`AccessCount`	Number of cache hits for the query text. Type: `Uint64`.
`CompiledAt`	Query compilation time. Type: `Timestamp`.
`UserSID`	Security ID of the user on whose behalf the query was compiled. May be empty for system queries. Type: `Utf8`.
`LastAccessedAt`	Time of the last access to the query compilation result in the cache. Type: `Timestamp`.
`CompilationDurationMs`	Query compilation duration in milliseconds. Type: `Uint64`.
`Warnings`	Warnings that occurred during query compilation. Type: `Utf8`.
`Metadata`	Query parameter types in JSON format. Contains the `parameters` key with parameter names and their types. Type: `Utf8`.
`IsTruncated`	Flag indicating whether the query text was truncated due to exceeding the 10 KB limit. Type: `Bool`.
`QueryType`	Query type, one of: `QUERY_TYPE_SQL_DML` — Table Service `QUERY_TYPE_SQL_GENERIC_QUERY` — Query Service `QUERY_TYPE_SQL_GENERIC_CONCURRENT_QUERY` — Query Service in concurrent mode May be empty for old entries. Type: `Utf8`.
`Syntax`	Query syntax, one of: `SYNTAX_YQL_V1` — YQL `SYNTAX_UNSPECIFIED` — for old entries without syntax information `SYNTAX_PG` — deprecated value for entries compiled before the removal of experimental PostgreSQL compatibility; new queries with this syntax are not accepted Type: `Utf8`.

Example queries

View all queries in the compilation cache:

```
SELECT * FROM `.sys/compile_cache_queries`
```

Top 20 most popular queries by number of accesses from 3 nodes:

```
SELECT
    Query,
    SUM(AccessCount) AS Hits
FROM `.sys/compile_cache_queries`
WHERE NodeId IN (50000, 50001, 50003)
GROUP BY Query
ORDER BY Hits DESC
LIMIT 20
```

Activity statistics by user:

```
SELECT
    UserSID,
    COUNT(DISTINCT QueryId) AS Plans,
    SUM(AccessCount) AS Hits,
    AVG(CompilationDurationMs) AS AvgCompileMs
FROM `.sys/compile_cache_queries`
```

```
GROUP BY UserSID
ORDER BY Hits DESC
```

Search for queries with long compilation:

```
SELECT
    Query,
    NodeId,
    CompilationDurationMs,
    AccessCount
FROM `.sys/compile_cache_queries`
WHERE CompilationDurationMs > 1000
ORDER BY CompilationDurationMs DESC
```

History of overloaded partitions

The following system views contain a history of high load moments on individual partitions of database tables:

- `top_partitions_one_minute` — data is split into minute intervals, contains history for the last 6 hours.
- `top_partitions_one_hour` — data is split into hour intervals, contains history for the last 2 weeks.

Partitions with peak load over 70% (`CPUCores` > 0.7) are included in the views. Within one interval, partitions are ranked by peak load value.

Both views contain the same set of fields:

The view keys are:

- `IntervalEnd` — interval end time;
- `Rank` — partition rank by peak load `CPUCores` in this interval.

For example, if a table has 10 partitions, then `top_partitions_one_hour` for the hour interval `"20.12.2024 10:00-11:00"` will return 10 rows sorted in descending order of `CPUCores`. They will have `Rank` from 1 to 10 and the same `IntervalEnd` `"20.12.2024 11:00"`.

Column	Description
`IntervalEnd`	End time of the minute or hour interval for which statistics are collected. Type: `Timestamp`. Key: `0`.
`Rank`	Partition rank within the interval (by `CPUCores`). Type: `Uint32`. Key: `1`.
`TabletId`	Tablet ID serving the partition. Type: `Uint64`.
`FollowerId`	Replica ID of the partition tablet. Replica value 0 means leader. Type: `Uint32`
`Path`	Full table path. Type: `Utf8`.
`PeakTime`	Peak time within the interval. Type: `Timestamp`.
`CPUCores`	Peak load value on the partition (fraction of CPU core time spent by the partition actor). Type: `Double`.
`NodeId`	Node ID where the partition was located at the peak. Type: `Uint32`.
`DataSize`	Approximate partition size in bytes at the peak. Type: `Uint64`.
`RowCount`	Approximate row count at the peak. Type: `Uint64`.
`IndexSize`	Partition index size in the tablet at the peak. Type: `Uint64`.
`InFlightTxCount`	Number of transactions in progress at the peak. Type: `Uint32`.

Example queries

The following query outputs partitions with CPU consumption over 70% in the specified time interval, with tablet IDs and their sizes at the time of exceeding. The query is run against the `.sys/top_partitions_one_minute` view, which contains data for the last 6 hours broken down by minute intervals:

```
SELECT
    IntervalEnd,
    CPUCores,
    Path,
    TabletId,
    DataSize
FROM `.sys/top_partitions_one_minute`
WHERE CPUCores > 0.7
```

```
AND IntervalEnd BETWEEN Timestamp("2000-01-01T00:00:00Z") AND Timestamp("2099-12-31T00:00:00Z")
ORDER BY IntervalEnd desc, CPUCores desc
```

The following query outputs partitions with CPU consumption over 90% in the specified time interval, with tablet IDs and their sizes at the time of exceeding. The query is run against the `.sys/top_partitions_one_hour` view, which contains data for the last 2 weeks broken down by hourly intervals:

```
SELECT
    IntervalEnd,
    CPUCores,
    Path,
    TabletId,
    DataSize
FROM `.sys/top_partitions_one_hour`
WHERE CPUCores > 0.9
AND IntervalEnd BETWEEN Timestamp("2000-01-01T00:00:00Z") AND Timestamp("2099-12-31T00:00:00Z")
ORDER BY IntervalEnd desc, CPUCores desc
```

Partition history with broken locks

The following system views contain the history of moments with a non-zero number of broken locks `LocksBroken` in individual partitions of database tables:

- `top_partitions_by_tli_one_minute` — data is broken down into minute intervals, contains history for the last 6 hours.
- `top_partitions_by_tli_one_hour` — data is broken down into hourly intervals, contains history for the last 2 weeks.

The views output the top 10 partitions with a non-zero number of broken locks `LocksBroken`. Within a single interval, partitions are ranked by the number of broken locks `LocksBroken`.

The view keys are:

- `IntervalEnd` - the end time of the interval.
- `Rank` - the rank of the partition by the number of broken locks `LocksBroken` in this interval.

For example, `top_partitions_by_tli_one_hour` for an hourly interval `"20.12.2024 10:00-11:00"` will output 10 rows sorted in descending order of `LocksBroken`. They will have `Rank` from 1 to 10 and the same `IntervalEnd` `"20.12.2024 11:00"`.

Both views contain the same set of fields:

Column	Description
`IntervalEnd`	The end time of the minute or hour interval for which statistics were collected. Type: `Timestamp`. Key: `0`.
`Rank`	The rank of the partition within the interval (by `LocksBroken`). Type: `Uint32`. Key: `1`.
`TabletId`	ID of the tablet serving the partition. Type: `Uint64`.
`FollowerId`	ID of the partition tablet replica. Value 0 means the leader. Type: `Uint32`
`Path`	Full path to the table. Type: `Utf8`.
`LocksAcquired`	Number of locks set "on a key range" in this interval. Type: `Uint64`.
`LocksWholeShard`	Number of locks set "on the entire partition" in this interval. Type: `Uint64`.
`LocksBroken`	Number of broken locks in this interval. Type: `Uint64`.
`NodeId`	ID of the node where the partition was located at the peak time. Type: `Uint32`.
`DataSize`	Approximate size of the partition in bytes at the peak time. Type: `Uint64`.
`RowCount`	Approximate number of rows at the peak time. Type: `Uint64`.
`IndexSize`	Size of the partition index in the tablet at the peak time. Type: `Uint64`.

Query examples

The following query outputs partitions in the specified time interval, with tablet IDs and the number of broken locks. The query is run against the `.sys/top_partitions_by_tli_one_minute` view:

```
SELECT
    IntervalEnd,
    LocksBroken,
    Path,
    TabletId
```

```
FROM `.sys/top_partitions_by_tli_one_hour`
WHERE IntervalEnd BETWEEN Timestamp("2000-01-01T00:00:00Z") AND Timestamp("2099-12-31T00:00:00Z")
ORDER BY IntervalEnd desc, LocksBroken desc
```

Resource pool information

The `resource_pools` system view contains information about settings of resource pools.

System view structure:

Column	Description
`Name`	Resource pool name. Type: `Utf8`. Key: `0`.
`ConcurrentQueryLimit`	Maximum number of concurrently executing queries in the resource pool. Type: `Int32`.
`QueueSize`	Maximum queue size. Type: `Int32`.
`DatabaseLoadCpuThreshold`	CPU load threshold of the entire database, in percent, after which queries are not sent for execution and remain in the queue. Type: `Double`.
`ResourceWeight`	Weights for distributing resources among pools. Type: `Double`.
`TotalCpuLimitPercentPerNode`	Percentage of available CPU that all queries on the node can use in this resource pool. Type: `Double`.
`QueryCpuLimitPercentPerNode`	Percentage of available CPU on the node for one query in the resource pool. Type: `Double`.
`QueryMemoryLimitPercentPerNode`	Percentage of available memory on the node that a query can use in this resource pool. Type: `Double`.

Example

The following query outputs information about the settings of the resource pool named `default`:

```
SELECT
    Name,
    ConcurrentQueryLimit,
    QueueSize,
    DatabaseLoadCpuThreshold,
    ResourceWeight,
    TotalCpuLimitPercentPerNode,
    QueryCpuLimitPercentPerNode,
    QueryMemoryLimitPercentPerNode
FROM `.sys/resource_pools`
WHERE Name = "default";
```

Resource pool classifier information

The `resource_pools_classifiers` system view contains information about settings of resource pool classifiers.

System view structure:

Column	Description
`Name`	Resource pool classifier name. Type: `Utf8`. Key: `0`.
`Rank`	Priority of selecting the resource pool classifier. Type: `Int64`.
`MemberName`	User or group of users that will be sent to the specified resource pool. Type: `Utf8`.
`ResourcePool`	The name of the resource pool to which queries will be sent. Type: `Utf8`.

Example

The following query outputs information about the settings of the resource pool classifier named `olap`:

```
SELECT
    Name,
    Rank,
    MemberName,
    ResourcePool
FROM `.sys/resource_pools_classifiers`
WHERE Name = "olap";
```

Users, groups, and access rights

The following system views contain information about users, access groups, user membership in groups, and access rights granted to groups or directly to users.

Information about users

The `auth_users` view contains a list of local users YDB. It does not include users authenticated through external systems such as LDAP.

Administrators have full access to this view. Regular users can only view their own data.

Table structure:

Column	Description
`Sid`	User SID. Type: `Utf8`. Key: `0`.
`IsEnabled`	Indicates whether login is allowed for this user; used for explicit blocking by an administrator. Independent of `IsLockedOut`. Type: `Bool`.
`IsLockedOut`	Indicates that this user is automatically blocked due to exceeding the number of failed authentications. Does not depend on `IsEnabled`. Type: `Bool`.
`CreatedAt`	User creation time. Type: `Timestamp`.
`LastSuccessfulAttemptAt`	Time of last successful authentication. Type: `Timestamp`.
`LastFailedAttemptAt`	Time of last failed authentication. Type: `Timestamp`.
`FailedAttemptCount`	Number of failed authentications. Type: `Uint32`.
`PasswordHash`	JSON string containing the password hash, salt, and hashing algorithm. Type: `Utf8`.

Information about groups

The `auth_groups` view contains a list of access groups.

Only administrators have access to this view.

Table structure:

Column	Description
`Sid`	Group SID. Type: `Utf8`. Key: `0`.

Information about group membership

The `auth_group_members` view contains information about membership in access groups.

Only administrators have access to this view.

Table structure:

Column	Description
`GroupSid`	Group SID. Type: `Utf8`. Key: `0`.
`MemberSid`	Group member SID. Can be either a user SID or a group SID. Type: `Utf8`. Key: `1`.

Information about access rights

The views contain a list of granted access rights.

Include two views:

- `auth_permissions`: Explicitly granted access rights.
- `auth_effective_permissions`: Effective access rights considering inheritance.

In this view, the user sees only those access objects for which they have the `ydb.granular.describe_schema` right.

Table structure:

Column	Description

`Path`	Path to the access object. Type: `Utf8`. Key: `0`.
`Sid`	SID of the access subject. Type: `Utf8`. Key: `1`.
`Permission`	Name of the access right YDB. Type: `Utf8`. Key: `2`.

Example queries

Getting explicitly granted rights on an access object - the `my_table` table:

```
SELECT *
FROM `.sys/auth_permissions`
WHERE Path = "my_table"
```

Getting effective rights on an access object - the `my_table` table:

```
SELECT *
FROM `.sys/auth_effective_permissions`
WHERE Path = "my_table"
```

Getting rights granted to user `user3`:

```
SELECT *
FROM `.sys/auth_permissions`
WHERE Sid = "user3"
```

Information about owners of access objects

The `auth_owners` view displays information about owners of access objects.

In this view, the user sees only those access objects for which they have been granted the `ydb.granular.describe_schema` right.

Table structure:

Column	Description
`Path`	Path to the access object. Type: `Utf8`. Key: `0`.
`Sid`	SID of the access object owner. Type: `Utf8`.

Streaming queries

Viewing information about streaming queries

The system view `streaming_queries` contains information about all created streaming queries.

In this view, the user sees only those streaming queries for which they have been granted the `ydb.granular.describe_schema` right.

Table structure:

Column	Description
`Path`	Full path to the query. Type: `Utf8`. Key: `0`.
`Status`	Query execution status, one of: `CREATING` - query is being created, `CREATED` - query created but not started, `STARTING` - query is starting, `RUNNING` - query is running, `STOPPING` - query is stopping, `STOPPED` - query stopped, `SUSPENDED` - query completed with error and is waiting for backoff to retry, Type: `Utf8`
`Issues`	Query execution errors in JSON format Type: `Utf8`
`Plan`	Query plan in JSON format Type: `Utf8`
`Ast`	Query AST Type: `Utf8`
`Text`	Query text Type: `Utf8`

`Run`	Whether the query is currently being run by the user Type: `Bool`
`ResourcePool`	Name of the resource pool the query was bound to (see example) Type: `Utf8`
`RetryCount`	Number of query restarts Type: `Uint64`
`LastFailAt`	Time of the last query execution error Type: `Timestamp`
`SuspendedUntil`	Time when an attempt will be made to resume a stopped query Type: `Timestamp`

Change Data Capture

With Change Data Capture (CDC), you can track changes in table data. YDB provides access to changefeeds so that data consumers can monitor changes in near real time.

Enabling and disabling CDC

CDC is represented as a data schema object: a changefeed that can be added to a table or deleted from them using the ADD CHANGEFEED and DROP CHANGEFEED directives of the YQL `ALTER TABLE` statement.

Reading data from a topic

Before reading data, add a consumer. Below is a sample command that adds a consumer named `my_consumer` to the `updates_feed` changefeed of the `table` table in the `my` directory:

```
ydb topic consumer add \
  my/table/updates_feed \
  --consumer=my_consumer
```

Next, you can use the created consumer to start tracking changes. Below is a sample command for tracking data changes in the CLI:

```
ydb topic read \
  my/table/updates_feed \
  --consumer=my_consumer \
  --format=newline-delimited \
  --wait
```

Impact on table write performance

When writing data to a table with CDC enabled, there are additional overheads for the following operations:

- Making records and saving them to a changefeed.
- Storing records in a changefeed.
- In some modes (such as `OLD_IMAGE` and `NEW_AND_OLD_IMAGES`), data needs to be pre-fetched even if a user query doesn't require this.

As a result, queries may take longer to execute and size limits for stored data may be exceeded.

In real-world use cases, enabling CDC has virtually no impact on the query execution time (whatever the mode), since almost all data required for making records is stored in the cache , while the records themselves are sent to a topic asynchronously. However, record delivery background activity slightly (by 1% to 10%) increases CPU utilization.

When creating a changefeed for a table, the number of partitions of its storage (topic) is determined based on the current number of table partitions. If the number of source table partitions changes significantly (for example, after uploading a large amount of data or as a result of intensive accesses), an imbalance occurs between the table partitions and the topic partitions. This imbalance can also result in longer execution time for queries to modify data in the table or in unnecessary storage overheads for the changefeed. You can recreate the changefeed to correct the imbalance.

Load testing

As a load generator, you can use the feature of emulating an online store built into the YDB CLI:

1. Initialize a test.
2. Add a changefeed:

```
ALTER TABLE `orders` ADD CHANGEFEED `updates` WITH (
    FORMAT = 'JSON',
    MODE = 'UPDATES'
);
```

1. Create a consumer:

```
ydb topic consumer add \
  orders/updates \
  --consumer=my_consumer
```

2. Start tracking changes:

```
ydb topic read \
  orders/updates \
  --consumer=my_consumer \
  --format=newline-delimited \
  --wait
```

3. Generate a load.

   The following changefeed appears in the CLI:

```
...
{"update":{"created":"2022-06-24T11:35:00.000000Z","customer":"Name366"},"key":[1319569999286404932]}
{"update":{"created":"2022-06-24T11:35:00.000000Z","customer":"Name3894"},"key":[452209497351143909]}
{"update":{"created":"2022-06-24T11:35:00.000000Z","customer":"Name7773"},"key":[2377978894183850258]}
...
```

Managing YDB using Terraform

Terraform can create, delete, and modify the following objects inside a YDB cluster:

- tables
- indexes of tables
- change data capture for tables
- topics

> **Warning**
>
> Currently, the YDB provider for Terraform is under development, and its functionality will be expanded.

To get started, you need to:

1. Deploy the YDB cluster
2. Create a database (described in paragraph 1 for the appropriate type of cluster deployment)
3. Install Terraform
4. Install and configure Terraform provider for YDB

Configuring the Terraform provider to work with YDB

1. You need to download provider code
2. Build the provider by executing `$make local-build` in the root directory of the provider's code. To do this, you need to additionally install the make utility and go
   The provider will be installed in the Terraform plugins folder - `~/.terraform.d/plugins/terraform.storage.ydb.tech/...`
3. Add the provider to `~/.terraformrc` by adding the following content to the `provider_installation` section (if there was no such section yet, then create):

```
provider_installation {
  direct {
    exclude = ["terraform.storage.ydb.tech/*/*"]
  }

  filesystem_mirror {
    path    = "/PATH_TO_HOME/.terraform.d/plugins"
    include = ["terraform.storage.ydb.tech/*/*"]
  }
}
```

4. Next, we configure the YDB provider itself to work (for example, in the file `provider.tf` in the working directory):

```
terraform {
  required_providers {
    ydb = {
      source = "terraform.storage.ydb.tech/provider/ydb"
    }
  }
  required_version = ">= 0.13"
}

provider "ydb" {
  token = "<TOKEN>"
  //OR for static credentials
  user = "<USER>"
  password = "<PASSWORD>"
}
```

Where:

- `token` - specifies the access token to the database if authentication is used, for example, using a third-party IAM provider.
- `user` - the username for accessing the database in case of using authentication by username and password
- `password` - the password for accessing the database in case of using authentication by username and password

Using the Terraform provider YDB

The following commands are used to apply changes to terraform resources:

1. `terraform init` - initialization of the terraform module (performed in the terraform resource directory).
2. `terraform validate` - checking the syntax of terraform resource configuration files.
3. `terraform apply` - direct application of the terraform resource configuration.

For ease of use, it is recommended to name terraform files as follows:

1. `provider.tf` - contains the settings of the terraform provider itself.
2. `main.tf` - contains a set of resources to create.

Database connection

For all resources describing data schema objects, you must specify the database details in which they are located. To do this, provide one of the two arguments:

- The connection string `connection_string` is an expression of the form `grpc(s)://HOST:PORT/?database=/database/path`, where `grpc(s)://HOST:PORT/` endpoint, and `/database/path` is the path of the database.
  For example, `grpcs://example.com:2135?database=/Root/testdb0`.

- `database_endpoint` - used when working with the [topics] resource (#topic_resource) (analogous to `connection_string` when working with table resources).

> **Note**
>
> The user can transfer the connection string to the database using standard Terraform tools - via variables.

If you are using the creation of `ydb_table_changefeed` or `ydb_topic` resources and authorization is not enabled on the YDB server, then in the DB config config.yaml you need to specify:

```
...
pqconfig:
  require_credentials_in_new_protocol: false
  check_acl: false
```

Example of using all types of YDB Terraform provider resources

This example combines all types of resources that are available in the YDB Terraform provider:

```
variable "db-connect" {
  type = string
  default = "grpc(s)://HOST:PORT/?database=/database/path" # you need to specify the path to the database
}

resource "ydb_table" "table" {
  path        = "1/2/3/tftest"
  connection_string = var.db-connect
  column {
    name = "a"
    type = "Utf8"
  }
  column {
    name = "b"
    type = "String"
  }
  column {
    name = "ttlBase"
    type = "Uint32"
  }
  ttl {
    column_name = "ttlBase"
    expire_interval = "P7D"
    unit = "milliseconds"
  }

  primary_key = ["b", "a"]

  partitioning_settings {
    auto_partitioning_min_partitions_count = 5
    auto_partitioning_max_partitions_count = 8
    auto_partitioning_partition_size_mb    = 256
    auto_partitioning_by_load              = true
  }
}

resource "ydb_table_index" "table_index" {
  table_path        = ydb_table.table.path
  connection_string = ydb_table.table.connection_string
  name              = "my_index"
  type              = "global_sync" # "global_async"
  columns           = ["a", "b"]

  depends_on = [ydb_table.table] # link to the table creation resource
}

resource "ydb_table_changefeed" "table_changefeed" {
  table_id = ydb_table.table.id
  name     = "changefeed"
  mode     = "NEW_IMAGE"
  format   = "JSON"
  consumer {
    name = "test"
    supported_codecs = ["raw", "gzip"]
  }

  depends_on = [ydb_table.table] # link to the table creation resource
}

resource "ydb_topic" "test" {
  database_endpoint = ydb_table.table.connection_string
  name              = "1/2/test"
  supported_codecs  = ["zstd"]

  consumer {
    name             = "test-consumer3"
    starting_message_timestamp_ms = 0
    supported_codecs = ["zstd","raw"]
  }
```

```
  consumer {
    name             = "test-consumer1"
    starting_message_timestamp_ms = 2000
    supported_codecs = ["zstd"]
  }

  consumer {
    name             = "test-consumer2"
    starting_message_timestamp_ms = 0
    supported_codecs = ["zstd"]
  }
}
```

All resources of the YDB Terraform provider will be described in detail below.

String table

> **Note**
>
> Working with column-oriented tables via Terraform is not yet available.

The `ydb_table` resource is used to work with tables.

Example:

```
resource "ydb_table" "ydb_table" {
  path = "path/to/table" # path relative to the base root
  connection_string = "grpc(s)://HOST:PORT/?database=/database/path" #DB connection example
  column {
    name = "a"
    type = "Utf8"
    not_null = true
  }
  column {
    name = "b"
    type = "Uint32"
    not_null = true
  }
  column {
    name = "c"
    type = String
    not_null = false
  }
  column {
    name = "f"
    type = "Utf8"
  }
  column {
    name = "e"
    type = "String"
  }
  column {
    name = "d"
    type = "Timestamp"
  }
  primary_key = ["b", "a"]
}
```

The following arguments are supported:

- `path` - (required) is the path of the table relative to the root of the database (example - `/path/to/table`).
- `connection_string` — (required) connection string.
- `column` — (required) column properties (see the column argument).
- `family` - (optional) is a column group (see the family argument).
- `primary_key` — (required) primary key of the table that contains an ordered list of column names of the primary key.
- `ttl` — (optional) TTL (see the ttl argument).
- `partitioning_settings` — (optional) partitioning settings (see the argument partitioning_settings).
- `key_bloom_filter` — (optional) (bool) use Bloom filter for primary key, the default value is false.
- `read_replicas_settings` — (optional) settings for read replicas.

column

The `column` argument describes the column properties of the table.

> **Warning**
>
> Using Terraform, you cannot only add columns but not delete them. To delete a column, use the YDB tools, then delete the column from the resource description. When trying to apply changes to the table's columns (changing the data type or name), Terraform will not try to delete them but will try to do an update-in-place, though the changes will not be applied.

Example:

```
column {
  name     = "column_name"
```

```
  type     = "Utf8"
  family   = "some_family"
  not_null = true
}
```

- `name` - (required) is the column's name.
- `type` — (required) YQL data type columns. Simple column types are allowed. However, container types cannot be used as data types of table columns.
- `family` - (optional) is the name of the column group (see the family argument).
- `not_null` — (optional) column cannot contain `NULL`. The default value: `false`.

family

The `family` argument describes column group properties.

- `name` - (required) is the name of the column group.
- `data` — (required) storage device type for column data of this group.
- `compression` — (required) data compression codec.

Example:

```
family {
  name        = "my_family"
  data        = "ssd"
  compression = "lz4"
}
```

partitioning_settings

The `partitioning_settings` argument describes partitioning settings.

Example:

```
partitioning_settings {
  auto_partitioning_min_partitions_count = 5
  auto_partitioning_max_partitions_count = 8
  auto_partitioning_partition_size_mb    = 256
  auto_partitioning_by_load              = true
}
```

- `uniform_partitions` — (optional) the number of preallocated partitions.
- `partition_at_keys` — (optional) partitioning by primary key.
- `auto_partitioning_min_partitions_count` — (optional) minimum possible number of partitions when auto-partitioning.
- `auto_partitioning_max_partitions_count` — (optional) maximum possible number of partitions when auto-partitioning.
- `auto_partitioning_partition_size_mb` — (optional) setting the value of auto-partitioning by size in megabytes.
- `auto_partitioning_by_size_enabled` — (optional) enabling auto-partitioning by size (bool), enabled by default (true).
- `auto_partitioning_by_load` — (optional) enabling autopartition by load (bool), disabled by default (false).

See the links above for more information about the parameters and their default values.

ttl

The `ttl` argument describes the Time To Live settings.

Example:

```
ttl {
  column_name     = "column_name"
  expire_interval = "PT1H" # 1 hour
  unit = "seconds" # for numeric column types (non-ISO8601)
}
```

- `column_name` - (required) is the column name for TTL.
- `expire_interval` — (required) interval in ISO 8601 format (for example, `P1D` is an interval of 1 day, that is, 24 hours).
- `unit` — (optional) is set if the column with ttl has a numeric type. Supported values:
  - `seconds`
  - `milliseconds`
  - `microseconds`
  - `nanoseconds`

Secondary index of the table

The ydb_table_index resource is used to work with a table index.

Example:

```
resource "ydb_table_index" "ydb_table_index" {
  table_path = "path/to/table" # path relative to the base root
  connection_string = "grpc(s)://HOST:PORT/?database=/database/path" #DB connection example
  name              = "my_index"
  type              = "global_sync" # "global_async"
  columns           = ["a", "b"]
```

```
  cover             = ["c"]
}
```

The following arguments are supported:

- `table_path` - is the path of the table. Specified if `table_id` is not specified.
- `connection_string` — connection string. Specified if `table_id` is not specified.
- `table_id` - terraform-table identifier. Specify if `table_path` or `connection_string` is not specified.
- `name` - (required) is the name of the index.
- `type` - (required) is the index type global_sync | global_async.
- `columns` - (required) is an ordered list of column names participating in the index.
- `cover` - (required) is a list of additional columns for the covering index.

Change Data Capture

The `ydb_table_changefeed` resource is used to work with the change data capture of the table.

Example:

```
resource "ydb_table_changefeed" "ydb_table_changefeed" {
  table_id = ydb_table.ydb_table.id
  name     = "changefeed"
  mode     = "NEW_IMAGE"
  format   = "JSON"
}
```

The following arguments are supported:

- `table_path` - is the path of the table. Specified if `table_id` is not specified.
- `connection_string` — connection string. Specified if `table_id` is not specified.
- `table_id` — terraform-table identifier. Specify if `table_path` or `connection_string` is not specified.
- `name` - (required) is the name of the change stream.
- `mode` - (required) is the mode of operation of the change data capture.
- `format` - (required) is the format of the change data capture.
- `virtual_timestamps` — (optional) using virtual timestamps.
- `retention_period` — (optional) data storage time in ISO 8601 format.
- `consumer` - (optional) is a reader of the change data capture (see the argument #consumer).

consumer

The `consumer` argument describes the reader of the change data capture.

- `name` - (required) is the reader's name.
- `supported_codecs` — (optional) supported data codec.
- `starting_message_timestamp_ms` — (optional) timestamp in UNIX timestamp format in milliseconds, from which the reader will start reading the data.

Usage examples

Creating a table in an existing database

```
resource "ydb_table" "ydb_table" {
  # Path to the table
  path = "path/to/table" # path relative to the base root

  connection_string = "grpc(s)://HOST:PORT/?database=/database/path" #DB connection example

  column {
    name = "a"
    type = "Uint64"
    not_null = true
  }
  column {
    name     = "b"
    type     = "Uint32"
    not_null = true
  }
  column {
    name = "c"
    type = String
    not_null = false
  }
  column {
    name = "f"
    type = "Utf8"
  }
  column {
    name = "e"
    type = "String"
  }
  column {
    name = "d"
    type = "Timestamp"
  }
  # Primary key
  primary_key = [
```

```
    "a", "b"
  ]
}
```

Creating a table, index, and change stream

```
resource "ydb_table" "ydb_table" {
  # Path to the table
  path = "path/to/table" # path relative to the base root

  # ConnectionString to the database.
  connection_string = "grpc(s)://HOST:PORT/?database=/database/path" #DB connection example

  column {
    name = "a"
    type = "Uint64"
    not_null = true
  }
  column {
    name     = "b"
    type     = "Uint32"
    not_null = true
  }
  column {
    name = "c"
    type = "Utf8"
  }
  column {
    name = "f"
    type = "Utf8"
  }
  column {
    name = "e"
    type = "String"
  }
  column {
    name = "d"
    type = "Timestamp"
  }

  # Primary key
  primary_key = [
    "a", "b"
  ]


  ttl {
    column_name     = "d"
    expire_interval = "PT5S"
  }

  partitioning_settings {
    auto_partitioning_by_load = false
    auto_partitioning_partition_size_mb    = 256
    auto_partitioning_min_partitions_count = 6
    auto_partitioning_max_partitions_count = 8
  }

  read_replicas_settings = "PER_AZ:1"

  key_bloom_filter = true # Default = false
}

resource "ydb_table_changefeed" "ydb_table_changefeed" {
  table_id = ydb_table.ydb_table.id
  name = "changefeed"
  mode = "NEW_IMAGE"
  format = "JSON"

  consumer {
    name = "test_consumer"
  }

  depends_on = [ydb_table.ydb_table] # link to the table creation resource
}

resource "ydb_table_index" "ydb_table_index" {
  table_id = ydb_table.ydb_table.id
  name = "some_index"
  columns = ["c", "d"]
  cover = ["e"]
  type = "global_sync"

  depends_on = [ydb_table.ydb_table] # link to the table creation resource
}
```

Topic configuration management YDB via Terraform

The `ydb_topic` resource is used to work with topics

Note

The topic cannot be created in the root of the database; you need to specify at least one directory in the name of the topic. When trying to create a topic in the root of the database, the provider will return an error.

Description of the `ydb_topic` resource

Example:

```
resource "ydb_topic" "ydb_topic" {
  database_endpoint = "grpcs://example.com:2135/?database=/Root/testdb0" #database connection example
  name              = "test/test1"
  supported_codecs  = ["zstd"]

  consumer {
    name             = "test-consumer1"
    starting_message_timestamp_ms = 0
    supported_codecs = ["zstd","raw"]
  }

  consumer {
    name             = "test-consumer2"
    starting_message_timestamp_ms = 2000
    supported_codecs = ["zstd"]
  }

  consumer {
    name             = "test-consumer3"
    starting_message_timestamp_ms = 0
    supported_codecs = ["zstd"]
  }
}
```

The following arguments are supported:

- `name` - (required) is the name of the topic.
- `database_endpoint` - (required) is the full path to the database, for example: `"grpcs://example.com:2135/?database=/Root/testdb0"`; analogous to `connection_string` for tables.
- `retention_period_ms` - the duration of data storage in milliseconds; the default value is `86400000` (day).
- `partitions_count` - the number of partitions; the default value is `2`.
- `supported_codecs` - supported data compression codecs, the default value is `"gzip", "raw", "zstd"`.
- `consumer` - readers for the topic.

Description of the data consumer `consumer`:

- `name` - (required) is the reader's name.
- `supported_codecs` - supported data compression encodings, by default - `"gzip", "raw", "zstd"`.
- `starting_message_timestamp_ms` - timestamp in UNIX timestamp format in milliseconds, from which the reader will start reading the data; the default value is 0, which means "from the beginning".

Custom attributes in tables

You can use custom attributes to store any information and process it in your app or using the CLI.

For example, your application uses a database table. It stores versions of the table's scheme in custom attributes. You need to migrate data when the table scheme changes. To make sure that your application runs a relevant migration, rather than a previous one, when started, access the custom attributes to check the scheme version:

- The version in the attribute is `1`, and your application knows how to work with version `2`. So, you apply the `1` > `2` migration and update the attribute value with `2`.
- In the attribute, the version is `2`, and your application knows how to work with version `2`, so no migration is needed.
- The version in the attribute is `2`, and your application knows how to work with version `1`: you terminate your application with an exception, notifying the user of an attempt to run an older app version against the new data scheme.

When you use custom attributes, you no longer need to store scheme versions in a separate table.

Set a custom attribute when creating a table

Go

To set a custom attribute when creating the `series` table, pass the `scheme_version` key and the attribute value `1` in the `options.WithAttribute` option of the `CreateTable` method:

```
err := client.Do(ctx, func(ctx context.Context, s table.Session) error {
  return s.CreateTable(ctx, "episodes",
    options.WithColumn("series_id", types.Optional(types.TypeUint64)),
    options.WithColumn("season_id", types.Optional(types.TypeUint64)),
    options.WithColumn("episode_id", types.Optional(types.TypeUint64)),
    options.WithColumn("title", types.Optional(types.TypeText)),
    options.WithPrimaryKeyColumn("series_id", "season_id", "episode_id"),
    options.WithAttribute("scheme_version", "1"),
  )
})
```

Set a custom attribute for an existing table

Go

To set a custom attribute for the existing `series` table, pass the `scheme_version` key and the attribute value `1` in the `options.WithAddAttribute` option of the `AlterTable` method:

```
err = db.Table().Do(ctx,
  func(ctx context.Context, s table.Session) (err error) {
    return s.AlterTable(ctx, path.Join(db.Name(), "series"),
      options.WithAddAttribute("scheme_version", "1"),
    )
  },
)
```

CLI

To set a custom attribute for the existing `series` table, pass the `scheme_version` key and the attribute value `1` in the `--attribute` option of the ydb table attribute add command:

```
ydb table attribute add --attribute scheme_version=1 series
```

Update the custom attribute

Go

To update the custom attribute when changing the table scheme, pass the `scheme_version` key and the new attribute value of `2` in the `WithAlterAttribute` option of the `AlterTable` method:

```
err = db.Table().Do(ctx,
  func(ctx context.Context, s table.Session) (err error) {
    return s.AlterTable(ctx, path.Join(db.Name(), "series"),
      options.WithAddColumn("air_date", types.Optional(types.TypeUint64)),
      options.WithAlterAttribute("scheme_version", "2"),
    )
  },
)
```

CLI

To edit the custom attribute for the existing `series` table, pass the `scheme_version` key and the attribute value `2` in the `--attribute` option of the ydb table attribute add command:

```
ydb table attribute add --attribute scheme_version=2 series
```

View your custom attributes

Go

To retrieve the `series` table scheme, including the custom attributes, use the method `table.Session.DescribeTable()`:

```
err := c.Do(ctx,
  func(ctx context.Context, s table.Session) error {
    description, err := s.DescribeTable(ctx, path.Join(prefix, "series"))
    if err != nil {
      return err
    }
    for k, v := range description.Attributes {
      log.Println(k, "=", v)
    }
    return nil
  },
)
```

CLI

To get the data about the `series` table scheme, including the custom attributes, use the command ydb scheme describe:

```
ydb scheme describe series
```

Result:

```
...
Attributes:
┌────────────────┬────────┐
| Name           | Value |
├────────────────┼────────┤
| scheme_version | 2     |
└────────────────┴────────┘
...
```

Delete the custom attribute

Go

To drop the custom attribute, pass the `scheme_version` key in the option `WithDropAttribute` of the `AlterTable` method:

```
err = db.Table().Do(ctx,
  func(ctx context.Context, s table.Session) (err error) {
    return s.AlterTable(ctx, path.Join(db.Name(), "series"),
      options.WithDropAttribute("scheme_version"),
    )
  },
)
```

CLI

To drop the custom attribute, use the `scheme_version` key in the `--attributes` option of the ydb table attribute drop command:

```
ydb table attribute drop --attributes scheme_version series
```

Workload Manager — resource consumption management

Resource pools allow isolating database resources between executing queries or configuring resource allocation strategies in case of oversubscription (requesting more resources than available in the system). All resource pools are equal, without any hierarchy, and affect each other only when there is a general resource shortage.

For example, one of the typical resource isolation scenarios is separating two classes of consumers:

1. A regular robot process that builds a report once a day.
2. Analysts who run ad hoc queries.

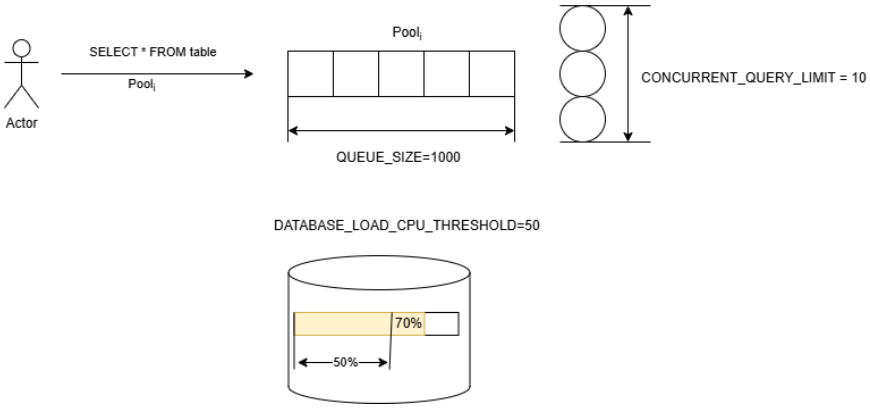
> **Warning**
>
> The resource consumption management functionality described is in Preview.

Creating a resource pool

The example below shows the syntax for creating a separate resource pool named "olap" that will run analytical queries:

```
CREATE RESOURCE POOL olap WITH (
    CONCURRENT_QUERY_LIMIT=10,
    QUEUE_SIZE=1000,
    DATABASE_LOAD_CPU_THRESHOLD=80,
    RESOURCES_WEIGHT=100,
    QUERY_CPU_LIMIT_PERCENT_PER_NODE=50,
    TOTAL_CPU_LIMIT_PERCENT_PER_NODE=70
)
```

You can find the full list of resource pool parameters in the Parameters reference. Some parameters are global for the entire database (for example, `CONCURRENT_QUERY_LIMIT`, `QUEUE_SIZE`, `DATABASE_LOAD_CPU_THRESHOLD`), while others apply only to a single compute node (for example, `QUERY_CPU_LIMIT_PERCENT_PER_NODE`, `TOTAL_CPU_LIMIT_PERCENT_PER_NODE`, `TOTAL_MEMORY_LIMIT_PERCENT_PER_NODE`). CPU can be shared among all pools in case of oversubscription on a single compute node using `RESOURCES_WEIGHT`.



Let's look at the example above to see what these parameters actually mean and how they will affect resource allocation. Suppose the YDB database has nodes with each. In total, such a database has . Then on each node, for the resource pool named `olap`, the following will be allocated:

In total, with uniform resource distribution across the entire database, the resource pool will be allocated:

For one query in this resource pool, the following will be allocated:

How CONCURRENT_QUERY_LIMIT and QUEUE_SIZE work

Suppose 9 queries are already running in the resource pool `olap`. When a new query arrives, it will immediately start executing in parallel with the other 9 queries. Now there will be 10 queries running in the pool. If an 11th query arrives, it will not start executing but will be placed in the waiting queue. When at least one of the 10 running queries completes, the 11th query will be taken from the queue and start executing.

If there are already queries in the queue, then when the 1001st query is sent, the client will immediately receive an error in response, and this query will not be executed. Example error:

```
Issues:
<main>: Error: Request was rejected, number of local pending requests is 20, number of global delayed/running
requests is 0, sum of them is larger than allowed limit 1 (including concurrent query limit 1) for pool olap
<main>: Error: Query failed during adding/waiting in workload pool olap
```

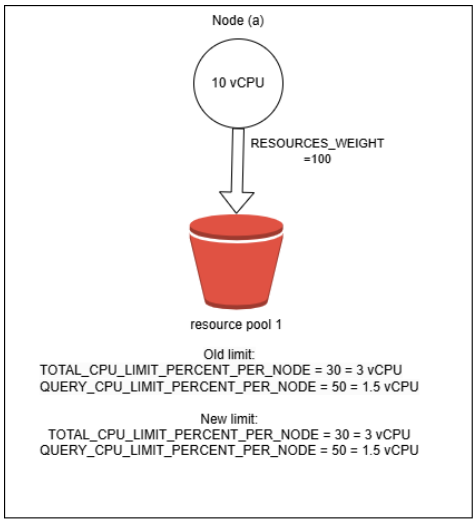
The number of concurrently executing queries is affected not only by `CONCURRENT_QUERY_LIMIT` but also by `DATABASE_LOAD_CPU_THRESHOLD`.

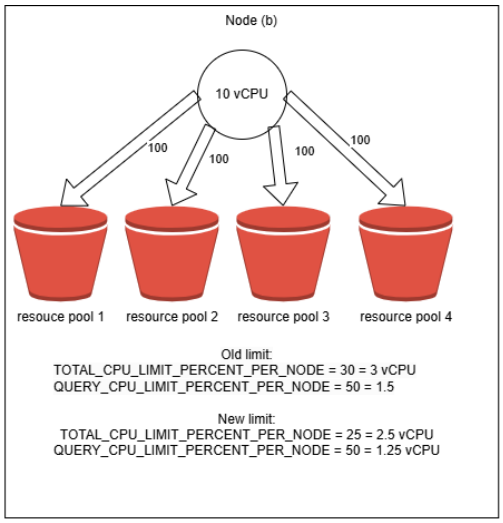
How DATABASE_LOAD_CPU_THRESHOLD works

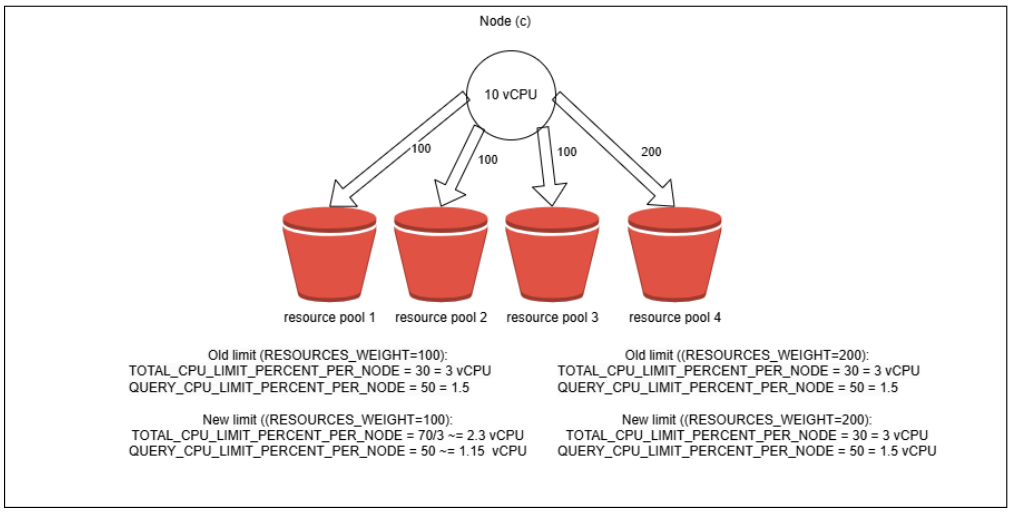
When a query enters a resource pool for which `DATABASE_LOAD_CPU_THRESHOLD` is set, 10% of the available CPU on the node is immediately reserved, based on the assumption that the query will require at least that amount of resources. Then, every 10 seconds, the consumed resources are recalculated across the entire database, allowing the initial 10% estimate to be refined. This means that if more than 10 queries arrive at a cluster node simultaneously, no more than 10 queries will be started for execution, and the rest will wait for the actual CPU consumption to be refined.

As with `CONCURRENT_QUERY_LIMIT`, when the specified load threshold is exceeded, queries are sent to the waiting queue.

Resource allocation according to RESOURCES_WEIGHT







The `RESOURCES_WEIGHT` parameter only takes effect in case of oversubscription and when there is more than one resource pool in the system. In the current implementation, `RESOURCES_WEIGHT` only affects the allocation of `vCPU` resources. When queries appear in a resource pool, it starts participating in resource allocation. For this, the pools recalculate their limits according to the Max-min fairness algorithm. The actual resource redistribution is performed on each compute node individually, as shown in the figure above.

Suppose we have a node in the system with available. The following limits are set:

- ,
- .

In this case, the resource pool will have a limit of per node and per query in this pool (figure *a*). If there are 4 such pools in the system and they all try to use maximum resources, this would amount to , which exceeds the limit of available resources on the node (). In this case, `RESOURCES_WEIGHT` takes effect, and each pool will be allocated (figure *b*).

If you need to increase the allocated resources for a specific pool, you can change its weight, for example, to 200. Then this pool will get , and the remaining pools will equally share the remaining , which amounts to per pool (figure *c*).

Warning

The current resource allocation algorithm may be changed in the future without backward compatibility support.

Default resource pool

Even if no resource pool has been created, the system always has a resource pool `default` that cannot be deleted. Any query executing in the system always belongs to some pool — there is no situation where a query is not attached to any resource pool. By default, the settings of the resource pool `default` are as follows:

```
CREATE RESOURCE POOL default WITH (
    CONCURRENT_QUERY_LIMIT=-1,
    QUEUE_SIZE=-1,
    DATABASE_LOAD_CPU_THRESHOLD=-1,
    RESOURCES_WEIGHT=-1,
    TOTAL_MEMORY_LIMIT_PERCENT_PER_NODE=-1,
    QUERY_CPU_LIMIT_PERCENT_PER_NODE=-1,
    TOTAL_CPU_LIMIT_PERCENT_PER_NODE=-1
)
```

This means that no restrictions are applied in the resource pool `default`: it operates independently of other pools and has no limits on consumed resources. In the resource pool `default`, you can change parameters using the ALTER RESOURCE POOL query,

except for the parameters `CONCURRENT_QUERY_LIMIT`, `DATABASE_LOAD_CPU_THRESHOLD`, and `QUEUE_SIZE`. This restriction is intentionally introduced to minimize risks associated with incorrect configuration of the default resource pool.

Resource pool ACL management

Permissions for creating, modifying, and deleting a resource pool

To create, modify, or delete a resource pool, you need to grant access permissions according to the permissions described in the reference for CREATE RESOURCE POOL. For example, to create resource pools, you need to have the `CREATE TABLE` permission on the directory `.metadata/workload_manager/pools`, which can be granted with a query of the following form:

```
GRANT CREATE TABLE ON `.metadata/workload_manager/pools` TO user1;
```

Permissions for executing a query in a resource pool

To execute a query in a pool, the user must have the access permission `SELECT` on this pool. Example of granting permissions:

```
GRANT SELECT
    ON `.metadata/workload_manager/pools/olap`
    TO `user1@domain`;
```

Creating a resource pool classifier

Resource pool classifiers allow you to specify rules by which queries will be distributed among resource pools. The example below shows a resource pool classifier that sends queries from all users to a resource pool named `olap`:

```
CREATE RESOURCE POOL CLASSIFIER olap_classifier
WITH (
    RESOURCE_POOL = 'olap',
    MEMBER_NAME = 'all-users@well-known'
);
```

- `RESOURCE_POOL` — the name of the resource pool to which the query satisfying the requirements specified in the resource pool classifier will be sent.
- `MEMBER_NAME` — a user group or user whose queries will be sent to the specified resource pool.

ACL management of the resource pool classifier

Resource pool classifiers are global for the entire database and apply to all users. To create, delete, or modify a resource pool classifier, you need to have the access permission `USE` on the entire database, which can be granted with a query of the form:

```
GRANT USE ON `/my_db` TO user1;
```

Warning

To use the classifier, the user must have access to the resource pool to which this classifier refers. If there is no such access, the classifier is skipped and the next one is checked. In future versions, this behavior may change, so do not use the lack of permissions on the pool as a control mechanism for which classifier will be triggered.

Order of resource pool classifier selection in case of conflicts

```
CREATE RESOURCE POOL CLASSIFIER olap1_classifier
WITH (
    RESOURCE_POOL = 'olap1',
    MEMBER_NAME = 'user1@domain'
);

CREATE RESOURCE POOL CLASSIFIER olap2_classifier
WITH (
    RESOURCE_POOL = 'olap2',
    MEMBER_NAME = 'user1@domain'
);
```

Suppose there are two resource pool classifiers with conflicting conditions, and user `user1@domain` matches both resource pools: `olap1` and `olap2`. If no classifier existed in the system before, then `olap1` is set for `RANK=1000`, and `olap2` for `RANK=2000`. Resource pool classifiers with a lower `RANK` value have higher priority. In this example, since `olap1` has a higher priority `RANK` than `olap2`, it will be selected.

You can also set `RANK` for resource pool classifiers when creating them using the CREATE RESOURCE POOL CLASSIFIER syntax construct, or change `RANK` for existing resource pool classifiers using ALTER RESOURCE POOL CLASSIFIER.

The system cannot have two classifiers with the same `RANK` value, which makes it possible to uniquely determine which resource pool will be selected in case of conflicting conditions.

Example of a priority resource pool

Consider an example of resource allocation between an analytics team and a conditional CEO. It is important for the CEO to have priority over the computing resources used for analytical tasks, but it is useful to allow the analytics team to utilize more cluster resources during periods when the CEO is not using them. The configuration for this scenario might look as follows:

```
CREATE RESOURCE POOL olap WITH (
    CONCURRENT_QUERY_LIMIT=20,
    QUEUE_SIZE=100,
```

```
    DATABASE_LOAD_CPU_THRESHOLD=80,
    RESOURCES_WEIGHT=20,
    QUERY_CPU_LIMIT_PERCENT_PER_NODE=80,
    TOTAL_CPU_LIMIT_PERCENT_PER_NODE=100
);

CREATE RESOURCE POOL the_ceo WITH (
    CONCURRENT_QUERY_LIMIT=20,
    QUEUE_SIZE=100,
    RESOURCES_WEIGHT=100,
    QUERY_CPU_LIMIT_PERCENT_PER_NODE=100,
    TOTAL_CPU_LIMIT_PERCENT_PER_NODE=100
);
```

In the example above, two resource pools are created: `olap` for the analytics team and `the_ceo` for the CEO.

- **Resource pool `olap`**:
  - Has a weight of 20.
  - The limit on running queries when the database is overloaded is 80% of available resources.
- **Resource pool `the_ceo`**:
  - Has a higher weight — 80.
  - Has no limit on running queries when overloaded.

A weight of 80 for `the_ceo` effectively means that when competing for resources, pool `the_ceo` will receive 4 times more priority than pool `olap`. If queries arrive in both pools, the system will recalculate the limits, and for `olap` the `TOTAL_CPU_LIMIT_PERCENT_PER_NODE` limit will be reduced to 20%, while for `the_ceo` it will be increased to 80%. This resource redistribution is based on weights, as described above.

Explicit selection of a resource pool for a query

If necessary, the user can explicitly specify in which pool a given query should be executed. Currently, this can be done as follows:

- **YDB UI** — in the query launch settings window `Query execution settings` via the `Resource pool` parameter.
- **YDB CLI** — in the `ydb sql` command with the `--resource-pool` parameter, for example, `ydb sql --resource-pool my_pool -s "SELECT 1"`.
- **YDB CLI (interactive mode)** — using the command `SET resource_pool = my_pool`, where `my_pool` is the name of the resource pool.
- **YDB CPP SDK** — in the query launch settings via the ResourcePool parameter.
- **YDB GO SDK** — in the query launch settings `ExecuteOption` via the WithResourcePool call.
- **YDB Java SDK** — in the query launch settings `ExecuteQuerySettings` via the withResourcePool call, available since version 2.3.12.
- **YDB Python SDK** — via the pool_id parameter of the `QuerySession.execute()`, `QueryTxContext.execute()` and `QuerySessionPool.execute_with_retries()` methods, available since version 3.31.2.

Diagnostics

Query plan

Detailed information about query plans can be found on the query plan structure page. To get information about the resource pool used, you need to run a command that retrieves statistics in `json-unicode` format. Example command:

```
ydb -p <profile_name> sql -s 'select 1' --stats full --format json-unicode
```

In the body of the query plan obtained using the command above, you can find useful attributes for diagnosing resource pool operation. Example of such information:

```
"Node Type" : "Query",
"Stats" : {
  "TotalDurationUs": 28795,
  "ProcessCpuTimeUs": 45,
  "Compilation": {
    "FromCache": false,
    "CpuTimeUs": 7280,
    "DurationUs": 21700
  },
  "ResourcePoolId": "default",
  "QueuedTimeUs": 0
},
"PlanNodeType" : "Query"
```

Useful attributes:

- `TotalDurationUs` — total query execution time, including queue wait time.
- `ResourcePoolId` — name of the resource pool to which the query was assigned.
- `QueuedTimeUs` — total query queue wait time.

Status of a running query

Information about how a query is processed in Workload Manager can be obtained from the system view `.sys/query_sessions`. This view contains the following fields:

- `WmPoolId` `(Utf8)` - ID of the pool in which the query is executed.
- `WmState` `(Utf8)` - Query status in WM.
- `WmEnterTime` `(Timestamp)` - Time when the query transitioned to PENDING or DELAYED status.
- `WmExitTime` `(Timestamp)` - Time when the query was submitted for execution.

Possible values of the WmState field:

- `NONE` - Not being processed.
- `PENDING` - Being processed (during classification/routing).
- `DELAYED` - In queue.
- `EXITED` - Submitted for execution.

The following query outputs information about all active queries in the system:

```
select
    Query,          -- Запрос
    WmPoolId,       -- Идентификатор пула
    WmState,        -- Статус запроса в WM
    WmEnterTime,    -- Время, когда запрос перешел в статус PENDING или DELAYED
    WmExitTime      -- Время, когда запрос передан на выполнение
from `.sys/query_sessions`
where State = 'EXECUTING'
```

Metrics

Information about resource pool metrics can be found in the metrics reference.

System views

Information about system views related to resource pools and resource pool classifiers can be found on the Resource pool information page.

See also

- CREATE RESOURCE POOL
- ALTER RESOURCE POOL
- DROP RESOURCE POOL
- CREATE RESOURCE POOL CLASSIFIER
- ALTER RESOURCE POOL CLASSIFIER
- DROP RESOURCE POOL CLASSIFIER

Shared topic consumer: architecture and limitations

Shared consumer is a reading model for a topic where a consumer is assigned one or more messages, rather than an entire partition. This allows multiple consumers to process messages from one partition in parallel and use topics as message queues. A typical scenario is message exchange between microservices; reading via the Amazon SQS API works through a shared consumer.

For a general description of the model, consumer settings, DLQ policies, and message ordering, see the Shared consumer section. This article describes the internal architecture of the mechanism: how the server tracks processing state, distributes read requests across partitions, what limitations to consider when designing high-load queues, and what resource consumption the inflight creates.

Implementation specifics

For each pair of 'shared consumer — topic partition', the server maintains its own message processing state. The state consists of a continuous block of messages from the topic partition — the inflight. The first message of the inflight is a message that has not yet been processed, is being processed, or is waiting to be moved to the DLQ. If the first message is successfully processed, the beginning of the block moves to the next message. Messages for processing are only delivered from the messages in the inflight.

Up to 120,000 messages can be in the inflight of one partition. This means that if a large number of consumers (more than 120,000) read from one partition, no more than 120,000 consumers will receive messages for processing, even if there are more messages in the partition. Increasing the number of partitions in a topic increases the total inflight size — the total inflight of a topic equals the number of partitions multiplied by 120,000 messages. If you need to process a large number of messages simultaneously, you should increase the number of partitions in the topic.

If a topic consists of several partitions, read requests are evenly distributed across all partitions. It may happen that a request hits a partition that has no messages. In this case, the consumer will receive a response that there are no messages to process. If a partition has no messages to process for a long time, it is excluded from read request distribution. The partition will return to distribution as soon as messages to process appear in it.

For shared consumers with message ordering preservation, this means that if a large number of messages from one message-group-id (more than 120,000 messages) are written consecutively to a partition, the entire inflight will be occupied by messages from one message-group-id, and messages from other groups will not be delivered until messages from other groups appear in the inflight.

Fair queues

When delivering messages for reading, the server takes into account the `message-group-id` of messages in the inflight. Trying to deliver messages from different groups, the server distributes the load among writers more evenly.

This allows smoothing out peaks from individual writers: if one writer sends many more messages to the queue than others, processing of messages from 'small' writers gets priority and is not blocked by the flow from a single source.

For FIFO queues and shared consumers with message ordering preservation, the order of messages within one `message-group-id` is guaranteed: the consumer always receives the group's messages in the order they were written.

Resource consumption

The inflight state consumes RAM and disk resources. For each message in the inflight, the server stores about 32 bytes of metadata — this is not the message body, but metadata about its processing status.

RAM

The entire inflight is stored in the server's memory. To estimate memory consumption, it is convenient to calculate per million messages: **1 million messages in the inflight ≈ 32 MB** of RAM.

For example, a topic with 10 partitions and one shared consumer with a fully filled inflight (120,000 messages per partition) holds up to 1.2 million messages in inflight — about **38 MB** of memory for the service state alone. For one "shared consumer — partition" pair at maximum inflight, it accounts for about **3.8 MB**. If a topic has multiple shared consumers, memory consumption multiplies by their number.

When designing, consider:

- the more partitions and consumers, the higher the memory consumption ceiling
- the longer messages are being processed (long message processing time, slow consumers), the longer they stay in inflight and occupy memory
- increasing the number of partitions expands parallelism, but at the same time increases the total inflight volume.

Disk

Information about message status is also stored on disk — from **32 bytes per message**. Storage consists of two parts:

1. **Snapshot** — the full state of all messages in inflight is periodically written.
2. **Change log** — the server processes pending read requests in batches and, when processing each batch, appends changes to message statuses to the log.

Consequently, high read frequency and a large inflight increase the volume of disk writes. When planning cluster capacity, allocate disk resources alongside RAM — especially for topics with a large number of partitions, multiple shared consumers, and long message processing times.

For Analysts

This section provides examples and recommendations for handling analytical (OLAP) scenarios in YDB.

This section includes the following materials

- Dataset import

Related sections

- Questions and answers about analytics in YDB
- Column-Oriented Tables
- Aggregate functions

Quickstart: Loading TPC-H data from Object Storage and executing analytical queries

This quickstart describes the process of loading the TPC-H test dataset (scale factor 10, 10 GB) into YDB from external S3-compatible storage and executing analytical queries.

After completing the steps, the following will be created in the database:

- External data source `EXTERNAL DATA SOURCE`.
- External tables `EXTERNAL TABLE` for Parquet file schemas.
- YDB column-oriented tables optimized for analytical queries.

Actions include:

- Loading data from Object Storage into YDB tables using `UPSERT INTO... SELECT FROM`.
- Executing a test query (TPC-H Q0) for verification.

Prerequisites

- An existing YDB database.
- A tool for executing YQL/SQL queries (YDB UI or CLI).

Creating an external data source

In this step, an `EXTERNAL DATA SOURCE` object will be created, which is a named reference to external data storage. It contains information about the data location and access method.

`EXTERNAL DATA SOURCE` is an entity in YDB that describes a connection to external storage, such as S3-compatible Object Storage.

```
CREATE EXTERNAL DATA SOURCE IF NOT EXISTS `external/tpc` WITH (
    SOURCE_TYPE="ObjectStorage",
    LOCATION="https://storage.yandexcloud.net/tpc/",
    AUTH_METHOD="NONE"
);
```

Parameters:

- `SOURCE_TYPE="ObjectStorage"`: storage type (S3-compatible).
- `LOCATION="https://storage.yandexcloud.net/tpc/"`: bucket URL.
- `AUTH_METHOD="NONE"`: public access without authentication.

Creating external tables

In this step, schemas (columns and types) are defined for Parquet files in the external source. This allows accessing data through SQL queries in YDB, linking the logical structure with physical files.

`EXTERNAL TABLE` is a "virtual" table that describes the structure of the data located in an external source. This links the logical table schema with the physical files.

```
CREATE EXTERNAL TABLE IF NOT EXISTS `external/tpch/s10/customer` (
    c_acctbal Double NOT NULL,
    c_mktsegment String NOT NULL,
    c_phone String NOT NULL,
    c_nationkey Int32 NOT NULL,
    c_custkey Int32 NOT NULL,
    c_name String NOT NULL,
    c_comment String NOT NULL,
    c_address String NOT NULL
) WITH (
    DATA_SOURCE="external/tpc",
    LOCATION="/h/s10/parquet/customer/",
    FORMAT="parquet"
);
```

Parameters:

- `DATA_SOURCE="external/tpc"`: reference to the source.
- `LOCATION="/h/s10/parquet/customer/"`: path to files.
- `FORMAT="parquet"`: data format.

Create the remaining tables

```
CREATE EXTERNAL TABLE IF NOT EXISTS `external/tpch/s10/customer` (
    c_acctbal Double NOT NULL,
    c_mktsegment String NOT NULL,
    c_phone String NOT NULL,
    c_nationkey Int32 NOT NULL,
    c_custkey Int32 NOT NULL,
    c_name String NOT NULL,
    c_comment String NOT NULL,
    c_address String NOT NULL
) WITH (
    DATA_SOURCE="external/tpc",
    LOCATION="/h/s10/parquet/customer/",
    FORMAT="parquet"
);

CREATE EXTERNAL TABLE IF NOT EXISTS `external/tpch/s10/lineitem` (
```

```
    l_orderkey Int64 NOT NULL,
    l_partkey Int32 NOT NULL,
    l_suppkey Int32 NOT NULL,
    l_linenumber Int32 NOT NULL,
    l_quantity Double NOT NULL,
    l_extendedprice Double NOT NULL,
    l_discount Double NOT NULL,
    l_tax Double NOT NULL,
    l_returnflag String NOT NULL,
    l_linestatus String NOT NULL,
    l_shipdate Date NOT NULL,
    l_commitdate Date NOT NULL,
    l_receiptdate Date NOT NULL,
    l_shipinstruct String NOT NULL,
    l_shipmode String NOT NULL,
    l_comment String NOT NULL
) WITH (
    DATA_SOURCE="external/tpc",
    LOCATION="/h/s10/parquet/lineitem/",
    FORMAT="parquet"
);

CREATE EXTERNAL TABLE IF NOT EXISTS `external/tpch/s10/nation` (
    n_nationkey Int32 NOT NULL,
    n_name String NOT NULL,
    n_regionkey Int32 NOT NULL,
    n_comment String NOT NULL
) WITH (
    DATA_SOURCE="external/tpc",
    LOCATION="/h/s10/parquet/nation/",
    FORMAT="parquet"
);

CREATE EXTERNAL TABLE IF NOT EXISTS `external/tpch/s10/orders` (
    o_orderkey Int64 NOT NULL,
    o_custkey Int32 NOT NULL,
    o_orderstatus String NOT NULL,
    o_totalprice Double NOT NULL,
    o_orderdate Date NOT NULL,
    o_orderpriority String NOT NULL,
    o_clerk String NOT NULL,
    o_shippriority Int32 NOT NULL,
    o_comment String NOT NULL
) WITH (
    DATA_SOURCE="external/tpc",
    LOCATION="/h/s10/parquet/orders/",
    FORMAT="parquet"
);

CREATE EXTERNAL TABLE IF NOT EXISTS `external/tpch/s10/part` (
    p_partkey Int32 NOT NULL,
    p_name String NOT NULL,
    p_mfgr String NOT NULL,
    p_brand String NOT NULL,
    p_type String NOT NULL,
    p_size Int32 NOT NULL,
    p_container String NOT NULL,
    p_retailprice Double NOT NULL,
    p_comment String NOT NULL
) WITH (
    DATA_SOURCE="external/tpc",
    LOCATION="/h/s10/parquet/part/",
    FORMAT="parquet"
);

CREATE EXTERNAL TABLE IF NOT EXISTS `external/tpch/s10/partsupp` (
    ps_partkey Int32 NOT NULL,
    ps_suppkey Int32 NOT NULL,
    ps_availqty Int32 NOT NULL,
    ps_supplycost Double NOT NULL,
    ps_comment String NOT NULL
) WITH (
    DATA_SOURCE="external/tpc",
    LOCATION="/h/s10/parquet/partsupp/",
    FORMAT="parquet"
);

CREATE EXTERNAL TABLE IF NOT EXISTS `external/tpch/s10/region` (
    r_regionkey Int32 NOT NULL,
    r_name String NOT NULL,
    r_comment String NOT NULL
) WITH (
    DATA_SOURCE="external/tpc",
    LOCATION="/h/s10/parquet/region/",
    FORMAT="parquet"
);

CREATE EXTERNAL TABLE IF NOT EXISTS `external/tpch/s10/supplier` (
    s_suppkey Int32 NOT NULL,
    s_name String NOT NULL,
    s_address String NOT NULL,
```

```
    s_nationkey Int32 NOT NULL,
    s_phone String NOT NULL,
    s_acctbal Double NOT NULL,
    s_comment String NOT NULL
) WITH (
    DATA_SOURCE="external/tpc",
    LOCATION="/h/s10/parquet/supplier/",
    FORMAT="parquet"
);
```

Creating column-oriented tables

In this step, internal YDB tables are created for data storage. They use the column-oriented format and partitioning, which provides high performance for analytical queries through parallel processing.

```
CREATE TABLE IF NOT EXISTS `tpch/s10/customer` (
    c_acctbal Double NOT NULL,
    c_mktsegment String NOT NULL,
    c_phone String NOT NULL,
    c_nationkey Int32 NOT NULL,
    c_custkey Int32 NOT NULL,
    c_name String NOT NULL,
    c_comment String NOT NULL,
    c_address String NOT NULL,
    PRIMARY KEY (c_custkey)
)
PARTITION BY HASH (c_custkey)
WITH (
    STORE = COLUMN,
    PARTITION_COUNT = 64
);
```

Parameters:

- `PRIMARY KEY (c_custkey)` : table primary key.
- `PARTITION BY HASH (c_custkey)` : hash partitioning for parallelism.
- `STORE = COLUMN` : column-oriented data storage format for accelerating analytical queries.
- `PARTITION_COUNT = 64` : number of partitions for data distribution.

Create the remaining tables

```
CREATE TABLE IF NOT EXISTS `tpch/s10/customer` (
    c_acctbal Double NOT NULL,
    c_mktsegment String NOT NULL,
    c_phone String NOT NULL,
    c_nationkey Int32 NOT NULL,
    c_custkey Int32 NOT NULL,
    c_name String NOT NULL,
    c_comment String NOT NULL,
    c_address String NOT NULL,
    PRIMARY KEY (c_custkey)
)
PARTITION BY HASH (c_custkey)
WITH (
    STORE = COLUMN,
    PARTITION_COUNT = 64
);

CREATE TABLE IF NOT EXISTS `tpch/s10/lineitem` (
    l_orderkey Int64 NOT NULL,
    l_partkey Int32 NOT NULL,
    l_suppkey Int32 NOT NULL,
    l_linenumber Int32 NOT NULL,
    l_quantity Double NOT NULL,
    l_extendedprice Double NOT NULL,
    l_discount Double NOT NULL,
    l_tax Double NOT NULL,
    l_returnflag String NOT NULL,
    l_linestatus String NOT NULL,
    l_shipdate Date NOT NULL,
    l_commitdate Date NOT NULL,
    l_receiptdate Date NOT NULL,
    l_shipinstruct String NOT NULL,
    l_shipmode String NOT NULL,
    l_comment String NOT NULL,
    PRIMARY KEY (l_orderkey, l_linenumber)
)
PARTITION BY HASH (l_orderkey)
WITH (
    STORE = COLUMN,
    PARTITION_COUNT = 64
);

CREATE TABLE IF NOT EXISTS `tpch/s10/nation` (
    n_nationkey Int32 NOT NULL,
    n_name String NOT NULL,
    n_regionkey Int32 NOT NULL,
    n_comment String NOT NULL,
```

```
    PRIMARY KEY (n_nationkey)
)
PARTITION BY HASH (n_nationkey)
WITH (
    STORE = COLUMN,
    PARTITION_COUNT = 64
);

CREATE TABLE IF NOT EXISTS `tpch/s10/orders` (
    o_orderkey Int64 NOT NULL,
    o_custkey Int32 NOT NULL,
    o_orderstatus String NOT NULL,
    o_totalprice Double NOT NULL,
    o_orderdate Date NOT NULL,
    o_orderpriority String NOT NULL,
    o_clerk String NOT NULL,
    o_shippriority Int32 NOT NULL,
    o_comment String NOT NULL,
    PRIMARY KEY (o_orderkey)
)
PARTITION BY HASH (o_orderkey)
WITH (
    STORE = COLUMN,
    PARTITION_COUNT = 64
);

CREATE TABLE IF NOT EXISTS `tpch/s10/part` (
    p_partkey Int32 NOT NULL,
    p_name String NOT NULL,
    p_mfgr String NOT NULL,
    p_brand String NOT NULL,
    p_type String NOT NULL,
    p_size Int32 NOT NULL,
    p_container String NOT NULL,
    p_retailprice Double NOT NULL,
    p_comment String NOT NULL,
    PRIMARY KEY (p_partkey)
)
PARTITION BY HASH (p_partkey)
WITH (
    STORE = COLUMN,
    PARTITION_COUNT = 64
);

CREATE TABLE IF NOT EXISTS `tpch/s10/partsupp` (
    ps_partkey Int32 NOT NULL,
    ps_suppkey Int32 NOT NULL,
    ps_availqty Int32 NOT NULL,
    ps_supplycost Double NOT NULL,
    ps_comment String NOT NULL,
    PRIMARY KEY (ps_partkey, ps_suppkey)
)
PARTITION BY HASH (ps_partkey, ps_suppkey)
WITH (
    STORE = COLUMN,
    PARTITION_COUNT = 64
);

CREATE TABLE IF NOT EXISTS `tpch/s10/region` (
    r_regionkey Int32 NOT NULL,
    r_name String NOT NULL,
    r_comment String NOT NULL,
    PRIMARY KEY (r_regionkey)
)
PARTITION BY HASH (r_regionkey)
WITH (
    STORE = COLUMN,
    PARTITION_COUNT = 64
);

CREATE TABLE IF NOT EXISTS `tpch/s10/supplier` (
    s_suppkey Int32 NOT NULL,
    s_name String NOT NULL,
    s_address String NOT NULL,
    s_nationkey Int32 NOT NULL,
    s_phone String NOT NULL,
    s_acctbal Double NOT NULL,
    s_comment String NOT NULL,
    PRIMARY KEY (s_suppkey)
)
PARTITION BY HASH (s_suppkey)
WITH (
    STORE = COLUMN,
    PARTITION_COUNT = 64
);
```

Loading data into column-oriented tables

In this step, data is copied from the external storage into internal YDB tables. This provides the local storage for fast analytics. The process may take time (for 10 GB — on the order of minutes, depending on resources).

```
UPSERT INTO `tpch/s10/customer` SELECT * FROM `external/tpch/s10/customer`;
UPSERT INTO `tpch/s10/lineitem` SELECT * FROM `external/tpch/s10/lineitem`;
UPSERT INTO `tpch/s10/nation`   SELECT * FROM `external/tpch/s10/nation`;
UPSERT INTO `tpch/s10/orders`   SELECT * FROM `external/tpch/s10/orders`;
UPSERT INTO `tpch/s10/part`     SELECT * FROM `external/tpch/s10/part`;
UPSERT INTO `tpch/s10/partsupp` SELECT * FROM `external/tpch/s10/partsupp`;
UPSERT INTO `tpch/s10/region`   SELECT * FROM `external/tpch/s10/region`;
UPSERT INTO `tpch/s10/supplier` SELECT * FROM `external/tpch/s10/supplier`;
```

Executing a test analytical query

In this step, the TPC-H Q0 test query is executed to verify the correctness of data loading and performance. The query aggregates metrics from the `lineitem` table (about 6 million rows), with grouping and subsequent data sorting, demonstrating the efficiency of column-oriented tables.

```
SELECT
    l_returnflag,
    l_linestatus,
    SUM(l_quantity) AS sum_qty,
    SUM(l_extendedprice) AS sum_base_price,
    SUM(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
    SUM(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
    AVG(l_quantity) AS avg_qty,
    AVG(l_extendedprice) AS avg_price,
    AVG(l_discount) AS avg_disc,
    COUNT(*) AS count_order
FROM
    `tpch/s10/lineitem`
WHERE
    l_shipdate <= Date('1998-12-01') - Interval("P90D")
GROUP BY
    l_returnflag,
    l_linestatus
ORDER BY
    l_returnflag,
    l_linestatus;
```

Next steps

- Execute other queries from the TPC-H suite on the loaded tables using YDB CLI and TPC-H workload testing.
- Integration with BI systems for data visualization and analysis.

Overview

These pages describe popular datasets that you can load into YDB to familiarize yourself with the database's functionality and test various use cases.

Prerequisites

To load the datasets, you will need:

1. Installed YDB CLI
2. [Optional] Configured connection profile to YDB to avoid specifying connection parameters with every command

General Information on Data Loading

YDB supports importing data from CSV files using the command `ydb import file csv`. Example command:

```
ydb import file csv --header --null-value "" --path <table_path> <file>.csv
```

Where:

- `--header` indicates that the first row of the file contains the column names, and the actual data starts from the second row;
- `--null-value ""` specifies that an empty string in the CSV will be interpreted as a null value during data import into the table.

A table must already exist in YDB for data import. The primary way to create a table is by executing the `CREATE TABLE` YQL query. Instead of writing the query manually, you can try running the import command from a file, as shown in any example in this section, without creating the table first. In this case, the CLI will suggest a `CREATE TABLE` query, which you can use as a base, edit if necessary, and execute.

To import data into YDB, a table must be pre-created. Typically, a table is created using the YQL `CREATE TABLE` query. However, instead of crafting such a query manually, you can initiate the import command `ydb import file csv` as shown in the import examples in this section. If the table doesn't exist, the CLI will automatically suggest a `CREATE TABLE` query that you can use to create the table.

> **Selecting a Primary Key**
>
> YDB requires a primary key for the table. It significantly speeds up data loading and processing, and it also allows for deduplication: rows with identical values in the primary key columns replace each other.
>
> If the imported dataset doesn't have suitable columns for a primary key, we add a new column with row numbers and use it as the primary key, as each row number is unique within the file.

Features and Limitations

When woimporting data to YDB, consider the following points:

1. **Column Names**: Column names should not contain spaces or special characters.
2. **Data Types**:
   - Date/time strings with timezone (e.g., "2019-11-01 00:00:00 UTC") will be imported as Text type.
   - The Bool type is not supported as a column type; use Text or Int64 instead.

Available Datasets

- Chess Position Evaluations - Stockfish engine chess position evaluations
- Video Game Sales - video game sales data
- E-Commerce Behavior Data - user behavior data from an online store
- COVID-19 Open Research Dataset - open research dataset on COVID-19
- Netflix Movies and TV Shows - data on Netflix movies and shows
- Animal Crossing New Horizons Catalog - item catalog from the game

Overview

These pages describe popular datasets that you can load into YDB to familiarize yourself with the database's functionality and test various use cases.

Prerequisites

To load the datasets, you will need:

1. Installed YDB CLI
2. [Optional] Configured connection profile to YDB to avoid specifying connection parameters with every command

General Information on Data Loading

YDB supports importing data from CSV files using the command `ydb import file csv`. Example command:

```
ydb import file csv --header --null-value "" --path <table_path> <file>.csv
```

Where:

- `--header` indicates that the first row of the file contains the column names, and the actual data starts from the second row;
- `--null-value ""` specifies that an empty string in the CSV will be interpreted as a null value during data import into the table.

A table must already exist in YDB for data import. The primary way to create a table is by executing the `CREATE TABLE` YQL query. Instead of writing the query manually, you can try running the import command from a file, as shown in any example in this section, without creating the table first. In this case, the CLI will suggest a `CREATE TABLE` query, which you can use as a base, edit if necessary, and execute.

To import data into YDB, a table must be pre-created. Typically, a table is created using the YQL `CREATE TABLE` query. However, instead of crafting such a query manually, you can initiate the import command `ydb import file csv` as shown in the import examples in this section. If the table doesn't exist, the CLI will automatically suggest a `CREATE TABLE` query that you can use to create the table.

> **Selecting a Primary Key**
>
> YDB requires a primary key for the table. It significantly speeds up data loading and processing, and it also allows for deduplication: rows with identical values in the primary key columns replace each other.
>
> If the imported dataset doesn't have suitable columns for a primary key, we add a new column with row numbers and use it as the primary key, as each row number is unique within the file.

Features and Limitations

When woimporting data to YDB, consider the following points:

1. **Column Names**: Column names should not contain spaces or special characters.
2. **Data Types**:
   - Date/time strings with timezone (e.g., "2019-11-01 00:00:00 UTC") will be imported as Text type.
   - The Bool type is not supported as a column type; use Text or Int64 instead.

Available Datasets

- Chess Position Evaluations - Stockfish engine chess position evaluations
- Video Game Sales - video game sales data
- E-Commerce Behavior Data - user behavior data from an online store
- COVID-19 Open Research Dataset - open research dataset on COVID-19
- Netflix Movies and TV Shows - data on Netflix movies and shows
- Animal Crossing New Horizons Catalog - item catalog from the game

Chess Position Evaluations

> **Note**
>
> This page is part of the Dataset Import section, which includes examples of loading popular datasets into YDB. Before starting, please review the general information on requirements and the import process.

The dataset includes 513 million chess position evaluations performed by the Stockfish engine for analysis on the Lichess platform.

Source: Kaggle - Chess Position Evaluations

Size: 59.66 GB

Loading Example

1. Download the `evals.csv` file from Kaggle.
2. Create a table in YDB using one of the following methods:

   YDB UI

   For more information on YDB UI.

   ```
   CREATE TABLE `evals` (
       `fen` Text NOT NULL,
       `line` Text NOT NULL,
       `depth` Uint64,
       `knodes` Uint64,
       `cp` Double,
       `mate` Double,
       PRIMARY KEY (`fen`, `line`)
   )
   WITH (
       STORE = COLUMN,
       UNIFORM_PARTITIONS = 50
   );
   ```

   YDB CLI

   ```
   ydb sql -s \
   'CREATE TABLE `evals` (
       `fen` Text NOT NULL,
       `line` Text NOT NULL,
       `depth` Uint64,
       `knodes` Uint64,
       `cp` Double,
       `mate` Double,
       PRIMARY KEY (`fen`, `line`)
   )
   WITH (
       STORE = COLUMN,
       UNIFORM_PARTITIONS = 50
   );'
   ```

3. Execute the import command:

   ```
   ydb import file csv --header --null-value "" --path evals evals.csv
   ```

Analytical Query Example

Identify positions with the highest number of moves analyzed by the Stockfish engine:

YDB UI

```
SELECT
    fen,
    MAX(depth) AS max_depth,
    SUM(knodes) AS total_knodes
FROM evals
GROUP BY fen
ORDER BY max_depth DESC
LIMIT 10;
```

YDB CLI

```
ydb sql -s \
'SELECT
    fen,
    MAX(depth) AS max_depth,
    SUM(knodes) AS total_knodes
FROM evals
GROUP BY fen
ORDER BY max_depth DESC
LIMIT 10;'
```

This query performs the following actions:

- Finds positions (represented in FEN format) with the maximum analysis depth.
- Sums the number of analyzed nodes (knodes) for each position.
- Sorts results by maximum analysis depth in descending order.
- Outputs the top 10 positions with the highest analysis depth.

Video Game Sales

> **Note**
>
> This page is part of the Dataset Import section, which includes examples of loading popular datasets into YDB. Before starting, please review the general information on requirements and the import process.

Data on video game sales.

Source: Kaggle - Video Game Sales

Size: 1.36 MB

Loading Example

1. Download and unzip the `vgsales.csv` file from Kaggle.
2. Create a table in YDB using one of the following methods:

   YDB UI

   For more information on YDB UI.

   ```
   CREATE TABLE `vgsales` (
       `Rank` Uint64 NOT NULL,
       `Name` Text NOT NULL,
       `Platform` Text NOT NULL,
       `Year` Text NOT NULL,
       `Genre` Text NOT NULL,
       `Publisher` Text NOT NULL,
       `NA_Sales` Double NOT NULL,
       `EU_Sales` Double NOT NULL,
       `JP_Sales` Double NOT NULL,
       `Other_Sales` Double NOT NULL,
       `Global_Sales` Double NOT NULL,
       PRIMARY KEY (`Rank`)
   )
   WITH (
       STORE = COLUMN
   );
   ```

   YDB CLI

   ```
   ydb sql -s \
   'CREATE TABLE `vgsales` (
       `Rank` Uint64 NOT NULL,
       `Name` Text NOT NULL,
       `Platform` Text NOT NULL,
       `Year` Text NOT NULL,
       `Genre` Text NOT NULL,
       `Publisher` Text NOT NULL,
       `NA_Sales` Double NOT NULL,
       `EU_Sales` Double NOT NULL,
       `JP_Sales` Double NOT NULL,
       `Other_Sales` Double NOT NULL,
       `Global_Sales` Double NOT NULL,
       PRIMARY KEY (`Rank`)
   )
   WITH (
       STORE = COLUMN
   );'
   ```

3. Execute the import command:

   ```
   ydb import file csv --header --null-value "" --path vgsales vgsales.csv
   ```

Analytical Query Example

To identify the publisher with the highest average game sales in North America, execute the query:

YDB UI

```
SELECT
    Publisher,
    AVG(NA_Sales) AS average_na_sales
FROM vgsales
GROUP BY Publisher
ORDER BY average_na_sales DESC
LIMIT 1;
```

YDB CLI

```
ydb sql -s \
'SELECT
    Publisher,
    AVG(NA_Sales) AS average_na_sales
```

```
FROM vgsales
GROUP BY Publisher
ORDER BY average_na_sales DESC
LIMIT 1;'
```

Result:

```
┌─────────────┬──────────────────┐
│ Publisher │ average_na_sales │
├─────────────┼──────────────────┤
│ "Palcom"  │ 3.38             │
└─────────────┴──────────────────┘
```

This query helps find the publisher with the greatest success in North America by average sales.

E-Commerce Behavior Data

> **Note**
>
> This page is part of the Dataset Import section, which includes examples of loading popular datasets into YDB. Before starting, please review the general information on requirements and the import process.

User behavior data from a multi-category online store.

Source: Kaggle - E-commerce behavior data

Size: 9 GB

Loading Example

1. Download and unzip the `2019-Nov.csv` file from Kaggle.
2. The dataset includes completely identical rows. Since YDB requires unique primary key values, add a new column named `row_id` to the file, where the key value will be equal to the row number in the original file. This prevents the removal of duplicate data. This operation can be carried out using the awk command:

```
awk 'NR==1 {print "row_id," \$0; next} {print NR-1 "," \$0}' 2019-Nov.csv > temp.csv && mv temp.csv 2019
-Nov.csv
```

3. Create a table in YDB using one of the following methods:

   YDB UI

   For more information on YDB UI.

```
CREATE TABLE `ecommerce_table` (
    `row_id` Uint64 NOT NULL,
    `event_time` Text NOT NULL,
    `event_type` Text NOT NULL,
    `product_id` Uint64 NOT NULL,
    `category_id` Uint64,
    `category_code` Text,
    `brand` Text,
    `price` Double NOT NULL,
    `user_id` Uint64 NOT NULL,
    `user_session` Text NOT NULL,
    PRIMARY KEY (`row_id`)
)
WITH (
    STORE = COLUMN,
    UNIFORM_PARTITIONS = 50
);
```

   YDB CLI

```
ydb sql -s \
'CREATE TABLE `ecommerce_table` (
    `row_id` Uint64 NOT NULL,
    `event_time` Text NOT NULL,
    `event_type` Text NOT NULL,
    `product_id` Uint64 NOT NULL,
    `category_id` Uint64,
    `category_code` Text,
    `brand` Text,
    `price` Double NOT NULL,
    `user_id` Uint64 NOT NULL,
    `user_session` Text NOT NULL,
    PRIMARY KEY (`row_id`)
)
WITH (
    STORE = COLUMN,
    UNIFORM_PARTITIONS = 50
);'
```

4. Execute the import command:

```
ydb import file csv --header --null-value "" --path ecommerce_table 2019-Nov.csv
```

Analytical Query Example

Identify the most popular product categories on November 1, 2019:

YDB UI

```
SELECT
    category_code,
    COUNT(*) AS view_count
FROM ecommerce_table
WHERE
    SUBSTRING(CAST(event_time AS String), 0, 10) = '2019-11-01'
    AND event_type = 'view'
```

```
GROUP BY category_code
ORDER BY view_count DESC
LIMIT 10;
```

YDB CLI

```
ydb sql -s \
'SELECT
    category_code,
    COUNT(*) AS view_count
FROM ecommerce_table
WHERE
    SUBSTRING(CAST(event_time AS String), 0, 10) = "2019-11-01"
    AND event_type = "view"
GROUP BY category_code
ORDER BY view_count DESC
LIMIT 10;'
```

Result:

```
┌──────────────────────────────────────┬────────────┐
| category_code                        | view_count |
├──────────────────────────────────────┼────────────┤
| null                                 | 453024     |
├──────────────────────────────────────┼────────────┤
| "electronics.smartphone"             | 360650     |
├──────────────────────────────────────┼────────────┤
| "electronics.clocks"                 | 43581      |
├──────────────────────────────────────┼────────────┤
| "computers.notebook"                 | 40878      |
├──────────────────────────────────────┼────────────┤
| "electronics.video.tv"               | 40383      |
├──────────────────────────────────────┼────────────┤
| "electronics.audio.headphone"        | 37489      |
├──────────────────────────────────────┼────────────┤
| "apparel.shoes"                      | 31013      |
├──────────────────────────────────────┼────────────┤
| "appliances.kitchen.washer"          | 28028      |
├──────────────────────────────────────┼────────────┤
| "appliances.kitchen.refrigerators"   | 27808      |
├──────────────────────────────────────┼────────────┤
| "appliances.environment.vacuum"      | 26477      |
└──────────────────────────────────────┴────────────┘
```

COVID-19 Open Research Dataset

> **Note**
>
> This page is part of the Dataset Import section, which includes examples of loading popular datasets into YDB. Before starting, please review the general information on requirements and the import process.

An open dataset of COVID-19 research.

Source: Kaggle - COVID-19 Open Research Dataset Challenge

Size: 1.65 GB (metadata.csv file)

Loading Example

1. Download and unzip the `metadata.csv` file from Kaggle.
2. The dataset includes completely identical rows. Since YDB requires unique primary key values, add a new column named `row_id` to the file, where the key value will be equal to the row number in the original file. This prevents the removal of duplicate data. This operation can be carried out using the awk command:

   ```
   awk 'NR==1 {print "row_id," \$0; next} {print NR-1 "," \$0}' metadata.csv > temp.csv && mv temp.csv meta
   data.csv
   ```

3. Create a table in YDB using one of the following methods:

   YDB UI

   For more information on YDB UI.

   ```
   CREATE TABLE `covid_research` (
       `row_id` Uint64 NOT NULL,
       `cord_uid` Text NOT NULL,
       `sha` Text NOT NULL,
       `source_x` Text NOT NULL,
       `title` Text NOT NULL,
       `doi` Text NOT NULL,
       `pmcid` Text NOT NULL,
       `pubmed_id` Text NOT NULL,
       `license` Text NOT NULL,
       `abstract` Text NOT NULL,
       `publish_time` Text NOT NULL,
       `authors` Text NOT NULL,
       `journal` Text NOT NULL,
       `mag_id` Text,
       `who_covidence_id` Text,
       `arxiv_id` Text,
       `pdf_json_files` Text NOT NULL,
       `pmc_json_files` Text NOT NULL,
       `url` Text NOT NULL,
       `s2_id` Uint64,
       PRIMARY KEY (`row_id`)
   )
   WITH (
       STORE = COLUMN
   );
   ```

   YDB CLI

   ```
   ydb sql -s \
   'CREATE TABLE `covid_research` (
       `row_id` Uint64 NOT NULL,
       `cord_uid` Text NOT NULL,
       `sha` Text NOT NULL,
       `source_x` Text NOT NULL,
       `title` Text NOT NULL,
       `doi` Text NOT NULL,
       `pmcid` Text NOT NULL,
       `pubmed_id` Text NOT NULL,
       `license` Text NOT NULL,
       `abstract` Text NOT NULL,
       `publish_time` Text NOT NULL,
       `authors` Text NOT NULL,
       `journal` Text NOT NULL,
       `mag_id` Text,
       `who_covidence_id` Text,
       `arxiv_id` Text,
       `pdf_json_files` Text NOT NULL,
       `pmc_json_files` Text NOT NULL,
       `url` Text NOT NULL,
       `s2_id` Uint64,
       PRIMARY KEY (`row_id`)
   )
   WITH (
       STORE = COLUMN
   );'
   ```

4. Execute the import command:

```
ydb import file csv --header --null-value "" --path covid_research metadata.csv
```

Analytical Query Example

Run a query to determine the journals with the highest number of publications:

YDB UI

```
SELECT
    journal,
    COUNT(*) AS publication_count
FROM covid_research
WHERE journal IS NOT NULL AND journal != ''
GROUP BY journal
ORDER BY publication_count DESC
LIMIT 10;
```

YDB CLI

```
ydb sql -s \
'SELECT
    journal,
    COUNT(*) AS publication_count
FROM covid_research
WHERE journal IS NOT NULL AND journal != ""
GROUP BY journal
ORDER BY publication_count DESC
LIMIT 10;'
```

Result:

```
┌──────────────────────────────────┬───────────────────┐
| journal                          | publication_count |
├──────────────────────────────────┼───────────────────┤
| "PLoS One"                       | 9953              |
├──────────────────────────────────┼───────────────────┤
| "bioRxiv"                        | 8961              |
├──────────────────────────────────┼───────────────────┤
| "Int J Environ Res Public Health" | 8201              |
├──────────────────────────────────┼───────────────────┤
| "BMJ"                            | 6928              |
├──────────────────────────────────┼───────────────────┤
| "Sci Rep"                        | 5935              |
├──────────────────────────────────┼───────────────────┤
| "Cureus"                         | 4212              |
├──────────────────────────────────┼───────────────────┤
| "Reactions Weekly"               | 3891              |
├──────────────────────────────────┼───────────────────┤
| "Front Psychol"                  | 3541              |
├──────────────────────────────────┼───────────────────┤
| "BMJ Open"                       | 3515              |
├──────────────────────────────────┼───────────────────┤
| "Front Immunol"                  | 3442              |
└──────────────────────────────────┴───────────────────┘
```

Netflix Movies and TV Shows

> **Note**
>
> This page is part of the Dataset Import section, which includes examples of loading popular datasets into YDB. Before starting, please review the general information on requirements and the import process.

Data on movies and TV shows available on Netflix.

Source: Kaggle - Netflix Movies and TV Shows

Size: 3.4 MB

Loading Example

1. Download and unzip the `netflix_titles.csv` file from Kaggle.
2. Create a table in YDB using one of the following methods:

   YDB UI

   For more information on YDB UI.

   ```
   CREATE TABLE `netflix` (
       `show_id` Text NOT NULL,
       `type` Text NOT NULL,
       `title` Text NOT NULL,
       `director` Text NOT NULL,
       `cast` Text,
       `country` Text NOT NULL,
       `date_added` Text NOT NULL,
       `release_year` Uint64 NOT NULL,
       `rating` Text NOT NULL,
       `duration` Text NOT NULL,
       `listed_in` Text NOT NULL,
       `description` Text NOT NULL,
       PRIMARY KEY (`show_id`)
   )
   WITH (
       STORE = COLUMN
   );
   ```

   YDB CLI

   ```
   ydb sql -s \
   'CREATE TABLE `netflix` (
       `show_id` Text NOT NULL,
       `type` Text NOT NULL,
       `title` Text NOT NULL,
       `director` Text NOT NULL,
       `cast` Text,
       `country` Text NOT NULL,
       `date_added` Text NOT NULL,
       `release_year` Uint64 NOT NULL,
       `rating` Text NOT NULL,
       `duration` Text NOT NULL,
       `listed_in` Text NOT NULL,
       `description` Text NOT NULL,
       PRIMARY KEY (`show_id`)
   )
   WITH (
       STORE = COLUMN
   );'
   ```

3. Execute the import command:

   ```
   ydb import file csv --header --null-value "" --path netflix netflix_titles.csv
   ```

Analytical Query Example

Identify the top three countries with the most content added to Netflix in 2020:

YDB UI

```
SELECT
    country,
    COUNT(*) AS count
FROM netflix
WHERE
    CAST(SUBSTRING(CAST(date_added AS String), 7, 4) AS Int32) = 2020
    AND date_added IS NOT NULL
GROUP BY country
ORDER BY count DESC
LIMIT 3;
```

YDB CLI

```
ydb sql -s \
'SELECT
    country,
    COUNT(*) AS count
FROM netflix
WHERE
    CAST(SUBSTRING(CAST(date_added AS String), 7, 4) AS Int32) = 2020
    AND date_added IS NOT NULL
GROUP BY country
ORDER BY count DESC
LIMIT 3;'
```

Result:

```
┌─────────────────┬───────┐
| country         | count |
├─────────────────┼───────┤
| "United States" | 22    |
├─────────────────┼───────┤
| ""              | 7     |
├─────────────────┼───────┤
| "Canada"        | 3     |
└─────────────────┴───────┘
```

Animal Crossing New Horizons Catalog

> **Note**
>
> This page is part of the Dataset Import section, which includes examples of loading popular datasets into YDB. Before starting, please review the general information on requirements and the import process.

A catalog of items from the popular game Animal Crossing: New Horizons.

Source: Kaggle - Animal Crossing New Horizons Catalog

Size: 51 KB

Loading Example

1. Download and unzip the `accessories.csv` file from Kaggle.
2. This file includes a BOM (Byte Order Mark). However, the import command does not support files with a BOM. To resolve this, remove the BOM bytes from the beginning of the file by executing the following command:

   ```
   sed -i '1s/^\xEF\xBB\xBF//' accessories.csv
   ```

3. The column names in the file contain spaces, which are incompatible with YDB since YDB does not support spaces in column names. Replace spaces in the column names with underscores, for example, by executing the following command:

   ```
   sed -i '1s/ /_/g' accessories.csv
   ```

4. Create a table in YDB using one of the following methods:

   YDB UI

   For more information on YDB UI.

   ```
   CREATE TABLE `accessories` (
       `Name` Text NOT NULL,
       `Variation` Text NOT NULL,
       `DIY` Text NOT NULL,
       `Buy` Text NOT NULL,
       `Sell` Uint64 NOT NULL,
       `Color_1` Text NOT NULL,
       `Color_2` Text NOT NULL,
       `Size` Text NOT NULL,
       `Miles_Price` Text NOT NULL,
       `Source` Text NOT NULL,
       `Source_Notes` Text NOT NULL,
       `Seasonal_Availability` Text NOT NULL,
       `Mannequin_Piece` Text NOT NULL,
       `Version` Text NOT NULL,
       `Style` Text NOT NULL,
       `Label_Themes` Text NOT NULL,
       `Type` Text NOT NULL,
       `Villager_Equippable` Text NOT NULL,
       `Catalog` Text NOT NULL,
       `Filename` Text NOT NULL,
       `Internal_ID` Uint64 NOT NULL,
       `Unique_Entry_ID` Text NOT NULL,
       PRIMARY KEY (`Unique_Entry_ID`)
   )
   WITH (
       STORE = COLUMN
   );
   ```

   YDB CLI

   ```
   ydb sql -s \
   'CREATE TABLE `accessories` (
       `Name` Text NOT NULL,
       `Variation` Text NOT NULL,
       `DIY` Text NOT NULL,
       `Buy` Text NOT NULL,
       `Sell` Uint64 NOT NULL,
       `Color_1` Text NOT NULL,
       `Color_2` Text NOT NULL,
       `Size` Text NOT NULL,
       `Miles_Price` Text NOT NULL,
       `Source` Text NOT NULL,
       `Source_Notes` Text NOT NULL,
       `Seasonal_Availability` Text NOT NULL,
       `Mannequin_Piece` Text NOT NULL,
       `Version` Text NOT NULL,
       `Style` Text NOT NULL,
       `Label_Themes` Text NOT NULL,
       `Type` Text NOT NULL,
       `Villager_Equippable` Text NOT NULL,
       `Catalog` Text NOT NULL,
       `Filename` Text NOT NULL,
       `Internal_ID` Uint64 NOT NULL,
       `Unique_Entry_ID` Text NOT NULL,
   ```

```
    PRIMARY KEY (`Unique_Entry_ID`)
)
WITH (
    STORE = COLUMN
);'
```

5. Execute the import command:

```
ydb import file csv --header --path accessories accessories.csv
```

Analytical Query Example

Identify the top five most popular primary colors of accessories:

YDB UI

```
SELECT
    Color_1,
    COUNT(*) AS color_count
FROM accessories
GROUP BY Color_1
ORDER BY color_count DESC
LIMIT 5;
```

YDB CLI

```
ydb sql -s \
'SELECT
    Color_1,
    COUNT(*) AS color_count
FROM accessories
GROUP BY Color_1
ORDER BY color_count DESC
LIMIT 5;'
```

Result:

```
┌──────────┬─────────────┐
| Color_1  | color_count |
├──────────┼─────────────┤
| "Black"  | 31          |
├──────────┼─────────────┤
| "Green"  | 27          |
├──────────┼─────────────┤
| "Pink"   | 20          |
├──────────┼─────────────┤
| "Red"    | 20          |
├──────────┼─────────────┤
| "Yellow" | 19          |
└──────────┴─────────────┘
```

Working with slowly changing dimensions

This section contains practical guides for implementing slowly changing dimensions (Slowly Changing Dimensions, SCD) — a popular approach to managing historical data in analytical stores.

The section describes the SCD1 and SCD2 variants:

- SCD Type 1: Old attribute values are replaced with new ones, keeping only the current state of data.
- SCD Type 2: The full history of changes is preserved by adding new records for each new attribute version.

Features of SCD1

Slowly changing dimensions (type 1) or SCD1 (Type 1) is an approach where when an attribute of the dimension changes, the old value is replaced with a new one. Only the current state of data is stored. This approach is used when:

- historical information is not required;
- it is important to have only current data;
- it is required to minimize the size of the data store;
- it is required to have a simple data structure for analytics.

Features of SCD2 and append-only approach

Slowly changing dimensions (type 2) or SCD2 (Type 2) is an approach where when an attribute of the dimension changes, a new record is created, and the old one is marked as not current. Thus, the history of changes is preserved. This approach is used when:

- it is required to track the history of data changes;
- it is required to perform data analysis with regard to time periods;
- it is important to preserve the audit trail of changes;
- it is required to have the ability to restore the state of data at a specific point in time.

Available guides

The section discusses different technical ways to implement these mechanisms:

- Using the combination of Change Data Capture (CDC) and Transfer for automatic streaming replication of changes from source tables.
  - Using the TRANSFER mechanism to implement SCD1 based on data from CDC sources in YDB
  - Using the TRANSFER mechanism to implement SCD2 based on data from CDC sources in YDB
- Using periodic YQL queries that process batches of changes from the intermediate table and merge them into the main SCD table.
  - Using the merge process to implement SCD2 in YDB

Working with slowly changing dimensions

This section contains practical guides for implementing slowly changing dimensions (Slowly Changing Dimensions, SCD) — a popular approach to managing historical data in analytical stores.

The section describes the SCD1 and SCD2 variants:

- SCD Type 1: Old attribute values are replaced with new ones, keeping only the current state of data.
- SCD Type 2: The full history of changes is preserved by adding new records for each new attribute version.

Features of SCD1

Slowly changing dimensions (type 1) or SCD1 (Type 1) is an approach where when an attribute of the dimension changes, the old value is replaced with a new one. Only the current state of data is stored. This approach is used when:

- historical information is not required;
- it is important to have only current data;
- it is required to minimize the size of the data store;
- it is required to have a simple data structure for analytics.

Features of SCD2 and append-only approach

Slowly changing dimensions (type 2) or SCD2 (Type 2) is an approach where when an attribute of the dimension changes, a new record is created, and the old one is marked as not current. Thus, the history of changes is preserved. This approach is used when:

- it is required to track the history of data changes;
- it is required to perform data analysis with regard to time periods;
- it is important to preserve the audit trail of changes;
- it is required to have the ability to restore the state of data at a specific point in time.

Available guides

The section discusses different technical ways to implement these mechanisms:

- Using the combination of Change Data Capture (CDC) and Transfer for automatic streaming replication of changes from source tables.
  - Using the TRANSFER mechanism to implement SCD1 based on data from CDC sources in YDB
  - Using the TRANSFER mechanism to implement SCD2 based on data from CDC sources in YDB
- Using periodic YQL queries that process batches of changes from the intermediate table and merge them into the main SCD table.
  - Using the merge process to implement SCD2 in YDB

Using the TRANSFER mechanism to implement SCD1 based on data from CDC sources in YDB

This article describes the implementation of the Slowly Changing Dimensions Type 1 (SCD1) pattern in YDB.

Used tools

For delivering data to the SCD1 table, the following combination of functional capabilities of YDB will be used in this article:

1. The source table will be row-oriented for operational transactional changes;
2. The target table will be column-oriented for efficient execution of analytical queries;
3. The subscription to changes in the source table will be implemented through the Change Data Capture (CDC) mechanism;
4. The buffering of changes will be handled by an implicitly created CDC topic;
5. The automatic transfer of data from the CDC topic to the target table will be handled by transfer.

> **Warning**
>
> Transfer cannot delete rows from tables, it can only perform `UPSERT` in the target table.
> Therefore, using Transfer, only SCD1-soft can be implemented: deletion is marked with a flag and the moment of deletion is saved. If physical data cleaning is needed, it can be performed through an explicit command `DELETE`.

Create a source data table that will generate CDC events

```
CREATE TABLE source_customers (
    id Utf8 NOT NULL,
    attribute1 Utf8,
    attribute2 Utf8,
    change_time Timestamp,
    PRIMARY KEY (id)
);

ALTER TABLE `source_customers` ADD CHANGEFEED `updates` WITH (
  FORMAT = 'DEBEZIUM_JSON',
  MODE = 'NEW_AND_OLD_IMAGES'
);
```

Create a target data table in SCD1 format

```
CREATE TABLE dimension_scd1 (
    id Utf8 NOT NULL,
    attribute1 Utf8,
    attribute2 Utf8,
    is_deleted Uint8,
    last_update_timestamp Timestamp,
    PRIMARY KEY (id)
)
PARTITION BY HASH(id)
WITH (
  STORE=COLUMN
)
```

Create a transfer and lambda function to process CDC data

Features of processing CDC data in Debezium format:

- The message format contains the `payload.before` and `payload.after` fields with the state before and after the change.
- The `payload.op` field indicates the type of operation: "c" (create), "u" (update), "d" (delete).
- The `payload.ts_ms` field contains the event timestamp in milliseconds.
- When deleting (`op = "d"`), the data is in the `before` field, and when creating/updating - in the `after` field. If deletion of a non-existent row is performed, a message with the operation type `op = "d"` will be formed, but with empty `before` and `after` fields.

```
$transformation_lambda = ($msg) -> {
    $cdc_data = CAST($msg._data AS Json);

    -- Define the type of operation
    $operation = Json::ConvertToString($cdc_data.payload.op);
    $is_deleted = $operation == "d";

    -- Get data depending on the type of operation
    $data = IF($is_deleted, $cdc_data.payload.before, $cdc_data.payload.after);

    -- If the data has not come (the DELETE command has been executed on a non-existent key, then the record
will be ignored)
    return IF($data IS NOT NULL,
        <|
            id: Unwrap(CAST(Json::ConvertToString($data.id) AS Utf8)),
            attribute1: CAST(Json::ConvertToString($data.attribute1) AS Utf8),
            attribute2: CAST(Json::ConvertToString($data.attribute2) AS Utf8),
            is_deleted: CAST($is_deleted AS Uint8),
            last_update_timestamp: DateTime::FromMilliseconds(Json::ConvertToUint64($cdc_data.payload.source.
ts_ms))
      |>
    , NULL);
};
```

```
-- In this example, a high frequency of updates to the target table is configured.
-- This is done solely for clarity. For production scenarios, larger values should be set
CREATE TRANSFER dimension_scd1_cdc_transfer
  FROM `source_customers/updates` TO dimension_scd1 USING $transformation_lambda
  WITH (
    FLUSH_INTERVAL=Interval("PT1S")
  );
```

Note

The `id` column in the receiving table `dimension_scd1` is declared as `Utf8 NOT NULL`, while in Json CDC, data that cannot be converted to a string may be passed, i.e., the result of converting data from Json may be a `NULL` value. The `Unwrap` function guarantees that after its execution, there cannot be a `NULL` value or a runtime error. This ensures that the result of executing the lambda function will either be a fully correct data structure, or a runtime error.

Demonstration of work

To demonstrate the work with CDC data, data will be written to the source table in YDB, which will generate CDC events:

Inserting a new record

```
INSERT INTO source_customers (id, attribute1, attribute2)
VALUES ('CUSTOMER_1001', 'John Doe', 'New York');
```

This action will create a CDC event with the operation type `op = "c"` approximately of the following form:

```
{
  "payload":{
    "op":"c",
    "source":{
      "txId":18446744073709551615,
      "connector":"ydb",
      "version":"1.0.0",
      "step":1755868513260,
      "ts_ms":1755868513219,
      "snapshot":false
    },
    "after":{
      "attribute1":"John Doe",
      "id":"CUSTOMER_1001",
      "attribute2":"New York"
    }
  }
}
```

The content of the `dimension_scd1` table after executing the command above will be as follows:

id	attribute1	attribute2	is_deleted	last_update_timestamp
CUSTOMER_1001	John Doe	New York	0	2025-08-22T13:15:13.219000Z

Updating an existing record

```
UPSERT INTO source_customers (id, attribute1, attribute2)
VALUES ('CUSTOMER_1001', 'John Doe', 'Los Angeles');
```

This action will create a CDC event with the operation type `op = "u"` approximately of the following form:

```
{
  "payload":{
      "op":"u",
      "source":{
        "txId":18446744073709551615,
        "connector":"ydb",
        "version":"1.0.0",
        "step":1755868795050,
        "ts_ms":1755868795719,
        "snapshot":false
    },
    "after":{
      "attribute1":"John Doe",
      "id":"CUSTOMER_1001",
      "attribute2":"Los Angeles"
    },
    "before":{
      "attribute1":"John Doe",
      "id":"CUSTOMER_1001",
      "attribute2":"New York"
    }
  }
}
```

The content of the `dimension_scd1` table after executing the command above will be as follows:

id	attribute1	attribute2	is_deleted	last_update_timestamp
CUSTOMER_1001	John Doe	Los Angeles	0	2025-08-22T13:19:55.719000Z

Deleting a record

```
DELETE FROM source_customers WHERE id = 'CUSTOMER_1001';
```

This action will create a CDC event with the operation type `op = "d"` approximately of the following form:


```
{
  "payload":{
    "op":"d",
    "source":{
      "txId":18446744073709551615,
      "connector":"ydb",
      "version":"1.0.0",
      "step":1755868931000,
      "ts_ms":1755868931109,
      "snapshot":false
    },
    "before":{
      "attribute1":"John Doe",
      "id":"CUSTOMER_1001",
      "attribute2":"Los Angeles"
    }
  }
}
```


The content of the `dimension_scd1` table after executing the command above will be as follows:

id	attribute1	attribute2	is_deleted	last_update_timestamp
CUSTOMER_1001	John Doe	Los Angeles	1	2025-08-22T13:22:11.109000Z

Example of deleting data from the target table

To delete data from the target table, you can use an explicit `DELETE` command:

```
DELETE from dimension_scd1 WHERE is_deleted != 0ut
```

Using the TRANSFER mechanism to implement SCD2 based on data from CDC sources in YDB

This article describes the implementation of the Slowly Changing Dimensions Type 2 (SCD2) pattern in YDB.

Used tools

For delivering data to the SCD2 table, the following combination of functional capabilities of YDB will be used in this article:

1. The source table will be row-oriented for operational transactional changes;
2. The target table will be column-oriented for efficient execution of analytical queries;
3. The subscription to changes in the source table will be implemented through the Change Data Capture (CDC) mechanism;
4. The buffering of changes will be handled by an implicitly created CDC topic;
5. The automatic transfer of data from the CDC topic to the target table will be handled by transfer.

Create a source data table that will generate CDC events

```
CREATE TABLE source_customers (
    id Utf8 NOT NULL,
    attribute1 Utf8,
    attribute2 Utf8,
    change_time Timestamp,
    PRIMARY KEY (id)
);

ALTER TABLE `source_customers` ADD CHANGEFEED `updates` WITH (
  FORMAT = 'DEBEZIUM_JSON',
  MODE = 'NEW_AND_OLD_IMAGES'
);
```

Create a target data table in SCD2 format

```
  CREATE TABLE dimension_scd2 (
    id Utf8 NOT NULL,
    attribute1 Utf8,
    attribute2 Utf8,
    valid_from Timestamp NOT NULL,
    deleted_at Timestamp ,
    PRIMARY KEY (valid_from, id)
  )
  PARTITION BY HASH(valid_from, id)
  WITH (
    STORE=COLUMN
  )
```

Create a transfer and lambda function to process CDC data

Features of processing CDC data in Debezium format:

- When creating a record (`op = "c"`), data is taken from the `after` field, and the value of the moment of change of the record from the `changed_at` field is saved in the `valid_from` field.
- When updating a record (`op = "u"`), data is also taken from the `after` field, and the value of the moment of change of the record from the `changed_at` field is saved in the `valid_from` field.
- When deleting a record (`op = "d"`), data is taken from the `before` field, and the system moment of time of receiving information about the deletion of the record is set in the `deleted_at` field.

```
$transformation_lambda = ($msg) -> {
$cdc_data = CAST($msg._data AS Json);

-- Define the type of operation
$operation = Json::ConvertToString($cdc_data.payload.op);
$is_deleted = $operation == "d";

-- Get data depending on the type of operation
$data = IF($is_deleted, $cdc_data.payload.before, $cdc_data.payload.after);

$result =  IF($data is not NULL, <|
        id: Unwrap(CAST(Json::ConvertToString($data.id) AS Utf8)),
        attribute1: CAST(Json::ConvertToString($data.attribute1) AS Utf8),
        attribute2: CAST(Json::ConvertToString($data.attribute2) AS Utf8),
        deleted_at: IF($is_deleted, DateTime::FromMilliseconds(Json::ConvertToUint64($cdc_data.payload.sourc
e.ts_ms)), NULL),
        valid_from : Unwrap(cast(datetime::MakeDatetime(datetime::ParseIso8601(Json::ConvertToString($data.ch
ange_time))) as timestamp)),
    |>, NULL);

return $result;
};

-- In this example, a high frequency of updates to the target table is configured.
-- This is done solely for clarity. For production scenarios, larger values should be set
  CREATE TRANSFER dimension_scd2_cdc_transfer
    FROM `source_customers/updates` TO dimension_scd2 USING $transformation_lambda
    WITH (
      FLUSH_INTERVAL=Interval("PT1S")
    );
```

Note

The `id` column in the receiving table `dimension_scd2` is declared as `Utf8 NOT NULL`, while in Json CDC, data that cannot be converted to a string may be passed, i.e., the result of converting data from Json may be a `NULL` value. The `Unwrap` function guarantees that after its execution, there cannot be a `NULL` value or a runtime error. This ensures that the result of executing the lambda function will either be a fully correct data structure, or a runtime error.

Demonstration of work

To demonstrate the work with CDC data, data will be written to the source table in YDB, which will generate CDC events:

Inserting a new record

```
INSERT INTO source_customers (id, attribute1, attribute2, change_time)
VALUES ('CUSTOMER_1001', 'John Doe', 'Los Angeles', CurrentUtcTimestamp());
```

This action will create a CDC event with the operation type `op = "c"` approximately of the following form:


```
{
  "payload":{
    "op":"c",
    "source":{
      "txId":18446744073709551615,
      "connector":"ydb",
      "version":"1.0.0",
      "step":1755883683690,
      "ts_ms":1755883683652,
      "snapshot":false
    },
    "after":{
      "attribute1":"John Doe",
      "change_time":"2025-08-22T17:28:03.648313Z",
      "id":"CUSTOMER_1001",
      "attribute2":"Los Angeles"
    }
  }
}
```


The content of the `dimension_scd2` table after executing the command above will be as follows:

id	attribute1	attribute2	deleted_at	valid_from
CUSTOMER_1001	John Doe	New York	NULL	2025-08-22T17:28:03.648313Z

Updating an existing record

```
UPSERT INTO source_customers (id, attribute1, attribute2, change_time)
VALUES ('CUSTOMER_1001', 'John Doe 2', 'Los Angeles 2', CurrentUtcTimestamp());
```

This action will create a CDC event with the operation type `op = "u"` approximately of the following form:


```
{
  "payload":{
    "op":"u",
    "source":{
      "txId":18446744073709551615,
      "connector":"ydb",
      "version":"1.0.0",
      "step":1755883278000,
      "ts_ms":1755883278357,
      "snapshot":false
    },
    "after":{
      "attribute1":"John Doe 2",
      "change_time":"2025-08-22T17:31:18.357503Z",
      "id":"CUSTOMER_1001",
      "attribute2":"Los Angeles 2"
    },
    "before":{
      "attribute1":"John Doe 2",
      "change_time":"2025-08-22T17:28:03.648313Z",
      "id":"CUSTOMER_1001",
      "attribute2":"Los Angeles 2"
    }
  }
}
```


The content of the `dimension_scd2` table after executing the command above will be as follows:

id	attribute1	attribute2	deleted_at	valid_from
CUSTOMER_1001	John Doe	New York	NULL	2025-08-22T17:28:03.648313Z
CUSTOMER_1001	John Doe 2	New York 2	NULL	2025-08-22T17:31:18.357503Z

Deleting a record

```
DELETE FROM source_customers WHERE id = 'CUSTOMER_1001';
```

This action will create a CDC event with the operation type `op = "d"` approximately of the following form:

```
{
  "payload":{
    "op":"d",
    "source":{
      "txId":18446744073709551615,
      "connector":"ydb",
      "version":"1.0.0",
      "step":1755883827000,
      "ts_ms":1755883827948,
      "snapshot":false
    },
    "before":{
      "attribute1":"John Doe",
      "change_time":"2025-08-22T17:38:03.648313Z",
      "id":"CUSTOMER_1001",
      "attribute2":"Los Angeles"
    }
  }
}
```

The content of the `dimension_scd2` table after executing the command above will be as follows:

id	attribute1	attribute2	deleted_at	valid_from
CUSTOMER_1001	John Doe	New York	NULL	2025-08-22T17:28:03.648313Z
CUSTOMER_1001	John Doe 2	New York 2	NULL	2025-08-22T17:31:18.357503Z
CUSTOMER_1001	John Doe 2	New York 2	2025-08-22T17:38:03.648313Z	2025-08-22T17:38:03.648313Z

Example query to retrieve current data

To retrieve data from SCD2 append-only tables, you can use the following query, which fetches data as of `2025-08-22 19:11:30`:

```
DECLARE $as_of AS Timestamp;
$as_of = Timestamp("2025-08-22T19:11:30.000000Z");

SELECT
  id,
  data.attribute1 AS attribute1,
  data.attribute2 AS attribute2,
  data.valid_from AS valid_from,
  data.deleted_at AS deleted_at
FROM (
  SELECT
    id,
    MAX_BY(TableRow(), valid_from) AS data
  FROM dimension_scd2
  WHERE
      valid_from <= $as_of
      AND (deleted_at IS NULL OR deleted_at > $as_of)  -- The record has not yet been deleted as of time X
  GROUP BY id
)
```

Using the merge process to implement SCD2 in YDB

This article describes the implementation of the Slowly Changing Dimensions Type 2 (SCD2) pattern in YDB using the change merge process.

Tools used

To populate data into the SCD2 table in this article, the following combination of features available in YDB will be used:

1. The source table `dimension_scd_changes`, containing information about attributes, their values, and moments of data changes.
2. The target table `dimension_scd2_final` to store the resulting data.
3. An external application should periodically run a query to merge changes accumulated in the `dimension_scd_changes` table into the `dimension_scd2_final` table.
4. To deliver data from staging tables into SCD2 format, it is convenient to use the built-in YDB transfer mechanism.

> **Note**
>
> The tables `dimension_scd_changes`, `dimension_scd2_final` are given for illustration. For actual queries, you will need to adapt the structure of tables and their attributes accordingly.

Creating a table to receive all Changes `dimension_scd_changes`

```
CREATE TABLE dimension_scd_changes (
    id Utf8 NOT NULL, -- Business key
    attribute1 Utf8,  -- Data attribute
    attribute2 Utf8,  -- Data attribute
    change_time Timestamp NOT NULL, -- Moment of data change
    operation Utf8, -- Type of data change
    PRIMARY KEY (change_time, id)
)
PARTITION BY HASH(change_time, id)
WITH (
    STORE=COLUMN
)
```

Table field descriptions:

- `id` — business key of the record;
- `attribute1`, `attribute2` — dimension attributes;
- `change_time` — timestamp when the data changed;
- `operation` — type of change: `CREATE`, `UPDATE`, or `DELETE`.

The primary key is created as `PRIMARY KEY (change_time, id)` because there may be multiple changes on the same business key, and it is important to store all those changes. For details on choosing the primary key and partitioning key, see the documentation: choosing a primary key and choosing a partition key.

Creating the final SCD2 table `dimension_scd2_final`

```
CREATE TABLE dimension_scd2_final (
    id Utf8 NOT NULL, -- Business key
    attribute1 Utf8,  -- Data attribute
    attribute2 Utf8,  -- Data attribute
    valid_from Timestamp NOT NULL, -- Timestamp from which the data is valid
    valid_to Timestamp,            -- Timestamp until which the data is valid.
                                   -- If the data is currently valid, then valid_to is NULL
    is_current Uint8,              -- Indicator that the data is currently valid.
    is_deleted Uint8,              -- Indicator that the data was deleted. If deleted, is_current = FALSE
    PRIMARY KEY (valid_from, id)
)
PARTITION BY HASH(valid_from, id)
WITH(
    STORE=COLUMN
)
```

Table field descriptions:

- `id` — business key of the record;
- `attribute1`, `attribute2` — dimension attributes;
- `valid_from` — timestamp from which the record becomes valid;
- `valid_to` — timestamp until which the record was valid, or `NULL` for current records;
- `is_current` — flag indicating if the record is current (1 for current, 0 for historical);
- `is_deleted` — flag indicating if the record was deleted (1 if deleted, 0 otherwise).

The primary key is created as `PRIMARY KEY (valid_from, id)` for the same reasons as above: there may be multiple changes per key, and all those changes must be stored.

Loading data into the change table

You can use any available method to load data into the change table, as well as the automatic delivery of changes using the transfer mechanism.

Example query for explicitly loading changes:

```
UPSERT INTO dimension_scd_changes (id, attribute1, attribute2, change_time, operation)
VALUES ('CUSTOMER_1001', 'John Doe', 'Los Angeles', Unwrap(CAST('2025-08-22T17:00:00Z' as Timestamp)), 'CREAT
E');

UPSERT INTO dimension_scd_changes (id, attribute1, attribute2, change_time, operation)
VALUES ('CUSTOMER_1002', 'John Doe', 'New York', Unwrap(CAST('2025-08-22T17:00:00Z' as Timestamp)), 'CREAT
E');

UPSERT INTO dimension_scd_changes (id, attribute1, attribute2, change_time, operation)
VALUES ('CUSTOMER_1001', 'John Doe', 'San Francisco', Unwrap(CAST('2025-08-22T19:00:00Z' as Timestamp)), 'UPD
ATE');

UPSERT INTO dimension_scd_changes (id, attribute1, attribute2, change_time, operation)
VALUES ('CUSTOMER_1002', 'John Doe', 'New York', Unwrap(CAST('2025-08-22T21:00:00Z' as Timestamp)), 'DELET
E');
```

Query for applying changes in SCD2 format

A special query is used to transform data from the changes table into SCD2 format and load it into the final table. This query should be run regularly—with whatever frequency you want to update data in the final table. For automatic execution, you can use the YDB integration with Apache Airflow™:

```
-- Step 1: Read all new events from the `dimension_scd_changes` table.
-- This named expression ($changes) serves as the initial dataset for all subsequent processing in this run.
$changes = (
    SELECT
        id,
        attribute1,
        attribute2,
        change_time,
        String::AsciiToUpper(operation) AS op
    FROM dimension_scd_changes
);

-- Step 2: Filter events, keeping only those not yet present in the target table.
-- The goal of this step is to ensure read-level idempotency, avoiding reprocessing
-- of already loaded data in case of a script failure and restart.
$unprocessed_data = (
    SELECT
        chg.id AS id,
        chg.attribute1 AS attribute1,
        chg.attribute2 AS attribute2,
        chg.change_time AS change_time,
        chg.op AS op
    FROM $changes AS chg
    LEFT JOIN dimension_scd2_final AS scd
        ON chg.id = scd.id AND chg.change_time = scd.valid_from -- Look for records by each entity (id) and c
hange timestamp
    WHERE scd.id IS NULL -- Exclude rows already transferred to the dimension_scd2_final table earlier
);

-- Step 3: Find active records (`is_current=1`) in the target table that have received updates.
-- Generate "closing" versions for them, setting `valid_to` to the timestamp
-- of the earliest change from the new batch ($unprocessed_data).
$close_open_intervals = (
    SELECT
        target.id AS id,
        target.attribute1 as attribute1,
        target.attribute2 as attribute2,
        target.valid_from as valid_from,
        0ut AS is_current, -- The closing record is no longer current
        unprocessed_data.change_time AS valid_to,
        target.is_deleted as is_deleted
    FROM dimension_scd2_final AS target
    INNER JOIN (
        SELECT
            id,
            MIN(change_time) AS change_time
        FROM $unprocessed_data
        GROUP BY id
    ) AS unprocessed_data
    ON target.id = unprocessed_data.id
    WHERE target.is_current = 1ut
);

-- Step 4: Transform the stream of unprocessed events into versioned records (rows to insert).
-- Here, all necessary attributes for new versions are calculated: `valid_to`, `is_current`, `is_deleted`.
$updated_data = (
    SELECT
        t.id AS id,
        t.attribute1 AS attribute1,
        t.attribute2 AS attribute2,
        t.is_deleted AS is_deleted,
        -- Logic for the `is_current` flag: it is set to 1 only for the last
        -- record in the chain (`next_change_time IS NULL`), and only if it is not
        -- a delete operation (`is_deleted == 0`).
        IF(t.next_change_time IS NOT NULL OR t.is_deleted == 1ut, 0ut, 1ut) AS is_current,
        t.change_time AS valid_from,
        t.next_change_time AS valid_to
    FROM (
```

```
        -- Subquery calculates the delete flag (`is_deleted`)
        -- and the timestamp of the next event (`next_change_time`) using the LEAD window function.
        SELECT
            unprocessed_data.id AS id,
            unprocessed_data.attribute1 AS attribute1,
            unprocessed_data.attribute2 AS attribute2,
            unprocessed_data.op AS op,
            unprocessed_data.change_time AS change_time,
            IF(unprocessed_data.op = "DELETE", 1ut, 0ut) AS is_deleted,
            LEAD(unprocessed_data.change_time) OVER (PARTITION BY id ORDER BY unprocessed_data.change_time) A
S next_change_time
        FROM $unprocessed_data AS unprocessed_data
    ) AS t
);

-- Step 5: Atomically apply all calculated changes to the target table.
-- UPSERT will update existing records (from $close_open_intervals) and insert new ones (from $updated_data).
UPSERT INTO dimension_scd2_final (id, attribute1, attribute2, is_current, is_deleted, valid_from, valid_to)
SELECT
    id,
    attribute1,
    attribute2,
    is_current,
    is_deleted,
    valid_from,
    valid_to
FROM $close_open_intervals
UNION ALL
SELECT
    id,
    attribute1,
    attribute2,
    is_current,
    is_deleted,
    valid_from,
    valid_to
FROM $updated_data;

-- Step 6: Clear the staging table of processed records.
DELETE FROM dimension_scd_changes ON
SELECT id, change_time FROM $changes;
```

Demonstration of work

In the example below, the entity **Customer** is considered:

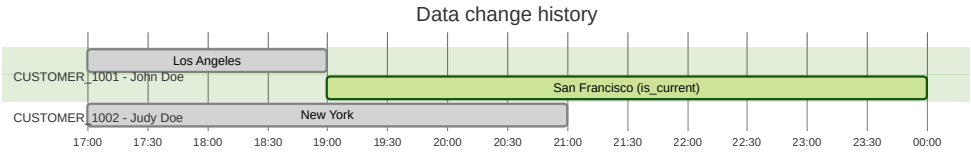
- business key — field `id`,
- attributes — `attribute1` (full name) and `attribute2` (city).

At `2025-08-22 17:00` two customers are created (John in Los Angeles - with id `CUSTOMER_1001` and Judy in New York with id `CUSTOMER_1002`), at `2025-08-22 19:00` customer `CUSTOMER_1001` updates his city to San Francisco `UPDATE`, and at `2025-08-22 21:00` customer `CUSTOMER_1002` is deleted `DELETE`.

id	attribute1	attribute2	change_time	operation
CUSTOMER_1001	John Doe	Los Angeles	2025-08-22 17:00	CREATE
CUSTOMER_1002	Judy Doe	New York	2025-08-22 17:00	CREATE
CUSTOMER_1001	John Doe	San Francisco	2025-08-22 19:00	UPDATE
CUSTOMER_1002	Judy Doe	New York	2025-08-22 21:00	DELETE

The SCD2 process converts such events into interval-based record versions with the `valid_from` and `valid_to` fields. For example, `CUSTOMER_1001` will have two sequential versions: first with the city LA, then with the city SF (the current record, where `valid_to = NULL`). `CUSTOMER_1002` will have one outdated version and a final record with flags `is_deleted=1` and `is_current=0`, indicating that the user has been deleted.

Below are the original events and their corresponding versions in the final table.



id	attribute1	attribute2	valid_from	valid_to	is_current	is_deleted
CUSTOMER_1001	John Doe	Los Angeles	2025-08-22 17:00	2025-08-22 19:00	0	0
CUSTOMER_1001	John Doe	San Francisco	2025-08-22 19:00	NULL	1	0
CUSTOMER_1002	Judy Doe	New York	2025-08-22 17:00	2025-08-22 21:00	0	0
CUSTOMER_1002	Judy Doe	New York	2025-08-22 21:00	NULL	0	1

Retrieving data from the SCD2 table

Retrieving actual data

```
SELECT
    id,
    attribute1,
    attribute2,
    valid_from,
    valid_to
FROM dimension_scd2_final
WHERE is_current = 1ut;
```

Result:

id	attribute1	attribute2	valid_from	valid_to	is_current	is_deleted
CUSTOMER_1001	John Doe	San Francisco	2025-08-22 19:00	NULL	1	0

Retrieving data at a specific point in time

```
$as_of = Timestamp("2025-08-22T19:11:30.000000Z");

SELECT
    id,
    attribute1,
    attribute2,
    valid_from,
    valid_to
FROM dimension_scd2_final
WHERE valid_from <= $as_of
        AND (valid_to IS NULL OR valid_to > $as_of) -- Get records that were valid at the $as_of time
        AND is_deleted = 0ut -- Only records that are not deleted
```

Result:

id	attribute1	attribute2	valid_from	valid_to
CUSTOMER_1001	John Doe	San Francisco	2025-08-22 19:00	NULL
CUSTOMER_1002	Judy Doe	New York	2025-08-22 17:00	2025-08-22 21:00

Retrieving the history of changes for a specific record

```
SELECT
    id,
    attribute1,
    attribute2,
    valid_from,
    valid_to,
    is_current,
    is_deleted
FROM dimension_scd2_final
WHERE id = 'CUSTOMER_1001'
ORDER BY valid_from;
```

Result:

id	attribute1	attribute2	valid_from	valid_to	is_current	is_deleted
CUSTOMER_1001	John Doe	Los Angeles	2025-08-22 17:00	2025-08-22 19:00	0	0
CUSTOMER_1001	John Doe	San Francisco	2025-08-22 19:00	NULL	1	0

Known limitations

This section covers important features of YDB that you need to consider when designing applications and writing queries. For each feature, the current behavior and possible workarounds are described.

Transactions and isolation

Transaction isolation levels

YDB supports different isolation levels for row-based (OLTP) and column-based (OLAP) tables.

- Row-based tables. They support transactions with isolation levels Serializable, `Online Read-Only`, and others. This ensures strict consistency for OLTP workloads.
- Column-based tables. All operations are performed in `Serializable` mode. This guarantees that any analytical transaction works with a consistent snapshot of data, as if no further changes were occurring in the database.

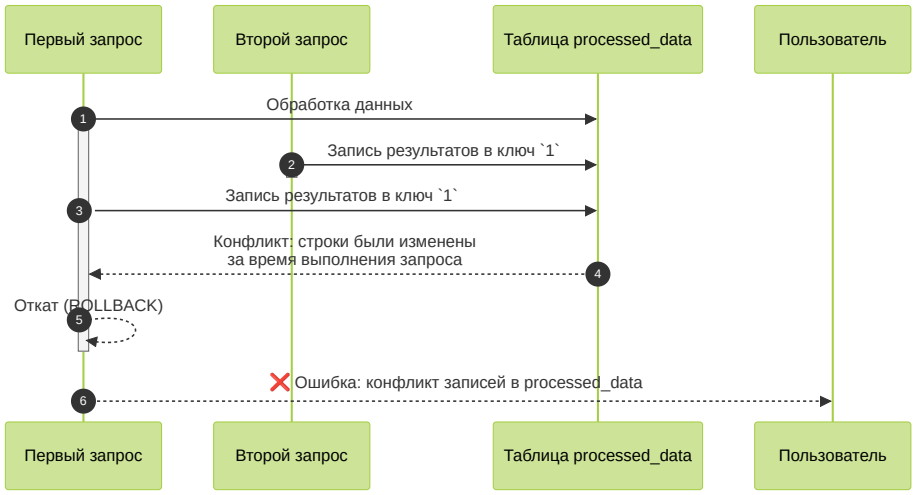
The following discusses the specifics of working with `Serializable` mode in the context of analytical (OLAP) queries.

Features of Serializable isolation with concurrent data access

The `Serializable` isolation level guarantees no read anomalies but imposes certain requirements on the design of ETL/ELT pipelines. Conflicts occur if data that a transaction has read or intends to modify has been changed by another, already completed transaction.

Write-write conflict

Two concurrent tasks attempt to write data to the same key range in the output table. The first task, which started execution but is slower, will be canceled upon commit attempt because the second, faster task has already modified those rows.

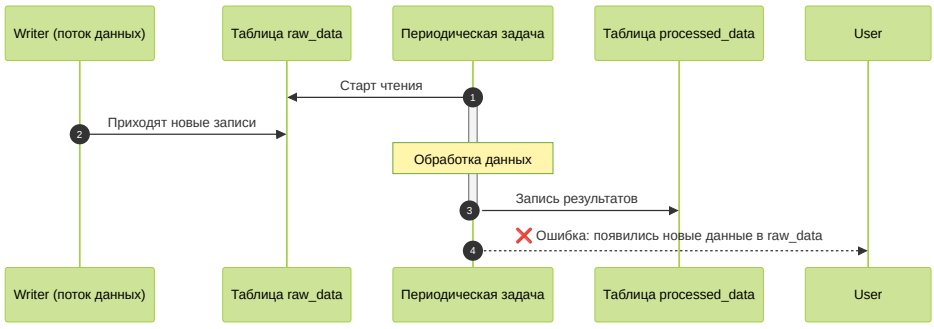


Solution options:

- **Load partitioning:** split the input data so that parallel queries write data to different tables or keys. For example, process data by day or by user IDs in separate tasks.
- **Using intermediate tables:** each task writes its result to its own temporary table. After all tasks complete, data from the temporary tables is moved to the final table in a single transaction using `INSERT INTO ... SELECT FROM`.

Read-write conflict

A long-running analytical query reads data from the `raw_data` table, into which writes are still occurring at the same time. By the time the query finishes reading and processing, the source table has already changed. YDB detects this and cancels the query to guarantee snapshot consistency.



Solution

Using intermediate tables: Create a temporary table with the required data and perform further processing with it. This fixes the data state and eliminates conflicts.

```
CREATE TABLE temp_snapshot AS SELECT * FROM raw_data WHERE ...;
-- Next, we work only with temp_snapshot
INSERT INTO processed_data SELECT process(t.*) FROM temp_snapshot AS t;
```

Modifying queries to column-based and row-based tables simultaneously are prohibited

Within a single transaction, it is impossible to perform data modification operations (DML) simultaneously for both row-based and column-based tables.

Solution

Split the logic into two sequential transactions. First, perform the operation on the row-based table, and after its successful completion, on the column-based table (or vice versa, depending on business logic).

Syntax specifics

Common table expressions (CTEs) are not supported

YQL does not support the `WITH ... AS (CTE)` syntax. Instead, it uses the named expression mechanism with variables starting with the `$` sign.

Solution

Use named expressions. This is a syntactic equivalent of CTE that allows decomposing complex queries.

A part of a query can be extracted into a separate expression and given a name starting with `$` using the named expression mechanism. Such an expression can be used multiple times within a single query. Usage is supported for both table and scalar expressions.

```
-- parameter declaration
DECLARE $days AS Int32;
$cutoff = CurrentUtcTimestamp() - $days * Interval("P1D"); -- creating a scalar named expression

-- creating a table named expression
$base = (
  SELECT *
  FROM raw_events
  WHERE event_ts >= $cutoff -- using a scalar variable
);

-- using a named expression
SELECT * FROM $base WHERE event_ts > CurrentUtcTimestamp()
```

YDB guarantees that when a named expression is used multiple times within a single transaction, the same data will be read. This is ensured by the Serializable transaction isolation level.

Lack of support for correlated subqueries

A correlated subquery is a subquery that references columns from an outer query. Such subqueries are not supported in YQL.
Most use cases of correlated subqueries can be replaced using `JOIN` and aggregate functions.
The complete support matrix and rewrite rules are available in Correlated subqueries, EXISTS, and NOT EXISTS.

EXISTS

Transformation of correlated `EXISTS` → `LEFT SEMI JOIN` or `INNER JOIN` with unique keys on the right side.

Original query:

```
SELECT a.* FROM A a WHERE EXISTS (
  SELECT 1 FROM B b WHERE b.key = a.key AND b.flag = 1
);
```

Solution with LEFT SEMI JOIN

```
$B_match = (
  SELECT key
  FROM B
  WHERE flag = 1
);

SELECT a.*
FROM A AS a
LEFT SEMI JOIN $B_match AS b
ON b.key = a.key;
```

Alternative with INNER JOIN

Deduplicate the matching keys on the right side before joining. This prevents several matching rows from duplicating a row from the left side. Applying `DISTINCT` to the left-side columns is unnecessary and can collapse equal rows from the outer input.

```
$B_match = (
  SELECT key
  FROM B
  WHERE flag = 1
  GROUP BY key
);

SELECT a.*
FROM A AS a
INNER JOIN $B_match AS b
ON b.key = a.key;
```

Subquery with aggregate

Scalar subquery with aggregate → aggregation + JOIN

Original query:

```
SELECT a.*, (SELECT MAX(ts) FROM B b WHERE b.user_id = a.user_id) AS last_ts
FROM A a;
```

Solution

```
$B_last = (
  SELECT user_id, MAX(ts) AS last_ts
  FROM B
  GROUP BY user_id
);

SELECT a.*, bl.last_ts
FROM A AS a
LEFT JOIN $B_last AS bl
ON bl.user_id = a.user_id;
```

NOT EXISTS

Transformation of correlated `NOT EXISTS` → `LEFT ONLY JOIN`.

Original query:

```
SELECT a.* FROM A a WHERE NOT EXISTS (
  SELECT 1 FROM B b WHERE b.key = a.key AND b.flag = 1
);
```

Solution

```
$B_match = (
  SELECT key
  FROM B
  WHERE flag = 1
);

SELECT a.*
FROM A AS a
LEFT ONLY JOIN $B_match AS b
ON b.key = a.key;
```

Support for equi-JOIN only

JOIN conditions can only contain the equality operator (=). Non-equi joins (>, <, >=, <=, BETWEEN) are not supported. The inequality condition can be moved to the WHERE clause after a CROSS JOIN.

> **Warning**
>
> CROSS JOIN creates a Cartesian product of tables. This approach is not recommended for large tables as it leads to an explosive growth of intermediate data volume and performance degradation. Use it only if one of the tables is very small or both tables have been pre-filtered to a small size.

Original query:

```
SELECT e.event_id, e.user_id, e.ts
FROM events AS e
JOIN periods AS p
ON e.user_id = p.user_id
 AND e.ts >= p.start_ts
```

Solution

```
SELECT e.event_id, e.user_id, e.ts
FROM events AS e
CROSS JOIN periods AS p
WHERE e.user_id = p.user_id
  AND e.ts >= p.start_ts
```

Importing data using federated queries

YDB supports federated queries to external data sources (such as ClickHouse, PostgreSQL, etc.). This mechanism is designed for fast ad-hoc analytics and on-the-fly data joining, but is not an optimal tool for bulk and regular loading of large data volumes (ETL/ELT). When using federated queries for import, you may encounter limitations on supported data types and query execution.

Solution

1. Export data from your DBMS to one of the open formats (recommended `CSV`) into a Object Storage bucket. Use the `INSERT INTO ... SELECT FROM` command to read data from an external table associated with your bucket in Object Storage. This approach allows efficient parallelization of data reading.
2. For continuous replication tasks or building complex pipelines, use standard industry tools that integrate with YDB:
   - Change Data Capture (CDC): tools like Debezium can capture changes from the transaction log of your OLTP database and deliver them to YDB.

- ETL/ELT frameworks: systems such as Apache Spark or Apache NiFi have connectors to YDB and allow building flexible and powerful data processing and loading pipelines.

List of limitations:

> **Warning**
>
> External connectors are an experimental feature of YDB. To work with external DBMS through connectors, you need to deploy fq-connector-go and explicitly enable the corresponding sources in the cluster configuration YDB. The functionality may change, so use in production is not recommended without thorough testing.

- ClickHouse
- Greenplum
- Microsoft SQL Server
- MySQL
- PostgreSQL
- YDB (federated queries between YDB databases)

```
-- Parameters (for example, as variables)
DECLARE $yc_key    AS String;
DECLARE $yc_secret AS String;

-- 1) External S3 source with Yandex Cloud endpoint
CREATE EXTERNAL DATA SOURCE s3_backup_ds
WITH (
  SOURCE_TYPE = "S3",
  LOCATION    = "https://storage.yandexcloud.net",  -- endpoint YC
  AUTH_METHOD = "AWS",
  AWS_ACCESS_KEY_ID     = $yc_key,
  AWS_SECRET_ACCESS_KEY = $yc_secret
);

-- 2) External "table" (directory with CSV in bucket)
CREATE EXTERNAL TABLE s3_my_columnstore_table_backup
WITH (
  DATA_SOURCE = "s3_backup_ds",
  LOCATION    = "s3://my-bucket/ydb-backups/processed_data/full/",  -- path in Object Storage
  FORMAT      = "CSV"
)(
  id Uint64,
  event_dt Datetime,
  value String
  -- ... other columns of your CSV table ...
);

-- 3) Export
INSERT INTO s3_my_columnstore_table_backup
SELECT * FROM my_columnstore_table;

-- 4) Data recovery
INSERT INTO my_columnstore_table
SELECT *
FROM s3_my_columnstore_table_backup;
```

Number of partitions is fixed at creation

In YDB, the number of partitions (data segments) for a columnar table is set once at creation and cannot be changed later.

Choosing the correct number of partitions is an important aspect of data schema design.

- Too few partitions: may lead to uneven load on compute nodes (hotspots) and limit query parallelism.
- Too many partitions: may create excessive load on the database management component (Scheme Shard) and increase query processing overhead.

Solution

- Initial number of partitions: for a basic estimate of the number of partitions, you can use the formula `(number of nodes * 4)`. This will maximize cluster resource utilization when executing parallel queries.
- Choose the number of partitions considering the expected growth of data volume and increase in the number of nodes in the cluster.
- The total number of partitions across all tables in a single database must not exceed **2000**.
- If you need to increase the number of partitions, you can create a new table and move data into it using the `CREATE TABLE (PRIMARY KEY (a, b)) PARTITION BY HASH(a) WITH(STORE=COLUMN, PARTITION_COUNT=96) new_table AS SELECT * FROM old_table;` query.

No secondary indexes or skip indexes

The performance of analytical queries with columnar data storage is achieved through mechanisms based on physical data organization: columnar storage, partitioning, sorting by primary key.

Solution

Pay special attention to partitioning keys and primary keys, as described in the Choosing keys for maximum column-oriented table performance section.

Mixing OLTP and OLAP in the same database is not recommended

OLTP and OLAP workloads impose opposite resource requirements, and mixing them almost always leads to mutual performance degradation.

- OLTP workload (row-based tables) is a set of short, fast transactions (key-based read/write) that are latency-critical.
- OLAP workload (column-based tables) is typically long-running and resource-intensive queries that scan large volumes of data, perform aggregation, and actively consume CPU, memory, and disk I/O.

When a heavy OLAP query starts executing, it can monopolize cluster resources, causing fast OLTP queries to queue up and increasing their response time.

It is not recommended to combine OLTP and OLAP workloads in the same database. OLAP workloads can negatively affect OLTP, increasing query execution time.

Solution

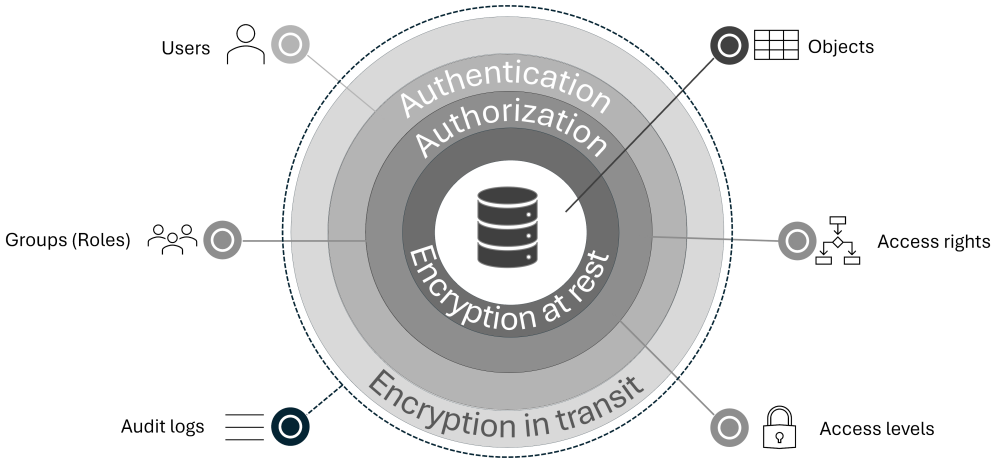
To ensure stable and predictable performance for both workloads, use two separate YDB databases: one for OLTP and another for OLAP. To synchronize data between them, use the Change Data Capture (CDC) mechanism and the Data transfer service, which allows streaming changes from the OLTP database to the OLAP database.

YDB for Security Engineers

This section of YDB documentation covers security-related aspects of working with YDB. It'll be useful for compliance purposes too.

YDB security elements and concepts

Security model in YDB introduces the following concepts:



- **Access subjects**:
  - **Users**. YDB supports both internal users and external users from third-party directory services, such as LDAP and IAM systems.
  - **Groups**. YDB allows grouping users into named collections. The list of users in a group can be modified later. A group can be empty.
- **Access objects** in YDB are scheme objects (tables, views, etc) for which access rights are configured.
- **Access rights** in YDB are used to determine the list of permitted operations with access objects for a given user or group.

  Access rights represent permission for an access subject to perform a specific set of operations (create, drop, select, update, etc) in a cluster or database on a specific access object.

  Access rights can be granted to a user or a group. When a user is added to a group, the user gets the access rights that were granted to the group. When a user is removed from a group, the user loses the access rights of the group.

  For more information about access rights, see Right.
- **Access levels** in YDB are used to determine the list of additional cluster management operations permitted for a given user or group. YDB uses hierarchical access levels: database, viewer, monitoring, and administration. Higher levels automatically include all lower level privileges. In addition, there are two special non-hierarchical lists: bootstrap (for initial cluster bootstrap) and register node (for dynamic node registration).

  For more information about access levels, see Access level lists.
- **Authentication and authorization**. The access control system in YDB provides data protection in a YDB cluster. Due to the access system, only authorized access subjects (users and groups) can work with data. Access to data can be restricted.
  - **Authentication**. When a user connects to a YDB cluster, YDB first identifies the user's account. This process is called authentication. YDB supports various authentication modes. For more information, see Authentication.

    Regardless of an authentication mode, after passing authentication, a user gets a SID and an authentication token.

    - YDB cluster uses a SID for user identification. For example, a SID for a local user is the user login. SIDs for external users also include information about the system where they were created. User SIDs can also be found in system views describing the security configuration.
    - The authentication token is used by YDB nodes to authorize user access before processing user requests.

      The user can then use the received authentication token repeatedly when making requests to the YDB cluster. For more information about the authentication token and related configuration parameters, see auth_config.
  - **Authorization**. Based on the authentication data, a user then goes through authorization — a process that verifies whether a user has sufficient access rights and access levels to perform user operations.
- **Audit logs**. YDB provides audit logs that include data about all operations that attempted to change the YDB objects, such as changing access rights, creating or deleting scheme objects, whether successful or not. Audit logs are intended for people responsible for information security.
- **Encryption**. YDB is a distributed system typically running on a cluster, often spanning multiple datacenters. To protect user data, YDB provides the following technologies:
  - encryption in transit to secure data transmitted between a client and YDB, and between nodes of the YDB cluster.
  - data encryption at rest.

Authentication

Once a network connection is established, the server starts to accept client requests with authentication information for processing. The server uses it to identify the client's account and to verify access to execute the query.

> **Note**
>
> An authentication client refers to a user undergoing the authentication process when accessing YDB. Examples of clients include the SDK or CLI.

The following authentication modes are supported:

- Anonymous authentication.
- Authentication by username and password.
- LDAP authentication.
- Authentication through a third-party IAM provider, for example, Yandex Identity and Access Management.

Anonymous authentication

Anonymous authentication allows you to connect to YDB without specifying any credentials like username and password. This type of access should be used only for educational purposes in local databases that cannot be accessed over the network.

However, if a user or token is specified, the corresponding authentication mode will work with subsequent authorization.

> **Warning**
>
> Anonymous authentication should be used only for informational purposes for local databases that are not accessible over the network.

To enable anonymous authentication, use `false` in the `enforce_user_token_requirement` key of the cluster's configuration file.

Authenticating by username and password

Authentication by username and password using the YDB server is available only to local users. Authentication of external users involves third-party servers.

This access type implies that each database user has a username and password.

Only digits and lowercase Latin letters can be used in usernames. Password complexity requirements can be configured.

The username and hashed password are stored in a table inside the authentication component. The password is hashed using the Argon2 method. Only the system administrator has access to this table.

A token is returned in response to the username and password. Tokens have a default lifetime of 12 hours. To rotate tokens, the client, such as the SDK, independently sends requests to the authentication service. Tokens accelerate authentication and enhance security.

Authentication by username and password includes the following steps:

1. The client accesses the database and presents their username and password to the YDB authentication service.
2. The service validates authentication data. If the data matches, it generates a token and returns it to the authentication service.
3. The client accesses the database, presenting their token as authentication data.

To enable authentication by username and password, ensure that the `use_login_provider` and `enable_login_authentication` parameters are set to the default value of `true` in the configuration file. Besides, to disable anonymous authentication, set the `enforce_user_token_requirement` parameter to `true`.

To learn how to manage roles and users, see Authorization.

Password complexity

YDB allows configuring requirements for password complexity. If a password specified in the `CREATE USER` or `ALTER USER` command does not meet complexity requirements, the command will result in an error. By default, YDB has no password complexity requirements. A password of any length is accepted, including an empty string. A password can contain any number of digits and uppercase or lowercase letters, as well as special characters from the `!@#$%^&*()_+{}|<>?=` list. To set requirements for password complexity, define parameters in the `password_complexity` section in the configuration.

Password brute-force protection

YDB provides password brute-force protection. A user is locked out after exceeding a specified number of failed attempts to enter a password. After a certain period, the user will be unlocked and able to log in again.

By default, a user has four attempts to enter a password. If a user fails to enter the correct password in four attempts, the user will be locked out for an hour. You can change these lockout settings in the `auth_config` section of the configuration.

If necessary, a YDB cluster or database administrator can unlock a user before the lockout period expires.

Manual user lockout

YDB provides another method for disabling authentication for a user, manual user lockout by a YDB cluster or database administrator. An administrator can unlock user accounts that were previously locked manually or automatically after exceeding the number of failed attempts to enter the correct password. For more information about manual user lockout, see the `ALTER USER LOGIN/NOLOGIN` command description.

LDAP directory integration

YDB supports authentication and authorization via an LDAP directory. To use this feature, an LDAP directory service must be deployed and accessible from the YDB servers.

Examples of supported LDAP implementations include OpenLDAP and Active Directory.

Authentication through a third-party IAM provider

- **Anonymous**: Empty token passed in a request.
- **Access Token**: Fixed token set as a parameter for the client (SDK or CLI) and passed in requests.
- **Refresh Token**: OAuth token of a user's personal account set as a parameter for the client (SDK or CLI), which the client periodically sends to the IAM API in the background to rotate a token (obtain a new one) to pass in requests.
- **Service Account Key**: Service account attributes and a signature key set as parameters for the client (SDK or CLI), which the client periodically sends to the IAM API in the background to rotate a token (obtain a new one) to pass in requests.
- **Metadata**: Client (SDK or CLI) periodically accesses a local service to rotate a token (obtain a new one) to pass in requests.
- **OAuth 2.0 token exchange** - The client (SDK or CLI) exchanges a token of another type for an access token using the OAuth 2.0 token exchange protocol, then it uses the access token in YDB API requests.

Any owner of a valid token can get access to perform operations; therefore, the principal objective of the security system is to ensure that a token remains private and to protect it from being compromised.

Authentication modes with token rotation, such as **Refresh Token** and **Service Account Key**, provide a higher level of security compared to the **Access Token** mode that uses a fixed token, since only secrets with a short validity period are transmitted to the YDB server over the network.

The highest level of security and performance is provided when using the **Metadata** mode, since it eliminates the need to work with secrets when deploying an application and allows accessing the IAM system and caching a token in advance, before running the application.

When choosing the authentication mode among those supported by the server and environment, follow the recommendations below:

- **You would normally use Anonymous** on self-deployed local YDB clusters that are inaccessible over the network.
- **You would use Access Token** when other modes are not supported on server side or for setup/debugging purposes. It does not require that the client access IAM. However, if the IAM system supports an API for token rotation, fixed tokens issued by this IAM usually have a short validity period, which makes it necessary to update them manually in the IAM system on a regular basis.
- **Refresh Token** can be used when performing one-time manual operations under a personal account, for example, related to DB data maintenance, performing ad-hoc operations in the CLI, or running applications from a workstation. You can manually obtain this token from IAM once to have it last a long time and save it in an environment variable on a personal workstation to use automatically and with no additional authentication parameters on CLI launch.
- **Service Account Key** is mainly used for applications designed to run in environments where the **Metadata** mode is supported, when testing them outside these environments (for example, on a workstation). It can also be used for applications outside these environments, working as an analog of **Refresh Token** for service accounts. Unlike a personal account, service account access objects and roles can be restricted.
- **Metadata** is used when deploying applications in clouds. Currently, this mode is supported on virtual machines and in Cloud Functions Yandex.Cloud.

The token to specify in request parameters can be obtained in the IAM system that the specific YDB deployment is associated with. In particular, YDB in Yandex.Cloud uses Yandex.Passport OAuth and Yandex.Cloud service accounts. When using YDB in a corporate context, a company's standard centralized authentication system may be used.

When using modes in which the YDB client accesses the IAM system, the IAM URL that provides an API for issuing tokens can be set additionally. By default, existing SDKs and CLIs attempt to access the Yandex.Cloud IAM API hosted at `iam.api.cloud.yandex.net:443`.

Authentication

Authentication using the LDAP protocol is similar to the static credentials authentication process (using a login and password). The difference is that the LDAP directory acts as the authentication component. The LDAP directory is used solely to verify the login/password pair.

> **Note**
>
> Since the LDAP directory is an external, independent service, YDB cannot manage user accounts within it. For successful authentication, the user must already exist in the LDAP directory. The commands `CREATE USER`, `CREATE GROUP`, `ALTER USER`, `ALTER GROUP`, `DROP USER`, and `DROP GROUP` do not affect the list of users and groups in the LDAP directory. Information on managing accounts should be found in the documentation for the specific LDAP directory implementation in use.

Currently, YDB supports only one method of LDAP authentication, known as the `search+bind` method, which involves several steps. Upon receiving the username and password of the user being authenticated, a *bind* operation is performed using the credentials of a special service account specified in the ldap_authentication section. These credentials are defined by the **bind_dn** and **bind_password** configuration parameters. After the service account is successfully authenticated, a search is conducted in the LDAP directory for the user attempting to authenticate in the system. The *search* operation is performed across the entire subtree rooted at the location specified by the **base_dn** configuration parameter and uses the filter defined in the **search_filter** configuration parameter.

Once the user entry is found, YDB performs another *bind* operation using the found user's entry and the password provided earlier. The success of this second *bind* operation determines whether the user authentication is successful.

After successful authentication, a token is generated. This token is then used in place of the username and password, speeding up the authentication process and enhancing security.

> **Note**
>
> When using LDAP authentication, no user passwords are stored in YDB.

Token verification

After a user is authenticated in the system, a token is generated and verified before executing the requested operation. During the token verification process, the system determines on whose behalf the action is being requested and identifies the groups the user belongs to. For users from the LDAP directory, the token does not include information about group memberships. Therefore, after the token is verified, an additional query is made to the LDAP server to retrieve the list of groups the user is a member of.

Groups, like users, are entities that can have assigned access rights to perform operations on database schema objects and other resources. These assigned rights determine which operations a user is authorized to perform.

The process of retrieving a user's group list from an LDAP directory is similar to the steps taken during authentication. First, a *bind* operation is performed using the service user credentials specified by the **bind_dn** and **bind_password** parameters in the ldap_authentication section of the configuration file. After successful authentication, a search is conducted for the user entry associated with the previously generated token. This search uses the **search_filter** parameter. If the user still exists, the result of the *search* operation will be a list of attribute values specified by the **requested_group_attribute** parameter. If this parameter is not set, the *memberOf* attribute is used as the default for reverse group membership. The *memberOf* attribute contains the distinguished names (DNs) of the groups to which the user belongs.

Group search

By default, YDB only searches for groups in which the user is a direct member. However, by enabling the **extended_settings.enable_nested_groups_search** flag in the ldap_authentication section, YDB will attempt to retrieve groups at all levels of nesting, not just those the user directly belongs to. If YDB is configured to work with Active Directory, the Active Directory-specific matching rule LDAP_MATCHING_RULE_IN_CHAIN will be used to find all nested groups. This rule allows for the retrieval of all nested groups with a single query. For LDAP servers based on OpenLDAP, group searches will be conducted using recursive graph traversal, which generally requires multiple queries. In both Active Directory and OpenLDAP configurations, the group search is performed only within the subtree specified by the **base_dn** parameter.

Note

In the current implementation, the group names that YDB uses match the values stored in the *memberOf* attribute. These names can be long and difficult to read.

Example:

```
cn=Developers,ou=Groups,dc=mycompany,dc=net@ldap
```

Note

In the configuration file section that specifies authentication information, the refresh rate for user and group information can be set using the **refresh_time** parameter. For more detailed information about configuration files, refer to the cluster configuration section.

Warning

It should be noted that currently, YDB does not have the capability to track group renaming on the LDAP server side. Consequently, a group with a new name will not retain the rights assigned to the group under its previous name.

LDAP users and groups in YDB

Since YDB supports various methods of user authentication (login and password authentication, IAM provider usage, LDAP directory), it is often helpful to identify the specific source of authentication when handling user and group names. For all authentication types except login and password, a suffix in the format `<username>@<domain>` is appended to user and group names.

For LDAP users, the `<domain>` is determined by the **ldap_authentication_domain** configuration parameter in the configuration section. By default, this parameter is set to `ldap`, so all usernames authenticated through the LDAP directory, as well as their corresponding group names in YDB, will follow this format:

- `user1@ldap`
- `group1@ldap`
- `group2@ldap`

Warning

To indicate that the entered login should be recognized as a username from the LDAP directory, rather than for login and password authentication, you need to append the LDAP authentication domain suffix. This suffix is specified through the **ldap_authentication_domain** configuration parameter.

Below are examples of authenticating the user `user1` using the YDB CLI:

- Authentication of a user from the LDAP directory: `ydb --user user1@ldap -p ydb_profile scheme ls`
- Authentication of a user using the internal YDB mechanism: `ydb --user user1 -p ydb_profile scheme ls`

TLS connection

Depending on the specified configuration parameters, YDB can establish either an encrypted or unencrypted connection. An encrypted connection with the LDAP server is established using the TLS protocol, which is recommended for production clusters. There are two ways to enable a TLS connection:

- Automatically via the `ldaps` connection scheme.
- Using the `StartTls` LDAP protocol extension*.

When using an unencrypted connection, all data transmitted in requests to the LDAP server, including passwords, will be sent in plain text. This method is easier to set up and is more suited for experimentation or testing purposes.

LDAPS

To have YDB automatically establish an encrypted connection with the LDAP server, the **scheme** value in the configuration parameter should be set to `ldaps`. The TLS handshake will be initiated on the port specified in the configuration. If no port is specified, the default port 636 will be used for the `ldaps` scheme. The LDAP server must be configured to accept TLS connections on the specified ports.

LDAP protocol extension `StartTls`

`StartTls` is an LDAP protocol extension that enables message encryption using the TLS protocol. It allows a combination of encrypted and plain-text message transmission within a single connection to the LDAP server. YDB sends a `StartTls` request to the LDAP server to initiate a TLS connection. In YDB, enabling or disabling TLS within an active session is not supported. Therefore, once an encrypted connection is established using `StartTls`, all subsequent messages sent to the LDAP server will be encrypted. One advantage of using this extension, provided the LDAP server is appropriately configured, is the capability to initiate a TLS connection over an unencrypted port. The extension can be enabled in the `use_tls` section of the configuration file.

Authorization

Basic concepts

Authorization in YDB is based on the concepts of:

- Access object
- Access subject
- Access right
- Access control list
- Owner
- User
- Group

Regardless of the authentication method, authorization is always performed on the server side of YDB based on the stored information about access objects and rights. Access rights determine the set of operations available to perform.

Authorization is performed for each user action: the rights are not cached, as they can be revoked or granted at any time.

User

To create, alter, and delete users in YDB, the following commands are available:

- CREATE USER
- ALTER USER
- DROP USER

> **Note**
>
> The scope of the commands `CREATE USER`, `ALTER USER`, and `DROP USER` does not extend to external user directories. Keep this in mind if users with third-party authentication (e.g., LDAP) are connecting to YDB. For example, the `CREATE USER` command does not create a user in the LDAP directory. Learn more about YDB's interaction with the LDAP directory.

> **Note**
>
> There is a separate user `root` with maximum rights. It is created during the initial deployment of the cluster, during which a password must be set immediately. It is not recommended to use this account long-term; instead, users with limited rights should be created.
>
> More about initial deployment:
>
> - Ansible
> - Kubernetes
> - Manually

YDB allows working with users from different directories and systems, and they differ by SID using a suffix.

The suffix `@<subsystem>` identifies the "user source" or "auth domain", within which the uniqueness of all `login` is guaranteed. For example, in the case of LDAP authentication, user names will be `user1@ldap` and `user2@ldap`.
If a `login` without a suffix is specified, it implies users directly created in the YDB cluster.

Group

Any user can be included in or excluded from a certain access group. Once a user is included in a group, they receive all the rights to database objects that were provided to the access group.
With access groups in YDB, business roles for user applications can be implemented by pre-configuring the required access rights to the necessary objects.

> **Note**
>
> An access group can be empty when it does not include any users.
>
> Access groups can be nested.

To create, alter, and delete groups, the following types of YQL queries are available:

- CREATE GROUP
- ALTER GROUP
- DROP GROUP

Right

Rights in YDB are tied not to the subject, but to the access object.

Each access object has a list of permissions — ACL (Access Control List) — it stores all the rights provided to access subjects (users and groups) for the object.

By default, rights are inherited from parents to descendants in the access objects tree.

The following types of YQL queries are used for managing rights:

- GRANT.
- REVOKE.

The following CLI commands are used for managing rights:

- chown

- grant
- revoke
- set
- clear
- clear-inheritance
- set-inheritance

The following CLI commands are used to view the ACL of an access object:

- describe
- list

Object Owner

Each access object has an owner. By default, it becomes the access subject who created the access object.

> **Note**
>
> For the owner, permission lists on this access object are not checked.
>
> They have a full set of rights on the object.

An object owner exists for the entire cluster and each database.

The owner can be changed using the CLI command `chown`.

The owner of an object can be viewed using the CLI command `describe`.

Access level lists

In addition to access control lists that control access to specific scheme objects, YDB uses access level lists to define hierarchical access levels for cluster-wide operations.

For operations where both access control lists and access level lists are checked, both checks are applied together: an action is allowed only if both checks allow it, and denied if either check fails. For other operations, only the corresponding check mechanism is used.

Hierarchy of access levels

Access level lists form a hierarchy (used by the YDB UI, viewer, and many other cluster-wide actions; ordered from lower to higher privileges):

- `database_allowed_sids` (`Database`) - access to operations within a specific database scope
- `viewer_allowed_sids` (`Viewer`) - read-only access to cluster-wide state
- `monitoring_allowed_sids` (`Monitoring`) - access to operational actions in YDB UI
- `administration_allowed_sids` (`Administration`) - administrative actions on the cluster and databases

Higher levels automatically include all lower level privileges, so a subject only needs to appear in one list. For example, presence in `administration_allowed_sids` automatically grants `monitoring`, `viewer`, and `database` privileges.
See Access level descriptions for details.

Additionally, there are two separate access level lists for specific operations:

- `bootstrap_allowed_sids` — allows cluster bootstrap operations
- `register_dynamic_node_allowed_sids` — allows node registration in the cluster

Access level descriptions

Access level lists are configured in the security configuration and determine privileges for:

- **Database** (presence in `database_allowed_sids`) — access only within a specific database scope. Subjects can open YDB UI and work with that database data, but cannot run cluster-wide requests (for example, listing cluster nodes). Requests without a specified database are forbidden.
- **Viewer** (presence in `viewer_allowed_sids`) — read-only access to cluster-wide state: subjects can open YDB UI pages and diagnostics, but cannot run actions that change system state.
- **Monitoring** (presence in `monitoring_allowed_sids`) — access to operational actions in YDB UI, including actions that can change system state. For example, backup, database restore, or executing YQL statements from YDB UI.
- **Administration** (presence in `administration_allowed_sids`) — Full administrative access to the cluster and its databases. Also used for config changes, scheme operations requiring admin privileges, and other administrative checks.
- **Register node** (presence in `register_dynamic_node_allowed_sids`) — a separate (non-hierarchical) level for dynamic node registration in the cluster. It does not automatically grant `database` / `viewer` / `monitoring` / `administration` privileges. For technical reasons, if this list is configured (non-empty), it must include `root@builtin`.
- **Bootstrap** (presence in `bootstrap_allowed_sids`) — a separate (non-hierarchical) level only for cluster bootstrap operations. It is used in an uninitialized state, when the authentication subsystem is not yet functional. Bootstrap is allowed if the subject is in `bootstrap_allowed_sids` or `administration_allowed_sids`; `bootstrap` by itself does not grant full administrative privileges.

Initial cluster security configuration

Initial security is configured automatically when the YDB cluster starts for the first time.

During this process YDB adds a superuser and a set of roles for user access management.

> **Note**
>
> For information about overriding and skipping initial security configuration, see the following sections:
>
> - Skipping initial security configuration
> - Overriding initial security configuration

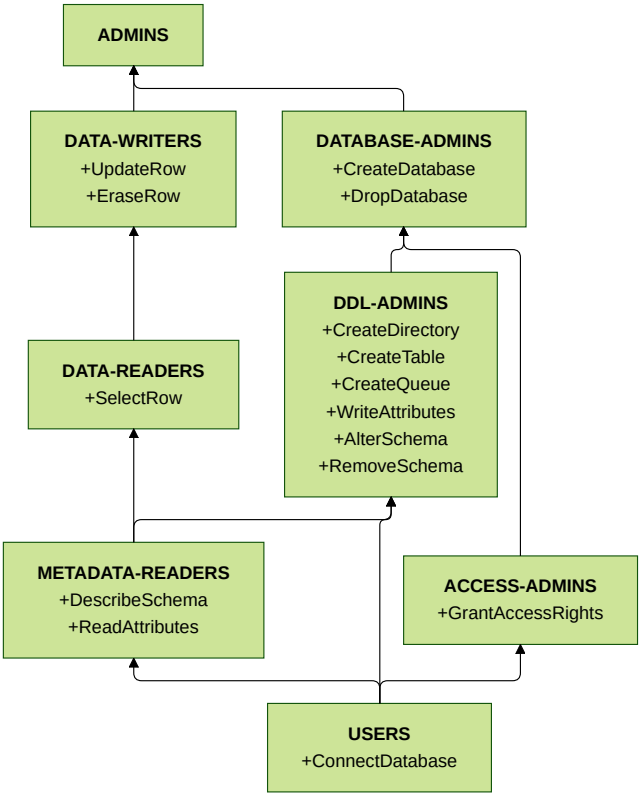
Roles

Role	Description
`ADMINS`	Provides unlimited access rights for the entire YDB cluster scheme.
`DATABASE-ADMINS`	Provides access rights to manage databases, their scheme, and scheme access rights. No data access.
`ACCESS-ADMINS`	Provides access rights to manage scheme access rights. No data access.
`DDL-ADMINS`	Provides access rights to manage the scheme. No data access.
`DATA-WRITERS`	Provides access rights for scheme objects, including reading and modifying data.
`DATA-READERS`	Provides access rights for scheme objects and reading data.
`METADATA-READERS`	Provides access rights for scheme objects. No data access.
`USERS`	Provides access rights for databases. This is a common group for all users.

Groups

Roles in YDB are implemented as a hierarchy of user groups and a set of access rights for these groups. Access rights for the groups are granted on the cluster scheme root.

Groups can be nested, and a child group inherits the access rights of its parent group:



For example, users in the `DATA-WRITERS` group are allowed to:

- View the scheme — `METADATA-READERS`
- Read data — `DATA-READERS`
- Change data — `DATA-WRITERS`

Users in the `DDL-ADMINS` group are allowed to:

- View the scheme — `METADATA-READERS`
- Change the scheme — `DDL-ADMINS`

Users in the `ADMINS` group are allowed to perform all operations on the scheme and data.

Superuser

A superuser belongs to the `ADMINS` and `USERS` groups and has full access rights to the cluster scheme.

By default, a superuser is the `root` user with an empty password.

A group for all users

The `USERS` group is a common group for all local users. When you add new users, they are automatically added to the `USERS` group.

For more information about managing groups and users, see Authorization.

Overriding initial security configuration

You can override the initial security configuration with a custom set of users, groups, and access rights.

To specify custom users, groups, and access rights to be created during the initial security configuration, define the `default_users`, `default_groups`, or `default_access` parameters in the `security_config` section in the cluster configuration file.

Skipping initial security configuration

You can skip initial security configuration by setting the `security_config.disable_builtin_security` parameter to `true`.

Audit log

An audit log is a stream of records that document the operation of the YDB cluster. Unlike technical logs, which help detect failures and troubleshoot issues, the audit log provides data relevant to security. It serves as a source of information that answers the questions: who did what, when, and from where.

A single audit log record may look like this:


```
{"@timestamp":"2025-11-03T18:07:39.056211Z","@log_type":"audit","operation":"ExecuteQueryRequest","databas
e":"/my_dir/db1","status":"SUCCESS","subject":"serviceaccount@as","remote_address":"ipv6:[xxxx:xxx:xxx:xxx:x:
xxxx:xxx:xxxx]"}
```


Examples of typical audit log events:

- Data access through DML requests.
- Schema or configuration management operations.
- Changes to permissions or access-control settings.
- Administrative user actions.

The `audit_config` section in the cluster configuration defines which audit logs are collected, how they need to be serialized and where they are delivered. See the audit log configuration section for details.

Key concepts

Audit events

An *audit event* is a record in the audit log that captures a single security-relevant action. Every event includes attributes that describe different aspects of the event. The common attributes are listed in the Common attributes section.

Audit event sources

An *audit event source* is a YDB service or subsystem that can emit audit events. Each source is identified by a unique identifier (UID) and may expose additional attributes specific to the component. Some sources require extra configuration, such as feature flags, before the source starts emitting events. See the Audit event sources overview for details.

Log classes

Audit events are grouped into *log classes* that represent broad categories of operations. You can enable or disable logging for each class in configuration and, if necessary, tailor the configuration per class. The available log classes are:

Log class	Description
`ClusterAdmin`	Cluster administration requests.
`DatabaseAdmin`	Database administration requests.
`Login`	Login requests.
`NodeRegistration`	Node registration.
`Ddl`	DDL requests.
`Dml`	DML requests.
`Operations`	Asynchronous remote procedure call (RPC) operations that require polling to track the result.
`ExportImport`	Export and import data operations.
`Acl`	Access Control List (ACL) operations.
`AuditHeartbeat`	Synthetic heartbeat messages that confirm audit logging remains operational.
`Default`	Default settings for any component that doesn't have a configuration entry.

At the moment, not all audit event sources categorize events into log classes. For most of them, the basic configuration is sufficient to capture their events. See the Audit event sources overview section for details.

Log phases

Some audit event sources divide the request processing into stages. *Logging phase* indicates the processing stages at which audit logging records events. Specifying logging phases is useful when you need fine-grained visibility into request execution and want to capture events before and after critical processing steps. The available log phases are:

Log phase	Description
`Received`	A request is received and the initial checks and authentication are made. The `status` attribute is set to `IN-PROCESS`. This phase is disabled by default; you must include `Received` in `log_class_config.log_phase` to enable it.
`Completed`	A request is completely finished. The `status` attribute is set to `SUCCESS` or `ERROR`. This phase is enabled by default when `log_class_config.log_phase` is not set.

Audit log destinations

Audit log destination is a target where the audit log stream can be delivered.

You can currently configure the following destinations for the audit log:

- A file on each YDB cluster node.
- An agent for delivering Unified Agent metrics.
- The standard error stream, `stderr`.

You can use any of the listed destinations or their combinations. See the audit log configuration for details.

If you forward the stream to a file, file-system permissions control access to the audit log. Saving the audit log to a file is recommended for production installations.

For test installations, forward the audit log to the standard error stream (`stderr`). Further stream processing depends on the YDB cluster logging settings.

Audit event sources overview

The table below summarizes the built-in audit event sources. Use it to identify which source emits the events you need and how to enable those events.

Source / UID	What it records	Configuration requirements
Schemeshard `schemeshard`	Schema operations, ACL modifications, and user management actions.	Included in the basic audit configuration.
gRPC services `grpc-proxy`	Non-internal requests handled by YDB gRPC endpoints.	Enable the relevant log classes and optional log phases.
gRPC connection `grpc-conn`	Client connection and disconnection events.	Enable the `enable_grpc_audit` feature flag.
gRPC authentication `grpc-login`	gRPC authentication attempts.	Enable the `Login` class in `log_class_config` .
Monitoring service `monitoring`	HTTP requests handled by the monitoring endpoint.	Enable the `ClusterAdmin` class in `log_class_config` .
Heartbeat `audit`	Synthetic heartbeat events proving that audit logging is alive.	Enable the `AuditHeartbeat` class in `log_class_config` and optionally adjust heartbeat settings.
BlobStorage Controller `bsc`	Console-driven BlobStorage Controller configuration changes.	Included in the basic audit configuration.
Distconf `distconf`	Distributed configuration updates.	Included in the basic audit configuration.
Web login `web-login`	Interactions with the web console authentication widget.	Included in the basic audit configuration.
Console `console`	Database lifecycle operations and dynamic configuration changes.	Included in the basic audit configuration.

Audit event attributes

Audit log event attributes are divided into two groups:

- Common attributes present in many *audit event sources* and always carry the same meaning.
- Attributes specific to the source that generates the event.

In this section, you will find a reference guide to the attributes in audit events. It covers both common attributes and source-specific ones. For each source, its UID, recorded operations, and configuration requirements are also provided.

Common attributes

The table below lists the common attributes.

Attribute	Description
`subject`	Event source SID if mandatory authentication is enabled, or `{none}` otherwise.
`sanitized_token`	A partially masked authentication token that was used to execute the request. Can be used to link related events while keeping the original credentials hidden. If authentication was not performed, the value will be `{none}` .
`operation`	Operation name (for example, `ALTER DATABASE` , `CREATE TABLE`).
`component`	Unique identifier (UID) of the *audit event source*.
`status`	Operation completion status. Acceptable values: • `SUCCESS` : The operation completed successfully. • `ERROR` : The operation failed. • `IN-PROCESS` : The operation is in progress.
`reason`	Error message.
`request_id`	Unique ID of the request that invoked the operation. You can use the `request_id` to differentiate events related to different operations and link the events together to build a single audit-related operation context.
`remote_address`	IP of the client that delivered the request.
`detailed_status`	The status delivered by a YDB *audit event source*.
`database`	Database path (for example, `/my_dir/db`).
`cloud_id`	Cloud identifier of the YDB database.

`folder_id`	Folder identifier of the YDB cluster or database.
`resource_id`	Resource identifier of the YDB database. *Optional.*

Schemeshard

UID: `schemeshard`.
Logged operations: Schema operations triggered by DDL queries, ACL modifications, and user management operations.
How to enable: Only basic audit configuration required.

The table below lists additional attributes specific to the `Schemeshard` source.

Attribute	Description
Common schemeshard attributes	**>**
`tx_id`	Unique transaction ID. This ID can be used to differentiate events related to different operations. *Required.*
`paths`	List of paths in the database that are changed by the operation (for example, `[/my_dir/db/table-a, /my_dir/db/table-b]`). *Optional.*
Ownership and permission attributes	**>**
`new_owner`	SID of the new owner of the object when ownership is transferred.
`acl_add`	List of added permissions in short notation (for example, `[+R:someuser]`).
`acl_remove`	List of revoked permissions in short notation (for example, `[-R:someuser]`).
Custom attributes	**>**
`user_attrs_add`	List of custom attributes added when creating objects or updating attributes (for example, `[attr_name1: A, attr_name2: B]`).
`user_attrs_remove`	List of custom attributes removed when creating objects or updating attributes (for example, `[attr_name1, attr_name2]`).
Login/Auth specific	**>**
`login_user`	User name recorded by login operations.
`login_group`	Group name recorded by login operations.
`login_member`	Membership changes.
`login_user_change`	Changes applied to user settings.
`login_user_level`	Privilege level of the user recorded by audit events. This attribute only uses the `admin` value.
Import/Export operation attributes	**>**
`id`	Unique identifier for export or import operations.
`uid`	User-defined label for operations.
`start_time`	Operation start time in ISO 8601 format.
`end_time`	Operation end time in ISO 8601 format.
`last_login`	User's last successful login time in ISO 8601 format.
Export-specific	**>**
`export_type`	Export destination. Possible values: `yt`, `s3`.
`export_item_count`	Number of exported items.
`export_yt_prefix`	YT destination path prefix.
`export_s3_bucket`	S3 bucket used for exports.
`export_s3_prefix`	S3 destination prefix.
Import-specific	**>**
`import_type`	Import source type. It's always `s3`.
`import_item_count`	Number of imported items.
`import_s3_bucket`	S3 bucket used for imports.
`import_s3_prefix`	S3 source prefix.

gRPC services

UID: `grpc-proxy`.
Logged operations: All non-internal gRPC requests.
How to enable: Requires specifying log classes in audit configuration.
Log classes: Depends on the RPC request type: `Ddl`, `Dml`, `Operations`, `ClusterAdmin`, `DatabaseAdmin`, or other classes.
Log phases: `Received`, `Completed`.

The table below lists additional attributes specific to the `gRPC services` source.

Attribute	Description
Common gRPC attributes	**>**
`grpc_method`	RPC method name. *Optional.*
`request`	Sanitized representation of the incoming request. *Optional.*
`start_time`	Operation start time in ISO 8601 format. *Required.*
`end_time`	Operation end time in ISO 8601 format. *Optional.*
Transaction attributes	**>**
`tx_id`	Transaction identifier.
`begin_tx`	Flag set to `1` when the request starts a new transaction.
`commit_tx`	Shows whether the request commits the transaction. Possible values: `true`, `false`.
Request fields	**>**
`query_text`	Sanitized YQL query text.
`prepared_query_id`	Identifier of a prepared query.
`program_text`	MiniKQL program sent with the request.
`schema_changes`	Description of schema modifications requested in the operation.
`table`	Full table path.
`row_count`	Number of rows processed by a bulk upsert request.
`tablet_id`	Tablet identifier.

gRPC connection

UID: `grpc-conn`.
Logged operations: Connection state changes (connect/disconnect).
How to enable: Enable the `enable_grpc_audit` feature flag.

This source uses only common attributes.

gRPC authentication

UID: `grpc-login`.
Logged operations: gRPC authentication.
How to enable: Requires specifying log classes in audit configuration.
Log classes: `Login`.
Log phases: `Completed`.

The table below lists additional attributes specific to the `gRPC authentication` source.

Attribute	Description
`login_user`	User name. *Required.*
`login_user_level`	Privilege level of the user recorded by audit events. This attribute only uses the `admin` value. *Optional.*

Monitoring service

UID: `monitoring`.
Logged operations: HTTP requests handled by the monitoring service.
How to enable: Requires specifying log classes in audit configuration.
Log classes: `ClusterAdmin`.
Log phases: `Received`, `Completed`.

The table below lists additional attributes specific to the `Monitoring service` source.

Attribute	Description
`method`	HTTP request method. For example `POST`, `GET`. *Required.*

`url`	Request path without query parameters. *Required.*
`params`	Raw query parameters. *Optional.*
`body`	Request body (truncated to 2 MB with the `TRUNCATED_BY_YDB` suffix). *Optional.*

Heartbeat

UID: `audit`.
Logged operations: Periodic audit heartbeat messages.
How to enable: Enable this source by specifying log classes in audit configuration.
Log classes: `AuditHeartbeat`.
Log phases: `Completed`.

The table below lists additional attributes specific to the `Heartbeat` source.

Attribute	Description
`node_id`	Node identifier where the event occurred. *Required.*

BlobStorage Controller

UID: `bsc`.
Logged operations: Configuration replacement requests (`TEvControllerReplaceConfigRequest`) emitted by the console.
How to enable: Only basic audit configuration required.

The table below lists additional attributes specific to the `BlobStorage Controller` source.

Attribute	Description
`old_config`	Snapshot of the previous BlobStorage Controller configuration in YAML form. *Optional.*
`new_config`	Snapshot of the configuration that replaced the previous one. *Optional.*

Distconf

UID: `distconf`.
Logged operations: Distributed configuration changes.
How to enable: Only basic audit configuration required.

The table below lists additional attributes specific to the `Distconf` source.

Attribute	Description
`old_config`	Snapshot of the configuration that was active before the distributed update was accepted. Distconf serializes it in YAML. *Required.*
`new_config`	Snapshot of the configuration that Distconf committed after the change. *Required.*

Web login

UID: `web-login`.
Logged operations: Tracks interactions with the YDB web console authentication widget.
How to enable: Only basic audit configuration required.

This source uses only common attributes.

Console

UID: `console`.
Logged operations: Database lifecycle operations and dynamic configuration changes.
How to enable: Only basic audit configuration required.

The table below lists additional attributes specific to the `Console` source.

Attribute	Description
`old_config`	Snapshot of the configuration that was in effect before the console request was applied. *Optional.*
`new_config`	Snapshot of the configuration that the console applied. *Optional.*

Audit log configuration

Enabling audit log

Audit logging works cluster-wide. For the *basic configuration*, add the `audit_config` section to the cluster configuration and specify one or more stream destinations (`file_backend`, `unified_agent_backend`, `stderr_backend`):

```
audit_config:
  file_backend:
    format: audit_log_format
    file_path: "path_to_log_file"
  unified_agent_backend:
    format: audit_log_format
    log_name: session_meta_log_name
```

```
  stderr_backend:
    format: audit_log_format
```

Audit config parameters

All fields are optional.

Field	Description
`stderr_backend`	Forward the audit log to the standard error stream (`stderr`). See the backend settings for details.
`file_backend`	Write the audit log to a file at each cluster node. See the backend settings for details.
`unified_agent_backend`	Stream the audit log to the Unified Agent. In addition, you need to define the `uaclient_config` section in the cluster configuration. See the backend settings for details.
`log_class_config`	An array of audit rules for different log classes. See the log class configuration.
`heartbeat`	Optional heartbeat configuration. See the heartbeat settings.

Backend settings

Each backend supports the following fields:

Field	Description
`format`	Audit log format. The default value is `JSON`. See Log format for details. *Optional.*
`file_path`	Path to the file that the audit log will be streamed to. If the path and the file are missing, they will be created on each node at cluster startup. If the file exists, the data will be appended to it. Only for `file_backend`. *Required.*
`log_name`	The session metadata delivered with the message. Using the metadata, you can redirect the log stream to one or more child channels based on the condition: `_log_name: "session_meta_log_name"`. Only for `unified_agent_backend`. *Optional.*
`log_json_envelope`	JSON template that wraps each log record. The template must contain the `%message%` placeholder, which is replaced with the serialized audit record. See the Envelope format. *Optional.*

Log format

The `format` field specifies the serialization format for audit events. The supported formats are:

Format	Description
`JSON`	Each audit event is serialized as a single-line JSON object preceded by an ISO 8601 timestamp. Example: `<time>: {"k1": "v1", "k2": "v2", ...}` *`k1`, `k2`, ..., `kn` represent audit log attributes; `v1`, `v2`, ..., `vn` represent their values.*
`TXT`	Each audit event is serialized as a single-line text string in the `key=value` format, preceded by an ISO 8601 timestamp. Example: `<time>: k1=v1, k2=v2, ...` *`k1`, `k2`, ..., `kn` represent audit log attributes; `v1`, `v2`, ..., `vn` represent their values.*
`JSON_LOG_COMPATIBLE`	Each audit event is serialized as a single-line JSON object suitable for output to destinations shared with debug logs. The object contains the `@timestamp` field with the ISO 8601 timestamp and the `@log_type` field set to `audit`. Example: `{"@timestamp": "<ISO 8601 time>", "@log_type": "audit", "k1": "v1", "k2": "v2", ...}` *`@timestamp` stores the ISO 8601 timestamp; `k1`, `k2`, ..., `kn` represent audit log attributes; `v1`, `v2`, ..., `vn` represent their values.*

Envelope format

Backends can wrap audit events into a custom envelope before delivering them to the backend by specifying the `log_json_envelope` field. The template must contain the `%message%` placeholder, which is replaced with the serialized audit record in the selected format.

For example, the following configuration outputs audit events to `stderr` in JSON format, wrapped in a custom envelope:

```
audit_config:
  stderr_backend:
    format: JSON
    log_json_envelope: '{"audit": %message%, "source": "ydb-audit-log"}'
```

See the Example with Envelope JSON section for output details.

Log class configuration

Each entry in `log_class_config` accepts the following fields:

Field	Description
`log_class`	Class name to configure. Uses values from the log classes list. The `log_class_config` list must not contain two classes with the same name. *Required.*

`enable_logging`	Enables audit event emission for the selected log class. Disabled by default. *Optional.*
`exclude_account_type`	Array of account type (`Anonymous` , `User` , `Service` , `ServiceImpersonatedFromUser`) that should exclude events even if logging is enabled. *Optional.*
`log_phase`	Array of request processing phases to log. See the Log phases. *Optional.*

Heartbeat settings

Heartbeat events help you monitor the health of the audit logging subsystem. They allow you to create alerts for missing audit events without false positives during periods of no activity.

`heartbeat.interval_seconds` controls how often audit heartbeat events are written. A value of `0` disables heartbeat messages.

Config samples

Simple configuration. Below is a simple configuration that saves the audit log text to a file in `TXT` format.

```
audit_config:
  file_backend:
    format: TXT
    file_path: "/var/log/ydb-audit.log"
```

Advanced configuration. The following configuration demonstrates more advanced settings:

- It sends the audit log to Unified Agent in `TXT` format with the `audit` label and also outputs it to `stderr` in `JSON` format.
- The `Default` settings enable logging for all classes in the `Completed` phase.
- Additionally, `ClusterAdmin` is configured to log the `Received` phase, and `DatabaseAdmin` is configured to exclude events from anonymous users:

```
audit_config:
  unified_agent_backend:
    format: TXT
    log_name: audit
  stderr_backend:
    format: JSON
  log_class_config:
    - log_class: ClusterAdmin
      enable_logging: true
      log_phase: [Received, Completed]
    - log_class: DatabaseAdmin
      enable_logging: true
      log_phase: [Completed]
      exclude_account_type: [Anonymous]
    - log_class: Default
      enable_logging: true
  heartbeat:
    interval_seconds: 60
```

Examples

The following tabs show the same audit log event written using different backend settings.

JSON

The `JSON` format produces entries like:

```
2023-03-14T10:41:36.485788Z: {"paths":"[/my_dir/db1/some_dir]","tx_id":"281474976775658","database":"/my_dir/db1","remote_address":"ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]:xxxxx","status":"SUCCESS","subject":"{none}","sanitized_token":"{none}", "detailed_status":"StatusAccepted","operation":"MODIFY ACL","component":"schemeshard","acl_add":"[+(ConnDB):subject:-]"}
2023-03-13T20:07:30.927210Z: {"reason":"Check failed: path: '/my_dir/db1/some_dir', error: path exist, request accepts it (id: [OwnerId: 72075186224037889, LocalPathId: 3], type: EPathTypeDir, state: EPathStateNoChanges)","paths":"[/my_dir/db1/some_dir]","tx_id":"844424930216970","database":"/my_dir/db1","remote_address":"ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]:xxxxx","status":"SUCCESS","subject":"{none}","sanitized_token":"{none}","detailed_status":"StatusAlreadyExists","operation":"CREATE DIRECTORY","component":"schemeshard"}
2025-11-03T18:07:39.056211Z: {"@log_type":"audit","begin_tx":1,"commit_tx":1,"component":"grpc-proxy","database":"/my_dir/db1","detailed_status":"SUCCESS","end_time":"2025-11-03T18:07:39.056204Z","grpc_method":"Ydb.Query.V1.QueryService/ExecuteQuery","operation":"ExecuteQueryRequest","query_text":"SELECT * FROM `my_row_table`;","remote_address":"ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]","sanitized_token":"xxxxxxxx.**","start_time":"2025-11-03T18:07:39.054863Z","status":"SUCCESS","subject":"serviceaccount@as"}
2025-11-03T17:41:44.203214Z: {"component":"monitoring","remote_address":"ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]","operation":"HTTP REQUEST","method":"POST","url":"/viewer/query","params":"base64=false&schema=multipart","body":"{\"query\":\"SELECT * FROM `my_row_table`;\",\"database\":\"/local\",\"action\":\"execute-query\",\"syntax\":\"yql_v1\"}","status":"IN-PROCESS","reason":"Execute"}
```

TXT

The `TXT` format produces entries like:

```
2023-03-14T10:41:36.485788Z: component=schemeshard, tx_id=281474976775658, remote_address=ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]:xxxxx, subject={none}, database=/my_dir/db1, operation=MODIFY ACL, paths=[/my_dir/db1/some_dir], status=SUCCESS, detailed_status=StatusSuccess, acl_add=[+(ConnDB):subject:-]
2023-03-13T20:07:30.927210Z: component=schemeshard, tx_id=281474976775657, remote_address=ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]:xxxxx, subject={none}, database=/my_dir/db1, operation=CREATE DIRECTORY, paths=[/my_dir/
```

```
db1/some_dir], status=SUCCESS, detailed_status=StatusAlreadyExists, reason=Check failed: path: '/my_dir/db1/s
ome_dir', error: path exist, request accepts it (id: [OwnerId: 72075186224037889, LocalPathId: 3], type: EPat
hTypeDir, state: EPathStateNoChanges)
2025-11-03T18:07:39.056211Z: component=grpc-proxy, tx_id=281474976775656, remote_address=ipv6:[xxxx:xxx:xxx:x
xx:x:xxxx:xxx:xxxx]:xxxxx, subject=serviceaccount@as, database=/my_dir/db1, operation=ExecuteQueryRequest, qu
ery_text=SELECT * FROM `my_row_table`; status=SUCCESS, detailed_status=StatusSuccess, begin_tx=1, commit_tx=
1, end_time=2025-11-03T18:07:39.056204Z, grpc_method=Ydb.Query.V1.QueryService/ExecuteQuery, sanitized_token=
xxxxxxxx.**, start_time=2025-11-03T18:07:39.054863Z
2025-11-03T17:41:44.203214Z: component=monitoring, remote_address=ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx], op
eration=HTTP REQUEST, method=POST, url=/viewer/query, params=base64=false&schema=multipart, body={"query":"SE
LECT * FROM `my_row_table`;","database":"/local","action":"execute-query","syntax":"yql_v1"}, status=IN-PROCE
SS, reason=Execute
```

JSON_LOG_COMPATIBLE

The `JSON_LOG_COMPATIBLE` format produces entries like:


```
{"@timestamp":"2023-03-14T10:41:36.485788Z","@log_type":"audit","paths":"[/my_dir/db1/some_dir]","tx_id":"281
474976775658","database":"/my_dir/db1","remote_address":"ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]:xxxxx","stat
us":"SUCCESS","subject":"{none}","detailed_status":"StatusAccepted","operation":"MODIFY ACL","component":"sch
emeshard","acl_add":"[+(ConnDB):subject:-]"}
{"@timestamp":"2023-03-13T20:07:30.927210Z","@log_type":"audit","reason":"Check failed: path: '/my_dir/db1/so
me_dir', error: path exist, request accepts it (id: [OwnerId: 72075186224037889, LocalPathId: 3], type: EPath
TypeDir, state: EPathStateNoChanges)","paths":"[/my_dir/db1/some_dir]","tx_id":"844424930216970","databas
e":"/my_dir/db1","remote_address":"ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]:xxxxx","status":"SUCCESS","subjec
t":"{none}","detailed_status":"StatusAlreadyExists","operation":"CREATE DIRECTORY","component":"schemeshard"}
{"@timestamp":"2025-11-03T18:07:39.056211Z","@log_type":"audit","begin_tx":1,"commit_tx":1,"component":"grpc-
proxy","database":"/my_dir/db1","detailed_status":"SUCCESS","end_time":"2025-11-03T18:07:39.056204Z","grpc_me
thod":"Ydb.Query.V1.QueryService/ExecuteQuery","operation":"ExecuteQueryRequest","query_text":"SELECT * FROM
`my_row_table`;","remote_address":"ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]","start_time":"2025-11-03T18:07:3
9.054863Z","status":"SUCCESS","subject":"serviceaccount@as","sanitized_token":"xxxxxxxx.**"}
{"@timestamp":"2025-11-03T17:41:44.203214Z","@log_type":"audit","component":"monitoring","remote_address":"ip
v6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]","operation":"HTTP REQUEST","method":"POST","url":"/viewer/query","para
ms":"base64=false&schema=multipart","body":"{\"query\":\"SELECT * FROM `my_row_table`;\",\"database\":\"/loca
l\",\"action\":\"execute-query\",\"syntax\":\"yql_v1\"}","status":"IN-PROCESS","reason":"Execute"}
```


Envelope JSON

The JSON envelope template `{"audit": %message%, "source": "ydb-audit-log"}` produces entries like:


```
{"audit":"2023-03-14T10:41:36.485788Z: {\"paths\":\"[/my_dir/db1/some_dir]\",\"tx_id\":\"281474976775658
\",\"database\":\"/my_dir/db1\",\"remote_address\":\"ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]:xxxxx\",\"status
\":\"SUCCESS\",\"subject\":\"{none}\",\"detailed_status\":\"StatusAccepted\",\"operation\":\"MODIFY ACL\",\"c
omponent\":\"schemeshard\",\"acl_add\":\"[+(ConnDB):subject:-]\"}\n","source":"ydb-audit-log"}
{"audit":"2023-03-13T20:07:30.927210Z: {\"reason\":\"Check failed: path: '/my_dir/db1/some_dir', error: path
exist, request accepts it (id: [OwnerId: 72075186224037889, LocalPathId: 3], type: EPathTypeDir, state: EPath
StateNoChanges)\",\"paths\":\"[/my_dir/db1/some_dir]\",\"tx_id\":\"844424930216970\",\"database\":\"/my_dir/d
b1\",\"remote_address\":\"ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]:xxxxx\",\"status\":\"SUCCESS\",\"subject
\":\"{none}\",\"detailed_status\":\"StatusAlreadyExists\",\"operation\":\"CREATE DIRECTORY\",\"component
\":\"schemeshard\"}\n","source":"ydb-audit-log"}
{"audit":"2025-11-03T18:07:39.056211Z: {\"@log_type\":\"audit\",\"begin_tx\":1,\"commit_tx\":1,\"component
\":\"grpc-proxy\",\"database\":\"/my_dir/db1\",\"detailed_status\":\"SUCCESS\",\"end_time\":\"2025-11-03T18:0
7:39.056204Z\",\"grpc_method\":\"Ydb.Query.V1.QueryService/ExecuteQuery\",\"operation\":\"ExecuteQueryRequest
\",\"query_text\":\"SELECT * FROM `my_row_table`;\",\"remote_address\":\"ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xx
xx]\",\"sanitized_token\":\"xxxxxxxx.**\",\"start_time\":\"2025-11-03T18:07:39.054863Z\",\"status\":\"SUCCESS
\",\"subject\":\"serviceaccount@as\"}\n","source":"ydb-audit-log"}
{"audit":"2025-11-03T17:41:44.203214Z: {\"component\":\"monitoring\",\"remote_address\":\"ipv6:[xxxx:xxx:xxx:
xxx:x:xxxx:xxx:xxxx]\",\"operation\":\"HTTP REQUEST\",\"method\":\"POST\",\"url\":\"/viewer/query\",\"params
\":\"base64=false&schema=multipart\",\"body\":\"{\\\"query\\\":\\\"SELECT * FROM `my_row_table`;\\\",\\\"data
base\\\":\\\"/local\\\",\\\"action\\\":\\\"execute-query\\\",\\\"syntax\\\":\\\"yql_v1\\\"}\",\"status\":\"IN
-PROCESS\",\"reason\":\"Execute\"}\n","source":"ydb-audit-log"}
```


Pretty-JSON

The same audit log entry formatted for readability:


```
{
  "paths": "[/my_dir/db1/some_dir]",
  "tx_id": "281474976775658",
  "database": "/my_dir/db1",
  "remote_address": "ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]:xxxxx",
  "status": "SUCCESS",
  "subject": "{none}",
  "detailed_status": "StatusAccepted",
  "operation": "MODIFY ACL",
  "component": "schemeshard",
  "acl_add": "[+(ConnDB):subject:-]"
}
```

Encryption in YDB

YDB provides two main approaches for user data encryption:

- Data at rest encryption
- Data in transit encryption

It is recommended to use both of them simultaneously in any production environments that handle sensitive data.

Data at rest encryption

YDB supports transparent data encryption at the DS proxy level using the ChaCha8 algorithm. YDB includes two implementations of this algorithm, which switch depending on the availability of the AVX-512F instruction set.

By default, data at rest encryption is disabled. For instructions on enabling it, refer to the Blob Storage Configuration section.

For more details on the implementation, refer to ydb/core/blobstorage/dsproxy/dsproxy_encrypt.cpp and ydb/core/blobstorage/crypto.

Data in transit encryption

As YDB is a distributed system typically running on a cluster, often spanning multiple datacenters or availability zones, user data is routinely transferred over the network. Various protocols can be involved, and each can be configured to run over TLS. Below is a list of protocols supported by YDB:

- Interconnect, a specialized protocol for all communication between YDB nodes.
- YDB as a server:
  - gRPC for external communication with client applications designed to work natively with YDB via the SDK or CLI.
  - Kafka wire protocol for external communication with client applications initially designed to work with Apache Kafka.
  - HTTP for running the YDB UI, exposing metrics, and other miscellaneous endpoints.
- YDB as a client:
  - LDAP for user authentication.
  - Federated queries, a feature that allows YDB to query various external data sources. Some sources are queried directly from the `ydbd` process, while others are proxied via a separate connector process.
  - Tracing data sent to an external collector via gRPC.
- In asynchronous replication between two YDB databases, one serves as a client to the other.

By default, data in transit encryption is disabled and must be enabled separately for each protocol. They can either share the same set of TLS certificates or use dedicated ones. For instructions on how to enable TLS, refer to the tls section.

Short access control notation

When describing or logging the permissions granted to users (e.g., in the audit log records), a special short notation for access control may be used. The notation can vary slightly depending on the list of permissions and their inheritance from child objects.

Notation format

Each entry begins with a `+` sign and consists of 2 or 3 attributes listed through the `:` symbol.
These attributes are:

- List of permissions. If there are multiple, they are wrapped in round brackets and separated by `|`. *Mandatory.*
- SID of the subject granted permissions. *Mandatory.*
- Inheritance type. *Optional.*

When the inheritance type is not specified, it means that permission wasn't inherited.

Examples

- `+R:subject:O`
- `+W:subject`
- `+(SR|UR):subject`
- `+(SR|ConnDB):subject:OC+`

List of permissions

A short abbreviation is used to record each permission.

Permission Groups

Permission groups are unions of several permissions. Where possible, one of the groups will be indicated in the short notation.
For example, `+R:subject` — permission to read.

Group	Description
`L`	(list) enumeration. It consists of permissions to read ACL attributes and describe objects.
`R`	(read) reading. It consists of permissions to enumerate and read from a table and a topic.
`W`	(write) writing. It consists of permissions to update and delete table records, write ACL attributes, create subdirectories, create tables, and topics, modify and delete objects, and change user attributes.
`U`	(use) use. It consists of permissions for reading, writing, granting access rights, and sending requests to the database.
`UL`	(use legacy) obsolete version of use. It consists of permissions for reading, writing, and granting access rights.
`M`	(manage) management. It consists of permissions to create and delete databases.
`F`	(full) all rights. It consists of permissions for use and management.
`FL`	(full legacy) obsolete version of all rights. It consists of permissions for use (obsolete) and management.

Simple Permissions

If there's no matching permission group, the list of permissions will be provided in parentheses separated by the vertical bar `|` symbol.

For example, `+(SR|UR):subject` — permission for reading and updating table records.

Permission	Description
`SR`	(select row) reading from the table
`UR`	(update row) updating table records
`ER`	(erase row) deleting table records
`RA`	(read attributes) reading ACL attributes
`WA`	(write attributes) writing ACL attributes
`CD`	(create directory) creating subdirectory
`CT`	(create table) creating table
`CQ`	(create queue) creating queue
`RS`	(remove schema) deleting objects
`DS`	(describe schema) describing objects, listing directories content
`AS`	(alter schema) modifying objects
`CDB`	(create database) creating database
`DDB`	(drop database) deleting database
`GAR`	(grant access rights) granting access rights (not exceeding their own)
`WUA`	(write user attributes) changing user attributes

`ConnDB`	(connect database) connecting and sending requests to the database

Inheritance Types

One or more inheritance flags can be used to describe the passing of permissions to child objects.

Flag	Description
`-`	without inheritance
`O`	this entry will be inherited by child objects
`C`	this entry will be inherited by child containers
`+`	this entry will be used only for inheritance and will not be used for access checking on the current object

YDB Development

This section contains guidelines for YDB developers and contributors.

- Working on a change
- Build and test using Ya Make build system
- Releases

Development process: working on a change for YDB

This section contains a step-by-step scenario which helps you complete necessary configuration steps, and learn how to bring a change to the YDB project. This scenario does not have to be strictly followed, you may develop your own approach based on the provided information.

Set up the environment

GitHub account

You need to have a GitHub account to suggest any changes to the YDB source code. Register at GitHub if haven't done it yet.

SSH key pair

- To connect to GitHub, you can use ssh or a token. The recommended method is ssh keys.
- If you don't have keys yet, create new keys. Full instructions are on this GitHub page.

Remote development

If you are developing on a remote host, you can use the key from your laptop. Configure SSH agent forwarding using GitHub's guide.

Git CLI

You need to have the `git` command-line utility installed to run commands from the console. Visit the Downloads page of the official website for installation instructions.

To install it under Linux/Ubuntu run:

```
sudo apt-get update
sudo apt-get install git
```

Build dependencies

You need to have some libraries installed on the development machine.

To install it under Linux/Ubuntu run:

```
sudo apt-get update
sudo apt-get install libidn11-dev libaio-dev libc6-dev
```

GitHub CLI (optional)

Using GitHub CLI enables you to create Pull Requests and manage repositories from a command line. You can also use GitHub UI for such actions.

Install GitHub CLI as described at the home page. For Linux Ubuntu, you can go directly to the installation instructions.

Run authentication configuration:

```
gh auth login
```

You will be asked several questions interactively, answer them as follows:

Question	Answer
What account do you want to log into?	**GitHub.com**
What is your preferred protocol for Git operations?	**SSH**
Upload your SSH public key to your GitHub account?	Choose a file with a public key (extention `.pub`) of those created on the "Create SSH key pair" step, for instance `/home/user/.ssh/id_ed25519.pub`
Title for your SSH key	**GitHub CLI** (leave default)
How would you like to authenticate GitHub CLI	**Paste your authentication token**

After the last answer, you will be asked for a token which you can generate in the GitHub UI:

> **Tip**
>
> You can generate a Personal Access Token here.
> The minimum required scopes are 'repo', 'read:org', 'admin:public_key'.

Open the https://github.com/settings/tokens, click on "Generate new token" / "Classic", tick FOUR boxes:

- **Box** `workflow`
- Three others as adivised in the tip: "repo", "admin:public_key" and "read:org" (under "admin:org")

And copy-paste the shown token to complete the GitHub CLI configuration.

Fork and clone repository

YDB official repository is https://github.com/ydb-platform/ydb, located under the YDB organization account `ydb-platform`.

To work on the YDB code changes, you need to create a fork repository under your GitHub account. Create a fork by pressing the `Fork` button on the official YDB repository page.

After your fork is set up, create a local git repository with two remotes:

- `official`: official YDB repository, for main and stable branches
- `fork`: your YDB repository fork, for your development branches

```
mkdir -p ~/ydbwork
cd ~/ydbwork
git clone -o official git@github.com:ydb-platform/ydb.git
```

```
cd ydb
git remote add fork git@github.com:{your_github_user_name}/ydb.git
```

Once completed, you have a YDB Git repository set up in `~/ydbwork/ydb`.

Forking a repository is an instant action, however cloning to the local machine takes some time to transfer about 650 MB of repository data over the network.

Next, let's configure the default `git push` behavior:

```
git config push.default current
git config push.autoSetupRemote true
```

This way, `git push {remote}` command will automatically set upstream for the current branch to the `{remote}` and consecutive `git push` commands will only push current branch.

If you intend to use GitHub CLI, then set `ydb-platform/ydb` as a default repository for GitHub CLI:

```
gh repo set-default ydb-platform/ydb
```

Configure commit authorship

Run the following command to set up your name and email for commits pushed using Git (replace example name and email with your real ones):

```
git config --global user.name "Marco Polo"
git config --global user.email "marco@ydb.tech"
```

Working on a feature

To start working on a feature, ensure the steps specified in the Setup the environment section above are completed.

Refresh trunk

Usually you need a fresh revision to branch from. Sync your local `main` branch by running the following command in the repository:

If your current local branch is `main`:

```
git pull --ff-only official main
```

If your current local branch is not `main`:

```
cd ~/ydbwork/ydb
git fetch official main:main
```

This command updates your local `main` branch without checking it out.

Create a development branch

Create a development branch using Git (replace "feature42" with a name for your new branch):

```
git checkout -b feature42
```

Make changes and commits

Edit files locally, use standard Git commands to add files, verify status, make commits, and push changes to your fork repository:

```
git add .
git status
```

```
git commit -m "Implemented feature 42"
git push fork
```

Consecutive pushes do not require an upstream or a branch name:

```
git push
```

Create a pull request to the official repository

When the changes are completed and locally tested (see Ya Build and Test), create Pull Request.

GitHub UI

Visit your branch's page on GitHub.com (`https://github.com/{your_github_user_name}/ydb/tree/{branch_name}`), press `Contribute` and then `Open Pull Request` .
You can also use the link in the `git push` output to open a Pull Request:

```
...
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for '{branch_name}' on GitHub by visiting:
remote:      https://github.com/{your_github_user_name}/test/pull/new/{branch_name}
...
```

GitHub CLI

Install and configure GitHub CLI.

```
cd ~/ydbwork/ydb
```

```
gh pr create --title "Feature 42 implemented"
```

After answering some questions, the Pull Request will be created and you will get a link to its page on GitHub.com.

Precommit checks

Prior to merging, the precommit checks are run for the Pull Request.

For changes in the YDB code, precommit checks build YDB artifacts, and run tests as described in `ya.make` files. Build and test run on a specific commit which merges your changes to the current `main` branch. If there are merge conflicts, build/test checks cannot be run, and you need to rebase your changes as described below.

You can see the checks status on the Pull Request page. Also, key information for YDB build/test checks progress and status is published to the comments of the Pull Ruquest.

If you are not a member of the YDB team, build/test checks do not run until a team member reviews your changes and approves the PR for tests by assigning a label `ok-to-test` .

Checks are restarted every time you push changes to the pull request, cancelling the previous run if it's still in progress. Each iteration of checks produces its own comment on the pull request page, so you can see the history of checks.

If you are a member of the YDB team, you can also restart checks on a new merge commit without pushing. To do so, add label `rebase-and-check` to the PR.

Test results

You can click on the test amounts in different sections of the test results comment to get to the simple HTML test report. In this report you can see which tests have been failed/passed, and get to their logs.

Test history

Each time when tests are run by the YDB CI, their results are uploaded to the test history application. There's a link "Test history" in the comment with test results heading to the page with the relevant run in this application.

In the "Test History" YDB team members can browse test runs, search for tests, see the logs, and compare them between different test runs. If some test is failed in a particular precommit check, it can be seen in its history if this failure had been introduced by the change, or the test had been broken/flaky earlier.

Review and merge

The Pull Request can be merged after obtaining an approval from the YDB team member. Comments are used for communication. Finally a reviewer from the YDB team clicks on the 'Merge' button.

Update changes

If there's a Pull Request opened for some development branch in your repository, it will update every time you push to that branch, restarting the checks.

Rebase changes

If you have conflicts on the Pull Request, you may rebase your changes on top of the actual trunk from the official repository. To do so, refresh main branch state on your local machine, and run the rebase command:

```
# Assuming your active branch is your development branch
git fetch official main:main
git rebase main
```

Cherry-picking fixes to the stable branch

When required to cherry-pick a fix to the stable branch, first branch off of the stable branch:

```
git fetch official
git checkout -b "cherry-pick-fix42" official/stable-24-1
```

Then cherry-pick the fix and push the branch to your fork:

```
git cherry-pick {fixes_commit_hash}
git push fork
```

And then create a PR from your branch with the cherry-picked fix to the stable branch. It is done similarly to opening a PR to `main`, but make sure to double-check the target branch.

If you are using GitHub CLI, pass `-B` argument to specify the target branch:

```
gh pr create --title "Title" -B stable-24-1
```

Build and test YDB using Ya Make

Ya Make is a build and test system used historically for YDB development. Initially designed for C++, now it supports number of programming languages including Java, Go, and Python.

Ya Make build configuration language is a primary one for YDB, with a `ya.make` file in each directory representing Ya Make targets.

Setup the development environment as described in Working on a change - Setup environment arcticle to work with `Ya Make`.

Running Ya commands

There's a `ya` script in the YDB repository root to run `Ya Make` commands from the console. You can add it to the PATH evniromnet variable to enable launching without specifiying a full path. For Linux/Bash and GitHub repo cloned to `~/ydbwork/ydb` you can use the following command:

```
echo "alias ya='~/ydbwork/ydb/ya'" >> ~/.bashrc
source ~/.bashrc
```

Run `ya` without parameters to get help:

```
$ ya
Yet another build tool.

Usage: ya [--precise] [--profile] [--error-file ERROR_FILE] [--keep-tmp] [--no-logs] [--no-report] [--no-tmp-
dir] [--print-path] [--version] [-v] [--diag] [--help] <SUBCOMMAND> [OPTION]...

Options:
...

Available subcommands:
...
```

You can get detailed help on any subcommand launching it with a `--help` flag, for instance:

```
$ ya make --help
Build and run tests
To see more help use -hh/-hhh

Usage:
  ya make [OPTION]... [TARGET]...

Examples:
  ya make -r               Build current directory in release mode
  ya make -t -j16 library  Build and test library with 16 threads
  ya make --checkout -j0   Checkout absent directories without build

Options:
...
```

The `ya` script downloads required platform-specific artifacts when started, and caches them locally. Periodically, the script is updated with the links to the new versions of the artifacts.

Setup IDE

If you're using IDE for development, there's a command `ya ide` which helps you create a project with configured tools. The following IDEs are supported: goland, idea, pycharm, venv, vscode (multilanguage, clangd, go, py).

Go to the directory in the source code which you need to be a root of your project. Run the `ya ide` command specifying the IDE name, and the target directory to write the IDE project configuration in a `-P` parameter. For instance, to work on the YQL library changes in vscode you can run the following command:

```
cd ~/ydbwork/ydb/library/yql
ya ide vscode -P=~/ydbwork/vscode/yqllib
```

Now you can open the `~/ydbwork/vscode/yqllib/ide.code-workspace` from vscode.

Build a target

There are 3 basic types of targets in `Ya Make`: Program, Test, and Library. To build a target run `ya make` with the directory name. For instance, to build a YDB CLI run:

```
cd ~/ydbwork/ydb
ya make ydb/apps/ydb
```

You can also run `ya make` from inside a target directory without parameters:

```
cd ~/ydbwork/ydb/ydb/apps/ydb
ya make
```

Run tests

Running a `ya test` command in some directory will build all test binaries located inside its subdirectories, and start tests.

For instance, to run YDB Core small tests run:

```
cd ~/ydbwork/ydb
ya test ydb/core
```

To run medium and large tests, add options `-tt` and `-ttt` to the `ya test` call, respectively.

Manage YDB Releases

There are two products based on the source code from the YDB repository with independent release cycles:

- YDB server
- YDB command-line interface (CLI)

YDB Server Release Cycle

Release Numbers and Schedule

YDB server version consists of three numbers separated by dots:

1. The last two digits of calendar year of the release
2. Major release ordinal number in a given year
3. Minor release ordinal number for a given major release

Thus, YDB server major version is a combination of the first two numbers (for example, `23.3`), and the full version is a combination of all three (for example, `23.3.5`).

YDB server release schedule typically includes 4 major releases per year, so the release `YY.1` is the first one, and `YY.4` is the last one for a year `YY`. The number of minor releases is not fixed, and may vary from one major release to another.

Compatibility

YDB maintains compatibility between major versions to ensure a cluster can operate while its nodes run two adjacent major versions of the YDB server executable. You may refer the Updating YDB article to learn more about the cluster upgrade procedure.

Given the above compatibility target, major releases go in pairs: odd numbered releases add new functionality switched off by feature flags, and even numbered releases enable that functionality by default.

For instance, release `23.1` comes with the new functionality switched off. It can be incrementally rolled out to a cluster running `22.4`, without downtime. As soon as the whole cluster runs `23.1` nodes, you can manually toggle feature flags to test new functionality and later further upgrade it to `23.2` to fully leverage this new functionality.

Release Branches and Tags

A release cycle for an odd major release starts by a member of a YDB Release team forking a new branch from the `main` branch. Major release branch name starts with prefix `stable-`, followed with the major version with dots replaced by dashes (for example, `stable-23-1`).

A release cycle for an even major release starts by branching from the preceding odd major version branch. The branch follows the same naming convention.

All major version releases, both odd and even, go through the comprehensive testing process producing a number of minor versions. Each minor version is created by tagging a relevant commit of the release branch with a full version number. So, there can be tags `24.1.1`, `24.1.2` etc. on the `stable-24-1` branch. As soon as a minor version proves its quality, we consider it as stable, and register a Release on GitHub linked to its tag, add it to downloads and changelog documentation pages, etc. Thus, there can be more that one stable release for a major version.

Testing

Release testing is iterative. Each iteration starts by assigning a tag to a commit on the release branch, specifying a minor version to be tested. For example, the minor version tag `23.3.5` marks a 5th testing iteration for the major release `23.3`.

A tag can be considered to be either "candidate" or "stable". Initially, the first tag is created in the release branch right after its creation. This tag is considered as "candidate".

During a testing iteration, code from release branches undergoes an extensive testing including deployment on UAT, prestable, and production environments of companies using YDB. To perform such testing, YDB code from a GitHub release tag is imported into the corporate context of a given user, following its internal policies and standards. Then it's built, deployed to the necessary environments, and thoroughly tested.

Based on a list of uncovered problems, the YDB Release team decides if the current minor release can be promoted to be called "stable", or a new testing iteration must be started over with a new minor release tag. In fact, as soon as a critical problem is discovered during testing, developers fix it in the `main` branch, and backport changes to the release branch right away. So, by the time testing iteration finishes, there will be a new tag and a new testing iteration if there are some new commits on top of the current tag.

Stable Release

If testing iteration proves the quality of a minor release, the YDB Release team prepares the release notes, and publishes the YDB server release on both the GitHub Releases and Downloads pages, therefore declaring it as "stable".

YDB CLI (Command-Line Interface) Release Cycle

Release Numbers and Schedule

YDB CLI version consists of three numbers separated by dots:

1. Major release ordinal number (currently, `2`)
2. Minor release ordinal number for a given major release
3. Patch number

For example, `2.8.0` is the 2nd major release, 8th minor, without additional patches.

There's no schedule for the YDB CLI minor releases, a new release comes as soon as there's some new valuable functionality. Initially, every new minor release has `0` as a patch number. If there are critical bugs found in that version, or some minor part of functionality did not catch it as planned, a patch can be released, incrementing only the patch number, like it was for `2.1.1`.

In general, release cycle for YDB CLI is much simpler and shorter than for the server, producing more frequent releases.

Release Tags

Tags for YDB CLI are assigned on the `main` branch by a member of the YDB Release team after running tests for some revision. To distinguish from the YDB server tags, YDB CLI tags have a `CLI_` prefix before the version number, for example CLI_2.8.0.

Stable Release

To declare a YDB CLI tag as stable, a member of the YDB Release team prepares the release notes, and publishes the release on the GitHub Releases and Downloads pages.

Contributing to YDB documentation

YDB follows the "Documentation as Code" approach, meaning that the YDB documentation is developed using similar techniques and tools as its main C++ source code.

The documentation source code consists of Markdown files and YAML configuration files located in the ydb/docs folder of the primary YDB GitHub repository. The compiler for this source code is an open-source tool called Diplodoc. See its documentation for details on its Markdown syntax flavor, configuration options, extensions, and more.

The process of suggesting changes to the documentation source code is mostly similar to changing any other YDB source code, so most of Development process: working on a change for YDB applies. The main additional considerations are:

- Extra precommit checks run for pull requests to the documentation. One of these checks posts a comment with a link to an online preview of the changes or a list of errors.
- The code review process includes additional steps. See Review process for YDB documentation and YDB documentation style guide.
- For small changes like fixing a typo, you can use the "Edit this file" feature in the GitHub web interface. Each documentation page has an "Edit on GitHub" link (represented by a pencil icon in the top-right corner) that directs you to the page's source code in the GitHub web interface.

After a pull request to the documentation is merged into the `main` branch, the CI/CD pipeline automatically deploys it to the YDB website. Documentation is also automatically deployed for stable YDB server versions from `git` branches named `stable-*`, where these versions are developed. If C++ code and documentation for a feature were committed separately and a new stable branch was forked between these commits, backporting some changes to the stable branch might be necessary. The same applies to typo fixes and other "bug fixes" to the documentation content. See Manage YDB Releases for more details on the YDB release process.

See also

- YDB Documentation Structure
- YDB documentation genres
- GitHub documentation
- Git documentation

Review process for YDB documentation

Building on the high-level overview in Contributing to YDB documentation, this article dives deeper into what happens during the documentation pull request review stage.

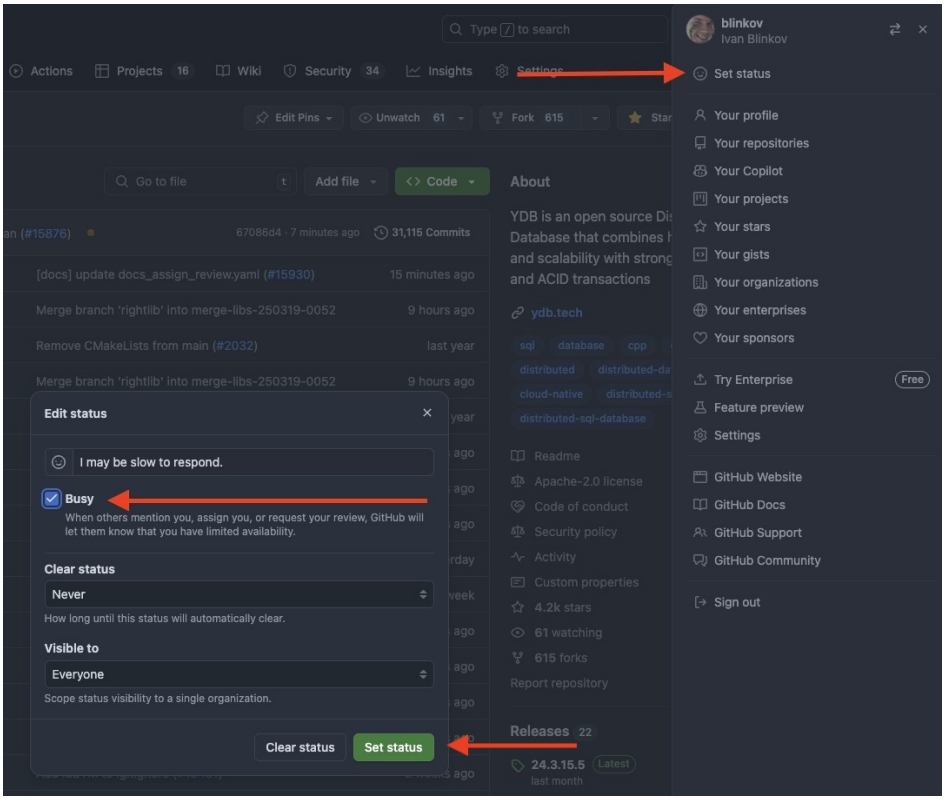
Roles

- **Author** — the person suggesting a change to the documentation.
- **Primary reviewer** — the person thoroughly inspecting the suggested change according to the checklist.
- **Final reviewer** — the person double-checking the change using the same checklist and approving the pull request on GitHub.

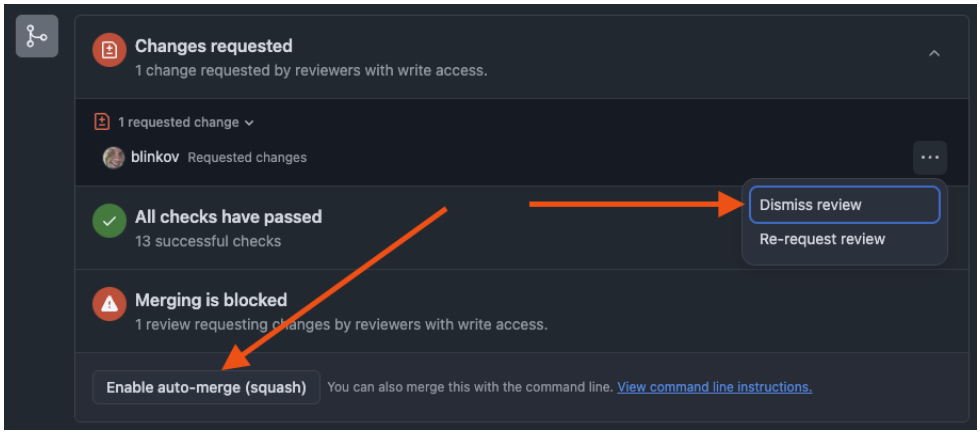
Process

1. **Author** opens a GitHub pull request with suggested changes to the ydb/docs folder. Following the style guide from the beginning makes the review process smoother.
2. **Author** ensures the pull request is in a reviewable state by meeting all of the following criteria:
   2.1. The pull request has `* Documentation` as the only changelog category. If done correctly, automation marks the pull request with a "documentation" label.
   2.2. The pull request is *not* marked as a draft.
   2.3. The suggested change builds successfully, and automation has posted a comment with a preview link (instead of errors). The preview shows the changed content as expected.
3. *(optional)* **Author** shares a link to the pull request in a community or documentation-related chat for extra visibility.
4. A **primary reviewer** gets automatically assigned or picks the pull request from the inbound list via the "assign yourself" button, and then provides the initial set of feedback and suggestions.

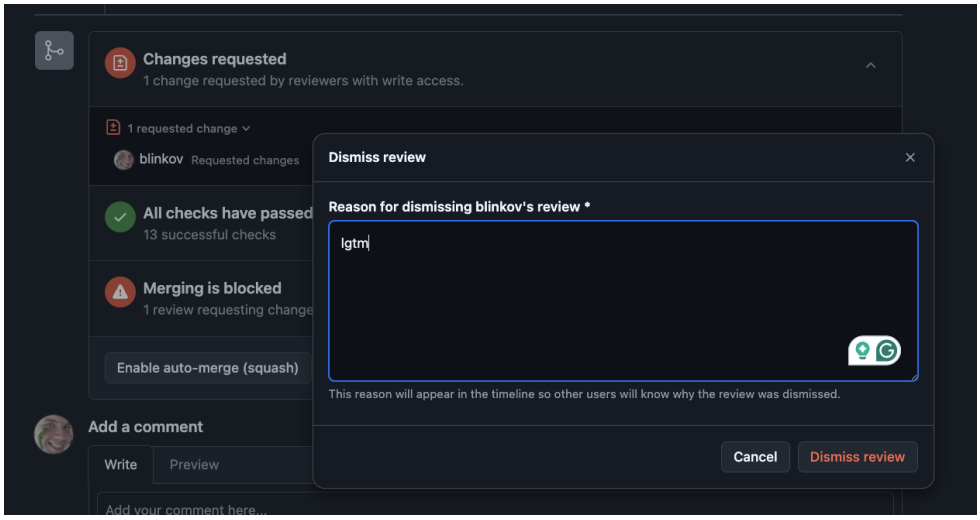
   Temporarily excluding yourself from automatic assignment if you're a primary reviewer



5. The **author** and **primary reviewer** iterate until the suggested change passes the checklist. The **primary reviewer** provides feedback via comments on the pull request, while the **author** addresses them. The expected turnaround time for each review iteration is two business days, up to a few weeks in case of force majeure. Additionally, the **primary reviewer** periodically checks pull requests assigned to them and clarifies the status with authors if nothing happens for several business days (in GitHub comments, plus in personal communication if possible).
6. Once the **primary reviewer** confirms that the pull request meets the checklist requirements, they:
   - Enable auto-merge for the pull request. If the target branch of the pull request differs from `main`, the auto-merge feature will not be available: in this case, the **primary reviewer** should manually click the "Squash and merge" button after approval by the **final reviewer** in step 7.



- Dismiss their stale review with a "lgtm" comment.



- The **primary reviewer** passes the process to the **final reviewer** for additional review with a fresh set of eyes. This typically happens via a direct personal message, but since whole step 6 is done simultaneously, this event is still visible in the pull request itself through the previous sub-steps (enabled auto-merge and/or dismissed review from the **primary reviewer**).

7. Depending on the **final reviewer**'s verdict:
   - If the **final reviewer** approves, the pull request starts meeting one of the mandatory conditions for merging. Thus, if the build is still passing, GitHub's auto-merge likely merges the pull request automatically. Otherwise, any issues must be addressed manually.
   - If the **final reviewer** provides additional feedback or suggestions, the process returns to step 5.
8. If auto-merge was not enabled in step 6, the **primary reviewer** clicks the "Squash and merge" button.
9. After the content lands in the `main` branch, it will be automatically published to the official YDB website via CI/CD.
10. YDB documentation is multilingual, and **authors** are expected to provide synchronized changes for all supported languages (currently English and Russian), if applicable. If the **author** does not know all required languages, using an LLM or machine translation is acceptable. The translation timing depends on complexity:
    - For simple changes, it's usually best to translate at the beginning and go through the review process with a single pull request covering all languages.
    - For complex changes likely requiring multiple review iterations, it's acceptable to first complete the review process in one language and then start a separate translation pull request after approval.

Checklist

If the PR is new content for the main branch

- [ ] The text is understandable for the article's target audience.
- [ ] The text is technically accurate.
- [ ] The text is grammatically correct, with no punctuation, spelling, or typographical errors.
- [ ] Terminology is consistent. The first mention of each term used in the article is a link to its explanation in the YDB glossary or a well-known source like Wikipedia.
- [ ] Each new article is correctly placed in the documentation structure.
- [ ] Each article follows a single genre and aligns with its place in the documentation structure.
- [ ] Each new article includes links to all relevant existing documentation pages, either inline or in a "See also" section.
- [ ] Relevant existing articles are updated with links to new articles.
- [ ] All new articles are listed in YAML files with table of contents and their folder's `index.md`.
- [ ] All renamed or moved articles are reflected in redirects.yaml.
- [ ] The article's voice, tone, and style match the rest of the documentation or, at a minimum, remain consistent within the article.

> **Tip**
>
> This checklist is a condensed version of YDB documentation style guide and serves as a reminder. Feel free to copy-paste it into the pull request description and check off items as you go. Refer to the full style guide for initial understanding and additional details.

If the PR is a backport to a stable branch

The documentation content is independently published on the website from multiple Git branches. Similar to source code development, `main` is the documentation for upcoming releases, while the documentation for each specific YDB release is published from a stable branch. For example, YDB v25.1 corresponds to the `stable-25-1` branch. If the documentation content needs to be cherry-picked from `main` to a stable branch, a separate PR is created with `stable-***` as the target branch. The review process for such PRs is different, as the content has already been reviewed when it was merged into `main`. Use the following checklist instead:

- [ ] The backport PR mentions identifiers or links to one or more PRs where this content was introduced in the `main` branch.
- [ ] The content of the backport PR matches the original PRs to `main`. The content has been correctly moved if there was a structural refactoring between branches.
- [ ] The PR compiles correctly, the preview generates without errors or warnings, and the content appears in the correct locations.
- [ ] There are no merge artifacts such as duplicate content, `>>>>>>>`-like Git markers, or similar issues.
- [ ] The feature appears to be available in the corresponding YDB release. Double-check with the author or feature owner if unsure.

What documentation review is not

Testing

Documentation review is not a replacement for testing. If the documentation includes instructions, the **author** is responsible for ensuring their correctness, implementing automated tests to maintain accuracy over time, etc.

The **primary reviewer** and/or **final reviewer** may choose to follow the instructions to see how they work in practice, but this is not mandatory.

Technical design review

Documentation review is not a technical design review. Documentation is typically written for mostly completed features, so significant changes to product behavior are rarely possible at this stage. However, the **primary reviewer** and/or **final reviewer** may highlight any inconsistencies, odd behaviors, or usability concerns. It is the **author**'s responsibility to address them immediately if possible or consider them for future iterations of the feature being described.

See also

- GitHub documentation
- Git documentation

YDB documentation style guide

The YDB documentation style guide is designed to help writers create clear, consistent, and developer-friendly documentation.

Core principles

- **Target audience understanding:**
  - Before starting a new article or improving an existing one, take a moment to narrow down its target audience. Consider their specific role (application developers, DevOps engineers, security engineers, etc.), technical background, and familiarity with the subject and YDB in general.
  - Ensure the text is understandable to the article's intended audience.
- **Clear language:**
  - Use plain, simple language that directly communicates ideas.
  - Avoid complicated phrases that might be challenging for non-native speakers.
  - Avoid jargon and slang unless they are industry-standard.
- **Consistent terminology:**
  - For each uncommon term in the article, ensure that the first usage links to an explanation:
    - For YDB-specific terms, link to the YDB glossary. If the glossary lists multiple synonyms for the term, use the primary one from the glossary header. If the glossary entry is missing, add it.
    - For generic terms that some readers might not know, link to their Wikipedia page or a similar well-known source. Avoid linking to blog posts, social media, or random third-party articles.
  - If there's an abbreviation, spell it out in full the first time it appears in the article and explicitly show the abbreviation in parentheses. For example, Structured Query Language (SQL).
- **Structured content:**
  - Each article should have only *one* goal for the reader to achieve. If the text pursues multiple goals, it likely needs to be split into multiple articles. If possible, explicitly state this goal at the beginning of the article and reiterate it at the end.
  - Each article should follow only one genre. If not, it likely needs to be split into multiple articles.
  - Organize articles with clear headings, subheadings, bullet lists, notes, tables, and code blocks to help readers quickly navigate the information.
  - Follow the overall documentation structure when creating new articles.

Voice and tone

- **Conversational.** Aim for a tone that feels like a knowledgeable friend explaining concepts in an approachable manner.
- **Friendly and respectful.** Maintain a balance of professionalism and warmth that invites engagement.
- **Inclusive language.** Write in a neutral and respectful way, avoiding biased or exclusive language.
- **Context-dependent.** Adjust your tone based on the type of content. For tutorials, be more encouraging; for reference material, be more concise.
- **Active voice.** Prefer active constructions to ensure clarity and immediacy, making instructions easier to follow.

Language-specific

Ensure that text follows proper language rules with no typos, grammar, punctuation, or spelling issues. Additionally, follow these YDB-specific rules.

English-only

- Use the serial comma where appropriate.
- Use title case for headers, preferably following the Chicago Manual of Style rules.
- Use double quotes (`"`).

Russian-only

- Use `ё` where appropriate.
- Use guillemets for quotes (`«` and `»`).
- Use `—` for long dashes.
- Use semicolons for bullet and numbered lists where appropriate, i.e., for all items except the last one, unless items contain multiple sentences.
- Do *not* use title case for headers.

Formatting and structure

- **Clear hierarchy.** Use headings and subheadings to define sections and make the document easy to scan.
- **Lists and bullet points.** Use bullet points or numbered lists to break down steps, features, or recommendations.
- **Visual highlights.** If there's an important warning, information, or tip, highlight it using a `{% note info %} ... {% endnote %}` tag. For smaller inline highlights, sparingly use **bold** or *italic*.
- **Code and samples.** When including code, use proper syntax highlighting, formatting, and comments to ensure readability. Show example output for queries and CLI commands. Use `code blocks` for everything likely to appear in a console or IDE, but not for visual highlights.
- **File naming.** Use dashes instead of underscores for spaces in new file and folder names (e.g., `new-folder/new-file.md` instead of `new_folder/new_file.md`). However, underscores are acceptable in keywords that should be written specifically with underscores, such as in setting names.
- **Linking and references.**
  - Provide clear and descriptive links to related resources or additional documentation.
  - Links internal to documentation should always include a `.md` suffix and always be relative (that is, no `https://ydb.tech` prefix); otherwise, link consistency checking doesn't work, and they will eventually start leading to 404 errors. Internal links shouldn't contain `?version=...` , `?revision=...` , or similar parameters.
  - Instead of repeating the target's header for internal links, use `[{#T}](path/to/an/article.md)` .
- **No copy-pasting.** If a piece of information needs to be displayed more than once, create a separate Markdown file in the `_includes` folder, then add it to all necessary places via the `include` feature instead of duplicating content by copying and pasting.
- **Markdown syntax style.** YDB documentation uses an automated linter for Markdown files. Refer to .yfmlint for the up-to-date list of enforced rules.

- **Variable usage.** The YDB documentation has a configuration file, presets.yaml, that lists variables to prevent typos or conditionally hide content. Use these variables when appropriate, particularly `YDB` instead of `YDB`.
- **Diagrams.** Prefer built-in Mermaid diagrams when possible. If using an external tool, submit the source file in the same `_assets` folder near the image for future edits. Ensure diagrams look good in both light and dark modes.
- **Proper article placement.** New articles should be correctly placed in the documentation structure.
- **Genre consistency.** Articles should not mix multiple genres, and the genre should match the article's place in the documentation structure.

Documentation integration

- **Cross-referencing.** Include links to all relevant existing documentation pages, either inline or in a "See also" section at the end.
- **Bidirectional linking.** Update relevant existing articles with links to new articles.
- **Index inclusion.** Mention all articles in table of contents (`toc_i.yaml` or `toc_p.yaml`) and their folder's `index.md`.
- **Source code links.** Link directly to source files on GitHub when relevant. If the target is likely to change significantly over time, use a link to a specific commit or stable branch.
- **Glossary links:**
  - When a YDB-specific term is mentioned in an article for the first time, make it a clickable link to the related glossary entry.
  - If there's a separate detailed article covering the same topic as a glossary term, link to it from the glossary term description.

Usage and flexibility

- **Guidelines, not rigid rules.** This style guide offers recommendations to improve clarity and consistency, but flexibility is allowed when deviations benefit the content.
- **Supplementary resources.** Writers are encouraged to consult external style guides (e.g., The Chicago Manual of Style, Merriam-Webster) when in doubt.
- **Updates.** This guide is meant to evolve with the rest of the content over time. If you contribute to YDB documentation frequently, check back periodically for updates.

Things to avoid

- Jargon and slang.
- Using "we".
- Jokes and other emotional content.
- Excessively long sentences.
- Unrelated topics and references, such as culture, religion, or politics.
- Overly detailed implementation specifics (except in YDB Development, YDB Server changelog, and Videos).
- Reusing third-party content without written permission or a compatible open-source license explicitly allowing it.
- HTML inside Markdown, except as a workaround for missing visual styles.

See also

- Contributing to YDB documentation
- Review process for YDB documentation
- YDB Documentation Structure
- YDB documentation genres

YDB Documentation Structure

This article complements YDB documentation style guide. It explains the current top-level folders of the documentation and what kind of content belongs in each. As a rule of thumb, most top-level sections focus either on a specific target audience (if named "For ...") or on a specific genre.

- Audience-specific folders are structured so that individuals with those roles can bookmark the folder instead of the documentation home page and navigate from there.
- Genre-specific folders are mainly designed to be found as needed through the built-in documentation search, third-party search engines, or LLMs.

Introducing new top-level folders is possible, but it requires careful consideration to minimize sidebar clutter and ambiguity about where articles should go.

General Rules

- YDB documentation is multilingual, and the file structure for each language must be consistent for the language switcher to function correctly. The file content should also be as close as possible between languages. The only exception is the Public materials folder, which intentionally contains different content for different languages. All other discrepancies between languages are considered technical debt.
- Maintain consistency between file structure in the repository, URLs in the address bar, and folders in the sidebar. Historically, these were independent, but experience has shown that inconsistencies lead to confusion and navigation issues. The documentation is gradually transitioning to a unified structure.
- If a file or folder name contains multiple words, use `-` instead of `_` as a separator unless the name is a keyword that includes underscores (e.g., configuration section names).
- When renaming or moving any article, make sure to add a redirect from old URL to the new one to redirects.yaml.

List of top-level folders

- **Quick start.** A single guide for beginners explaining how to set up a single-node YDB server and run initial queries.
- **Concepts.** A high-level theoretical overview of YDB as a technology, covering its features, terminology, and architecture. Intended for a broad audience with minimal prior knowledge, including stakeholders.
- **For DevOps.** A folder for DevOps engineers responsible for setting up and running YDB clusters. Most content consists of practical guides for specific cluster-related tasks. Since YDB supports multiple deployment options, guides that differ based on deployment method are placed in respective subfolders (Ansible, Kubernetes, or Manual), each following a consistent internal structure. Role-specific theoretical information is also included here.
- **For Application Developers.** A folder for application developers working with YDB. Primarily consists of practical guides and some theory.
- **For Security Engineers.** A folder for security engineers responsible for securing and auditing YDB clusters and applications that interact with them. Contains mostly practical guides and some role-specific theory.
- **For Contributors.** A folder for YDB core team members and external contributors. It explains various YDB development processes and provides deeper insights into how some components work. Mostly theory with some practical guides.
- **Reference.** A detailed reference section covering various aspects of YDB, designed to be found as needed or discovered via inbound links. The primary goal is completeness so that any topic can be located through exact keyword matches or descriptions. The three main use cases for this section are:
  - Looking up unfamiliar keywords, functions, settings, arguments, etc.
  - Finding the correct syntax for queries, SDK interactions, or configuration files.
  - Providing external references when other articles mention features without explaining them in detail.
- **Recipes.** Mini-guides explaining specific tasks with YDB, often with examples and code snippets. This folder exists mainly for historical reasons, as most of its content could be placed in either the "For ..." folders or "Questions and answers."
- **Troubleshooting.** A mix of theory on potential issues related to YDB and applications working with them, as well as practical guides for diagnosing and resolving them.
- **Questions and answers.** A StackOverflow-style section with frequently asked questions. Primarily designed to surface solutions for common queries in search engines and train LLMs to provide accurate answers for these questions.
- **Public materials.** A collection of links to videos and articles about YDB. Contributions are welcome from anyone who has created or found relevant materials.
- **Downloads.** A collection of links to download YDB binaries.
- **Changelog.** Release notes for each new version of the YDB server and other related binaries.

YDB documentation genres

This article complements YDB documentation style guide by describing the main genres used in YDB documentation. Understanding these genres helps contributors place new content in the appropriate section and maintain a consistent structure.

Theory

Primary goal for the reader: build a solid knowledge foundation by understanding the fundamental concepts, architecture, and principles behind YDB.

- Introduces key concepts and terminology
- Explains how things work both from the user's perspective and under the hood
- Provides high-level overviews of system components
- Helps users understand the "why" behind design decisions
- Can be targeted at either a broad audience with minimal prior knowledge or a specific role
- Can include diagrams and other visualizations to help convey information

Theory documentation is primarily found in the "Concepts" section but also appears in role-specific folders when the theoretical information is relevant only to a particular audience.

Guide

Primary goal for the reader: accomplish a specific practical task or implement a particular solution with YDB by following instructions.

Guides are practical, step-by-step instructions that help users accomplish a specific goal with YDB. Each article in this genre:

- Provides a clear goal and sequential instructions to achieve it
- Includes concrete examples and commands
- Focuses on practical implementation
- Addresses specific use cases or scenarios
- Can include screenshots to illustrate steps

Guides are primarily found in role-specific folders like "For DevOps", "For Application Developers", and "For Security Engineers", as well as in the "Troubleshooting" section.

Reference

Primary goal for the reader: find additional information about a specific niche topic related to YDB.

Reference documentation provides comprehensive, detailed information about YDB components, queries, APIs, configuration options, CLI commands, UI pages, and more. This genre:

- Aims for completeness and precision
- Serves as a lookup resource for specific details
- Documents all available options, parameters, and settings
- Is organized for quick information retrieval via a search engine or LLM
- Includes syntax, data types, parameters, return values, defaults, configuration, and examples

Reference documentation is designed to be found as needed and is the most detailed level of documentation. It's particularly useful when users need specific information about functions, settings, or keywords. This content is primarily found in the "Reference" section.

FAQ

Primary goal for the reader: quickly find answers to common questions encountered when working with YDB.

Frequently Asked Questions (FAQ) documentation answers common questions about YDB in a direct question-and-answer format. This genre:

- Addresses specific, commonly asked questions
- Provides concise, focused answers
- Is organized by topic or category
- Helps users quickly find solutions to common problems
- Is optimized for search engine or LLM discovery

FAQ content is primarily found in the "Questions and answers" section and is designed to help users who are searching for specific solutions to common situations.

Recipe

Primary goal for the reader: implement a specific, focused solution to a common issue or use case with YDB.

Recipes are short, focused mini-guides that demonstrate how to accomplish specific tasks with YDB. This genre:

- Provides concise solutions to specific problems
- Includes code snippets and examples
- Focuses on practical implementation
- Is more targeted and narrower in scope than full-fledged guides
- Often follows a problem-solution format

Recipes are primarily found in the "Recipes" section, though similar content may also appear in role-specific folders.

Release notes

Primary goal for the reader: stay informed about new features, improvements, bug fixes, and breaking changes in YDB releases.

Release notes document changes, improvements, and fixes in each new version of YDB. This genre:

- Lists new features and enhancements
- Documents bug fixes and resolved issues
- Highlights breaking changes and deprecations

- Provides upgrade instructions when necessary
- Is organized chronologically by version number

Release notes are found in the "Changelog" section and help users understand what has changed between versions and decide whether to upgrade.

Collection of links

Primary goal for the reader: discover additional resources, learning materials, and external content related to YDB.

Collections of links provide curated lists of resources related to YDB. This genre:

- Aggregates related external or internal resources
- Provides brief descriptions of each linked resource
- May be organized by topic, format, or relevance
- Helps users discover additional learning materials
- Can include videos, articles, downloads, and other content

Collections of links are primarily found in the "Public materials" and "Downloads" sections, serving as gateways to external resources about YDB.

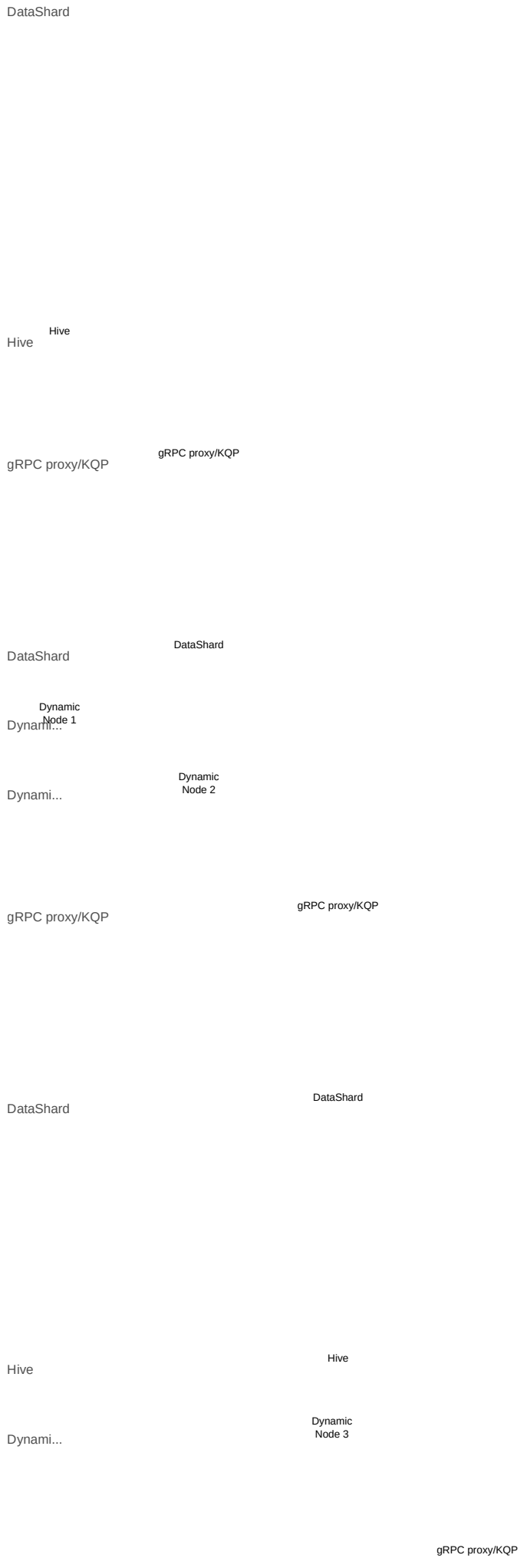
General YDB schema

An approximate general YDB schema is shown below.

gRPC proxy/KQP
gRPC proxy/KQP

DataShard
DataShard

DataShard


DataShard
Hive
Hive
gRPC proxy/KQP
gRPC proxy/KQP
DataShard
DataShard
Dynamic
Node 1
Dynami...
Dynami...
Dynamic
Node 2
gRPC proxy/KQP
gRPC proxy/KQP
DataShard
DataShard
Hive
Hive
Dynami...
Dynamic
Node 3
gRPC proxy/KQP

gRPC proxy/KQP
BSC
Static...
gRPC proxy/KQP
CMS
Static...
NodeBroker
Tenant 1
Tenant 2
Static
BSC
Static
Node 1
gRPC proxy/KQP
CMS
Static
Node 2
NodeBroker
Tenant 1
Tenant 2
Static
BlobStorage

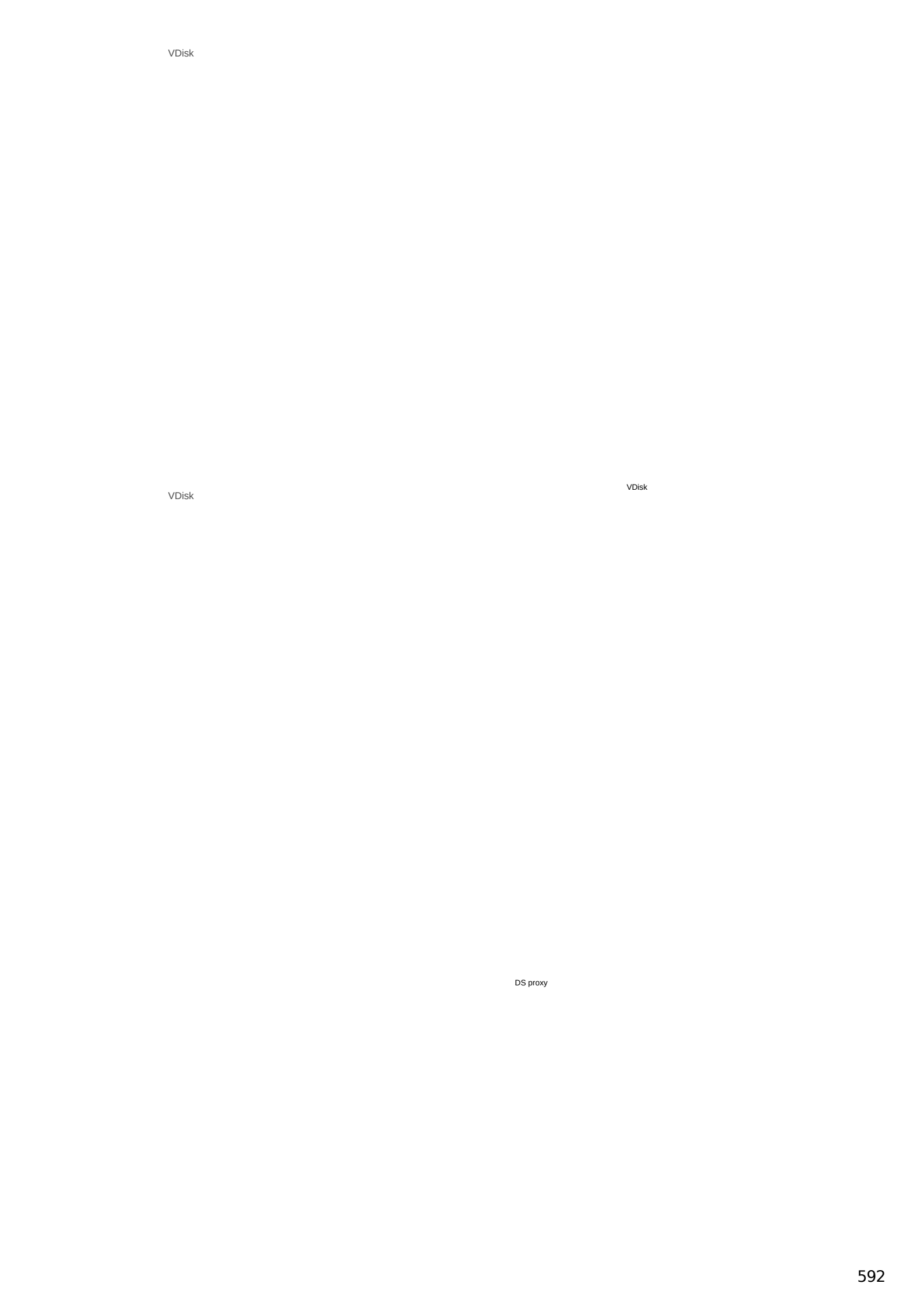
BlobStorage
NodeWarden
NodeWarden
NodeWarden
NodeWarden
PDisk

PDisk

PDisk

PDisk

VDisk


VDisk
VDisk
VDisk
DS proxy

DS proxy

DS proxy

DS proxy

NodeWarden

NodeWarden

NodeWarden

NodeWarden

NodeWarden

NodeWarden

DS proxy

DS proxy

DS proxy

Nodes

A single YDB installation consists of a *cluster* that is divided into *nodes*. A node is an individual process in the system, usually `ydbd`. Each node is part of the cluster and can exchange data with other nodes via *Interconnect*. Each node has its own ID, typically called `NodeId`. The `NodeId` is a 20-bit integer equal to or greater than 1. `NodeId` 0 is reserved for internal purposes and usually indicates the current node or no node at all.

A number of services run on each node and are implemented via *actors*.

Nodes can be either static or dynamic.

A static node configuration, which includes their complete list with the address for connecting via Interconnect, is stored in a configuration file and is read once when the process starts. The set of static nodes changes very rarely, typically during cluster expansion or when moving nodes from one physical machine to another. To change the set of static nodes, you must apply the updated configuration to **every** node and then perform a rolling restart of the entire cluster.

Dynamic nodes are not known in advance and are added to the system as new processes are started. This may occur, for example, when new tenants are created in YDB installations as a database. When a dynamic node is registered, its process first connects to one of the static nodes via gRPC, transmits information about itself through a special service called *Node Broker*, and receives a `NodeId` to use when logging into the system. The mechanism for assigning nodes is somewhat similar to DHCP in the context of distributing IP addresses.

Tablets

Special microservices called tablets run on each node. Each tablet is a singleton that has a specific type and ID. It means that only one tablet with a specific ID can run in the cluster at any given time. A tablet can launch on any suitable node. A generation is an important property of a tablet that increases with each subsequent launch.

> **Note**
>
> The distributed nature of the system and various issues, such as network partitioning problems, may result in a situation when the same tablet is actually running on two different nodes simultaneously. However, the distributed storage guarantees that only one of the duplicate tablets will successfully complete operations that change its state, and that the generation in which each successful operation runs will not decrease over time.

For cluster-level system tablets, the node on which the tablet runs is chosen by a Bootstrapper, which implements distributed consensus. User tablets are managed by a special tablet called Hive. Hive ensures that all tablets are running, distributes tablets across nodes, and manages tablet channels between storage groups.

You can find out on which node the tablet in the current generation is running through the *StateStorage* service. To send messages to tablets, use a special set of libraries named *tablet pipe*. With this, knowing the ID of the target tablet, you can easily send the desired message to it.

A tablet can be divided into two parts: the basic tablet and the user logic.

The basic tablet is a set of tables, each of which may consist of one or more key columns of an arbitrary type and a set of data columns. Each table may have its own schema. Additionally, tables can be created and deleted while the tablet is running. The basic tablet interface allows you to perform read and update operations on these tables.

User logic is located between the basic tablet and the user, allowing you to process specific requests for this type of tablet, reliably saving changes to distributed storage. A running tablet typically uses a template that stores all data in memory, reads it only at the start, and synchronously changes the data in memory and in storage after a successful commit.

How does a tablet store data, and what are they like?

A basic tablet is an LSM tree that holds all of its table data. One level below the basic tablet is distributed storage, which, roughly speaking, is a KeyValue storage that stores binary large objects (blobs). A *BLOB* is a binary fragment from 1 byte to 10 MB in size, which has a fixed ID (usually called *BlobId* and of the `TLogoBlobID` type) and contains related data. The storage is immutable, meaning that only one value corresponds to each ID, and it cannot change over time. You can write and read a blob and then delete it when it is no longer needed.

To learn more about blobs and distributed storage, see YDB distributed storage.

For distributed storage, blobs are an opaque entity. A tablet can store several types of blobs. The most frequently written blob is a (recovery) log blob. A tablet's log is arranged as a list of blobs, each containing information about changes made to the tables. When run, the tablet finds the last blob in the log and then recursively reads all related blobs following the links. The log may also contain links to snapshot blobs, which contain data from multiple log blobs after a merge (the merge operation in the LSM tree).

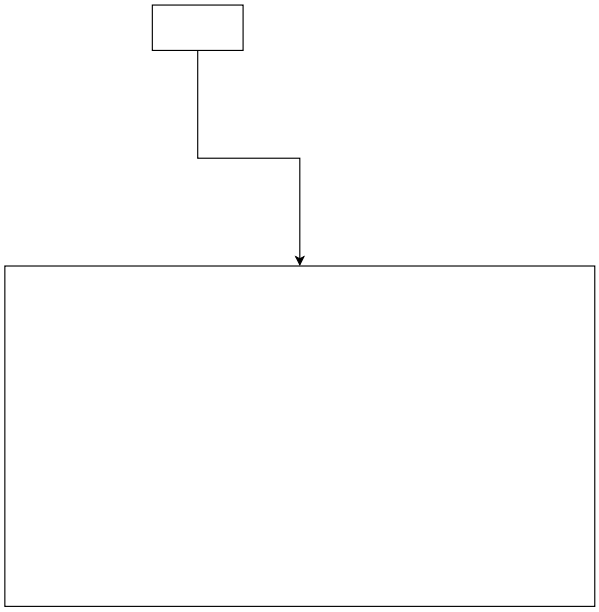
The tablet writes blobs of different types to different *channels*. A channel specifies the branch of storage in which to store blobs and performs various functions, such as:

1. Selecting a storage type (different channels may be linked to different types of storage devices: SSD, HDD, or NVMe).
2. Load balancing, as each channel has a limit on IOPS, available space, and bandwidth.
3. Specifying the data type. When restoring the log, only the blobs from the null channel are read, allowing you to distinguish them from other blobs.

Tablet channel history

As mentioned previously, each group has a constant amount of data that fits into it and shares the bandwidth's throughput and the number of operations per second among all consumers. The load on tablets may vary. As a result, a group may become overloaded and it will be necessary to switch writes to another group. To address this, the concept of history is introduced, which allows you to determine the group that a blob is written to based on its channel and generation.

This mechanism works as follows:



TabletId

TabletId

Channel

Channel

Generation

Generation

Step

Step

Cookie

Cookie

BlobSize

BlobSize

PartId

PartId

BlobId

BlobId

64

64

8

8

32

32

32

32

24

24

28 (26)

28 (26)

4

4

TTabletStorageInfo

Channel 0

Channel 1

Channel 2

Channel 2

Channel N

Channel N

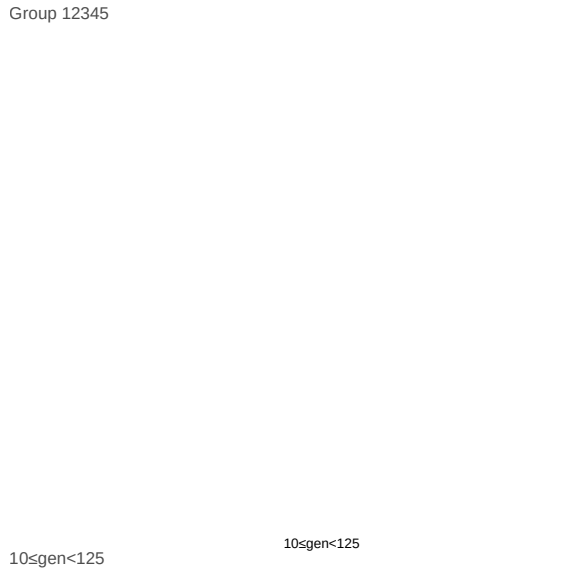
TTabletChannelInfo

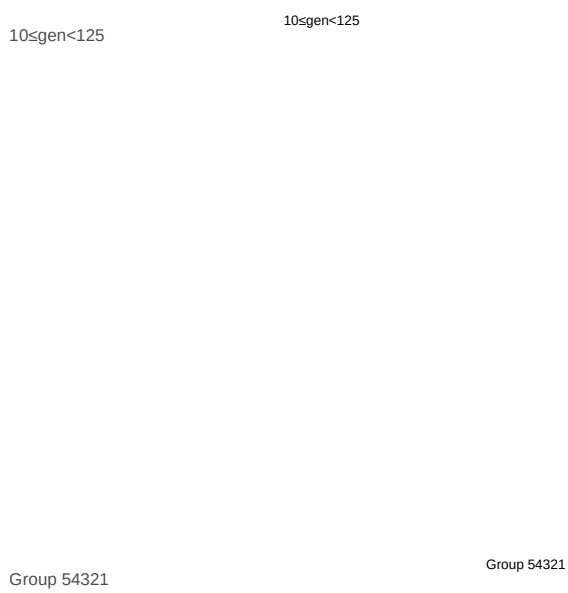
TTabletChannelInfo

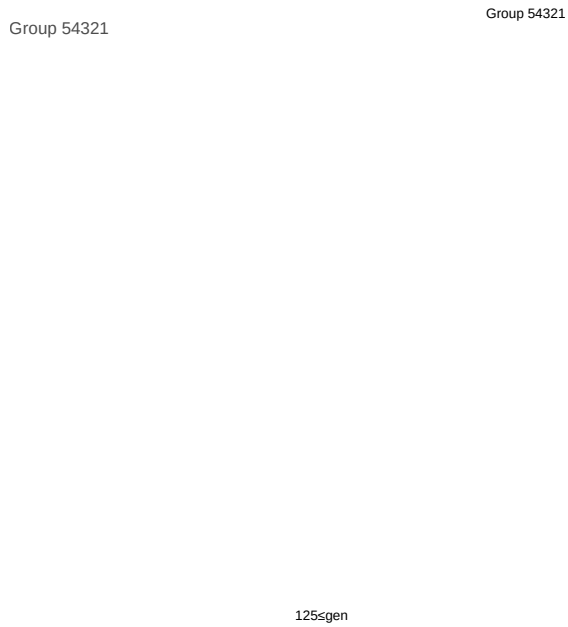
0≤gen<10

0≤gen<10

Group 12345


Group 12345
10≤gen<125


10≤gen<125
Group 54321


Group 54321
125≤gen

For each channel, the `TTabletStorageInfo` structure contains the `TTabletChannelInfo` substructure with generation ranges and the group number corresponding to each range. The ranges are strictly adjacent to each other, with the last range being open. Group numbers may overlap in different ranges and even across different channels, which is legal and quite common.

When writing a blob, a tablet selects the most recent range for the corresponding channel, as a write is always performed on behalf of a tablet's current generation. When reading a blob, the group number is fetched based on the `BlobId.Generation` of the blob being read.

YDB distributed storage

YDB distributed storage is a subsystem of YDB that ensures reliable data storage.

It allows you to store *blobs* (binary fragments ranging from 1 byte to 10 megabytes in size) with a unique identifier.

Description of the distributed storage Interface

Blob ID Format

Each blob has a 192-bit ID consisting of the following fields (in the order used for sorting):

1. **TabletId (64 bits):** ID of the blob owner tablet.
2. **Channel (8 bits):** Channel sequence number.
3. **Generation (32 bits):** Generation in which the tablet that captured this blob was run.
4. **Step (32 bits):** Blob group internal ID within the Generation.
5. **Cookie (24 bits):** ID used if the Step is insufficient.
6. **CrcMode (2 bits):** Selects a mode for redundant blob integrity verification at the distributed storage level.
7. **BlobSize (26 bits):** Blob data size.
8. **PartId (4 bits):** Fragment number when using blob erasure coding. At the "distributed storage <-> tablet" communication level, this parameter is always 0, referring to the entire blob.

Two blobs are considered different if at least one of the first five parameters (TabletId, Channel, Generation, Step, or Cookie) differs in their IDs. Therefore, it is impossible to write two blobs that differ only in BlobSize and/or CrcMode.

For debugging purposes, there is a string representation of the blob ID in the format `[TabletId:Generation:Step:Channel:Cookie:BlobSize:PartId]`, for example, `[12345:1:1:0:0:1000:0]`.

When writing a blob, the tablet selects the Channel, Step, and Cookie parameters. TabletId is fixed and must point to the tablet performing the write operation, while Generation must indicate the generation in which the tablet performing the operation is running.

When performing reads, the blob ID is specified, which can be arbitrary but is preferably preset.

Groups

Blobs are written to a logical entity called a *group*. A special actor called *DS proxy* is created on every node for each group that is written to. This actor is responsible for performing all operations related to the group. The actor is created automatically by the NodeWarden service, which will be described below.

Physically, a group is a set of multiple physical devices (OS block devices) located on different nodes, so that the failure of one device correlates as little as possible with the failure of another device. These devices are usually located in different racks or datacenters. On each of these devices, some space is allocated for the group, which is managed by a special service called *VDisk*. Each VDisk runs on top of a block storage device, from which it is separated by another service called *PDisk*. Blobs are broken into fragments based on erasure coding, with these fragments written to VDisks. Before splitting into fragments, optional encryption of the data in the group can be performed.

This scheme is shown in the figure below.

PDisk 1:1000

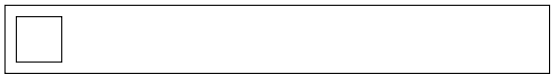
PDisk 1:1000

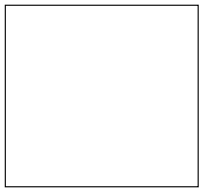
PDisk 1:1001

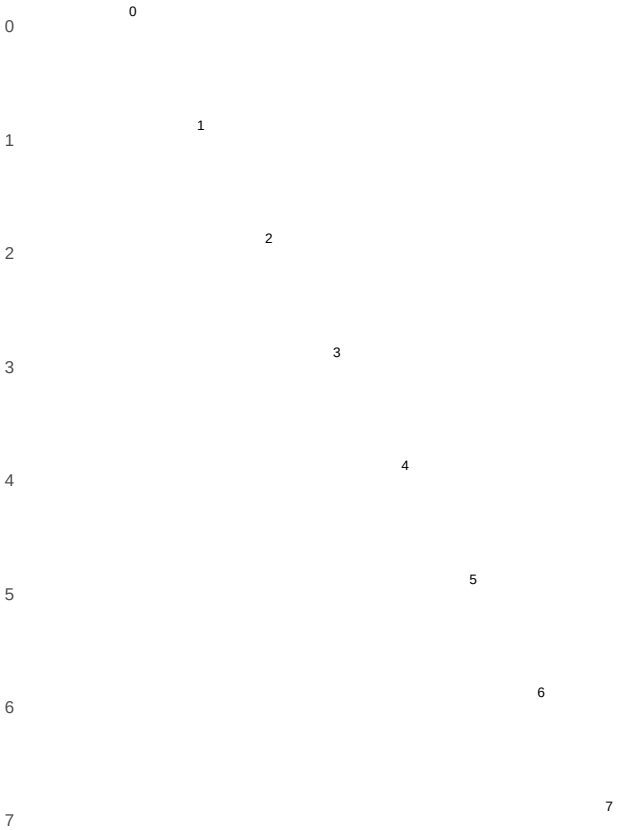
PDisk 1:1001
PDisk 2:1000
PDisk 2:1000
PDisk 2:1001
PDisk 2:1001
PDisk 3:1000
PDisk 3:1000
PDisk 3:1001
PDisk 3:1001
PDisk 4:1000

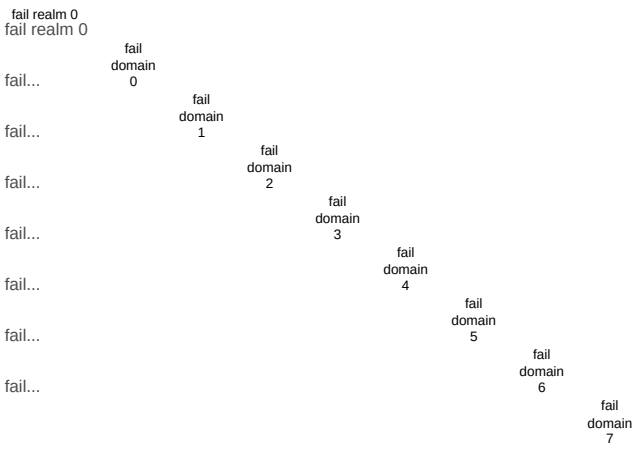
VDisks from different groups are shown as multicolored squares; one color stands for one group.

A group can be treated as a set of VDisks:



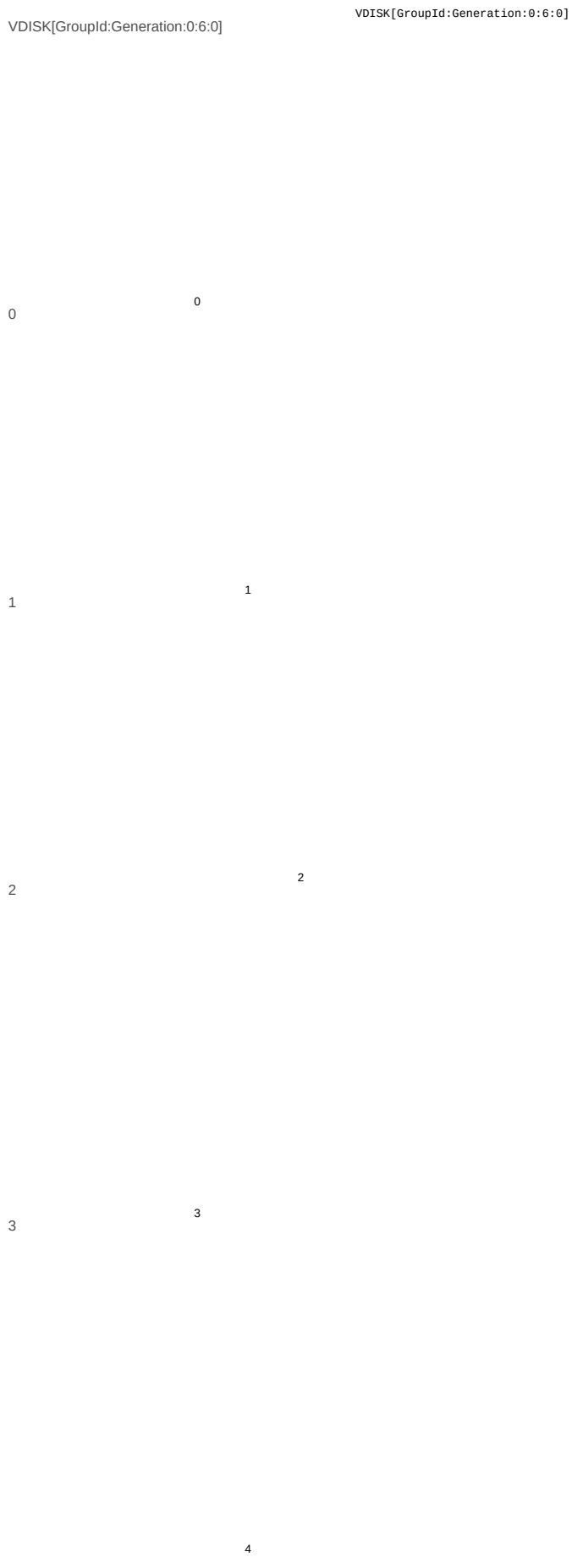



0
0
1
1
2
2
3
3
4
4
5
5
6
6
7
7


fail realm 0
fail realm 0
fail domain 0
fail...
fail domain 1
fail...
fail domain 2
fail...
fail domain 3
fail...
fail domain 4
fail...
fail domain 5
fail...
fail domain 6
fail...
fail domain 7

fail...

VDISK[GroupId:Generation:0:6:0]



4
5
5
6
6
7
7
8

fail realm 0
fail realm 0
fail realm 1
fail realm 1
fail realm 2
fail realm 2
fail...
fail
domain
0
fail...
fail
domain
1
fail
domain
2

fail...

VDISK[GroupId:Generation:1...Viewer does not support full syntax

VDISK[GroupId:Generation:1:2:0]

Each VDisk within a group has a sequence number, and disks are numbered 0 to N-1, where N is the number of disks in the group.

In addition, the group disks are grouped into fail domains, and fail domains are grouped into fail realms. Each fail domain usually has exactly one disk inside (although, in theory, it may have more, but this is not used in practice), while multiple fail realms are only used for groups whose data is stored in all three datacenters. Thus, in addition to a group sequence number, each VDisk is assigned an ID that consists of a fail realm index, the index that a fail domain has in a fail realm, and the index that a VDisk has in the fail domain. In string form, this ID is written as `VDISK[GroupId:GroupGeneration:FailRealm:FailDomain:VDisk]`.

All fail realms have the same number of fail domains, and all fail domains include the same number of disks. The number of fail realms, the number of fail domains inside each fail realm, and the number of disks inside each fail domain make up the geometry of the group. The geometry depends on the way the data is encoded in the group. For example, for block-4-2: `numFailRealms = 1`, `numFailDomainsInFailRealm >= 8` (only 8 fail domains are used in practice), `numVDisksInFailDomain >= 1` (strictly 1 fail domain is used in practice). For mirror-3-dc: `numFailRealms >= 3`, `numFailDomainsInFailRealm >= 3`, and `numVDisksInFailDomain >= 1` (3x3x1 are used).

Each PDisk has an ID that consists of the number of the node it is running on and the internal number of the PDisk inside this node. This ID is usually written as `NodeId:PDiskId`. For example, `1:1000`. If you know the PDisk ID, you can calculate the service ActorId of this disk and send it a message.

Each VDisk runs on top of a specific PDisk and has a *slot ID* comprising three fields (NodeID:PDiskId:VSlotId), as well as the above-mentioned VDisk ID. Strictly speaking, there are different concepts: a slot is a reserved location on a PDisk occupied by a VDisk, while a VDisk is an element of a group that occupies a certain slot and performs operations with the slot. Similar to PDisks, if you know the slot ID, you can calculate the service ActorId of the running VDisk and send it a message. To send messages from the DS proxy to the VDisk, an intermediate actor called *BS_QUEUE* is used.

The composition of each group is not constant and may change while the system is running. Hence the concept of group generation. Each "GroupId:GroupGeneration" pair corresponds to a fixed set of slots (a vector consisting of N slot IDs, where N is equal to group size) that stores the data of an entire group. *Group generation is not to be confused with tablet generation, as they are not related in any way*.

As a rule, groups of two adjacent generations differ by no more than one slot.

Subgroups

A special concept of a *subgroup* is introduced for each blob. It is an ordered subset of group disks with a strictly constant number of elements that will store the blob's data, depending on the encoding type (the number of elements in a group must be the same or greater). For single-datacenter groups with conventional encoding, this subset is selected as the first N elements of a cyclic disk permutation in the group, where the permutation depends on the BlobId hash.

Each disk in the subgroup corresponds to a disk in the group but is limited by the allowed number of stored blobs. For example, for block-4-2 encoding with four data parts and two parity parts, the functional purpose of the disks in a subgroup is as follows:

Number in the Subgroup	Possible PartIds
0	1
1	2
2	3
3	4
4	5
5	6
6	1,2,3,4,5,6
7	1,2,3,4,5,6

In this case, PartId=1..4 corresponds to data fragments (resulting from dividing the original blob into 4 equal parts), while PartId=5..6 represents parity fragments. Disks numbered 6 and 7 in the subgroup are called *handoff disks*. Any part, either one or more, can be written to them. The respective blob parts can only be written to disks 0..5.

In practice, when performing writes, the system tries to write 6 parts to the first 6 disks of the subgroup, and in the vast majority of cases, these attempts are successful. However, if any of the disks are unavailable, a write operation cannot succeed, which is when handoff disks come into play, receiving the parts belonging to the disks that did not respond in time. It may happen that several fragments of the same blob are sent to the same handoff disk as a result of complex failures and races. This is acceptable, although it makes no sense in terms of storage: each fragment should ideally be stored on a unique disk.

Redundancy recovery

If a disk fails, YDB can automatically reconfigure a storage group. Whether the disk failure is caused by the whole server failure or not is irrelevant in this context. Auto reconfiguration of storage groups reduces the risk of data loss in the event of a sequence of failures,

provided these failures occur with sufficient time intervals to recover redundancy. By default, reconfiguration begins one hour after YDB detects a failure.

Disk group reconfiguration replaces the VDisk located on the failed hardware with a new VDisk, and the system tries to place it on operational hardware. The same rules apply as when creating a storage group:

- The new VDisk is created in a fail domain that is different from any other VDisks in the group.
- In the `mirror-3-dc` mode, it is created within the same fail realm as the failed VDisk.

To ensure reconfiguration is possible, a cluster should have free slots available for creating VDisks in different fail domains. When determining the number of slots to keep free, consider the risk of hardware failure, the time required to replicate data, and the time needed to replace the failed hardware.

The disk group reconfiguration process increases the load on other VDisks in the group as well as on the network. The total data replication speed is limited on both the source and target VDisks to minimize the impact of redundancy recovery on system performance.

The time required to restore redundancy depends on the amount of data and hardware performance. For example, replication on fast NVMe SSDs may take an hour, while it could take more than 24 hours on large HDDs.

Disk group reconfiguration is limited or totally impossible when a cluster's hardware is distributed across the minimum required number of fail domains:

- If an entire fail domain is down, reconfiguration becomes impractical, as a new VDisk can only be placed in the fail domain that is down.
- Reconfiguration only happens when part of a fail domain is down. However, the load previously handled by the failed hardware will be redistributed across the surviving hardware, remaining in the same fail domain.

The load can be redistributed across all the hardware that is still running if the number of fail domains in a cluster exceeds the minimum amount required for creating storage groups by at least one. This means having 9 fail domains for `block-4-2` and 4 fail domains in each fail realm for `mirror-3-dc`, which is recommended.

Metadata distribution subsystems: StateStorage, Board, SchemeBoard

In a YDB cluster, three interconnected subsystems ensure metadata distribution between nodes: **StateStorage**, **Board**, and **SchemeBoard**. Each solves its own task, but all three are built on the same architectural principle — a distributed quorum service with deterministic replica addressing.

This article explains why these subsystems are needed, how they are structured, and how they work. An overview without core details for documentation users is in the Metadata distribution services section. Instructions for configuring and changing the configuration are described in the Configuring metadata distribution subsystems section.

Why metadata distribution subsystems are needed

A YDB cluster is a distributed system where millions of tablets can run simultaneously on thousands of nodes. For cluster components to interact with each other, they need to know:

- Where the leader of a specific tablet currently runs and how to reach it (**StateStorage**).
- Which nodes provide certain services, for example, connection points for clients (**Board**).
- what the current database schema is (**SchemeBoard**).

Distributing this data from a single cluster node is bad — it creates high load and problems when that node fails. Therefore, metadata is distributed across many cluster nodes using three specialized subsystems.

Subsystems

StateStorage — tablet state storage

StateStorage is a distributed service that stores the current state of tablets: where the leader runs, what its actor identifier, generation, and step are.

What StateStorage stores

For each tablet, StateStorage stores:

- **Current leader** — the actor identifier (ActorId) of the tablet leader.
- **Generation and step** (`generation:step`) — monotonically increasing counters used to resolve conflicts during leader election.
- **Follower list** — actor identifiers of the tablet replicas (if any).
- **Lock** — a flag indicating that the tablet is locked during the election of a new leader.

How StateStorage is used

StateStorage acts as a **tablet name resolution service**: any cluster node can use `TabletId` to find out `ActorId` of the current leader and directly contact it.

StateStorage also participates in the **tablet leader election process**. When a tablet starts or restarts, it registers itself in StateStorage and receives confirmation from the quorum of replicas.

> **Note**
>
> Data in StateStorage is **volatile**: it is stored only in the memory of replicas and is lost when processes restart. StateStorage is not a persistent store — it contains only information that can be easily restored both when tablets start and when the replicas themselves start.

Addressing in StateStorage

The set of replicas for a specific tablet is computed based on its `TabletId`. This means that replicas of different tablets can reside on different nodes, ensuring even load distribution.

Board — a service bulletin board

Board is a distributed service for publishing and searching metadata in the format “path → set of records”. It works on the publish-subscribe model: one or more actors publish information under a path, and other actors subscribe to that path and receive the current list of publications.

What Board stores

Board stores “path → payload” pairs. Multiple actors can publish data under the same path simultaneously — Board stores all publications and provides them to subscribers as a list.

How Board is used

The main use of Board is **storing information about endpoints** (connection points) of databases. When a database node starts, it publishes its address in Board under a path corresponding to the database name. Clients and other cluster components subscribe to that path and receive the current list of available endpoints.

How it works:

1. A publisher actor registers a record in Board under a given path.
2. A subscriber actor requests the list of records for a path and receives notifications about changes.

Unlike StateStorage, Board is not tied to specific tablets — it is intended for arbitrary services that need to make some data easily accessible in the cluster.

Addressing in Board

The selection of replicas for a specific path is computed from the hash of that path. This ensures deterministic routing: all publications and subscriptions for the same path land on the same replicas.

SchemeBoard — distribution of schema metadata

SchemeBoard is a distributed service for storing and distributing database schema metadata: tables, indexes, access rights, and other schema objects.

What SchemeBoard stores

SchemeBoard stores descriptions of schema objects: tables, directories, indexes, topics, and other schema objects. For each object, its full description is stored, including structure, settings, and access rights.

How SchemeBoard is used

SchemeBoard is a **cache of schema metadata** for all cluster components. When a database node receives a query, it needs to know the structure of the tables the query works with. Instead of accessing SchemeShard (a tablet that is the source of truth for the schema) every time, the node reads the schema from a local cache that is synchronized through SchemeBoard.

How it works:

1. When the schema changes, SchemeShard publishes an update to SchemeBoard.
2. Database nodes subscribe to changes in the schema objects they need and receive notifications about updates.

This allows database nodes to work with the current schema without constantly accessing SchemeShard, which significantly reduces its load and decreases query execution latency.

Addressing in SchemeBoard

The selection of replicas for a specific schema object is computed from the hash of the path to that object. This ensures deterministic routing of queries to schema metadata.

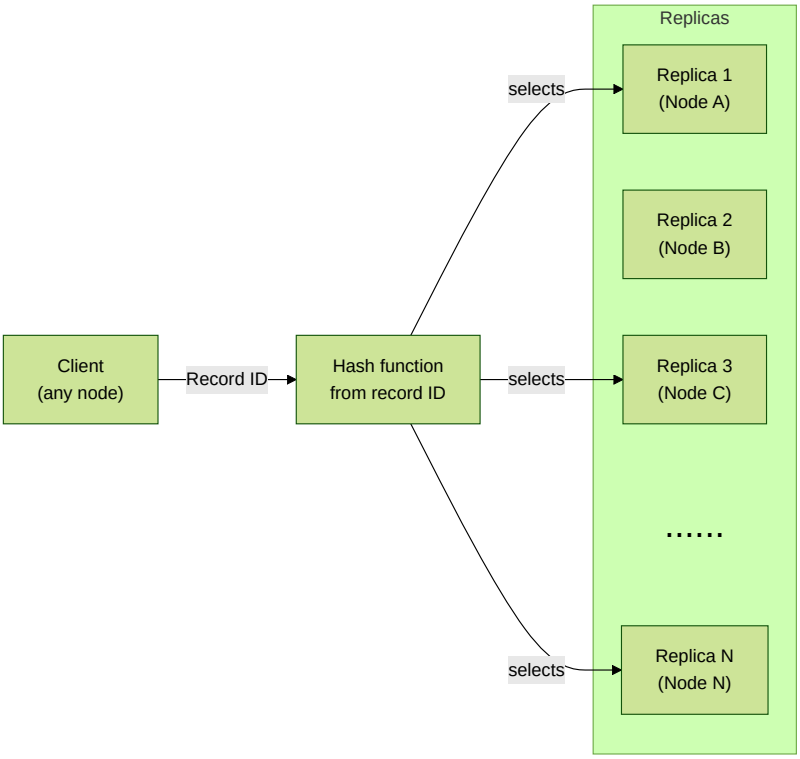
Comparison of subsystems

Characteristic	StateStorage	Board	SchemeBoard
Purpose	Storing tablet state	Publishing service metadata	Distributing schema metadata
Data type	Tablet leader state (ActorId, generation:step)	Arbitrary path → payload pairs	Schema object descriptions
Addressing key	TabletId	Path (string)	Path to the schema object
Main consumers	Tablet Pipe	gRPC proxy, clients (endpoint discovery)	Database nodes (schema cache)
Data source	The tablets themselves (on startup or leader election)	Publisher actors	SchemeShard

Deterministic replica selection

Each record in the subsystem is addressed by an identifier — for example, `TabletId` for StateStorage or a path to a schema object for SchemeBoard. Using this identifier, a hash function computes a fixed set of replicas on which the record is stored.

This is a key property: **any cluster node can independently compute on which replicas the record it needs is stored**. This makes the subsystems scalable and fault-tolerant.



Quorum

For each record, a certain number of replicas `nto_select` is selected. The system attempts to perform write operations with all `nto_select` replicas, but a failure of a minority of replicas is tolerated, and such a failure does not lead to temporary unavailability or suspension of work.
For a successful write operation, it is sufficient to get a response from the **majority** of the selected replicas — that is, from `nto_select / 2 + 1` replicas, where `nto_select` is the total number of replicas in the selection. This means that some replicas may be unavailable, and the subsystem will continue to work.

It is the quorum principle that allows safely taking nodes out for maintenance and performing rolling restarts of the cluster without losing availability.

Key concepts

Replica

Replica is an actor running on a cluster node and storing a part of the subsystem's metadata. A replica accepts read and write requests and participates in forming a quorum.

Each replica is independent: it continues to serve requests even if other replicas are temporarily unavailable. Data that could not be written to an unavailable replica will be delivered to it later when it recovers — either explicitly through a retry or automatically through a state synchronization mechanism.

Ring

Motivation

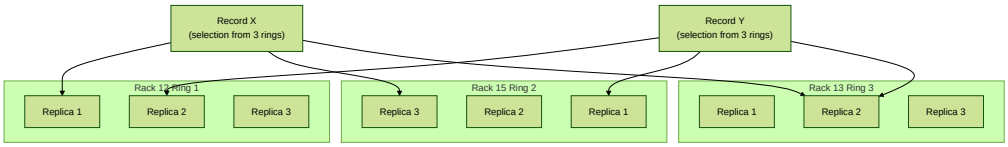
To distribute load evenly across cluster nodes, subsystems may need a large number of replicas: the more replicas there are, the fewer requests each replica handles. However, as the number of replicas grows, fault tolerance becomes a concern: if a failure domain (server rack) fails, several replicas may go down at once, affecting the quorum.

Definition and operating principle

A **ring** is a group of replicas from which **no more than one** replica is selected for each specific record. The replica selection within a ring is deterministic and based on a hash function of the record identifier. Thus, different records are served by different replicas within the same ring, ensuring even request distribution.

Rings and failure domains

Replicas of each ring are placed within a single **failure domain** (fail domain) — one or several server racks. Replicas from different rings are placed in different racks. Replicas from different rings must not be placed in the same rack. This guarantees that if any rack fails, **no more than one replica** is lost from each selection — the one belonging to the ring whose replicas were in that rack.

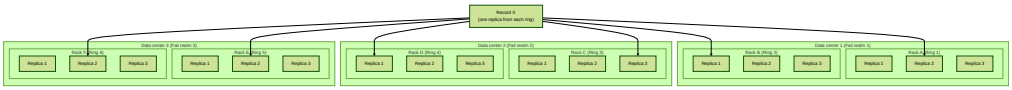


Rings and failure realms

In clusters with multiple **failure realms** (fail realm) — for example, data centers — additional restrictions apply to the placement of replicas across rings:

- Replicas from different failure realms are not included in the same ring.
- The number of rings in each failure realm is limited.

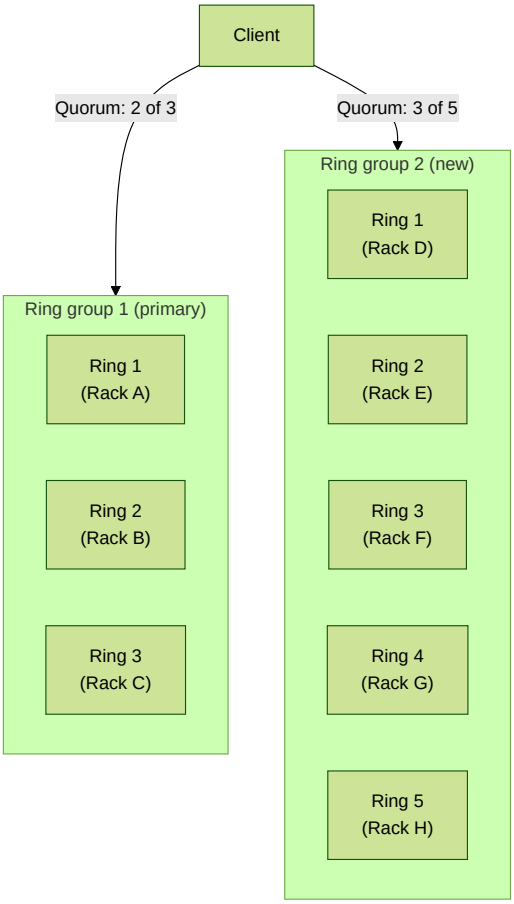
These restrictions make it possible to control how many rings a failure of an entire data center will affect: if each failure realm contains no more than `k` rings, then when it fails, no more than `k` replicas from each selection will go down.



Ring group

A **ring group** is a set of rings for which a **separate quorum** is assembled. A subsystem can work with several ring groups simultaneously, assembling a quorum in each of them independently.

Ring groups are a mechanism for **seamless configuration changes**. They allow introducing new replica sets and retiring old ones without stopping the cluster. For more details, see the section Ring groups and configuration changes.



Ring groups and configuration changes

Seamless configuration changes

Changing subsystem configuration — for example, moving replicas to other nodes or changing the number of replicas — is done by adding and removing ring groups. This allows changing the configuration without stopping the cluster or losing availability.

The configuration change process consists of several steps:

1. A new ring group with the required configuration is added. At this stage, it is marked with the `write_only` flag — this means that the new group receives all records (synchronizes with the data) but **does not participate in the read quorum**. Read requests are still served by the old group.
2. After the new group has synchronized and become fully functional, the `write_only` flag is removed. Now both groups participate in the quorum.
3. The old group is marked with the `write_only` flag, and the new one becomes primary.
4. The old group is removed.

A pause (at least one minute) must be maintained between steps so that the configuration has time to propagate to all cluster nodes.

Detailed instructions for manual configuration changes are provided in the section Configuring metadata distribution subsystems.

Automatic reconfiguration (SelfHeal)

In clusters with V2 configuration, **Metadata Distribution SelfHeal** automatically manages the State Storage, Board, and SchemeBoard configurations. It monitors the state of cluster nodes and, when necessary:

- Moves replicas from failed nodes to healthy ones.
- Can add new replicas as the cluster expands, subject to the settings and available nodes.

SelfHeal works through the same ring group mechanism as manual reconfiguration, but performs all steps automatically. For more details, see Metadata Distribution SelfHeal.

Ring groups in a two-data-center configuration

In a configuration with two data centers (the bridge mode), each data center (pile) has its **own ring group**. This allows:

- Each data center operates autonomously, gathering a quorum within its own group.
- Quickly and seamlessly switch which data center is primary, without changing the replica configuration.

For more details about bridge mode, see the Bridge mode section.

Fault tolerance and failure model

Metadata distribution subsystems are designed with the YDB failure model in mind, based on the concepts of failure domains (usually server racks) and failure regions (usually data centers).

The replica placement rule (different rings in different racks; replicas of the same ring may be in one or several racks) provides the following guarantees:

- **Single node failure**: no more than one replica is lost from each sample. The quorum is preserved.
- **Failure of an entire rack**: since all replicas of one ring are in the same rack, exactly one replica is lost from each sample. With `nto_select = 5`, the quorum (3 of 5) is preserved even when an entire rack is lost.

- **Rolling restart**: sequential restart of nodes does not break the quorum, since only part of the replicas is unavailable at any given moment.

Related materials

- Metadata distribution services — an overview for documentation users.
- Configuring metadata distribution subsystems — instructions for manually changing the configuration.
- Metadata Distribution SelfHeal — automatic management of the State Storage, Board, and SchemeBoard configurations.
- Bridge mode — a configuration with two data centers and the role of ring groups in it.
- Cluster topology — the failure model, failure domains, and failure regions.
- Glossary — definitions of terms: StateStorage, Board, SchemeBoard, tablet, ActorId.

LocalDB: persistent uncommitted changes

Tablets may need to save a large number of changes over a long period and then commit or abort them entirely. To support this scenario, LocalDB allows marking uncommitted changes in tables with a unique 64-bit transaction identifier (`TxId`). This data is stored in the table alongside committed data but is not visible to other queries until commit. Committing or aborting a transaction is atomic and cheap, and over time this data is integrated into the table in the normal committed form. Thus, uncommitted data is not stored in memory for a long time and is not limited by its capacity.

This mechanism is used as the basis for the following functionality:

- Saving uncommitted changes in long YQL transactions and the ability to see these changes in the same transaction before they are committed.
- Saving effects in volatile distributed transactions with the ability to roll back.
- Cross-cluster consistent replication, where changes are streamed in small batches and periodically committed, making the next consistent snapshot of the source database visible.

Limitations

This mechanism has the following limitations:

1. For a single key, changes can be made in parallel within different `TxId`, but these transactions must be committed in the same order in which the changes were written. For example, if there is a change for key K from transaction tx1 and then from transaction tx2, you can commit transaction tx1 and then tx2, and both changes will be visible. However, if you commit tx2 first, then tx1 must be aborted.
2. The number of uncommitted transactions and the amount of data in them for a specific key must be limited by higher-level layers.
3. `TxId` must be globally unique and must not be reused after a transaction commit or abort, including across different shards.

Writing uncommitted changes

In the recovery log (see flat_redo_writer.h), there are the following events:

- EvUpdateTx saves changes to the table with the specified uncommitted `TxId`. This event is created by the TDatabase::UpdateTx method of the local database (LocalDB).
- EvRemoveTx is used to remove the specified `TxId` when a transaction is aborted. This event is created by the TDatabase::RemoveTx method of the local database.
- EvCommitTx is used to commit the specified `TxId` with the specified MVCC commit version. This event is created by the TDatabase::CommitTx method of the local database.

Storing uncommitted changes in MemTable

MemTable in LocalDB is a small key-sorted tree of recently made changes that is stored in memory. The key in this tree is the table key, and the value is a pointer to a chain of changes for the corresponding key. For each change, the MVCC version of that change is specified (a pair `Step` / `TxId` with the global commit time). The rows in the change chain are merged within a single MemTable. For example, suppose we had the following operations for key K:

Version	Operation
`v1000/10`	`UPDATE ... SET A = 1`
`v2000/11`	`UPDATE ... SET B = 2`
`v3000/12`	`UPDATE ... SET C = 3`

Then the chain of rows with changes for this key in a single MemTable would look as follows:

Version	Row
`v3000/12`	`SET A = 1, B = 2, C = 3`
`v2000/11`	`SET A = 1, B = 2`
`v1000/10`	`SET A = 1`

In some cases, a MemTable may split between operations, which happens, for example, at the start of compaction. Then the MemTable from the example above would look like this:

MemTable	Version	Row
Epoch 2	`v3000/12`	`SET B = 2, C = 3`
Epoch 2	`v2000/11`	`SET B = 2`
Epoch 1	`v1000/10`	`SET A = 1`

New changes are applied to the current MemTable, even if they have not yet been committed. When writing, uncommitted changes are marked with a special version, specifying the maximum value for `Step` (as if implying a distant future) and the `TxId` of the transaction as `TxId`. Merging of lower-level changes is not performed. For example, suppose we then performed the following uncommitted operations for key K:

TxId	Operation
15	`UPDATE ... SET C = 10`
13	`UPDATE ... SET B = 20`

The chain of changes in the MemTable will now look like this:

Version	Row

`v{max}/13`	`SET B = 20`
`v{max}/15`	`SET C = 10`
`v3000/12`	`SET A = 1, B = 2, C = 3`
`v2000/11`	`SET A = 1, B = 2`
`v1000/10`	`SET A = 1`

During reading, for changes with version `Step == max`, the iterator checks their `TxId` against the transaction table. For committed transactions, the MVCC commit version is indicated there. Then it applies all committed changes until it finds and applies the first merged record with version `Step != max`.

Suppose we committed a transaction with `TxId = 13` at point `v4000/20`. At that moment, a record `[13] => v4000/20` appears in the transaction table, and this transaction becomes committed. All subsequent reads will see changes with `TxId = 13`, skipping the transaction with `TxId = 15` because it was not committed before `TxId = 13`. The chain of changes for key K does not change.

Now suppose we performed `SET A = 30` in version `v5000/21`. The resulting chain of changes will look as follows:

Version	Row
`v5000/21`	`SET A = 30, B = 20, C = 3`
`v{max}/13`	`SET B = 20`
`v{max}/15`	`SET C = 10`
`v3000/12`	`SET A = 1, B = 2, C = 3`
`v2000/11`	`SET A = 1, B = 2`
`v1000/10`	`SET A = 1`

A new row was added in the merged state, including changes for `TxId = 13`. Since transaction `TxId = 15` is not committed, its changes were skipped, which is reflected in the merged state for `v5000/21`. It is important that the upstream code does not commit `TxId = 15` after this, because such a commit would lead to an anomaly: some versions during reading will see the transaction as committed, and some will not.

Compaction of uncommitted changes

Compaction takes a part of the table data, merges them in sorted order, and writes them as a new SST that replaces the compacted data. If a MemTable participates in compaction, this also implies compaction of a part of the change log. In the compacted log, there may be `EvRemoveTx` / `EvCommitTx` messages that modify the committed transaction table — they must also end up in persistent storage. This is implemented by writing `TxStatus` blobs (see flat_page_txstatus.h), which LocalDB saves along with writing the SST with data. These blobs store the current list of committed and deleted transactions, which replaces the entire previously written log in terms of `EvRemoveTx` / `EvCommitTx` events. The compaction receives a reference to the entire current table, but during writing, the table is filtered, preserving only those transactions that were mentioned in the compacting MemTable or previous `TxStatus` pages.

At the level of data pages (see flat_page_data.h), uncommitted changes from MemTable (or other SSTs) are aggregated by `TxId` while preserving their relative order and are added before the main record as uncommitted delta records. A record has MVCC flags (`HasHistory`, `IsVersioned`, `IsErased`) that indicate the presence of MVCC history for the key and the presence of fields with the MVCC version of the row. For delta records, a special IsDelta flag is used, which is a `HasHistory` flag without specifying other MVCC flags. For rows with the delta flag, after the fixed part, the TDelta structure is stored in the extension fields, which specifies the `TxId` of the transaction that was uncommitted at the time of compaction.

There can be multiple delta records for a single key, and optionally the most recent committed record. Since only one record per key can exist on a data page, the offset table points to the first delta record, and the remaining records are accessible through the alternative offsets table for that record:

Offset	Description
-X*8	Main offset
...	...
-16	Delta 2 offset
-8	Delta 1 offset
0	Delta 0 header
...	...
Delta 1 offset	Delta 1 header
...	...
Main offset	Main header

Having a pointer to the Delta 0 record, the remaining records for the key can be obtained via the `GetAltRecord(size_t index)` method, where `index` is the record number in the chain (1 for Delta 1). The chain ends either with a pointer to the main record (without the delta flag) or 0 if the main record is absent.

Continuing the example started above, suppose that after writing the transaction with `TxId = 13`, the current MemTable was fully compacted. The record for the 32-bit key K may look as follows (offsets are relative to the pointer in the record list on the page):

Offset	Value	Description
-16	58	Main offset

-8	29	Delta 1 offset
0	0x21	Delta 0: IsDelta + ERowOp::Upsert
1	0x00	.. key column non-NULL
2	K	.. key column (32-bit)
6	0x00	.. column A empty
7	0	.. column A (32-bit)
11	0x01	.. column B = ECellOp::Set
12	20	.. column B (32-bit)
16	0x00	.. column C empty
17	0	.. column C (32-bit)
21	13	.. TDelta::TxId
29	0x21	Delta 1: IsDelta + ERowOp::Upsert
30	0x00	.. key column non-NULL
31	K	.. key column (32-bit)
35	0x00	.. column A empty
36	0	.. column A (32-bit)
40	0x00	.. column B empty
41	0	.. column B (32-bit)
45	0x01	.. column C = ECellOp::Set
46	10	.. column C (32-bit)
50	15	.. TDelta::TxId
58	0x61	Main: HasHistory + IsVersioned + ERowOp::Upsert
59	0x00	.. key column non-NULL
60	K	.. key column (32-bit)
64	0x01	.. column A = ECellOp::Set
65	1	.. column A (32-bit)
69	0x01	.. column B = ECellOp::Set
70	2	.. column B (32-bit)
74	0x01	.. column C = ECellOp::Set
75	3	.. column C (32-bit)
79	3000	.. RowVersion.Step
87	12	.. RowVersion.TxId
95	-	End of record

The remaining two records will be in the historical data at keys `(RowId, 2000, 11)` and `(RowId, 1000, 10)`, respectively. Their presence is indicated by the `HasHistory` flag in the main record.

During compaction, the iterator operates in a special mode, iterating over all deltas and all versions for each key. The compaction scan implementation (see flat_ops_compact.h) first aggregates the deltas that were uncommitted at the start of compaction by `TxId`, preserving their relative order. If changes from different `TxId` overlap, their order may change arbitrarily. Such reordering is valid because such transactions overlap in write order and the upper level is not allowed to commit them simultaneously. Once all uncommitted deltas are aggregated, they are written in the correct order to the resulting SST (see flat_part_writer.h and flat_page_writer.h). After that, iteration over row versions for the key begins, which are written to the SST in decreasing version order.

When the iterator encounters the first committed delta, the commit version of that delta is taken as the row version. All committed deltas below, including the first committed row version from each LSM level participating in the compaction, are merged into the row state. To move to the next version, the iterator skips committed deltas with a version greater than or equal to the last version, and the process repeats.

If a large number of deltas are written and compacted for a single key, and then quickly committed with different versions, the formation of all row versions will grow quadratically with the number of deltas. This is one reason why upper levels must control the number of deltas per key and prevent their uncontrolled growth. Another current limitation is that all uncommitted deltas must be kept in memory, because they can be committed at any moment and we must check this during search. Additionally, all deltas for a key must reside on the same page.

Statistics on uncommitted transactions

Optimistically, most write transactions eventually commit successfully, but sometimes transactions have to be aborted, even after compaction. Since after their abort a large amount of data may actually be deleted, for garbage collection purposes an `TxIdStats` page is saved in SST (see flat_page_txidstat.h) with statistics on the number of rows and bytes occupied by each `TxId`. As

transactions are aborted, the compaction strategy aggregates the amount of freed bytes and may decide to run additional compactions for garbage collection.

`TxIdStats` pages are also used to gradually reduce the commit table. After a transaction is committed, a record of it is stored in an in-memory hash table as long as there is at least one reference to the corresponding `TxId`. As SST is compacted, uncommitted deltas are overwritten as committed records and references on `TxIdStats` pages disappear. If the corresponding `TxId` was committed or aborted, and there are no references to it in either MemTable or SST, then this record is removed from the commit table and frees memory.

The size of the in-memory commit table is limited at the datashard level by the number of open transactions at any given time. Open transactions are those that have not yet been committed or aborted. During compaction, only deltas for transactions that were open at the start of compaction can be written to SST, because committed deltas are rewritten as regular committed rows, and aborted transactions leave no traces. Accordingly, the size of the commit table is limited by the number of open transactions multiplied by the number of SSTs.

Borrowing SSTs with uncommitted changes

When copying tables, as well as during DataShard splits and merges, the mechanism of "borrowing" SSTs in LocalDB is used. In this case, SSTs from the source shard are merged into the corresponding table on the destination shard. With the addition of uncommitted changes, there may be both changes from still open transactions (relevant in case of replication) and committed but not yet compacted changes. To ensure that data on the destination shards is read in the same form as on the source shard, `TxStatus` blobs are also borrowed and merged. Information from them is added to the commit table.

Transaction information may differ on different shards. Consider a new example:

1. A transaction writes a change for key K with `TxId` to shard S, while the data is compacted and the changes end up in a large SST.
2. Shard S starts exceeding the size threshold and splits into two shards L and R.
3. SSTs that contain changes for key K end up on both shards with a filter applied.
4. Key K ends up in shard L, but the transaction also exists on shard R because there is a reference to it from a borrowed SST.
5. The transaction commits the change on shard L, but no commit occurs on shard R, since logically and from the writer's perspective there were no changes on shard R.
6. Since the transaction completes, `TxId` on shard R is rolled back after some time, because in reality nothing changes in the shard's data.
7. Suppose later shards L and R are merged again.
8. In such a situation, `TxStatus` from shard L will have a commit, and `TxStatus` from shard R will have an abort for the same `TxId`. Since transactions successfully commit all changes, and an abort may be due to a filter applied, in case of merging conflicting information for `TxId`, the commit "wins" over the abort.

In practice, DataShard is fully compacted before each split or merge, so conflicting commit status information should not arise. For example, this means that transactions cannot decide to commit only part of the changes; all changes for a specific `TxId` must be committed together. It also means that transaction IDs cannot be reused, including across different shards. Only globally unique identifiers can also be used as `TxId` for uncommitted changes.

Want to join the YDB development team?

Learn about the YDB team and open positions, as well as opportunities for students.

Hive

Hive is a tablet responsible for managing other tablets, including selecting nodes for them to run on and deciding when to rebalance tablets.

The creation and deletion of tablets is initiated by the SchemeShard tablet. When a tablet is created, Hive assigns it a unique TabletId, fills in TabletStorageInfo, chooses the most suitable node, and sends a command to start the tablet on that node. In some abnormal situations, a tablet may interrupt its operation, in which case the node on which it was running sends a message to Hive. Hive also assumes that if the connection with a certain node is lost, the tablets running on it have stopped. In such cases, Hive restarts the tablets on other nodes, incrementing the generation.

A YDB cluster runs multiple Hives:

- A single *root Hive* responsible for the system tablets of all databases in the cluster. All nodes in the cluster are registered in the root Hive.
- A *database Hive* (one per database) is responsible for the tablets servicing the user load of a specific database. Only the compute nodes of the database are registered in that database's Hive.

When a node registers, it informs Hive of the types of tablets and the number of tablets that can be run on it.

Resource Usage Metrics

Hive evaluates resource usage to evenly distribute tablets across nodes. For each tablet, the usage of four types of resources is tracked:

1. *CPU* — processor consumption, calculated as the number of microseconds spent on tablet work in the last second. A value of one second corresponds to 100% load of a single core.
2. *Memory* — the amount of RAM consumed by the tablet.
3. *Network* — the amount of traffic generated by the tablet.
4. *Counter* — a fake resource for even distribution of tablets. If a tablet has a nonzero consumption of any other resource, its Counter value is 0; otherwise, it is 1. This way, Counter is used for any tablets for which there is no data on real consumption, as well as for tablets where real consumption tracking is disabled. By default, this applies only to column-oriented tables.

Additionally, to determine overloaded nodes, YDB uses memory consumption and processor resources in the actor system thread pools on each node. These values are converted into relative values (a number from 0 to 1). The maximum of these relative values is used as the node's overall resource consumption value — *Node usage*. Hive also applies aggregation over a window to all metrics to account for load spikes.

Resource usage information is used for choosing a node for a tablet. For example, if Hive has information only on the CPU consumption of a tablet, it tries to find a node with the lowest CPU load. If information on multiple resources is available, the highest of the resource usage values is used.

Autobalancing

At certain moments, Hive may start an auto-balancing process that moves tablets between nodes to improve load distribution. The situations when autobalancing occurs are listed below. The auto-balancer works iteratively, making decisions about moving tablets one at a time. It selects the most loaded node, chooses a tablet that runs on this node using weighted randomness, and chooses a more suitable node for it. This process is repeated until balance is restored. The way node load is determined depends on the type of balancing. For example, in case of CPU consumption imbalance, CPU usage is used. For uneven distribution of column-oriented tables, the number of tablets is used instead.

Resource Usage Imbalance

To quantify the balance of resource usage, Hive uses the *Scatter* metric, which is calculated separately for each resource using the following formula:

and are, respectively, the maximum and minimum usage of a resource across all nodes. To normalize consumption on each node, the number of available resources on the node is used, which may vary between nodes. Under low loads, this metric may fluctuate significantly. To avoid this, when calculating , it is assumed that resource usage cannot be lower than 30%. The balancer is triggered if exceeds a threshold.

The maximum value of across different resources is available as the sensor `Hive/MAX(BalanceScatter)` in the `tablets` subgroup.

Node Overload

Overloaded nodes can negatively affect YDB performance. CPU overload raises latencies, and consuming all memory can cause the node to crash with an out-of-memory error. The balancer is triggered when the load of one node exceeds 90% while the load on another node falls below 70%.

The maximum resource usage on a node is reported by the `Hive/MAX(BalanceUsageMax)` sensor in the `tablets` subgroup.

Even Distribution for a Specific Object

For tablets that use the Counter resource, the evenness of tablet distribution for each object (each table) is tracked in the form of the `ObjectImbalance` metric, similar to the described above. Restarting nodes may break the balance in tablet distribution and trigger balancing.

The maximum value of `ObjectImbalance` across different objects is reported by the `Hive/MAX(BalanceObjectImbalance)` sensor in the `tablets` subgroup.

Tablet Boot Process

This article describes the process of launching tablets from the perspective of Hive. Hive makes decisions about launching tablets in various situations:

- A tablet has just been created.
- Connection is lost with the node where the tablet was running.
- The node where the tablet was running sends a message that it has stopped working (for example, due to loss of connection with the cluster disk subsystem).
- When moving a tablet as part of auto-balancing.

In any of these situations, the following occurs:

1. The tablet is added to the boot queue.
2. When processing the queue, a suitable node is selected for it.
3. A command to start the tablet is sent to the node.

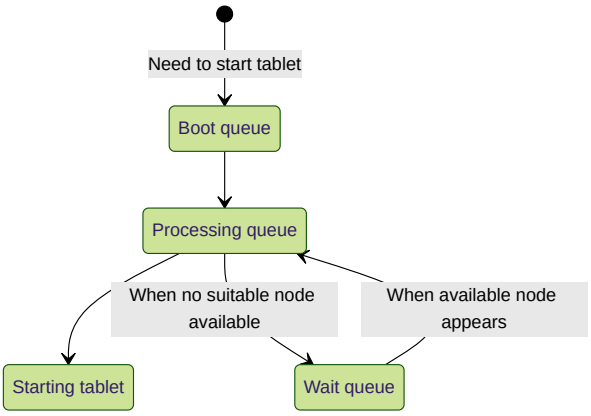
Boot Queue

The boot queue, or *Boot queue*, is stored in Hive's memory and is prioritized. Tablet priority is determined by the following factors:

1. Tablet type — system tablets have higher priority than user tablets.
2. Resource consumption metrics — tablets with higher consumption have higher priority.
3. Tablets that restart frequently have lower priority.

When processing the queue, a limited number of tablets are processed at once (`max_boot_batch_size` in configuration). This is necessary so that when starting a large number of tablets, Hive does not stop responding to other requests for a long time.

If when processing a tablet it turns out that it cannot be started on any of the nodes, then this tablet is postponed to a separate *Wait queue*. When node availability changes (a new node connects, or a restriction is removed from a node in Hive UI), Hive returns to these tablets and when processing the boot queue alternates tablets from the Boot Queue and tablets from the Wait Queue.



Warning

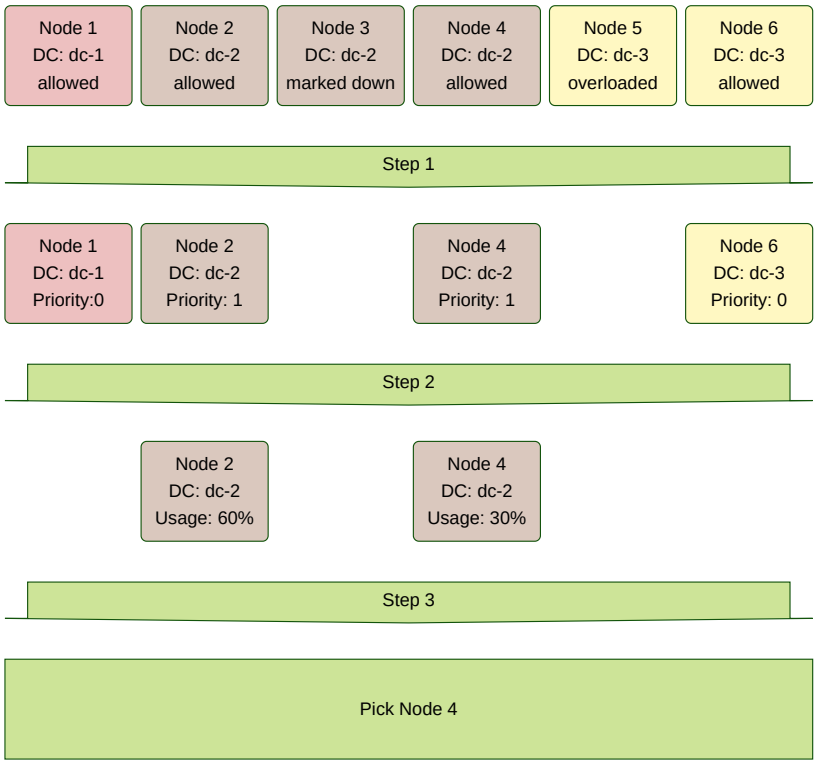
Simultaneous startup of many tablets can create increased load on a node. Therefore, the maximum number of simultaneously starting tablets on one node is limited by the `max_tablets_scheduled` value from configuration. At the same time, if one of the nodes hits this limit, Hive stops starting new tablets on other nodes too, so that this does not affect the uniformity of distribution. This behavior can be controlled using the `boot_strategy` parameter.

Node Selection

There are strict restrictions on which nodes are allowed to start a tablet: not every node can start every type of tablet; tablets of a certain database can only be started on nodes of that database. Additionally, when **moving** tablets, overloaded nodes are not considered.

1. From all suitable nodes, nodes with maximum priority are selected. Priority is determined based on the data center where the node is located. You can explicitly specify data center priorities in the `default_tablet_preference` subsection in the configuration. For coordinators and mediators, priorities are determined dynamically to maintain them in the same data center when possible. Additionally, if a tablet terminates with an error on a certain node, the priority of that node is lowered for the next start of this tablet.
2. For nodes with maximum priority, a target metric is calculated, which almost matches the Node usage metric. It differs in that only those resources consumed by this particular tablet are considered, as well as the presence of a penalty for the number of tablets of the same schema object.
3. Finally, based on this metric, a node is selected. The selection algorithm at this stage is determined by the `node_select_strategy` parameter in the configuration. By default, a random node is selected from 7% of nodes with the minimum metric value.

Below is an illustration of the process.



Boot Process

Each node runs a Local service responsible for interacting with Hive. When starting a tablet, Hive sends a tablet start command to the Local service of the required node, containing all information necessary for startup: TabletID, generation, channel history, and startup mode (leader or follower). After the tablet starts, Local reports this to Hive. From this moment, the tablet is considered started from Hive's perspective and remains so until:

- Local reports that it has stopped working
- communication with Local is disrupted

In these situations, the startup process will begin again for the next generation of the tablet.

DataShard: locks and transaction change visibility

When a long-running YQL transaction writes data to tables, it may try to read the same table later. To support observing data consistent with transaction changes DataShard tablets support writing uncommitted changes as part of a transaction, include these changes in subsequent queries by the same transaction, and allow atomic commits of these changes as long as serializable isolation is not violated.

The underlying LocalDB support also allows very large transaction commits, not limited by the size of a single message between distributed actors.

High level overview

Complex YQL transactions (either interactive, i.e. when client begins a transaction and uses it to perform queries without committing in the same query, or that involve multiple sub-queries) are split into multiple "phases" by QP, where output of one phase potentially acts as input to the next phase. For example, when a YQL query contains a `JOIN`, the first phase may be reading the first table, and the second phase may use the output of the first table to perform lookup queries in the second table.

For read-only queries QP uses global MVCC snapshots to ensure consistency between sub-queries. But when transaction also writes, it needs to ensure serializable isolation is not violated at commit time. Currently, this is achieved using Optimistic concurrency control, where reads add optimistic "locks" to observed ranges, and writes by other transactions "break" those locks at their commit time. Transaction may successfully commit as long as none of those locks are broken at commit time, otherwise it fails with a "transaction locks invalidated" error.

There's another way to look at optimistic locks. A single transaction may read from multiple shards using read timestamps (this may be a single global MVCC snapshot timestamp, or multiple timestamps, different for each read), while other transactions concurrently write to the same tables or shards. When transaction commits, it is assigned a single commit timestamp in the global serializable order of execution. As long as all of those reads could be repeated at the commit timestamp, without any change to observed results, the transaction might as well have executed (in its entirety) at the commit timestamp. The optimistic lock, as long as it's not broken, tells the transaction that it is possible to move all reads to the commit timestamp.

Uncommitted changes are not too different from reads in that regard. As long as DataShard can store those changes and then "move" them to the final commit timestamp without conflicts, the transaction may commit, otherwise it must abort. The main difference is that unlike read locks (which are stored in-memory), uncommitted changes are persistent, must be tracked across reboots, and must be cleaned up correctly when no longer needed.

How locks are used for reads

Operations proposed to DataShards are assigned a globally unique 64-bit `TxId`, which are allocated in large batches from global TxAllocator tablets. When QP performs the first read in a multi-phase transaction, it also uses this `TxId` as a lock identifier (historically named `LockTxId`), which is then used in all subsequent queries in the same YQL transaction. DataShard will add new locks when operation has `LockTxId` specified (it is not zero):

- See LockTxId field in `TEvRead` read requests
- See LockTxId field in `TEvWrite` write requests
- See LockTxId field in `TDataTransaction` messages (used for encoding data transaction body)

You may also see the `LockNodeId` field, which specifies the originating node id of the lock, which is used by DataShard for subscribing to lock status, and cleaning up locks when they are no longer needed.

Note that `LockTxId` is just a unique number, that is used across multiple operations in the same YQL transaction, while `TxId` is unique for every operation for a single DataShard. The use of the first `TxId` as `LockTxId` is not required, but since QP already has a globally unique number, and `LockTxId` is unrelated to `TxId`, it elides an extra allocation.

Locks table (see datashard_locks.h and datashard_locks.cpp) indexes locks by their primary key ranges in a range tree, allowing finding and "breaking" them by point keys. In the simplest case when read operation reads a range it is added using a SetLock method, and when write operation writes a key it breaks other locks using a BreakLocks method.

When the lock is added for the first time, it is assigned a monotonically increasing `Counter` in the current tablet's `Generation` (see TLock message), and a row with these numbers is added to the virtual `/sys/locks` table (which is no longer used). These locks are then returned in result messages (e.g. see TEvReadResult).

In the successful scenario, the lock exists and is not broken, `Generation` and `Counter` fields do not change.

In the unsuccessful scenarios, previously acquired locks are broken. For example, when changing the `Generation` of the lock (disabling the lock on restart) or the `Counter` (on an explicit error status, or when disabling and recreating the lock in the same generation).

The first `Generation` and `Counter` for each shard are remembered by QP, and changes during transaction lifetime are indicative of possible inconsistencies and serializable isolation violations. Read-Write transactions, or transactions that did not use global MVCC snapshots for efficiency reasons, perform the final commit that validates `Generation` / `Counter` values and commit only succeeds when all of them match.

Reads using global MVCC snapshots are already consistent. Nevertheless, they still acquire locks in case transaction might perform a write later in the lifecycle.

When acquiring locks DataShard performs additional checks, on whether or not conflicting changes have been committed "above" the snapshot. When the conflict is detected, the read succeeds, however the lock will have `Counter` equal to ErrorAlreadyBroken, to signal that even though the read is consistent, writes will never succeed. Such transactions may stop trying to add new locks, and succeed when it turns out the full YQL transaction is read-only. When such transactions try to write, however, they are aborted early as it would be impossible to commit them anyway.

How locks are used for writes

When QP needs to make uncommitted changes in a YQL transaction, it uses DataShard write transactions with a non-zero `LockTxId`. DataShard will then use this `LockTxId` as `TxId` for persisting uncommitted changes, available as long as the lock is valid. Internally locks that have uncommitted writes are called write locks. Such locks also become persistent (see the Locks table and related tables below), surviving DataShard restarts.

There are some limitations to such uncommitted write transactions:

- Transaction must run in an immediate transaction mode (i.e. uncommitted writes cannot be distributed, uncommitted writes to different shards are performed independently instead)

- Transaction must have a valid `LockNodeId` specified, DataShard subscribes to lock status using this node and automatically rolls back uncommitted changes when the lock expires (e.g. when transaction aborts unexpectedly, node is restarted and transaction state is lost, etc.)
- Transaction must have a valid MVCC snapshot specified, which is used as the conflict detection baseline (and reads when needed), and expected to be used across all reads and writes in the same YQL transaction.
- The specified lock must be valid and non-broken, otherwise the specified `LockTxId` must not have any uncompacted data in LocalDB. This protects against edge cases where transaction rolls back due to lock status failure, and QP tries writing to the shard again.

When the YQL transaction later reads using the same `LockTxId`, reads will use a per-query transaction map with LocalDB, where the `LockTxId` appears as if it's already committed, allowing transaction to observe its own changes, but not other uncommitted changes. Since reads are performed using an MVCC snapshot, the transaction map will have a special entry `[LockTxId] => v{min}`, so the uncommitted change is visible in all snapshots.

Uncommitted writes need additional conflict detection (see CheckWriteConflicts in the MiniKQL engine host implementation). When multiple uncommitted transactions write to the same key, DataShard needs to ensure correctness by breaking conflicting transactions. Transaction observer objects (e.g. TLockedWriteTxObserver) are used to detect these conflicts, where LocalDB calls back various interface methods whenever a change is skipped or applied during reads, and a special read is used before each write to detect other uncommitted writes to the same key. Whenever a conflict is detected, it is added to conflict sets between locks, so each lock remembers which other locks must be broken when it commits, and which locks will break this lock when they commit.

Reads also need additional checks for conflicts when uncommitted writes are involved. Because reads need to not only be internally consistent, but also match the eventual state at the commit timestamp. Transaction observers are used to gather uncommitted writes from other transactions (reads need to be able to eventually move to the commit timestamp), which introduce conflict graph edges from write locks to read locks.

Even more complicated is the case where change visibility writes happen over committed changes after the transaction MVCC read snapshot, and transaction reads rows including those uncommitted changes. For example, let's consider this case:

1. Key K initially has `A = 1` at version `v4000/100` (the two numbers in version are `Step=4000` and `TxId=100` of the commit timestamp)
2. Tx1 is started with an MVCC snapshot `v5000/max`
3. Tx2 commits a blind `UPSERT` with `B = 2` at version `v6000/102`
4. Tx1 performs an uncommitted blind `UPSERT` with `C = 3` and `TxId` 101
5. At this point Tx1 may still commit successfully, because it didn't read key K and change with `C = 3` may still move to some future commit timestamp
6. Tx1 performs a read of key K, which happens at snapshot `v5000/max`:
    - This read will have `[101] => v{min}` in its custom transaction map
    - The iteration will be positioned at the first delta with `C = 3` (since `v{min} <= v5000/max`), which will be applied to the row state
    - All other committed deltas and rows will also be applied, i.e. the row state would include `B = 2` which is currently committed
    - However, there is a conflict: `B = 2` is committed above the MVCC read snapshot (`v6000/102 > v5000/max`)
    - This will be detected in OnApplyCommitted callback, which calls CheckReadConflict
    - Since this introduces a read inconsistency, the lock will be immediately broken, and an inconsistent read flag will be raised
7. Not only the above read would fail, Tx1 would not be able to commit since serializable isolation can no longer be provided
8. The application will get a "transaction locks invalidated" error and retry the transaction from the beginning

Interaction with change collectors

DataShards may have change collectors, which log table changes and stream them to other subsystems. This is used to support Change Data Capture and Asynchronous secondary indexes. Depending on the mode, change collector may need to know the previous row state, and may perform a row read before each write.

When transaction has uncommitted changes, change collectors need to process those as well, but those must not be streamed until they are committed, and moreover streamed changes must match the eventual commit order of those changes. Changes are first accumulated in a separate LockChangeRecords table, and when transaction eventually commits they are added to the output stream in bulk using a single record in a ChangeRecordCommits table. This way all change processing is done gradually, it matches the size of each individual write, and there is no expensive post-processing at commit time.

Interaction with distributed transactions

When distributed transaction starts execution, it first validates locks and sends its validation result to other participants. When all successful validation results are received, the transaction body may execute and apply its intended side-effects. In other words, when all reads from all involved shards may successfully move to the commit timestamp, the transaction may execute all buffered UPSERTs, otherwise transaction body is not executed at all participants, and all accumulated changes are rolled back. Normally, validation result is persisted, and correctness is preserved by runtime key conflicts between transactions. However, uncommitted writes add a serious complication, since when the write lock is committed DataShard doesn't even know which keys are involved (it would need to keep all keys in memory, which is prohibitively expensive). Left unchecked, DataShard might validate the write lock, send a successful result to other participants, while another conflicting transaction breaks this lock and rolls back all changes. It is preferred to optimize for the case where transactions don't conflict, and stopping the pipeline when uncommitted writes are involved would be prohibitively expensive.

Instead, when the write lock is validated, it becomes "frozen", and cannot be broken anymore. When a conflicting transaction tries to break a frozen lock, it is temporarily paused, and waits until the transaction with that frozen lock is first resolved. DataShard tracks which locks break which other locks on commit, and to avoid potential deadlocks it also tracks when commit of a transaction A may break validation results for transaction B and vice versa. When A must go before B in the global order from coordinators, DataShard won't start transaction B validation until A completes, but as long as transactions don't conflict they may be executed out-of-order.

Committing changes

When QP needs to commit previously uncommitted changes, it proposes a transaction that commits previously acquired locks. Specifically this transaction must not have a `LockTxId` specified (the commit is not setting any new locks), and must include a previously set lock in Locks transaction field, and with Op set to Commit. The commit transaction may either be immediate (when the YQL transaction involves a single shard, whether or not multiple phases have been used), or prepared as a distributed transaction with multiple participants.

To support distributed transactions all shards that validate locks must be included in SendingShards, and all shards that have side-effects must be included in ReceivingShards. During validation sending shards will generate persistent ReadSets and send them to all

receiving shards, and receiving shards will wait for all expected ReadSets before executing the transaction. When transaction is executed with all successful validation results it will commit the lock by calling KqpCommitLocks. Otherwise, transaction body will not be executed, and lock is erased with all uncommitted changes rolled back by calling KqpEraseLocks, and it cannot be retried later on commit failure.

> **Note**
>
> Note that all uncommitted changes with the same `LockTxId` must be included in a commit transaction, and transaction must never try to partially commit. For example, when a transaction involves multiple writes, and one of those writes fails with an error, it would not be correct to skip the failed write and partially commit. Shards might merge later and non-matching transaction status would lead to consistency anomalies.

The commit transaction may also have additional side-effects, which are atomically executed after the lock is committed. QP will try to accumulate side-effects in memory until the same table is read in the same transaction, or until transaction commits, to reduce latency and fuse side-effects with commit as much as possible. When it is possible to accumulate all side-effects in memory, no uncommitted changes are persisted, and only read locks are optionally acquired.

When YQL transaction needs to rollback it runs an empty transactions with the Rollback lock op. Even if it didn't, when the TLockHandle is destroyed, subscribed DataShards will clean it up automatically in their TxRemoveLock internal transaction. The explicit removal is preferred, since uncommitted changes is a finite resource and asynchronous cleanup via `TLockHandle` would not ensure that resource is freed before new transactions try to write more uncommitted changes.

Limitations

Currently, uncommitted changes have a downside that all new writes have to search for conflicts. A single uncommitted write is enough to cause all new writes at a specific DataShard to switch to become non-blind, i.e. every write will have to perform a read first, which makes them more expensive, increases latency and decreases throughput. For maximum performance it is recommended to execute transactions where all reads are performed first, and a small amount of blind writes are performed last. This way uncommitted writes will not be used and DataShard performance will be optimal.

Due to LocalDB limitations DataShard also needs to ensure it doesn't accumulate too many open transactions, and the locks table is already limited to ~10k locks, which includes write locks. DataShard also has to count the number of uncommitted changes before each uncommitted write, which is implemented by counting skips in the transaction observer, and throwing TLockedWriteLimitException when the limit is exceeded.

Persistent locks survive DataShard reboots and restore the last state. Even though it is possible to persist specific ranges, this is not used in practice (DataShard would need to persist ranges during reads, which is expensive), and read locks are restored in the worst case "whole shard" as their range.

DataShard: distributed transactions

YDB implements distributed transactions, which are based on ideas from the Calvin: Fast Distributed Transactions for Partitioned Database Systems paper. These transactions consist of a set of operations performed by a group of participants, such as DataShards. Unlike Calvin, these operations are not required to be deterministic. To execute a distributed transaction, a proposer prepares the transaction at each participant, assigns a position (or a timestamp) to the transaction in the global transaction execution order using one of the coordinator tablets, and collects the transaction results. Each participant receives and processes a subset of transactions it is involved in, following a specific order. Participants may process their part of the larger transaction at different speeds rather than simultaneously. Distributed transactions share the same timestamp across all participating shards and must include all changes from transactions with preceding timestamps. When viewed as a logical sequence, timestamps act as a single logical timeline where any distributed transaction fully happens at a single point in time.

When the execution of a transaction depends on the state of other participants, the participants exchange data using so-called ReadSets. These are persistent messages exchanged between participants that are delivered at least once and contain read results with the state of the transaction. The use of ReadSets causes transactions to go through additional phases:

1. **Reading phase**: The participant reads, persists, and sends data that is needed by other participants. During this phase, QP transactions (type `TX_KIND_DATA`, which have a non-empty `TDataTransaction.KqpTransaction` field and subtype `KQP_TX_TYPE_DATA`) validate optimistic locks. Older MiniKQL transactions (type `TX_KIND_DATA`, which have a non-empty `TDataTransaction.MiniKQL` field) perform reads and send arbitrary table data during this phase. Another example of using the reading phase is the distributed TTL transaction for deleting expired rows. The primary shard generates a bitmask matching expired rows, ensuring that both the primary and index shards delete the same rows.
2. **Waiting phase**: The participant waits until it has received all the necessary data from the other participants.
3. **Execution phase**: The participant uses both local and remote data to determine whether to abort or complete the transaction. The participant generates and applies the effects specified in the transaction body if the transaction is completed successfully. The transaction body typically includes a program that uses the same input data and leads all participants to come to the same conclusion.

Participants are allowed to execute transactions in any order for efficiency. However, it's crucial that other transactions can't observe this order. Transaction ordering based on a coordinator's assigned timestamps ensures strict serializable isolation. In practice, single-shard transactions don't involve a coordinator, and shards use a locally consistent timestamp for such transactions. Variations in the arrival times of distributed transaction timestamps weaken the isolation level to serializable.

Additionally, YDB has support for "volatile" distributed transactions. These transactions allow participants, including coordinators, to store transaction data in volatile memory, which is lost when the shards are restarted until the transaction is completed and the effects are persisted. This also allows participants to abort the transaction until the very last moment, which will be guaranteed to abort for all other participants. Using volatile memory removes persistent storage from the critical path before the transaction execution, reducing latency.

When executing the user's YQL transactions, YDB currently uses distributed transactions only for the final commit of non-read-only transactions. Queries before the commit of a YQL transaction are executed as single-shard operations, using optimistic locks and global multi-version concurrency control (MVCC) snapshots to ensure data consistency.

Basic distributed transactions protocol

Operations that can be performed as distributed transactions in YDB include various types of participants. The basic protocol for distributed transactions is the same regardless of the type of transaction, with some notable differences in the schema changes, which have additional requirements to ensure these transactions are idempotent.

Distributed transactions are managed using proposer actors. Some examples of these are:

- TKqpDataExecutor executes DML queries, including distributed commits.
- SchemeShard executes distributed transactions for schema changes.
- TDistEraser executes a distributed transaction to consistently erase rows in tables with secondary indexes that match time-to-live (TTL) rules.

Distributed transactions in YDB are similar to two-phase commit protocols. The proposer actor goes through the following phases when executing a distributed transaction:

1. **Determining participants**: The proposer actor selects specific shards (`TabletId`) that are required for transaction execution. A table may consist of many shards (`DataShard` tablets with unique `TabletId` identifiers), but a particular transaction may only affect a smaller set of these shards based on the affected primary keys. This subset is fixed at the start of the transaction and cannot be changed later. Transactions that only affect a single shard are called "single-shard" transactions and are processed in what is known as the "immediate execution" mode.
2. **Prepare phase**: The proposer sends a special event, usually called `TEvProposeTransaction` (there is also the `TEvWrite` variant in DataShards), which specifies a `TxId`, a transaction identifier, unique within a particular cluster, and includes the transaction body (operations and parameters). Participants validate whether the specified transaction can be executed, select a range of allowed timestamps, `MinStep` and `MaxStep`, and reply with a `PREPARED` status on success.
   - For single-shard transactions, the proposer typically specifies an "immediate execution" mode (`Immediate`). The shard executes such transactions as soon as possible (at an unspecified timestamp consistent with other transactions) and replies with the result rather than `PREPARED`, which causes the planning phase to be skipped. Some special single-shard operations, such as `TEvUploadRowsRequest`, which implements `BulkUpsert`, don't even have a globally unique `TxId`.
   - The persistent transaction body is stored in the shard's local database, and the participant must ensure that it is executed when planned. In certain cases (for example, when performing a blind `UPSERT` into multiple shards), the participants must also ensure that the transaction is executed successfully, which may be in conflict with specific schema changes.
   - The volatile transaction body is stored in memory, and the participant responds with `PREPARED` as soon as possible. Future execution, whether successful or not, is not guaranteed in any way.
   - The proposer moves on to the next phase when they have received responses from all the participants.
   - It's not safe to re-send the propose event, except for schema operations, which, thanks to special idempotency fields, guarantee that a particular transaction will be executed exactly once.
3. **Planning phase**: When the proposer has received `PREPARED` replies from all participants, it calculates the aggregated `MinStep` and `MaxStep` values and selects a coordinator to assign the timestamp to the transaction. A `TEvTxProxy::TEvProposeTransaction` event is sent to the selected coordinator, which includes the `TxId` and a list of participants.
   - The transaction may only involve shards from the same database. Each shard attaches its ProcessingParams to the reply, which has the same list of coordinators when shards belong to the same database.
   - The coordinator is selected based on the received `ProcessingParams` because, historically, queries could be executed without specifying a database. The list of coordinators can only be determined from the participants.

    - When the `TEvTxProxy::TEvProposeTransaction` event is re-sent (currently only for schema transactions), it is possible that the transaction may have multiple timestamps associated with it. This is not typically a problem as the transaction will execute at the earliest possible timestamp, and any later timestamps will be ignored (the transaction will have been completed and removed by the time they occur).
4. **Execution phase**: The proposer waits for responses from the selected coordinator and participants, collecting the overall transaction outcome.
    - In some cases, such as a temporary network disconnection or a shard restart, the proposer may try to re-establish the connection and wait for the result. This process may continue until the transaction has been completed and the result is available.
    - When it is impossible to retrieve the result of a transaction from at least one of participant due to network issues, the transaction usually fails with an `UNDETERMINED` status, indicating that it is impossible to determine whether the transaction was successful.

Prepare phase in the DataShard tablet

Distributed transactions in the DataShard tablet begin with the `Prepare` phase, which can be proposed by one of the following events:

- TEvDataShard::TEvProposeTransaction provides an entry point for different types of transactions
- TDataEvents::TEvWrite provides a special entry point for transactions that write data and commit YQL transactions

Events that don't have an `Immediate` execution mode specified will begin the `Prepare` phase for the distributed transaction. The transaction body will be validated to determine whether it's even possible to execute it (for example, by using the CheckDataTxUnit unit for generic data transactions). A range of timestamps will then be selected:

- `MinStep` is selected based on the current mediator time or the wall clock.
- `MaxStep` is determined by a planning timeout, which at the moment is 30 seconds for data transactions.

Then, the transaction is written to disk (for persistent transactions) or kept in memory (for volatile transactions), the shard replies with a `PREPARED` status and starts waiting for a plan that specifies the `PlanStep` for the given `TxId`. The planning deadline is important because if the proposer fails unexpectedly, the shard cannot determine whether the proposer has successfully planned the transaction for a future timestamp, so the shard must ensure that the transaction will be executed when planned (unless it's volatile). Since transactions that are not yet planned block some concurrent operations (such as schema and partitioning changes), a deadline is used to make it impossible to plan a transaction after a certain time. When the mediator time exceeds `MaxStep` and there is no corresponding plan for the transaction, then the protocol guarantees that it will not be possible to plan the transaction anymore. The transaction that reaches the deadline can then be safely removed.

Transactions are stored on disk and in memory using the TTransQueue class. Basic information about persistent transactions is stored in the TxMain table, which is loaded into memory when DataShard starts. The potentially large transaction body is stored in the TxDetails table and is not kept in memory while waiting. The transaction body is loaded into memory just before conflict analysis with other transactions in the pipeline.

Volatile transactions are stored in memory and are currently lost when DataShard restarts (they may be migrated during graceful restarts in the future). The restart aborts the transaction for all participants, and any participant can initiate the abort before the transaction body is executed and its effects are persisted. Shards use this feature to make schema and partitioning changes faster by aborting all pending transactions without waiting for a planning deadline.

The distributed transaction body needs to have enough information about the other participants so that each shard can know when it needs to generate and send outgoing ReadSets and which shards should expect and wait for incoming ReadSets. QP transactions currently use ReadSets for validating and committing optimistic locks, which are described using TKqpLocks generated by the `TKqpDataExecutor` actor. This message describes the following shard sets:

- `SendingShards` are shards that send ReadSets to all shards in the `ReceivingShards` set.
- `ReceivingShards` are shards that expect ReadSets from all shards in the `SendingShards` set.

Volatile transactions expect all shards to be in the `SendingShards` set because any shard may abort the transaction and need to send its commit decision to other shards. They also expect all shards that apply changes to be in the `ReceivingShards` set. Whether or not changes are committed depends on the decisions of other shards. Exchanged ReadSet data is serialized into TReadSetData message with a single `Decision` field specified.

An example of a distributed transaction that doesn't use ReadSets is a persistent distributed transaction with blind writes. In this case, after successful planning, the transaction cannot be aborted, and the shards must ensure the future success of the transaction during the `Prepare` phase.

Planning phase

When all participants have provided their `PREPARED` responses, the proposer calculates the maximum `MinStep` and the minimum `MaxStep`, then selects a coordinator (which is currently implemented using a `TxId` hash) and sends a `TEvTxProxy::TEvProposeTransaction` event, which includes the `TxId` and a list of participants with the operation type (read or write) for each participant (even though this information is not currently used). The selected coordinator then selects the closest matching `PlanStep` and associates it with the specified `TxId` and the list of participants. Plan steps for persistent transactions are allocated every 10 milliseconds (this setting is called plan resolution), and the association is also stored on disk. Plan steps for volatile transactions are selected from those reserved for volatile planning and can be as frequent as every millisecond, and the association is only stored in memory.

Planning each plan step (which can contain zero or more transactions) involves distributing participants (and their transactions) to mediators. This is currently done by hashing the `TabletId` of each participant modulo the number of mediators. Each mediator receives a subset of the plan step in increasing the `PlanStep` order. The subset only includes matching participants and may be empty, even if the full plan step is not. If a mediator restarts or the network becomes temporarily disconnected, the coordinator reconnects and sends all unacknowledged plan steps again in order.

Plan steps with persistent transactions are only sent to mediators after being fully persisted to disk. They are only removed from the coordinator's local database when acknowledged by participants and are guaranteed to be delivered at least once. Plan steps with volatile transactions, on the other hand, are only stored in memory and may be lost if the coordinator restarts. When a plan step is resent, it may or may not include acknowledged transactions or previously sent volatile transactions that still need to be acknowledged. This includes empty plan steps. Only the latest empty plan step will be kept in memory for re-sending.

To reduce the number of errors during graceful restarts, the coordinator leaves its state actor in memory even after the tablet stops working. The address of this state actor is persisted after the instance has been fully initialized and before it is ready to accept new requests. New instances contact this state actor and transfer the last known in-memory state, including the list of planned volatile transactions. This state actor is also used to transfer any unused volatile planning reserves, allowing new instances to start faster without having to wait until those reserves expire.

Mediators receive a stream of `TEvTxCoordinator::TEvCoordinatorStep` events from each coordinator and merge them using the matching `PlanStep` field. Merged plan steps with steps less or equal to the minimum of the last step received from each coordinator are considered complete and are sent to participants using `TEvTxProcessing::TEvPlanStep` events. Each participant receives an event with the `PlanStep`, specifying the timestamp, and a list of `TxId` that must be executed at that timestamp. Transactions within each plan step are ordered based on their `TxId`. The `(Step, TxId)` pairs are then used as the global MVCC version in the database.

Participants acknowledge that they have received (and persisted in the case of a persistent transaction) each plan step by sending a `TEvTxProcessing::TEvPlanStepAccepted` event to the sender (which is a mediator tablet) and a `TEvTxProcessing::TEvPlanStepAck` event to the specified coordinator tablet actor (as specified in the `AckTo` field of each transaction). Plan steps and/or transactions will be considered delivered when these events have been processed and will not be resent.

Based on `TEvTxProcessing::TEvPlanStepAccepted` events, mediators also track which `PlanStep` has been delivered to all participants, inclusive. This maximum `PlanStep` is known as the current mediator time and is distributed to nodes with running DataShards through subscriptions to the `TimeCast` service. The current mediator time indicates that all possible `TEvTxProcessing::TEvPlanStep` events have been received and acknowledged by the shards up to and including the specified `PlanStep`. Therefore, shards should be aware of all transactions up to this timestamp. The current mediator time is helpful as it allows the shards to keep track of the time progressing even when they are not participating in distributed transactions. For efficiency, all shards are partitioned into several timecast buckets at each mediator. The current time in each bucket advances when all participants in transactions in that bucket acknowledge their `TEvTxProcessing::TEvPlanStep` events. The current mediator time is available to shards when they subscribe by sending TEvRegisterTablet event to the time cast service during shard startup and get the address of an atomic variable from the TEvRegisterTabletResult event. This atomic variable allows the system to avoid broadcasting many frequent events to idle shards.

DataShards handle `TEvTxProcessing::TEvPlanStep` events in the TTxPlanStep transaction. Transactions are found by their corresponding `TxId`, get their `Step` assigned, and then are added to the `Pipeline`. The `Pipeline` uses the `PlanQueue` to limit the number of concurrently running transactions and executes them in the `(Step, TxId)` order.

Execution phase in the DataShard tablet

The PlanQueue unit allows distributed transactions to start, subject to concurrency limits, in the increasing `(Step, TxId)` order. The transaction body is loaded from disk when needed (for evicted persistent transactions), QP transactions finalize execution plans and arrive at the BuildAndWaitDependencies unit. This unit analyzes transaction keys and ranges declared for reading and writing and may add dependencies on earlier conflicting transactions. For example, when transaction `A` writes to key `K` and a later transaction `B` reads from key `K`, then transaction `B` depends on transaction `A`, and transaction `B` cannot start until transaction `A` completes. Transactions leave `BuildAndWaitDependencies` when they no longer have direct dependencies on other transactions.

Next, persistent QP transactions execute the read phase (which includes validating optimistic locks) and generate outgoing OutReadSets in the BuildKqpDataTxOutRS unit. Then, the StoreAndSendOutRS persists outgoing ReadSets and access logs for optimistic locks. Optimistic locks that have attached uncommitted changes are marked with the `Frozen` flag, which prevents them from being aborted until the transaction completes. Otherwise, lock validity is ensured by assigning writes with a higher MVCC version and ensuring the correct execution order of conflicting transactions. Operations with access logs or outgoing ReadSets are added to the `Incomplete` set, which ensures that new writes can't change the validity of previous reads and generally need to use a higher MVCC version. However, new reads don't necessarily need to block on the outcome of the incomplete transaction and can use a lower MVCC version as long as it's consistent with other transactions.

Persistent QP transactions prepare data structures for incoming InReadSets in the PrepareKqpDataTxInRS unit and begin waiting for all necessary ReadSets from other participants in the LoadAndWaitInRS unit. In some cases, such as blind writes to multiple shards without lock validation, distributed transactions may not require the exchange of ReadSets, and the ReadSet-related units don't perform any actions in these scenarios.

Finally, the QP transaction operation reaches the ExecuteKqpDataTx unit. This unit validates local optimistic locks using previously persisted `AccessLog` data when available, validates ReadSets received from other participants, and, if everything checks out, executes the transaction body and returns a result. If lock validation fails locally or remotely, the transaction body is not executed, and the operation fails with an `ABORTED` status.

Volatile transactions

Volatile distributed transactions are stored only in memory and are lost when the shard is restarted. Nevertheless, they must guarantee that the distributed transaction either commits at all participants or is aborted promptly. The advantage of volatile transactions is that they don't require separate read, wait, and execution phases. Instead, they are executed atomically in a single phase with 1RTT storage latency from the start of the distributed commit to the successful reply. This means they usually don't slow down the pipeline and can increase transaction throughput. Since any shard can abort a volatile transaction without waiting for a planning deadline, it also limits the unavailability of the shard during partitioning and schema changes.

Volatile transactions are based on persistent uncommitted changes in the local database. During the execution phase, DataShard optimistically assumes that all remote locks will be validated, applies effects in the form of uncommitted changes (using the globally unique `TxId` of the distributed transaction), and adds the transaction record to the VolatileTxManager in a `Waiting` state for other participants to make decisions. A successful reply is only sent when all transaction effects are persistent, and a committed decision has been received from all other participants. Reads use the `TxMap` to observe all pending changes and check their status using the `TxObserver`. When a read result depends on the outcome of a volatile transaction, operations subscribe to its status and restart after it has been decided (either by reading committed changes or skipping them if the transaction aborts).

Having uncommitted and not yet aborted changes limits the shard's ability to perform blind writes. Because uncommitted changes must be committed in the same order in which they were applied to any given key, and shards don't keep these keys in memory after transactions have been executed, DataShard needs to read each key and detect conflicts before it can perform any write. These conflicting changes may be uncommitted changes associated with an optimistic lock (and these locks must be broken along with rolling back the changes) or uncommitted changes from waiting for volatile transactions. We don't want to block writes unnecessarily, so even non-volatile operations can switch to a volatile commit. Such operations allocate a `GlobalTxId` when necessary (this is a per-cluster unique `TxId` that is needed when the request doesn't provide one, such as in `BulkUpsert`) and write changes to conflicting keys as uncommitted. The transaction record is then added to the `VolatileTxManager` without specifying other participants and becomes initially committed. It also specifies a list of dependencies, which are transactions that must be completed before the transaction can be committed. These transactions don't block reads and are eventually committed in the local database after all their dependencies have been completed or aborted.

To reduce stalls in the pipeline, transactions use the conflict cache for keys that are declared for writes in distributed transactions. These keys are read while transactions are waiting in the queue, and conflicting transaction sets are cached by the RegisterDistributedWrites function call. All writes to cached keys update these conflict sets and keep them up-to-date. This allows distributed transactions with writes to execute faster by processing lists of conflicting transactions using a hash table lookup even when the table data is evicted from memory.

DataShard may have change collectors, such as async indexes and/or Change Data Capture (CDC). Collecting these changes for volatile transactions is similar to uncommitted changes in transactions, generating a stream of uncommitted change records using the `TxId` as its `LockId`. These records are then either added atomically to shard change records upon commit or deleted upon abort. Depending on the settings of the change collector, it may also need to read the current row state. Suppose this row state depends on other volatile transactions that are waiting. In that case, it is handled similarly to any other read by adding a dependency and restarting when these dependencies are resolved. This can cause the transaction pipeline to stall, but it's conceptually similar to persistent distributed transactions with ReadSets that read-write conflicts can also stall.

Volatile transactions also generate and write to disk OutReadSets for all other participants specified in the `ReceivingShards` sets in the same local database transaction, which writes uncommitted effects. When these uncommitted changes become persistent, these ReadSets are sent to other participants, notifying them that this shard is ready to commit the distributed transaction and will not change its decision until the transaction is aborted by another participant.

Shards forget about volatile transactions that do not persist in their transaction records when they are restarted and will not send outgoing ReadSets for transactions they no longer know anything about. Shards that successfully execute the transaction body send a special event called ReadSet Expectation to inform about them waiting for an incoming ReadSet. This is done even before the effects are persisted, so the shards have a chance to find out about aborted transactions as early as possible. Restarted shards will receive a request for a transaction they don't know about and respond with a special ReadSet without data. In this way, all participants will find out about the aborted transaction and abort it as well.

Volatile transaction guarantees

To reiterate, volatile transactions are stored in memory by coordinators and participants and may fail for various reasons. It's important to understand why a volatile transaction either commits or aborts for all participants and why it's not possible for a transaction to commit only for a subset of participants.

Some examples of potential sources of error for the transaction:

1. Any shard can unexpectedly restart, including when a new instance starts while the old one is still running and is unaware of the new one. Since transactions are only stored in memory, the new instance may not be aware of transactions that failed to save their effects to disk. A transaction may also have been successfully executed and committed without leaving any traces except for its committed effects.
2. The coordinator may unexpectedly restart without transferring its in-memory state, and the transaction may have been sent to some participants but not all of them.
3. Any shard might have decided to abort the transaction due to an error.

A vital guarantee is that if any shard successfully persists, uncommitted changes, and a transaction record, and if it receives a `DECISION_COMMIT` from all other participants, the transaction will be eventually committed at all participants and can't be rolled back. It's also important to note that if any shard aborts a transaction due to an error (such as forgetting about the transaction after restarting), the transaction will never be able to complete and will eventually roll back.

These guarantees are based on the following:

1. ReadSets with `DECISION_COMMIT` are sent to other participants only after all uncommitted changes, outgoing ReadSets, and a transaction record have been persisted to disk. Therefore, `DECISION_COMMIT` messages are persistent and will be delivered to other participants until acknowledged. Even if another shard aborts the transaction, these messages will continue to be delivered.
2. ReadSets are only acknowledged either when a shard successfully writes to disk or when a read-only lease is active and an unexpected ReadSet is received. Specifically, an older tablet instance will not acknowledge a ReadSet for a transaction already prepared and executed by a newer tablet. The local database ensures that a new tablet will not be activated until the read-only lease on older tablets has expired (provided that the monotonic clock frequency is not more than twice as fast). A successful write confirms that the tablet held the storage lock at the beginning of the commit, and no newer tablets were running simultaneously.
    - Volatile transactions use an optimization where received ReadSets are not written to disk. Acknowledgment is sent only after a commit or an abort (removal) of the transaction record has been fully persisted on disk.
    - When all participants have generated and persisted in their outgoing `DECISION_COMMIT` ReadSets, they will continue to receive the complete set of commit decisions until they have fully committed the transaction effects and sent their acknowledgments.
3. Any message related to an aborted transaction is only sent after the abort has been written to disk. This is necessary to handle situations where multiple generations of a particular shard are running at the same time. After a newer generation has successfully committed a transaction and acknowledged ReadSets, an older generation (which would fail when trying to access the disk or validate a read-only lease) may receive a `NO_DATA` reply to its ReadSet Expectation (which was acknowledged by the newer generation). However, the older generation will not be able to commit the removal of the transaction record and cannot reply with an error incorrectly. When the removal of the transaction record (and the transaction effects) have been fully committed, the current generation can be assured that newer generations won't start with the transaction again, and it won't commit.
4. After collecting all `DECISION_COMMIT` ReadSets, a successful reply does not need to wait for the final commit in the local database. Instead, it has to wait until the uncommitted effects and the transaction record have been persisted. This allows the successful reply to have a 1RTT storage latency on the critical path caused by writing uncommitted changes and the transaction record. This successful response is stable:
    - If a shard is restarted, its effects and transaction record will be persistent. In the worst case, it will be recovered in the `Waiting` state.
    - From the MVCC perspective, the transaction has already been completed, so any new reads will have an MVCC version that includes these changes. Any new read will begin waiting for the transaction to resolve, which is now guaranteed to succeed.
    - The restored `Waiting` transaction could not have its incoming ReadSets acknowledged, so they will be resent. Since we received `DECISION_COMMIT` earlier, they will be received again, and the transaction will quickly move to the `Committed` state.
5. As long as the transaction is in the queue, and especially while its `PlanStep` is not yet known, incoming ReadSets are stored in memory and will not be acknowledged. Acknowledgments are delayed until the transaction has been fully committed or is aborted prematurely (in which case, no changes have been made, and no transaction record has been committed, so newer generations cannot recover or commit the transaction).

Volatile distributed transactions also ensure that changes are visible and stable across multiple shards participating in the same distributed transaction. For instance, persistent distributed transactions allow the following anomaly:

1. Distributed transaction `Tx1` performs a blind write to keys `x` and `y` in two different shards.
2. Single-shard transaction `Tx2` reads key `x` and observes a value written by `Tx1`.
3. Single-shard transaction `Tx3` (which starts after `Tx2` has completed) reads key `y` and does not observe a value written by `Tx1`.

This anomaly can occur when a shard with key `x` quickly receives a transaction plan step with `Tx1` and performs a write. A subsequent read in `Tx2`, which arrives a bit later, will have its local MVCC version include changes to key `x` and observe the change. However, the shard with key `y` may be running on a slower node and may not be aware of the `Tx1` plan step when a subsequent read in `Tx3` arrives. Since the time cast bucket doesn't have `Tx1` acknowledged, its mediator time will remain in the past, and `Tx3` will have a local MVCC version that doesn't include changes to key `y`. From the user's perspective, `Tx2` must happen before `Tx3`, but the global serializable order turns out to be `Tx3`, `Tx1`, and `Tx2`.

Volatile transactions cannot observe this particular anomaly because a read that observes changes to key `x` must have received `DECISION_COMMIT` from the other shard with key `y`. This means that the transaction record has been persisted in both shards and, crucially, the shard with key `y` has marked it as completed. A read from key `y` will choose a local MVCC version that includes changes made by `Tx1` and will need to wait until `Tx1` is resolved.

In other words, if a client observes the result of a volatile transaction at a certain point in time, all subsequent reads will also observe the result of that volatile transaction, and changes stability is not violated.

Indirect planning of volatile transactions

The coordinator restart may cause the plan step to only reach a subset of the participants in a volatile distributed transaction. As a result, some shards may execute the transaction and begin waiting for ReadSets, while other shards continue waiting for the plan step to arrive. Due to the planning timeout of 30 seconds, some transactions could experience excessive delays before eventually aborting.

When any participant receives the first plan step of a volatile transaction, they will include the `PlanStep` in the ReadSets they send to other participants. The other participants then indirectly learn about the `PlanStep` assigned to the transaction and remember it as the `PredictedPlanStep`. Even if their own plan step is lost, DataShard will add the transaction to the `PlanQueue` when its mediator time (directly or indirectly) reaches the `PredictedPlanStep`, as if they had received a plan step with that predicted `PlanStep` and `TxId`. If the `PredictedPlanStep` is in the past, the transaction will be quickly aborted, as if it had reached the planning deadline.

Since the same transaction can be assigned to multiple plan steps, with some getting lost, different participants may have a distinct `PlanStep` assigned to the same transaction. The same transaction will then try to execute at different timestamps. DataShards verify that `DECISION_COMMIT` messages have their `PlanStep` matching their assigned `PlanStep` and only process them if the steps match. The transaction will be aborted if there is a mismatch in the plan steps.

Internals of the V2 configuration mechanism

The V2 configuration in YDB provides a unified approach to managing cluster settings. While the DevOps section describes how to use this mechanism, this article focuses on its technical design. It will be useful for YDB developers and contributors who want to make changes to this mechanism, as well as anyone who wants to gain a deeper understanding of what happens in a YDB cluster when the configuration changes.

The YDB cluster configuration is stored in several locations, and various cluster components coordinate synchronization between them:

- as a set of node configuration files in the file system of each YDB node (required to connect to the cluster when a node starts up).
- in a special metadata storage area on each PDisk (a quorum of PDisks is considered the single source of truth for the configuration).
- in the local database of the DS Controller tablet (required for the distributed storage).
- in the local database of the Console tablet.

Some parameters take effect on a node immediately after the modified configuration is delivered to its location, while others only take effect after the node is restarted.

Distributed configuration system (Distconf)

Distconf is a V2 configuration management system for a YDB cluster based on Node Warden, storage nodes, and their PDisks. All V2 configuration changes, including initial setup, pass through it.

The Leader's Role

The central element of the Distconf system is the **leader**: the only node in the cluster that currently has the authority to initiate and coordinate configuration changes. Having a single decision-making point eliminates races and conflicts, ensuring that changes are applied consistently. If the current leader fails, the cluster automatically starts the process of electing a new leader. Later in this section, we will look at how elections work and how the leader manages the cluster.

Quorum and Source of Truth

A key element of Distconf's reliability is storing the configuration directly on PDisks. Any change is considered successfully applied only when it is written to the metadata area on a **quorum** of disks. This means that even if some nodes or disks fail, the system can recover the current and consistent configuration by reading it from the remaining ones. It is the quorum storage on PDisks, not the state of any single node, that is the single source of truth about the cluster configuration. This mechanism is implemented in `distconf_persistent_storage.cpp`.

It is worth noting that writing to the PDisk metadata area for Distconf is a special low-level operation that occurs directly through `NodeWarden` and **does not use** the standard data path through `DSProxy`.

The quorum mechanism itself is hierarchical and implemented in `distconf_quorum.h`. The basic principle is "majority of majorities". To make a decision (for example, to apply a new configuration), the following is required:

1. A majority of responding disks within each data center. A data center that meets this condition is considered "operational".
2. A majority of "operational" data centers in the cluster.

For configurations affecting static storage groups, the requirement is even stricter: each static group must confirm its own internal quorum of VDisks in accordance with its fault tolerance scheme.

Binding Process and Leader Election

Leader election in Distconf is carried out through the `Binding` process, described in `distconf_binding.cpp`.

1. **Discovery**: Storage nodes get a full list of static cluster nodes through the standard `TEvInterconnect::TEvNodesInfo` mechanism.
2. **Building the binding tree**: Each node initiates a `bind` to a random node from the common list to which it is not yet subordinate. This process forms an acyclic graph, which in intermediate stages is a forest (a set of disconnected trees), and eventually a single tree spanning all nodes.
3. **Cycle prevention and topology exchange**: The cycle prevention mechanism is based on the constant exchange of information about the current topology. When node `A` tries to connect to node `B`, it sends its entire known subtree to `B`. Node `B` rejects the request if `A` is already its descendant. In case of a simultaneous attempt at a mutual connection, the conflict is resolved in favor of the node with the greater `NodeId` value.
4. **Leader determination**: In the process of merging trees, there inevitably remains one node that cannot connect to anyone other node because all other nodes are already in its subtree. This node becomes the root of the final tree and is declared the **leader**.

If the current leader fails, the `Binding` process is restarted, and the cluster elects a new leader.

Scatter/Gather — Command Propagation Mechanism

To manage the cluster, the leader uses the `Scatter/Gather` mechanism (`distconf_scatter_gather.cpp`), which operates on top of the tree built during the `Binding` process.

- **Scatter**
  When the leader needs to send a command to all nodes (for example, to propose a new configuration), it sends it to its direct children in the tree. Each node, upon receiving the command, retransmits it to its children. This is how the command is efficiently propagated throughout the tree to the leaves.
- **Gather**
  After executing the command, the nodes must report the result. The responses are collected in reverse order: the leaves send the results to their parents, who in turn aggregate them and send them up. As a result, the leader receives a generalized result from the entire cluster.

This mechanism is used for distributed operations, including a two-phase commit when changing the configuration. Scatter and gather tasks (`TScatterTasks`) are tracked by the leader to monitor the execution of operations.

Finite-State Machine (FSM) and Change Lifecycle

The entire configuration change process on the leader is managed by a finite-state machine (FSM), implemented in `distconf_fsm.cpp`. The FSM ensures that only one configuration change operation is performed at a time, preventing races and conflicts.

When a change request is received, the FSM transitions to the `IN_PROGRESS` state, blocking new requests.

The leader uses `Scatter/Gather` to perform a two-phase commit: first, it sends a Propose, and after receiving a quorum of confirmations, a Commit command.

After the successful completion or cancellation of the operation, the FSM returns to the `RELAX` state, allowing the processing of subsequent requests.

Configuration Management via InvokeOnRoot

`TEvNodeConfigInvokeOnRoot` requests are a unified mechanism for any cluster configuration changes. These commands can be initiated by both a **system administrator** (for example, via the CLI) and **other YDB components** in automatic mode (for example, by the `BlobStorageController` tablet during the `SelfHeal` process).

Regardless of the source, any such request is processed according to the same scenario:

1. The request is delivered to the Distconf leader node (if it was not sent to it, it is redirected automatically).
2. The leader starts the FSM and performs the Scatter/Gather process as described above.

This mechanism ensures that any configuration change, regardless of its nature, goes through a strict validation and quorum confirmation procedure.

The main commands supported by this mechanism:

- `UpdateConfig`: Make partial changes to the current configuration. The changes are transmitted as a `TStorageConfig` protobuf message.
- `QueryConfig`: Request the current and proposed configuration. The response contains `TStorageConfig` protobuf messages.
- `ReplaceStorageConfig`: Replace the current configuration with a new one, passed as YAML.
- `FetchStorageConfig`: Return the loaded YAML cluster configuration.
- `ReassignGroupDisk`: Replace a disk in a static storage group.
- `StaticVDiskSlain`: Handle a VDisk failure event in a static group.
- `DropDonor`: Remove donor disks after data migration is complete.
- `BootstrapCluster`: Initiate the initial creation of the cluster.

Integration and Additional Mechanisms

In addition to the main processes, Distconf works closely with other system components:

- **Distributed Storage Controller**

  The DS-controller receives configuration changes from Distconf and uses them for the operation of the distributed storage.
- **Database nodes**

  Database nodes subscribe to `TEvNodeWardenDynamicConfigPush` events to receive real-time configuration updates.
- **SelfHeal**

  When using Distconf for a static group, SelfHeal works similarly to dynamic groups.
- **Local YAML files on nodes**

  These are stored in the directory specified by the `ydbd --config-dir` server startup argument and are updated upon receiving information about each configuration change. These files are needed when a node starts to discover PDisks and other cluster nodes, as well as to establish initial network connections. The data in these files may be outdated, especially if the node has been shut down for a long time.

Basic Distconf workflow

1. When a node starts, Distconf tries to read the configuration from local PDisks.
2. It connects to a random storage node to check if the configuration is up-to-date.
3. If it fails to connect, but has a quorum of connected nodes, the node becomes the leader.
4. The leader tries to initiate the initial cluster setup, if allowed.
5. The leader sends the current configuration to all nodes via Scatter/Gather.
6. Nodes save the received configuration in the local PDisk metadata area and in the `--config-dir` directories.

Final configuration distribution

Storage location	Contains
`TPDiskMetadataRecord` on a quorum of PDisks	The true current configuration
Local `--config-dir` directory	Initial YAML for startup (may be outdated)
Console	Up-to-date copy (with minimal delay)
DS-controller	Subset of the configuration required for distributed storage

Bridge mode

> **Feature of Yandex Enterprise Database**
>
> This functionality is available only in the Yandex Enterprise Database. In the open-source version of YDB it is absent.

For a general description of the mode and use cases, see Bridge cluster operation mode. This article is intended for YDB developers and contributors and describes the technical implementation details of bridge mode.

YDB supports an arbitrary number of piles; for simplicity, the following discussion considers the case of two piles, referred to as pile A and pile B.

Configuration storage

Distconf stores the configuration of two static groups, two sets of state storage, as well as scheme board and board. It also stores the cluster operating mode.

The configuration of static groups, state storage, and boards uses a storage scheme with two quorum sets:

- Quorum A — includes only nodes from pile A. A write is considered successful if it has been applied on a majority of nodes in pile A and on the quorum of disks of the static group of pile A;
- Quorum B — includes only nodes from pile B. A write is considered successful if it has been applied on a majority of nodes in pile B and on the quorum of disks of the static group of pile B.

If one of the piles fails, writing to the corresponding quorum becomes impossible.

Distconf uses both quorums (A and B) to store configuration:

- Configuration with mode `PRIMARY/DISCONNECTED` is written only to quorum A;
- Configuration with mode `DISCONNECTED/PRIMARY` is written only to quorum B;
- All other modes are written to both quorums — A and B.

When performing failover, the administrator puts the cluster into mode `PRIMARY/DISCONNECTED` or `DISCONNECTED/PRIMARY`. In this mode, the configuration of shared components is stored only on the quorum of the `PRIMARY` pile.

To start recovery, the administrator puts the pair of the live pile and the failed pile into mode `PRIMARY/NOT_SYNCHRONIZED`. This configuration is stored on all quorums.

Writing configuration is forbidden if the new configuration is incompatible with the last one stored on the node. For example, a transition from `PRIMARY/DISCONNECTED` to `DISCONNECTED/PRIMARY` is not allowed. This prevents a situation where pile A and pile B simultaneously switch to modes that both claim the `PRIMARY` role.

To prevent pile A and pile B from interacting when configurations conflict, interconnect session establishment is blocked when such inconsistencies are detected on nodes.

When operating in bridge mode, Distconf creates a subtree in each pile, and the roots of these subtrees are attached to the Distconf leader. This minimizes the reconfiguration of Distconf node links when a pile is lost.

Starting static groups

The configuration of all static groups is stored in Distconf. The root of the Distconf subtree in each pile can obtain the configuration and decide which quorum is required to apply it based on that configuration. A `DISCONNECTED` pile does not participate in the quorum; therefore, applying configuration without a `DISCONNECTED` pile requires quorums A and B; applying configuration `PRIMARY/DISCONNECTED` requires only quorum A; applying configuration `DISCONNECTED/PRIMARY` requires only quorum B.

When the required quorum is formed, Distconf applies the configuration, and the Node Warden starts the VDisk and PDisk of the static groups.

Access to the static group is provided by the `dsproxy-proxy` service, which exposes the dsproxy interface to tablets and performs operations synchronously with both groups in the pair.

The dsproxy-proxy of a static group changes its operating mode according to the configuration received from Distconf. The dsproxy-proxy configuration is stored in the configuration of the shared components.

dsproxy-proxy behavior

In mode `PRIMARY/SYNCHRONIZED`, writes are performed synchronously to both groups; reads are performed from the `PRIMARY`, and each read issues a ping request to the second group to ensure timely detection of link failure or a change in that group's operating mode.

In mode `PRIMARY/DISCONNECTED`, reads and writes are performed only in the `PRIMARY` group.

In mode `PRIMARY/NOT_SYNCHRONIZED`, dsproxy-proxy supports several submodes that determine which kinds of writes are performed in the `NOT_SYNCHRONIZED` group:

- `WRITE_KEEP` — only `KEEP` flags are written;
- `WRITE_KEEP_BARRIER_DONOTKEEP` — barriers and `KEEP` and `DO_NOT_KEEP` flags are written;
- `WRITE_KEEP_BARRIER_DONOTKEEP_DATA` — data, barriers, and `KEEP` and `DO_NOT_KEEP` flags are written.

The dsproxy-proxy configuration includes:

- mode;
- submode;
- `PPGeneration` — the generation number of the dsproxy-proxy configuration;
- the generation number of the dsproxy group configuration from pile A;
- the generation number of the dsproxy group configuration from pile B.

When the dsproxy-proxy configuration changes, the configuration generation number and the group configuration generation are always updated for the `PRIMARY` group; for the others, they are updated except when in the `DISCONNECTED` state.

The group configuration includes:

- mode;
- submode;
- `PPGeneration` — the generation number of the controlling dsproxy-proxy configuration.

On each operation, the group checks the mode, submode, and `PPGeneration`. This ensures that operations from stale dsproxy-proxy instances that missed a mode or submode change are not executed, and avoids cross-interaction when two `PRIMARY` instances appear.

To transition from mode `PRIMARY/NOT_SYNCHRONIZED` to `PRIMARY/SYNCHRONIZED`, synchronization is performed. During synchronization, missing data is copied from the `PRIMARY` group to the `NOT_SYNCHRONIZED` group, and lock information is copied in both directions.

Bootstrap of BSController and other system tablets

The node list for starting a tablet need only specify one set of nodes that includes a sufficient number of nodes from each pile. The Bootstrapper must analyze the state storage proxy-proxy configuration and use it to determine whether the tablet can be started on a given node. If the tablet cannot be started on the node (because the node is not in the `PRIMARY` pile), the Bootstrapper must walk the tree of Bootstrappers connected to it and assign as the launcher a Bootstrapper from the `PRIMARY` pile. If no such Bootstrapper exists, the current Bootstrapper must not start the tablet.

Configuration and startup of other tablets

Dynamic storage groups are managed by the BS Controller. Instead of a single regular storage pool each time, three storage pools are created: in pool A — groups from pile A; in pool B — groups from pile B; in pool C — proxy-proxy groups formed from pairs of groups from pools A and B. Tablets managed by Hive use only the proxy-proxy groups from pool C.

Tablets are started only in the `PRIMARY` or `PROMOTED` pile. Moving tablets from `PRIMARY` to `PROMOTED` is done as smoothly as possible; moving tablets from `DISCONNECTED` to `PRIMARY` is done as quickly as possible.

Interconnect behavior

In the interconnect, metadata exchange is separated from session establishment. Metadata exchange occurs before session establishment and allows a pair of nodes to determine the set of interconnect features in use and to exchange configuration. The interconnect with nodes from the disconnected half of the cluster is torn down, and no connection attempts are made. Connection establishment is only possible after the pile is moved to state `PRIMARY/NOT_SYNCHRONIZED`.

Want to join the YDB development team?

Learn about the YDB team and open positions, as well as opportunities for students.

Spilling Service

Overview

The **Spilling Service** is an actor service that provides temporary storage for data blobs in the YDB system. The service operates as a key-value store where clients can save data using a unique identifier and later retrieve it with that identifier.

Architecture

Main Components

- **Task queue**: The service maintains an internal queue of read and write operations. All spilling requests are placed in this queue and processed asynchronously.
- **Thread pool**: A pool of worker threads is used to perform I/O operations.
- **File management**: The service automatically creates, deletes, and manages files on disk.
- **Resource monitoring**: The service monitors disk space usage, the number of active operations, and other performance metrics.

Data Storage

Data is saved in files on the local file system. The Spilling Service ensures:

- distribution of records across files
- file deletion
- data lifecycle management

In case of an unexpected restart, obsolete files are automatically deleted.

Component Interaction

System components are integrated with the Spilling Service and interact with it through actor system events, explained below.

Memory State Monitoring

Compute nodes continuously monitor memory state through the allocator. The allocator informs nodes about decreasing free memory volume. However, the system does not wait for complete memory exhaustion because the spilling process also requires additional memory resources for serialization and buffering.

Event Dispatch

When spilling is required, the compute component (data transfer system or compute core) performs the following actions:

1. Serializes data into a blob.
2. Generates a unique identifier for the blob.
3. Creates a spilling request with the blob and the generated identifier.
4. Sends the request to the Spilling Service.
5. Releases resources and enters waiting mode, allowing other tasks to utilize computational resources.

Waiting for Results

After sending the request, the compute component releases resources for other tasks and enters waiting mode, allowing the system to optimally utilize cluster computing resources until the external storage write is complete.

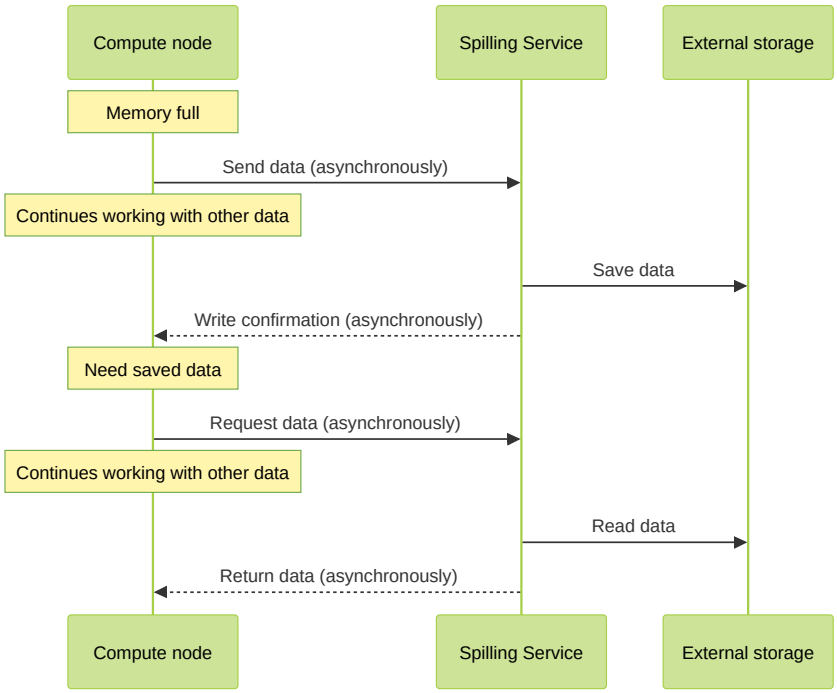
Response Handling

The Spilling Service processes the request and returns a write confirmation for the specified identifier or an error message. The compute component can continue only after receiving confirmation.

Data Reading

When data recovery is needed, the component sends a read request with the blob identifier. The Spilling Service reads data from external storage and returns a response with the recovered data. During data loading, freed computational resources are utilized to process other tasks.

Spilling Workflow Diagram



Configuration

Detailed information about configuring the Spilling Service is available in the Spilling configuration section.

See Also

- Spilling Concept
- Spilling configuration
- YDB monitoring
- Performance diagnostics

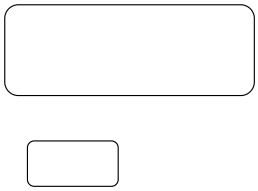
Overview of testing with load actors

Testing system performance is an important stage of adding changes to the YDB core. With load testing, you can:

- Determine the performance metrics and compare them to the values before the changes.
- Test how the system runs under high or peak loads.

The task to load a high-performance distributed system is non-trivial. A software developer often has to run multiple client instances to achieve the required server load. YDB implements a simple and convenient load generation mechanism: load actors. Actors are created and run directly on a cluster, which allows avoiding the use of additional resources for starting client instances. Load actors can be run on arbitrary cluster nodes: one, all, or only selected ones. You can create any number of actors on any node.

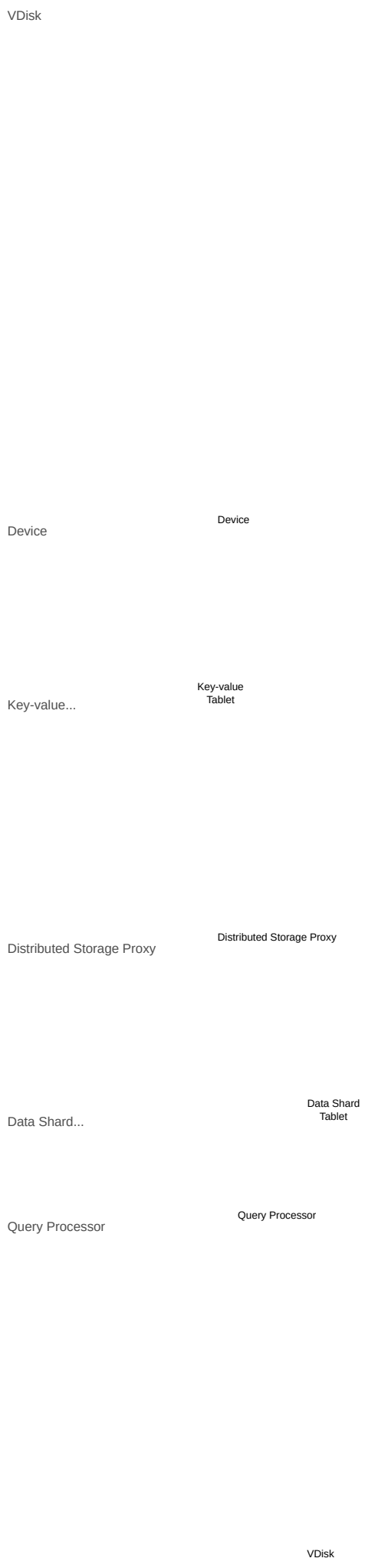
With load actors, you can test both the entire system and its individual components:



PDisk

PDisk

VDisk


VDisk
Device
Device
Key-value...
Key-value
Tablet
Distributed Storage Proxy
Distributed Storage Proxy
Data Shard...
Data Shard
Tablet
Query Processor
Query Processor
VDisk

VDisk

PDisk

PDisk

Distributed Storage Group

Distributed Storage Group

KqpLoad

KqpLoad

KeyValueLoad

KeyValueLoad

StorageLoad

StorageLoad

VDiskLoad

VDiskLoad

PDiskWriteLoad, PDiskReadLoad, PDiskLogLoad...

Load actors

YDB components

DeviceText is not SVG - cannot display

For example, you can generate a load on Distributed Storage without using tablet and Query Processor layers. This lets you test different system layers separately and find bottlenecks in an efficient way. By combining a variety of actor types, you can run different types of load.

> **Note**
>
> This feature is under development; the source code is available in the *main* branch of the YDB repository. To learn how to build YDB from the source code, see the instructions.

Actor types

Type	Description
KqpLoad	Generates a load on the Query Processor layer and loads all cluster components.
KeyValueLoad	Loads a key-value tablet.
StorageLoad	Loads Distributed Storage without using tablet and Query Processor layers.
VDiskLoad	Tests the performance of writes to the VDisk.
PDiskWriteLoad	Tests the performance of writes to the PDisk.
PDiskReadLoad	Tests the performance of reads from the PDisk.
PDiskLogLoad	Tests if cuts from the middle of the PDisk log are correct.
MemoryLoad	Allocates memory, useful when testing the logic.

Stop	Stops either all or only the specified actors.

Running a load

You can run load using the following tools:

- Cluster YDB UI: Allows you to create, based on a configuration, and start a load actor either on the current node or all tenant nodes at once.
- The `ydbd` utility: Allows you to send the actor configuration to any cluster node specifying the nodes to create and run the actor on.

The use case described below shows how to create and run the KqpLoad actor. The actor accesses the `/slice/db` database as a key-value store using 64 threads with a 30-second load. Before the test, the actor creates the necessary tables and deletes them once the test is completed. When being created, the actor is automatically assigned a tag. The same tag will be assigned to the test result.

YDB UI

1. Open the page for managing load actors on the desired node (for example, `http://<address>:8765/actors/load`, where `address` is the address of the cluster node to run the load on).
2. Paste the actor configuration into the input/output field:

```
KqpLoad: {
    DurationSeconds: 30
    WindowDuration: 1
    WorkingDir: "/slice/db"
    NumOfSessions: 64
    UniformPartitionsCount: 1000
    DeleteTableOnFinish: 1
    WorkloadType: 0
    Kv: {
        InitRowCount: 1000
        PartitionsByLoad: true
        MaxFirstKey: 18446744073709551615
        StringLen: 8
        ColumnsCnt: 2
        RowsCnt: 1
    }
}
```

3. To create and run the actor, click:

- **Start new load on current node**: Runs the load on the current node.
- **Start new load on all tenant nodes**: Runs the load on all the tenant nodes at once.

You'll see the following message in the input/output field:

```
{"status":"OK","tag":1}
```

- `status`: Load run status.
- `tag`: Tag assigned to the load.

CLI

1. Create an actor configuration file:

```
KqpLoad: {
    DurationSeconds: 30
    WindowDuration: 1
    WorkingDir: "/slice/db"
    NumOfSessions: 64
    UniformPartitionsCount: 1000
    DeleteTableOnFinish: 1
    WorkloadType: 0
    Kv: {
        InitRowCount: 1000
        PartitionsByLoad: true
        MaxFirstKey: 18446744073709551615
        StringLen: 8
        ColumnsCnt: 2
        RowsCnt: 1
    }
}
```

2. Start the actor:

```
curl <endpoint>/actors/load -H "Content-Type: application/x-protobuf-text" --data mode=start --data all_nodes
=<start_on_all_nodes> --data config="$(cat proto_file)"
```

- `endpoint`: Node HTTP endpoint (for example, `http://<address>:<port>`, where `address` is the node address and `port` is the node HTTP port).
- `proto_file`: Path to the actor configuration file.
- `start_on_all_nodes`: `true` to start load on all nodes of a tenant, `false` to start load only on node with given `endpoint`.

Viewing test results

You can view the test results using the YDB UI. For a description of output parameters, see the documentation of the respective actor.

YDB UI

1. Open the page for managing load actors on the desired node (for example, `http://<address>:<port>/actors/load`, where `address` is the node address and `port` is the HTTP port used for monitoring the node under load).
2. Click **Results**.

This shows the results of completed tests. Find the results with the appropriate tag.

Tag# 1

Finish reason# OK called StartDeathProcess
Finish time# 2022-12-02T10:03:45.207002Z

Window	Txs	Txs/Sec	Errors	p50(ms)	p95(ms)	p99(ms)	pMax(ms)
total	333324	11110.8	0	4.895	6.303	11.455	64.511
30	10377	10377	0	4.927	10.111	18.303	34.559
29	11101	11101	0	5.055	6.015	6.527	8.319
28	7618	7618	0	5.023	7.903	19.711	64.511
27	10394	10394	0	4.895	10.623	18.815	36.351
26	11134	11134	0	5.055	5.983	6.463	7.519
25	11203	11203	0	5.023	6.111	7.871	13.567

Query Processor Load

Runs general performance testing for the YDB cluster by loading all components via the Query Processor layer. The load is similar to that from the workload YDB CLI subcommand, but it is generated from within the cluster.

You can run two types of load:

- **Stock**: Simulates a warehouse of an online store: creates multi-product orders, gets a list of orders per customer.
- **Key-value**: Uses the DB as a key-value store.

Before this test, the necessary tables are created. After it's completed, they are deleted.

Actor parameters

The basic actor parameters are described below. For the full list of parameters, see the load_test.proto file in the YDB Git repository.

Parameter	Description
`DurationSeconds`	Load duration in seconds.
`WindowDuration`	Statistics aggregation window duration.
`WorkingDir`	Path to the directory to create test tables in.
`NumOfSessions`	The number of parallel threads creating the load. Each thread writes data to its own session.
`DeleteTableOnFinish`	Set it to `False` if you do not want the created tables deleted after the load stops. This might be helpful when a large table is created upon the actor's first run, and then queries are made to that table.
`UniformPartitionsCount`	The number of partitions created in test tables.
`WorkloadType`	Type of load. For Stock: • `0`: InsertRandomOrder. • `1`: SubmitRandomOrder. • `2`: SubmitSameOrder. • `3`: GetRandomCustomerHistory. • `4`: GetCustomerHistory. For Key-Value: • `0`: UpsertRandom. • `1`: InsertRandom. • `2`: SelectRandom.
`Workload`	Kind of load. `Stock`: • `ProductCount`: Number of products. • `Quantity`: Quantity of each product in stock. • `OrderCount`: Initial number of orders in the database. • `Limit`: Minimum number of shards for tables. `Kv`: • `InitRowCount`: Before load is generated, the load actor writes the specified number of rows to the table. • `StringLen`: Length of the `value` string. • `ColumnsCnt`: Number of columns to use in the table. • `RowsCnt`: Number of rows to insert or read per SQL query.

Examples

The following actor runs a stock load on the `/slice/db` database by making simple UPSERT queries of `64` threads during `30` seconds.

```
KqpLoad: {
    DurationSeconds: 30
    WindowDuration: 1
    WorkingDir: "/slice/db"
    NumOfSessions: 64
    UniformPartitionsCount: 1000
    DeleteTableOnFinish: 1
    WorkloadType: 0
    Stock: {
        ProductCount: 100
        Quantity: 1000
        OrderCount: 100
        Limit: 10
    }
}
```

As a result of the test, the number of successful transactions per second, the number of transaction execution retries, and the number of errors are output.

KeyValueLoad

Loads a key-value tablet.

Actor configuration

```
message TKeyValueLoad {
    message TWorkerConfig {
        optional string KeyPrefix = 1;
        optional uint32 MaxInFlight = 2;
        optional uint32 Size = 11; // data size, bytes
        optional bool IsInline = 9 [default = false];
        optional uint32 LoopAtKeyCount = 10 [default = 0]; // 0 means "do not loop"
    }
    optional uint64 Tag = 1;
    optional uint64 TargetTabletId = 2;
    optional uint32 DurationSeconds = 5;
    repeated TWorkerConfig Workers = 7;
}
```

StorageLoad

Tests the read/write performance to and from Distributed Storage. The load is generated on Distributed Storage directly without using any tablet and Query Processor layers. When testing write performance, the actor writes data to the specified storage group. To test read performance, the actor first writes data to the specified storage group and then reads the data. After the load is removed, all the data written by the actor is deleted.

You can generate three types of load:

- **Continuous:** The actor ensures that the specified number of requests are running concurrently. To generate a continuous load, set a zero interval between requests (e.g., `WriteIntervals: { Weight: 1.0 Uniform: { MinUs: 0 MaxUs: 0 } }`), while keeping the `MaxInFlightWriteRequests` parameter value different from zero and omit the `WriteHardRateDispatcher` parameter.
- **Interval:** The actor runs requests at specific intervals. To generate an interval load, set a non-zero interval between requests, e.g., `WriteIntervals: { Weight: 1.0 Uniform: { MinUs: 50000 MaxUs: 50000 } }` and don't set the `WriteHardRateDispatcher` parameter. The maximum number of in-flight requests is set by the `InFlightWrites` parameter (0 means unlimited).
- **Hard rate:** The actor runs requests at certain intervals, but the interval length is adjusted to maintain a configured request rate per second. If the duration of the load is limited by `LoadDuration` than the request rate may differ between start and finish of the workload and will adjust gradually throughout all the main load cycle. To generate a load of this type, set the parameters of hard rate load (parameter `WriteHardRateDispatcher`). Note that if this parameter is set, the hard rate type of load will be launched, regardless the value of the `WriteIntervals` parameter. The maximum number of in-flight requests is set by the `InFlightWrites` parameter (0 means unlimited).

Actor parameters

The basic actor parameters are described below. For the full list of parameters, see the load_test.proto file in the YDB Git repository.

Parameter	Description
`DurationSeconds`	Load duration. The timer starts upon completion of the initial data allocation.
`Tablets`	The load is generated on behalf of a tablet with the following parameters: • `TabletId`: Tablet ID. It must be unique for each load actor across all the cluster nodes. This parameter and `TabletName` are mutually exclusive. • `TabletName`: Tablet name. If the parameter is set, tablets' IDs will be assigned automatically, tablets launched on the same node with the same name will be given the same ID, tablets launched on different nodes will be given different IDs. • `Channel`: Tablet channel. • `GroupId`: ID of the storage group to get loaded. • `Generation`: Tablet generation.
`WriteSizes`	Size of the data to write. It is selected randomly for each request from the `Min` - `Max` range. You can set multiple `WriteSizes` ranges, in which case a value from a specific range will be selected based on its `Weight`.
`WriteHardRateDispatcher`	Setting up the parameters of load with hard rate for write requests. If this parameter is set than the value of `WriteIntervals` is ignored.
`WriteIntervals`	Setting up the parameters for probabilistic distribution of intervals between the records loaded at intervals (in milliseconds). You can set multiple `WriteIntervals` ranges, in which case a value from a specific range will be selected based on its `Weight`.
`MaxInFlightWriteRequests`	The maximum number of write requests being processed simultaneously.
`ReadSizes`	Size of the data to read. It is selected randomly for each request from the `Min` - `Max` range. You can set multiple `ReadSizes` ranges, in which case a value from a specific range will be selected based on its `Weight`.
`WriteHardRateDispatcher`	Setting up the parameters of load with hard rate for read requests. If this parameter is set than the value of `ReadIntervals` is ignored.
`ReadIntervals`	Setting up the parameters for probabilistic distribution of intervals between the queries loaded by intervals (in milliseconds). You can set multiple `ReadIntervals` ranges, in which case a value from a specific range will be selected based on its `Weight`.
`MaxInFlightReadRequests`	The maximum number of read requests being processed simultaneously.
`FlushIntervals`	Setting up the parameters for probabilistic distribution of intervals (in microseconds) between the queries used to delete data written by the write requests in the main load cycle of the StorageLoad actor. You can set multiple `FlushIntervals` ranges, in which case a value from a specific range will be selected based on its `Weight`. Only one flush request will be processed concurrently.
`PutHandleClass`	Class of data writes to the disk subsystem. If the `TabletLog` value is set, the write operation has the highest priority.
`GetHandleClass`	Class of data reads from the disk subsystem. If the `FastRead` is set, the read operation is performed with the highest speed possible.
`Initial allocation`	Setting up the parameters for initial data allocation. It defines the amount of data to be written before the start of the main load cycle. This data can be read by read requests along with the data written in the main load cycle.

Write requests class

Class	Description
`TabletLog`	The highest priority of write operation.

`AsyncBlob`	Used for writing SSTables and their parts.
`UserData`	Used for writing user data as separate blobs.

Read requests class

Class	Description
`AsyncRead`	Used for reading compacted tablets' data.
`FastRead`	Used for fast reads initiated by user.
`Discover`	Reads from Discover query.
`LowRead`	Low priority reads executed on the background.

Parameters of probabilistic distribution

An interval written as a repeated `TIntervalInfo` field is calculated by the following algorithm:

- An element from the `TIntervalInfo` array is selected at random with the probability proportionate to its weight.
- For an element of the `TIntervalUniform` type, the value is chosen with equal probability in the range Min-Max. `Min-Max` (if `MinMs/MaxMs` is used, the value is in milliseconds; if `MinUs/MaxUs` is used, the value is in microseconds).
- For an element of the `TIntervalPoisson` type, the interval is selected using the formula `Min(log(-x / Frequency), MaxIntervalMs)`, where `x` is a random value in the interval `[0, 1]`. As a result, the intervals follow the Poisson distribution with the given `Frequency`, but with the interval within `MaxIntervalMs`.

A similar approach is used for the probabilistic distribution of the size of the written data. However, in this case, only the data size follows a uniform probability distribution, within the interval `[Min, Max]`.

Parameters of load with hard rate

Parameter	Description
`RequestRateAtStart`	Requests per second at the moment of load start. If load duration limit is not set then the request rate will remain the same and equal to the value of this parameter.
`RequestRateOnFinish`	Requests per second at the moment of load finish.

Parameters of initial data allocation

Parameter	Description
`TotalSize`	Total size of allocated data. This parameter and `BlobsNumber` are mutually exclusive.
`BlobsNumber`	Total number of allocated blobs.
`BlobSizes`	Size of the blobs to write. It is selected randomly for each request from the `Min` - `Max` range. You can set multiple `WriteSizes` ranges, in which case a value from a specific range will be selected based on its `Weight`.
`MaxWritesInFlight`	Maximum number of simultaneously processed write requests. If this parameter is not set then the number of simultaneously processed requests is not limited.
`MaxWriteBytesInFlight`	Maximum number of total amount of simultaneously processed write requests' data. If this parameter is not set then the total amount of data being written concurrently is unlimited.
`PutHandleClass`	Class of data writes to the disk subsystem.
`DelayAfterCompletionSec`	The amount of time in seconds the actor will wait upon completing the initial data allocation before starting the main load cycle. If its value is `0` or not set the load will start immediately after the completion of the data allocaion.

An interval written as a repeated `TIntervalInfo` field is calculated by the following algorithm:

- An element from the `TIntervalInfo` array is selected at random with the probability proportionate to its weight.
- For an element of the `TIntervalUniform` type, the value is chosen with equal probability in the range Min-Max. `Min-Max` (if `MinMs/MaxMs` is used, the value is in milliseconds; if `MinUs/MaxUs` is used, the value is in microseconds).
- For an element of the `TIntervalPoisson` type, the interval is selected using the formula `Min(log(-x / Frequency), MaxIntervalMs)`, where `x` is a random value in the interval `[0, 1]`. As a result, the intervals follow the Poisson distribution with the given `Frequency`, but with the interval within `MaxIntervalMs`.

A similar approach is used for the probabilistic distribution of the size of the written data. However, in this case, only the data size follows a uniform probability distribution, within the interval `[Min, Max]`.

Examples

Write load

The following actor writes data to the group with the ID `2181038080` during `60` seconds. The size per write is `4096` bytes, the number of in-flight requests is no more than `256` (continuous load):

```
StorageLoad: {
    DurationSeconds: 60
    Tablets: {
        Tablets: { TabletId: 1000 Channel: 0 GroupId: 2181038080 Generation: 1 }
        WriteSizes: { Weight: 1.0 Min: 4096 Max: 4096 }
        WriteIntervals: { Weight: 1.0 Uniform: { MinUs: 0 MaxUs: 0 } }
```

```
        MaxInFlightWriteRequests: 256
        FlushIntervals: { Weight: 1.0 Uniform: { MinUs: 10000000 MaxUs: 10000000 } }
        PutHandleClass: TabletLog
    }
}
```

When viewing test results, the following values should be of most interest to you:

- `Writes per second`: Number of writes per second, e.g., `28690.29`.
- `Speed@ 100%`: 100 percentile of write speed in MB/s, e.g., `108.84`.

Read load

To generate a read load, you need to write data first. Data is written by requests of `4096` bytes every `50` ms with no more than `1` in-flight request (interval load). If a request fails to complete within `50` ms, the actor will wait until it is complete and run another request in `50` ms. Data older than `10` s is deleted. Data reads are performed by requests of `4096` bytes with `16` in-flight requests allowed (continuous load):

```
StorageLoad: {
    DurationSeconds: 60
    Tablets: {
        Tablets: { TabletId: 5000 Channel: 0 GroupId: 2181038080 Generation: 1 }
        WriteSizes: { Weight: 1.0 Min: 4096 Max: 4096}
        WriteIntervals: { Weight: 1.0 Uniform: { MinUs: 50000 MaxUs: 50000 } }
        MaxInFlightWriteRequests: 1

        ReadSizes: { Weight: 1.0 Min: 4096 Max: 4096 }
        ReadIntervals: { Weight: 1.0 Uniform: { MinUs: 0 MaxUs: 0 } }
        MaxInFlightReadRequests: 16
        FlushIntervals: { Weight: 1.0 Uniform: { MinUs: 10000000 MaxUs: 10000000 } }
        PutHandleClass: TabletLog
        GetHandleClass: FastRead
    }
}
```

When viewing test results, the following value should be of most interest to you:

- `ReadSpeed@ 100%`: 100 percentile of read speed in MB/s, e.g., `60.86`.

Read only

Before the start of the main load cycle the `1 GB` data block of blobs with sizes between `1 MB` and `5 MB` is allocated. To avoid overloading the system with write requests the number of simultaneously processed requests is limited by the value of `5`. After completing the initial data allocation the main cycle is launched. It consists of read requests sent with increasing rate: from `10` to `50` requests per second, the rate will increase gradually for `300` seconds.

```
StorageLoad: {
    DurationSeconds: 300
    Tablets: {
        Tablets: { TabletId: 5000 Channel: 0 GroupId: 2181038080 Generation: 1 }

        MaxInFlightReadRequests: 10
        GetHandleClass: FastRead
        ReadHardRateDispatcher {
                RequestsPerSecondAtStart: 10
                RequestsPerSecondOnFinish: 50
        }

        InitialAllocation {
                TotalSize: 1000000000
                BlobSizes: { Weight: 1.0 Min: 1000000 Max: 5000000 }
                MaxWritesInFlight: 5
        }
    }
}
```

Calculated percentiles will only represent the requests of the main load cycle and won't include write requests sent during the initial data allocation. The graphs in Monitoring should be of interest, for example, they allow to trace the request latency degradation caused by the increasing load.

VDiskLoad

Generates a write-only load on the VDisk. Simulates a Distributed Storage Proxy. The test outputs the VDisk write performance in operations per second.

Actor parameters

The basic actor parameters are described below. For the full list of parameters, see the load_test.proto file in the YDB Git repository.

Parameter	Description
`VDiskId`	Parameters of the VDisk used to generate load. • `GroupID`: Group ID. • `GroupGeneration`: Group generation. • `Ring`: Group ring ID. • `Domain`: Ring fail domain ID. • `VDisk`: Index of the VDisk in the fail domain.
`GroupInfo`	Description of the group hosting the loaded VDisk (of the appropriate generation).
`TabletId`	ID of the tablet that generates the load. It must be unique for each load actor.
`Channel`	ID of the channel inside the tablet that will be specified in the BLOB write and garbage collection commands.
`DurationSeconds`	The total test time in seconds; when it expires, the load stops automatically.
`WriteIntervals`	Setting up the parameters for probabilistic distribution of intervals between the records.
`WriteSizes`	Size of the data to write. It is selected randomly for each request from the `Min`-`Max` range. You can set multiple `WriteSizes` ranges, in which case a value from a specific range will be selected based on its `Weight`.
`InFlightPutsMax`	Maximum number of concurrent BLOB write queries against the VDisk (TEvVPut queries); if omitted, the number of queries is unlimited.
`InFlightPutBytesMax`	Maximum number of bytes in the concurrent BLOB write queries against the VDisk (TEvVPut-requests).
`PutHandleClass`	Class of data writes to the disk subsystem. If the `TabletLog` value is set, the write operation has the highest priority.
`BarrierAdvanceIntervals`	Setting up the parameters for probabilistic distribution of intervals between the advance of the garbage collection barrier and the write step.
`StepDistance`	Distance between the currently written step `Gen:Step` of the BLOB and its currently collected step. The higher is the value, the more data is stored. Data is written from `Step = X` and deleted from all the BLOBs where `Step = X - StepDistance`. The `Step` is periodically incremented by one (with the `BarrierAdvanceIntervals` period).

Parameters of probabilistic distribution

An interval written as a repeated `TIntervalInfo` field is calculated by the following algorithm:

- An element from the `TIntervalInfo` array is selected at random with the probability proportionate to its weight.
- For an element of the `TIntervalUniform` type, the value is chosen with equal probability in the range Min-Max. `Min-Max` (if `MinMs/MaxMs` is used, the value is in milliseconds; if `MinUs/MaxUs` is used, the value is in microseconds).
- For an element of the `TIntervalPoisson` type, the interval is selected using the formula `Min(log(-x / Frequency), MaxIntervalMs)`, where `x` is a random value in the interval `[0, 1]`. As a result, the intervals follow the Poisson distribution with the given `Frequency`, but with the interval within `MaxIntervalMs`.

A similar approach is used for the probabilistic distribution of the size of the written data. However, in this case, only the data size follows a uniform probability distribution, within the interval `[Min, Max]`.

PDiskWriteLoad

Tests the performance of writes to the PDisk. The load is generated on behalf of a VDisk. The actor creates chunks on the specified PDisk and writes random data to them. After the load stops, the data written by the actor is deleted.

You can generate two types of load:

- **Continuous:** The actor ensures the specified number of requests are run concurrently. To generate a continuous load, set a zero interval between requests, e.g., `IntervalMsMin: 0` and `IntervalMsMax: 0`, while keeping the `InFlightWrites` parameter different from zero.
- **Interval:** The actor runs requests at preset intervals. To generate interval load, set a non-zero interval between requests, e.g., `IntervalMsMin: 10` and `IntervalMsMax: 100`. You can set the maximum number of in-flight requests using the `InFlightWrites` parameter. If its value is `0`, their number is unlimited.

Actor parameters

The basic actor parameters are described below. For the full list of parameters, see the load_test.proto file in the YDB Git repository.

Parameter	Description
`PDiskId`	ID of the Pdisk being loaded on the node.
`PDiskGuid`	Globally unique ID of the PDisk being loaded.
`VDiskId`	The load is generated on behalf of a VDisk with the following parameters: • `GroupID`: Group ID. • `GroupGeneration`: Group generation. • `Ring`: Group ring ID. • `Domain`: Ring fail domain ID. • `VDisk`: Index of the VDisk in the fail domain.
`Chunks`	Chunk parameters. `Slots`: Number of slots per chunk, determines the write size. You can specify multiple `Chunks`, in which case a specific chunk to write data to is selected based on its `Weight`.
`DurationSeconds`	Load duration in seconds.
`IntervalMsMin`, `IntervalMsMax`	Minimum and maximum intervals between requests under interval load, in milliseconds. The interval value is selected randomly from the specified range.
`InFlightWrites`	Number of simultaneously processed write requests.
`LogMode`	Logging mode. In `LOG_SEQUENTIAL` mode, data is first written to a chunk and then, once the write is committed, to a log.
`Sequential`	Type of writes. • `True`: Sequential. • `False`: Random.
`IsWardenlessTest`	Set it to `False` in case the PDiskReadLoad actor is run on the cluster; otherwise, e.g. when it is run during unit tests, set it to `True`.

Examples

The following actor writes data blocks of `32` MB during `120` seconds with `64` in-flight requests (continuous load):

```
PDiskWriteLoad: {
    PDiskId: 1000
    PDiskGuid: 2258451612736857634
    VDiskId: {
        GroupID: 11234
        GroupGeneration: 5
        Ring: 1
        Domain: 1
        VDisk: 3
    }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    DurationSeconds: 120
    IntervalMsMin: 0
    IntervalMsMax: 0
    InFlightWrites: 64
    LogMode: LOG_SEQUENTIAL
    Sequential: false
    IsWardenlessTest: false
}
```

When viewing test results, the following value should be of most interest to you:

- `Average speed since start`: Average write speed since start, in MB/s, e.g., `615.484013`.

PDiskReadLoad

Tests the performance of reads from the PDisk. The load is generated on behalf of a VDisk. The actor creates chunks on the specified PDisk, writes random data to them, and reads the data from them using the specified parameters. After the load stops, the data written by the actor is deleted.

You can generate two types of load:

- **Continuous:** The actor ensures the specified number of requests are run concurrently. To generate continuous load, set a zero interval between requests (e.g., `IntervalMsMin: 0` and `IntervalMsMax: 0`), while keeping the `InFlightReads` parameter different from zero.
- **Interval:** The actor runs requests at preset intervals. To generate interval load, set a non-zero interval between requests, e.g., `IntervalMsMin: 10` and `IntervalMsMax: 100`. You can set the maximum number of in-flight requests using the `InFlightReads` parameter. If its value is `0`, their number is unlimited.

Actor parameters

The basic actor parameters are described below. For the full list of parameters, see the load_test.proto file in the YDB Git repository.

Parameter	Description
`PDiskId`	ID of the Pdisk being loaded on the node.
`PDiskGuid`	Globally unique ID of the PDisk being loaded.
`VDiskId`	The load is generated on behalf of a VDisk with the following parameters: • `GroupID`: Group ID. • `GroupGeneration`: Group generation. • `Ring`: Group ring ID. • `Domain`: Ring fail domain ID. • `VDisk`: Index of the VDisk in the fail domain.
`Chunks`	Chunk parameters. `Slots`: Number of slots per chunk, determines the write size. You can specify multiple `Chunks`, in which case a specific chunk to read data from is selected based on its `Weight`.
`DurationSeconds`	Load duration in seconds.
`IntervalMsMin`, `IntervalMsMax`	Minimum and maximum intervals between requests under interval load, in milliseconds. The interval value is selected randomly from the specified range.
`InFlightReads`	Number of simultaneously processed read requests.
`Sequential`	Type of reads. • `True`: Sequential. • `False`: Random.
`IsWardenlessTest`	Set it to `False` in case the PDiskReadLoad actor is run on the cluster; otherwise, e.g. when it is run during unit tests, set it to `True`.

Examples

The following actor reads data blocks of 32 MB during 120 seconds with 64 in-flight requests (continuous load):

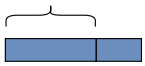
```
PDiskReadLoad: {
    PDiskId: 1000
    PDiskGuid: 2258451612736857634
    VDiskId: {
        GroupID: 11234
        GroupGeneration: 5
        Ring: 1
        Domain: 1
        VDisk: 3
    }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    DurationSeconds: 120
    IntervalMsMin: 0
    IntervalMsMax: 0
    InFlightReads: 64
    Sequential: false
    IsWardenlessTest: false
}
```

When viewing test results, the following value should be of most interest to you:

- `Average speed since start`: Average read speed since start, in MB/s, e.g., `1257.148154`.

PDiskLogLoad

All VDisks hosted on a certain PDisk log data about their own performance to the common PDisk log. VDisks gradually delete their obsolete data at the beginning of the log to free up disk space. Sometimes, after one VDisk completes logging data and before another one starts logging it, a section with useless obsolete data may appear. In this case, such data is deleted automatically, and the PDiskLogLoad actor will perform a test to check whether such an operation is running correctly.



SizeInterv...

SizeInterval VDisk1

BurstSize VDisk1

BurstSize VDisk1

BurstInter...

BurstInterval VDisk1

StorageDuration VDisk1

StorageDuration VDisk1

SizeInterval VDisk2

SizeInterv...
BurstInter...
BurstInterval VDisk2
StorageDuration VDisk2
StorageDuration VDisk2
BurstSize VDisk2
BurstSize VDisk2
Data that is being dele...
Data that is being deleted
Timeline,...Text is not SVG - cannot display
Timeline, bytes

Note

This ad-hoc actor is used for testing specific functionality. This is not a load actor. It is designed to check whether something works properly.

Actor parameters

The basic actor parameters are described below. For the full list of parameters, see the load_test.proto file in the YDB Git repository.

Parameter	Description
`PDiskId`	ID of the Pdisk being loaded on the node.
`PDiskGuid`	Globally unique ID of the PDisk being loaded.
`VDiskId`	Parameters of the VDisk used to generate load. • `GroupID`: Group ID. • `GroupGeneration`: Group generation. • `Ring`: Group ring ID. • `Domain`: Ring fail domain ID. • `VDisk`: Index of the VDisk in the fail domain.
`MaxInFlight`	Number of simultaneously processed requests.
`SizeIntervalMin`	Minimum size of log record in bytes.
`SizeIntervalMax`	Maximum size of log record in bytes.
`BurstInterval`	Interval between logging sessions in bytes.
`BurstSize`	Total amount of data to log per session, in bytes.
`StorageDuration`	Virtual time in bytes. Indicates how long the VDisk should store its data in the log.
`IsWardenlessTest`	Set it to `False` in case the PDiskReadLoad actor is run on the cluster; otherwise, e.g. when it is run during unit tests, set it to `True`.

Examples

The following actor simulates the performance of two VDisks. The first VDisk logs a message of `65536` bytes every `65536` bytes and deletes data that exceeds `1048576` bytes. The second one writes `1024` bytes as messages of `128` bytes every `2147483647` bytes and deletes data that exceeds `2147483647` bytes.

```
PDiskLogLoad: {
    Tag: 1
    PDiskId: 1
    PDiskGuid: 12345
    DurationSeconds: 60
    Workers: {
        VDiskId: {GroupID: 1 GroupGeneration: 5 Ring: 1 Domain: 1 VDisk: 1}
        MaxInFlight: 1
        SizeIntervalMin: 65536
        SizeIntervalMax: 65536
        BurstInterval: 65536
        BurstSize: 65536
        StorageDuration: 1048576
    }
    Workers: {
        VDiskId: {GroupID: 2 GroupGeneration: 5 Ring: 1 Domain: 1 VDisk: 1}
        MaxInFlight: 1
        SizeIntervalMin: 128
        SizeIntervalMax: 128
        BurstInterval: 2147483647
        BurstSize: 1024
        StorageDuration: 2147483647
    }
    IsWardenlessTest: false
}
```

The test passes if none of the cluster nodes got overloaded and the status of the PDisk in question is `Normal`. You can check this using the cluster YDB UI.

MemoryLoad

Allocates memory blocks of the specified size at certain intervals. After the load is removed, the allocated memory is released. Using this actor, you can test the logic, e.g., whether a certain trigger is fired when the RSS limit is reached.

> **Note**
>
> This ad-hoc actor is used for testing specific functionality. This is not a load actor. It is designed to check whether something works properly.

Actor parameters

Parameter	Description
`DurationSeconds`	Load duration in seconds.
`BlockSize`	Allocated block size in bytes.
`IntervalUs`	Interval between block allocations in microseconds.

Examples

The following actor allocates blocks of `1048576` bytes every `9000000` microseconds during `3600` seconds and takes up 32 GB while running:

```
MemoryLoad: {
    DurationSeconds: 3600
    BlockSize: 1048576
    IntervalUs: 9000000
}
```

Stop

Using this command, you can stop either entire load or only the specified part of it.

Actor parameters

Parameter	Description
`Tag`	Tag of the load actor to stop. You can view the tag in the cluster YDB UI.
`RemoveAllTags`	If this parameter value is set to `True`, all the load actors are stopped.

Examples

The command below stops the load tagged `123`:

```
Stop: {
    Tag: 123
}
```

To stop the entire load, run this command:

```
Stop: {
    RemoveAllTags: true
}
```

Chaos Testing

Chaos testing is a methodology for verifying the resilience of YDB by deliberately injecting failures into a running cluster. The goal is to ensure that YDB correctly survives real-world failures: node losses, network partitions, disk problems, and other abnormal situations. The **Nemesis** tool is responsible for injecting this chaos.

What Is Tested

Chaos testing verifies cluster behavior under the following types of failures:

Network Failures

- Node network isolation (blocking incoming and outgoing traffic)
- System time skew on nodes

Node Failures

- Forceful termination of cluster node processes
- Stopping and restarting nodes
- Suspending node processes

Tablet Failures

Tablets are the primary computational units of YDB. Resilience is verified by forcefully terminating various tablet types:

- **Coordinator** — distributed transaction coordinator
- **Hive** — tablet placement manager
- **BSController** — distributed storage controller
- **SchemeShard** — schema manager
- **DataShard** — data storage tablets
- **Mediator** — transaction mediator
- **PersQueue** — message queue tablets
- Other system tablets

Tablet rebalancing between nodes via Hive is also tested.

Disk Failures

- Safely taking a disk out of service on a node
- Cleaning up disks on nodes

Multi-Datacenter Cluster Scenarios

- Stopping all nodes in a single datacenter
- Network isolation of a datacenter

Bridge Mode Cluster Scenarios

- Stopping all nodes in a single pile
- Network isolation of a pile

Integration with Stress Testing

Chaos testing is typically run in conjunction with stress testing workloads from the ydb/tests/stress directory. This combination ensures that the cluster is tested under load conditions while experiencing various failure scenarios.

How Verification Works

While failures are being injected, the system checks two properties on demand:

- **Liveness** — the cluster remains available and continues to process requests
- **Safety** — no signs of data correctness violations or internal system invariant breaches appear in cluster logs and metrics

Failures are injected automatically on a schedule, and check results are aggregated and available for analysis.

The Nemesis Tool

YDB uses the **Nemesis** tool for chaos testing — a fault injection application located in the YDB repository on GitHub. It is deployed directly on the nodes of the cluster under test and manages fault injection according to a configured schedule.

> **Warning**
>
> Nemesis only works with YDB clusters that were deployed using the `ydbd_slice` utility.

Installation

Deploy Nemesis to your cluster:

```
# Single-file config (cluster.yaml contains both hosts and database template)
nemesis install --yaml-config-location /path/to/cluster.yaml

# Two-file config (separate cluster.yaml and databases.yaml)
nemesis install \
    --yaml-config-location /path/to/cluster.yaml \
    --database-config-location /path/to/databases.yaml
```

The first host in the cluster configuration becomes the orchestrator; all other hosts become agents. Services are deployed as systemd units and started automatically.

The primary way to observe a Nemesis run and inspect its results is the web UI served by the orchestrator. Open the URL printed at the end of `nemesis install` in a browser; by default it is available at:

```
http://<orchestrator_host>:31434/static/index.html
```

The UI displays:

- **Active faults** — currently injected faults and their execution logs, grouped by category (network, node, tablet, disk, datacenter, pile)
- **Schedules** — fault types registered for automatic injection, their configured intervals, and the next scheduled run
- **Manual controls** — buttons to inject individual faults on demand
- **Execution history** — past fault injections with timestamps and target hosts
- **Liveness checks** — cluster-wide health checks run by the orchestrator
- **Safety checks** — violation detectors over the local logs for the last 24 hours

The same data is available via the HTTP API on the orchestrator if you need to scrape it programmatically.

Interpreting Results

A Nemesis run is judged against two properties: **liveness** and **safety**. The pass/fail criteria are:

- **Pass** — throughout the run, all liveness checks report the cluster as healthy and all safety wardens return empty violation lists
- **Fail** — at least one liveness check reported the cluster as unavailable (cluster did not recover after a fault within the expected window), **or** at least one safety warden returned a non-empty list of violations (for example, errors or assertions in cluster logs, schemeshard inconsistencies, datashard invariant breaches)

In the UI, healthy checks appear in the violations list; failed checks display the violation messages produced by the corresponding warden.

To investigate a problem, open the failing check in the UI and read the listed violation messages.

For the meaning of individual checks and the exact patterns each warden looks for, see the Nemesis README.

Stopping Services

Stop all Nemesis services on the cluster:

```
nemesis stop --yaml-config-location /path/to/cluster.yaml
```

Extending Nemesis

To add a new chaos type (fault) or a new safety/liveness check, see the README. It describes how to implement and register new fault runners, planners, and warden classes.

YDB reference

This documentation section contains reference information about various aspects of YDB, including:

- YQL - Overview
- Using YDB UI
- Integrations YDB
- YDB CLI
- Reference for YDB SDK
- Languages and APIs
- Kafka API
- YDB Cluster Configuration
- Reference on YDB observability
- YDB DSTool overview
- ydbops utility overview
- YDB Docker container reference

YQL - Overview

YQL (YDB Query Language) is a universal declarative query language for YDB, a dialect of SQL. YQL has been natively designed for large distributed databases, and therefore has a number of differences from the SQL standard.

YDB tools support interfaces for sending YQL queries and receiving their execution results:

- YDB CLI
- YDB SDK

This documentation section contains the YQL reference that includes the sections:

- Data types with a description of data types used in YQL
- Syntax with a full list of YQL commands
- Built-in functions with a description of the available built-in functions

You can also take a tutorial to get familiar with the basic YQL commands, in the YQL tutorial section.

YQL - Overview

YQL (YDB Query Language) is a universal declarative query language for YDB, a dialect of SQL. YQL has been natively designed for large distributed databases, and therefore has a number of differences from the SQL standard.

YDB tools support interfaces for sending YQL queries and receiving their execution results:

- YDB CLI
- YDB SDK

This documentation section contains the YQL reference that includes the sections:

- Data types with a description of data types used in YQL
- Syntax with a full list of YQL commands
- Built-in functions with a description of the available built-in functions

You can also take a tutorial to get familiar with the basic YQL commands, in the YQL tutorial section.

YQL data types

This section contains articles on YQL data types:

- Simple/Primitive types
- Serial types
- Optional types
- Containers
- Special types
- Type casting
- Text representation of data types
- Data representation in JSON format

Primitive data types

The terms "simple", "primitive", and "elementary" data types are used as synonyms.

Numeric types

Type	Description	Notes
`Bool`	Boolean value.	
`Int8`	Signed integer. Valid values: from -2^7 to 2^7-1.	
`Int16`	Signed integer. Valid values: from -2^{15} to $2^{15}-1$.	
`Int32`	Signed integer. Valid values: from -2^{31} to $2^{31}-1$.	
`Int64`	Signed integer. Valid values: from -2^{63} to $2^{63}-1$.	
`Uint8`	Unsigned integer. Valid values: from 0 to 2^8-1.	
`Uint16`	Unsigned integer. Valid values: from 0 to $2^{16}-1$.	
`Uint32`	Unsigned integer. Valid values: from 0 to $2^{32}-1$.	
`Uint64`	Unsigned integer. Valid values: from 0 to $2^{64}-1$.	
`Float`	Floating-point number with variable precision, 4 bytes in size.	Cannot be used in the primary key and in the columns that form the secondary index key
`Double`	Floating-point number with variable precision, 8 bytes in size.	Cannot be used in the primary key and in the columns that form the secondary index key
`Decimal(precision, scale)`	Floating-point number with fixed precision, 16 bytes in size. Precision is the maximum total number of stored decimal digits, takes values from 1 to 35. Scale is the maximum number of stored decimal digits to the right of the decimal point, takes values from 0 to the value of precision.	
`DyNumber`	Binary representation of a floating-point number with up to 38 digits of precision. Valid values: positive from 1×10^{-130} to $1\times10^{126}-1$, negative from $-1\times10^{126}-1$ to -1×10^{-130} and 0. Compatible with the `Number` type of AWS DynamoDB. Not recommended for use in ydb-native applications.	Not supported in columnar tables

Examples

```
SELECT
  Bool("true"),
  Uint8("0"),
  Int32("-1"),
  Uint32("2"),
  Int64("-3"),
  Uint64("4"),
  Float("-5"),
  Double("6"),
  Decimal("1.23", 5, 2), -- up to 5 decimal places, of which 2 are after the decimal point
  DyNumber("1"),
  String("foo"),
  Utf8("hello"),
  Yson("<a=1>[3;%false]"),
  Json(@@{"a":1,"b":null}@@),
  Date("2017-11-27"),
  Datetime("2017-11-27T13:24:00Z"),
  Timestamp("2017-11-27T13:24:00.123456Z"),
  Interval("P1W2DT2H3M4.567890S"),
  TzDate("2017-11-27,Europe/Moscow"),
  TzDatetime("2017-11-27T13:24:00,America/Los_Angeles"),
  TzTimestamp("2017-11-27T13:24:00.123456,GMT"),
  Uuid("f9d5cc3f-f1dc-4d9c-b97e-766e57ca4ccb");
```

String types

Type	Description	Notes
`String`	String, can contain arbitrary binary data	

`Bytes` (alias for `String`)	String, can contain arbitrary binary data	
`Utf8`	Text in UTF-8 encoding	
`Text` (alias for `Utf8`)	Text in UTF-8 encoding	
`Json`	JSON in text representation	Does not support comparison, cannot be used in the primary key and in the columns that form the secondary index key
`JsonDocument`	JSON in binary indexed representation	Does not support comparison, cannot be used in the primary key and in the columns that form the secondary index key
`Yson`	YSON in text or binary representation	Does not support comparison, cannot be used in the primary key and in the columns that form the secondary index key
`Uuid`	Universal identifier UUID	Not supported in columnar tables

The maximum size of a value in a cell of a non-key column with any string data type is 8 MB.

{% endnote %}

In contrast to the `Json` data type, which stores the original text representation provided by the user, `JsonDocument` uses a binary indexed representation. An important semantic difference is that `JsonDocument` does not preserve formatting, the order of keys in objects, or their duplicates.

Thanks to the indexed representation, `JsonDocument` allows you to traverse the document model using `JsonPath` without the need to parse the entire content. This enables efficient execution of operations from the JSON API, reducing latency and the cost of user queries. Query execution on `JsonDocument` can be several times more efficient, depending on the type of load.

Due to the added redundancy, `JsonDocument` is less efficient in terms of storage. The additional storage overhead depends on the specific content and averages 20–30% of the original volume. Storing data in `JsonDocument` format requires additional conversion from the text representation, which makes writing less efficient. However, for most read-intensive scenarios involving JSON data processing, this data type is preferable and is recommended for use.

The Double type is used to store numbers (JSON Number) in JsonDocument and for arithmetic operations on them in the JSON API. Loss of precision is possible when using non-standard number representations in the original JSON document.

Date and time

Type	Description	Possible values	Size (bytes)	Notes
`Date`	UTC midnight, accuracy to the day	from 00:00 January 1, 1970 to 00:00 January 1, 2106	2	—
`Date32`	UTC midnight, accuracy to the day	from 00:00 January 1, 144169 BC to 00:00 January 1, 148107 AD	4	—
`Datetime`	UTC time, accuracy to the second	from 00:00 January 1, 1970 to 00:00 January 1, 2106	4	—
`Datetime64`	UTC time, accuracy to the second	from 00:00 January 1, 144169 BC to 00:00 January 1, 148107 AD	8	—
`Timestamp`	UTC time, accuracy to the microsecond	from 00:00 January 1, 1970 to 00:00 January 1, 2106	8	—
`Timestamp64`	UTC time, accuracy to the microsecond	from 00:00 January 1, 144169 BC to 00:00 January 1, 148107 AD	8	—
`Interval`	Time interval, accuracy to the microsecond	from -136 years to +136 years	8	Not supported in columnar tables
`Interval64`	Time interval, accuracy to the microsecond	from -292277 years to +292277 years	8	Not supported in columnar tables
`TzDate`	UTC time corresponding to midnight in a given timezone	from 00:00 January 1, 1970 to 00:00 January 1, 2106		Not supported in table columns
`TzDate32`	UTC time corresponding to midnight in a given timezone	from 00:00 January 1, 144169 BC to 00:00 January 1, 148107 AD	4 and timezone mark	—
`TzDateTime`	UTC time with timezone mark and accuracy to the second	from 00:00 January 1, 1970 to 00:00 January 1, 2106		Not supported in table columns
`TzDateTime64`	UTC time with timezone mark and accuracy to the second	from 00:00 January 1, 144169 BC to 00:00 January 1, 148107 AD	8 and timezone mark	—
`TzTimestamp`	UTC time with timezone mark and accuracy to the microsecond	from 00:00 January 1, 1970 to 00:00 January 1, 2106		Not supported in table columns
`TzTimestamp64`	UTC time with timezone mark and accuracy to the microsecond	from 00:00 January 1, 144169 BC to 00:00 January 1, 148107 AD	8 and timezone mark	—

[1] Midnight is understood as the moment when all *time* components are equal to zero.

Interval type behavior and limitations

The `Interval` type uses syntax based on the ISO 8601 standard, with several specifics:

- only intervals that can be represented as an integer number of microseconds are supported (intervals in years `Y` and months `M` are not supported due to their variable duration);
- intervals with start/end points or repetitions are not supported;
- the use of a negative sign to indicate a shift into the past is supported;
- combining the week literal `W` with other literals is supported;
- microseconds can be specified as a fractional part of a second.

Interval data examples

```
SELECT
    Interval("P1W2DT2H3M4.567890S"), -- interval 1 week 2 days 2 hours 3 minutes 4.567890 seconds
    Interval("P1W"),                 -- interval 1 week (7 days)
    Interval("-P1D");                -- interval in the past 1 day (24 hours)
```

Time zone label support features

The time zone label for the `TzDate`, `TzDatetime`, `TzTimestamp` types is an attribute that is used:

- When converting (using CAST, DateTime::Parse, DateTime::Format) to and from a string.
- In DateTime::Split — the time zone component appears in `Resource<TM>`.

The actual time position value for these types is stored in UTC, and the time zone label does not participate in other calculations. For example:

```
SELECT -- these expressions are always true for any time zones: the time zone does not affect the point in ti
me.
    AddTimezone(CurrentUtcDate(), "Europe/Moscow") ==
        AddTimezone(CurrentUtcDate(), "America/New_York"),
    AddTimezone(CurrentUtcDatetime(), "Europe/Moscow") ==
        AddTimezone(CurrentUtcDatetime(), "America/New_York");
```

It is important to understand that when converting between `TzDate` and `TzDatetime` or `TzTimestamp`, the date corresponds not to midnight local time in the timezone, but to midnight UTC for the date in UTC.

Casting of primitive data types

Explicit casting

Explicit casting using CAST:

Casting to numeric types

Type	Bool	Int8	Int16	Int32	Int64	Uint8	Uint16	Uint32	Uint64	Float	Double	Decimal
Bool	—	Yes[1]	Yes[1]	Yes[1]	Yes[1]	Yes[1]	Yes[1]	Yes[1]	Yes[1]	Yes[1]	Yes[1]	No
Int8	Yes[2]	—	Yes	Yes	Yes	Yes[3]	Yes[3]	Yes[3]	Yes[3]	Yes	Yes	Yes
Int16	Yes[2]	Yes[4]	—	Yes	Yes	Yes[3,4]	Yes[3]	Yes[3]	Yes[3]	Yes	Yes	Yes
Int32	Yes[2]	Yes[4]	Yes[4]	—	Yes	Yes[3,4]	Yes[3,4]	Yes[3]	Yes[3]	Yes	Yes	Yes
Int64	Yes[2]	Yes[4]	Yes[4]	Yes[4]	—	Yes[3,4]	Yes[3,4]	Yes[3,4]	Yes[3]	Yes	Yes	Yes
Uint8	Yes[2]	Yes[4]	Yes	Yes	Yes	—	Yes	Yes	Yes	Yes	Yes	Yes
Uint16	Yes[2]	Yes[4]	Yes[4]	Yes	Yes	Yes[4]	—	Yes	Yes	Yes	Yes	Yes
Uint32	Yes[2]	Yes[4]	Yes[4]	Yes[4]	Yes	Yes[4]	Yes[4]	—	Yes	Yes	Yes	Yes
Uint64	Yes[2]	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes[4]	—	Yes	Yes	Yes
Float	Yes[2]	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes[3,4]	Yes[3,4]	Yes[3,4]	Yes[3,4]	—	Yes	No
Double	Yes[2]	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes[3,4]	Yes[3,4]	Yes[3,4]	Yes[3,4]	Yes	—	No
Decimal	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	—
String	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Bytes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Utf8	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Text	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Json	No	No	No	No	No	No	No	No	No	No	No	No
Yson	Yes[5]	Yes[5]	Yes[5]	Yes[5]	Yes[5]	Yes[5]	Yes[5]	Yes[5]	Yes[5]	Yes[5]	Yes[5]	No
Uuid	No	No	No	No	No	No	No	No	No	No	No	No
Date	No	Yes[4]	Yes[4]	Yes	Yes	Yes[4]	Yes	Yes	Yes	Yes	Yes	No
Datetime	No	Yes[4]	Yes[4]	Yes[4]	Yes	Yes[4]	Yes[4]	Yes	Yes	Yes	Yes	No

Timestamp	No	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes	Yes	Yes	No
Interval	No	Yes[4]	Yes[4]	Yes[4]	Yes	Yes[3,4]	Yes[3,4]	Yes[3,4]	Yes[3]	Yes	Yes	No
Date32	No	Yes[4]	Yes[4]	Yes	Yes	Yes[4]	Yes	Yes	Yes	Yes	Yes	No
Datetime64	No	Yes[4]	Yes[4]	Yes[4]	Yes	Yes[4]	Yes[4]	Yes	Yes	Yes	Yes	No
Timestamp64	No	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes	Yes	Yes	No
Interval64	No	Yes[4]	Yes[4]	Yes[4]	Yes	Yes[3,4]	Yes[3,4]	Yes[3,4]	Yes[3]	Yes	Yes	No
[1] `True` is converted to `1`, `False` is converted to `0`.												
[2] Any value other than `0` is converted to `True`, `0` is converted to `False`.												
[3] Only possible if the value is non-negative.												
[4] Only possible if the value falls within the range of allowed values.												
[5] Using the built-in function Yson::ConvertTo.												

Casting to date and time data types

Type	Date	Datetime	Timestamp	Interval	Date32	Datetime64	Timestamp64	Interval64
Bool	No	No	No	No	No	No	No	No
Int8	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Int16	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Int32	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Int64	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Uint8	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Uint16	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Uint32	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Uint64	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Float	No	No	No	No	No	No	No	No
Double	No	No	No	No	No	No	No	No
Decimal	No	No	No	No	No	No	No	No
String	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Bytes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Utf8	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Text	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Json	No	No	No	No	No	No	No	No
Yson	No	No	No	No	No	No	No	No
Uuid	No	No	No	No	No	No	No	No
Date	—	Yes	Yes	No	Yes	Yes	Yes	No
Datetime	Yes	—	Yes	No	Yes	Yes	Yes	No
Timestamp	Yes	Yes	—	No	Yes	Yes	Yes	No
Interval	No	No	No	—	No	No	No	Yes
Date32	Yes	Yes	Yes	No	—	Yes	Yes	No
Datetime64	Yes	Yes	Yes	No	Yes	—	Yes	No

Timestamp64	Yes	Yes	Yes	No	Yes	Yes	—	No
Interval64	No	No	No	Yes	No	No	No	—

Casting to other data types

Type	String	Bytes	Utf8	Text	Json	Yson	Uuid
Bool	Yes	Yes	No	No	No	No	No
Int8	Yes	Yes	No	No	No	No	No
Int16	Yes	Yes	No	No	No	No	No
Int32	Yes	Yes	No	No	No	No	No
Int64	Yes	Yes	No	No	No	No	No
Uint8	Yes	Yes	No	No	No	No	No
Uint16	Yes	Yes	No	No	No	No	No
Uint32	Yes	Yes	No	No	No	No	No
Uint64	Yes	Yes	No	No	No	No	No
Float	Yes	Yes	No	No	No	No	No
Double	Yes	Yes	No	No	No	No	No
Decimal	Yes	Yes	No	No	No	No	No
String	—	—	Yes	Yes	Yes	Yes	Yes
Bytes	—	—	Yes	Yes	Yes	Yes	Yes
Utf8	Yes	Yes	—	—	No	No	No
Text	Yes	Yes	—	—	No	No	No
Json	Yes	Yes	Yes	Yes	—	No	No
Yson	Yes[1]	Yes[1]	No	No	No	No	No
Uuid	Yes	Yes	Yes	Yes	No	No	—
Date	Yes	Yes	Yes	Yes	No	No	No
Datetime	Yes	Yes	Yes	Yes	No	No	No
Timestamp	Yes	Yes	Yes	Yes	No	No	No
Interval	Yes	Yes	Yes	Yes	No	No	No
Date32	Yes	Yes	Yes	Yes	No	No	No
Datetime64	Yes	Yes	Yes	Yes	No	No	No
Timestamp64	Yes	Yes	Yes	Yes	No	No	No
Interval64	Yes	Yes	Yes	Yes	No	No	No

[1] Using the built-in function Yson::ConvertTo.

Examples

```
SELECT
    CAST("12345" AS Double),               -- 12345.0
    CAST(1.2345 AS Uint8),                 -- 1
    CAST(12345 AS String),                 -- "12345"
    CAST("1.2345" AS Decimal(5, 2)),       -- 1.23
    CAST("xyz" AS Uint64) IS NULL,         -- true, since it failed
    CAST(-1 AS Uint16) IS NULL,            -- true, negative to unsigned
    CAST([-1, 0, 1] AS List<Uint8?>),             -- [null, 0, 1]
-- The element type is optional: the failed element is null.
    CAST(["3.14", "bad", "42"] AS List<Float>),   -- [3.14, 42]
-- The element type is not optional: the failed element is removed.
    CAST(255 AS Uint8),                    -- 255
    CAST(256 AS Uint8) IS NULL;            -- true, out of range
```

Implicit casting

Implicit type casting that occurs in basic operations (`+`, `-`, `*`, `/`, `%`) between different data types. The table cells indicate the result type of the operation if it is possible:

Numeric types

If the numeric types do not match, a BITCAST of both arguments to the result type is performed first, and then the operation.

Type	Int8	Int16	Int32	Int64	Uint8	Uint16	Uint32	Uint64	Float	Double

Int8	—	`Int16`	`Int32`	`Int64`	`Int8`	`Uint16`	`Uint32`	`Uint64`	`Float`	`Double`
Int16	`Int16`	—	`Int32`	`Int64`	`Int16`	`Int16`	`Uint32`	`Uint64`	`Float`	`Double`
Int32	`Int32`	`Int32`	—	`Int64`	`Int32`	`Int32`	`Int32`	`Uint64`	`Float`	`Double`
Int64	`Int64`	`Int64`	`Int64`	—	`Int64`	`Int64`	`Int64`	`Int64`	`Float`	`Double`
Uint8	`Int8`	`Int16`	`Int32`	`Int64`	—	`Uint16`	`Uint32`	`Uint64`	`Float`	`Double`
Uint16	`Uint16`	`Int16`	`Int32`	`Int64`	`Uint16`	—	`Uint32`	`Uint64`	`Float`	`Double`
Uint32	`Uint32`	`Uint32`	`Int32`	`Int64`	`Uint32`	`Uint32`	—	`Uint64`	`Float`	`Double`
Uint64	`Uint64`	`Uint64`	`Uint64`	`Int64`	`Uint64`	`Uint64`	`Uint64`	—	`Float`	`Double`
Float	`Float`	`Float`	`Float`	`Float`	`Float`	`Float`	`Float`	`Float`	—	`Double`
Double	`Double`	`Double`	`Double`	`Double`	`Double`	`Double`	`Double`	`Double`	`Double`	—

Date and time types

Type	Date	Datetime	Timestamp	Interval	TzDate	TzDatetime	TzTimestamp	Date32	Datetime64	Timestamp64
Date	—	`DateTime`	`Timestamp`	—	`TzDate`	`TzDatetime`	`TzTimestamp`	`Date32`	`DateTime64`	`Timestamp64`
Datetime	—	—	`Timestamp`	—	—	`TzDatetime`	`TzTimestamp`	—	`Datetime64`	`Timestamp64`
Timestamp	—	—	—	—	—	—	`TzTimestamp`	—	—	`Timestamp64`
Interval	—	—	—	—	—	—	—	—	—	—
TzDate	—	—	—	—	—	`TzDatetime`	`TzTimestamp`	—	—	—
TzDatetime	—	—	—	—	—	—	`TzTimestamp`	—	—	—
TzTimestamp	—	—	—	—	—	—	—	—	—	—
Date32	—	—	—	—	—	—	—	—	`DateTime64`	`Timestamp64`
Datetime64	—	—	—	—	—	—	—	—	—	`Timestamp64`
Timestamp64	—	—	—	—	—	—	—	—	—	—
Interval64	—	—	—	—	—	—	—	—	—	—
TzDate32	—	—	—	—	—	—	—	—	—	—
TzDatetime64	—	—	—	—	—	—	—	—	—	—
TzTimestamp64	—	—	—	—	—	—	—	—	—	—

Serial data types

Serial data types are integers with an additional value-generation mechanism. They are used for auto-increment columns: each new row inserted into the table automatically gets a unique value in such a column (similar to the SERIAL type in PostgreSQL or AUTO_INCREMENT in MySQL).

> **Note**
>
> Using serial types as a primary key is not recommended: monotonically increasing values lead to uneven data distribution and hot partitions. For details, see Selecting a primary key for maximum performance.

Usage example

```
CREATE TABLE users (
    user_id Serial,
    name Utf8,
    email Utf8,
    PRIMARY KEY (user_id)
);
```

```
UPSERT INTO users (name, email) VALUES ('Alice', 'alice@example.com');
INSERT INTO users (name, email) VALUES ('Bob', 'bob@example.com');
REPLACE INTO users (name, email) VALUES ('John', 'john@example.com');
```

```
SELECT * FROM users;
```

email	name	user_id
`alice@example.com`	Alice	1
`bob@example.com`	Bob	2
`john@example.com`	John	3

You can supply a value for a `Serial` column explicitly on insert; the row is then handled like a plain integer column, and the `Sequence` is not affected:

```
UPSERT INTO users (user_id, name, email) VALUES (4, 'Peter', 'peter@example.com');
```

Description

Only columns that participate in a table's primary key may use the `Serial` type.

Defining this type on a column creates a separate schema object, `Sequence`, bound to that column and used as a generator of sequence values. This object is private and hidden from the user. The `Sequence` is destroyed together with the table.

Sequence values start at 1, increment by 1, and are bounded according to the type used.

Type	Maximum value	Value type
`SmallSerial`		`Int16`
`Serial2`		`Int16`
`Serial`		`Int32`
`Serial4`		`Int32`
`Serial8`		`Int64`
`BigSerial`		`Int64`

If the sequence overflows on insert, an error is returned:

```
Error: Failed to get next val for sequence: /dev/test/users/_serial_column_user_id, status: SCHEME_ERROR
    <main>: Error: sequence [OwnerId: <some>, LocalPathId: <some>] doesn't have any more values available, co
de: 200503
```

The next value is allocated by the generator before the row is actually inserted and is considered consumed even if the row is not successfully written (for example, when the transaction rolls back). Therefore the set of values in such a column may have gaps and consist of several disjoint ranges.

For tables with auto-increment columns, the copy, dump, restore, and import/export operations are supported.

Data types accepting NULL

Any typed data in YQL, including table columns, can be either non-nullable (guaranteed value) or nullable (empty value denoted as `NULL`). Data types that can include `NULL` values are called *optional* or, in SQL terms, *nullable*.

Optional data types in the text format use the question mark at the end (for example, `String?`) or the notation `Optional<...>`. The following operations are most often performed on optional data types:

- IS NULL: Matching an empty value
- COALESCE: Leave the filled values unchanged and replace `NULL` with the default value that follows
- UNWRAP: Extract the value of the original type from the optional data type, `T?`. is converted to `T`
- JUST: Add optionality to the current type, `T` is converted to `T?`.
- NOTHING: Create an empty value with the specified type.

`Optional` (nullable) isn't a property of a data type or column, but a container type where containers can be arbitrarily nested into each other. For example, a column with the type `Optional<Optional<Boolean>>` can accept 4 values: `NULL` of the whole container, `NULL` of the inner container, `TRUE`, and `FALSE`. The above-declared type differs from `List<List<Boolean>>`, because it uses `NULL` as an empty list, and you can't put more than one non-null element in it. In addition, `Optional<Optional<T>>` type values are returned as results when searching by the key in the `Dict(k,v)` dictionary with `Optional<T>` type values. Using this type of result data, you can distinguish between a `NULL` value in the dictionary and a situation when the key is missing.

Note

Container types (including `Optional<T>` containers and more complex types derived from them) can't currently be used as column data types when creating YDB tables.
YQL queries can return values of container types and accept them as input parameters.

Example

```
$dict = {"a":1, "b":null};
$found = $dict["b"];
select if($found is not null, unwrap($found), -1);
```

Result:

```
# column0
null
```

Logical and arithmetic operations with NULL

The `NULL` literal has a separate singular `Null` type and can be implicitly converted to any optional type (for example, the nested type `Optional<Optional<...Optional<T>...>>`). In ANSI SQL, `NULL` means "an unknown value", that's why logical and arithmetic operations involving `NULL` or empty `Optional` have certain specifics.

Examples

```
SELECT
    True OR NULL,        -- Just(True) (works the same way as True OR <unknown value of type Bool>)
    False AND NULL,      -- Just(False)
    True AND NULL,       -- NULL   (more precise than Nothing<Bool?> – <unknown value of type Bool>)
    NULL OR NOT NULL,    -- NULL   (all NULLs are "different")
    1 + NULL,            -- NULL   (Nothing<Int32?>) - the result of adding 1 together with
                         --         unknown value of type Int)
    1 == NULL,           -- NULL   (the result of adding 1 together with unknown value of type Int)
    (1, NULL) == (1, 2), -- NULL   (composite elements are compared by component
                         --         through `AND`)
    (2, NULL) == (1, 3), -- Just(False) (expression is equivalent to 2 == 1 AND NULL == 3)
```

Data types that do not allow NULL values

Primitive types in YQL cannot hold a `NULL` value: the container described above, `Optional`, is intended for storing `NULL`. In SQL terms, primitive types in YQL are *non-nullable* types.

In YQL, there is no implicit type conversion from Optional to T, so the enforceability of the NOT NULL constraint on a table column is ensured at the query compilation stage in YDB.

You can create a non-nullable column in a YDB table using the CREATE TABLE operation with the keyword `NOT NULL`:

```
CREATE TABLE t (
    Key Uint64 NOT NULL,
    Value String NOT NULL,
    PRIMARY KEY (Key))
```

After that, write operations to table `t` will only be executed if the values to be inserted into the `key` and `value` columns do not contain `NULL` values.

Example of the interaction between the NOT NULL constraint and YQL functions

Many of the YQL functions have optional types as return values. Since YQL is a strongly-typed language, a query like

```
CREATE TABLE t (
    c Utf8 NOT NULL,
```

```
    PRIMARY KEY (c)
);
INSERT INTO t(c)
SELECT CAST('q' AS Utf8);
```

cannot be executed. The reason for this is the type mismatch between the column `c`, which has the type `Utf8`, and the result of the `CAST` function, which has the type `Optional<Utf8>`. To make the query work correctly in such scenarios, it is necessary to use the COALESCE function, whose argument can specify a fallback value to insert into the table in case the function (in the example, CAST) returns an empty `Optional`. If, in the case of an empty `Optional`, the insertion should not be performed and an error should be returned, the UNWRAP function can be used to unpack the contents of the optional type.

Containers

YQL supports container types to define complex data structures organized in various ways.
Values of container types can be passed to YQL queries as input parameters or returned from YQL queries as columns of the set of results.
Container types can't be used as column data types for YDB tables.

Type	Declaration, example	Description
List	`List<Type>`, `List<Int32>`	A variable-length sequence consisting of same-type elements.
Dictionary	`Dict<KeyType, ValueType>`, `Dict<String,Int32>`	Set of key-value pairs with a fixed type of keys and values.
Set	`Set<KeyType>`, `Set<String>`	A set of elements with a fixed type is a special case of a dictionary with the `Void` value type.
Tuple	`Tuple<Type1, ..., TypeN>`, `Tuple<Int32,Double>`	Set of unnamed fixed-length elements with types specified for all elements.
Structure	`Struct<Name1:Type1, ..., NameN:TypeN>`, `Struct<Name:String,Age:Int32>`	A set of named fields with specified value types, fixed at query start (must be data-independent).
Stream	`Stream<Type>`, `Stream<Int32>`	Single-pass iterator by same-type values, not serializable
Variant on tuple	`Variant<Type1, Type2>`, `Variant<Int32,String>`	A tuple known to have exactly one element filled
Variant on structure	`Variant<Name1:Type1, Name2:Type2>`, `Variant<value:Int32,error:String>`	A structure known to have exactly one element filled
Enumeration	`Enum<Name1, Name2>`, `Enum<value,error>`	A container with exactly one enumeration element selected and defined only by its name.

If necessary, you can nest containers in any combination, for example, `List<Tuple<Int32,Int32>>`.

In certain contexts, optional values can also be considered a container type (`Optional<Type>`) that behaves like a list of length 0 or 1.

To create literals of list containers, dictionary containers, set containers, tuple containers, or structure containers, you can use the operator notation.
To create a variant literal over a tuple or structure, use the function Variant.
To create an enumeration literal, use the function Enum.

To access the container elements, use a dot or square brackets, depending on the container type.

Special data types

Type	Description
`Callable`	A callable value that can be executed by passing arguments in parentheses in YQL SQL syntax.
`Resource`	Resource is an opaque pointer to a resource you can pass between the user defined functions (UDF). The type of the returned and accepted resource is declared inside a function using a string label. When passing a resource, YQL checks for label matching to prevent passing of resources between incompatible functions. If the labels mismatch, a type error occurs.
`Tagged`	Tagged is the option to assign an application name to any other type.
`Generic`	The data type used for data types.
`Unit`	Unit is the data type used for non-enumerable entities (data sources and data sinks, atoms, etc.).
`Null`	Void is a singular data type with the only possible null value. It's the type of the `NULL` literal and can be converted to any `Optional` type.
`Void`	Void is a singular data type with the only possible `"null"` value.
`EmptyList`	A singular data type with the only possible [] value. It's the type of the `[]` literal and can be converted to any `List` type.
`EmptyDict`	A singular data type with the only possible {} value. It's a type of the `{}` literal and can be converted to any `Dict` or `Set` type.

Rules for type casting using the operator CAST

Rules for casting primitive data types

- When casting primitive data types, some of the source information may be discarded unless contained in the target type. For example:
  - The `Float` / `Double` fractional part, when casting to integer types.
  - The `Datetime` / `Timestamp` time, when casting to `Date`.
  - The timezone, when casting from timezone types to date/time types without a timezone.
- If, in a certain combination of the source and target type, casting can't be performed for all possible values of the source type, then, if the casting fails, `CAST` returns `NULL`. In such cases, one `Optional` level is added to the return value type, unless already present. For example, the constructs: `CAST("3.14" AS Float?)` and `CAST("3.14" AS Float)` are fully equivalent and return `Float?`.
- If casting is possible for all values of the source type, then adding '?' works the same way as `Just` on top: `CAST(3.14 AS Utf8?)` is same as `Just(CAST(3.14 AS Utf8))`.

All combinations of primitive data types for which `CAST` can be used are described here.

Casting rules for containers

Rules for Optional

- If a higher `Optional` level is set for the target type than for the source type, it's same as adding `Just` on top of `CAST` with a lower `Optional` level.
- If the source type has a higher level of `Optional` for the source type, then `NULL` at any level higher than the target level results in `NULL`.
- At equal levels of `Optional`, the `NULL` value preserves the same level.

```
SELECT
    CAST(1 AS Int32?),                  -- is equivalent to Just(1)
    CAST(Just(2/1) AS Float??),         -- [2]
    CAST(Just(3/0) AS Float??) IS NULL; -- false: the result is Just(NULL)
```

Rules for List/Dict

- To create a list, `CAST` is applied to each item in the source list to cast it to the target type.
- If the target item type is non-optional and `CAST` on the item might fail, then such casting is discarded. In this case, the resulting list might be shorter or even empty if every casting failed.
- For dictionaries, the casting is totally similar to lists, with `CAST` being applied to keys and values.

```
SELECT
    CAST([-1, 0, 1] AS List<Uint8?>),             -- [null, 0, 1]
    CAST(["3.14", "bad", "42"] AS List<Float>),   -- [3.14, 42]

    CAST({-1:3.14, 7:1.6} AS Dict<Uint8, Utf8>),  -- {7: "1.6"}
    CAST({-1:3.14, 7:1.6} AS Dict<Uint8?, Utf8>); -- {7: "1.6", null:"3.14"}
```

Rules for Struct/Tuple

- A structure or tuple is created by applying `CAST` to each item of the source type to cast it to an item with the same name or target type index.
- If some field is missing in the target type, it's simply discarded.
- If some field is missing in the source value type, then it can be added only if it's optional and accepts the `NULL` value.
- If some field is non-optional in the target type, but its casting might fail, then `CAST` adds Optional to the structure or tuple level and might return `NULL` for the entire result.

```
SELECT
    CAST((-1, 0, 1) AS Tuple<Uint16?, Uint16?, Utf8>), -- (null, 0, "1")
    CAST((-2, 0) AS Tuple<Uint16, Utf8>),              -- null
    CAST((3, 4) AS Tuple<Uint16, String>),             -- (3, "4"): the type is Tuple<Uint16, String>?
    CAST(("4",) AS Tuple<Uint16, String?>),            -- (4, null)
    CAST((5, 6, null) AS Tuple<Uint8?>);               -- (5,): the items were removed.

SELECT -- One field was removed and one field was added: ("three":null, "two": "42")
    CAST(<|one:"8912", two:42|> AS Struct<two:Utf8, three:Date?>);
```

Rules for Variant

- A variant with a specific name or index is cast to a variant with the same name or index.
- If casting of a variant might fail and the type of this variant is non-optional, then `CAST` adds Optional to the top level and can return `NULL`.
- If some variant is missing in the target type, then `CAST` adds Optional to the top level and returns `NULL` for such a value.

Nested containers

- All of the above rules are applied recursively for nested containers.

Text representation of data types

Introduction

Since YQL is a strongly typed language, the data type is important for many of its aspects. To make data type management easy, YQL has a data type definition convention in text format. It's mentioned in many places in the documentation. There's also a library that provides functions for building a data type based on a text description (for example, when manually defining the signature for the called value) or for serializing the data type into a string for debugging purposes.

Functions for data types are described in the article. Below is the format of text representation of data types.

General conventions

- Primitive data types are represented in text format simply by referencing their name.
- A complex data type is composed of other data types. If you depict this structure as a tree, it has primitive data types as leaves and containers as other nodes. You may treat special data types as exceptions, because they can function as both.
- The text representation repeats the structure of this tree from the root to the leaves: each node of the tree specifies the name of the current data type, and proceeding to a deeper level is denoted by different types of brackets.
- Feel free to use spaces and line breaks if they improve readability.
- If the ID contains something else except the Latin letters and numbers, put it in single quotes and use C-escaping.

Containers

- Use angle brackets to specify the types of container elements.

  Example: `List<Int32>`.
- If a container can hold multiple heterogeneous elements, they are listed inside angle brackets with a comma.

  Example: `Tuple<Int32, String>`.
- If a container can hold named elements, use comma-separated name-type pairs with a colon in-between instead of comma-separated data types.

  Example: `Struct<a:Int32, b:String>`.
- The underlying `Variant` container type is chosen based on the presence of names in arguments.

  Example: `Variant<Int32, String>` is a variant on tuple, `Variant<a:Int32, b:String>` is a variant on structure.

Types that allow NULL

- They are called `Optional` in YQL terms, or nullable in the classic SQL terms.
- Formally, this type is a container. So, you may declare it as `Optional<...>`, but the shortcut notation of a question mark suffix is usually used instead.

  Example: `String?`.

Called values

- The basic form of the called values looks as follows: `(arg1, arg2, ...) -> result`.

  An example of declaring a function signature that accepts two strings and returns a number: `(String, String) -> Int64`.
- The called values can return the called values: in this case, they make up a chain of the required length.

  Example: `(String, String) -> (String, String) -> Int64`.
- Optional arguments must have the `Optional` type at the top level and be enclosed in square brackets.

  Example: `(String, [String?, Double?]) -> Int64`.
- The arguments of the called values can contain flags.

  Currently, the only possible flag is `AutoMap`. It means that if NULL is passed to this argument, the result must also be set to NULL without running the function.

  Example: `(String{Flags: AutoMap}) -> Int64`.
- Use this particular format if you need `Optional<Callable...>`, because the trailing question mark refers to the result of the called value.

Resources

- Unlike containers, a resource isn't parameterized by the element type (it's a pointer in memory and YQL knows nothing about its contents). Instead, a resource is parameterized by a string label that can safeguard against passing resources between incompatible functions.

  Example: `Resource<Foo>`.

Data representation in JSON format

Bool

Boolean value.

- Type in JSON: `bool`.
- Sample YDB value: `true`.
- Sample JSON value: `true`.

Int8, Int16, Int32, Int64

Signed integer types.

- Type in JSON: `number`.
- Sample YDB value: `123456`, `-123456`.
- Sample JSON value: `123456`, `-123456`.

Uint8, Uint16, Uint32, Uint64

Unsigned integer types.

- Type in JSON: `number`.
- Sample YDB value: `123456`.
- Sample JSON value: `123456`.

Float

Real 4-byte number.

- Type in JSON: `number`.
- Sample YDB value: `0.12345679`.
- Sample JSON value: `0.12345679`.

Double

Real 8-byte number.

- Type in JSON: `number`.
- Sample YDB value: `0.12345678901234568`.
- Sample JSON value: `0.12345678901234568`.

Decimal

Fixed-precision number.

- Type in JSON: `string`.
- Sample YDB value: `-320.789`.
- Sample JSON value: `"-320.789"`.

String, Yson

Binary strings. Encoding algorithm depending on the byte value:

- [0-31] — `\u00XX` (6 characters denoting the Unicode character code).
- [32-126] — as is. These are readable single-byte characters that don't need to be escaped.
- [127-255] — `\u00XX`.

Decoding is a reverse process. Character codes in `\u00XX`, maximum 255.

- Type in JSON: `string`.
- Sample YDB value: A sequence of 4 bytes:
  - 5 `0x05`: A control character.
  - 10 `0x0a`: The `\n` newline character.
  - 107 `0x6b`: The `k` character.
  - 255 `0xff`: The `ÿ` character in Unicode.
- Sample JSON value: `"\u0005\nk\u00FF"`.

Utf8, Json, Uuid

String types in UTF-8. Such strings are represented in JSON as strings with JSON characters escaped: `\\`, `\"`, `\n`, `\r`, `\t`, `\f`.

- Type in JSON: `string`.
- Sample YDB value: C++ code:

```
"Escaped characters: "
"\\ \" \f \b \t \r\n"
"Non-escaped characters: "
"/ ' < > & []() ".
```

- Sample JSON value: `"Escaped characters: \\ \" \f \b \t \r\nNon-escaped characters: / ' < > & []() "`.

Date

Date. Uint64, unix time days.

- Type in JSON: `string`.
- Sample YDB value: `18367`.
- Sample JSON value: `"2020-04-15"`.

Datetime

Date and time. Uint64, unix time seconds.

- Type in JSON: `string`.
- Sample YDB value: `1586966302`.
- Sample JSON value: `"2020-04-15T15:58:22Z"`.

Timestamp

Date and time. Uint64, unix time microseconds.

- Type in JSON: `string`.
- Sample YDB value: `1586966302504185`.
- Sample JSON value: `"2020-04-15T15:58:22.504185Z"`.

Interval

Time interval. Int64, precision to the microsecond, the interval values must not exceed 24 hours.

- Type in JSON: `number`.
- Sample YDB value: `123456`, `-123456`.
- Sample JSON value: `123456`, `-123456`.

Optional

Means that the value can be `null`. If the value is `null`, then in JSON it's also `null`. If the value is not `null`, then the JSON value is expressed as if the type isn't `Optional`.

- Type in JSON is missing.
- Sample YDB value: `null`.
- Sample JSON value: `null`.

List

List. An ordered set of values of a given type.

- Type in JSON: `array`.
- Sample YDB value:
  - Type: `List<Int32>`.
  - Value: `1, 10, 100`.
- Sample JSON value: `[1,10,100]`.

Stream

Stream. Single-pass iterator by same-type values,

- Type in JSON: `array`.
- Sample YDB value:
  - Type: `Stream<Int32>`.
  - Value: `1, 10, 100`.
- Sample JSON value: `[1,10,100]`.

Struct

Structure. An unordered set of values with the specified names and type.

- Type in JSON: `object`.
- Sample YDB value:
  - Type: `Struct<'Id':Uint32,'Name':String,'Value':Int32,'Description':Utf8?>`;
  - Value: `"Id":1,"Name":"Anna","Value":-100,"Description":null`.
- Sample JSON value: `{"Id":1,"Name":"Anna","Value":-100,"Description":null}`.

Tuple

Tuple. An ordered set of values of the set types.

- Type in JSON: `array`.
- Sample YDB value:
  - Type: `Tuple<Int32??,Int64???,String??,Utf8???>`;
  - Value: `10,-1,null,"Some string"`.
- Sample JSON value: `[10,-1,null,"Some string"]`.

Dict

Dictionary. An unordered set of key-value pairs. The type is set both for the key and the value. It's written in JSON to an array of arrays including two items.

- Type in JSON: `array`.
- Sample YDB value:
  - Type: `Dict<Int64,String>`.
  - Value: `1:"Value1",2:"Value2"`.

- Sample JSON value: `[[1,"Value1"],[2,"Value2"]]`.

List of articles on YQL syntax

- Lexical structure
- Expressions
- SELECT
- VALUES
- CREATE TABLE
- DROP TABLE
- INSERT
- SHOW CREATE
- ANALYZE
- ALTER TABLE
- UPDATE
- DELETE
- REPLACE
- UPSERT
- BATCH UPDATE
- BATCH DELETE
- GROUP BY
- JOIN
- WINDOW
- FLATTEN
- ACTION
- DISCARD
- INTO RESULT
- PRAGMA
- DECLARE
- CREATE TOPIC
- ALTER TOPIC
- DROP TOPIC
- CREATE ASYNC REPLICATION
- ALTER ASYNC REPLICATION
- DROP ASYNC REPLICATION
- CREATE TRANSFER
- ALTER TRANSFER
- DROP TRANSFER
- CREATE BACKUP COLLECTION
- BACKUP
- RESTORE
- DROP BACKUP COLLECTION
- COMMIT
- CREATE VIEW
- ALTER VIEW
- DROP VIEW
- CREATE OBJECT (TYPE SECRET)
- CREATE USER
- ALTER USER
- DROP USER
- CREATE GROUP
- ALTER GROUP
- DROP GROUP
- GRANT
- REVOKE
- Unsupported statements

Lexical structure

The query in the YQL language is a valid UTF-8 text consisting of **statements** separated by semicolons (`;`).
The last semicolon can be omitted.
Each command is a sequence of **tokens** that are valid for this command.
Tokens can be **keywords**, **identifiers**, **literals**, and so on.
Tokens are separated by whitespace characters (space, tab, line feed) or **comments**. The comment is not a part of the command and is syntactically equivalent to a space character.

Syntax compatibility modes

Two syntax compatibility modes are supported:

- Advanced C++ (default)
- ANSI SQL

ANSI SQL mode is enabled with a special comment `--!ansi_lexer`, which must be in the beginning of the query.

Specifics of interpretation of lexical elements in different compatibility modes are described below.

Comments

The following types of comments are supported:

- Single-line comment: starts with `--` (two minus characters following one another) and continues to the end of the line
- Multiline comment: starts with `/*` and ends with `*/`

```
SELECT 1; -- A single-line comment
/*
   Some multi-line comment
*/
```

In C++ syntax compatibility mode (default), a multiline comment ends with the **nearest** `*/`.
The ANSI SQL syntax compatibility mode accounts for nesting of multiline comments:

```
--!ansi_lexer
SELECT * FROM T; /* this is a comment /* this is a nested comment, without ansi_lexer it raises an error  */
*/
```

Keywords and identifiers

Keywords are tokens that have a fixed value in the YQL language. Examples of keywords: `SELECT`, `INSERT`, `FROM`, `ACTION`, and so on. Keywords are case-insensitive, that is, `SELECT` and `SeLEcT` are equivalent to each other.
The list of keywords is not fixed and is going to expand as the language develops. A keyword can't contain numbers and begin or end with an underscore.

Identifiers are tokens that identify the names of tables, columns, and other objects in YQL. Identifiers in YQL are always case-sensitive.
An identifier can be written in the body of the program without any special formatting, if the identifier:

- Is not a keyword
- Begins with a Latin letter or underscore
- Is followed by a Latin letter, an underscore, or a number

```
SELECT my_column FROM my_table; -- my_column and my_table are identifiers
```

To include an arbitrary ID in the body of a query, the ID is enclosed in backticks:

```
SELECT `column with space` from T;
SELECT * FROM `my_dir/my_table`
```

IDs in backticks are never interpreted as keywords:

```
SELECT `select` FROM T; -- select - Column name in the T table
```

When using backticks, you can use the standard C escaping:

```
SELECT 1 as `column with\n newline, \x0a newline and \` backtick `;
```

In ANSI SQL syntax compatibility mode, arbitrary IDs can also be enclosed in double quotes. To include a double quote in a quoted ID, use two double quotes:

```
--!ansi_lexer
SELECT 1 as "column with "" double quote"; -- column name will be: column with " double quote
```

SQL hints

SQL hints are special settings with which a user can modify a query execution plan
(for example, enable/disable specific optimizations or force the JOIN execution strategy).
Unlike PRAGMA, SQL hints act locally – they are linked to a specific point in the YQL query (normally, after the keyword)
and affect only the corresponding statement or even a part of it.
SQL hints are a set of settings "name-value list" and defined inside special comments —
comments with SQL hints must have `+` as the first character:

```
--+ Name1(Value1 Value2 Value3) Name2(Value4) ...
```

An SQL hint name must be comprised of ASCII alphanumeric characters and start with a letter. Hint names are case insensitive.
A hint name must be followed by a custom number of space-separated values. A value can be a custom set of characters.
If there's a space or parenthesis in a set of characters, single quotation marks must be used:

```
--+ foo('value with space and paren)')
```

```
--+ foo('value1' value2)
-- equivalent to
--+ foo(value1 value2)
```

To escape a single quotation within a value, double it:

```
--+ foo('value with single quote '' inside')
```

If there're two or more hints with the same name in the list, the latter is used:

```
--+ foo(v1 v2) bar(v3) foo()
-- equivalent to
--+ bar(v3) foo()
```

Unknown SQL hint names (or syntactically incorrect hints) never result in errors, they're simply ignored:

```
--+ foo(value1) bar(value2  baz(value3)
-- due to a missing closing parenthesis in bar, is equivalent to
--+ foo(value1)
```

Thanks to this behavior, previous valid YQL queries with comments that look like hints remain intact.
Syntactically correct SQL hints in a place unexpected for YQL result in a warning:

```
-- presently, hints after SELECT are not supported
SELECT /*+ foo(123) */ 1; -- warning 'Hint foo will not be used'
```

What's important is that SQL hints are hints for an optimizer, so:

- Hints never affect search results.
- As YQL optimizers improve, a situation is possible when a hint becomes outdated and is ignored (for example, the algorithm based on a given hint completely changes or the optimizer becomes so sophisticated that it can be expected to choose the best solution, so some manual settings are likely to interfere).

String literals

A string literal (constant) is expressed as a sequence of characters enclosed in single quotes. Inside a string literal, you can use the C-style escaping rules:

```
SELECT 'string with\n newline, \x0a newline and \' backtick ';
```

In the C++ syntax compatibility mode (default), you can use double quotes instead of single quotes:

```
SELECT "string with\n newline, \x0a newline and \" backtick ";
```

In ASNI SQL compatibility mode, double quotes are used for IDs, and the only escaping that can be used for string literals is a pair of single quotes:

```
--!ansi_lexer
SELECT 'string with '' quote'; -- result: string with ' quote
```

Based on string literals, simple literals can be obtained.

Multi-line string literals

A multiline string literal is expressed as an arbitrary set of characters enclosed in double at signs `@@`:

```
$text = @@some
multiline
text@@;
SELECT LENGTH($text);
```

If you need to use double at signs in your text, duplicate them:

```
$text = @@some
multiline with double at: @@@@
text@@;
SELECT $text;
```

Typed string literals

- For string literals, including multi-string ones, the `String` type is used by default (see also PRAGMA UnicodeLiterals).
- You can use the following suffixes to explicitly control the literal type:
  - `s` — `String`
  - `u` — `Utf8`
  - `y` — `Yson`
  - `j` — `Json`

Example

```
SELECT "foo"u, '[1;2]'y, @@{"a":null}@@j;
```

Numeric literals

- Integer literals have the default type `Int32`, if they fit within the Int32 range. Otherwise, they automatically expand to `Int64`.
- You can use the following suffixes to explicitly control the literal type:
  - `l`: `Int64`
  - `s`: `Int16`
  - `t`: `Int8`
- Add the suffix `u` to convert a type to its corresponding unsigned type:
  - `ul`: `Uint64`
  - `u`: `Uint32`
  - `us`: `Uint16`
  - `ut`: `Uint8`
- You can also use hexadecimal, octal, and binary format for integer literals using the prefixes `0x`, `0o` and `0b`, respectively. You can arbitrarily combine them with the above-mentioned suffixes.
- Floating point literals have the `Double` type by default, but you can use the suffix `f` to narrow it down to `Float`.

```
SELECT
  123l AS `Int64`,
  0b01u AS `Uint32`,
  0xfful AS `Uint64`,
  0o7ut AS `Uint8`,
  456s AS `Int16`,
  1.2345f AS `Float`;
```

Expressions

String concatenation

Executed using the binary operator `||`.

As with other binary operators, if the data on either side is `NULL`, the result is also `NULL`.

Don't confuse this operator with a logical "or": in SQL, it's denoted by the `OR` keyword. It's also not worth doing concatenation using `+`.

Examples

```
SELECT "fo" || "o";
```

Matching a string by pattern

`REGEXP` and `RLIKE` are aliases used to call Re2::Grep. `MATCH`: Same for Re2::Match.

`LIKE` works as follows:

- Patterns can include two special characters:
  - `%`: Zero or more of any characters.
  - `_`: Exactly one of any character.

All other characters are literals that represent themselves.

- As opposed to `REGEXP`, `LIKE` must be matched exactly. For example, to search a substring, add `%` at the beginning and end of the pattern.
- `ILIKE` is a case-insensitive version of `LIKE`.
- If `LIKE` is applied to the key column of the sorted table and the pattern doesn't start with a special character, filtering by prefix drills down directly to the cluster level, which in some cases lets you avoid the full table scan. This optimization is disabled for `ILIKE`.
- To escape special characters, specify the escaped character after the pattern using the `ESCAPE '?'` keyword. Instead of `?` you can use any character except `%`, `_` and `\`. For example, if you use a question mark as an escape character, the expressions `?%`, `?_` and `??` will match their second character in the template: percent, underscore, and question mark, respectively. The escape character is undefined by default.

The most popular way to use the `LIKE` and `REGEXP` keywords is to filter a table using the statements with the `WHERE` clause. However, there are no restrictions on using templates in this context: you can use them in most of contexts involving strings, for example, with concatenation by using `||`.

LIKE / ILIKE with a fulltext index

For fulltext indexes with n-grams (`use_filter_ngram` / `use_filter_edge_ngram`), `LIKE` / `ILIKE` over the indexed text column is supported.

```
SELECT id, title
FROM articles VIEW ft_idx
WHERE body ILIKE "%learn%ing%"
LIMIT 20;
```

Examples

```
SELECT * FROM my_table
WHERE string_column REGEXP '\\d+';
-- the second slash is required because
-- all the standard string literals in SQL
-- can accept C-escaped strings
```

```
SELECT
    string_column LIKE '___!_!_!_!!!!!!' ESCAPE '!'
    -- searches for a string of exactly 9 characters:
    --   3 arbitrary characters
    --   followed by 3 underscores
    --  and 3 exclamation marks
FROM my_table;
```

```
SELECT * FROM my_table
WHERE key LIKE 'foo%bar';
-- if the table is sorted by key, it will only scan the keys,
-- starting with "foo", and then, among them,
-- will leave only those that end in "bar"
```

Operators

Arithmetic operators

The operators `+`, `-`, `*`, `/`, `%` are defined for primitive data types that are variations of numbers.

For the Decimal data type, bankers rounding is used (to the nearest even integer).

Examples

```
SELECT 2 + 2;
```

```
SELECT 0.0 / 0.0;
```

Comparison operators

The operators `=`, `==`, `!=`, `<>`, `>`, `<`, `>=`, `<=` are defined for:

- Primitive data types except Yson and Json.
- Tuples and structures with the same set of fields. No order is defined for structures, but you can check for (non-)equality. Tuples are compared element-by-element left to right.

Examples

```
SELECT 2 > 1;
```

Logical operators

Use the operators `AND`, `OR`, `XOR` for logical operations on Boolean values (`Bool`).

Examples*

```
SELECT 3 > 0 AND false;
```

Bitwise operators

Bitwise operations on numbers:

- `&`, `|`, `^`: AND, OR, and XOR, respectively. Don't confuse bitwise operations with the related keywords. The keywords `AND`, `OR`, and `XOR` are used *for Boolean values only*, but not for numbers.
- `~`: A negation.
- `<`, `>`: Left or right shifts.
- `|<`, `>|`: Circular left or right shifts.

Examples

```
SELECT
    key << 10 AS key,
    ~value AS value
FROM my_table;
```

Precedence and associativity of operators

Operator precedence determines the order of evaluation of an expression that contains different operators.
For example, the expression `1 + 2 * 3` is evaluated as `1 + (2 * 3)` because the multiplication operator has a higher precedence than the addition operator.

Associativity determines the order of evaluating expressions containing operators of the same type.
For example, the expression `1 + 2 + 3` is evaluated as `(1 + 2) + 3` because the addition operator is left-associative.
On the other hand, the expression `a ?? b ?? c` is evaluated as `a ?? (b ?? c)` because the `??` operator is right-associative

The table below shows precedence and associativity of YQL operators.
The operators in the table are listed in descending order of precedence.

Priority	Operator	Description	Associativity
1	`a[], a.foo, a()`	Accessing a container item, calling a function	Left
2	`+a, -a, ~a, NOT a`	Unary operators: plus, minus, bitwise and logical negation	Right
3	`a||b`	String concatenation	Left
4	`a*b, a/b, a%b`	Multiplication, division, remainder of division	Left
5	`a+b, a-b`	Addition/Subtraction	Left
6	`a ?? b`	Operator notation for NVL/COALESCE	Right
7	`a<b, a>b, a|<b, a>|b, a|b, a^b, a&b`	Shift operators and logical bit operators	Left
8	`a<b, a=b, a=b, a>b`	Comparison	Left
9	`a IN b`	Occurrence of an element in a set	Left
9	`a==b, a=b, a!=b, a<>b,` `a is (not) distinct from b`	Comparison for (non-)equality	Left
10	`a XOR b`	Logical XOR	Left
11	`a AND b`	Logical AND	Left
12	`a OR b`	Logical OR	Left

IS [NOT]NULL

Matching an empty value (`NULL`). Since `NULL` is a special value equal to nothing, the ordinary comparison operators can't be used to match it.

Examples

```
SELECT key FROM my_table
WHERE value IS NOT NULL;
```

IS [NOT]DISTINCT FROM

Comparing of two values. Unlike the regular comparison operators, NULLs are treated as equal to each other.
More precisely, the comparison is carried out according to the following rules:

1. The operators `IS DISTINCT FROM` / `IS NOT DISTINCT FROM` are defined for those and only for those arguments for which the operators `!=` and `=` are defined.
2. The result of `IS NOT DISTINCT FROM` is equal to the logical negation of the `IS DISTINCT FROM` result for these arguments.
3. If the result of the `==` operator is not equal to zero for some arguments, then it is equal to the result of the `IS NOT DISTINCT FROM` operator for the same arguments.
4. If both arguments are empty `Optional` or `NULL`s, then the value of `IS NOT DISTINCT FROM` is `True`.
5. The result of `IS NOT DISTINCT FROM` for an empty `Optional` or `NULL` and filled-in `Optional` or non-`Optional` value is `False`.

For values of composite types, these rules are used recursively.

BETWEEN

Checking whether a value is in a range. It's equivalent to two conditions with `>=` and `<=` (range boundaries are included). Can be used with the `NOT` prefix to support inversion.

Examples

```
SELECT * FROM my_table
WHERE key BETWEEN 10 AND 20;
```

IN

Checking whether a value is inside of a set of values. It's logically equivalent to a chain of equality comparisons using `OR` but implemented more efficiently.

> **Warning**
>
> Unlike a similar keyword in Python, in YQL `IN` **DOES NOT** search for a substring inside a string. To search for a substring, use the function String::Contains or LIKE/REGEXP mentioned above.

Immediately after `IN`, you can specify the `COMPACT` modifier.
If `COMPACT` is not specified, then `IN` with a subquery is executed as a relevant `JOIN` (`LEFT SEMI` for `IN` and `LEFT ONLY` for `NOT IN`), if possible.
Using the `COMPACT` modifier forces the in-memory execution strategy: a hash table is immediately built from the contents of the right `IN` part in-memory, and then the left part is filtered.

The `COMPACT` modifier must be used with care. Since the hash table is built in-memory, the query may fail if the right part of `IN` contains many large or different elements.

Examples

```
SELECT column IN (1, 2, 3)
FROM my_table;
```

```
SELECT * FROM my_table
WHERE string_column IN ("a", "b", "c");
```

```
$foo = AsList(1, 2, 3);
SELECT 1 IN $foo;
```

```
$values = (SELECT column + 1 FROM table);
SELECT * FROM my_table WHERE
    -- filtering by an in-memory hash table for one_table
    column1 IN COMPACT $values AND
    -- followed by LEFT ONLY JOIN with other_table
    column2 NOT IN (SELECT other_column FROM other_table);
```

AS

Can be used in the following scenarios:

- Adding a short name (alias) for columns or tables within the query.
- Using named arguments in function calls.
- To specify the target type in the case of explicit type casting, see CAST.

Examples

```
SELECT key AS k FROM my_table;
```

```
SELECT t.key FROM my_table AS t;
```

```
SELECT
    MyFunction(key, 123 AS my_optional_arg)
FROM my_table;
```

CAST

Tries to cast the value to the specified type. The attempt may fail and return `NULL`. When used with numbers, it may lose precision or most significant bits.

For the Decimal parametric data type, two additional arguments are specified:

- Total number of decimal places (up to 35, inclusive).
- Number of places after the decimal point (out of the total number, meaning it can't be larger than the previous argument).

Examples

```
SELECT
    CAST("12345" AS Double),               -- 12345.0
    CAST(1.2345 AS Uint8),                 -- 1
    CAST(12345 AS String),                 -- "12345"
    CAST("1.2345" AS Decimal(5, 2)),       -- 1.23
    CAST("xyz" AS Uint64) IS NULL,         -- true, since it failed
    CAST(-1 AS Uint16) IS NULL,            -- true, negative to unsigned
    CAST([-1, 0, 1] AS List<Uint8?>),            -- [null, 0, 1]
-- The element type is optional: the failed element is null.
    CAST(["3.14", "bad", "42"] AS List<Float>),   -- [3.14, 42]
-- The element type is not optional: the failed element is removed.
    CAST(255 AS Uint8),                    -- 255
    CAST(256 AS Uint8) IS NULL;            -- true, out of range
```

BITCAST

Performs a bitwise conversion of an integer value to the specified integer type. The conversion is always successful, but may lose precision or high-order bits.

Examples

```
SELECT
    BITCAST(100000ul AS Uint32),     -- 100000
    BITCAST(100000ul AS Int16),      -- -31072
    BITCAST(100000ul AS Uint16),     -- 34464
    BITCAST(-1 AS Int16),            -- -1
    BITCAST(-1 AS Uint16);           -- 65535
```

CASE

Conditional expressions and branching. It's similar to `if`, `switch` and ternary operators in the imperative programming languages. If the result of the `WHEN` expression is `true`, the value of the `CASE` expression becomes the result following the condition, and the rest of the `CASE` expression isn't calculated. If the condition is not met, all the `WHEN` clauses that follow are checked. If none of the `WHEN` clauses are met, the `CASE` value is assigned the result from the `ELSE` clause.
The `ELSE` branch is mandatory in the `CASE` expression. Expressions in `WHEN` are checked sequentially, from top to bottom.

Since its syntax is quite sophisticated, it's often more convenient to use the built-in function IF.

Examples

```
SELECT
  CASE
    WHEN value > 0
    THEN "positive"
    ELSE "negative"
  END
FROM my_table;
```

```
SELECT
  CASE value
    WHEN 0 THEN "zero"
    WHEN 1 THEN "one"
    ELSE "not zero or one"
  END
FROM my_table;
```

Named expressions

Complex queries may be sophisticated, containing lots of nested levels and/or repeating parts. In YQL, you can use named expressions to assign a name to an arbitrary expression or subquery. Named expressions can be referenced in other expressions or subqueries. In this case, the original expression/subquery is actually substituted at point of use.

A named expression is defined as follows:

```
<named-expr> = <expression> | <subquery>;
```

Here `<named-expr>` consists of a $ character and an arbitrary non-empty identifier (for example, `$foo`).

If the expression on the right is a tuple, you can automatically unpack it by specifying several named expressions separated by commas on the left:

```
<named-expr1>, <named-expr2>, <named-expr3> ... = <expression-returning-tuple>;
```

In this case, the number of expressions must match the tuple size.

Each named expression has a scope. It starts immediately after the definition of a named expression and ends at the end of the nearest enclosed namescope (for example, at the end of the query or at the end of the body of the lambda function, ACTION). Redefining a named expression with the same name hides the previous expression from the current scope.

If the named expression has never been used, a warning is issued. To avoid such a warning, use the underscore as the first character in the ID (for example, `$_foo`).
The named expression `$_` is called an anonymous named expression and is processed in a special way: it works as if `$_` would be automatically replaced by `$_<some_uniq_name>`.
Anonymous named expressions are convenient when you don't need the expression value. For example, to fetch the second element from a tuple of three elements, you can write:

```
$_, $second, $_ = AsTuple(1, 2, 3);
select $second;
```

An attempt to reference an anonymous named expression results in an error:

```
$_ = 1;
select $_; --- error: Unable to reference anonymous name $_
```

Anonymous argument names are also supported for lambda functions, ACTION.

Note

If named expression substitution results in completely identical subgraphs in the query execution graph, the graphs are combined to execute a subgraph only once.

Examples

```
$multiplier = 712;
SELECT
  a * $multiplier, -- $multiplier is 712
  b * $multiplier,
  (a + b) * $multiplier
FROM abc_table;
$multiplier = c;
SELECT
  a * $multiplier -- $multiplier is column c
FROM abc_table;
```

```
$intermediate = (
  SELECT
    value * value AS square,
    value
  FROM my_table
);
SELECT a.square * b.value
FROM $intermediate AS a
INNER JOIN $intermediate AS b
ON a.value == b.square;
```

```
$a, $_, $c = AsTuple(1, 5u, "test"); -- unpack a tuple
SELECT $a, $c;
```

```
$x, $y = AsTuple($y, $x); -- swap expression values
```

Table expressions

A table expression is an expression that returns a table. Table expressions in YQL are as follows:

- Subqueries: `(SELECT key, subkey FROM T)`
- Named subqueries: `$foo = SELECT * FROM T;` (in this case, `$foo` is also a table expression)

Semantics of a table expression depends on the context where it is used. In YQL, table expressions can be used in the following contexts:

- Table context: after FROM. In this case, table expressions work as expected: for example, `$input = SELECT a, b, c FROM T; SELECT * FROM $input` returns a table with three columns. The table context also occurs after UNION ALL, JOIN;

- Vector context: after IN. In this context, the table expression must contain exactly one column (the name of this column doesn't affect the expression result in any way). A table expression in a vector context is typed as a list (the type of the list element is the same as the column type in this case). Example: `SELECT * FROM T WHERE key IN (SELECT k FROM T1)`;
- A scalar context arises *in all the other cases*. As in a vector context, a table expression must contain exactly one column, but the value of the table expression is a scalar, that is, an arbitrarily selected value of this column (if no rows are returned, the result is `NULL`). Example: `$count = SELECT COUNT(*) FROM T; SELECT * FROM T ORDER BY key LIMIT $count / 2`;

The order of rows in a table context, the order of elements in a vector context, and the rule for selecting a value from a scalar context (if multiple values are returned), aren't defined. This order also cannot be affected by `ORDER BY`: `ORDER BY` without `LIMIT` is ignored in table expressions with a warning, and `ORDER BY` with `LIMIT` defines a set of elements rather than the order within that set.

Lambda functions

Let you combine multiple expressions into a single callable value.

List arguments in round brackets, following them by the arrow and lambda function body. The lambda function body includes either an expression in round brackets or curly brackets around an optional chain of named expressions assignments and the call result after the `RETURN` keyword in the last expression.

The scope for the lambda body: first the local named expressions, then arguments, then named expressions defined above by the lambda function at the top level of the query.

Only use pure expressions inside the lambda body (those might also be other lambdas, possibly passed through arguments). However, you can't use SELECT, INSERT INTO, or other top-level expressions.

One or more of the last lambda parameters can be marked with a question mark as optional: if they haven't been specified when calling lambda, they are assigned the `NULL` value.

Examples

```
$f = ($y) -> {
    $prefix = "x";
    RETURN $prefix || $y;
};

$g = ($y) -> ("x" || $y);

$h = ($x, $y?) -> ($x + ($y ?? 0));

SELECT $f("y"), $g("z"), $h(1), $h(2, 3); -- "xy", "xz", 1, 5
```

```
-- if the lambda result is calculated by a single expression, then you can use a more compact syntax:
$f = ($x, $_) -> ($x || "suffix"); -- the second argument is not used
SELECT $f("prefix_", "whatever");
```

Accessing containers

For accessing the values inside containers:

- `Struct<>`, `Tuple<>` and `Variant<>`, use a **dot**. The set of keys (for the tuple and the corresponding variant — indexes) is known at the query compilation time. The key is **validated** before beginning the query execution.
- `List<>` and `Dict<>`, use **square brackets**. The set of keys (set of indexes for keys) is known only at the query execution time. The key is **not validated** before beginning the query execution. If no value is found, an empty value (NULL) is returned.

Description and list of available containers.

When using this syntax to access containers within table columns, be sure to specify the full column name, including the table name or table alias separated by a dot (see the first example below).

Examples

```
SELECT
  t.struct.member,
  t.tuple.7,
  t.dict["key"],
  t.list[7]
FROM my_table AS t;
```

```
SELECT
  Sample::ReturnsStruct().member;
```

Correlated subqueries, EXISTS, and NOT EXISTS

A correlated subquery refers to a column or table alias from an outer query. Each subquery in YQL has its own scope and cannot refer to columns or table aliases from an outer query. Therefore, YQL does not support correlated subqueries. Support for correlated subqueries is planned for future releases.

Support matrix

SQL pattern	YQL support	Use instead
An uncorrelated subquery in `FROM` or `IN`	Supported	Use the subquery directly
An uncorrelated `EXISTS (SELECT ...)`	Supported; returns `true` if the subquery contains at least one row	Use directly when the condition does not depend on an outer row
An uncorrelated `NOT EXISTS (SELECT ...)`	Supported; returns `true` if the subquery is empty	Use directly when the condition does not depend on an outer row
`EXISTS` referring to an outer column	Not supported	`LEFT SEMI JOIN`, or `INNER JOIN` with unique right-side keys
`NOT EXISTS` referring to an outer column	Not supported	`LEFT ONLY JOIN`
A correlated scalar or aggregate subquery	Not supported	Precompute the result and use `JOIN`

EXISTS

An uncorrelated `EXISTS` checks whether its subquery contains at least one row. Its result does not depend on a row from an outer query. For example, the following query returns every client if the `orders` table is non-empty and no clients otherwise:

```
SELECT client_id, name
FROM clients
WHERE EXISTS (
    SELECT 1
    FROM orders
);
```

The following common SQL query is not supported because the inner query refers to the outer alias `c`:

```
-- Not supported in YQL.
SELECT c.client_id, c.name
FROM clients AS c
WHERE EXISTS (
    SELECT 1
    FROM orders AS o
    WHERE o.client_id = c.client_id
);
```

Use `LEFT SEMI JOIN` to return a client only when at least one matching order exists:

```
SELECT c.client_id, c.name
FROM clients AS c
LEFT SEMI JOIN orders AS o
ON o.client_id = c.client_id;
```

`LEFT SEMI JOIN` returns columns from the left side only. Several matching rows on the right side do not duplicate a row from the left side, so this rewrite preserves the existence-check semantics.

Alternatively, use `INNER JOIN` after deduplicating the matching keys on the right side:

```
$order_clients = (
    SELECT client_id
    FROM orders
    GROUP BY client_id
);

SELECT c.client_id, c.name
FROM clients AS c
INNER JOIN $order_clients AS o
ON o.client_id = c.client_id;
```

The right side contains at most one row for each key, so the join does not duplicate rows from the left side. Applying `DISTINCT` to the left-side columns is unnecessary and can collapse equal rows from the outer input.

For a single non-optional key, `IN` is another possible rewrite:

```
SELECT c.client_id, c.name
FROM clients AS c
WHERE c.client_id IN (
    SELECT o.client_id
    FROM orders AS o
);
```

NOT EXISTS

An uncorrelated `NOT EXISTS` returns the inverse of `EXISTS`: `true` when its subquery is empty and `false` when the subquery contains at least one row. Its result also does not depend on a row from an outer query.

Use `LEFT ONLY JOIN` to return a client only when no matching order exists:

```
SELECT c.client_id, c.name
FROM clients AS c
LEFT ONLY JOIN orders AS o
ON o.client_id = c.client_id;
```

Correlated aggregate subqueries

Precompute the aggregate and join it to the outer table:

```
$last_order = (
    SELECT client_id, MAX(created_at) AS last_order_at
    FROM orders
    GROUP BY client_id
);

SELECT c.client_id, o.last_order_at
FROM clients AS c
LEFT JOIN $last_order AS o
ON o.client_id = c.client_id;
```

See also

- JOIN
- Expressions: IN
- Known limitations

Classic SQL constructs not supported yet

Correlated EXISTS and NOT EXISTS

`EXISTS` and `NOT EXISTS` can be used with uncorrelated subqueries. Correlated subqueries are not supported. To select rows based on the presence or absence of matching rows, use `LEFT SEMI JOIN` or `LEFT ONLY JOIN`. For details, see Correlated subqueries, EXISTS, and NOT EXISTS.

INTERSECT and EXCEPT

YQL does not support `INTERSECT` or `EXCEPT`.

NATURAL JOIN

An alternative is to explicitly list the matching columns on both sides.

NOW() / CURRENT_TIME()

An alternative is to use the functions CurrentUtcDate, CurrentUtcDatetime and CurrentUtcTimestamp.

ACTION

DEFINE ACTION

Specifies a named action that is a parameterizable block of multiple top-level expressions.

Syntax

1. `DEFINE ACTION`: action definition.
2. Action name that will be used to access the defined action further in the query.
3. The values of parameter names are listed in parentheses.
4. `AS` keyword.
5. List of top-level expressions.
6. `END DEFINE`: The marker of the last expression inside the action.

One or more of the last parameters can be marked with a question mark `?` as optional. If they are omitted during the call, they will be assigned the `NULL` value.

DO

Executes an `ACTION` with the specified parameters.

Syntax

1. `DO`: Executing an action.
2. The named expression for which the action is defined.
3. The values to be used as parameters are listed in parentheses.

`EMPTY_ACTION`: An action that does nothing.

Example

```
DEFINE ACTION $hello_world($name, $suffix?) AS
    $name = $name ?? ($suffix ?? "world");
    SELECT "Hello, " || $name || "!";
END DEFINE;

DO EMPTY_ACTION();
DO $hello_world(NULL);
DO $hello_world("John");
DO $hello_world(NULL, "Earth");
```

BEGIN .. END DO

Performing an action without declaring it (anonymous action).

Syntax

1. `BEGIN`.
2. List of top-level expressions.
3. `END DO`.

An anonymous action can't include any parameters.

Example

```
DO BEGIN
    SELECT 1;
    SELECT 2 -- here and in the previous example, you might omit ';' before END
END DO
```

ALTER ASYNC REPLICATION

The `ALTER ASYNC REPLICATION` statement modifies the status and parameters of an asynchronous replication instance.

Syntax

```
ALTER ASYNC REPLICATION <name> SET (option = value [, ...])
```

Parameters

- `name` — a name of the asynchronous replication instance.
- `SET (option = value [, ...])` — asynchronous replication parameters:
  - `STATE` — the state of asynchronous replication. This parameter can only be used in combination with the `FAILOVER_MODE` parameter (see below). Valid values are:
    - `DONE` — completion of the asynchronous replication process.
  - `FAILOVER_MODE` — the mode for changing the replication state. This parameter can only be used in combination with the `STATE` parameter. Valid values are:
    - `FORCE` — forced failover.
- Authentication settings for the source database (one of the following):
  - Using a token:
    - `TOKEN_SECRET_PATH` — the secret that contains the token.
  - Using a username and password:
    - `USER` — the username.
    - `PASSWORD_SECRET_PATH` — the secret that contains the password.
  - Using a service agent:
    - `SERVICE_ACCOUNT_ID` — the identificator of the service account.
    - `INITIAL_TOKEN_SECRET_PATH` — the secret that contains the account's token. It is used for initial authentication.

Examples

The following statement forces the asynchronous replication process to complete:

```
ALTER ASYNC REPLICATION my_replication SET (STATE = "DONE", FAILOVER_MODE = "FORCE");
```

The following query changes the secret:

```
ALTER ASYNC REPLICATION my_replication SET (TOKEN_SECRET_PATH = "my_token");
```

See also

- CREATE ASYNC REPLICATION
- DROP ASYNC REPLICATION

ALTER DATABASE

Changes database settings.

Syntax

```
ALTER DATABASE path action;
```

Parameters

- `path` — path to the database;
- `action` — any of the database modification actions described below:
  - Change database owner.
  - Change database settings.

Change database owner

Changes the database owner. Only a database administrator can perform this operation.

Syntax

```
ALTER DATABASE path OWNER TO user_name;
```

Parameters

- `path` — path to the database;
- `user_name` — name of the user who will become the database owner.

Examples

Make user `user1` the owner of database `/Root/test`:

```
ALTER DATABASE `/Root/test` OWNER TO user1;
```

Change database settings

Changes database settings. Only a database administrator can perform this operation.

Syntax

```
ALTER DATABASE path SET (key = value, ...)
```

Parameters

- `path` — path to the database.
- `key` — name of the setting to change:
  - `MAX_PATHS` — maximum number of paths (schema objects) in the database.
  - `MAX_SHARDS` — maximum number of tablets in the database.
  - `MAX_CHILDREN_IN_DIR` — maximum number of objects in a directory.
  - `MAX_SHARDS_IN_PATH` — maximum number of tablets associated with a single schema object. For example, the maximum number of partitions of one table.
- `value` — new value for the setting.

Examples

Change the limit on the maximum number of paths (schema objects) for database `/Root/test`:

```
ALTER DATABASE `/Root/test` SET (MAX_PATHS = 20000);
```

ALTER GROUP

Adds/removes the group to/from a specific user. You can list multiple users under one operator.

Syntax:

```
ALTER GROUP role_name ADD USER user_name [, ... ]
ALTER GROUP role_name DROP USER user_name [, ... ]
```

- `role_name`: The name of the group.
- `user_name`: The name of the user.

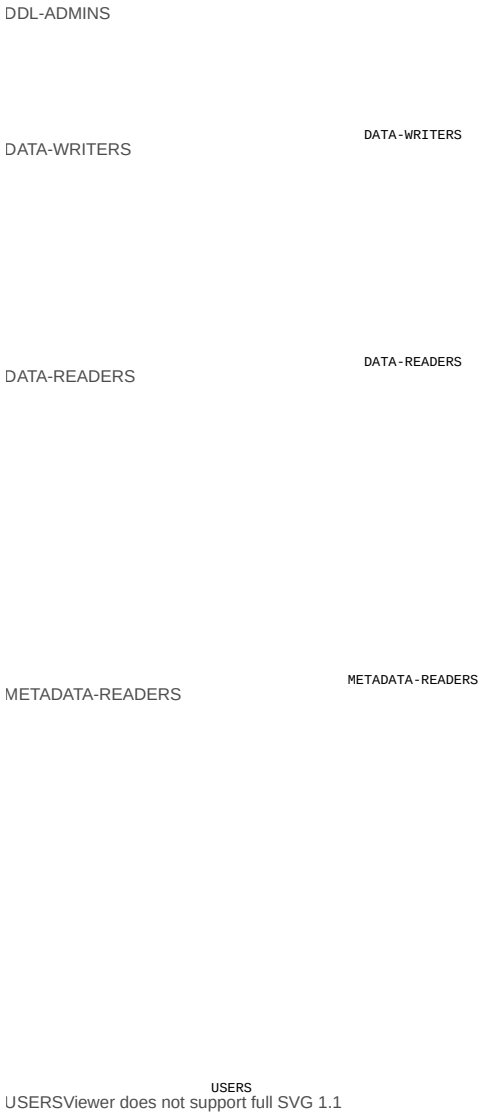
Built-in groups

The YDB cluster has built-in groups providing predefined role sets:

Group	Description
`ADMINS`	Unlimited rights over the entire cluster schema
`DATABASE-ADMINS`	Rights to create and delete databases (`CreateDatabase` , `DropDatabase`)
`ACCESS-ADMINS`	Rights to manage other users' permissions (`GrantAccessRights`)
`DDL-ADMINS`	Rights to alter database schemas (`CreateDirectory` , `CreateTable` , `WriteAttributes` , `AlterSchema` , `RemoveSchema`)
`DATA-WRITERS`	Rights to modify data (`UpdateRow` , `EraseRow`)
`DATA-READERS`	Rights to read data (`SelectRow`)
`METADATA-READERS`	Rights to read metadata, without access to data (`DescribeSchema` and `ReadAttributes`)
`USERS`	Rights to connect to databases (`ConnectDatabase`)

By default, all users are included in the `USERS` group, and the `root` user is included in the `ADMINS` group.

Below is a diagram demonstrating how groups inherit permissions from each other. For example, `DATA-WRITERS` includes all permissions of `DATA-READERS`:


DDL-ADMINS
DATA-WRITERS
DATA-WRITERS
DATA-READERS
DATA-READERS
METADATA-READERS
METADATA-READERS
USERS
USERSViewer does not support full SVG 1.1

ALTER TABLE

Using the `ALTER TABLE` command, you can change the column composition and additional parameters row and column tables. You can specify multiple actions in a single command. In general, the `ALTER TABLE` command looks like this:

```
ALTER TABLE table_name action1, action2, ..., actionN;
```

`action` is any table modification action from those described below:

- Renaming a table.
- Working with columns of row and column tables.
- Adding or removing a change stream.
- Working with indexes.
- Working with column groups of a row table.
- Changing additional table parameters.
- Running forced compaction.
- Changing the NOT NULL constraint.

Adding, deleting, and renaming an index

Adding an index

`ADD INDEX` — adds an index with the specified name and type for the given set of columns in row tables. Grammar:

```
ALTER TABLE `<table_name>`
  ADD INDEX `<index_name>`
    [GLOBAL|LOCAL]
    [UNIQUE]
    [SYNC|ASYNC]
    [USING <index_type>]
    ON ( <index_columns> )
    [COVER ( <cover_columns> )]
    [WITH ( <parameter_name> = <parameter_value>[, ...])]
  [,   ...]
```

- `GLOBAL/LOCAL` — global or local index, depending on the index type (`<index_type>`), only one of them may be available:
  - `GLOBAL` — an index implemented as a separate table or a set of tables. Synchronous update of such an index requires distributed transactions.
  - `LOCAL` — a local index within a shard of a columnar or row-based table, does not require distributed transactions during update, but does not provide pruning during search.
- `<index_name>` — unique index name by which data can be accessed.
- `SYNC/ASYNC` — indicator of index synchrony.
  - `SYNC` - synchronous index. Default value.
  - `ASYNC` - asynchronous index.
- `UNIQUE` — indicator of a unique secondary index. A unique index must be global synchronous (`GLOBAL UNIQUE SYNC`) and must not contain the `USING <index_type>` construct.
- `<index_type>` - index type, currently supported:
  - `secondary` — secondary index. Only the `GLOBAL` mode is available for secondary indexes. This is the default index type.
  - `vector_kmeans_tree` — vector index. More details are described in the Vector index section.
  - `fulltext_plain` — basic full-text index. More details are described in Fulltext index.
  - `fulltext_relevance` — full-text index with BM25 statistics for relevance calculation. More details are described in Fulltext index.
  - `json` — JSON index to speed up predicates `JSON_EXISTS` and `JSON_VALUE` on a column of type `Json` or `JsonDocument` . More details are described in JSON-index.
  - `bloom_filter` — local Bloom index. Only `LOCAL` is available. See ALTER TABLE ADD INDEX.
  - `bloom_ngram_filter` — local N-gram Bloom index. Only `LOCAL` is available. See ALTER TABLE ADD INDEX.
- `<index_columns>` — comma-separated list of column names of the table being created, which determines the composition and order of columns included in the index key. Must be specified. The index key will consist of these columns with the addition of the table's primary key columns.
- `<cover_columns>` — comma-separated list of column names of the table being created that will be stored in the index in addition to the index key columns, allowing you to get additional data without accessing the table. Empty by default.
- `<parameter_name>` and `<parameter_value>` are index parameters specific to a particular `<index_type>` .

Parameters for all index types:

- - maximum number of `parallel` handlers based on partitions involved in index building (an integer between `1` and `MaxBuildIndexShardsInFlight` from `SchemeShardConfig`).
- If the parameter is not specified, the default value `32` or `MaxBuildIndexShardsInFlight` is currently used, whichever is smaller. `MaxBuildIndexShardsInFlight` defaults to `1000` . In future versions, the default parallelism selection logic may change.
- You can set a lower limit to reduce the impact of index building on database performance.
- You can also set a higher limit to speed up index building if you have sufficient hardware resources.

Parameters specific to vector indexes:

- common parameters for all vector indexes:
  - `vector_dimension` - embedding vector dimensionality (should be between 1 and 16384)
  - `vector_type` - vector value type (`float` , `uint8` , or `int8`)
  - `distance` - distance function (`cosine` , `manhattan` , or `euclidean`), mutually exclusive with `similarity`
  - `similarity` - similarity function (`inner_product` or `cosine`), mutually exclusive with `distance`
- specific parameters for `vector_kmeans_tree` (read more about the index type):
  - `clusters` - number of centroids for k-means algorithm (should be between 2 and 2048)
  - `levels` - number of levels in the tree (should be between 1 and 16)
  - `overlap_clusters` - the number of nearest clusters to add each vector to (default 1)
  - the total number of nodes in the tree, calculated as `clusters` raised to the power of `levels` , should be no more than 1073741824
  - the product of `vector_dimension` and `clusters` should be no more than 4194304

> **Note**
>
> For vector indexes, the `vector_type` and `vector_dimension` parameters can be omitted if the table is not empty — they are determined automatically from the row contents. The `levels` and `clusters` parameters are also determined automatically, and for them the table can be empty, but doing so is strongly not recommended because the default values in this case are `levels` =1, `clusters` =2; it is much better to create the index on a table that already has data loaded, so that the values can be correctly determined.

Parameters specific to full-text indexes:

- common parameters for all fulltext indexes:
  - `tokenizer` - tokenizer type (`standard`, `whitespace`, or `keyword`)
  - `use_filter_lowercase` - lowercase filter (`true` or `false`)
  - `use_filter_length` - token length filter (`true` or `false`); when `true`, tokens shorter than `filter_length_min` or longer than `filter_length_max` are not indexed and are ignored during search
  - `filter_length_min` - minimum token length (positive integer); only applied when `use_filter_length=true`
  - `filter_length_max` - maximum token length (positive integer); only applied when `use_filter_length=true`
  - `use_filter_snowball` - Snowball stemmer filter (`true` or `false`)
  - `language` - language for the Snowball stemmer (for example, `english`, `russian`)
  - `use_filter_ngram` - n-gram filter (`true` or `false`)
  - `use_filter_edge_ngram` - edge n-gram filter (`true` or `false`)
  - `filter_ngram_min_length` - minimum n-gram length (positive integer)
  - `filter_ngram_max_length` - maximum n-gram length (positive integer)

Parameters of local bloom indexes

- `bloom_filter`
  - `false_positive_probability`: Target false-positive rate of the filter: the fraction of fragments that are not skipped even though the requested value is not there (range `(0, 1)`). A lower value reduces extra reads but increases index size.
    - If omitted, the default is `0.0001` for row-oriented tables and `0.1` for column-oriented tables. The stricter default on row-oriented tables matches OLTP point lookups; on column-oriented tables it reflects analytical scans, where a larger trade-off between index size and fragment skipping is acceptable.
  - Indexed columns: YQL types that support equality comparison, except `Yson`, `Json`, and `JsonDocument` (see comparison operators). One column in a column-oriented table; multiple columns in a row-oriented table. On row-oriented tables, the indexed columns must form a left prefix of the primary key.
- `bloom_ngram_filter` (`String` and `Utf8` columns; column-oriented tables only)
  - `ngram_size`: N-gram length, an integer from `3` to `8` (default `3`).
  - `false_positive_probability`: Target false-positive rate (range `(0, 1)`; default `0.1`).
  - `case_sensitive`: Whether n-grams respect character case: `true` or `false` (default `true`).

You can also add a secondary index using the table index YDB CLI command.

Limitations

The `ADD INDEX` operation for creating global secondary (`GLOBAL`, `UNIQUE`, etc.) and vector indexes is supported only for row tables. For columnar tables, via `ADD INDEX`, only local bloom indexes are supported.

Features of local bloom indexes:

- The index is always local (`LOCAL`); there is no global variant.
- Queries do not use the `VIEW <index>` syntax (unlike, for example, fulltext indexes).
- The filter is applied on read only to data fragments where an index block was already stored during a write or portion merge, other fragments are not skipped by this index until merge.
- On row-oriented tables, `bloom_filter` is implemented as a **prefix bloom filter**: the indexed columns must form a left prefix of the primary key. The filter is built over that key prefix and accelerates point lookups and range scans that constrain the prefix. Multiple bloom indexes over the same prefix length are not allowed.

Limitations

- `COVER (...)` and extra index columns are not supported.
- Column-oriented tables allow only one indexed column. Row-oriented tables allow multiple indexed columns.
- `bloom_ngram_filter` is not supported on row-oriented tables.
- On row-oriented tables, the indexed columns of a `bloom_filter` must be a left prefix of the primary key. Non-prefix column sets are rejected.
- On row-oriented tables, two `bloom_filter` indexes cannot share the same prefix length (the same set of leading primary-key columns).

Examples

Secondary index:

```
ALTER TABLE `series`
  ADD INDEX `title_index`
  GLOBAL ON (`title`);
```

Vector index:

```
ALTER TABLE `series`
  ADD INDEX emb_cosine_idx GLOBAL SYNC USING vector_kmeans_tree
  ON (embedding) COVER (title)
  WITH (
    distance="cosine", vector_type="float", vector_dimension=512
  );
```

Full-text index:

```
ALTER TABLE `series`
  ADD INDEX ft_idx GLOBAL USING fulltext_plain
```

```
  ON (title)
  WITH (tokenizer=standard, use_filter_lowercase=true);
```

JSON index:

```
ALTER TABLE `series`
  ADD INDEX json_idx GLOBAL USING json
  ON (metadata);
```

Bloom index:

```
ALTER TABLE `/Root/Table`
  ADD INDEX idx_bloom LOCAL USING bloom_filter
  ON (resource_id)
  WITH (false_positive_probability = 0.01);
```

Bloom n-gram index:

```
ALTER TABLE `/Root/Table`
  ADD INDEX idx_ngram LOCAL USING bloom_ngram_filter
  ON (message)
  WITH (
    ngram_size = 3,
    false_positive_probability = 0.01,
    case_sensitive = true
  );

# # Changing index parameters {#alter-index}

Индексы имеют параметры, зависящие от типа, которые можно настраивать. Глобальные индексы, [синхронные](yfmva
r-0-yfmvarend#sync) или [асинхронные](yfmvar-1-yfmvarend#async), реализованы в виде скрытых таблиц, и их пара
метры автоматического партиционирования и реплик можно регулировать так же, как и настройки обычных таблиц.

{% note info %}

В настоящее время задание настроек партиционирования вторичных индексов при создании индекса не поддерживаетс
я ни в операторе [`ALTER TABLE ADD INDEX`](#add-index), ни в операторе [`CREATE TABLE INDEX`](../create_tabl
e/secondary_index.md).

{% endnote %}

```yql
ALTER TABLE <table_name> ALTER INDEX <index_name> SET <setting_name> <value>;
ALTER TABLE <table_name> ALTER INDEX <index_name> SET (<setting_name_1> = <value_1>, ...);
```

- `<table_name>` - name of the table whose index needs to be changed.
- `<index_name>` - name of the index to change.
- `<setting_name>` - name of the parameter to change. The set of allowed parameters depends on the index type:
  - for global secondary indexes:
    - AUTO_PARTITIONING_BY_SIZE
    - AUTO_PARTITIONING_BY_LOAD
    - AUTO_PARTITIONING_PARTITION_SIZE_MB
    - AUTO_PARTITIONING_MIN_PARTITIONS_COUNT
    - AUTO_PARTITIONING_MAX_PARTITIONS_COUNT
    - READ_REPLICAS_SETTINGS
  - for local bloom indexes (see Parameters of local bloom indexes):
    - `FALSE_POSITIVE_PROBABILITY`
    - `NGRAM_SIZE` and `CASE_SENSITIVE` (only for `bloom_ngram_filter`)

> **Note**
>
> The `RESET` operation for `ALTER INDEX` is not supported.

- `<value>` - new parameter value. Possible values include:
  - `ENABLED` or `DISABLED` for the `AUTO_PARTITIONING_BY_SIZE` and `AUTO_PARTITIONING_BY_LOAD` parameters
  - `"PER_AZ:<count>"` or `"ANY_AZ:<count>"` where `<count>` is the number of replicas for `READ_REPLICAS_SETTINGS`
  - for other parameters — an integer of type `Uint64`
  - for `FALSE_POSITIVE_PROBABILITY` — a floating-point number in the range `(0, 1)`; a smaller value usually reduces the number of false positives but increases the index size
  - for `NGRAM_SIZE` — an integer in the range from `3` to `8` (usually recommended to start with `3`)
  - for `CASE_SENSITIVE` — `true` or `false`

Example

The code in the following example enables automatic partitioning by load for the index named `title_index` in the table `series`, sets the minimum number of partitions to 5, and starts one replica in each availability zone (AZ) for each partition:

```
ALTER TABLE `series` ALTER INDEX `title_index` SET (
    AUTO_PARTITIONING_BY_LOAD = ENABLED,
    AUTO_PARTITIONING_MIN_PARTITIONS_COUNT = 5,
```
```

```
    READ_REPLICAS_SETTINGS = "PER_AZ:1"
);
```

For local bloom indexes, you can also change their specific parameters, for example:

```
ALTER TABLE `/Root/Table` ALTER INDEX idx_ngram SET (
    ngram_size = 4,
    false_positive_probability = 0.005,
    case_sensitive = false
);
```

Deleting an index

`DROP INDEX` — deletes the index with the specified name. The code below will delete the index named `title_index`.

```
ALTER TABLE `series` DROP INDEX `title_index`;
```

You can also delete an index using the table index YDB CLI command.

Renaming a secondary index

`RENAME INDEX` — renames the index with the specified name. If an index with the new name already exists, an error will be returned.

The ability to atomically replace an index under load is supported by the ydb table index rename YDB CLI command and specialized YDB SDK methods.

This applies to global secondary indexes (hidden index table and `--replace` mode). Local bloom indexes are not applicable to such atomic replacement under load.

Example of renaming an index:

```
ALTER TABLE `series` RENAME INDEX `title_index` TO `title_index_new`;
```

Changing columns

YDB supports adding columns to row and column tables, deleting non-key columns from tables, and changing properties of existing columns.

ADD COLUMN

Builds a new column with the specified name, type, and options for the specified table.

```
ALTER TABLE table_name ADD COLUMN column_name column_data_type [FAMILY <family_name>] [NULL | NOT NULL] [DEFA
ULT <default_value>] [COMPRESSION([algorithm=<algorithm_name>[, level=<value>]])] [ENCODING([OFF|DICT])];
```

When adding a `NOT NULL` column to a row-oriented table, you must also specify `DEFAULT` so that the new column can be populated in existing rows. Column-oriented tables do not support adding a `NOT NULL` column because they do not support `DEFAULT` values.

Request parameters

table_name

The path of the table to which you want to add a new column.

column_name

The name of the column to be created. When choosing a name for the column, consider the common column naming rules.

column_data_type

The data type of the column. The complete list of data types supported by YDB is available in the YQL data types section.

FAMILY <family_name> (column setting)

> **Warning**
>
> Supported only for row-oriented tables.

Specifies the belonging of this column to the specified group of columns. For more details, see the section Column groups.

DEFAULT <default_value>

> **Warning**
>
> The `DEFAULT` option is supported:
>
> - Only for row-oriented tables.
> - Only with literal values.

Allows you to set a default value for the column. If no value is specified for this column when inserting a row, the specified default value will be used. The default value must match the data type of the column.

The `DEFAULT false NOT NULL` construct is not allowed due to ambiguity of interpretation. In this case, you should use a comma-separated list or change the order of the options.

NULL

This column can contain `NULL` values (by default).

NOT NULL

This column does not accept `NULL` values.

COMPRESSION([algorithm=<algorithm_name>[, level=]])

> **Warning**
>
> Supported only for column-oriented tables.

The following compression parameters can be set for columns:

- `algorithm` — the data compression algorithm. Allowed values: `off` (disable compression), `lz4`, `zstd`.
- `level` — the compression level, supported only for the `zstd` algorithm (values from 0 to 22 are allowed).

If `COMPRESSION()` is specified without parameters, the default compression is used for the column. Currently, it is `lz4`; in future versions, it will be possible to configure the default compression at the cluster or table level.

ENCODING([OFF|DICT])

> **Warning**
>
> Supported only for column-oriented tables.

Allows you to set the data encoding method for the column.

Available options:

- `ENCODING(DICT)` — enables dictionary encoding. Repeating values are replaced with small integer identifiers, and the values themselves are stored in a dictionary. Dictionary encoding is effective for columns with low cardinality (a small number of unique

values). It reduces the amount of stored data and speeds up some operations. It is supported only for comparable data types, such as `String`, `Timestamp`, `UInt64`, and others. Using `ENCODING(DICT)` for incomparable types, such as `Json`, `JsonDocument`, or `Yson`, will result in an error.

- `ENCODING(OFF)` — disables special encoding. Data will be stored in the standard format without additional encoding.

If `ENCODING()` is set without parameters, the default encoding will be used for the column. Currently, it is `OFF`; in future versions, it will be possible to configure the default encoding at the database or table level.

Example

The code below will add a column named `views` with data type `Uint64` to the `episodes` table.

```
ALTER TABLE episodes ADD COLUMN views Uint64;
```

The code below will add a column named `rate` with data type `Double` and default value `5.0` to the `episodes` table.

```
ALTER TABLE episodes ADD COLUMN rate Double NOT NULL DEFAULT 5.0;
ALTER TABLE episodes ADD COLUMN rate Double (DEFAULT 5.0, NOT NULL); -- alternative syntax
```

ALTER COLUMN

Modifies properties of an existing column in the specified table. Property changes are applied without recreating the column. Some properties apply only to newly written data or during compaction (see the description of each property for details).

```
ALTER TABLE table_name ALTER COLUMN column_name SET [FAMILY <family_name>]{% if feature_alter_column_not_null
== true %} [NOT NULL]{% endif %} [DEFAULT <default_value>] [COMPRESSION([algorithm=<algorithm_name>[, level=<
value>]])] [ENCODING([OFF|DICT])];
ALTER TABLE table_name ALTER COLUMN column_name DROP [FAMILY] [NOT NULL] [DEFAULT] [COMPRESSION] [ENCODING];
```

Request parameters

table_name

The path of the table containing the column to change.

column_name

The name of the column to change in the specified table.

SET

Set a column property.

DROP

Remove a column property.

FAMILY <family_name> (column setting)

> **Warning**
>
> Supported only for row-oriented tables.

Specifies that this column belongs to the specified column group. For more information, see Column groups.

DEFAULT <default_value>

> **Warning**
>
> The `DEFAULT` option is supported:
>
> - Only for row tables.
> - Only with literal values.

Sets a default value for the column. If no value is specified for this column when inserting a row, the specified default value is used. The default value must match the column's data type.

NOT NULL

The following operations are available:

- `DROP NOT NULL` — removes the `NOT NULL` constraint from the column, again allowing `NULL` values.

For details, see Changing the `NOT NULL` constraint.

COMPRESSION([algorithm=<algorithm_name>[, level=]])

> **Warning**
>
> Supported only for column-oriented tables.

The following compression parameters can be set for columns:

- `algorithm` — data compression algorithm. Valid values: `off` (disable compression), `lz4`, `zstd`.
- `level` — compression level, supported only for the `zstd` algorithm (valid values from 0 to 22).

If `COMPRESSION()` is specified without parameters, the default compression is used for the column. Currently this is `lz4`; future versions will allow configuring default compression at the cluster or table level.

ENCODING([OFF|DICT])

> **Warning**
>
> Supported only for column-oriented tables.

Sets the encoding method for the column data.

Available options:

- `ENCODING(DICT)`: enables dictionary encoding. Duplicate values are replaced with small integer identifiers, and the values themselves are stored in a dictionary. Dictionary encoding is effective for columns with low cardinality (a small number of unique values). It reduces the amount of stored data and speeds up some operations. Supported only for comparable data types, such as `String`, `Timestamp`, `UInt64`, and others. For non-comparable types, such as `Json`, `JsonDocument`, or `Yson`, using `ENCODING(DICT)` will result in an error.
- `ENCODING(OFF)`: disables special encoding. Data will be stored in standard format without additional encoding.

If `ENCODING()` is specified without parameters, the default encoding will be used for the column. Currently it is `OFF`; future versions will allow configuring the default encoding at the database or table level.

A single `ALTER TABLE` statement can specify multiple `ALTER COLUMN` actions separated by commas.

Examples

The code below sets a default value for the `rate` column of the `episodes` table.

```
ALTER TABLE episodes ALTER COLUMN rate SET DEFAULT 5.0;
```

The code below changes default values for several columns of a table in a single statement — setting new defaults for `col_1` and `col_2` and clearing the default for `col_3`.

```
ALTER TABLE default_columns
    ALTER COLUMN col_1 SET DEFAULT "new_a"u,
    ALTER COLUMN col_2 SET DEFAULT 99,
    ALTER COLUMN col_3 DROP DEFAULT;
```

Reset column compression settings:

```
ALTER TABLE compressed_table ALTER COLUMN info SET COMPRESSION();
```

After the query runs, the column uses the default compression algorithm again (see the `COMPRESSION` option above).

Enable dictionary encoding on a column:

```
ALTER TABLE movies ALTER COLUMN genre SET ENCODING(DICT);
```

DROP COLUMN

Deletes a column with the specified name from the specified table.

```
ALTER TABLE table_name DROP COLUMN column_name;
```

Request parameters

table_name

The path of the table from which you want to delete a column.

column_name

The name of the column to be deleted.

Example

The code below will delete the column named `views` from the `episodes` table.

```
ALTER TABLE episodes DROP COLUMN views;
```

Changing column groups

Warning

Supported only for row-oriented tables.

The mechanism of column groups allows for improved performance of partial row read operations by dividing the storage of table columns into several groups. The most commonly used scenario is to organize the storage of infrequently used attributes into a separate column group.

Creating column groups

`ADD FAMILY`: Creates a new group of columns in the table. The code below creates the `family_small` column group in the `series_with_families` table.

```
ALTER TABLE series_with_families ADD FAMILY family_small (
    DATA = "ssd",
    COMPRESSION = "off"
);
```

Modifying column groups

Using the `ALTER COLUMN` command, you can change a column group for the specified column. The code below for the `release_date` column in the `series_with_families` table changes the column group to `family_small`.

```
ALTER TABLE series_with_families ALTER COLUMN release_date SET FAMILY family_small;
```

The two previous commands from listings 8 and 9 can be combined into one `ALTER TABLE` call. The code below creates the `family_small` column group and sets it for the `release_date` column in the `series_with_families` table.

```
ALTER TABLE series_with_families
    ADD FAMILY family_small (
        DATA = "ssd",
        COMPRESSION = "off"
    ),
    ALTER COLUMN release_date SET FAMILY family_small;
```

Using the `ALTER FAMILY` command, you can change the parameters of the column group.

Changing storage type

The code below changes the storage type to `rot` for the `default` column group in the `series_with_families` table:

```
ALTER TABLE series_with_families ALTER FAMILY default SET DATA "rot";
```

Note

Available types of storage devices depend on the YDB cluster configuration.

Changing compression codec

The code below changes the compression codec to `lz4` for the `default` column group in the `series_with_families` table:

```
ALTER TABLE series_with_families ALTER FAMILY default SET COMPRESSION "lz4";
```

Changing cache mode

When switching the cache mode to `in_memory` for an existing table with `ALTER TABLE`, the pages that are not yet in memory are loaded automatically.

If the table was previously in `in_memory` mode and `ALTER TABLE` then sets the cache mode to `regular`, the pages that are already in memory stay there, but they may later be evicted according to the general caching policy.

The code below changes the cache mode to `in_memory` for the `default` column group in the `series_with_families` table:

```
ALTER TABLE series_with_families ALTER FAMILY default SET CACHE_MODE "in_memory";
```

You can specify any column group parameters described in the `CREATE TABLE` command.

Changing the `NOT NULL` constraint

The `NOT NULL` constraint is a column-level data integrity constraint that prohibits writing `NULL` values. It ensures that the column always contains a valid value.

Dropping `NOT NULL`

`DROP NOT NULL` removes the `NOT NULL` constraint from the specified column.

For example, the following query will remove the `NOT NULL` constraint from column `column_name` in table `table_name`:

```
ALTER TABLE table_name ALTER COLUMN column_name DROP NOT NULL;
```

`DROP NOT NULL` can be applied to non-key columns in both row-oriented and column-oriented tables.

See also

- ALTER COLUMN

Modifying additional table parameters

Most parameters of row and column tables in YDB, listed on the table description page, can be modified using the `ALTER` command.

Generally, the command to modify any table parameter looks as follows:

```
ALTER TABLE table_name SET (key = value);
```

`key` — the name of the parameter, `value` — its new value.

Example of modifying the `TTL` parameter, which controls the time-to-live of records in a table:

```
ALTER TABLE series SET (TTL = Interval("PT0S") ON expire_at);
```

Resetting Additional Table Parameters

Some table parameters in YDB, listed on the table description page, can be reset using the `ALTER` command. The command to reset a table parameter looks as follows:

```
ALTER TABLE table_name RESET (key);
```

`key` — the name of the parameter.

For example, such a command will reset (remove) the `TTL` settings for row or column tables:

```
ALTER TABLE series RESET (TTL);
```

Adding or removing a changefeed

> **Warning**
>
> Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

`ADD CHANGEFEED <name> WITH (<option> = <value>[, ...])`: Adds a changefeed with the specified name and options.

Changefeed options

- `MODE`: Operation mode. Specifies what to write to a changefeed each time table data is altered.
  - `KEYS_ONLY`: Only the primary key components and change flag are written.
  - `UPDATES`: Updated column values that result from updates are written.
  - `NEW_IMAGE`: Any column values resulting from updates are written.
  - `OLD_IMAGE`: Any column values before updates are written.
  - `NEW_AND_OLD_IMAGES`: A combination of `NEW_IMAGE` and `OLD_IMAGE` modes. Any column values *prior to* and *resulting from* updates are written.
- `FORMAT`: Data write format.
  - `JSON`: Write data in JSON format.
  - `DEBEZIUM_JSON`: Write data in the Debezium-like JSON format.
- `VIRTUAL_TIMESTAMPS`: Enabling/disabling virtual timestamps. Disabled by default.
- `USER_SIDS`: Enabling/disabling user SIDs. Disabled by default.
- `TRACE_IDS`: Enabling/disabling OpenTelemetry trace identifiers. Disabled by default.
- `BARRIERS_INTERVAL` — barrier emission interval. The value type is `Interval`. Disabled by default.
- `RETENTION_PERIOD`: Record retention period. The value type is `Interval` and the default value is 24 hours (`Interval('PT24H')`).
- `TOPIC_AUTO_PARTITIONING`: Topic autopartitioning mode:
  - `ENABLED` – An autopartitioned topic will be created for this changefeed. The number of partitions in such a topic increases automatically as the table update rate increases. Topic autopartitioning parameters can be configured.
  - `DISABLED` – A topic without autopartitioning will be created for this changefeed. This is the default value.
- `TOPIC_MIN_ACTIVE_PARTITIONS`: The initial number of topic partitions. By default, the initial number of topic partitions is equal to the number of table partitions. For autopartitioned topics, the number of partitions increases automatically as the table update rate increases.
- `INITIAL_SCAN`: Enables/disables initial table scan. Disabled by default.

The code below adds a changefeed named `updates_feed`, where the values of updated table columns will be exported in JSON format:

```
ALTER TABLE `series` ADD CHANGEFEED `updates_feed` WITH (
    FORMAT = 'JSON',
    MODE = 'UPDATES'
);
```

Records in this changefeed will be stored for 24 hours (default value). The code in the following example will create a changefeed with a record retention period of 12 hours:

```
ALTER TABLE `series` ADD CHANGEFEED `updates_feed` WITH (
    FORMAT = 'JSON',
    MODE = 'UPDATES',
    RETENTION_PERIOD = Interval('PT12H')
);
```

The example of creating a changefeed with enabled virtual timestamps:

```
ALTER TABLE `series` ADD CHANGEFEED `updates_feed` WITH (
    FORMAT = 'JSON',
    MODE = 'UPDATES',
    VIRTUAL_TIMESTAMPS = TRUE
);
```

The example of creating a changefeed with virtual timestamps and barriers every 10 seconds:

```
ALTER TABLE `series` ADD CHANGEFEED `updates_feed` WITH (
    FORMAT = 'JSON',
    MODE = 'UPDATES',
    VIRTUAL_TIMESTAMPS = TRUE,
    BARRIERS_INTERVAL = Interval('PT10S')
);
```

Example of creating a changefeed with initial scan:

```
ALTER TABLE `series` ADD CHANGEFEED `updates_feed` WITH (
    FORMAT = 'JSON',
    MODE = 'UPDATES',
    INITIAL_SCAN = TRUE
);
```

Example of creating a changefeed with autopartitioning:

```
ALTER TABLE `series` ADD CHANGEFEED `updates_feed` WITH (
    FORMAT = 'JSON',
    MODE = 'UPDATES',
    TOPIC_AUTO_PARTITIONING = 'ENABLED',
    TOPIC_MIN_ACTIVE_PARTITIONS = 2
);
```

`DROP CHANGEFEED` : Deletes the changefeed with the specified name. The code below deletes the `updates_feed` changefeed:

```
ALTER TABLE `series` DROP CHANGEFEED `updates_feed`;
```

Renaming a table

```
ALTER TABLE old_table_name RENAME TO new_table_name;
```

Note

When choosing a name for the table, consider the common schema object naming rules.

If a table with the new name already exists, an error will be returned. The ability to transactionally replace a table under load is supported by specialized methods in CLI and SDK.

Warning

If a YQL query contains multiple `ALTER TABLE ... RENAME TO ...` commands, each will be executed in auto-commit mode in a separate transaction. From the perspective of an external process, the tables will be renamed sequentially, one after another. To rename multiple tables in a single transaction, use specialized methods available in CLI and SDK.

Renaming can be used to move a table from one directory within the database to another, for example:

```
ALTER TABLE `table1` RENAME TO `/backup/table1`;
```

Running forced table compaction

```
ALTER TABLE table_name COMPACT [WITH (key = value [, ...])];
```

> **Warning**
>
> Supported only for row-oriented tables.

YDB automatically performs compaction of tables and usually does not require manual intervention. However, using the `ALTER TABLE ... COMPACT` command, you can explicitly start compaction of data in a given table without waiting for automatic compaction. The command is useful, for example, in the following scenarios:

- Bulk data deletion was performed and you want to free up the occupied space faster.
- Compression settings or column groups were changed, and you want to apply them to existing data faster.

The command creates a background operation and synchronously waits for its completion. The wait can be interrupted, and the operation will continue running in the background. Background compaction operations can be managed via the CLI and YDB UI.

Parameters

The following parameters can be specified in the `WITH` block:

- `CASCADE` — whether to compact the table together with its secondary indexes. Valid values: `TRUE` and `FALSE`. Default value: `FALSE` (only the table itself is compacted).
- `PARALLEL` — the maximum number of partitions that can be compacted simultaneously within this operation. Valid values are positive integers. Default value: 1. The overall limit on concurrently running manual compactions for the entire database is set separately in the YDB cluster configuration.

Behavior

- You can monitor the progress of running compaction operations using the CLI command `ydb operation list compaction`. Using the corresponding commands, you can get the status of a specific operation, cancel an operation, or delete the record of a completed operation.
- You can also view and manage operations using the YDB UI. To do this, go to the database page, open the `Operations` tab, and select the operation type `Compaction`.
- If a compaction operation is already running for the same table (or for one of its secondary indexes when `CASCADE = TRUE`), the launch will fail with an error. You need to either wait for the previous operation to complete or cancel it.
- Running the command requires the same permissions as other `ALTER TABLE` actions. Concurrent table schema changes are not blocked.
- Forced compaction also materializes borrowed data. As a result, the total data volume in the database may increase. The borrowing mechanism is used, for example, to implement CoW table copying, so when compacting copied tables, their volume may eventually increase by the size of the original table.

Examples

Run compaction of only the main table:

```
ALTER TABLE series COMPACT;
```

Run compaction of the table together with all its secondary indexes:

```
ALTER TABLE series COMPACT WITH (CASCADE = TRUE);
```

Run compaction of only the impl table of a specific secondary index:

```
ALTER TABLE `series/idx_release/indexImplTable` COMPACT;
```

Run compaction of the table, setting the number of simultaneously processed partitions:

```
ALTER TABLE series COMPACT WITH (PARALLEL = 5);
```

ALTER TRANSFER

The `ALTER TRANSFER` statement modifies the parameters and state of a transfer instance.

Syntax

```
ALTER TRANSFER <name> [SET USING lambda | SET (option = value [, ...])]
```

where:

- `name` — the name of the transfer instance.
- `lambda` — the lambda-function for message transformation.
- `SET (option = value [, ...])` — the transfer parameters.

Parameters

- `STATE` — the transfer state. Possible values:
- `PAUSED` — pauses the transfer.
- `ACTIVE` — resumes a paused transfer.
- Table write batching parameters let you balance the latency of records appearing in the table against the resources required by the transfer. Batching parameters affect the processing of each topic partition independently. Change batching parameters with caution, as this can either improve or degrade message stream processing speed, and may even lead to a denial of service if the parameters are misconfigured. For example, writing to the table in small batches can overload the table and degrade its performance, while an excessively large batch size can cause the server to run out of available memory.
  - `BATCH_SIZE_BYTES` — the batch size in bytes. Default: 8 MB.
  - `FLUSH_INTERVAL` — the table write interval. Default: 60 seconds. Data is written to the table at this interval, even if the batch has not reached the size specified in the `BATCH_SIZE_BYTES` parameter.
- Authentication settings for the topic database (one of the following):
  - Using a token:
    - `TOKEN_SECRET_PATH` — the secret that contains the token.
  - Using a username and password:
    - `USER` — the username.
    - `PASSWORD_SECRET_PATH` — the secret that contains the password.
  - Using a service agent:
    - `SERVICE_ACCOUNT_ID` — the identificator of the service account.
    - `INITIAL_TOKEN_SECRET_PATH` — the secret that contains the account's token. It is used for initial authentication.

Permissions

Modifying a transfer requires the `ALTER SCHEMA` permissions.

Examples

The following query modifies the message transformation lambda-function:

```
$new_lambda = ($msg) -> {
    return [
        <|
            partition: $msg._partition,
            offset: $msg._offset,
            message: CAST($msg._data || ' altered' AS Utf8)
        |>
    ];
};

ALTER TRANSFER my_transfer SET USING $new_lambda;
```

The following query pauses the transfer:

```
ALTER TRANSFER my_transfer SET (STATE = "PAUSED");
```

The following query modifies the batching parameters:

```
ALTER TRANSFER my_transfer SET (
    BATCH_SIZE_BYTES = 1048576,
    FLUSH_INTERVAL = Interval('PT60S')
);
```

The following query changes the secret:

```
ALTER TRANSFER my_transfer SET (
    TOKEN_SECRET_PATH = "my_token"
);
```

Lambda function

A message transformation lambda function takes a single structured parameter containing the message from the topic and returns a list of structures corresponding to the table rows for insertion.

Example:

```
$lambda = ($msg) -> {
  return [
    <|
      column_1: $msg._create_timestamp,
      column_2: $msg._data
    |>
  ];
};
```

In this example:

- `$msg` — the message received from the topic.
- `column_1` and `column_2` — the names of the table columns.
- `$msg._create_timestamp` and `$msg._data` — the values that will be written to the table. The value types must match the table column types. For example, if the `column_2` table column has the `String` type, the type of `$msg._data` must also be `String`.

The following fields are available in a topic message:

Attribute	Value type	Description
`_create_timestamp`	`Timestamp`	Message creation time
`_data`	`String`	Message body
`_offset`	`Uint64`	Message offset
`_partition`	`Uint32`	Message's partition number
`_producer_id`	`String`	Producer ID
`_seq_no`	`Uint64`	Message sequence number
`_write_timestamp`	`Timestamp`	Message write time

Testing lambda functions

To test a lambda function during development, you can simulate a topic message by passing a structure with the same fields that the transfer will provide. Example:

```
$lambda = ($msg) -> {
  return [
    <|
      offset: $msg._offset,
      data: $msg._data
    |>
  ];
};

$msg = <|
  _data: "value",
  _offset: CAST(1 AS Uint64),
  _partition: CAST(2 AS Uint32),
  _producer_id: "producer",
  _seq_no: CAST(3 AS Uint64)
|>;

SELECT $lambda($msg);
```

If a lambda function contains complex transformation logic, you can extract it into a separate lambda function to simplify testing.

```
$extract_value = ($data) -> {
  -- complex transformations
  return $data;
};

$lambda = ($msg) -> {
  return [
    <|
      column: $extract_value($msg._data)
    |>
  ];
};

-- You can test the extract_value lambda function like this

SELECT $extract_value('converted value');
```

See Also

- CREATE TRANSFER
- DROP TRANSFER
- Data transfer

ALTER SECRET

The `ALTER SECRET` statement modifies an existing secret.

Syntax

```
ALTER SECRET secret_name
WITH (option = value[, ...])
```

- `secret_name` — the name of the secret to modify.
- `option` — command option:
  - `value` — string with the secret value.

Permissions

Modifying a secret requires the right `ALTER SCHEMA`.

Examples

Change the value of secret `secret_name` to `secret_value_new`:

```
ALTER SECRET secret_name WITH (value = "secret_value_new");
```

See also

- CREATE SECRET
- DROP SECRET

ALTER VIEW

`ALTER VIEW` changes the definition of a view.

> **Warning**
>
> This feature is not supported yet.

Instead, you can redefine a view by dropping it and recreating it with a different query or options:

```
DROP VIEW redefined_view;
CREATE VIEW redefined_view ...;
```

Please note that the two statements are executed separately, unlike a single `ALTER VIEW` statement. If a view is recreated in this way, it might be possible to observe the view in a deleted state for a brief moment.

See also

- CREATE VIEW
- DROP VIEW

ALTER TOPIC

You can use the `ALTER TOPIC` command to change the topic settings, as well as add, update, or delete its consumers.

Here is the general format of the `ALTER TOPIC` command:

```
ALTER TOPIC topic_path action1, action2, ..., actionN;
```

`action` is one of the alter actions described below.

Updating a set of consumers

`ADD CONSUMER`: Adds a consumer to a topic.

The following example will add a consumer with default settings to the topic.

```
ALTER TOPIC `my_topic` ADD CONSUMER new_consumer;
```

When adding consumers, you can specify their settings, for example:

```
ALTER TOPIC `my_topic` ADD CONSUMER new_consumer2 WITH (important = false);
```

Full list of available topic consumer settings

- `important`: Defines an important consumer. No data will be deleted from the topic until all the important consumers read them. Value type: `boolean`, default value: `false`.
- `read_from`: Sets up the message write time starting from which the consumer will receive data. Data written before this time will not be read. Value type: `Datetime` OR `Timestamp` OR `integer` (unix-timestamp in the numeric format). Default value: `0` (read from the earliest available message).
- `supported_codecs`: List of codecs supported by the consumer.

`DROP CONSUMER`: Deletes the consumer from the topic.

```
ALTER TOPIC `my_topic` DROP CONSUMER old_consumer;
```

Updating consumer settings

`ALTER CONSUMER`: Adds a consumer for a topic.

Here is the general syntax for `ALTER CONSUMER`:

```
ALTER TOPIC `topic_name` ALTER CONSUMER consumer_name consumer_action;
```

Supports the following types of `consumer_action`:

- `SET`: Sets consumer settings

The following example will assign the `important` parameter to the consumer.

```
ALTER TOPIC `my_topic` ALTER CONSUMER my_consumer SET (important = true);
```

You can specify several `ALTER CONSUMER` statements for a consumer. However, the settings applied by them shouldn't repeat.

This is a valid statement:

```
ALTER TOPIC `my_topic`
    ALTER CONSUMER my_consumer SET (important = true)
    ALTER CONSUMER my_consumer SET (read_from = 0);
```

But this statement will raise an error.

```
ALTER TOPIC `my_topic`
    ALTER CONSUMER my_consumer SET (important = true)
    ALTER CONSUMER my_consumer SET (important = false);
```

Updating topic settings

Using the `SET (option = value[, ...])` action, you can update your topic settings.

The example below will change the retention period for the topic and the writing quota per partition:

```
ALTER TOPIC `my_topic` SET (
    retention_period = Interval('PT36H'),
    partition_write_speed_bytes_per_second = 3000000
);
```

Full list of available topic settings

- `min_active_partitions`: Minimum number of topic partitions. During automatic load balancing, the number of active partitions will not decrease below this value. Value type: `integer`, default value: `1`.

- `partition_count_limit`: Maximum number of active partitions in the topic. `0` is interpreted as unlimited. Value type: `integer`, default value: `0`.
- `retention_period`: Data retention period in the topic. Value type: `Interval`.
- `retention_storage_mb`: Limit on the maximum disk space occupied by the topic data. When this value is exceeded, the older data is cleared, like under a retention policy. The consumed space may exceed the set value when autopartitioning is enabled. Value type: `integer`, default value: `0` (no limit).
- `partition_write_speed_bytes_per_second`: Maximum allowed write speed per partition. If a write speed for a given partition exceeds this value, the write speed will be capped. Value type: `integer`, default value: `2097152` (2MB).
- `partition_write_burst_bytes`: Write quota allocated for write bursts. When set to zero, the actual write_burst value is equalled to the quota value (this allows write bursts of up to one second). Value type: `integer`, default value: `0`.
- `metering_mode`: Resource metering mode (`RESERVED_CAPACITY` - based on the allocated resources or `REQUEST_UNITS` - based on actual usage). This option applies to topics in serverless databases. Value type: `String`.
- `supported_codecs`: List of codecs supported by the topic. Value type: `String`.

Change autopartitioning strategies for the topic

The following command sets the autopartitioning strategy to `UP`:

```
ALTER TOPIC `my_topic` SET (
    min_active_partitions = 1,
    max_active_partitions = 5,
    auto_partitioning_strategy = 'scale_up'
);
```

The following command pauses the topic autopartitioning:

```
ALTER TOPIC `my_topic` SET (
    auto_partitioning_strategy = 'paused'
);
```

The following command unpauses the topic autopartitioning:

```
ALTER TOPIC `my_topic` SET (
    auto_partitioning_strategy = 'scale_up'
);
```

ALTER STREAMING QUERY

`ALTER STREAMING QUERY` changes settings of streaming queries and controls their lifecycle (start and stop).

Syntax

```
ALTER STREAMING QUERY [IF EXISTS] <query_name> SET (
    <key1> = <value1>,
    <key2> = <value2>,
    ...
)
```

Parameters

- `IF EXISTS` — do not fail if the streaming query does not exist.
- `query_name` — name of the streaming query to change.
- `SET (<key> = <value>)` — optional list of settings to update.

Changing query settings

Syntax:

```
ALTER STREAMING QUERY [IF EXISTS] <query_name> SET (<key> = <value>)
```

Available parameters:

- `RUN = (TRUE|FALSE)` — start or stop the query.
- `RESOURCE_POOL = <resource_pool_name>` — name of the resource pool where the query runs.

When you run `SET (RUN = TRUE)`, read offsets from topics and aggregation state are restored from a checkpoint. If there is no checkpoint, reading starts from the latest data.

Examples of changing settings are below.

Permissions

Streaming queries require permission `ALTER SCHEMA`. Example grant for `my_streaming_query`:

```
GRANT ALTER SCHEMA ON my_streaming_query TO `user@domain`
```

Examples

Changing settings

Stop `my_streaming_query`:

```
ALTER STREAMING QUERY my_streaming_query SET (
    RUN = FALSE
)
```

Start `my_streaming_query` in resource pool `my_resource_pool`:

```
ALTER STREAMING QUERY my_streaming_query SET (
    RUN = TRUE,
    RESOURCE_POOL = my_resource_pool
)
```

Query status

Current status is available in the `Status` column of the ⟦C2⟧ system view System database views:

```
SELECT
    Path,
    Status,
    Text,
    Run
FROM
    `.sys/streaming_queries`
```

Possible status values:

1. `CREATING` — query is being created after `CREATE STREAMING QUERY`.
2. `CREATED` — query exists but is not running (for example, `RUN = FALSE`).
3. `STARTING` — query is starting.
4. `RUNNING` — query is executing.
5. `SUSPENDED` — query paused due to internal errors; the system will retry.
6. `STOPPING` — query is stopping after `ALTER STREAMING QUERY ... SET (RUN = FALSE)`.
7. `STOPPED` — query is stopped.

After successful DDL for create or alter, status is guaranteed to be `CREATED`, `STARTING`, `RUNNING`, `STOPPED`, or `SUSPENDED` depending on `RUN = (TRUE|FALSE)` and whether startup succeeded.

More examples for other data formats: Data formats for reading/writing from topics. For capabilities and limitations of streaming queries, see Streaming queries.

See also

- Streaming queries
- CREATE STREAMING QUERY
- DROP STREAMING QUERY

ALTER USER

Changes the user password.

Syntax

```
ALTER USER user_name [ WITH ] option [ ... ]
```

- `user_name`: The name of the user.
- `option` — The command option:
  - `PASSWORD 'password'` — changes the password to `password`.
  - `PASSWORD NULL` — sets an empty password.
  - `NOLOGIN` - disallows user login (user lockout).
  - `LOGIN` - allows user login (user unlocking).

ALTER RESOURCE POOL

`ALTER RESOURCE POOL` changes the definition of a resource pool.

Syntax

Modifying parameters

The syntax for modifying any resource pool parameter is as follows:

```
ALTER RESOURCE POOL <name> SET (<key> = <value>);
```

`<key>` is the parameter name, `<value>` is its new value.

For example, the following command enables a limit of 100 concurrent queries:

```
ALTER RESOURCE POOL olap SET (CONCURRENT_QUERY_LIMIT = "100");
```

Resetting parameters

The command to reset a resource pool parameter is as follows:

```
ALTER RESOURCE POOL <name> RESET (<key>);
```

`<key>` is the parameter name.

For example, the following command resets the `TOTAL_CPU_LIMIT_PERCENT_PER_NODE` settings for a resource pool:

```
ALTER RESOURCE POOL olap RESET (TOTAL_CPU_LIMIT_PERCENT_PER_NODE);
```

Permissions

A permission `ALTER SCHEMA` on the resource pool in the `.metadata/workload_manager/pools` directory is required. Example of granting such a permission:

```
GRANT 'ALTER SCHEMA' ON `.metadata/workload_manager/pools/olap_pool` TO `user1@domain`;
```

Parameters

- `CONCURRENT_QUERY_LIMIT` (Int32) — an optional field that sets the number of concurrently executing queries in the resource pool. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `QUEUE_SIZE` (Int32) — an optional field that defines the size of the waiting queue. The system can hold no more than queries at a time. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `DATABASE_LOAD_CPU_THRESHOLD` (Int32) — an optional field that sets the CPU load threshold for the entire database, after which queries are not sent for execution and remain in the queue. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `TOTAL_MEMORY_LIMIT_PERCENT_PER_NODE` (Double) — an optional field that defines the percentage of available memory on the node that all queries in this resource pool can use. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
  If a limit is set and the total memory consumption by queries in the pool reaches this limit:
  - New queries that require memory will fail with error `OVERLOADED`.
  - Already running queries that need additional memory will also fail with error `OVERLOADED`.
- `TOTAL_CPU_LIMIT_PERCENT_PER_NODE` (Double) — an optional field that sets the percentage of available CPU that all queries on the node in this resource pool can use. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `QUERY_CPU_LIMIT_PERCENT_PER_NODE` (Double) — an optional field that defines the percentage of available CPU on the node for a single query in the resource pool. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `RESOURCE_WEIGHT` (Int32) — an optional field that sets weights for distributing resources among pools. If the value is `-1`, weights are not used. Default value: `-1`. Valid values: .

> **Warning**
>
> The `QUERY_MEMORY_LIMIT_PERCENT_PER_NODE` parameter is currently not supported. To limit the amount of memory allocated to a resource pool, use the `TOTAL_MEMORY_LIMIT_PERCENT_PER_NODE` parameter.

See also

- Workload Manager — resource consumption management
- CREATE RESOURCE POOL
- DROP RESOURCE POOL

ALTER RESOURCE POOL CLASSIFIER

`ALTER RESOURCE POOL CLASSIFIER` changes the definition of a resource pool classifier.

Syntax

Changing parameters

The syntax for changing any resource pool classifier parameter is as follows:

```
ALTER RESOURCE POOL CLASSIFIER <name> SET (<key> = <value>);
```

`<key>` is the parameter name, `<value>` is its new value.

For example, the following command changes the user to which the rule applies:

```
ALTER RESOURCE POOL CLASSIFIER olap_classifier SET (MEMBER_NAME = "user2@domain");
```

Resetting parameters

The command to reset a resource pool classifier parameter is as follows:

```
ALTER RESOURCE POOL CLASSIFIER <name> RESET (<key>);
```

`<key>` is the parameter name.

For example, the following command resets the `MEMBER_NAME` setting:

```
ALTER RESOURCE POOL CLASSIFIER olap_classifier RESET (MEMBER_NAME);
```

Permissions

The `ALL` permission on the database is required. Example of granting it:

```
GRANT 'ALL' ON `/my_db` TO `user1@domain`;
```

Parameters

- `RANK` (Int64) — Optional field that defines the order in which resource pool classifiers are chosen. If omitted, the maximum existing `RANK` is taken and 1000 is added. Allowed values: a unique number in the range .
- `RESOURCE_POOL` (String) — Required field: name of the resource pool to which queries matching the classifier criteria are sent.
- `MEMBER_NAME` (String) — Optional field specifying which user or group is routed to the given resource pool. If omitted, the classifier ignores `MEMBER_NAME` and classification uses other criteria.

See also

- Workload Manager — resource consumption management
- CREATE RESOURCE POOL CLASSIFIER
- DROP RESOURCE POOL CLASSIFIER

ANALYZE

`ANALYZE` forces the collection of statistics for YDB cost-based optimizer.

Syntax

```
ANALYZE <path_to_table> [ (<column_name> [, ...]) ]
```

This command forces the synchronous collection of table statistics and column statistics for the specified columns or for all columns if none are specified. `ANALYZE` returns once all the requested statistics have been collected and are up to date.

- `path_to_table` — the path to the table for which statistics should be collected.
- `column_name` — collect column statistics only for the specified columns of the table.

The current set of statistics is described in Statistics for the Cost-Based Optimizer.

Background operation

SQL operation `ANALYZE` executes synchronously and waits for completion. In addition a background operation is created for observability. Use the operation list command to monitor the progress of background `ANALYZE` operations.

BATCH DELETE FROM

> **Tip**
>
> Before diving into `BATCH DELETE FROM`, it is recommended to familiarize yourself with the standard DELETE FROM.

`BATCH DELETE FROM` allows to delete records in large tables while minimizing the risk of lock invalidation and transaction rollback by weakening guarantees. Specifically, data deletion is performed as a series of transactions for each partition of the specified table separately, processing 10 000 rows per iteration. Each query processes up to 10 partitions concurrently.

This query, like the standard `DELETE FROM`, executes synchronously and returns a status. If an error occurs or the client disconnects, data deletion stops, and applied changes are not rolled back.

The semantics are inherited from the standard `DELETE FROM` with the following restrictions:

- Supported only for row-oriented tables.
- Supported only for queries with implicit transaction control.
- The use of subqueries and multiple statements in a single query is prohibited.
- The `RETURNING` clause is unavailable.

Example

```
BATCH DELETE FROM my_table
WHERE Key1 > 1 AND Key2 >= "One";
```

BATCH UPDATE

> **Tip**
>
> Before diving into `BATCH UPDATE`, it is recommended to familiarize yourself with the standard UPDATE.

`BATCH UPDATE` allows to update records in large tables while minimizing the risk of lock invalidation and transaction rollback by weakening guarantees. Specifically, data updates are performed as a series of transactions for each partition of the specified table separately, processing 10 000 rows per iteration. Each query processes up to 10 partitions concurrently.

This query, like the standard `UPDATE`, executes synchronously and returns a status. If an error occurs or the client disconnects, the data update stops, and the applied changes are not rolled back.

The semantics are inherited from the standard `UPDATE` with the following restrictions:

- Supported only for row-oriented tables.
- Supported only for queries with implicit transaction control.
- Only idempotent updates are supported: expressions following `SET` should not depend on the current values of the columns being modified.
- The use of subqueries and multiple statements in a single query is prohibited.
- The `RETURNING` clause is unavailable.

Example

```
BATCH UPDATE my_table
SET Value1 = "foo", Value2 = 0
WHERE Key1 > 1;
```

BACKUP

The `BACKUP` statement creates a backup of tables in a backup collection.

```
BACKUP collection_name [INCREMENTAL];
```

Parameters

- `collection_name` — name of the backup collection.
- `INCREMENTAL` — create an incremental backup instead of a full backup.

Backup types

Full backup

A full backup creates a snapshot of all tables in the collection at a specific point in time. This serves as the baseline for subsequent incremental backups.

```
BACKUP production_backups;
```

Incremental backup

An incremental backup captures only the changes (inserts, updates, deletes) since the previous backup in the chain. The collection must have been created with `INCREMENTAL_BACKUP_ENABLED = 'true'`.

```
BACKUP production_backups INCREMENTAL;
```

> **Warning**
>
> Incremental backups require a previous full backup in the same collection. Always create a full backup first before taking incremental backups.

Examples

Creating an initial full backup:

```
-- First, create a collection
CREATE BACKUP COLLECTION daily_backups (
    TABLE orders
) WITH (
    STORAGE = 'cluster',
    INCREMENTAL_BACKUP_ENABLED = 'true'
);

-- Then create the full backup
BACKUP daily_backups;
```

Creating incremental backups:

```
-- After the initial full backup, create incremental backups
BACKUP daily_backups INCREMENTAL;
```

Monitoring backup operations

Backup operations run asynchronously in the background. You can monitor their progress using YDB CLI:

```
# List backup operations
ydb operation list incbackup

# Get operation details
ydb operation get <operation-id>
```

See also

- Backup collections.
- CREATE BACKUP COLLECTION.
- RESTORE.
- DROP BACKUP COLLECTION.
- Backup and recovery guide.

CREATE ASYNC REPLICATION

The `CREATE ASYNC REPLICATION` statement creates an asynchronous replication instance.

Syntax

```
CREATE ASYNC REPLICATION <name>
FOR <remote_path> AS <local_path> [, <another_remote_path> AS <another_local_path>]
WITH (option = value [, ...])
```

Parameters

- `name` — a name of the asynchronous replication instance.
- `remote_path` — a relative or absolute path to a table or directory in the source database.
- `local_path` — a relative or absolute path to a target table or directory in the local database.
- `WITH (option = value [, ...])` — asynchronous replication parameters:
  - `CONNECTION_STRING` — a connection string for the source database (mandatory).
  - `CA_CERT` — a root certificate for TLS. Optional parameter. Can be specified if the source database supports an encrypted data interchange protocol (`CONNECTION_STRING` starts with `grpcs://`).
  - Authentication details for the source database (mandatory) depending on the authentication method:
    - Access token:
      - `TOKEN_SECRET_PATH` — the secret that contains the token.
    - Login and password:
      - `USER` — a database user name.
      - `PASSWORD_SECRET_PATH` — the secret that contains the password for the source database user.
    - Delegated service account:
      - `SERVICE_ACCOUNT_ID` — identifier of the service account.
      - `INITIAL_TOKEN_SECRET_PATH` — the secret that contains the token for the service account. Used for initialization.
- `CONSISTENCY_LEVEL` — consistency level of replicated data:
  - `ROW` — row-level data consistency. Default mode.
  - `GLOBAL` — global data consistency. Additionally can be specified:
    - `COMMIT_INTERVAL` — change commit interval in ISO 8601 format. The default value is 10 seconds.

Examples

> **Tip**
>
> Before creating an asynchronous replication instance, you must create a secret with authentication credentials for the source database or ensure that you have access to an existing secret.

The following statement creates an asynchronous replication instance to synchronize the `original_table` source table in the `/Root/another_database` database to the `replica_table` target table in the local database:

```
CREATE ASYNC REPLICATION my_replication_for_single_table
FOR original_table AS replica_table
WITH (
    CONNECTION_STRING = 'grpcs://example.com:2135/?database=/Root/another_database',
    TOKEN_SECRET_PATH = 'my_secret'
);
```

The statement above uses the token from the `my_secret` secret for authentication and the `grpcs://example.com:2135` endpoint to connect to the `/Root/another_database` database.

The following statement creates an asynchronous replication instance to replicate the source tables `original_table_1` and `original_table_2` to the target tables `replica_table_1` and `replica_table_2`:

```
CREATE ASYNC REPLICATION my_replication_for_multiple_tables
FOR original_table_1 AS replica_table_1, original_table_2 AS replica_table_2
WITH (
    CONNECTION_STRING = 'grpcs://example.com:2135/?database=/Root/another_database',
    TOKEN_SECRET_PATH = 'my_secret'
);
```

The following statement creates an asynchronous replication instance for the objects in the `original_dir` directory:

```
CREATE ASYNC REPLICATION my_replication_for_dir
FOR original_dir AS replica_dir
WITH (
    CONNECTION_STRING = 'grpcs://example.com:2135/?database=/Root/another_database',
    TOKEN_SECRET_PATH = 'my_secret'
);
```

The following statement creates an asynchronous replication instance for the objects in the `/Root/another_database` database:

```
CREATE ASYNC REPLICATION my_replication_for_database
FOR `/Root/another_database` AS `/Root/my_database`
WITH (
    CONNECTION_STRING = 'grpcs://example.com:2135/?database=/Root/another_database',
```

```
    TOKEN_SECRET_PATH = 'my_secret'
);
```

The following statement creates an asynchronous replication instance with a TLS root certificate specified:

```
CREATE ASYNC REPLICATION my_consistent_replication
FOR original_table AS replica_table
WITH (
    CONNECTION_STRING = 'grpcs://example.com:2135/?database=/Root/another_database',
    TOKEN_SECRET_PATH = 'my_secret',
    CA_CERT = '-----BEGIN CERTIFICATE-----...'
);
```

The following statement creates an asynchronous replication instance in global data consistency mode (default change commit interval is 10 seconds):

```
CREATE ASYNC REPLICATION my_consistent_replication
FOR original_table AS replica_table
WITH (
    CONNECTION_STRING = 'grpcs://example.com:2135/?database=/Root/another_database',
    TOKEN_SECRET_PATH = 'my_secret',
    CONSISTENCY_LEVEL = 'GLOBAL'
);
```

The following statement creates an asynchronous replication instance in global data consistency mode with a one-minute change commit interval:

```
CREATE ASYNC REPLICATION my_consistent_replication_1min_commit_interval
FOR original_table AS replica_table
WITH (
    CONNECTION_STRING = 'grpcs://example.com:2135/?database=/Root/another_database',
    TOKEN_SECRET_PATH = 'my_secret',
    CONSISTENCY_LEVEL = 'GLOBAL',
    COMMIT_INTERVAL = Interval('PT1M')
);
```

See also

- ALTER ASYNC REPLICATION
- DROP ASYNC REPLICATION

CREATE BACKUP COLLECTION

The `CREATE BACKUP COLLECTION` statement creates a backup collection.

```
CREATE BACKUP COLLECTION collection_name (
    TABLE table_name [, TABLE another_table_name ...]
) WITH (option = value [, ...]);
```

Parameters

- `collection_name` — name of the backup collection to create.
- `table_name` — full path to a table to include in the collection. Multiple tables can be specified.
- Options:
  - `STORAGE` — storage backend for backups. Supported values:
    - `'cluster'` — storage within the YDB cluster.
  - `INCREMENTAL_BACKUP_ENABLED` — enable incremental backup support. Set to `'true'` to enable incremental backups, `'false'` for full backups only.

> **Note**
>
> When choosing a name for the backup collection, please consider the common schema objects naming rules

Examples

Creating a backup collection with a single table:

```
CREATE BACKUP COLLECTION daily_backups (
    TABLE orders
) WITH (
    STORAGE = 'cluster',
    INCREMENTAL_BACKUP_ENABLED = 'true'
);
```

Creating a backup collection with multiple tables:

```
CREATE BACKUP COLLECTION production_backups (
    TABLE orders,
    TABLE products,
    TABLE customers
) WITH (
    STORAGE = 'cluster',
    INCREMENTAL_BACKUP_ENABLED = 'true'
);
```

See also

- Backup collections.
- BACKUP.
- RESTORE.
- DROP BACKUP COLLECTION.

CREATE GROUP

Creates a group with the specified name. Optionally, you can specify a list of users to add to the group.

Syntax

```
CREATE GROUP group_name [ WITH USER user_name [ , user_name [ ... ]] [ , ] ]
```

Parameters

- `group_name`: The name of the group. It may contain lowercase Latin letters and digits.
- `user_name`: The name of the user who will become a member of the group after its creation. It may contain lowercase Latin letters and digits.

Examples

```
CREATE GROUP group1;
```

```
CREATE GROUP group2 WITH USER user1;
```

```
CREATE GROUP group3 WITH USER user1, user2,;
```

```
CREATE GROUP group4 WITH USER user1, user3, user2;
```

CREATE EXTERNAL DATA SOURCE

The `CREATE EXTERNAL DATA SOURCE` statement creates an external data source.

```
CREATE EXTERNAL DATA SOURCE external_data_source WITH (
  SOURCE_TYPE="source_type",
  LOCATION="ip_address_or_fqdn:port",
  USE_TLS="use_tls",
  AUTH_METHOD="auth_method",
  LOGIN="login",
  PASSWORD_SECRET_PATH="password_secret_path"
)
```

Where:

- `external_data_source` — name of the external data source.
- `source_type` — type of the external data source. Possible values: `ClickHouse`, `PostgreSQL`, `ObjectStorage`.
- `ip_address_or_fqdn:port` — full network address of the external data source, including the port. The network address can be an IP address or FQDN.
- `use_tls` — flag indicating that the connection must use a secure (TLS) connection. Possible values: `TRUE`, `FALSE`.
- `auth_method` — authentication method for the external data source. For external sources of types `ClickHouse` and `PostgreSQL`, only the `BASIC` authentication type is supported. For the `ObjectStorage` external source, only the `NONE` authentication type is currently supported.
- `login` — login used to connect to the external data source.
- `password_secret_path` — secret containing the password for connecting to the external data source.

When using secure TLS channels, system certificates located on YDB servers are used.

Example

The query below creates an external data source named `TestDataSource` to a ClickHouse cluster with IP address `192.168.1.1` and port `8443`, with login `admin` and secret `test_secret`:

```
CREATE EXTERNAL DATA SOURCE TestDataSource WITH (
  SOURCE_TYPE="ClickHouse",
  LOCATION="192.168.1.1:8443",
  USE_TLS="TRUE",
  AUTH_METHOD="BASIC",
  LOGIN="admin",
  PASSWORD_SECRET_PATH="test_secret"
)
```

Connecting to ClickHouse

To create a connection to a ClickHouse cluster, create an external data source `EXTERNAL DATA SOURCE` and specify:

- in the `SOURCE_TYPE` field, the value `ClickHouse`;
- in the `LOCATION` field, the full network address of the ClickHouse cluster, including the port. The network address can be an IP address or FQDN. Connection to the ClickHouse cluster is currently always made over the HTTP protocol;
- in the `USE_TLS` field, the flag indicating that the connection to the ClickHouse cluster must use a secure (TLS) connection;
- in the `AUTH_METHOD` field, the value `BASIC`;
- in the `LOGIN` field, the login used to connect to the ClickHouse cluster;
- in the `PASSWORD_SECRET_PATH` field, the secret containing the password for connecting to the ClickHouse cluster.

When using secure TLS channels, system certificates located on YDB servers are used.

Example

The query below creates an external data source named `TestDataSource` to a ClickHouse cluster with IP address `192.168.1.1` and port `8443`, with login `admin` and secret `test_secret`:

```
CREATE EXTERNAL DATA SOURCE TestDataSource WITH (
  SOURCE_TYPE="ClickHouse",
  LOCATION="192.168.1.1:8443",
  USE_TLS="TRUE",
  AUTH_METHOD="BASIC",
  LOGIN="admin",
  PASSWORD_SECRET_PATH="test_secret"
)
```

Connecting to PostgreSQL

To create a connection to a PostgreSQL cluster, create an `EXTERNAL DATA SOURCE` object and specify in the fields:

- in the `SOURCE_TYPE` field, the value `PostgreSQL`;
- in the `LOCATION` field, the full network address of the PostgreSQL cluster, including the port. The network address can be an IP address or FQDN;
- in the `USE_TLS` field, the flag indicating that the connection to the PostgreSQL cluster must use a secure (TLS) connection;
- in the `AUTH_METHOD` field, the value `BASIC`;
- in the `LOGIN` field, the login used to connect to the PostgreSQL cluster;
- in the `PASSWORD_SECRET_PATH` field, the secret containing the password for connecting to the PostgreSQL cluster.

Connection to the PostgreSQL cluster is currently always made over the standard Frontend/Backend Protocol over TCP. When using secure TLS channels, system certificates located on YDB servers are used.

Example

The query below creates an external data source named `TestDataSource` to a PostgreSQL cluster with IP address `192.168.1.1` and port `5432`, with login `admin` and secret `test_secret`:

```
CREATE EXTERNAL DATA SOURCE TestDataSource WITH (
  SOURCE_TYPE="PostgreSQL",
  LOCATION="192.168.1.1:5432",
  USE_TLS="TRUE",
  AUTH_METHOD="BASIC",
  LOGIN="admin",
  PASSWORD_SECRET_PATH="test_secret"
)
```

Connecting to S3 (Object Storage)

To create an external data source pointing to a bucket with data in S3 (Object Storage), create an `EXTERNAL DATA SOURCE` object and specify in the fields:

- in the `SOURCE_TYPE` field, the value `ObjectStorage`;
- in the `LOCATION` field, the network path to the bucket;
- in the `AUTH_METHOD` field, the value `NONE`.

> **Note**
>
> Currently, only buckets that are not protected by authentication are supported.

Connection is possible to any data sources that support the AWS S3 access protocol. Any URLs to systems supporting this protocol can be specified.

Typical `LOCATION` field values when connecting to different systems for bucket `bucket`:

System name	URL
Object Storage	`https://storage.yandexcloud.net/bucket/`
AWS S3	`http://s3.amazonaws.com/bucket/`

Example

The query below creates an external data source named `TestDataSource` pointing to directory `folder` in bucket `bucket` in Object Storage:

```
CREATE EXTERNAL DATA SOURCE TestDataSource WITH (
  SOURCE_TYPE="ObjectStorage",
  LOCATION="http://s3.amazonaws.com/bucket/folder/",
  AUTH_METHOD="NONE"
)
```

CREATE STREAMING QUERY

`CREATE STREAMING QUERY` creates a streaming query.

Syntax

```
CREATE [OR REPLACE] STREAMING QUERY [IF NOT EXISTS] <query_name> [WITH (
    <key1> = <value1>,
    <key2> = <value2>,
    ...
)] AS
DO BEGIN
    <query_statement1>;
    <query_statement2>;
    ...
END DO
```

Parameters

- `OR REPLACE` — if a streaming query with this name already exists, it will be replaced with a new query while preserving the read offsets from the topic.
- `IF NOT EXISTS` — do not output an error if a streaming query with this name already exists; in this case, the existing query will remain unchanged.
- `query_name` — the name of the streaming query to create.
- `WITH (<key> = <value>)` — a list of settings for the new streaming query, optional.
- `AS DO BEGIN ... END DO` — the full text of the new streaming query, including all required SQL statements. Restrictions on the query text are given in Limitations, see below for query text examples.

The `OR REPLACE` and `IF NOT EXISTS` settings cannot be used simultaneously.

Available parameters of the `WITH` block:

- `RUN = (TRUE|FALSE)` — start the query after creation, default `TRUE`.
- `RESOURCE_POOL = <resource_pool_name>` — the name of the resource pool in which the query will run.

See below for examples of creating a streaming query.

Using a reader

A consumer is a named subscription to a topic that stores the current read position.

A consumer is created via the CLI or when creating a topic using CREATE TOPIC. The consumer name is specified in the query text with a pragma:

```
PRAGMA pq.Consumer="my_consumer";
```

If a consumer is not specified, reading from the topic is performed without one. Specifying a consumer allows tracking the read position and lag from the topic side, for example via the CLI.

Regardless of whether a reader exists, the read position of the streaming query is saved in a checkpoint.

Examples

Writing to a topic (JSON)

The query reads events from an input topic, forms a JSON object from individual fields, and writes the result to an output topic.

The `AsStruct` function creates a structure from the specified fields, `Yson::From` converts it to Yson, `Yson::SerializeJson` serializes it to a JSON string, and `ToBytes` converts it to the `String` type, which is required for writing to a topic.

> **Note**
>
> Streaming queries can work with local and external topics.
>
> In the example:
>
> - `ext_source` is a pre-created `external data source`.
> - `input_topic` and `output_topic` are local or external topics.

```
CREATE STREAMING QUERY my_streaming_query AS
DO BEGIN

    INSERT INTO output_topic -- or external topic ext_source.output_topic
    SELECT
        -- Formation of JSON from individual fields
        ToBytes(Unwrap(Yson::SerializeJson(Yson::From(
            AsStruct(Id AS id, Name AS name)
        ))))
    FROM
        ext_source.input_topic -- or local topic input_topic
    WITH (
        FORMAT = json_each_row,  -- Input data format
        SCHEMA = (               -- Input data schema
            Id Uint64 NOT NULL,
            Name Utf8 NOT NULL
        )
    );
```

```
END DO
```

Writing to a table

The query reads events from the topic and writes them to the `output_table` table. The table must be created in advance with a schema matching the selected columns.

> **Warning**
>
> Writing to tables in streaming queries is supported **only in UPSERT mode**. The `INSERT INTO` operation is not supported, because during reprocessing of events (the at-least-once guarantee) it would lead to duplicate rows. With `UPSERT`, if a row with such a primary key already exists, it will be updated; otherwise, a new row will be inserted, and `INSERT INTO` will fail with an error.

```
CREATE STREAMING QUERY my_streaming_query AS
DO BEGIN

    -- Write to table (only UPSERT, INSERT is not supported)
    UPSERT INTO output_table
    SELECT
        Id,
        Name
    FROM
        input_topic -- or external topic ext_source.input_topic
    WITH (
        FORMAT = json_each_row,  -- Input data format
        SCHEMA = (               -- Input data schema
            Id Uint64 NOT NULL,
            Name Utf8 NOT NULL
        )
    );

END DO
```

Running in a resource pool

The query is created in the specified resource pool but is not started automatically (`RUN = FALSE`). This allows you to check the configuration before starting or start the query later via ALTER STREAMING QUERY.

```
CREATE STREAMING QUERY my_streaming_query WITH (
    RUN = FALSE,                     -- Do not start automatically
    RESOURCE_POOL = my_resource_pool  -- Resource pool for execution
) AS
DO BEGIN

    INSERT INTO output_topic -- or external topic ext_source.output_topic
    SELECT
        ToBytes(Unwrap(Yson::SerializeJson(Yson::From(
            AsStruct(Id AS id, Name AS name)
        ))))
    FROM
        ext_source.input_topic -- or local topic input_topic
    WITH (
        FORMAT = json_each_row,
        SCHEMA = (
            Id Uint64 NOT NULL,
            Name Utf8 NOT NULL
        )
    );

END DO
```

Other examples: Typical streaming query patterns.

See also

- Typical streaming query patterns
- Streaming queries
- ALTER STREAMING QUERY
- DROP STREAMING QUERY

CREATE OBJECT (TYPE SECRET)

> **Alert**
>
> **This command is deprecated** and will be removed in future versions of YDB. The recommended syntax for working with secrets is described in the Secrets section.

The following SQL statement creates a secret:

```
CREATE OBJECT <secret_name> (TYPE SECRET) WITH value = "<secret_value>";
```

Where:

- `secret_name` - the name of the secret.
- `secret_value` - the contents of the secret.

Example

The following statement creates a secret named `MySecretName` with `MySecretData` as a value.

```
CREATE OBJECT MySecretName (TYPE SECRET) WITH value = "MySecretData";
```

CREATE OBJECT (TYPE SECRET_ACCESS)

Alert

This command is deprecated and will be removed in future versions of YDB. The recommended syntax for working with secrets is described in the Secrets section.

All rights to use a secret belong to the secret's creator. The creator can grant another user read access to the secret through secret access management.

Access to secrets is managed using special `SECRET_ACCESS` objects. To grant permission to use the secret `secret_name` to the user `user_name`, create a `SECRET_ACCESS` object named `secret_name:user_name`.

```
CREATE OBJECT `secret_name:user_name` (TYPE SECRET_ACCESS);
```

Where:

- `secret_name` — the name of the secret.
- `user_name` — the name of the user who receives access.

Example

The following statement grants access to the secret `MySecretName` to the user `another_user`:

```
CREATE OBJECT `MySecretName:another_user` (TYPE SECRET_ACCESS);
```

CREATE EXTERNAL TABLE

The `CREATE EXTERNAL TABLE` statement creates an external table with the given schema.

```
CREATE EXTERNAL TABLE table_name (
  column1 type1,
  column2 type2 NOT NULL,
  ...
  columnN typeN NULL
) WITH (
  DATA_SOURCE="data_source_name",
  LOCATION="path",
  FORMAT="format_name",
  COMPRESSION="compression_name"
);
```

Where:

- `column1 type1`, `columnN typeN NULL` — column definitions and types;
- `data_source_name` — name of the connection to S3 (Object Storage).
- `path` — path inside the data bucket. The path must refer to an existing folder in the bucket.
- `format_name` — one of the supported storage formats.
- `compression_name` — one of the supported compression algorithms.

Only a limited subset of data types is allowed:

- `Bool`.
- `Int8`, `Uint8`, `Int16`, `Uint16`, `Int32`, `Uint32`, `Int64`, `Uint64`.
- `Float`, `Double`.
- `Date`, `DateTime`.
- `String`, `Utf8`.

Without extra modifiers, a column has an optional type and may contain `NULL`. Use `NOT NULL` for a non-optional column.

Example

The following creates an external table `s3_test_data` with string columns `key` and `value` in CSV format under `test_folder` in the bucket, using external data source `bucket`:

```
CREATE EXTERNAL TABLE s3_test_data (
  key Utf8 NOT NULL,
  value Utf8 NOT NULL
) WITH (
  DATA_SOURCE="bucket",
  LOCATION="folder",
  FORMAT="csv_with_names",
  COMPRESSION="gzip"
);
```

CREATE OBJECT (TYPE SECRET)

> **Alert**
>
> **This command is deprecated** and will be removed in future versions of YDB. The recommended syntax for working with secrets is described in the Secrets section.

The following SQL statement creates a secret:

```
CREATE OBJECT <secret_name> (TYPE SECRET) WITH value = "<secret_value>";
```

Where:

- `secret_name` - the name of the secret.
- `secret_value` - the contents of the secret.

Example

The following statement creates a secret named `MySecretName` with `MySecretData` as a value.

```
CREATE OBJECT MySecretName (TYPE SECRET) WITH value = "MySecretData";
```

CREATE RESOURCE POOL

`CREATE RESOURCE POOL` creates a resource pool.

Syntax

```
CREATE RESOURCE POOL <name>
WITH ( <parameter_name> [= <parameter_value>] [, ... ] )
```

- `name` is the name of the resource pool being created. It must be unique. Path notation is not allowed (i.e., it must not contain `/`).
- `WITH ( <parameter_name> [= <parameter_value>] [, ... ] )` allows setting parameter values that define the behavior of the resource pool.

Parameters

- `CONCURRENT_QUERY_LIMIT` (Int32) — an optional field that sets the number of concurrently executing queries in the resource pool. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `QUEUE_SIZE` (Int32) — an optional field that defines the size of the waiting queue. The system can hold no more than queries at a time. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `DATABASE_LOAD_CPU_THRESHOLD` (Int32) — an optional field that sets the CPU load threshold for the entire database, after which queries are not sent for execution and remain in the queue. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `TOTAL_MEMORY_LIMIT_PERCENT_PER_NODE` (Double) is an optional field that defines the percentage of available memory on the node that all queries in this resource pool can use. If the value is `-1`, there are no limits. Default value: `-1`. Valid values: .

  If a limit is set and the total memory consumption by queries in the pool reaches this limit:

  - New queries that require memory will fail with error `OVERLOADED`.
  - Already running queries that need additional memory will also fail with error `OVERLOADED`.
- `TOTAL_CPU_LIMIT_PERCENT_PER_NODE` (Double) — an optional field that sets the percentage of available CPU that all queries on the node in this resource pool can use. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `QUERY_CPU_LIMIT_PERCENT_PER_NODE` (Double) — an optional field that defines the percentage of available CPU on the node for a single query in the resource pool. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `RESOURCE_WEIGHT` (Int32) — an optional field that sets weights for distributing resources among pools. If the value is `-1`, weights are not used. Default value: `-1`. Valid values: .

> **Warning**
>
> The `QUERY_MEMORY_LIMIT_PERCENT_PER_NODE` parameter is currently not supported. To limit the amount of memory allocated to a resource pool, use the `TOTAL_MEMORY_LIMIT_PERCENT_PER_NODE` parameter.

Notes

Queries are always executed in some resource pool. By default, all queries are sent to the resource pool `default`, which is created automatically and cannot be deleted — it is always present in the system.

If the value of parameter `CONCURRENT_QUERY_LIMIT` is set to 0, then all queries sent to this pool will be immediately terminated with status `PRECONDITION_FAILED`.

Permissions

Requires permission `CREATE TABLE` on the directory `.metadata/workload_manager/pools`, an example of granting such permission:

```
GRANT 'CREATE TABLE' ON `.metadata/workload_manager/pools` TO `user1@domain`;
```

```
CREATE RESOURCE POOL olap WITH (
    CONCURRENT_QUERY_LIMIT=20,
    QUEUE_SIZE=1000,
    DATABASE_LOAD_CPU_THRESHOLD=80,
    RESOURCE_WEIGHT=100,
    TOTAL_MEMORY_LIMIT_PERCENT_PER_NODE=80,
    TOTAL_CPU_LIMIT_PERCENT_PER_NODE=70
)
```

In the example above, a resource pool is created with the following limits:

- Maximum number of concurrent queries — 20.
- Maximum queue size — 1000.
- When the database load reaches 80%, queries stop running concurrently.
- All queries in the pool can consume no more than 80% of the available memory on the node.
- The total limit on available CPU for all queries in the pool on the node is 70%.
- The resource pool has a weight of 100, which only takes effect in case of oversubscription.

See also

- Workload Manager — resource consumption management
- ALTER RESOURCE POOL
- DROP RESOURCE POOL

CREATE RESOURCE POOL CLASSIFIER

`CREATE RESOURCE POOL CLASSIFIER` creates a resource pool classifier.

Syntax

```
CREATE RESOURCE POOL CLASSIFIER <name>
WITH ( <parameter_name> [= <parameter_value>] [, ... ] )
```

- `name` тАФ the name of the resource pool classifier being created. It must be unique. The name must not contain characters prohibited for schema objects.
- `WITH ( <parameter_name> [= <parameter_value>] [, ... ] )` тАФ allows you to set parameter values that define the behavior of the resource pool classifier.

Common parameters

- `RANK` (Int64) тАФ an optional field that specifies the selection order of the resource pool classifier. If the value is not specified, the maximum existing `RANK` is taken and 1000 is added to it. Valid values: a unique number in the range .
- `RESOURCE_POOL` (String) тАФ the name of the resource pool to which requests matching the classifier's predicates are directed.
- `ACTION` (Enum) тАФ the action applied to a request when the classifier matches. Allowed value тАФ `reject`: the request is rejected, and the user receives an error of the form `Request is rejected by classifier '<name>' (rank=<rank>)`.

> **Note**
>
> The `RESOURCE_POOL` and `ACTION` parameters are mutually exclusive: exactly one of them must be specified in a classifier.

Predicate parameters

A predicate is a condition checked for an incoming request. A classifier matches if **all** of its predicates are satisfied (logical **AND**). To express **OR** logic, create multiple classifiers with different `RANK` values. Classifiers are processed in ascending order of `RANK`; processing stops at the first match тАФ the request has that classifier's `ACTION` applied or is directed to its `RESOURCE_POOL`.

Predicate parameters are optional. A classifier without any predicates matches any request тАФ this is useful for a "catch-all" classifier with the maximum `RANK`, for example, to direct all unclassified traffic to a specific pool (`RESOURCE_POOL`) or to reject it (`ACTION='reject'`).

> **Warning**
>
> Be careful when creating a "catch-all" classifier: all classifiers following it by `RANK` will never fire тАФ processing stops at the first match, and the "catch-all" always matches.

List of predicates:

- `MEMBER_NAME` (String) тАФ the SID of the user or group on whose behalf the request was made. See below for details.
- `HAS_PATH` (String) тАФ path to a YDB object accessed by the request; supports wildcards `*` and `?`. See below for details.
- `HAS_APP_NAME` (String) тАФ client application identifier. See below for details.
- `HAS_FULL_SCAN` (String) тАФ path to an object for which a full scan is expected; supports wildcards `*` and `?`. See below for details.
- `HAS_STREAM` (Bool) тАФ indicates whether the request is streaming. See below for details.

MEMBER_NAME

`MEMBER_NAME` is compared character by character with the user's SID or any group SID from their authentication token. The SID format depends on how the user logged into the system.

- **Built-in YDB users (login/password)** тАФ the SID matches the username, without a suffix. For example, `user1`. For more information, see Authenticating by username and password.
- **Cloud users (Access Service)** тАФ the SID has the form `<subject_id>@as`, where `<subject_id>` is the user ID in IAM. The suffix is set by the `access_service_domain` parameter (default `as`). For example, `ajeb89hv69nujke769fa@as`. For more information, see Authentication through a third-party IAM provider.
- **LDAP** тАФ the SID has the form `<login>@<domain>`, where the domain is set by the `ldap_authentication_domain` parameter (default `ldap`). For example, `user1@ldap`. For more information, see LDAP directory integration.
- **External identity providers (OIDC)** тАФ the SID has the form `<login>@<domain>`, where the domain is set by the `external_idp_authentication_domain` parameter in the authentication configuration (default `sso`). For example, `user1@sso`.

You can specify either the SID of a specific user or the SID of a group. The `all-users@well-known` group is automatically added to all authenticated users тАФ it is convenient when you need to direct queries from all authenticated clients to a pool.

Example. Direct requests from user `user1@ldap` to the `olap` pool:

```
CREATE RESOURCE POOL CLASSIFIER cl_user WITH (
    RANK=100,
    RESOURCE_POOL='olap',
    MEMBER_NAME='user1@ldap'
);
```

HAS_PATH

`HAS_PATH` compares the paths of YDB objects accessed by the request against the specified mask. The mask supports wildcards: `*` тАФ any sequence of characters, `?` тАФ any single character. The predicate matches if at least one object in the request plan matches the mask.

Example. Direct requests to archive tables to the `pool_archive` pool:

```
CREATE RESOURCE POOL CLASSIFIER cl_archive WITH (
    RANK=700,
    RESOURCE_POOL='pool_archive',
    HAS_PATH='/Root/db/archive/*'
);
```

HAS_APP_NAME

`HAS_APP_NAME` compares the value against the client application identifier. The value is passed by the client in gRPC request metadata via the `x-ydb-application-name` header; YDB SDKs provide a way to set it on the client side. Comparison is exact match (no wildcards).

> **Warning**
>
> The `HAS_APP_NAME` value is set by the client and is not authenticated by the server ТАФ do not use it as an access control mechanism. For effective isolation, combine it with `MEMBER_NAME` or direct unrecognized requests to a sandbox pool with strict limits.

Setting the application identifier on the client:

- **YDB Embedded UI** ТАФ fixed value `ydb-ui`, set by the viewer and not user-configurable.
- **YDB CLI** ТАФ not supported: the client application identifier is not sent in requests.
- **YDB C++ SDK** ТАФ per request via the `Header` parameter of `TRequestSettings`: `settings.Header({{ NYdb::YDB_APPLICATION_NAME, "my-app" }})`, where the `YDB_APPLICATION_NAME` constant equals `x-ydb-application-name`.
- **YDB Go SDK** ТАФ at the driver level via the `WithApplicationName` option in the `ydb.Open` call.
- **YDB Java SDK** ТАФ at the transport level via the `GrpcTransportBuilder.withApplicationName` method.
- **YDB Python SDK** ТАФ no dedicated parameter; the value is set per request via a generic header: `settings.with_header("x-ydb-application-name", "my-app")` (the `BaseRequestSettings.with_header` method).

Example. Direct requests from the Embedded UI to the `pool_adhoc` pool:

```
CREATE RESOURCE POOL CLASSIFIER cl_adhoc_ui WITH (
    RANK=200,
    RESOURCE_POOL='pool_adhoc',
    HAS_APP_NAME='ydb-ui'
);
```

HAS_FULL_SCAN

`HAS_FULL_SCAN` identifies requests that contain a full scan of the specified objects. A full scan is a table read without a key or key-range constraint. The argument is a path mask supporting wildcards `*` and `?`; the predicate matches if there is at least one such object in the request plan. Both row-oriented and column-oriented tables are supported. Objects to which the full-scan concept does not apply (for example, topics) are not considered.

Details of full-scan detection:

- **`LIMIT` does not cancel a full scan.** Without a key condition, the physical plan processes the entire table; `LIMIT` only limits the size of the result.
- **Secondary indexes.** A full scan of a secondary index's implementation table is counted against its own path. For example, for the table

  ```
  CREATE TABLE orders (
      Id Uint64 NOT NULL,
      Status Utf8,
      PRIMARY KEY (Id),
      INDEX by_status GLOBAL ON (Status)
  );
  ```

  the index table has path `/Root/orders/by_status/indexImplTable`. The query `SELECT * FROM orders VIEW by_status` performs a full scan of the index table ТАФ the main `/Root/orders` is not scanned. Therefore:

  - `HAS_FULL_SCAN='/Root/orders'` ТАФ **will not match**;
  - `HAS_FULL_SCAN='/Root/orders/by_status/indexImplTable'` or `HAS_FULL_SCAN='/Root/orders/*'` ТАФ **will match**.

Example. Reject requests that cause a full scan of the orders archive:

```
CREATE RESOURCE POOL CLASSIFIER cl_fullscan_reject WITH (
    RANK=100,
    ACTION='reject',
    HAS_FULL_SCAN='/Root/db/orders_archive/*'
);
```

HAS_STREAM

`HAS_STREAM` determines whether a request is streaming ТАФ that is, performs a long-running continuous read and/or write over data streams. Allowed values:

- `true` ТАФ the classifier matches streaming requests;
- `false` ТАФ the classifier matches non-streaming requests.

Example. Direct streaming requests to the `pool_stream` pool:

```
CREATE RESOURCE POOL CLASSIFIER cl_stream WITH (
    RANK=500,
    RESOURCE_POOL='pool_stream',
    HAS_STREAM=true
);
```

Notes

If `RANK` is not specified in the DDL for creating a resource pool classifier, it will be assigned the default value . All `RANK` values must be unique to ensure a strictly deterministic order of resource pool selection in case of conflicting conditions. This behavior is chosen to allow adding new resource pool classifiers between existing ones.

It is also possible to have a classifier that references a non-existent resource pool or one to which the user does not have access. In that case, it is skipped.

For limitations on the number of classifiers, see the limitations page.

Permissions

The permission `USE` on the database is required.

Example of granting such a permission:

```
GRANT 'USE' ON `/my_db` TO `user1@domain`;
```

Examples

Below is a combined example that composes several classifiers and predicates: rejecting full scans of archive tables, isolating streaming requests, and dedicating a pool for interactive admin queries from the Embedded UI.

Creating resource pools:

```
CREATE RESOURCE POOL pool_stream WITH (
    TOTAL_CPU_LIMIT_PERCENT_PER_NODE=20
);

CREATE RESOURCE POOL pool_adhoc_admin WITH (
    TOTAL_CPU_LIMIT_PERCENT_PER_NODE=10
);
```

Creating classifiers:

```
-- Reject full scans of archive tables.
CREATE RESOURCE POOL CLASSIFIER cl_fullscan_reject WITH (
    RANK=100,
    ACTION='reject',
    HAS_FULL_SCAN='/Root/db/orders_archive/*'
);

-- Direct streaming requests to a dedicated pool.
CREATE RESOURCE POOL CLASSIFIER cl_stream WITH (
    RANK=200,
    RESOURCE_POOL='pool_stream',
    HAS_STREAM=true
);

-- Admin requests from the Embedded UI тАФ into the interactive-queries pool.
-- AND condition: both MEMBER_NAME and HAS_APP_NAME must match.
CREATE RESOURCE POOL CLASSIFIER cl_adhoc_admin WITH (
    RANK=300,
    RESOURCE_POOL='pool_adhoc_admin',
    MEMBER_NAME='admin',
    HAS_APP_NAME='ydb-ui'
);
```

Classifiers are processed in ascending order of `RANK`; the first matching classifier is applied to the request. A request that does not match any classifier is directed to the `default` pool.

See also

- Workload Manager — resource consumption management
- ALTER RESOURCE POOL CLASSIFIER
- DROP RESOURCE POOL CLASSIFIER

CREATE SECRET

The `CREATE SECRET` statement creates a secret.

Syntax

```
CREATE SECRET secret_name
WITH (option = value[, ...])
```

- `secret_name` — the name of the secret to create.
- `option` — command option:
  - `value` — string with the secret value.
  - `inherit_permissions` — when enabled, rights on the secret are inherited from the directory where the secret is created. When disabled, only the right `DESCRIBE SCHEMA` is inherited from the directory. The secret owner gets all possible rights on it in any case. Default is `False`.

Permissions

Creating a secret requires the right `CREATE TABLE`.

Examples

Create a secret in the database root named `secret_name` with value `secret_value`:

```
CREATE SECRET secret_name WITH (value = "secret_value");
```

Create a secret in directory `dir` in the database root named `secret_name` with value `secret_value`. If directory `dir` does not exist, it will be created:

```
CREATE SECRET `dir/secret_name` WITH (value = "secret_value");
```

Create a secret in the database root named `secret_name` with value `secret_value` with the same rights as the secret's parent directory:

```
CREATE SECRET secret_name WITH (value = "secret_value", inherit_permissions = True);
```

See also

- ALTER SECRET
- DROP SECRET

CREATE TABLE

The `CREATE TABLE` call creates a table with the specified data schema and key columns (`PRIMARY KEY`). It allows defining secondary indexes on the created table.

```
CREATE TABLE [IF NOT EXISTS] <table_name> (
  [<column_name> <column_data_type>] [FAMILY <family_name>] [NULL | NOT NULL] [DEFAULT <default_value>]
  [COMPRESSION([algorithm=<algorithm_name>[, level=<value>]])]
  [ENCODING([OFF|DICT])]
  [, ...],
    INDEX <index_name>
      [GLOBAL|LOCAL]
      [UNIQUE]
      [SYNC|ASYNC]
      [USING <index_type>]
      ON ( <index_columns> )
      [COVER ( <cover_columns> )]
      [WITH ( <parameter_name> = <parameter_value>[, ...])]
    [, ...]
  PRIMARY KEY ( <column>[, ...]),
  [FAMILY <column_family> ( family_options[, ...])]
)
[PARTITION BY HASH ( <column>[, ...])]
[WITH (<setting_name> = <setting_value>[, ...])]

[AS SELECT ...]
```

Query parameters

table_name

Path of the table being created.

When choosing a table name, follow the general naming rules for schema objects.

IF NOT EXISTS

If a table with the specified name already exists, the statement execution is completely skipped — no checks or schema comparison are performed, and no error occurs. Note that the existing table may differ in structure from the one you intended to create with this query — no comparison or equivalence check is performed.

column_name

Name of the column being created in the new table.

When choosing a column name, follow the general column naming rules.

column_data_type

Data type of the column. The full list of data types supported by YDB is available in the YQL data types section.

FAMILY <family_name> (column setting)

> **Warning**
>
> Supported only for row-oriented tables.

Specifies the belonging of this column to the specified group of columns. For more details, see the section Column groups.

DEFAULT <default_value>

> **Warning**
>
> The `DEFAULT` option is supported:
>
> - Only for row-oriented tables.
> - Only with literal values.

Allows you to set a default value for the column. If no value is specified for this column when inserting a row, the specified default value will be used. The default value must match the data type of the column.

The `DEFAULT false NOT NULL` construct is not allowed due to ambiguity of interpretation. In this case, you should use a comma-separated list or change the order of the options.

NULL

This column can contain `NULL` values (by default).

NOT NULL

This column does not accept `NULL` values.

COMPRESSION([algorithm=<algorithm_name>[, level=]])

> **Warning**
>
> Supported only for column-oriented tables.

The following compression parameters can be set for columns:

- `algorithm` — the data compression algorithm. Allowed values: `off` (disable compression), `lz4`, `zstd`.
- `level` — the compression level, supported only for the `zstd` algorithm (values from 0 to 22 are allowed).

If `COMPRESSION()` is specified without parameters, the default compression is used for the column. Currently, it is `lz4`; in future versions, it will be possible to configure the default compression at the cluster or table level.

ENCODING([OFF|DICT])

Warning

Supported only for column-oriented tables.

Allows you to set the data encoding method for the column.

Available options:

- `ENCODING(DICT)` — enables dictionary encoding. Repeating values are replaced with small integer identifiers, and the values themselves are stored in a dictionary. Dictionary encoding is effective for columns with low cardinality (a small number of unique values). It reduces the amount of stored data and speeds up some operations. It is supported only for comparable data types, such as `String`, `Timestamp`, `UInt64`, and others. Using `ENCODING(DICT)` for incomparable types, such as `Json`, `JsonDocument`, or `Yson`, will result in an error.
- `ENCODING(OFF)` — disables special encoding. Data will be stored in the standard format without additional encoding.

If `ENCODING()` is set without parameters, the default encoding will be used for the column. Currently, it is `OFF`; in future versions, it will be possible to configure the default encoding at the database or table level.

INDEX

Defining an index on the table. Supported types:

- secondary indexes,
- vector indexes,
- full-text indexes,
- Bloom indexes,
- JSON indexes.

PRIMARY KEY

Defining the table's primary key. Specifies the columns that make up the primary key in the order listed. For more details on choosing a primary key, see the Choosing a primary key section.

PARTITION BY HASH

Defining partitioning keys for **column-oriented** tables. Specifies the columns by whose hash the data partitioning is performed. The columns must be part of the primary key. However, the columns do not necessarily have to be a prefix or suffix — the requirement is to be part of the primary key.

If the parameter is not specified, the table will be split into partitions by the same columns that are part of the primary key. For guidance on how to choose partitioning keys for column-oriented tables, see the article Choosing keys for maximum column-oriented table performance.

For more details on partitioning column-oriented tables, see the Partitioning Column-Oriented Tables section.

FAMILY <column_family> (column group settings)

Defining a column group with specified parameters. For more details, see the Column groups section.

WITH

Additional table creation parameters. For more details, see the Additional parameters (WITH) section.

Note

YDB supports two types of tables:

- Row-oriented.
- Column-oriented.

The table type when created is specified by the `STORE` parameter in the `WITH` block, where `ROW` means row-oriented table and `COLUMN` means column-oriented table:

```
CREATE <table_name> (
  columns
  ...
)

WITH (
  STORE = COLUMN -- Default value ROW
)
```

By default, if the `STORE` parameter is not specified, a row-oriented table is created.

Note

When choosing a table name, follow the general naming rules for schema objects.

AS SELECT

Creating and populating a table based on the results of the `SELECT` query. For more details, see the Creating a table filled with query results section.

Table creation examples

Creating a row-oriented table

```
  CREATE TABLE <table_name> (
    a Uint64,
    b Uint64,
    c Float,
    PRIMARY KEY (a, b)
  );
```

Example of creating a table using a default value (DEFAULT):

```
  CREATE TABLE table_with_default (
  id Uint64,
  name String DEFAULT "unknown",
  score Double NOT NULL DEFAULT 0.0,
  PRIMARY KEY (id)
);
```

For key columns, only primitive and serial data types are allowed; for non-key columns, only primitive data types are allowed.

Without additional modifiers, the column acquires an optional type and allows `NULL` to be written as values. To obtain a non-optional type, use `NOT NULL`.

It is mandatory to specify `PRIMARY KEY` with a non-empty list of columns. These columns become part of the key in the order they are listed.

Example of creating a row table with partitioning options:

```
CREATE TABLE <table_name> (
  a Uint64,
  b Uint64,
  c Float,
  PRIMARY KEY (a, b)
)
WITH (
  AUTO_PARTITIONING_BY_SIZE = ENABLED,
  AUTO_PARTITIONING_PARTITION_SIZE_MB = 512
);
```

This code will create a row table with automatic partitioning enabled by partition size (`AUTO_PARTITIONING_BY_SIZE`) and a preferred partition size (`AUTO_PARTITIONING_PARTITION_SIZE_MB`) of 512 megabytes. The full list of row table partitioning options is in the Row table partitioning section of the Table article.

Creating a column table

```
CREATE TABLE table_name (
  a Uint64 NOT NULL,
  b Timestamp NOT NULL,
  c Float,
  PRIMARY KEY (a, b)
)
PARTITION BY HASH(b)
WITH (
  STORE = COLUMN
);
```

For column tables, you can explicitly specify which columns will be used for partitioning using the `PARTITION BY HASH` construct. Typically, primary key columns with a large number of unique values are chosen for this, for example, `Timestamp`. If `PARTITION BY HASH` is not specified, partitioning will occur automatically across all columns that are part of the primary key. For more details on selecting and working with partitioning keys in column tables, see the Choosing keys for maximum column-oriented table performance article.

Currently, column tables do not support automatic repartitioning, so it is important to specify the correct number of partitions when creating a table using the `AUTO_PARTITIONING_MIN_PARTITIONS_COUNT` parameter:

```
CREATE TABLE table_name (
  a Uint64 NOT NULL,
  b Timestamp NOT NULL,
  c Float,
  PRIMARY KEY (a, b)
)
PARTITION BY HASH(b)
WITH (
  STORE = COLUMN,
  AUTO_PARTITIONING_MIN_PARTITIONS_COUNT = 10
);
```

This code will create a column table with 10 partitions. For a full list of column table partitioning options, see the Partitioning Column-Oriented Tables section of the Table article.

When creating row tables, you can specify:

- Secondary index.
- Vector index.
- Full-text index.
- JSON index.
- Bloom index.
- Column groups.
- Additional parameters.
- Creating and populating a table based on query results.

For column tables, when creating them, you can specify:

- Bloom index.
- Column groups.
- Additional parameters.
- Creating and populating a table based on query results.

INDEX

The INDEX construct is used to define a secondary index in a row-oriented table:

```
CREATE TABLE `<table_name>` (
  ...
    INDEX `<index_name>`
    [GLOBAL|LOCAL]
    [UNIQUE]
    [SYNC|ASYNC]
    [USING <index_type>]
    ON ( <index_columns> )
    [COVER ( <cover_columns> )]
    [WITH ( <parameter_name> = <parameter_value>[, ...])]
  [,   ...]
)
```

where:

- `GLOBAL/LOCAL` — global or local index, depending on the index type (`<index_type>`), only one of them may be available:
  - `GLOBAL` — an index implemented as a separate table or a set of tables. Synchronous update of such an index requires distributed transactions.
  - `LOCAL` — a local index within a shard of a columnar or row-based table, does not require distributed transactions during update, but does not provide pruning during search.
- `<index_name>` — unique index name by which data can be accessed.
- `SYNC/ASYNC` — indicator of index synchrony.
  - `SYNC` - synchronous index. Default value.
  - `ASYNC` - asynchronous index.
- `UNIQUE` — indicator of a unique secondary index. A unique index must be global synchronous (`GLOBAL UNIQUE SYNC`) and must not contain the `USING <index_type>` construct.
- `<index_type>` - index type, currently supported:
  - `secondary` — secondary index. Only the `GLOBAL` mode is available for secondary indexes. This is the default index type.
  - `vector_kmeans_tree` — vector index. More details are described in the Vector index section.
  - `fulltext_plain` — basic full-text index. More details are described in Fulltext index.
  - `fulltext_relevance` — full-text index with BM25 statistics for relevance calculation. More details are described in Fulltext index.
  - `json` — JSON index to speed up predicates `JSON_EXISTS` and `JSON_VALUE` on a column of type `Json` or `JsonDocument`. More details are described in JSON-index.
  - `bloom_filter` — local Bloom index. Only `LOCAL` is available. See ALTER TABLE ADD INDEX.
  - `bloom_ngram_filter` — local N-gram Bloom index. Only `LOCAL` is available. See ALTER TABLE ADD INDEX.
- `<index_columns>` — comma-separated list of column names of the table being created, which determines the composition and order of columns included in the index key. Must be specified. The index key will consist of these columns with the addition of the table's primary key columns.
- `<cover_columns>` — comma-separated list of column names of the table being created that will be stored in the index in addition to the index key columns, allowing you to get additional data without accessing the table. Empty by default.
- `<parameter_name>` and `<parameter_value>` are index parameters specific to a particular `<index_type>` .

> **Warning**
>
> Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Example

```
CREATE TABLE my_table (
    a Uint64,
    b Bool,
    c Utf8,
    d Date,
    INDEX idx_d GLOBAL ON (d),
    INDEX idx_ba GLOBAL ASYNC ON (b, a) COVER (c),
    INDEX idx_uniq GLOBAL UNIQUE SYNC ON (c),
    PRIMARY KEY (a)
)
```

Vector index

Vector index in row-oriented tables is created using the same syntax as secondary indexes, by specifying `vector_kmeans_tree` as the index type. Subset of syntax available for vector indexes:

```
CREATE TABLE `<table_name>` (
    ...
    INDEX `<index_name>`
        GLOBAL
        [SYNC]
        USING vector_kmeans_tree
        ON ( <index_columns> )
        [COVER ( <cover_columns> )]
        [WITH ( <parameter_name> = <parameter_value>[, ...])]
    [,   ...]
)
```

Where:

- `<index_name>` - unique index name for data access
- `SYNC` - indicates synchronous data writing to the index. This is the only currently available option, and it is used by default.
- `<index_columns>` - comma-separated list of table columns used for index searches (the last column is used as embedding, others as filtering columns)
- `<cover_columns>` - list of additional table columns stored in the index to enable retrieval without accessing the main table
- `<parameter_name>` and `<parameter_value>` - list of key-value parameters:

- common parameters for all vector indexes:
  - `vector_dimension` - embedding vector dimensionality (should be between 1 and 16384)
  - `vector_type` - vector value type (`float` , `uint8` , or `int8`)
  - `distance` - distance function (`cosine` , `manhattan` , or `euclidean`), mutually exclusive with `similarity`
  - `similarity` - similarity function (`inner_product` or `cosine`), mutually exclusive with `distance`
- specific parameters for `vector_kmeans_tree` (read more about the index type):
  - `clusters` - number of centroids for k-means algorithm (should be between 2 and 2048)
  - `levels` - number of levels in the tree (should be between 1 and 16)
  - `overlap_clusters` - the number of nearest clusters to add each vector to (default 1)
  - the total number of nodes in the tree, calculated as `clusters` raised to the power of `levels` , should be no more than 1073741824
  - the product of `vector_dimension` and `clusters` should be no more than 4194304

Warning

It is recommended to create a vector index after loading data into the table, as an index created on an empty table will have only one cluster and will not speed up the search at all. For more details, see Updating Vector Indexes.

Warning

Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Example

```
CREATE TABLE user_articles (
    article_id Uint64,
    user String,
    title String,
    text String,
    embedding String,
    INDEX emb_cosine_idx GLOBAL SYNC USING vector_kmeans_tree
    ON (user, embedding) COVER (title, text)
    WITH (
        distance="cosine",
        vector_type="float",
        vector_dimension=512,
        clusters=128,
        levels=2,
        overlap_clusters=3
    ),
    PRIMARY KEY (article_id)
)
```

Fulltext index

Fulltext indexes in row-oriented tables are created using the same syntax as secondary indexes, by specifying `fulltext_plain` or `fulltext_relevance` as the index type. Subset of syntax available for fulltext indexes:

```
CREATE TABLE `<table_name>` (
    ...
    INDEX `<index_name>`
        GLOBAL
        [SYNC]
        USING fulltext_plain | fulltext_relevance
        ON ( <text_column> )
        [COVER ( <cover_columns> )]
        [WITH ( <parameter_name> = <parameter_value>[, ...])]
    [,   ...]
)
```

Where:

- `<index_name>` - unique index name for data access
- `SYNC` - indicates synchronous data writing to the index. This is the only currently available option, and it is used by default.
- `<text_column>` - a single table column with text content (currently only one indexed column is supported)
- `<cover_columns>` - list of additional table columns stored in the index to enable retrieval without accessing the main table
- `<parameter_name>` and `<parameter_value>` - list of key-value parameters:

- common parameters for all fulltext indexes:
  - `tokenizer` - tokenizer type (`standard` , `whitespace` , or `keyword`)
  - `use_filter_lowercase` - lowercase filter (`true` or `false`)
  - `use_filter_length` - token length filter (`true` or `false`); when `true` , tokens shorter than `filter_length_min` or longer than `filter_length_max` are not indexed and are ignored during search
  - `filter_length_min` - minimum token length (positive integer); only applied when `use_filter_length=true`
  - `filter_length_max` - maximum token length (positive integer); only applied when `use_filter_length=true`
  - `use_filter_snowball` - Snowball stemmer filter (`true` or `false`)
  - `language` - language for the Snowball stemmer (for example, `english` , `russian`)
  - `use_filter_ngram` - n-gram filter (`true` or `false`)
  - `use_filter_edge_ngram` - edge n-gram filter (`true` or `false`)
  - `filter_ngram_min_length` - minimum n-gram length (positive integer)
  - `filter_ngram_max_length` - maximum n-gram length (positive integer)

Warning

Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Examples

Basic example

```
CREATE TABLE articles (
    id Uint64,
    title String,
    body String,
    INDEX ft_idx GLOBAL USING fulltext_relevance
    ON (body) COVER (title)
    WITH (
        tokenizer=standard,
        use_filter_lowercase=true
    ),
    PRIMARY KEY (id)
)
```

Example with n-grams

N-grams allow finding substrings within words. When using the n-gram filter, each word is split into all possible substrings of specified length. For example, the word "search" with parameters `filter_ngram_min_length=3` and `filter_ngram_max_length=5` will be split into: "sea", "ear", "arc", "rch" (3-grams), "sear", "earc", "arch" (4-grams), "searc", "earch" (5-grams). This enables finding documents by partial matches anywhere within a word.

```
CREATE TABLE products (
    id Uint64,
    name String,
    description String,
    INDEX ft_ngram_idx GLOBAL USING fulltext_plain
    ON (name)
    WITH (
        tokenizer=standard,
        use_filter_lowercase=true,
        use_filter_ngram=true,
        filter_ngram_min_length=3,
        filter_ngram_max_length=5
    ),
```

```
    PRIMARY KEY (id)
)
```

Example with edge n-grams

Edge n-grams create substrings only from the beginning of a word. For example, the word "search" with parameters `filter_ngram_min_length=2` and `filter_ngram_max_length=10` will be split into: "se", "sea", "sear", "searc", "search". This is particularly useful for implementing autocomplete functionality, when you need to find words that start with a user-entered prefix.

```
CREATE TABLE users (
    id Uint64,
    username String,
    email String,
    INDEX ft_edge_ngram_idx GLOBAL USING fulltext_plain
    ON (username)
    WITH (
        tokenizer=standard,
        use_filter_lowercase=true,
        use_filter_edge_ngram=true,
        filter_ngram_min_length=2,
        filter_ngram_max_length=10
    ),
    PRIMARY KEY (id)
)
```

JSON-index

JSON indexes in row tables are created using the same syntax as secondary indexes, by specifying `json` as the index type. A subset of the syntax available for JSON indexes:

```
CREATE TABLE `<table_name>` (
    ...
    INDEX `<index_name>`
        GLOBAL
        [SYNC]
        USING json
        ON ( <json_column> )
    [,   ...]
)
```

Where:

- `<index_name>` — unique index name for data access.
- `SYNC` — specifies synchronous index update. This is the only mode available for JSON indexes; explicit specification is not required.
- `<json_column>` — table column of type `Json` or `JsonDocument`. A JSON index is built on a single column only.

A JSON index does not support the `COVER` expression — attempting to specify it will result in an error.

> **Warning**
>
> Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Example

```
CREATE TABLE documents (
    id Uint64 NOT NULL,
    payload JsonDocument NOT NULL,
    INDEX json_idx GLOBAL USING json ON (payload),
    PRIMARY KEY (id)
)
```

In this example, a table `documents` is created with a JSON index `json_idx` on column `payload`. The index will be used by queries whose `WHERE` predicate contains calls to `JSON_EXISTS` or `JSON_VALUE` on column `payload`.

Bloom skip index

Bloom skip indexes are local indexes and must be declared with `LOCAL`. In the `INDEX` clause use `bloom_filter` or `bloom_ngram_filter` (similar to a secondary index, but with `LOCAL` and the matching `USING` type). See also local indexes.

```
CREATE TABLE `<table_name>` (
    ...
    INDEX `<index_name>`
        LOCAL
        USING bloom_filter | bloom_ngram_filter
        ON ( <index_columns> )
        [WITH ( <parameter_name> = <parameter_value>[, ...])]
    [,   ...]
)
```

Where:

- `<index_name>`: Index name.
- `LOCAL`: Required keyword for Bloom skip indexes.
- `<index_columns>`: List of columns used to build the index; the number of columns depends on the table type and index type. On row-oriented tables, the indexed columns of a `bloom_filter` must be a left prefix of the primary key.
- `COVER (...)` and data columns are not supported for Bloom skip indexes.

`WITH (...)` parameters:

- `bloom_filter`
  - `false_positive_probability`: Target false-positive rate of the filter: the fraction of fragments that are not skipped even though the requested value is not there (range `(0, 1)`). A lower value reduces extra reads but increases index size.
    - If omitted, the default is `0.0001` for row-oriented tables and `0.1` for column-oriented tables. The stricter default on row-oriented tables matches OLTP point lookups; on column-oriented tables it reflects analytical scans, where a larger trade-off between index size and fragment skipping is acceptable.
  - Indexed columns: YQL types that support equality comparison, except `Yson`, `Json`, and `JsonDocument` (see comparison operators). One column in a column-oriented table; multiple columns in a row-oriented table. On row-oriented tables, the indexed columns must form a left prefix of the primary key.
- `bloom_ngram_filter` (`String` and `Utf8` columns; column-oriented tables only)
  - `ngram_size`: N-gram length, an integer from `3` to `8` (default `3`).
  - `false_positive_probability`: Target false-positive rate (range `(0, 1)`; default `0.1`).
  - `case_sensitive`: Whether n-grams respect character case: `true` or `false` (default `true`).

Creating and altering these indexes on an existing table is described in `ALTER TABLE ADD INDEX`.

Examples

`bloom_filter` index

```
CREATE TABLE events (
    id Uint64,
    resource_id Utf8,
    PRIMARY KEY (id),
    INDEX idx_bloom LOCAL USING bloom_filter
        ON (resource_id)
        WITH (false_positive_probability = 0.01)
);
```

`bloom_ngram_filter` index

```
CREATE TABLE events (
    id Uint64,
    message Utf8,
    PRIMARY KEY (id),
    INDEX idx_ngram LOCAL USING bloom_ngram_filter
        ON (message)
        WITH (
            ngram_size = 3,
            false_positive_probability = 0.01,
            case_sensitive = true
        )
);
```

min_max index

min-max index is a local index that can only be specified with the `LOCAL` keyword. When creating a table, the `min_max` type is used in the `INDEX` section (similar to a secondary index, but with a mandatory `LOCAL` and corresponding `USING`). See also local indexes.

```
CREATE TABLE `<table_name>` (
    ...
    INDEX `<index_name>`
        LOCAL
        USING min_max
        ON ( <index_column> )
    [,   ...]
)
```

Where:

- `<index_name>` — index name.
- `LOCAL` — required keyword for the min_max index.
- `<index_column>` — the column on which the index is built. You must specify exactly one column.
- For the min_max index, covering columns (`COVER (...)`) and additional data columns are not supported.

`WITH (...)` parameters:

The min_max index has no specific parameters `WITH (...)`.

Creating a min_max index for an existing table is described in the `ALTER TABLE ADD INDEX` section.

Example

```
CREATE TABLE events (
    id Uint64,
    created_at Timestamp,
    level Int32,
    PRIMARY KEY (id),
    INDEX idx_created_at LOCAL USING min_max
        ON (created_at),
    INDEX idx_level LOCAL USING min_max
        ON (level)
)
WITH (
    STORE = COLUMN
);
```

Column groups

> **Warning**
>
> Supported only for row-oriented tables.

Columns of the same table can be grouped to set the following parameters:

- `DATA`: A storage device type for the data in this column group. Acceptable values: `"ssd"`, `"rot"`.
- `COMPRESSION`: A data compression codec. Acceptable values: `"off"`, `"lz4"`.
- `CACHE_MODE`: Caching mode. Acceptable values: `"in_memory"`, `"regular"`.

By default, all columns are in the same group named `default`. If necessary, the parameters of this group can also be redefined, if they are not redefined, then predefined values are applied.

Example

In the example below, for the created table, the `family_large` group of columns is added and set for the `series_info` column, and the parameters for the default group, which is set by `default` for all other columns, are also redefined.

```
CREATE TABLE series_with_families (
    series_id Uint64,
    title Utf8,
    series_info Utf8 FAMILY family_large,
    release_date Uint64,
    PRIMARY KEY (series_id),
    FAMILY default (
        DATA = "ssd",
        COMPRESSION = "off",
        CACHE_MODE = "in_memory"
    ),
    FAMILY family_large (
        DATA = "rot",
        COMPRESSION = "lz4",
        CACHE_MODE = "regular"
    )
);
```

> **Note**
>
> Available types of storage devices depend on the YDB cluster configuration.

Additional parameters (WITH)

You can also specify a number of YDB-specific parameters for the table. When you create a table, those parameters are listed in the `WITH` clause:

```
CREATE TABLE table_name (...)
WITH (
    key1 = value1,
    key2 = value2,
    ...
)
```

Here, `key` is the name of the parameter and `value` is its value.

The list of allowable parameter names and their values is provided on the table description page YDB.

For example, such a query will create a string table with automatic partitioning enabled based on partition size and a preferred size of each partition being 512 megabytes:

```
CREATE TABLE my_table (
    id Uint64,
    title Utf8,
    PRIMARY KEY (id)
)
WITH (
    AUTO_PARTITIONING_BY_SIZE = ENABLED,
    AUTO_PARTITIONING_PARTITION_SIZE_MB = 512
);
```

A colum-oriented table is created by specifying the parameter `STORE = COLUMN` in the `WITH` clause:

```
 CREATE TABLE table_name (
    a Uint64 NOT NULL,
    b Timestamp NOT NULL,
    c Float,
    PRIMARY KEY (a, b)
  )
  PARTITION BY HASH(b)
  WITH (
    STORE = COLUMN
  );
```

The properties and capabilities of columnar tables are described in the article Table, and the specifics of their creation through YQL are described on the page CREATE TABLE.

Time to Live (TTL)

The TTL (Time to Live) — the lifespan of a row — can be specified in the WITH clause for row-based and columnar tables. TTL automatically deletes rows or evicts them to external storage when the specified number of seconds has passed since the time recorded in the TTL column. TTL can be specified when creating row-based and columnar tables or added later using the `ALTER TABLE` command only for row-based tables.

The short form of the TTL value for specifying the time to delete rows:

```
Interval("<literal>") ON column [AS <unit>]
```

The general form of the TTL value:

```
Interval("<literal1>") action1, ..., Interval("<literalN>") actionN ON column [AS <unit>]
```

- `action` — the action performed when the TTL expression triggers. Allowed values:
  - `DELETE` — delete the row;
  - `TO EXTERNAL DATA SOURCE <path>` — evict the row to external storage specified by the external data source at the path `<path>`.
- `<unit>` — the unit of measurement, specified only for columns with a numeric type:
  - `SECONDS`;
  - `MILLISECONDS`;
  - `MICROSECONDS`;
  - `NANOSECONDS`.

Example of creating a row-oriented and column-oriented tables with TTL:

Creating row-oriented table with TTL

```
CREATE TABLE my_table (
    id Uint64,
    title Utf8,
    expire_at Timestamp,
    PRIMARY KEY (id)
)
WITH (
    TTL = Interval("PT0S") ON expire_at
);
```

Creating column-oriented table with TTL

```
CREATE TABLE table_name (
    a Uint64 NOT NULL,
    b Timestamp NOT NULL,
    c Float,
    PRIMARY KEY (a, b)
)
PARTITION BY HASH(b)
WITH (
    STORE = COLUMN,
    TTL = Interval("PT0S") ON b
);
```

Example of creating a column-oriented table with eviction to external storage:

> **Warning**
>
> Supported only for column-oriented tables. Support for row-oriented tables is currently under development.

```
CREATE TABLE table_name (
    a Uint64 NOT NULL,
    b Timestamp NOT NULL,
    c Float,
    PRIMARY KEY (a, b)
)
PARTITION BY HASH(b)
WITH (
    STORE = COLUMN,
    TTL =
        Interval("PT1D") TO EXTERNAL DATA SOURCE `/Root/s3`,
        Interval("P2D") DELETE
    ON b
);
```

Creating a table filled with query results

> **Warning**
>
> Supported only for column-oriented tables. Support for row-oriented tables is currently under development.

`CREATE TABLE AS` creates a new table table filled with data from query results.

```
CREATE TABLE table_name (
    PRIMARY KEY ( column, ... )
)
WITH ( key = value, ... )
AS SELECT ...
```

Names and types of columns will correspond to the `SELECT` results.
Non-optional columns will also have the `NOT NULL` constraint.

The `CREATE TABLE AS` syntax allows you to specify only the primary key and `WITH` parameters, so when creating a table, specifying column names, secondary indexes, vector indexes, full-text indexes, local bloom indexes, local min_max indexes, and column groups is not supported. The column names and data types of the new table are automatically inherited from the result set of the SELECT query. You can change all of the above using `ALTER TABLE` after creating the table. Additionally, additional parameters are supported.

Considerations

> **Warning**
>
> Rows are overwritten, similar to using `REPLACE INTO`, but the order in which rows are written is unpredictable.
>
> If `SELECT` returns two or more rows with the same primary key value, after the `CREATE TABLE AS` is executed, there will only be one row with that primary key value in the created table. Which record from the `SELECT` was written to the table is undetermined.

- `CREATE TABLE AS` is supported only in the implicit transaction control mode. The table will appear at the specified path already populated.
- `CREATE TABLE AS` can only be a single DML/DDL statement in a query. It's possible to use PRAGMA, DECLARE or named expressions in the same query.
- `CREATE TABLE AS` doesn't cause lock conflicts with other transactions. It doesn't use locks. Reads use a consistent snapshot. Moving or splitting tablets doesn't cause errors.
- `CREATE TABLE AS` allows using column-oriented tables and row-oriented tables in the same query.
- `CREATE TABLE AS` creates a table in the temporary directory `.tmp/sessions`, and after successful data write moves it to the specified location. If the operation is interrupted due to an error, the temporary table is not deleted immediately but remains in the system for some time.

Examples

- Creating a columnar table from query results

```
CREATE TABLE my_table (
    PRIMARY KEY (key1, key2)
) WITH (
    STORE=COLUMN
) AS SELECT
    key AS key1,
    Unwrap(other_key) AS key2,
    value,
    String::Contains(value, "test") AS has_test
FROM other_table;
```

CREATE TRANSFER

Creates a transfer from a topic to a table.

Syntax:

```
CREATE TRANSFER transfer_name
FROM topic_name TO table_name USING lambda
WITH (option = value[, ...])
```

- `transfer_name` — the name of the transfer to be created.
- `topic_name` — the name of the topic containing the source messages for transformation and writing to the table.
- `table_name` — the name of the table where the data will be written.
- `lambda` — lambda-function for message transformation.
- `option` — a command option:
  - `CONNECTION_STRING` — the connection string to the database containing the topic. This is only specified if the topic is located in a different YDB database.
  - Authentication settings for the topic's database (required if the topic is in another database), using one of the following methods:
    - Using a token:
      - `TOKEN_SECRET_PATH` — the secret that contains the token.
    - Using a username and password:
      - `USER` — the username.
      - `PASSWORD_SECRET_PATH` — the secret that contains the password.
    - Using a service agent:
      - `SERVICE_ACCOUNT_ID` — the identificator of the service account.
      - `INITIAL_TOKEN_SECRET_PATH` — the secret that contains the account's token. It is used for initial authentication.
  - `CONSUMER` — the name of the source topic consumer. If a name is specified, a consumer with that name must already exist in the topic, and the transfer will start processing messages from the first uncommitted message in the topic. If no name is specified, a consumer will be added to the topic automatically, and the transfer will start processing messages from the first message stored in the topic. The name of the automatically created consumer can be obtained from the description of the transfer instance.
  - Table write batching parameters let you balance the latency of records appearing in the table against the resources required by the transfer. Batching parameters affect the processing of each topic partition independently. Change batching parameters with caution, as this can either improve or degrade message stream processing speed, and may even lead to a denial of service if the parameters are misconfigured. For example, writing to the table in small batches can overload the table and degrade its performance, while an excessively large batch size can cause the server to run out of available memory.
    - `BATCH_SIZE_BYTES` — the batch size in bytes. Default: 8 MB.
    - `FLUSH_INTERVAL` — the table write interval. Default: 60 seconds. Data is written to the table at this interval, even if the batch has not reached the size specified in the `BATCH_SIZE_BYTES` parameter.

Permissions

The following permissions are required to create a transfer:

- `CREATE TABLE` — to create a transfer instance;
- `ALTER SCHEMA` — to automatically create a topic consumer (if applicable);
- `SELECT ROW` — to read messages from the source topic;
- `UPDATE ROW` — to update rows in the destination table.

Examples

Creating a transfer instance from the `example_topic` topic to the `example_table` table in the current database:

```
CREATE TABLE example_table (
    partition Uint32 NOT NULL,
    offset Uint64 NOT NULL,
    message Utf8,
    PRIMARY KEY (partition, offset)
);

CREATE TOPIC example_topic;

$transformation_lambda = ($msg) -> {
    return [
        <|
            partition: $msg._partition,
            offset: $msg._offset,
            message: CAST($msg._data AS Utf8)
        |>
    ];
};

CREATE TRANSFER example_transfer
    FROM example_topic TO example_table USING $transformation_lambda;
```

Creating a transfer instance from the `example_topic` topic in the `/Root/another_database` database to the `example_table` table in the current database. Before creating the transfer, you need to create the destination table in the current database and create the source topic in the `/Root/another_database` database:

> **Tip**
>
> Before performing the operation, create a secret with authentication credentials to connect, or make sure it exists and you have access to it.

```
$transformation_lambda = ($msg) -> {
    return [
        <|
            partition: $msg._partition,
            offset: $msg._offset,
            message: CAST($msg._data AS Utf8)
        |>
    ];
};

CREATE TRANSFER example_transfer
    FROM example_topic TO example_table USING $transformation_lambda
WITH (
    CONNECTION_STRING = 'grpcs://example.com:2135/?database=/Root/another_database',
    TOKEN_SECRET_PATH = 'my_secret'
);
```

Creating a transfer instance and explicitly specifying the consumer `existing_consumer_of_topic`:

```
$transformation_lambda = ($msg) -> {
    return [
        <|
            partition: $msg._partition,
            offset: $msg._offset,
            message: CAST($msg._data AS Utf8)
        |>
    ];
};
CREATE TRANSFER example_transfer
    FROM example_topic TO example_table USING $transformation_lambda
WITH (
    CONSUMER = 'existing_consumer_of_topic'
);
```

Example of processing a message in JSON format:

```
// example message:
// {
//   "update": {
//     "operation":"value_1"
//   },
//   "key": [
//     "id_1",
//     "2019-01-01T15:30:00.000000Z"
//   ]
// }

$transformation_lambda = ($msg) -> {
    $json = CAST($msg._data AS JSON);
    return [
        <|
            timestamp: CAST(Yson::ConvertToString($json.key[1]) AS Timestamp),
            object_id: CAST(Yson::ConvertToString($json.key[0]) AS Utf8),
            operation: CAST(Yson::ConvertToString($json.update.operation) AS Utf8)
        |>
    ];
};

CREATE TRANSFER example_transfer
    FROM example_topic TO example_table USING $transformation_lambda;
```

Creating a transfer instance and explicitly specifying the batching option:

```
$transformation_lambda = ($msg) -> {
    return [
        <|
            partition: $msg._partition,
            offset: $msg._offset,
            message: CAST($msg._data AS Utf8)
        |>
    ];
};
CREATE TRANSFER example_transfer
    FROM example_topic TO example_table USING $transformation_lambda
WITH (
    BATCH_SIZE_BYTES = 1048576,
    FLUSH_INTERVAL = Interval('PT60S')
);
```

Lambda function

A message transformation lambda function takes a single structured parameter containing the message from the topic and returns a list of structures corresponding to the table rows for insertion.

Example:

```
$lambda = ($msg) -> {
  return [
    <|
      column_1: $msg._create_timestamp,
      column_2: $msg._data
    |>
  ];
};
```

In this example:

- `$msg` — the message received from the topic.
- `column_1` and `column_2` — the names of the table columns.
- `$msg._create_timestamp` and `$msg._data` — the values that will be written to the table. The value types must match the table column types. For example, if the `column_2` table column has the `String` type, the type of `$msg._data` must also be `String`.

The following fields are available in a topic message:

Attribute	Value type	Description
`_create_timestamp`	`Timestamp`	Message creation time
`_data`	`String`	Message body
`_offset`	`Uint64`	Message offset
`_partition`	`Uint32`	Message's partition number
`_producer_id`	`String`	Producer ID
`_seq_no`	`Uint64`	Message sequence number
`_write_timestamp`	`Timestamp`	Message write time

Testing lambda functions

To test a lambda function during development, you can simulate a topic message by passing a structure with the same fields that the transfer will provide. Example:

```
$lambda = ($msg) -> {
  return [
    <|
      offset: $msg._offset,
      data: $msg._data
    |>
  ];
};

$msg = <|
  _data: "value",
  _offset: CAST(1 AS Uint64),
  _partition: CAST(2 AS Uint32),
  _producer_id: "producer",
  _seq_no: CAST(3 AS Uint64)
|>;

SELECT $lambda($msg);
```

If a lambda function contains complex transformation logic, you can extract it into a separate lambda function to simplify testing.

```
$extract_value = ($data) -> {
  -- complex transformations
  return $data;
};

$lambda = ($msg) -> {
  return [
    <|
      column: $extract_value($msg._data)
    |>
  ];
};

-- You can test the extract_value lambda function like this

SELECT $extract_value('converted value');
```

See Also

- ALTER TRANSFER

- DROP TRANSFER
- Data transfer

CREATE VIEW

`CREATE VIEW` defines a view with a given query.

A view logically represents a table formed by a given query. The view does not physically store the table but executes the query to produce the data whenever the view is accessed.

Syntax

```
CREATE VIEW [IF NOT EXISTS] <name>
[ WITH ( <view_option_name> [= <view_option_value>] [, ... ] ) ]
AS <query>
```

Parameters

- `IF NOT EXISTS` - when specified, the statement does not return an error if a view with the given name already exists.
- `name` - the name of the view to be created. The name must be distinct from the names of all other schema objects.
- `query` - the `SELECT` query, which will be used to produce the logical table the view represents.
- `WITH ( <view_option_name> [= <view_option_value>] [, ... ] )` specifies optional parameters for a view. The following parameters are supported:
  - `security_invoker` (Bool) causes the underlying base relations to be checked against the privileges of the user of the view rather than the view owner.

> **Note**
>
> When choosing a name for the view, consider the common schema object naming rules.

Notes

The `security_invoker` option must always be set to true because the default behavior for views is to execute the query on behalf of the view's creator, which is not supported yet.

The execution context of the view's query differs from the context of the enclosing `SELECT`. It does not see previously defined `PRAGMA`s, named expressions, etc. Most importantly, users must specify the tables (or views) they select from in the view's query by their schema-qualified names. You can see in the examples that the absolute path like `/domain/database/path/to/underlying_table` is used to specify the table from which a view reads data. The particular context of the view's query compilation might change in the upcoming releases.

If you wish to specify column names that you would like to see in the output of the view, you might do so by modifying the view's query:

```
CREATE VIEW view_with_a_renamed_column WITH (security_invoker = TRUE) AS
SELECT
    original_column_name AS custom_column_name
FROM `/domain/database/path/to/underlying_table`;
```

Asterisk (`*`) expansion in the view's query happens each time you read from the view. The list of columns returned by the following statement:

```
/*
CREATE VIEW view_with_an_asterisk WITH (security_invoker = TRUE) AS
SELECT
    *
FROM `/domain/database/path/to/underlying_table`;
*/

SELECT * FROM view_with_an_asterisk;
```

will change if the list of columns of the `underlying_table` is altered.

Examples

Create a view that will list only recent series from the `series` table:

```
CREATE VIEW recent_series WITH (security_invoker = TRUE) AS
SELECT
    *
FROM `/domain/database/path/to/series`
WHERE
    release_date > Date("2020-01-01");
```

Create a view that will list the titles of the first episodes of the recent series:

```
CREATE VIEW recent_series_first_episodes_titles WITH (security_invoker = TRUE) AS
SELECT
    episodes.title AS first_episode
FROM `/domain/database/path/to/recent_series`
    AS recent_series
JOIN `/domain/database/path/to/episodes`
    AS episodes
USING(series_id)
WHERE episodes.season_id = 1 AND episodes.episode_id = 1;
```

See also

- ALTER VIEW
- DROP VIEW

CREATE RESOURCE POOL

`CREATE RESOURCE POOL` creates a resource pool.

Syntax

```
CREATE RESOURCE POOL <name>
WITH ( <parameter_name> [= <parameter_value>] [, ... ] )
```

- `name` is the name of the resource pool being created. It must be unique. Path notation is not allowed (i.e., it must not contain `/`).
- `WITH ( <parameter_name> [= <parameter_value>] [, ... ] )` allows setting parameter values that define the behavior of the resource pool.

Parameters

- `CONCURRENT_QUERY_LIMIT` (Int32) — an optional field that sets the number of concurrently executing queries in the resource pool. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `QUEUE_SIZE` (Int32) — an optional field that defines the size of the waiting queue. The system can hold no more than queries at a time. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `DATABASE_LOAD_CPU_THRESHOLD` (Int32) — an optional field that sets the CPU load threshold for the entire database, after which queries are not sent for execution and remain in the queue. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `TOTAL_MEMORY_LIMIT_PERCENT_PER_NODE` (Double) is an optional field that defines the percentage of available memory on the node that all queries in this resource pool can use. If the value is `-1`, there are no limits. Default value: `-1`. Valid values: .

  If a limit is set and the total memory consumption by queries in the pool reaches this limit:

  - New queries that require memory will fail with error `OVERLOADED`.
  - Already running queries that need additional memory will also fail with error `OVERLOADED`.
- `TOTAL_CPU_LIMIT_PERCENT_PER_NODE` (Double) — an optional field that sets the percentage of available CPU that all queries on the node in this resource pool can use. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `QUERY_CPU_LIMIT_PERCENT_PER_NODE` (Double) — an optional field that defines the percentage of available CPU on the node for a single query in the resource pool. If the value is `-1`, there is no limit. Default value: `-1`. Valid values: .
- `RESOURCE_WEIGHT` (Int32) — an optional field that sets weights for distributing resources among pools. If the value is `-1`, weights are not used. Default value: `-1`. Valid values: .

> **Warning**
>
> The `QUERY_MEMORY_LIMIT_PERCENT_PER_NODE` parameter is currently not supported. To limit the amount of memory allocated to a resource pool, use the `TOTAL_MEMORY_LIMIT_PERCENT_PER_NODE` parameter.

Notes

Queries are always executed in some resource pool. By default, all queries are sent to the resource pool `default`, which is created automatically and cannot be deleted — it is always present in the system.

If the value of parameter `CONCURRENT_QUERY_LIMIT` is set to 0, then all queries sent to this pool will be immediately terminated with status `PRECONDITION_FAILED`.

Permissions

Requires permission `CREATE TABLE` on the directory `.metadata/workload_manager/pools`, an example of granting such permission:

```
GRANT 'CREATE TABLE' ON `.metadata/workload_manager/pools` TO `user1@domain`;
```

```
CREATE RESOURCE POOL olap WITH (
    CONCURRENT_QUERY_LIMIT=20,
    QUEUE_SIZE=1000,
    DATABASE_LOAD_CPU_THRESHOLD=80,
    RESOURCE_WEIGHT=100,
    TOTAL_MEMORY_LIMIT_PERCENT_PER_NODE=80,
    TOTAL_CPU_LIMIT_PERCENT_PER_NODE=70
)
```

In the example above, a resource pool is created with the following limits:

- Maximum number of concurrent queries — 20.
- Maximum queue size — 1000.
- When the database load reaches 80%, queries stop running concurrently.
- All queries in the pool can consume no more than 80% of the available memory on the node.
- The total limit on available CPU for all queries in the pool on the node is 70%.
- The resource pool has a weight of 100, which only takes effect in case of oversubscription.

See also

- Workload Manager — resource consumption management
- ALTER RESOURCE POOL
- DROP RESOURCE POOL

CREATE RESOURCE POOL CLASSIFIER

`CREATE RESOURCE POOL CLASSIFIER` creates a resource pool classifier.

Syntax

```
CREATE RESOURCE POOL CLASSIFIER <name>
WITH ( <parameter_name> [= <parameter_value>] [, ... ] )
```

- `name` тАФ the name of the resource pool classifier being created. It must be unique. The name must not contain characters prohibited for schema objects.
- `WITH ( <parameter_name> [= <parameter_value>] [, ... ] )` тАФ allows you to set parameter values that define the behavior of the resource pool classifier.

Common parameters

- `RANK` (Int64) тАФ an optional field that specifies the selection order of the resource pool classifier. If the value is not specified, the maximum existing `RANK` is taken and 1000 is added to it. Valid values: a unique number in the range .
- `RESOURCE_POOL` (String) тАФ the name of the resource pool to which requests matching the classifier's predicates are directed.
- `ACTION` (Enum) тАФ the action applied to a request when the classifier matches. Allowed value тАФ `reject`: the request is rejected, and the user receives an error of the form `Request is rejected by classifier '<name>' (rank=<rank>)`.

> **Note**
>
> The `RESOURCE_POOL` and `ACTION` parameters are mutually exclusive: exactly one of them must be specified in a classifier.

Predicate parameters

A predicate is a condition checked for an incoming request. A classifier matches if **all** of its predicates are satisfied (logical **AND**). To express **OR** logic, create multiple classifiers with different `RANK` values. Classifiers are processed in ascending order of `RANK`; processing stops at the first match тАФ the request has that classifier's `ACTION` applied or is directed to its `RESOURCE_POOL`.

Predicate parameters are optional. A classifier without any predicates matches any request тАФ this is useful for a "catch-all" classifier with the maximum `RANK`, for example, to direct all unclassified traffic to a specific pool (`RESOURCE_POOL`) or to reject it (`ACTION='reject'`).

> **Warning**
>
> Be careful when creating a "catch-all" classifier: all classifiers following it by `RANK` will never fire тАФ processing stops at the first match, and the "catch-all" always matches.

List of predicates:

- `MEMBER_NAME` (String) тАФ the SID of the user or group on whose behalf the request was made. See below for details.
- `HAS_PATH` (String) тАФ path to a YDB object accessed by the request; supports wildcards `*` and `?`. See below for details.
- `HAS_APP_NAME` (String) тАФ client application identifier. See below for details.
- `HAS_FULL_SCAN` (String) тАФ path to an object for which a full scan is expected; supports wildcards `*` and `?`. See below for details.
- `HAS_STREAM` (Bool) тАФ indicates whether the request is streaming. See below for details.

MEMBER_NAME

`MEMBER_NAME` is compared character by character with the user's SID or any group SID from their authentication token. The SID format depends on how the user logged into the system.

- **Built-in YDB users (login/password)** тАФ the SID matches the username, without a suffix. For example, `user1`. For more information, see Authenticating by username and password.
- **Cloud users (Access Service)** тАФ the SID has the form `<subject_id>@as`, where `<subject_id>` is the user ID in IAM. The suffix is set by the `access_service_domain` parameter (default `as`). For example, `ajeb89hv69nujke769fa@as`. For more information, see Authentication through a third-party IAM provider.
- **LDAP** тАФ the SID has the form `<login>@<domain>`, where the domain is set by the `ldap_authentication_domain` parameter (default `ldap`). For example, `user1@ldap`. For more information, see LDAP directory integration.
- **External identity providers (OIDC)** тАФ the SID has the form `<login>@<domain>`, where the domain is set by the `external_idp_authentication_domain` parameter in the authentication configuration (default `sso`). For example, `user1@sso`.

You can specify either the SID of a specific user or the SID of a group. The `all-users@well-known` group is automatically added to all authenticated users тАФ it is convenient when you need to direct queries from all authenticated clients to a pool.

Example. Direct requests from user `user1@ldap` to the `olap` pool:

```
CREATE RESOURCE POOL CLASSIFIER cl_user WITH (
    RANK=100,
    RESOURCE_POOL='olap',
    MEMBER_NAME='user1@ldap'
);
```

HAS_PATH

`HAS_PATH` compares the paths of YDB objects accessed by the request against the specified mask. The mask supports wildcards: `*` тАФ any sequence of characters, `?` тАФ any single character. The predicate matches if at least one object in the request plan matches the mask.

Example. Direct requests to archive tables to the `pool_archive` pool:

```
CREATE RESOURCE POOL CLASSIFIER cl_archive WITH (
    RANK=700,
    RESOURCE_POOL='pool_archive',
    HAS_PATH='/Root/db/archive/*'
);
```

HAS_APP_NAME

`HAS_APP_NAME` compares the value against the client application identifier. The value is passed by the client in gRPC request metadata via the `x-ydb-application-name` header; YDB SDKs provide a way to set it on the client side. Comparison is exact match (no wildcards).

> **Warning**
>
> The `HAS_APP_NAME` value is set by the client and is not authenticated by the server ТАФ do not use it as an access control mechanism. For effective isolation, combine it with `MEMBER_NAME` or direct unrecognized requests to a sandbox pool with strict limits.

Setting the application identifier on the client:

- **YDB Embedded UI** ТАФ fixed value `ydb-ui`, set by the viewer and not user-configurable.
- **YDB CLI** ТАФ not supported: the client application identifier is not sent in requests.
- **YDB C++ SDK** ТАФ per request via the `Header` parameter of `TRequestSettings`: `settings.Header({{ NYdb::YDB_APPLICATION_NAME, "my-app" }})`, where the `YDB_APPLICATION_NAME` constant equals `x-ydb-application-name`.
- **YDB Go SDK** ТАФ at the driver level via the `WithApplicationName` option in the `ydb.Open` call.
- **YDB Java SDK** ТАФ at the transport level via the `GrpcTransportBuilder.withApplicationName` method.
- **YDB Python SDK** ТАФ no dedicated parameter; the value is set per request via a generic header: `settings.with_header("x-ydb-application-name", "my-app")` (the `BaseRequestSettings.with_header` method).

Example. Direct requests from the Embedded UI to the `pool_adhoc` pool:

```
CREATE RESOURCE POOL CLASSIFIER cl_adhoc_ui WITH (
    RANK=200,
    RESOURCE_POOL='pool_adhoc',
    HAS_APP_NAME='ydb-ui'
);
```

HAS_FULL_SCAN

`HAS_FULL_SCAN` identifies requests that contain a full scan of the specified objects. A full scan is a table read without a key or key-range constraint. The argument is a path mask supporting wildcards `*` and `?`; the predicate matches if there is at least one such object in the request plan. Both row-oriented and column-oriented tables are supported. Objects to which the full-scan concept does not apply (for example, topics) are not considered.

Details of full-scan detection:

- **`LIMIT` does not cancel a full scan.** Without a key condition, the physical plan processes the entire table; `LIMIT` only limits the size of the result.
- **Secondary indexes.** A full scan of a secondary index's implementation table is counted against its own path. For example, for the table

  ```
  CREATE TABLE orders (
      Id Uint64 NOT NULL,
      Status Utf8,
      PRIMARY KEY (Id),
      INDEX by_status GLOBAL ON (Status)
  );
  ```

  the index table has path `/Root/orders/by_status/indexImplTable`. The query `SELECT * FROM orders VIEW by_status` performs a full scan of the index table ТАФ the main `/Root/orders` is not scanned. Therefore:

  - `HAS_FULL_SCAN='/Root/orders'` ТАФ **will not match**;
  - `HAS_FULL_SCAN='/Root/orders/by_status/indexImplTable'` or `HAS_FULL_SCAN='/Root/orders/*'` ТАФ **will match**.

Example. Reject requests that cause a full scan of the orders archive:

```
CREATE RESOURCE POOL CLASSIFIER cl_fullscan_reject WITH (
    RANK=100,
    ACTION='reject',
    HAS_FULL_SCAN='/Root/db/orders_archive/*'
);
```

HAS_STREAM

`HAS_STREAM` determines whether a request is streaming ТАФ that is, performs a long-running continuous read and/or write over data streams. Allowed values:

- `true` ТАФ the classifier matches streaming requests;
- `false` ТАФ the classifier matches non-streaming requests.

Example. Direct streaming requests to the `pool_stream` pool:

```
CREATE RESOURCE POOL CLASSIFIER cl_stream WITH (
    RANK=500,
    RESOURCE_POOL='pool_stream',
    HAS_STREAM=true
);
```

Notes

If `RANK` is not specified in the DDL for creating a resource pool classifier, it will be assigned the default value . All `RANK` values must be unique to ensure a strictly deterministic order of resource pool selection in case of conflicting conditions. This behavior is chosen to allow adding new resource pool classifiers between existing ones.

It is also possible to have a classifier that references a non-existent resource pool or one to which the user does not have access. In that case, it is skipped.

For limitations on the number of classifiers, see the limitations page.

Permissions

The permission `USE` on the database is required.

Example of granting such a permission:

```
GRANT 'USE' ON `/my_db` TO `user1@domain`;
```

Examples

Below is a combined example that composes several classifiers and predicates: rejecting full scans of archive tables, isolating streaming requests, and dedicating a pool for interactive admin queries from the Embedded UI.

Creating resource pools:

```
CREATE RESOURCE POOL pool_stream WITH (
    TOTAL_CPU_LIMIT_PERCENT_PER_NODE=20
);

CREATE RESOURCE POOL pool_adhoc_admin WITH (
    TOTAL_CPU_LIMIT_PERCENT_PER_NODE=10
);
```

Creating classifiers:

```
-- Reject full scans of archive tables.
CREATE RESOURCE POOL CLASSIFIER cl_fullscan_reject WITH (
    RANK=100,
    ACTION='reject',
    HAS_FULL_SCAN='/Root/db/orders_archive/*'
);

-- Direct streaming requests to a dedicated pool.
CREATE RESOURCE POOL CLASSIFIER cl_stream WITH (
    RANK=200,
    RESOURCE_POOL='pool_stream',
    HAS_STREAM=true
);

-- Admin requests from the Embedded UI тАФ into the interactive-queries pool.
-- AND condition: both MEMBER_NAME and HAS_APP_NAME must match.
CREATE RESOURCE POOL CLASSIFIER cl_adhoc_admin WITH (
    RANK=300,
    RESOURCE_POOL='pool_adhoc_admin',
    MEMBER_NAME='admin',
    HAS_APP_NAME='ydb-ui'
);
```

Classifiers are processed in ascending order of `RANK`; the first matching classifier is applied to the request. A request that does not match any classifier is directed to the `default` pool.

See also

- Workload Manager — resource consumption management
- ALTER RESOURCE POOL CLASSIFIER
- DROP RESOURCE POOL CLASSIFIER

CREATE TOPIC

The `CREATE TOPIC` call creates a topic.

When creating a topic, you can add topic consumers to it and topic settings.

```
CREATE TOPIC topic_path (
    CONSUMER consumer1,
    CONSUMER consumer2 WITH (setting1 = value1)
) WITH (
    topic_setting2 = value2
);
```

All the parameters except the topic name are optional. By default, a topic is created without consumers. All the omitted settings are also set by default (both for the topic and its consumers).

> **Note**
>
> When choosing a name for the topic, please consider the common schema objects naming rules

Examples

Creating a topic without consumers with default settings:

```
CREATE TOPIC `my_topic`;
```

Creating a topic with a single consumer and the important option enabled:

```
CREATE TOPIC `my_topic` (
    CONSUMER my_consumer WITH (important = true)
);
```

Full list of available topic consumer settings

- `important`: Defines an important consumer. No data will be deleted from the topic until all the important consumers read them. Value type: `boolean`, default value: `false`.
- `read_from`: Sets up the message write time starting from which the consumer will receive data. Data written before this time will not be read. Value type: `Datetime` OR `Timestamp` OR `integer` (unix-timestamp in the numeric format). Default value: `0` (read from the earliest available message).
- `supported_codecs`: List of codecs supported by the consumer.

Creating a topic with the retention period of one day:

```
CREATE TOPIC `my_topic` WITH(
    retention_period = Interval('P1D')
);
```

Full list of available topic settings

- `min_active_partitions`: Minimum number of topic partitions. During automatic load balancing, the number of active partitions will not decrease below this value. Value type: `integer`, default value: `1`.
- `partition_count_limit`: Maximum number of active partitions in the topic. `0` is interpreted as unlimited. Value type: `integer`, default value: `0`.
- `retention_period`: Data retention period in the topic. Value type: `Interval`, default value: `24h`.
- `retention_storage_mb`: Limit on the maximum disk space occupied by the topic data. When this value is exceeded, the older data is cleared, like under a retention policy. The consumed space may exceed the set value when autopartitioning is enabled. Value type: `integer`, default value: `0` (no limit).
- `partition_write_speed_bytes_per_second`: Maximum allowed write speed per partition. If a write speed for a given partition exceeds this value, the write speed will be capped. Value type: `integer`, default value: `2097152` (2MB).
- `partition_write_burst_bytes`: Write quota allocated for write bursts. When set to zero, the actual write_burst value is equalled to the quota value (this allows write bursts of up to one second). Value type: `integer`, default value: `0`.
- `metering_mode`: Resource metering mode (`RESERVED_CAPACITY` - based on the allocated resources or `REQUEST_UNITS` - based on actual usage). This option applies to topics in serverless databases. Value type: `String`.
- `supported_codecs`: List of codecs supported by the topic. Value type: `String`.

CREATE USER

Creates a user with the specified name and password.

Syntax:

```
CREATE USER user_name [option]
```

- `user_name`: The name of the user. It may contain lowercase Latin letters and digits.
- `option` — command option:
  - `PASSWORD 'password'` — creates a user with the password `password`.
  - `PASSWORD NULL` — creates a user with an empty password (default).
  - `NOLOGIN` - disallows user login (user lockout).
  - `LOGIN` - allows user login (default).

> **Note**
>
> The scope of the commands `CREATE USER`, `ALTER USER`, and `DROP USER` does not extend to external user directories. Keep this in mind if users with third-party authentication (e.g., LDAP) are connecting to YDB. For example, the `CREATE USER` command does not create a user in the LDAP directory. Learn more about YDB's interaction with the LDAP directory.

COMMIT

By default, the entire YQL query is executed within a single transaction, and independent parts inside it are executed in parallel, if possible.
Using the `COMMIT;` keyword you can add a barrier to the execution process to delay execution of expressions that follow until all the preceding expressions have completed.

To commit in the same way automatically after each expression in the query, you can use `PRAGMA autocommit;`.

Examples

```
INSERT INTO result1 SELECT * FROM my_table;
INSERT INTO result2 SELECT * FROM my_table;
COMMIT;
-- result2 will already include the SELECT contents from the second line:
INSERT INTO result3 SELECT * FROM result2;
```

DECLARE

Declares a typed named expression whose value will be passed separately from the query text. With parameterization, you can separately develop an analytical solution and then launch it sequentially with different input values.

In the case of transactional load, parameters let you avoid recompilation of queries when repeating calls of the same type. This way you can reduce server utilization and exclude compilation time from the total time of query execution.

Passing of parameters is supported in the SDK, CLI, and graphical interfaces.

Syntax

```
DECLARE $named-node AS data_type;
```

1. `DECLARE` keyword.
2. `$named-node`: The name by which you can access the passed value. It must start with `$`.
3. `AS` keyword.
4. `data_type` is the data type represented as a string in the accepted format.

Only serializable data types are allowed:

- Primitive types.
- Optional types.
- Containers, except `Stream<Type>`.
- `Void` and `Null` are the supported special types.

Example

```
DECLARE $x AS String;
DECLARE $y AS String?;
DECLARE $z AS List<String>;

SELECT $x, $y, $z;
```

DELETE FROM

> **Warning**
>
> Currently, mixing column-oriented tables and row-oriented tables in a single transaction is supported only if the transaction performs read operations; no writes are allowed. Support for read-write transactions involving both table types is under development.
>
> If a write transaction includes both types of tables, it fails with the following error: `Write transactions that use both row-oriented and column-oriented tables are disabled at current time`.

Deletes rows that match the `WHERE` clause, from the table.

Example

```
DELETE FROM my_table
WHERE Key1 == 1 AND Key2 >= "One";
```

DELETE FROM ... ON

Deletes rows based on the results of a subquery. The set of columns returned by the subquery must be a subset of the table's columns being updated, and all columns of the table's primary key must be present in the returned columns. The data types of the columns returned by the subquery must match the data types of the corresponding columns in the table.

The primary key value is used to search for rows to be deleted from the table. The presence of other (non-key) columns of the table in the output of the subquery does not affect the results of the deletion operation.

Example

```
$to_delete = (
    SELECT Key, SubKey FROM my_table WHERE Value = "ToDelete" LIMIT 100
);

DELETE FROM my_table ON
SELECT * FROM $to_delete;
```

DELETE FROM ... RETURNING

Deletes rows and returns their values in a single operation. It allows to retrieve data from the rows being deleted without needing to perform a separate SELECT query beforehand.

Examples

- Return all values of modified rows

```
DELETE FROM orders
WHERE status = 'cancelled'
RETURNING *;
```

- Return specific columns

```
DELETE FROM orders
WHERE status = 'cancelled'
RETURNING order_id, order_date;
```

See also

- BATCH DELETE

DISCARD

Calculates `SELECT` without returning the result neither to the client or table.

It's good to combine it with `Ensure` to check the final calculation result against the user's criteria.

A query with `DISCARD` is executed in full — with all its filters, aggregations, and `Ensure` checks — but the result set is not returned to the client. The data is not sent over the network either: the client receives only the query execution status. For large outputs, this saves a noticeable amount of traffic and memory.

In a query consisting of multiple statements, `DISCARD` applies to an individual statement:

```
SELECT 1;          -- returned
DISCARD SELECT 2;  -- executed, but not included in the response
SELECT 3;          -- returned
```

The client will receive two result sets — from the first and third statements.

`DISCARD` applies only to a statement as a whole and cannot be used inside expressions. In particular, it is not allowed:

- in subqueries — `SELECT * FROM (DISCARD SELECT 1)`;
- in `WHERE ... IN (...)` — `SELECT * FROM my_table WHERE Key IN (DISCARD SELECT 1)`;
- in `UNION ALL` operands — `SELECT 1 UNION ALL DISCARD SELECT 2`.

Note

`DISCARD` is supported for queries executed via the Query Service starting from YDB version 26.2. When a query is executed via the legacy interfaces (Table Service, scan queries), `DISCARD` is ignored and the result is returned to the client — this behavior is preserved for backward compatibility.

Examples

```
DISCARD SELECT 1;
```

```
DISCARD SELECT Ensure(
    Data,
    Data < 1000000,
    "Value too big"
) FROM `result_table`;
```

If the `Ensure` condition is violated, the query fails with an error. Otherwise, the query completes successfully, and the client does not receive the result set that would otherwise have to be downloaded in full.

DROP ASYNC REPLICATION

The `DROP ASYNC REPLICATION` statement deletes an asynchronous replication instance. When an asynchronous replication instance is deleted, the following objects are also deleted:

- automatically created streams of changes
- replicas (optionally)

Syntax

```
DROP ASYNC REPLICATION <name> [CASCADE]
```

Parameters

- `name` — the name of the asynchronous replication instance.
- `CASCADE` — cascaded deletion of the replicas that were created for a given asynchronous replication instance.

Examples

This section contains examples of YQL statements that drop the asynchronous replication instance created with the following expression:

```
CREATE ASYNC REPLICATION my_replication
FOR original_table AS replica_table
WITH (
    CONNECTION_STRING = 'grpcs://example.com:2135/?database=/Root/another_database',
    TOKEN_SECRET_PATH = 'my_secret'
);
```

The following statement drops an asynchronous replication instance and the automatically created stream of changes for the `original_table` table, but the `replica_table` table is not deleted:

```
DROP ASYNC REPLICATION my_replication;
```

The following statement drops an asynchronous replication instance, the automatically created stream of changes for the `original_table` table, and the `replica_table` table:

```
DROP ASYNC REPLICATION my_replication CASCADE;
```

See also

- CREATE ASYNC REPLICATION
- ALTER ASYNC REPLICATION

DROP BACKUP COLLECTION

The `DROP BACKUP COLLECTION` statement deletes a backup collection and all its backups.

```
DROP BACKUP COLLECTION collection_name;
```

Parameters

- `collection_name` — name of the backup collection to drop.

> **Warning**
>
> This operation permanently deletes the backup collection and all backups it contains from the cluster. This action cannot be undone.

> **Note**
>
> Dropping a backup collection only affects cluster-stored data. Any backups previously exported to external storage (S3 or filesystem) are not affected and remain available for import.

Examples

Dropping a backup collection:

```
DROP BACKUP COLLECTION old_backups;
```

See also

- Backup collections.
- CREATE BACKUP COLLECTION.
- BACKUP.
- RESTORE.

DROP GROUP

Deletes the specified group. You can list multiple groups under one operator.

Syntax:

```
DROP GROUP [ IF EXISTS ] group_name [, ...]
```

- `IF EXISTS`: Suppress an error if the group doesn't exist.
- `group_name`: The name of the group to be deleted.

DROP EXTERNAL DATA SOURCE

Deletes the specified external data source.

If no external data source with that name exists, an error is returned.

Example

```
DROP EXTERNAL DATA SOURCE my_external_data_source;
```

DROP EXTERNAL TABLE

Deletes the specified external table.

If no external table with that name exists, an error is returned.

Example

```
DROP EXTERNAL TABLE my_table;
```

DROP STREAMING QUERY

`DROP STREAMING QUERY` deletes a streaming query.

Syntax

```
DROP STREAMING QUERY [IF EXISTS] <query_name>
```

Parameters

- `IF EXISTS` — do not fail if the streaming query does not exist.
- `query_name` — name of the streaming query to delete.

Permissions

Requires permission `REMOVE SCHEMA` on the streaming query. Example grant for `my_streaming_query`:

```
GRANT REMOVE SCHEMA ON my_streaming_query TO `user@domain`
```

Examples

Delete `my_streaming_query`:

```
DROP STREAMING QUERY my_streaming_query
```

See also

- Streaming queries
- CREATE STREAMING QUERY
- ALTER STREAMING QUERY

DROP OBJECT (TYPE SECRET)

> **Alert**
>
> **This command is deprecated** and will be removed in future versions of YDB. The recommended syntax for working with secrets is described in the Secrets section.

Deletes the specified secret.

If no secret with that name exists, an error is returned.

Example

```
DROP OBJECT my_secret (TYPE SECRET);
```

DROP OBJECT (TYPE SECRET_ACCESS)

> **Alert**
>
> **This command is deprecated** and will be removed in future versions of YDB. The recommended syntax for working with secrets is described in the Secrets section.

Deletes the specified access rule for a secret.

If no rule with that name exists, an error is returned.

Example

```
DROP OBJECT (TYPE SECRET_ACCESS) `MySecretName:another_user`;
```

DROP TABLE

Deletes the specified table.

If there is no such table, an error is returned.

Examples

```
DROP TABLE my_table;
```

DROP SECRET

The `DROP SECRET` statement deletes an existing secret.

Syntax

```
DROP SECRET secret_name
```

- `secret_name` — the name of the secret to delete.

Permissions

Deleting a secret requires the rights `REMOVE SCHEMA` and `ALTER SCHEMA`.

Examples

Delete the secret named `secret_name`:

```
DROP SECRET secret_name;
```

See also

- CREATE SECRET
- ALTER SECRET

DROP TRANSFER

The `DROP TRANSFER` statement deletes a transfer instance. If a consumer was created automatically when the transfer was created, it is also deleted. The system will keep trying to delete the consumer until the operation is successful.

The `DROP TRANSFER` statement does not delete the destination table or the source topic.

Syntax

```
DROP TRANSFER <name>
```

where:

- `name` — the name of the transfer instance.

Permissions

The following permissions are required to delete a transfer:

- `REMOVE SCHEMA` — to delete the transfer instance;
- `ALTER SCHEMA` — to delete the automatically created topic consumer (if applicable).

Examples

The following query deletes the transfer named `my_transfer`:

```
DROP TRANSFER my_transfer;
```

See Also

- CREATE TRANSFER
- ALTER TRANSFER
- Data transfer

DROP TOPIC

`DROP TOPIC` deletes the specified topic.

Syntax

```
DROP TOPIC <topic_path>;
```

Examples

The following command will delete the topic named `my_topic`:

```
DROP TOPIC my_topic;
```

See also

- CREATE TOPIC
- ALTER TOPIC

DROP VIEW

`DROP VIEW` deletes an existing view.

Syntax

```
DROP VIEW [IF EXISTS] <name>
```

Parameters

- `IF EXISTS` - when specified, the statement does not return an error if a view with the given name does not exist.
- `name` - the name of the view to be deleted.

Examples

The following command will drop the view named `recent_series`:

```
DROP VIEW recent_series;
```

See also

- CREATE VIEW
- ALTER VIEW

DROP RESOURCE POOL

`DROP RESOURCE POOL` removes a resource pool.

Syntax

```
DROP RESOURCE POOL <name>
```

Parameters

- `name` — name of the resource pool to drop.

Permissions

The `REMOVE SCHEMA` permission on the pool under `.metadata/workload_manager/pools` is required. Example:

```
GRANT 'REMOVE SCHEMA' ON `.metadata/workload_manager/pools` TO `user1@domain`;
```

Examples

The following removes the resource pool named `olap`:

```
DROP RESOURCE POOL olap;
```

See also

- Workload Manager — resource consumption management
- CREATE RESOURCE POOL
- ALTER RESOURCE POOL

DROP RESOURCE POOL CLASSIFIER

`DROP RESOURCE POOL CLASSIFIER` removes a resource pool classifier.

Syntax

```
DROP RESOURCE POOL CLASSIFIER <name>
```

Parameters

- `name` — name of the resource pool classifier to drop.

Permissions

The `ALL` permission on the database is required. Example:

```
GRANT 'ALL' ON `/my_db` TO `user1@domain`;
```

Examples

The following removes the classifier named `olap_classifier`:

```
DROP RESOURCE POOL CLASSIFIER olap_classifier;
```

See also

- Workload Manager — resource consumption management
- CREATE RESOURCE POOL CLASSIFIER
- ALTER RESOURCE POOL CLASSIFIER

DROP USER

Deletes the specified user. You can list multiple users under one operator.

Syntax:

```
DROP USER [ IF EXISTS ] user_name [, ...]
```

- `IF EXISTS` : Suppress an error if the user doesn't exist.
- `user_name` : The name of the user to be deleted. It also supports the ability to set a comma-separated list of users, for example: `DROP USER user1, user2, user3;`

GRANT

The `GRANT` command allows setting access rights to schema objects for a user or group of users.

Syntax:

```
GRANT {{permission_name} [, ...] | ALL [PRIVILEGES]} ON {path_to_scheme_object [, ...]} TO {role_name [,
...]} [WITH GRANT OPTION]
```

- `permission_name` - the name of the access right to schema objects that needs to be assigned.
- `path_to_scheme_object` - the path to the schema object to which rights are granted.
- `role_name` - the name of the user or group for which rights to the schema object are granted.

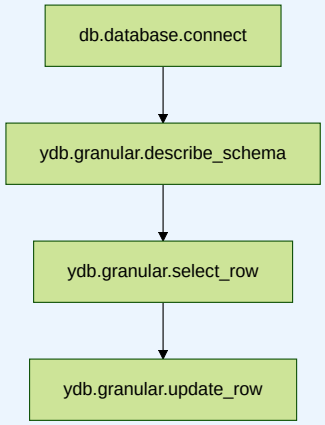
`WITH GRANT OPTION` - using this clause gives the user or group the right to manage access rights - to grant or revoke specific rights. The clause functions similarly to granting the `"ydb.access.grant"` right or `GRANT`.
A subject with the `ydb.access.grant` right cannot grant rights broader than they themselves have on the access object `path_to_scheme_object`.

Access rights

As names of access rights, you can use either the names of YDB rights or the corresponding YQL keywords.
The possible names of rights are listed in the table below.

YDB right	YQL keyword	Description
Database-level rights		
`ydb.database.connect`	`CONNECT`	The right to connect to a database
`ydb.database.create`	`CREATE`	The right to create new databases in the cluster
`ydb.database.drop`	`DROP`	The right to delete databases in the cluster
Elementary rights for database objects		
`ydb.granular.select_row`	`SELECT ROW`	The right to read rows from a table (select), read messages from topics, use secret values
`ydb.granular.update_row`	`UPDATE ROW`	The right to update rows in a table (insert, update, upsert, replace), write messages to topics
`ydb.granular.erase_row`	`ERASE ROW`	The right to delete rows from a table (delete)
`ydb.granular.create_directory`	`CREATE DIRECTORY`	The right to create and delete directories, including existing and nested ones
`ydb.granular.create_table`	`CREATE TABLE`	The right to create tables (including index, external, columnar), views, sequences
`ydb.granular.create_queue`	`CREATE QUEUE`	The right to create topics
`ydb.granular.remove_schema`	`REMOVE SCHEMA`	The right to delete objects (directories, tables, topics) that were created using rights
`ydb.granular.describe_schema`	`DESCRIBE SCHEMA`	The right to view existing access rights (ACL) on an access object, view descriptions of access objects (directories, tables, topics)
`ydb.granular.alter_schema`	`ALTER SCHEMA`	The right to modify access objects (directories, tables, topics), including users' rights to access objects
Additional flags		
`ydb.access.grant`	`GRANT`	The right to grant or revoke rights from other users to the extent not exceeding the current scope of the user's rights on the access object
`ydb.tables.modify`	`MODIFY TABLES`	`ydb.granular.update_row` + `ydb.granular.erase_row`
`ydb.tables.read`	`SELECT TABLES`	Alias for `ydb.granular.select_row`
`ydb.generic.list`	`LIST`	Alias for `ydb.granular.describe_schema`
`ydb.generic.read`	`SELECT`	`ydb.granular.select_row` + `ydb.generic.list`
`ydb.generic.write`	`INSERT`	`ydb.granular.update_row` + `ydb.granular.erase_row` + `ydb.granular.create_directory` + `ydb.granular.create_table` + `ydb.granular.create_queue` + `ydb.granular.remove_schema` + `ydb.granular.alter_schema`
`ydb.generic.use_legacy`	`USE LEGACY`	`ydb.generic.read` + `ydb.generic.write` + `ydb.access.grant`
`ydb.generic.use`	`USE`	`ydb.generic.use_legacy` + `ydb.database.connect`
`ydb.generic.manage`	`MANAGE`	`ydb.database.create` + `ydb.database.drop`
`ydb.generic.full_legacy`	`FULL LEGACY`	`ydb.generic.use_legacy` + `ydb.generic.manage`
`ydb.generic.full`	`FULL`	`ydb.generic.use` + `ydb.generic.manage`

- `ALL [PRIVILEGES]` is used to specify all possible rights on schema objects for users or groups. `PRIVILEGES` is an optional keyword needed for compatibility with the SQL standard.

> **Note**
>
> Rights `ydb.database.connect`, `ydb.granular.describe_schema`, `ydb.granular.select_row`, and `ydb.granular.update_row` should be considered as layers of rights.
>
> 
>
>
> For example, to update rows, you need not only the right `ydb.granular.update_row`, but also all the overlying rights.

Examples

- Assign the `ydb.generic.read` right to the table `/shop_db/orders` for the user `user1`:

```
GRANT 'ydb.generic.read' ON `/shop_db/orders` TO user1;
```

  The same command, using the keyword:

```
GRANT SELECT ON `/shop_db/orders` TO user1;
```

- Assign the rights `ydb.database.connect` and `ydb.generic.list` to the root of the database `/shop_db` for user `user2` and group `group1`:

```
GRANT LIST, CONNECT ON `/shop_db` TO user2, group1;
```

- Assign the `ydb.generic.use` right to the tables `/shop_db/orders` and `/shop_db/sellers` for users `user1@domain` and `user2@domain`:

```
GRANT 'ydb.generic.use' ON `/shop_db/orders`, `/shop_db/sellers` TO `user1@domain`, `user2@domain`;
```

- Grant all rights to the table `/shop_db/sellers` for the user `admin_user`:

```
GRANT ALL ON `/shop_db/sellers` TO admin_user;
```

INSERT INTO

> **Warning**
>
> Currently, mixing column-oriented tables and row-oriented tables in a single transaction is supported only if the transaction performs read operations; no writes are allowed. Support for read-write transactions involving both table types is under development.
>
> If a write transaction includes both types of tables, it fails with the following error: `Write transactions that use both row-oriented and column-oriented tables are disabled at current time`.

Adds rows to the table. If you try to insert a row into a table with an existing primary key value, the operation fails with the `PRECONDITION_FAILED` error code and the `Operation aborted due to constraint violation: insert_pk` message returned.

`INSERT INTO` lets you perform the following operations:

- Adding constant values using `VALUES`.

```
INSERT INTO my_table (Key1, Key2, Value1, Value2)
VALUES (345987,'ydb', 'Pied piper', 1414);
COMMIT;
```

```
INSERT INTO my_table (key, value)
VALUES ("foo", 1), ("bar", 2);
```

- Saving the `SELECT` result.

```
INSERT INTO my_table
SELECT Key AS Key1, "Empty" AS Key2, Value AS Value1
FROM my_table1;
```

When working with external file data sources, you can specify additional parameters:

- `FORMAT` — stored data format in file storage for federated queries. Allowed values: `csv_with_names`, `tsv_with_names`, `json_list`, `json_each_row`, `json_as_string`, `parquet`, `raw`.
- `COMPRESSION` — file compression in file storage for federated queries. Allowed values: gzip, zstd, lz4, brotli, bzip2, xz.
- `PARTITIONED_BY` — list of partition columns for data in file storage in federated queries. Lists columns in the order they appear in the file layout.
- `projection.enabled` — flag to enable extended data partitioning. Allowed values: `true`, `false`.
- `projection.<field_name>.type` — field type for extended data partitioning. Allowed values: `integer`, `enum`, `date`.
- `projection.<field_name>.<options>` — extended properties of a field for extended data partitioning.

Example

```
INSERT INTO `connection`.`test/`
WITH
(
  FORMAT = "csv_with_names"
)
SELECT
    "value" AS value, "name" AS name
```

Where:

- `connection` — name of the connection to S3 (Yandex Object Storage).
- `test/` — path inside the bucket where data is written. Files are created with random names.

INSERT INTO ... RETURNING

Inserts rows and returns their values in a single operation. It allows to retrieve data from the rows being inserted without needing to perform a separate SELECT query afterwards.

Examples

- Return all values of modified rows

```
INSERT INTO some_table (id, year, color, price)
VALUES (1103, 2023, 'blue', 400)
RETURNING *;
```

- Return specific columns

```
INSERT INTO some_table (id, color, price)
VALUES
    (1101, 'red', 200),
    (1102, 'green', 300)
RETURNING id, price;
```

INTO RESULT

Lets you set a custom label for SELECT.

Examples

```
SELECT 1 INTO RESULT foo;
```

```
SELECT * FROM
my_table
WHERE value % 2 == 0
INTO RESULT `Result name`;
```

PRAGMA

Definition

Redefinition of settings.

Syntax

`PRAGMA x.y = "z";` or `PRAGMA x.y("z", "z2", "z3");`:

- `x`: (optional) The category of the setting.
- `y`: The name of the setting.
- `z`: (optional for flags) The value of the setting. The following suffixes are acceptable:
  - `Kb`, `Mb`, `Gb`: For the data amounts.
  - `sec`, `min`, `h`, `d`: For the time values.

Examples

```
PRAGMA AutoCommit;
```

```
PRAGMA TablePathPrefix = "home/yql";
```

```
PRAGMA Warning("disable", "1101");
```

With some exceptions, you can return the settings values to their default states using `PRAGMA my_pragma = default;`.

For the full list of available settings, see the table below.

Scope

Unless otherwise specified, a pragma affects all the subsequent expressions up to the end of the module where it's used. If necessary and logically possible, you can change the value of this setting several times within a given query to make it different at different execution steps.

There are also special scoped pragmas with the scope defined by the same rules as the scope of named expressions. Unlike scoped pragmas, regular pragmas can only be used in the global scope (not inside lambda functions, `ACTION`, etc.).

Global

AutoCommit

Value type	Default
Flag	false

Automatically run COMMIT after every statement.

TablePathPrefix

Value type	Default
String	—

Add the specified prefix to the cluster table paths. It uses standard file system path concatenation, supporting parent folder `..` referencing and requiring no trailing slash. For example,

`PRAGMA TablePathPrefix = "home/yql"; SELECT * FROM test;`

The prefix is not added if the table name is an absolute path (starts with /).

UseTablePrefixForEach

Value type	Default
Flag	false

EACH uses TablePathPrefix for each list item.

Warning

Value type	Default
1. Action 2. Warning code or "*"	—

Action:

- `disable`: Disable.
- `error`: Treat as an error.
- `default`: Revert to the default behavior.

The warning code is returned with the text itself (it's displayed on the right side of the web interface).

Example

```
PRAGMA Warning("error", "*");
PRAGMA Warning("disable", "1101");
PRAGMA Warning("default", "4503");
```

In this case, all the warnings are treated as errors, except for the warning `1101` (that will be disabled) and `4503` (that will be processed by default, that is, remain a warning). Since warnings may be added in new YQL releases, use `PRAGMA Warning("error", "*");` with caution (at least cover such queries with autotests).

SimpleColumns

`SimpleColumns` / `DisableSimpleColumns`

Value type	Default
Flag	true

When you use `SELECT foo.* FROM ... AS foo`, remove the `foo.` prefix from the names of the result columns.

It can be also used with a JOIN, but in this case it may fail in the case of a name conflict (that can be resolved by using WITHOUT and renaming columns). For JOIN in SimpleColumns mode, an implicit Coalesce is applied to the key columns.

CoalesceJoinKeysOnQualifiedAll

`CoalesceJoinKeysOnQualifiedAll` / `DisableCoalesceJoinKeysOnQualifiedAll`

Value type	Default
Flag	true

Controls implicit Coalesce for the key `JOIN` columns in the SimpleColumns mode. If the flag is set, the Coalesce is made for key columns if there is at least one expression in the format `foo.*` or `*` in SELECT: for example, `SELECT a.* FROM T1 AS a JOIN T2 AS b USING(key)`. If the flag is not set, then Coalesce for JOIN keys is made only if there is an asterisk '*' after `SELECT`

StrictJoinKeyTypes

`StrictJoinKeyTypes` / `DisableStrictJoinKeyTypes`

Value type	Default
Flag	false

If the flag is set, then JOIN will require strict matching of key types.
By default, JOIN preconverts keys to a shared type, which might result in performance degradation.
StrictJoinKeyTypes is a scoped setting.

AnsiInForEmptyOrNullableItemsCollections

Value type	Default
Flag	false

This pragma brings the behavior of the `IN` operator in accordance with the standard when there's `NULL` in the left or right side of `IN`. The behavior of `IN` when on the right side there is a Tuple with elements of different types also changed. Examples:

```
1 IN (2, 3, NULL) = NULL (was Just(False))
NULL IN () = Just(False) (was NULL)
(1, null) IN ((2, 2), (3, 3)) = Just(False) (was NULL)
```

For more information about the `IN` behavior when operands include `NULL`s, see here. You can explicitly select the old behavior by specifying the pragma `DisableAnsiInForEmptyOrNullableItemsCollections`. If no pragma is set, then a warning is issued and the old version works.

AnsiRankForNullableKeys

Value type	Default
Flag	false

Aligns the RANK/DENSE_RANK behavior with the standard if there are optional types in the window sort keys or in the argument of such window functions. It means that:

- The result type is always Uint64 rather than Uint64?.
- NULLs in keys are treated as equal to each other (the current implementation returns NULL).

You can explicitly select the old behavior by using the `DisableAnsiRankForNullableKeys` pragma. If no pragma is set, then a warning is issued and the old version works.

AnsiCurrentRow

Value type	Default
Flag	false

Aligns the implicit setting of a window frame with the standard if there is ORDER BY.
If AnsiCurrentRow is not set, then the `(ORDER BY key)` window is equivalent to `(ORDER BY key ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)`.
The standard also requires that this window behave as `(ORDER BY key RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)`.
The difference is in how `CURRENT ROW` is interpreted. In `ROWS` mode `CURRENT ROW` is interpreted literally: the current row in a

partition.
In `RANGE` mode, the end of the `CURRENT ROW` frame means "the last row in a partition with a sort key equal to the current row".

AnsiOrderByLimitInUnionAll

Value type	Default
Flag	false

Aligns the UNION ALL behavior with the standard if there is `ORDER BY/LIMIT/DISCARD/INSERT INTO` in the combined subqueries. It means that:

- `ORDER BY/LIMIT/INSERT INTO` are allowed only after the last subquery.
- `DISCARD` is allowed only before the first subquery.
- The specified operators apply to the `UNION ALL` result (unlike the current behavior when they apply only to the subquery).
- To apply an operator to a subquery, enclose the subquery in parentheses.

You can explicitly select the old behavior by using the `DisableAnsiOrderByLimitInUnionAll` pragma. If no pragma is set, then a warning is issued and the old version works.

OrderedColumns

`OrderedColumns` / `DisableOrderedColumns`

Output the column order in SELECT/JOIN/UNION ALL and preserve it when writing the results. The order of columns is undefined by default.

PositionalUnionAll

Enable the standard column-by-column execution for UNION ALL. This automatically enables ordered columns.

RegexUseRe2

Value type	Default
Flag	false

Use Re2 UDF instead of Pcre to execute SQL the `REGEX`, `MATCH`, `RLIKE` statements. Re2 UDF can properly handle Unicode characters, unlike the default Pcre UDF.

ClassicDivision

Value type	Default
Flag	true

In the classical version, the result of integer division remains integer (by default).
If disabled, the result is always Double.
ClassicDivision is a scoped setting.

UnicodeLiterals

`UnicodeLiterals` / `DisableUnicodeLiterals`

Value type	Default
Flag	false

When this mode is enabled, string literals without suffixes like "foo"/'bar'/@@multiline@@ will be of type `Utf8`, when disabled - `String`.
UnicodeLiterals is a scoped setting.

WarnUntypedStringLiterals

`WarnUntypedStringLiterals` / `DisableWarnUntypedStringLiterals`

Value type	Default
Flag	false

When this mode is enabled, a warning will be generated for string literals without suffixes like "foo"/'bar'/@@multiline@@. It can be suppressed by explicitly choosing the suffix `s` for the `String` type, or `u` for the `Utf8` type.
WarnUntypedStringLiterals is a scoped setting.

AllowDotInAlias

Value type	Default
Flag	false

Enable dot in names of result columns. This behavior is disabled by default, since the further use of such columns in JOIN is not fully implemented.

WarnUnnamedColumns

Value type	Default
Flag	false

Generate a warning if a column name was automatically generated for an unnamed expression in `SELECT` (in the format `column[0-9]+`).

GroupByLimit

Value type	Default
Positive number	32

Increasing the limit on the number of dimensions in GROUP BY.

GroupByCubeLimit

Value type	Default
Positive number	5

Increasing the limit on the number of dimensions in GROUP BY.

Use this option with care, because the computational complexity of the query grows exponentially with the number of dimensions.

Yson

Managing the default behavior of Yson UDF, for more information, see the documentation and, in particular, Yson::Options.

`yson.AutoConvert`

Value type	Default
Flag	false

Automatic conversion of values to the required data type in all Yson UDF calls, including implicit calls.

`yson.Strict`

Value type	Default
Flag	true

Strict mode control in all Yson UDF calls, including implicit calls. If the value is omitted or is `"true"`, it enables the strict mode. If the value is `"false"`, it disables the strict mode.

`yson.DisableStrict`

Value type	Default
Flag	false

An inverted version of `yson.Strict`. If the value is omitted or is `"true"`, it disables the strict mode. If the value is `"false"`, it enables the strict mode.

REPLACE INTO

Warning

Currently, mixing column-oriented tables and row-oriented tables in a single transaction is supported only if the transaction performs read operations; no writes are allowed. Support for read-write transactions involving both table types is under development.

If a write transaction includes both types of tables, it fails with the following error: `Write transactions that use both row-oriented and column-oriented tables are disabled at current time`.

Saves data to a table, overwriting the rows based on the primary key. If the given primary key is missing, a new row is added to the table. If the given `PRIMARY_KEY` exists, the row is overwritten. The values of columns not involved in the operation are replaced by their default values.

Note

Unlike `INSERT INTO` and `UPDATE`, the queries `UPSERT INTO` and `REPLACE INTO` don't need to pre-fetch the data, hence they run faster.

Examples

- Setting values for `REPLACE INTO` using `VALUES`.

```
REPLACE INTO my_table (Key1, Key2, Value2) VALUES
    (1u, "One", 101),
    (2u, "Two", 102);
COMMIT;
```

- Fetching values for `REPLACE INTO` using a `SELECT`.

```
REPLACE INTO my_table
SELECT Key AS Key1, "Empty" AS Key2, Value AS Value1
FROM my_table1;
COMMIT;
```

REVOKE

The `REVOKE` command allows revoking access rights to schema objects for users or groups of users.

Syntax:

```
REVOKE [GRANT OPTION FOR] {{permission_name} [, ...] | ALL [PRIVILEGES]} ON {path_to_scheme_object [, ...]} F
ROM {role_name [, ...]}
```

- `permission_name` - the name of the access right to schema objects that needs to be revoked.
- `path_to_scheme_object` - the path to the schema object from which rights are revoked.
- `role_name` - the name of the user or group from which rights to the schema object are revoked.

`GRANT OPTION FOR` - using this clause revokes the right to manage access rights from the user or group. All rights previously granted by this user remain in effect. The clause functions similarly to revoking the `"ydb.access.grant"` right or `GRANT`.

Access rights

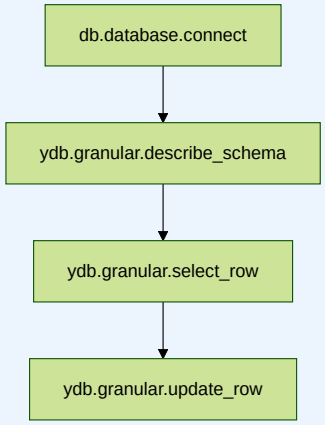
As names of access rights, you can use either the names of YDB rights or the corresponding YQL keywords.
The possible names of rights are listed in the table below.

YDB right	YQL keyword	Description
Database-level rights		
`ydb.database.connect`	`CONNECT`	The right to connect to a database
`ydb.database.create`	`CREATE`	The right to create new databases in the cluster
`ydb.database.drop`	`DROP`	The right to delete databases in the cluster
Elementary rights for database objects		
`ydb.granular.select_row`	`SELECT ROW`	The right to read rows from a table (select), read messages from topics, use secret values
`ydb.granular.update_row`	`UPDATE ROW`	The right to update rows in a table (insert, update, upsert, replace), write messages to topics
`ydb.granular.erase_row`	`ERASE ROW`	The right to delete rows from a table (delete)
`ydb.granular.create_directory`	`CREATE DIRECTORY`	The right to create and delete directories, including existing and nested ones
`ydb.granular.create_table`	`CREATE TABLE`	The right to create tables (including index, external, columnar), views, sequences
`ydb.granular.create_queue`	`CREATE QUEUE`	The right to create topics
`ydb.granular.remove_schema`	`REMOVE SCHEMA`	The right to delete objects (directories, tables, topics) that were created using rights
`ydb.granular.describe_schema`	`DESCRIBE SCHEMA`	The right to view existing access rights (ACL) on an access object, view descriptions of access objects (directories, tables, topics)
`ydb.granular.alter_schema`	`ALTER SCHEMA`	The right to modify access objects (directories, tables, topics), including users' rights to access objects
Additional flags		
`ydb.access.grant`	`GRANT`	The right to grant or revoke rights from other users to the extent not exceeding the current scope of the user's rights on the access object
`ydb.tables.modify`	`MODIFY TABLES`	`ydb.granular.update_row` + `ydb.granular.erase_row`
`ydb.tables.read`	`SELECT TABLES`	Alias for `ydb.granular.select_row`
`ydb.generic.list`	`LIST`	Alias for `ydb.granular.describe_schema`
`ydb.generic.read`	`SELECT`	`ydb.granular.select_row` + `ydb.generic.list`
`ydb.generic.write`	`INSERT`	`ydb.granular.update_row` + `ydb.granular.erase_row` + `ydb.granular.create_directory` + `ydb.granular.create_table` + `ydb.granular.create_queue` + `ydb.granular.remove_schema` + `ydb.granular.alter_schema`
`ydb.generic.use_legacy`	`USE LEGACY`	`ydb.generic.read` + `ydb.generic.write` + `ydb.access.grant`
`ydb.generic.use`	`USE`	`ydb.generic.use_legacy` + `ydb.database.connect`
`ydb.generic.manage`	`MANAGE`	`ydb.database.create` + `ydb.database.drop`
`ydb.generic.full_legacy`	`FULL LEGACY`	`ydb.generic.use_legacy` + `ydb.generic.manage`
`ydb.generic.full`	`FULL`	`ydb.generic.use` + `ydb.generic.manage`

- `ALL [PRIVILEGES]` is used to specify all possible rights on schema objects for users or groups. `PRIVILEGES` is an optional keyword needed for compatibility with the SQL standard.

Note

Rights `ydb.database.connect`, `ydb.granular.describe_schema`, `ydb.granular.select_row`, and `ydb.granular.update_row` should be considered as layers of rights.



For example, to update rows, you need not only the right `ydb.granular.update_row`, but also all the overlying rights.

Examples

- Revoke the `ydb.generic.read` right on the table `/shop_db/orders` from user `user1`:

  ```
  REVOKE 'ydb.generic.read' ON `/shop_db/orders` FROM user1;
  ```

  The same command, using the keyword:

  ```
  REVOKE SELECT ON `/shop_db/orders` FROM user1;
  ```

- Revoke the rights `ydb.database.connect`, `ydb.generic.list` on the root of the database `/shop_db` from user `user2` and group `group1`:

  ```
  REVOKE LIST, CONNECT ON `/shop_db` FROM user2, group1;
  ```

- Revoke the `ydb.generic.use` right on the tables `/shop_db/orders` and `/shop_db/sellers` from users `user1@domain` and `user2@domain`:

  ```
  REVOKE 'ydb.generic.use' ON `/shop_db/orders`, `/shop_db/sellers` FROM `user1@domain`, `user2@domain`;
  ```

- Revoke all rights on the table `/shop_db/sellers` from user `user`:

  ```
  REVOKE ALL ON `/shop_db/sellers` FROM user;
  ```

RESTORE

The `RESTORE` statement restores tables from a backup collection.

```
RESTORE collection_name;
```

Parameters

- `collection_name` — name of the backup collection to restore from.

Restore behavior

The restore operation:

- Finds the **current chain** of backups in the collection (the latest full backup followed by its incremental backups).
- Restores tables to the state of the **most recent** backup in that chain: applies the full backup and all subsequent incrementals in sequence.
- Fails if any of the tables being restored already exists at the same path.

Restoring to an arbitrary point "between" two saved backups is not supported: the result is always the state captured by one of the backups in the chain.

> **Warning**
>
> The restore operation fails if any of the tables being restored already exists at the same path. Rename or drop the conflicting tables before restoring.

Examples

Restoring from a backup collection:

```
-- Restore all tables from the collection
RESTORE production_backups;
```

Monitoring restore operations

Restore operations run asynchronously in the background. You can monitor their progress using YDB CLI:

```
# List backup and restore operations
ydb operation list incbackup

# Get operation details
ydb operation get <operation-id>
```

See also

- Backup collections
- CREATE BACKUP COLLECTION
- BACKUP
- DROP BACKUP COLLECTION
- Backup and recovery guide

SELECT

Returns the result of evaluating the expressions specified after `SELECT`.

Can be used in combination with other operations to achieve a different effect.

Examples

```
SELECT "Hello, world!";
```

```
SELECT 2 + 2;
```

Procedure for executing SELECT

The result of the `SELECT` query is computed as follows:

- the set of input tables is determined: expressions after FROM are evaluated
- is computed SAMPLE / TABLESAMPLE
- FLATTEN COLUMNS or FLATTEN BY is performed; aliases specified in `FLATTEN BY` become visible after this point.
- All JOIN are executed.
- columns specified in GROUP BY ... AS ... are added (or replaced) to the resulting data
- the WHERE clause is executed: all data that does not satisfy the predicate is filtered out.
- GROUP BY is performed, aggregate function values are computed.
- Filtering is performed using HAVING
- Values of window functions are computed
- Expressions in `SELECT` are evaluated.
- expressions in `SELECT` are assigned names defined by aliases.
- a top-level DISTINCT is applied to the columns obtained in this way
- All subqueries in UNION ALL are computed in the same way and combined (see PRAGMA AnsiOrderByLimitInUnionAll).
- Sorting is performed according to ORDER BY
- OFFSET and LIMIT are applied to the result.

Column order in YQL

In standard SQL, the order of columns specified in the projection (in `SELECT`) matters. Besides the fact that the column order must be preserved when displaying query results or when writing to a new table, some SQL constructs use this order. This applies in particular to UNION ALL and positional ORDER BY (ORDER BY ordinal).

By default, the order of columns is ignored in YQL:

- the order of columns in output tables and in query results is undefined
- The data schema of the `UNION ALL` result is output by column names, not by positions.

When `PRAGMA OrderedColumns;` is enabled, the order of columns is preserved in the query results and is derived from the order of columns in the input tables according to the following rules:

- `SELECT` with explicit column enumeration sets the corresponding order.
- `SELECT` with an asterisk (`SELECT * FROM ...`) inherits the order from its input.
- the order of columns after JOIN: first the columns from the left side, then from the right. If the order of either side present in the output `JOIN` is not defined, the order of the result columns is also not defined;
- the order of depends on the execution mode of `UNION ALL`.
- Column order for AS_TABLE is not defined.

Combination of queries

Results of multiple SELECT (or subqueries) can be combined using the keywords `UNION` and `UNION ALL`.

```
query1 UNION [ALL] query2 (UNION [ALL] query3 ...)
```

Union of more than two queries is interpreted as a left-associative operation, i.e.

```
query1 UNION query2 UNION ALL query3
```

is interpreted as

```
(query1 UNION query2) UNION ALL query3
```

If `ORDER BY/LIMIT/DISCARD/INTO RESULT` is present in the combined subqueries, the following rules apply:

- `ORDER BY/LIMIT/INTO RESULT` is allowed only after the last subquery
- `DISCARD` is allowed only before the first subquery.
- the specified operators act on the result `UNION [ALL]`, not on the subquery
- to apply an operator to a subquery, the subquery must be enclosed in parentheses.

Accessing multiple tables in a single query

In standard SQL, UNION ALL is used to query multiple tables, which combines the results of two or more `SELECT`. This is not very convenient for a use case where you need to run the same query across multiple tables (for example, containing data for different dates). In YQL, for convenience, in `SELECT` after `FROM` you can specify not only a single table or subquery, but also call built-in functions that allow combining data from multiple tables.

The following functions are defined for these purposes:

`CONCAT(`table1`, `table2`, `table3` VIEW view_name, ...)` — combines all tables listed in the arguments.

`EACH($list_of_strings)` or `EACH($list_of_strings VIEW view_name)` — combines all tables whose names are listed in the string list. Optionally, you can pass multiple lists in separate arguments, similar to `CONCAT`.

`RANGE(`prefix`, `min`, `max`, `suffix`, `view`)`: combines a range of tables. Arguments:

- prefix — directory for searching tables, specified without a trailing slash. The only required argument; if only it is specified, all tables in this directory are used.
- min, max — the next two arguments specify a range of names for including tables. The range is inclusive on both ends. If the range is not specified, all tables in the prefix directory are used. Names of tables or directories located in the directory specified in prefix are compared with the range `[min, max]` lexicographically, not concatenated, so it is important to specify the range without leading slashes.
- suffix — table name. Expected without a leading slash. If suffix is not specified, the arguments `[min, max]` specify a range of table names. If suffix is specified, the arguments `[min, max]` specify a range of folders in which a table with the name specified in the suffix argument exists.

`LIKE(`prefix`, `pattern`, `suffix`, `view`)` и `REGEXP(`prefix`, `pattern`, `suffix`, `view`)` — the pattern argument is specified in a format similar to the binary operators of the same name: LIKE and REGEXP.

`FILTER(`prefix`, `callable`, `suffix`, `view`)` — the callable argument must be a callable expression with signature `(String)->Bool`, which will be called for each table/subdirectory in the prefix directory. Only those tables for which the callable value returned `true` will participate in the query. As a callable value, it is most convenient to use lambda functions.

> **Warning**
>
> The order in which tables are merged by all the above functions is not guaranteed.
>
> The list of tables is computed **before** the query itself is executed. Therefore, tables created during the query will not be included in the function results.

By default, schemas of all participating tables are merged according to the rules of UNION ALL. If schema merging is not desired, you can use functions with the suffix `_STRICT`, for example `CONCAT_STRICT` or `RANGE_STRICT`, which work exactly like the original ones but treat any discrepancy in table schemas as an error.

To specify the cluster of the merged tables, you need to specify it before the function name.

All arguments of the functions described above can be declared separately using named expressions. In this case, simple expressions are also allowed in them by implicitly calling EvaluateExpr.

The name of the source table from which each row was originally obtained can be obtained using the TablePath() function.

Examples

```
SELECT * FROM CONCAT(
  `table1`,
  `table2`,
  `table3`);
```

```
$indices = ListFromRange(1, 4);
$tables = ListMap($indices, ($index) -> {
    RETURN "table" || CAST($index AS String);
});
SELECT * FROM EACH($tables); -- identical to the previous example
```

```
SELECT * FROM RANGE(`my_folder`);
```

```
SELECT * FROM some_cluster.RANGE( -- The cluster can be specified before the function name
  `my_folder`,
  `from_table`,
  `to_table`);
```

```
SELECT * FROM RANGE(
  `my_folder`,
  `from_folder`,
  `to_folder`,
  `my_table`);
```

```
SELECT * FROM RANGE(
  `my_folder`,
  `from_table`,
  `to_table`,
  ``,
  `my_view`);
```

```
SELECT * FROM LIKE(
  `my_folder`,
  "2017-03-%"
);
```

```
SELECT * FROM REGEXP(
  `my_folder`,
  "2017-03-1[2-4]?"
);
```

```
$callable = ($table_name) -> {
    return $table_name > "2017-03-13";
};

SELECT * FROM FILTER(
  `my_folder`,
  $callable
);
```

Supported constructs in SELECT

- FROM
- FROM AS_TABLE
- FROM SELECT
- DISTINCT
- UNIQUE DISTINCT
- UNION
- WITH
- WITHOUT
- WHERE
- ORDER BY
- ASSUME ORDER BY
- LIMIT OFFSET
- SAMPLE
- TABLESAMPLE
- JOIN
- GROUP BY
- FLATTEN
- WINDOW
- VIEW secondary_index
- VIEW vector_index
- VIEW fulltext_index
- VIEW json_index

FROM

Data source for `SELECT`. The argument can accept the table name, the result of another `SELECT`, or a named expression. Between `SELECT` and `FROM`, list the comma-separated column names from the source (or `*` to select all columns).

Examples

```
SELECT key FROM my_table;
```

```
SELECT * FROM
  (SELECT value FROM my_table);
```

```
$table_name = "my_table";
SELECT * FROM $table_name;
```

FROM AS_TABLE

Accessing named expressions as tables using the `AS_TABLE` function.

`AS_TABLE($variable)` lets you use the value of `$variable` as the data source for the query. In this case, the variable `$variable` must have the type `List<Struct<...>>`.

Example

```
$data = AsList(
    AsStruct(1u AS Key, "v1" AS Value),
    AsStruct(2u AS Key, "v2" AS Value),
    AsStruct(3u AS Key, "v3" AS Value));

SELECT Key, Value FROM AS_TABLE($data);
```

You should either explicitly specify the modifiable columns in both the source and the target when using expressions with modifying queries such as UPSERT INTO or INSERT INTO:

```
$data = AsList(
    AsStruct(1u AS Key, "v1" AS Value),
    AsStruct(2u AS Key, "v2" AS Value),
    AsStruct(3u AS Key, "v3" AS Value));

INSERT INTO `my_table` (Key, Value) SELECT Key, Value FROM AS_TABLE($data);
```

Or you should omit them completely:

```
$data = AsList(
    AsStruct(1u AS Key, "v1" AS Value),
    AsStruct(2u AS Key, "v2" AS Value),
    AsStruct(3u AS Key, "v3" AS Value));

INSERT INTO `my_table` SELECT * FROM AS_TABLE($data);
```

FROM ... SELECT ...

An inverted format, first specifying the data source and then the operation.

Examples

```
FROM my_table SELECT key, value;
```

```
FROM a_table AS a
JOIN b_table AS b
USING (key)
SELECT *;
```

FLATTEN

FLATTEN BY

Converts rows in the source table using vertical unpacking of containers of variable length (lists or dictionaries).

For example:

- Source table:

[a, b, c]	1
[d]	2
[]	3

- The table resulting from `FLATTEN BY` on the left column:

a	1
b	1
c	1
d	2

YDB tables don't support container types, so the FLATTEN BY function can only be applied to table-type variables created within a YQL query.

Example

```
$sample = AsList(
    AsStruct(AsList('a','b','c') AS value, CAST(1 AS Uint32) AS id),
    AsStruct(AsList('d') AS value, CAST(2 AS Uint32) AS id),
    AsStruct(AsList() AS value, CAST(3 AS Uint32) AS id)
);

SELECT value, id FROM as_table($sample) FLATTEN BY (value);
```

This conversion can be convenient in the following cases:

- When it is necessary to output statistics by cells from a container column (for example, via `GROUP BY`).
- When the cells in a container column store IDs from another table that you want to join with `JOIN`.

Syntax

- `FLATTEN BY` is specified after `FROM`, but before `GROUP BY`, if `GROUP BY` is present in the query.
- The type of the result column depends on the type of the source column:

Container type	Result type	Comments
`List<X>`	`X`	List cell type
`Dict<X,Y>`	`Tuple<X,Y>`	Tuple of two elements containing key-value pairs
`Optional<X>`	`X`	The result is almost equivalent to the clause `WHERE foo IS NOT NULL`, but the `foo` column type is changed to `X`

- By default, the result column replaces the source column. Use `FLATTEN BY foo AS bar` to keep the source container. As a result, the source container is still available as `foo` and the output container is available as `bar`.
- To build a Cartesian product of multiple container columns, use the clause `FLATTEN BY (a, b, c)`. Parentheses are mandatory to avoid grammar conflicts.
- Inside `FLATTEN BY`, you can only use column names from the input table. To apply `FLATTEN BY` to the calculation result, use a subquery.
- In `FLATTEN BY` you can use both columns and arbitrary named expressions (unlike columns, `AS` is required in this case). To avoid grammatical ambiguities of the expression after `FLATTEN BY`, make sure to use parentheses with the following: `... FLATTEN BY (ListSkip(col, 1) AS col) ...`
- If the source column had nested containers, for example, `List<DictX,Y>`, `FLATTEN BY` unpacks only the outer level. To completely unpack the nested containers, use a subquery.

> **Note**
>
> `FLATTEN BY` interprets optional data types as lists of length 0 or 1. The table rows with `NULL` are skipped, and the column type changes to a similar non-optional type.
>
> `FLATTEN BY` makes only one conversion at a time, so use `FLATTEN LIST BY` or `FLATTEN OPTIONAL BY` on optional containers, for example, `Optional<List<String>>`.

Specifying the container type

To specify the type of container to convert to, you can use:

- `FLATTEN LIST BY`

  For `Optional<List<T>>`, `FLATTEN LIST BY` will unpack the list, treating `NULL` as an empty list.
- `FLATTEN DICT BY`

For `Optional<Dict<T>>`, `FLATTEN DICT BY` will unpack the dictionary, interpreting `NULL` as an empty dictionary.

- `FLATTEN OPTIONAL BY`

  To filter the `NULL` values without serialization, specify the operation by using `FLATTEN OPTIONAL BY`.

Examples

```
SELECT
  t.item.0 AS key,
  t.item.1 AS value,
  t.dict_column AS original_dict,
  t.other_column AS other
FROM my_table AS t
FLATTEN DICT BY dict_column AS item;
```

```
SELECT * FROM (
    SELECT
        AsList(1, 2, 3) AS a,
        AsList("x", "y", "z") AS b
) FLATTEN LIST BY (a, b);
```

```
SELECT * FROM (
    SELECT
        "1;2;3" AS a,
        AsList("x", "y", "z") AS b
) FLATTEN LIST BY (String::SplitToList(a, ";") as a, b);
```

Analogues of FLATTEN BY in other DBMS

- PostgreSQL: `unnest`
- Hive: `LATERAL VIEW`
- MongoDB: `unwind`
- Google BigQuery: `FLATTEN`
- ClickHouse: `ARRAY JOIN / arrayJoin`

FLATTEN COLUMNS

Converts a table where all columns must be structures to a table with columns corresponding to each element of each structure from the source columns.

The names of the source column structures are not used and not returned in the result. Be sure that the structure element names aren't repeated in the source columns.

Example

```
SELECT x, y, z
FROM (
  SELECT
    AsStruct(
        1 AS x,
        "foo" AS y),
    AsStruct(
        false AS z)
) FLATTEN COLUMNS;
```

GROUP BY

Groups the results of `SELECT` by the values of specified columns or expressions. Together with `GROUP BY`, aggregate functions (`COUNT`, `MAX`, `MIN`, `SUM`, `AVG`) are often used to perform calculations in each group.

If `GROUP BY` is present in the query, then when selecting columns (between `SELECT ... FROM`), the following constructs are allowed:

1. Columns by which grouping is performed (present in the `GROUP BY` argument).
2. Aggregate functions (see the next section). Columns that are **not** grouped by can only be included as arguments of an aggregate function.
3. Functions that return the start and end time of the current window (`HOP_START` and `HOP_END`) (for `GROUP BY HOP`).
4. Arbitrary computations combining items 1-3.

It is possible to group by the result of evaluating an arbitrary expression on the source columns. In this case, to access the result of this expression, it is recommended to assign it a name using `AS`, see the second example.

Syntax

```
SELECT                            -- In SELECT you can use:
    column1,                      -- key columns specified in GROUP BY
    key_n,                        -- named expressions specified in GROUP BY
    column1 + key_n,              -- arbitrary non-aggregate functions on them
    Aggr_Func1( column2 ),        -- aggregate functions containing any columns in their arguments,
    Aggr_Func2( key_n + column2 ), --   including named expressions specified in GROUP BY
    ...
FROM table
GROUP BY
    column1, column2, ...,
    <expr> AS key_n           -- When grouping by an expression, it can be given a name via AS,
                              -- which can be used in SELECT
```

A query of the form `SELECT * FROM table GROUP BY k1, k2, ...` returns all columns listed in GROUP BY, that is, it is equivalent to the query `SELECT DISTINCT k1, k2, ... FROM table`.

The asterisk can also be used as an argument of the aggregate function `COUNT`. `COUNT(*)` means "number of rows in the group".

> **Note**
>
> Aggregate functions do not consider `NULL` in their arguments, except for the function `COUNT`.

Also in YQL, there is a mechanism of aggregate function factories, implemented using the functions `AGGREGATION_FACTORY` and `AGGREGATE_BY`.

Examples

```
SELECT key, COUNT(*) FROM my_table
GROUP BY key;
```

```
SELECT double_key, COUNT(*) FROM my_table
GROUP BY key + key AS double_key;
```

```
SELECT
   double_key,                          -- OK: key column
   COUNT(*) AS group_size,              -- OK: COUNT(*)
   SUM(key + subkey) AS sum1,           -- OK: aggregate function
   CAST(SUM(1 + 2) AS String) AS sum2,  -- OK: aggregate function with constant argument
   SUM(SUM(1) + key) AS sum3,           -- ERROR: nested aggregations are not allowed
   key AS k1,                           -- ERROR: using non-key column key without aggregation
   key * 2 AS dk1,                      -- ERROR in YQL: using non-key column key without aggregation
FROM my_table
GROUP BY
  key * 2 AS double_key,
  subkey as sk,
```

> **Warning**
>
> The ability to specify a name for a column or expression in `GROUP BY .. AS foo` is a YQL extension. Such a name becomes visible in `WHERE` despite the fact that filtering by `WHERE` is performed earlier than grouping. In particular, if the `T` table has two columns `foo` and `bar`, then in the `SELECT foo FROM T WHERE foo > 0 GROUP BY bar AS foo` query filtering will actually occur on the `bar` column from the original table.

GROUP BY ... SessionWindow()

YQL supports grouping by sessions. You can add a special function `SessionWindow` to the regular expressions in `GROUP BY`:

```
SELECT
  user,
  session_start,
  SessionStart() AS same_session_start, -- same as session_start
  COUNT(*) AS session_size,
```

```
  SUM(value) AS sum_over_session,
FROM my_table
GROUP BY user, SessionWindow(<time_expr>, <timeout_expr>) AS session_start
```

The following happens:

1. The input table is partitioned by the grouping keys specified in `GROUP BY`, ignoring SessionWindow (in this case, by `user`). If there is nothing else in `GROUP BY` besides SessionWindow, the input table falls into a single partition
2. Each partition is divided into non-overlapping subsets of rows (sessions). To do this, the partition is sorted in ascending order of the value of expression `time_expr`. Session boundaries are drawn between adjacent elements of the partition whose difference in `time_expr` values exceeds `timeout_expr`
3. The sessions obtained in this way are the final partitions on which aggregate functions are computed.

The key column of SessionWindow() (in the example, `session_start`) has the value "minimum `time_expr` in the session". Additionally, when SessionWindow() is present in `GROUP BY`, the special aggregate function SessionStart can be used.

An extended version of SessionWindow with four arguments is also supported:

`SessionWindow(<order_expr>, <init_lambda>, <update_lambda>, <calculate_lambda>)`

Here:

- `<order_expr>` – expression by which the original partition is sorted
- `<init_lambda>` – lambda function for initializing the session calculation state. It has the signature `(TableRow())->State`. It is called once on the first element (in sort order) of the original partition
- `<update_lambda>` – lambda function for updating the session calculation state and determining session boundaries. It has the signature `(TableRow(), State)->Tuple<Bool, State>`. It is called on each element of the original partition except the first. The new state value is computed based on the current table row and the previous state. If the first element of the returned tuple has the value `True`, then a new session starts from the *current* row. The key of the new session is obtained by applying `<calculate_lambda>` to the second element of the tuple.
- `<calculate_lambda>` – a lambda function for computing the session key (the "value" of SessionWindow(), which is also available via SessionStart()). The function has the signature `(TableRow(), State)->SessionKey`. It is called on the first element of a partition (after `<init_lambda>`) and on those elements for which `<update_lambda>` returned `True` as the first element of the tuple. It is worth noting that to start a new session, `<calculate_lambda>` must return a value that differs from the previous session key. At the same time, sessions with the same keys are not merged. For example, if `<calculate_lambda>` sequentially returns `0, 1, 0, 1`, then these will be four different sessions.

Using the extended version of SessionWindow, you can solve, for example, the following problem: split a partition into sessions as in the two-argument version of SessionWindow, but with a limit on the maximum session length set to some constant:

Example

```
$max_len = 1000; -- maximum session length
$timeout = 100; -- timeout (timeout_expr in the simplified version of SessionWindow)

$init = ($row) -> (AsTuple($row.ts, $row.ts)); -- session state - a tuple of 1) the value of the time column
ts on the first row of the session and 2) on the current row
$update = ($row, $state) -> {
  $is_end_session = $row.ts - $state.0 > $max_len OR $row.ts - $state.1 > $timeout;
  $new_state = AsTuple(IF($is_end_session, $row.ts, $state.0), $row.ts);
  return AsTuple($is_end_session, $new_state);
};
$calculate = ($row, $state) -> ($row.ts);
SELECT
  user,
  session_start,
  SessionStart() AS same_session_start, -- same as session_start
  COUNT(*) AS session_size,
  SUM(value) AS sum_over_session,
FROM my_table
GROUP BY user, SessionWindow(ts, $init, $update, $calculate) AS session_start
```

`SessionWindow` can be used in `GROUP BY` only once.

ROLLUP, CUBE, and GROUPING SETS

Results of aggregate function computation as subtotals for groups and grand totals for individual columns or the entire table.

Syntax

```
SELECT
    c1, c2,                          -- columns by which grouping is performed

AGGREGATE_FUNCTION(c3) AS outcome_c  -- aggregate function (SUM, AVG, MIN, MAX, COUNT)

FROM table_name

GROUP BY
    GROUP_BY_EXTENSION(c1, c2)       -- GROUP BY extension: ROLLUP, CUBE, or GROUPING SETS
```

- `ROLLUP`: groups column values in the order they are listed in the arguments (strictly from left to right), forming subtotals for each group and a grand total.
- `CUBE` — groups values for all possible combinations of columns, forming subtotals for each group and a grand total.
- `GROUPING SETS` — specifies groups for subtotals.

`ROLLUP`, `CUBE`, and `GROUPING SETS` can be combined with a comma.

GROUPING

In the subtotal, the values of columns not involved in calculations are replaced with `NULL`. In the grand total, the values of all columns are replaced with `NULL`. `GROUPING` is a function that allows you to distinguish the original `NULL` values from the `NULL` values that were added when generating the grand and intermediate totals.

`GROUPING` returns a bitmask:

- `0` — `NULL` for the original empty value.
- `1` — `NULL`, added for intermediate or grand total.

Example

```
SELECT
    column1,
    column2,
    column3,

    CASE GROUPING(
        column1,
        column2,
        column3,
    )
        WHEN 1  THEN "Subtotal: column1 and column2"
        WHEN 3  THEN "Subtotal: column1"
        WHEN 4  THEN "Subtotal: column2 and column3"
        WHEN 6  THEN "Subtotal: column3"
        WHEN 7  THEN "Grand total"
        ELSE         "Individual group"
    END AS subtotal,

    COUNT(*) AS rows_count

FROM my_table

GROUP BY
    ROLLUP(
        column1,
        column2,
        column3
    ),
    GROUPING SETS(
        (column2, column3),
        (column3)
        -- if you add (column2) here as well, then in total
        -- these ROLLUP and GROUPING SETS would give a result,
        -- similar to CUBE
    )
;
```

DISTINCT

Applying aggregate functions only to unique column values.

> **Note**
>
> Applying `DISTINCT` to computed values is not currently implemented. For this purpose, you can use a subquery or the `GROUP BY ... AS ...` expression.

Example

```
SELECT
  key,
  COUNT(DISTINCT value) AS count -- top 3 keys by number of unique values
FROM my_table
GROUP BY key
ORDER BY count DESC
LIMIT 3;
```

Also, the `DISTINCT` keyword can be used to select unique rows through `SELECT DISTINCT`.

COMPACT

Having the SQL hint `COMPACT` immediately after the keyword `GROUP` allows more efficient aggregation in cases where the query author knows in advance that none of the aggregation keys will have a large amount of data (on the order of a gigabyte or a million rows). If this assumption is not met, the operation may fail with an Out of Memory error or start working significantly slower compared to a regular GROUP BY.

Unlike a regular GROUP BY, the Map-side combiner stage and additional Reduce for each field with DISTINCT aggregation are disabled.

Example

```
SELECT
  key,
  COUNT(DISTINCT value) AS count -- top 3 keys by number of unique values
FROM my_table
```

```
GROUP /*+ COMPACT() */ BY key
ORDER BY count DESC
LIMIT 3;
```

GROUP BY ... HOP

`HOP` groups data by overlapping time windows (hopping windows). It is supported both in analytical queries to tables and in streaming queries to topics.

```
HOP(time_extractor, hop, interval, delay)
```

Where:

- `time_extractor` — SQL expression of type `Timestamp` that defines the event time. A timestamp is calculated from each input row, which determines the window assignment.
- `hop` : step between the starts of adjacent windows in ISO 8601 format, for example `"PT10S"` (10 seconds).
- `interval` — the size (duration) of each window in ISO 8601 format, for example `"PT30S"` (30 seconds).
- — `delay` for closing a window after its completion in ISO 8601 format. Used only in streaming queries (ignored when working with tables). For streaming queries, it is recommended to use HoppingWindow with watermarks instead of `delay` .

Also available are aggregate functions `HOP_START()` and `HOP_END()` , which return the start and end timestamps of the current window of type `Timestamp` respectively.

Description

Let's break down the algorithm using an example.

```
GROUP BY HOP(CAST(ts AS Timestamp), "PT10S", "PT30S", "PT20S")
```

In this example, `CAST(ts AS Timestamp)` extracts the event time from the `ts` column. The `hop` parameter is 10 seconds, `interval` is 30 seconds, `delay` is 20 seconds.

Windows are constructed according to the following rule:

- Window starts are aligned to moments that are multiples of `hop` (10 seconds), starting from 0: 0, 10, 20, and so on.
- The duration of each window is `interval` (30 seconds). The resulting windows are: `[0; 30)` , `[10; 40)` , `[20; 50)` , and so on.
- An event falls into all windows whose time range includes its time. For example, an event with a time of 25 seconds falls into windows `[0; 30)` , `[10; 40)` , and `[20; 50)` .
- A window is considered completed when an event with a timestamp not less than the end of this window + `delay` (20 seconds) is received. For example, window `[10; 40)` closes when an event with a timestamp of 60 or more is received.

Analytical `HOP` over a table

When working with tables, data is grouped by `GROUP BY` keys (ignoring `HOP`), forming groups of rows (hereinafter groups). Inside each group:

1. Rows are sorted in ascending order of `time_extractor` .
2. Each row is assigned to one or more overlapping windows.
3. On each window, the specified aggregate functions are computed.

The `delay` parameter is **not used** in analytical table processing: the data is already fully available, the traversal order is set by sorting by `time_extractor` , and the window completion is determined by the full group scan algorithm (see the closing rule in the description above).

Streaming `HOP` over a topic

When working with topics, data is grouped by `GROUP BY` keys (ignoring `HOP`), forming groups. Inside each group:

1. Events are processed in an order close to ascending `time_extractor` . Small deviations from strict order are allowed.
2. Each event is assigned to one or more overlapping windows.
3. On each window, the specified aggregate functions are computed.

In streaming queries, events may arrive not in strict chronological order. The `delay` parameter sets the waiting time after the formal window completion: the window does not close immediately, but after `delay` seconds, to give delayed events time to arrive. Events that arrive after the window is closed are ignored.

Limitations

`time_extractor` is an SQL expression that depends only on the input column values and must have type `Timestamp` .

To specify `hop` , `interval` , and `delay` , a string expression conforming to the ISO 8601 standard is used, for example, `PT10S` — 10 seconds, `PT1M` — 1 minute. This is the format used to construct the built-in type `Interval` from a string.

The values of parameters `interval` and `delay` must be divisible by the value of parameter `hop` . This requirement ensures alignment of window boundaries: each window starts at a time that is a multiple of `hop` and ends exactly after `interval` , which guarantees uniform coverage of the time axis without gaps. Parameters `hop` and `interval` must be positive.

Examples

```
SELECT
    sensor_id,
    HOP_END() AS window_end,
    AVG(temperature) AS avg_temp,
    COUNT(*) AS event_count
FROM sensor_data
GROUP BY
```

```
    sensor_id,
    HOP(CAST(event_time AS Timestamp), "PT10S", "PT1M", "PT30S");
```

GROUP BY ... HoppingWindow

groups events by overlapping time windows (hopping windows), similar to GROUP BY `HoppingWindow`. It is supported both in analytical queries to tables and in stream queries to topics. The main difference from `HOP` is that in stream queries, `HoppingWindow` uses the watermarks mechanism to determine when to close a window instead of a fixed parameter `delay`.

```
HoppingWindow(time_extractor, hop, interval)
```

Where:

- `time_extractor`: an SQL expression of type `Timestamp` that defines the event time. It must depend only on input columns.
- `hop`: the step (shift period) between the starts of adjacent windows in ISO 8601 format, for example `"PT10S"` (10 seconds).
- `interval`: the size (duration) of each window in ISO 8601 format, for example `"PT1M"` (1 minute). The value `interval` must be divisible by `hop`, because windows are aligned to multiples of the step interval.

The functions `HOP_START()` and `HOP_END()` are also available, returning timestamps of the start and end of the current window.

The window construction algorithm is the same as GROUP BY HOP: windows start at moments that are multiples of `hop` and have a duration of `interval`. An event falls into all windows whose time range includes its time.

Analytical `HoppingWindow` over a table

When working with tables, `HoppingWindow` performs grouping by time windows similarly to HOP, but without the `delay` parameter, which was always ignored in analytical use (the data in the table is already sorted).

1. The input table is partitioned by the grouping keys specified in `GROUP BY`, ignoring `HoppingWindow`.
2. Each partition is sorted by ascending `time_extractor`.
3. Each partition is divided into overlapping subsets of events (windows).
4. The specified aggregate functions are computed on each subset.

Streaming `HoppingWindow` over a topic

When working with topics, `HoppingWindow` uses watermarks to determine when to close a window. A window is closed when the watermark value is not less than the end of the window. This provides more accurate aggregation results compared to `HOP`, where the window is closed at a fixed `delay`.

1. The input topic is partitioned by the grouping keys specified in `GROUP BY`, ignoring `HoppingWindow`.
2. In each partition, the window advances independently of others.
3. Events are processed in an order close to ascending `time_extractor`. Minor reorderings of the input stream order are allowed.
4. Each partition is divided into overlapping subsets of events (windows).
5. A window is closed when a watermark is received whose value is not less than the end of the window. After closing, the aggregation result is output.
6. Events that arrive after the window is closed are not included in the results.

For `HoppingWindow` to work correctly in streaming mode, you need to configure the watermark in the WITH section of the source. For more information: Configuration.

Example

Below is streaming reading from a topic: in `SELECT` it is convenient to output the end of the window via `HOP_END()`. For **tables** in an analytical query, `HOP_START()` or `HOP_END()` are more often used, depending on which window boundary needs to be shown in the result; the meaning of the windows is the same, only the selected label differs.

```
SELECT
    key,
    HOP_END() AS window_end,
    COUNT(*) AS event_count
FROM
    my_topic
WITH (
    FORMAT = json_each_row,
    SCHEMA = (
        key String,
        event_time String
    ),
    WATERMARK = __ydb_write_time - Interval("PT5S")
)
GROUP BY
    key,
    HoppingWindow(__ydb_write_time, "PT10S", "PT1M");
```

HAVING

Filtering the `SELECT` result set based on the results of computing aggregate functions. The syntax is similar to the `WHERE` construct.

Example

```
SELECT
    key
FROM my_table
GROUP BY key
HAVING COUNT(value) > 100;
```

JOIN

It lets you combine multiple data sources (subqueries or tables) by equality of values in the specified columns or expressions (the `JOIN` keys).

Syntax

```
SELECT ...    FROM table_1
-- first JOIN step:
  <Join_Type> JOIN table_2 <Join_Condition>
  -- left subquery -- entries in table_1
  -- right subquery -- entries in table_2
-- next JOIN step:
  <Join_Type> JOIN table_n <Join_Condition>
  -- left subquery -- JOIN result in the previous step
  -- right subquery -- entries in table_n
-- JOIN can include the following steps
...
WHERE  ...
```

At each JOIN step, rules are used to establish correspondences between rows in the left and right data subqueries, creating a new subquery that includes every combination of rows that meet the JOIN conditions.

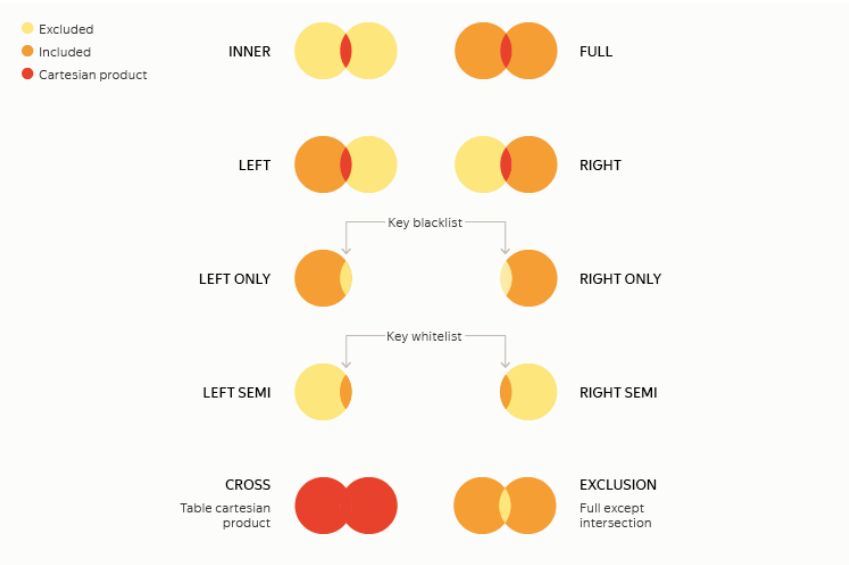
Attention!

Since columns in YQL are identified by their names, and you can't have two columns with the same name in the subquery, `SELECT * FROM ... JOIN ...` can't be executed if there are columns with identical names in the joined tables.

Types of join

- `INNER` (default): Rows from joined subqueries that don't match any rows on the other side won't be included in the result.
- `LEFT`: If there's no value in the right subquery, it adds a row to the result with column values from the left subquery, using `NULL` in columns from the right subquery
- `RIGHT`: If there's no value in the left subquery, it adds the row to the result, including column values from the right subquery, but using `NULL` in columns from the left subquery
- `FULL` = `LEFT` + `RIGHT`
- `LEFT/RIGHT SEMI`: One side of the subquery is a whitelist of keys, its values are not available. The result includes columns from one table only, no Cartesian product is created.
- `LEFT/RIGHT ONLY`: Subtracting the sets by keys (blacklist). It's almost the same as adding `IS NULL` to the key on the opposite side in the regular `LEFT/RIGHT` JOIN, but with no access to values: the same as `SEMI` JOIN.
- `CROSS`: A full Cartesian product of two tables without specifying key columns and no explicit `ON/USING`.
- `EXCLUSION`: Both sides minus the intersection.

Use `LEFT SEMI JOIN` instead of a correlated `EXISTS` subquery and `LEFT ONLY JOIN` instead of a correlated `NOT EXISTS` subquery. See Correlated subqueries, EXISTS, and NOT EXISTS.



Note

`NULL` is a special value to denote nothing. Hence, `NULL` values on both sides are not treated as equal to each other. This eliminates ambiguity in some types of `JOIN` and avoids a giant Cartesian product otherwise created.

Conditions for joining

For `CROSS JOIN`, no join condition is specified. The result includes the Cartesian product of the left and right subquery, meaning it combines everything with everything. The number of rows in the resulting subquery is the product of the number of rows in the left and right subqueries.

For any other JOIN types, specify the condition using one of the two methods:

1. `USING (column_name)`. Used if both the left and right subqueries share a column whose equality of values is a join condition.
2. `ON (equality_conditions)`. Lets you set a condition of equality for column values or expressions over columns in the left and right subqueries or use several such conditions combined by `and`.

Examples

```
SELECT    a.value as a_value, b.value as b_value
FROM      a_table AS a
FULL JOIN b_table AS b USING (key);
```

```
SELECT    a.value as a_value, b.value as b_value
FROM      a_table AS a
FULL JOIN b_table AS b ON a.key = b.key;
```

```
SELECT     a.value as a_value, b.value as b_value, c.column2
FROM       a_table AS a
CROSS JOIN b_table AS b
LEFT  JOIN c_table AS c ON c.ref = a.key and c.column1 = b.value;
```

To make sure no full scan of the right joined table is required, a secondary index can be applied to the columns included in the Join condition. Accessing a secondary index should be specified explicitly in `JOIN table_name VIEW index_name AS table_alias` format.

For example, creating an index to use in the Join condition:

```
ALTER TABLE b_table ADD INDEX b_index_ref GLOBAL ON(ref);
```

Using the created index:

```
SELECT    a.value as a_value, b.value as b_value
FROM      a_table AS a
INNER JOIN b_table VIEW b_index_ref AS b ON a.ref = b.ref;
```

OVER, PARTITION BY, and WINDOW

Window functions were introduced in the SQL:2003 standard and expanded in the SQL:2011 standard. They let you run calculations on a set of table rows that are related to the current row in some way.

Unlike aggregate functions, window functions don't group rows into one output row: the number of rows in the resulting table is always the same as in the source table.

If a query contains both aggregate and window functions, grouping is performed and aggregate function values are calculated first. The calculated values of aggregate functions can be used as window function arguments (but not the other way around).

Syntax

General syntax for calling a window function is as follows

```
function_name([expression [, expression ...]]) OVER (window_definition)
```

or

```
function_name([expression [, expression ...]]) OVER window_name
```

Here, window name (`window_name`) is an arbitrary ID that is unique within the query and `expression` is an arbitrary expression that contains no window function calls.

In the query, each window name must be mapped to the window definition (`window_definition`):

```
SELECT
    F0(...) OVER (window_definition_0),
    F1(...) OVER w1,
    F2(...) OVER w2,
    ...
FROM my_table
WINDOW
    w1 AS (window_definition_1),
    ...
    w2 AS (window_definition_2)
;
```

Here, the `window_definition` is written as

```
[ PARTITION BY (expression AS column_identifier | column_identifier) [, ...] ]
[ ORDER BY expression [ASC | DESC] ]
[ frame_definition ]
```

You can set an optional *frame definition* (`frame_definition`) one of two ways:

- `ROWS frame_begin`
- `ROWS BETWEEN frame_begin AND frame_end`

The frame start (`frame_begin`) and *frame end* (`frame_end`) are set one of the following ways:

- `UNBOUNDED PRECEDING`
- `offset PRECEDING`
- `CURRENT ROW`
- `offset FOLLOWING`
- `UNBOUNDED FOLLOWING`

Here, the *frame* `offset` is a non-negative numeric literal. If the frame end isn't set, the `CURRENT ROW` is assumed.

There should be no window function calls in any of the expressions inside the window definition.

Calculation algorithm

Partitioning

If `PARTITION BY` is set, the source table rows are grouped into *partitions*, which are then handled independently of each other.
If `PARTITION BY` isn't set, all rows in the source table are put in the same partition. If `ORDER BY` is set, it determines the order of rows in a partition.
Both in `PARTITION BY` and GROUP BY you can use aliases and SessionWindow.

If `ORDER BY` is omitted, the order of rows in the partition is undefined.

Frame

The `frame_definition` specifies a set of partition rows that fall into the *window frame* associated with the current row.

In `ROWS` mode (the only one that YQL currently supports), the window frame contains rows with the specified offsets relative to the current row in the partition. For example, if `ROWS BETWEEN 3 PRECEDING AND 5 FOLLOWING` is used, the window frame contains 3 rows preceding the current one, the current row, and 5 rows following it.

The set of rows in the window frame may change depending on which row is the current one. For example, for the first row in the partition, the `ROWS BETWEEN 3 PRECEDING AND 1 PRECEDING` window frame will have no rows.

Setting `UNBOUNDED PRECEDING` as the frame start means "from the first partition row" and `UNBOUNDED FOLLOWING` as the frame end — "up to the last partition row". Setting `CURRENT ROW` means "from/to the current row".

If no `frame_definition` is specified, a set of rows to be included in the window frame depends on whether there is `ORDER BY` in the `window_definition`.
Namely, if there is `ORDER BY`, then `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` is implicitly assumed. If none, then `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`.

Further, depending on the specific window function, it's calculated either based on the set of rows in the partition or the set of rows in the window frame.

List of available window functions

Examples

```
SELECT
    COUNT(*) OVER w AS rows_count_in_window,
    some_other_value -- access the current row
FROM `my_table`
WINDOW w AS (
    PARTITION BY partition_key_column
    ORDER BY int_column
);
```

```
SELECT
    LAG(my_column, 2) OVER w AS row_before_previous_one
FROM `my_table`
WINDOW w AS (
    PARTITION BY partition_key_column
);
```

```
SELECT
    -- AVG (like all aggregate functions used as window functions)
    -- is calculated on the window frame
    AVG(some_value) OVER w AS avg_of_prev_current_next,
    some_other_value -- access the current row
FROM my_table
WINDOW w AS (
    PARTITION BY partition_key_column
    ORDER BY int_column
    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
);
```

```
SELECT
    -- LAG doesn't depend on the window frame position
    LAG(my_column, 2) OVER w AS row_before_previous_one
FROM my_table
WINDOW w AS (
    PARTITION BY partition_key_column
    ORDER BY my_column
);
```

Implementation specifics

- Functions calculated on the `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` or `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` window frame are implemented efficiently (do not require additional memory and their computation runs on a partition in O(partition size) time).
- For the `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` window frame, you can choose the execution strategy in RAM by specifying the `COMPACT` hint after the `PARTITION` keyword.

  For example, `PARTITION COMPACT BY key` or `PARTITION COMPACT BY ()` (if `PARTITION BY` was missing initially).

  If the `COMPACT` hint is specified, this requires additional memory equal to O(partition size), but then no extra `JOIN` operation is made.
- If the window frame doesn't start with `UNBOUNDED PRECEDING`, calculating window functions on this window requires additional memory equal to O(the maximum number of rows from the window boundaries to the current row), while the computation time is equal to O(number_of_partition_rows * window_size).
- For the window frame starting with `UNBOUNDED PRECEDING` and ending with `N`, where `N` is neither `CURRENT ROW` nor `UNBOUNDED FOLLOWING`, additional memory equal to O(N) is required and the computation time is equal to O(N * number_of_partition_rows).
- The `LEAD(expr, N)` and `LAG(expr, N)` functions always require O(N) of RAM.

Given the above, a query with `ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING` should, if possible, be changed to `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` by reversing the `ORDER BY` sorting order.

DISTINCT

Selecting unique rows.

> **Note**
>
> Applying `DISTINCT` to calculated values is not currently implemented. For this purpose, use a subquery or the clause `GROUP BY ... AS ...`.

Example

```
SELECT DISTINCT value -- only unique values from the table
FROM my_table;
```

The `DISTINCT` keyword can also be used to apply aggregate functions only to distinct values. For more information, see the documentation for GROUP BY.

Removes duplicate rows from the result. Applies after the clause `GROUP BY ... AS ...`.

UNIQUE DISTINCT hints

Directly after `SELECT`, it is possible to add SQL hints `unique` or `distinct`, which declare that this projection generates data containing unique values in the specified set of columns of a row-based or columnar table. This can be used to optimize subsequent subqueries executed on this projection, or for writing to table meta-attributes during `INSERT` (currently not supported for columnar tables).

- Columns are specified in the hint values, separated by spaces.
- If the set of columns is not specified, uniqueness applies to the entire set of columns in this projection.
- `unique` - indicates unique or `null` values. According to the SQL standard, each null is unique: `NULL = NULL` -> `NULL`.
- `distinct` - indicates completely unique values including null: `NULL IS DISTINCT FROM NULL` -> `FALSE`.
- Multiple sets of columns can be specified in several hints for a single projection.
- If the hint contains a column that is not in the projection, it will be ignored.

Combining subquery results (UNION)

UNION

Union of the results of the underlying queries, with duplicates removed.
Behavior is identical to using `UNION ALL` followed by `SELECT DISTINCT *`.
Refer to UNION ALL for more details.

Examples

```
SELECT key FROM T1
UNION
SELECT key FROM T2 -- returns the set of distinct keys in the tables
```

UNION ALL

Concatenating results of multiple `SELECT` statements (or subqueries).

Two `UNION ALL` modes are supported: by column names (the default mode) and by column positions (corresponds to the ANSI SQL standard and is enabled by the PRAGMA).

In the "by name" mode, the output of the resulting data schema uses the following rules:

- The resulting table includes all columns that were found in at least one of the input tables.
- If a column wasn't present in all the input tables, then it's automatically assigned the optional data type (that can accept `NULL`).
- If a column in different input tables had different types, then the shared type (the broadest one) is output.
- If a column in different input tables had a heterogeneous type, for example, string and numeric, an error is raised.

The order of output columns in this mode is equal to the largest common prefix of the order of inputs, followed by all other columns in the alphabetic order.
If the largest common prefix is empty (for example, if the order isn't specified for one of the inputs), then the output order is undefined.

In the "by position" mode, the output of the resulting data schema uses the following rules:

- All inputs must have equal number of columns
- The order of columns must be defined for all inputs
- The names of the resulting columns must match the names of columns in the first table
- The type of the resulting columns is output as a common (widest) type of input column types having the same positions

The order of the output columns in this mode is the same as the order of columns in the first input.

Examples

```
SELECT 1 AS x
UNION ALL
SELECT 2 AS y
UNION ALL
SELECT 3 AS z;
```

In the default mode, this query returns a selection with three columns x, y, and z. When `PRAGMA PositionalUnionAll;` is enabled, the selection only includes the x column.

```
PRAGMA PositionalUnionAll;

SELECT 1 AS x, 2 as y
UNION ALL
SELECT * FROM AS_TABLE([<|x:3, y:4|>]); -- error: the order of columns in AS_TABLE is undefined
```

VIEW (INDEX)

To make a `SELECT` by secondary index of row-oriented table statement, use the following:

```
SELECT *
    FROM TableName VIEW IndexName
    WHERE …
```

> **Warning**
>
> Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Examples

- Select all the fields from the `series` row-oriented table using the `views_index` index with the `views >=someValue` criteria:

  ```
  SELECT series_id, title, info, release_date, views, uploaded_user_id
      FROM series VIEW views_index
      WHERE views >= someValue
  ```

- `JOIN` the `series` and `users` row-oriented tables on the `userName` field using the `users_index` and `name_index` indexes, respectively:

  ```
  SELECT t1.series_id, t1.title
      FROM series VIEW users_index AS t1
      INNER JOIN users VIEW name_index AS t2
      ON t1.uploaded_user_id == t2.user_id
      WHERE t2.name == userName;
  ```

VIEW (Vector index)

To select data from a row-oriented table using a vector index, use the following statements:

```
PRAGMA ydb.KMeansTreeSearchTopSize = "10";

SELECT ...
    FROM TableName VIEW IndexName
    WHERE ...
    ORDER BY Knn::SomeDistance(...)
    LIMIT ...
```

```
PRAGMA ydb.KMeansTreeSearchTopSize = "10";

SELECT ...
    FROM TableName VIEW IndexName
    WHERE ...
    ORDER BY Knn::SomeSimilarity(...) DESC
    LIMIT ...
```

Principles of operation and settings of the vector index are described in detail in Vector Indexes.

Note

A vector index supports a distance or similarity function from the Knn extension specified during its construction.

A vector index isn't automatically selected by the optimizer and must be specified explicitly using the `VIEW IndexName` expression.

If the `VIEW` expression is not used, the query will perform a full table scan with pairwise comparison of vectors. It is recommended to check the optimality of the written query using query plan analysis. In particular, ensure there is no full scan of the main table.

Warning

Indexed vector search completeness or performance may decrease after updating a large amount of data in a table with a vector index. For more details, see Updating Vector Indexes.

KMeansTreeSearchTopSize

Indexed vector search is based on an approximate algorithm (ANN, Approximate Nearest Neighbors). That means that indexed search may produce a result that differs from a similar full-scan nearest neighbor search.

Completeness of the indexed vector search is controlled by the following parameter: `PRAGMA ydb.KMeansTreeSearchTopSize`.

This parameter controls the maximum number of scanned clusters nearest to the requested search vector at every level of the search tree.
The parameter should be set explicitly for every search query.

The default value is 1. This means that only one nearest cluster is scanned at every level of the search tree by default. This parameter value maximizes search performance but results in minimum possible recall. To increase search recall (at the expense of slightly reduced search performance), you should increase the PRAGMA value, for example:

```
PRAGMA ydb.KMeansTreeSearchTopSize="10";

SELECT *
    FROM TableName VIEW IndexName
    ORDER BY Knn::CosineDistance(embedding, $target)
    LIMIT 10
```

Examples

- Select all the fields from the `series` row-oriented table using the `views_index` vector index created for `embedding` and cosine similarity:

  ```
  PRAGMA ydb.KMeansTreeSearchTopSize="10";
  SELECT series_id, title, info, release_date, views, uploaded_user_id, Knn::CosineSimilarity(embedding,
  $target) as similarity
      FROM series VIEW views_index
      ORDER BY similarity DESC
      LIMIT 10
  ```

- Select all the fields from the `series` row-oriented table using the `views_filtered_index` filtered vector index created for `embedding` and optimized for efficient filtering by `release_date`:

  ```
  PRAGMA ydb.KMeansTreeSearchTopSize="10";
  SELECT series_id, title, info, release_date, views, uploaded_user_id, Knn::CosineSimilarity(embedding,
  $target) as similarity
      FROM series VIEW views_filtered_index
      WHERE release_date = "2025-03-31"
      ORDER BY similarity DESC
      LIMIT 10
  ```

VIEW (Fulltext index)

To select data from a row-oriented table using a fulltext index, use the `VIEW` expression:

```
SELECT ...
FROM TableName VIEW IndexName
WHERE FulltextMatch(TextColumn, "query")
ORDER BY ...
```

Note

A fulltext index isn't automatically selected by the optimizer and must be specified explicitly using `VIEW IndexName`.

Fulltext search functions (`FulltextMatch`, `FulltextScore`) require `VIEW`. If `VIEW` isn't used, the query fails.

Only one fulltext predicate is supported per read through `VIEW`. `FulltextMatch` / `FulltextScore` can't be used under `OR` or `NOT`.
For relevance access, you must include `FulltextScore(...) > 0` in `WHERE`.

For function details, see Fulltext search, including
FulltextMatch and
FulltextScore.

FulltextMatch

FulltextMatch(text, query) filters rows by whether the text matches the fulltext query:

```
SELECT id, title
FROM articles VIEW ft_idx
WHERE FulltextMatch(body, "machine learning")
LIMIT 20;
```

Only the first two arguments can be positional. Additional parameters must be passed as **named arguments**:

- `Mode` (String): query mode:
  - `Keywords` (default) — the query is split into individual terms; how they are combined is determined by `DefaultOperator`
  - `Query` — extended syntax with logical operators: required terms via `+`, excluded terms via `-`, exact phrases in double quotes
  - `Wildcard` — wildcard search: `%` matches any substring, `_` matches a single character (similar to `LIKE`); requires an n-gram index
- `DefaultOperator` (String): term combination operator in `Keywords` mode:
  - `And` (default) — all query terms must be present in the text
  - `Or` — terms split into **required** (prefixed with `+`, must appear in every result) and **optional** (no prefix, count toward `MinimumShouldMatch`). Without any `+` prefix, all terms are optional and at least one must match (see example)
- `MinimumShouldMatch` (String): minimum number of **optional** terms that must match when `DefaultOperator = "Or"` — specified as an absolute number (for example, `"3"`) or a percentage of the optional terms (for example, `"50%"`). Required (`+`) terms are not counted

`Wildcard` mode and `%` / `_` patterns (requires n-grams)

If the index is created with n-gram filtering, you can use `%` and `_` patterns similar to `LIKE`.
To explicitly enable this behavior, pass `"Wildcard" AS Mode` as a named argument:

```
SELECT id, title
FROM articles VIEW ft_idx
WHERE FulltextMatch(body, "mach% learn%", "Wildcard" AS Mode)
LIMIT 20;
```

LIKE / ILIKE (use the fulltext index)

For fulltext indexes with n-grams, `LIKE` / `ILIKE` predicates over the indexed text column use the same logic as `FulltextMatch(..., "Wildcard" AS Mode)`:

```
SELECT id, title
FROM articles VIEW ft_idx
WHERE body ILIKE "%learn%ing%"
LIMIT 20;
```

FulltextScore (BM25 relevance)

FulltextScore(text, query) returns a relevance score and can be used for ranking.
Relevance scoring requires the fulltext_relevance index type.

```
SELECT id, title, FulltextScore(body, "machine learning") AS relevance
FROM articles VIEW ft_idx
WHERE FulltextScore(body, "machine learning") > 0
ORDER BY relevance DESC
LIMIT 10;
```

Optional parameters

Additional parameters must be passed as **named arguments**:

- `DefaultOperator` (String): term combination operator — `And` (default, all terms must be present) or `Or` (terms split into **required** with `+` prefix and **optional** without; at least `MinimumShouldMatch` optional terms must match, and every required term must be present)
- `MinimumShouldMatch` (String): when `DefaultOperator = "Or"`, minimum number of **optional** terms that must match — specified as an absolute number (for example, `"2"`) or a percentage of the optional terms (for example, `"50%"`). Required (`+`) terms are not counted
- `K1` (Double): term frequency saturation parameter in BM25 — controls how strongly repeated occurrences of a term affect the score; typical range: 1.2–2.0
- `B` (Double): document length normalization parameter in BM25 — `0.0` disables normalization, `1.0` fully normalizes by document length; typical value: 0.75

Example:

```
SELECT id, FulltextScore(body, "machine learning", "Or" AS DefaultOperator, "50%" AS MinimumShouldMatch) AS r
elevance
FROM articles VIEW ft_idx
WHERE FulltextScore(body, "machine learning", "Or" AS DefaultOperator, "50%" AS MinimumShouldMatch) > 0
ORDER BY relevance DESC;
```

Note

Only the first two arguments of `FulltextMatch` / `FulltextScore` can be positional. Use named arguments for all additional parameters.

The `FulltextScore(...)` expression is repeated in full in both `SELECT` and `WHERE` — YQL, like standard SQL, evaluates `WHERE` before `SELECT`, so aliases defined in `SELECT` are not available in `WHERE`. Both occurrences must be identical.

VIEW (JSON index)

To execute a `SELECT` query on a row table with explicit use of a JSON index, use the `VIEW` expression:

```
SELECT ...
FROM documents VIEW json_idx
WHERE <предикат на основе JSON_EXISTS / JSON_VALUE>
ORDER BY ...
```

In the query example above, `documents` is the name of the table containing a column of type `Json` or `JsonDocument`, and `json_idx` is the name of the JSON index created on that column.

If the predicate is not supported for execution via a JSON index, a query with an explicit `VIEW` expression fails with a compile-time error. Without the `VIEW` expression, the optimizer cannot select this index for such a predicate, and the query will be executed by another method (for example, selecting a different index or performing a full scan of the base table).

Note

A JSON index can be selected automatically by the optimizer if the predicate meets the requirements for index usage. For debugging and guaranteed index usage, specify it explicitly using `VIEW IndexName`.

For a description of supported predicates, see the Supported predicates section.

Automatic index selection

If the `WHERE` predicate contains `JSON_EXISTS` / `JSON_VALUE` calls on an indexed JSON column, the optimizer may use the JSON index without an explicit `VIEW`. Automatic selection follows these rules:

- The JSON index is considered by the optimizer with the lowest priority — it is a fallback option that is selected only after other access methods have been ruled out.
- The JSON index is not selected if the query can already be served by a primary key or another, more specific secondary index.
- Explicit specification of `VIEW` overrides the optimizer's decision and forces index usage.
- All indexed subexpressions in a single query must refer to the same indexed JSON column. If predicates on different indexed columns are combined via `AND` / `OR`, automatic selection is not applied.

Note

Automatic JSON index selection is rule-based: it uses the query structure and schema metadata, without data statistics. The selection logic may change in future versions of YDB, so for guaranteed index usage, specify it explicitly via `VIEW`.

JSON_EXISTS

JSON_EXISTS(doc, jsonpath) checks for the existence of a path or value inside a JSON document:

```
SELECT id, payload
FROM documents VIEW json_idx
WHERE JSON_EXISTS(payload, '$.user.id');
```

The JSON index supports almost all JsonPath syntax, except for nested predicates and boolean expressions at the context object level (`$`) in `JSON_EXISTS`. A more complex example of a supported expression:

```
SELECT id, payload
FROM documents VIEW json_idx
WHERE JSON_EXISTS(payload, '$.user ? (@.id > 100)');
```

JSON_VALUE

JSON_VALUE(doc, jsonpath RETURNING) extracts a scalar value by a JsonPath and returns it in the specified type. To use the JSON index, the `RETURNING <type>` clause is required:

```
SELECT id, payload
FROM documents VIEW json_idx
WHERE JSON_VALUE(payload, '$.user.name' RETURNING Utf8) = "Charlie"u;
```

When checking equality conditions or whether a value is in a given list (`IN`), the index search is performed by the 'path + value' token, which provides the highest selectivity. For other comparisons (`!=` , `<` , `>=` , `BETWEEN` , etc.), only the path token is used, and the final comparison accuracy is ensured by a post-filter.

Parameters

Three ways to pass query parameters are supported:

1. Direct comparison of the `JSON_VALUE` result with a parameter:

   ```
   DECLARE $id AS Int64;
   SELECT * FROM documents VIEW json_idx
   WHERE JSON_VALUE(payload, '$.owner_id' RETURNING Int64) = $id;
   ```

2. Checking whether the result is in a list of values:

```
DECLARE $tags AS List<Utf8>;
SELECT * FROM documents VIEW json_idx
WHERE JSON_VALUE(payload, '$.tag' RETURNING Utf8) IN $tags;
```

3. Passing a parameter to JsonPath via the `PASSING` clause:

```
DECLARE $v AS Int64;
SELECT * FROM documents VIEW json_idx
WHERE JSON_EXISTS(payload, '$.k ? (@ == $v)' PASSING $v AS v);
```

AND and OR combinations

`JSON_EXISTS` and `JSON_VALUE` on the same JSON column can be combined in a single `WHERE` using the `AND` and `OR` operators:

```
SELECT id, payload
FROM documents VIEW json_idx
WHERE JSON_VALUE(payload, '$.user.name' RETURNING Utf8) = "Charlie"u
  AND JSON_VALUE(payload, '$.user.id' RETURNING Int64) BETWEEN 100 AND 200;
```

Note

Query execution via a JSON index can only use a subset of expressions based on `JSON_EXISTS` and `JSON_VALUE` (see Supported predicates). Conditions that do not fall under these rules (for example, negations, comparisons with another column, comparison of two `JSON_VALUE` from different columns) are not indexed; in `AND` they go into a post-filter, in `OR` they lead to rejection of index usage for the entire group of conditions combined via `OR`.

Hybrid search (HybridRank)

Hybrid search fuses fulltext and vector ranking into a single result. A hybrid query is a `SELECT` over the base table whose `ORDER BY` key is a single `HybridRank` call:

```
PRAGMA ydb.KMeansTreeSearchTopSize = "10";

$queryText = "machine learning";
$queryVector = Knn::ToBinaryStringFloat([0.1, 0.2, 0.3, 0.4]);

SELECT id, title
FROM documents
ORDER BY HybridRank(
    FullTextScore(text, $queryText),
    Knn::CosineDistance(embedding, $queryVector))
LIMIT 10;
```

> **Note**
>
> A hybrid query reads through the base table and does **not** use `VIEW IndexName`: each branch's index is resolved from the corresponding `HybridRank` argument.
>
> The query requires both a fulltext_relevance index over the column passed to `FullTextScore` and a (non-prefixed) vector_kmeans_tree index over the column passed to `Knn`. The recall of the vector branch is controlled by KMeansTreeSearchTopSize.

HybridRank

The general form of the function:

```
HybridRank(
    score1, score2 [, ... scoreN]        -- 2+ scoring expressions: FullTextScore(...) or Knn::<Distance|Simi
larity>(...)
    [, "rrf" | "linear"  AS Mode]        -- fusion method (default "rrf")
    [, (w1, w2, ...)     AS Weights]     -- per-branch weights (default 1.0 each)
    [, k                 AS K]           -- RRF constant (default 60.0)
    [, true | false      AS Normalize]   -- "linear" mode only: min-max normalize (default true)
    [, (idx1, idx2, ...) AS Indexes]     -- per-branch index names (default: auto-detect)
    [, (lim1, lim2, ...) AS Limits]      -- per-branch candidate-pool sizes (default: LIMIT * 10)
    [, ($x) -> {...}     AS RankLambda]  -- custom fusion over per-branch ranks (replaces built-in Mode)
    [, ($x) -> {...}     AS ScoreLambda] -- custom fusion over per-branch raw scores (replaces built-in Mode)
)
```

`HybridRank` may only appear as the entire sort key of an `ORDER BY` clause. It takes **two or more scoring expressions** (positional), followed by optional **named arguments**.

Each scoring expression is one *branch* of the fusion, classified by its form:

- a FullTextScore(text, query) expression — a **fulltext** branch, resolved against a `fulltext_relevance` index over the `text` column;
- a Knn distance or similarity over a column (for example `Knn::CosineDistance(embedding, $queryVector)` or `Knn::CosineSimilarity(embedding, $queryVector)`) — a **vector** branch, resolved against a `vector_kmeans_tree` index over that column.

The argument order is not significant, and there may be more than two branches (for example a fulltext branch plus two vector branches). Each branch retrieves its own candidate pool and contributes one term to every document's fused score.

Fusion modes

Two fusion methods are available via the `Mode` argument:

- `rrf` (default) — Reciprocal Rank Fusion. Each branch contributes `weight / (K + rank)`, where `rank` is the document's position within that branch. Because RRF uses ranks rather than raw scores, the (non-comparable) magnitudes of the branch scores do not matter.
- `linear` — a weighted sum of the per-branch scores: `weight * score` summed over branches. By default the per-branch scores are min-max normalized to `[0, 1]` before summing; set `Normalize` to `false` to fuse the raw scores.

In both modes a document that is absent from a branch's candidate pool simply does not receive that branch's contribution.

Custom fusion lambda

Instead of the built-in `Mode` (`rrf` / `linear`), you can supply a custom fusion lambda that computes the fused score from the per-branch values yourself. Two mutually exclusive lambda kinds are available:

- `RankLambda` — the lambda receives the document's per-branch ranks as a `Dict<Int64, Int64>` (branch index → 1-based rank within that branch). A branch the document is absent from has no entry, so `$ranks[i]` is `NULL`.
- `ScoreLambda` — the lambda receives the document's per-branch raw scores as a `Dict<Int64, Double>` (branch index → the branch's score value: the fulltext relevance, or the vector distance/similarity). A branch the document is absent from has no entry, so `$scores[i]` is `NULL`.

The lambda takes a single argument (the dict) and must return a numeric value (`Double` is recommended); larger values rank higher. The lambda replaces the built-in fusion, so it cannot be combined with `Mode`, `Weights`, `K`, or `Normalize` — fold any weights and constants into the lambda body.

```
SELECT id, title
FROM documents
ORDER BY HybridRank(
```

```
    FullTextScore(text, $queryText),
    Knn::CosineDistance(embedding, $queryVector),
    ($ranks) -> {
        RETURN 1.0 / (60 + COALESCE($ranks[0], 100000))
             + 1.0 / (60 + COALESCE($ranks[1], 100000));
    } AS RankLambda)
LIMIT 10;
```

Named arguments

All optional parameters are passed as named arguments. The `Weights`, `Limits`, and `Indexes` arguments are tuples positionally parallel to the scoring expressions — they must have exactly one entry per branch.

Argument	Type	Description
`Mode`	String	Fusion method: `"rrf"` (default) or `"linear"`. Case-insensitive (`"RRF"` / `"Linear"` are accepted).
`Weights`	Tuple of numbers	Per-branch weight, one per scoring argument. Default `1.0` for every branch. A weight of `0` disables a branch.
`K`	Double	The RRF constant (`rrf` mode only). Default `60.0`. Larger values flatten the influence of top ranks.
`Normalize`	Bool	`linear` mode only. When `true` (default), per-branch scores are min-max normalized before summing; when `false`, raw scores are fused.
`Indexes`	Tuple of strings	Explicit index name per branch, overriding auto-detection. Required when a branch's column is covered by more than one matching index.
`Limits`	Tuple of positive integers	Explicit candidate-pool size per branch. By default each branch retrieves `LIMIT * HybridSearchFactor` (factor default `10`) candidates.
`RankLambda`	Lambda	Custom fusion over per-branch ranks. The lambda receives a `Dict<Int64, Int64>` (branch index → 1-based rank; absent branches have no entry) and returns a numeric score. Replaces the built-in `Mode`; cannot be combined with `Mode`, `Weights`, `K`, or `Normalize`.
`ScoreLambda`	Lambda	Custom fusion over per-branch raw scores. The lambda receives a `Dict<Int64, Double>` (branch index → raw branch score; absent branches have no entry) and returns a numeric score. Replaces the built-in `Mode`; cannot be combined with `Mode`, `Weights`, `K`, or `Normalize`.

Examples

Weighted RRF — bias the ranking toward the vector branch:

```
$queryText = "machine learning";
$queryVector = Knn::ToBinaryStringFloat([0.1, 0.2, 0.3, 0.4]);

SELECT id, title
FROM documents
ORDER BY HybridRank(
    FullTextScore(text, $queryText),
    Knn::CosineDistance(embedding, $queryVector),
    (1, 2) AS Weights)
LIMIT 10;
```

Three-branch RRF — combine text relevance with two vector branches over different embedding columns:

```
$queryText = "machine learning";
$titleVector = Knn::ToBinaryStringFloat([0.1, 0.2, 0.3, 0.4]);
$bodyVector = Knn::ToBinaryStringFloat([0.7, 0.8, 0.9, 1.0]);

SELECT id, title
FROM documents
ORDER BY HybridRank(
    FullTextScore(text, $queryText),
    Knn::CosineDistance(title_embedding, $titleVector),
    Knn::CosineDistance(body_embedding, $bodyVector),
    (1, 2, 3) AS Weights)
LIMIT 10;
```

Linear fusion over normalized scores:

```
$queryText = "machine learning";
$queryVector = Knn::ToBinaryStringFloat([0.1, 0.2, 0.3, 0.4]);

SELECT id, title
FROM documents
ORDER BY HybridRank(
    FullTextScore(text, $queryText),
    Knn::CosineDistance(embedding, $queryVector),
    "linear" AS Mode)
LIMIT 10;
```

Explicit index names and candidate-pool sizes. Because the per-branch pool sizes are given by `Limits`, the query's own `LIMIT` no longer needs to be a literal and may be a parameter:

```
DECLARE $limit AS Uint64;

$queryText = "machine learning";
$queryVector = Knn::ToBinaryStringFloat([0.1, 0.2, 0.3, 0.4]);

SELECT id, title
FROM documents
ORDER BY HybridRank(
    FullTextScore(text, $queryText),
    Knn::CosineDistance(embedding, $queryVector),
    ("ft_idx", "vec_idx") AS Indexes,
    (100, 200) AS Limits)
LIMIT $limit;
```

Custom `RankLambda` — spell out RRF by hand, weighting the vector branch 100× more than the text branch:

```
$queryText = "machine learning";
$queryVector = Knn::ToBinaryStringFloat([0.1, 0.2, 0.3, 0.4]);

SELECT id, title
FROM documents
ORDER BY HybridRank(
    FullTextScore(text, $queryText),
    Knn::CosineDistance(embedding, $queryVector),
    ($ranks) -> {
        RETURN   1.0 / (60 + COALESCE($ranks[0], 100000))
             + 100.0 / (60 + COALESCE($ranks[1], 100000));
    } AS RankLambda)
LIMIT 10;
```

Custom `ScoreLambda` — combine the raw fulltext relevance and the negated vector distance (closer = larger score):

```
$queryText = "machine learning";
$queryVector = Knn::ToBinaryStringFloat([0.1, 0.2, 0.3, 0.4]);

SELECT id, title
FROM documents
ORDER BY HybridRank(
    FullTextScore(text, $queryText),
    Knn::CosineDistance(embedding, $queryVector),
    ($scores) -> {
        RETURN COALESCE($scores[0], 0.0) - COALESCE($scores[1], 1000000.0);
    } AS ScoreLambda)
LIMIT 10;
```

Limitations

- `HybridRank(...)` must be the entire `ORDER BY` key; it cannot be negated, nested in a larger expression, or combined with other sort keys.
- At least two scoring arguments are required — a single branch is not a hybrid query.
- `LIMIT` must be a literal (it sizes the per-branch candidate pools). Use an explicit `AS Limits` to allow a parameterized `LIMIT`.
- A custom fusion lambda (`RankLambda` or `ScoreLambda`) replaces the built-in fusion and cannot be combined with `Mode`, `Weights`, `K`, or `Normalize`. At most one of `RankLambda` or `ScoreLambda` may be specified.
- Prefixed vector indexes are not supported yet.

WITH

Specified after the data source in `FROM` and used to provide additional hints for table usage. Hints cannot be specified for subqueries and named expressions.

The following values are supported:

- `INFER_SCHEMA` — sets the flag for inferring the table schema. The behavior is similar to setting the yt.InferSchema pragma, but only for a specific data source. You can specify the number of rows to infer (a number from 1 to 1000).
- `FORCE_INFER_SCHEMA` — sets the flag for forcing schema inference of the table. The behavior is similar to setting the yt.ForceInferSchema pragma, but only for a specific data source. You can specify the number of rows to infer (a number from 1 to 1000).
- `DIRECT_READ` — suppresses the operation of some optimizers and forces the table contents to be used as is. The behavior is similar to setting the debug DirectRead pragma, but only for a specific data source.
- `INLINE` — indicates that the table contents are small and should be processed using an in-memory representation. The actual table size is not checked, and if it is large, the query may fail due to memory exhaustion.
- `UNORDERED` — suppresses the use of the table's original sorting.
- `XLOCK` — indicates that an exclusive lock should be taken on the table. Useful when the table is read during the query metaprogram processing stage and then its contents are updated in the main query. It helps avoid data loss if an external process modifies the table between the execution of the metaprogram phase and the main part of the query.
- `SCHEMA` type — indicates that the specified table schema should be used entirely, ignoring the schema in the metadata.
- `COLUMNS` type — indicates that the specified types should be used for columns whose names match the column names in the table metadata, as well as which additional columns are present in the table.
- `IGNORETYPEV3`, `IGNORE_TYPE_V3` — sets the flag to ignore type_v3 types in the table. The behavior is similar to setting the yt.IgnoreTypeV3 pragma, but only for a specific data source.

When working with external file data sources, you can additionally specify a number of parameters:

- `FORMAT`: data storage format in file storages in federated queries. Allowed values: `csv_with_names`, `tsv_with_names`, `json_list`, `json_each_row`, `json_as_string`, `parquet`, `raw`.
- `COMPRESSION`: file compression format in file storages in federated queries. Valid values: gzip, zstd, lz4, brotli, bzip2, xz.
- `PARTITIONED_BY` - a list of partitioning columns of data in file storages in federated queries. Contains a list of columns in the order they are placed in the file storage.
- `projection.enabled` - a flag for enabling extended data partitioning. Valid values: `true`, `false`.
- `projection.<field_name>.type` - field type of extended data partitioning. Valid values: `integer`, `enum`, `date`.
- `projection.<field_name>.<options>` - extended properties of the field of extended data partitioning.

When reading from a topic in streaming queries, you can specify watermarks parameters:

- `WATERMARK` — expression for calculating the watermark. Currently, only the write time to a topic with a constant delay is supported. Format: `__ydb_write_time - Interval("<delay>")`, where `<delay>` is specified in ISO 8601 format.
- `WATERMARK_GRANULARITY` — watermark generation frequency. The smaller it is, the higher the CPU consumption by the query and the lower the response latency. Only relevant for streaming queries. Specified in ISO 8601 format. Default value — 1 second.
- `WATERMARK_IDLE_TIMEOUT` — period after which an idle partition will be excluded from the combined watermark calculation. Only relevant for streaming queries. Specified in ISO 8601 format. Default value — 5 seconds.

When specifying hints `SCHEMA` and `COLUMNS`, the type value must be a structure type.

Examples

```
SELECT key FROM my_table WITH INFER_SCHEMA;
SELECT key FROM my_table WITH FORCE_INFER_SCHEMA="42";
```

```
$s = (SELECT COUNT(*) FROM my_table WITH XLOCK);

INSERT INTO my_table WITH TRUNCATE
SELECT EvaluateExpr($s) AS a;
```

```
SELECT key, value FROM my_table WITH SCHEMA Struct<key:String, value:Int32>;
```

```
SELECT key, value FROM my_table WITH COLUMNS Struct<value:Int32?>;
```

```
SELECT key, value FROM EACH($my_tables) WITH SCHEMA Struct<key:String, value:List<Int32>>;
```

```
SELECT
    *
FROM
    my_topic
WITH (
    FORMAT = json_each_row,
    SCHEMA = (
        ts String
    ),
    WATERMARK = __ydb_write_time - Interval("PT5S"),
    WATERMARK_GRANULARITY = "PT1S",
    WATERMARK_IDLE_TIMEOUT = "PT5S"
);
```

WITHOUT

Excluding columns from the result of `SELECT *`.

Examples

```
SELECT * WITHOUT foo, bar FROM my_table;
```

```
PRAGMA simplecolumns;
SELECT * WITHOUT t.foo FROM my_table AS t
CROSS JOIN (SELECT 1 AS foo) AS v;
```

WHERE

Filtering rows in the `SELECT` result based on a condition in row-oriented or column-oriented.

Example

```
SELECT key FROM my_table
WHERE value > 0;
```

Warning

Correlated subqueries in `WHERE` are not supported: a subquery cannot refer to columns or table aliases from an outer query. To select rows that have matching rows in another table, use `LEFT SEMI JOIN`. To select rows without matches, use `LEFT ONLY JOIN`.

ORDER BY

Sorting the `SELECT` result using a comma-separated list of sorting criteria. As a criteria, you can use a column value or an expression on columns. Ordering by column sequence number is not supported (`ORDER BY N` where `N` is a number).

Each criteria can be followed by the sorting direction:

- `ASC` : Sorting in the ascending order. Applied by default.
- `DESC` : Sorting in the descending order.

Multiple sorting criteria will be applied left-to-right.

Example

```
SELECT key, string_column
FROM my_table
ORDER BY key DESC, LENGTH(string_column) ASC;
```

You can also use `ORDER BY` for window functions.

ASSUME ORDER BY

Checking that the `SELECT` result is sorted by the value in the specified column or multiple columns. The result of such a `SELECT` statement is treated as sorted, but without actually running a sort. Sort check is performed at the query execution stage.

As in case of `ORDER BY`, it supports setting the sort order using the keywords `ASC` (ascending order) and `DESC` (descending order). Expressions are not supported in `ASSUME ORDER BY`.

Examples

```
SELECT key || "suffix" as key, -CAST(subkey as Int32) as subkey
FROM my_table
ASSUME ORDER BY key, subkey DESC;
```

LIMIT and OFFSET

`LIMIT`: limits the output to the specified number of rows. By default, the output is not restricted.

`OFFSET`: specifies the offset from the beginning (in rows). By default, it's zero.

Examples

```
SELECT key FROM my_table
LIMIT 7;
```

```
SELECT key FROM my_table
LIMIT 7 OFFSET 3;
```

```
SELECT key FROM my_table
LIMIT 3, 7; -- equivalent to the previous example
```

TABLESAMPLE and SAMPLE

> **Warning**
>
> Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Building a random sample from the data source specified in `FROM`.

`TABLESAMPLE` is part of the SQL standard and works as follows:

- The operating mode is specified:
  - `BERNOULLI` means "slowly, straightforwardly going through all the data, but in a truly random way".
  - `SYSTEM` uses knowledge about the physical data storage of data to avoid full data scans, but somewhat sacrificing randomness of the sample.

The data is split into sufficiently large blocks, and the whole data blocks are sampled. For applied calculations on sufficiently large tables, the result may well be consistent.

- The size of the random sample is indicated as a percentage after the operating mode, in parentheses.
- To manage the block size in the `SYSTEM` mode, use the `yt.SamplingIoBlockSize` pragma.
- Optionally, it can be followed by the `REPEATABLE` keyword and an integer in parentheses to be used as a seed for a pseudorandom number generator.

`SAMPLE` is a shorter alias without sophisticated settings and sample size specified as a fraction. It currently corresponds to the `BERNOULLI` mode.

> **Note**
>
> In the `BERNOULLI` mode, if the `REPEATABLE` keyword is added, the seed is mixed with the chunk ID for each chunk in the table. That's why sampling from different tables with the same content might produce different results.

Examples

```
SELECT *
FROM my_table
TABLESAMPLE BERNOULLI(1.0) REPEATABLE(123); -- one percent of the table
```

```
SELECT *
FROM my_table
TABLESAMPLE SYSTEM(1.0); -- about one percent of the table
```

```
SELECT *
FROM my_table
SAMPLE 1.0 / 3; -- one-third of the table
```

MATCH_RECOGNIZE

The `MATCH_RECOGNIZE` expression performs pattern recognition in a sequence of rows and returns the found results. This functionality is important for various business areas, such as fraud detection, pricing analysis in finance, and sensor data processing. This area is known as Complex Event Processing (CEP), and pattern recognition is a valuable tool for this. An example of how `MATCH_RECOGNIZE` works is provided in the link.

Data processing algorithm

The `MATCH_RECOGNIZE` expression performs the following actions:

1. The input table is divided into non-overlapping groups. Each group consists of a set of rows from the input table with identical values in the columns listed after `PARTITION BY`.
2. Each group is ordered according to the `ORDER BY` clause.
3. Recognition of pattern from `PATTERN` is performed independently in each ordered group.
4. Pattern search in the sequence of rows is a step-by-step process: rows are checked one by one if they fit the pattern. Among all matches starting in the earliest row, the one consisting of the largest number of rows is selected. If no matches were found starting in the earliest row, the search continues starting from the next row.
5. After a match is found, the columns defined by expressions in the `MEASURES` block are calculated.
6. Depending on the `ROWS PER MATCH` mode, one or all rows for the found match are output.
7. The `AFTER MATCH SKIP` mode determines from which row the pattern recognition will resume.

Syntax

```
MATCH_RECOGNIZE (
    [ PARTITION BY <partition_1> [ ... , <partition_N> ] ]
    [ ORDER BY <sort_key_1> [ ... , <sort_key_N> ] ]
    [ MEASURES <expression_1> AS <column_name_1> [ ... , <expression_N> AS <column_name_N> ] ]
    [ ROWS PER MATCH ]
    [ AFTER MATCH SKIP ]
    PATTERN (<search_pattern>)
    DEFINE <variable_1> AS <predicate_1> [ ... , <variable_N> AS <predicate_N> ]
)
```

Here is a brief description of the SQL syntax elements of the `MATCH_RECOGNIZE` expression:

- `DEFINE`: Block for declaring variables that describe the search pattern and the conditions that rows must meet for each variable.
- `PATTERN`: Regular expressions describing the search pattern.
- `MEASURES`: Defines the list of columns for the returned data. Each column is specified by an SQL expression for its computation.
- `ROWS PER MATCH`: Determines the structure of the returned data and the number of rows for each match found.
- `AFTER MATCH SKIP`: Defines the method of moving to the point of the next match search.
- `ORDER BY`: Determines sorting of input data. Pattern search is performed within the data sorted according to the list of columns or expressions listed in `<sort_key_1> [ ... , <sort_key_N> ]`.
- `PARTITION BY`: Divides the input table according to the specified rules in accordance with `<partition_1> [ ... , <partition_N> ]`. Pattern search is performed independently in each part.

DEFINE

```
DEFINE <variable_1> AS <predicate_1> [ ... , <variable_N> AS <predicate_N> ]
```

`DEFINE` declares variables that are used to describe the desired pattern defined in `PATTERN`. Variables are named SQL statements evaluated over the input data. The syntax of the SQL statements in `DEFINE` is the same as the SQL statements of the `WHERE` predicate. For example, the `button = 1` expression searches for rows with the value `1` in the `button` column. Any SQL expressions that can be used to perform a search, including aggregation functions (`LAST`, `FIRST`). For example, `button > 2 AND zone_id < 12` or `LAST(button) > 10`.

In the example below, the SQL statement `A.button = 1` is declared as variable `A`.

```
DEFINE
    A AS A.button = 1
```

> **Note**
>
> `DEFINE` does not currently support aggregation functions (e.g., `AVG`, `MIN`, or `MAX`) and `PREV` and `NEXT` functions.

When processing each row of data, all SQL statements describing variables in `DEFINE` are calculated. When the SQL-expression describing the corresponding variable from `DEFINE` gets the `TRUE` value, such a row is labeled with the `DEFINE` variable name and added to the list of rows subject to pattern matching.

Example

When defining variables in SQL expressions, you can reference other variables:

```
DEFINE
    A AS A.button = 1 AND LAST(A.zone_id) = 12,
    B AS B.button = 2 AND FIRST(A.zone_id) = 12
```

An input data row will be computed as variable `A` if it contains a `button` column with value `1` and the last row of the set of previously matched `A` has a `zone_id` column with value `12`. The row will be computed as variable `B` if the data row contains a `button` column with value `2` and the first row of the set of previously matched variables `A` has a `zone_id` column with value `12`.

PATTERN

```
PATTERN (<search_pattern>)
```

The `PATTERN` keyword describes the search pattern in the format derived from variables in the `DEFINE` section. The `PATTERN` syntax is similar to the one of regular expressions.

> **Warning**
>
> If a variable used in the `PATTERN` section has not been previously described in the `DEFINE` section, it is assumed that it is always `TRUE`.

You can use quantifiers in `PATTERN`. In regular expressions, they determine the number of repetitions of an element or subsequence in the matched pattern. Here is the list of supported quantifiers:

Quantifier	Description
`A+`	One or more occurrences of `A`
`A*`	Zero or more occurrences of `A`
`A?`	Zero or one occurrence of `A`
`B{n}`	Exactly `n` occurrences of `B`
`C{n, m}`	From `n` to `m` occurrences of `C`
`D{n,}`	At least `n` occurrences of `D`
`(A|B)`	Occurrence of `A` or `B` in the data
`(A|B){,m}`	From zero to `m` occurrences of `A` or `B`

Supported pattern search sequences:

Supported sequences	Syntax	Description
Sequence	`A B+ C+ D+`	The system searches for the exact specified sequence, the occurrence of other variables within the sequence is not allowed. The pattern search is performed in the order of the pattern variables.
One of	`A | B | C`	Variables are listed in any order with a pipe | between them. The search is performed for any variable from the specified list.
Grouping	`(A | B)+ | C`	Variables inside round brackets are considered a single group. In this case, quantifiers apply to the entire group.
Exclusion from result	`{- A B+ C -}`	Rows found by the pattern in parentheses will be excluded from the result in `ALL ROWS PER MATCH` mode

Example

```
PATTERN (B1 E* B2+ B3)
DEFINE
    B1 as B1.button = 1,
    B2 as B2.button = 2,
    B3 as B3.button = 3
```

The `DEFINE` section describes the `B1`, `B2`, and `B3` variables, while it does not describe `E`. Such notation allows interpreting `E` as any event, so the following pattern will be searched: one `button 1` click, one or more `button 2` clicks, and one `button 3` click. Meanwhile, between a click of `button 1` and `button 2`, any number of any other events may occur.

MEASURES

```
MEASURES <expression_1> AS <column_name_1> [ ... , <expression_N> AS <column_name_N> ]
```

`MEASURES` describes the set of returned columns when a pattern is found. A set of returned columns should be represented by an SQL expression with the aggregate functions over the variables declared in the `DEFINE` statement.

Example

The input data for the example are:

ts	button	device_id	zone_id
100	1	3	0
200	1	3	1
300	2	2	0
400	3	1	1

```
MEASURES
    AGGREGATE_LIST(B1.zone_id * 10 + B1.device_id) AS ids,
    COUNT(DISTINCT B1.zone_id) AS count_zones,
    LAST(B3.ts) - FIRST(B1.ts) AS time_diff,
    42 AS meaning_of_life
PATTERN (B1+ B2 B3)
DEFINE
    B1 AS B1.button = 1,
    B2 AS B2.button = 2,
    B3 AS B3.button = 3
```

Result:

ids	count_zones	time_diff	meaning_of_life
[3,13]	2	300	42

The `ids` column contains the list of `zone_id * 10 + device_id` values counted among the rows matched with the `B1` variable. The `count_zones` column contains the number of unique values of the `zone_id` column among the rows matched with the `B1` variable. Column `time_diff` contains the difference between the value of column `ts` in the last row of the set of rows matched with variable `B3` and the value of column `ts` in the first row of the set of rows matched with variable `B1`. The `meaning_of_life` column contains the constant `42`. Thus, an expression in `MEASURES` may contain aggregate functions over multiple variables, but there must be only one variable within a single aggregate function.

ROWS PER MATCH

`ROWS PER MATCH` determines the number of result rows for each match found, as well as the number of columns returned. The default mode is `ONE ROW PER MATCH`.

`ONE ROW PER MATCH` sets the `ROWS PER MATCH` mode to output one row for the match found. The structure of the returned data corresponds to the columns listed in `PARTITION BY` and `MEASURES`.

`ALL ROWS PER MATCH` sets the `ROWS PER MATCH` mode to output all rows of the match found except explicitly excluded by parentheses. In addition to the columns of the source table, the structure of the returned data includes the columns listed in the `MEASURES`.

Examples

The input data for all examples are:

ts	button
100	1
200	2
300	3

Example 1

```
MEASURES
    FIRST(B1.ts) AS first_ts,
    FIRST(B2.ts) AS mid_ts,
    LAST(B3.ts) AS last_ts
ONE ROW PER MATCH
PATTERN (B1 {- B2 -} B3)
DEFINE
    B1 AS B1.button = 1,
    B2 AS B2.button = 2,
    B3 AS B3.button = 3
```

Result:

first_ts	mid_ts	last_ts
100	200	300

Example 2

```
MEASURES
    FIRST(B1.ts) AS first_ts,
    FIRST(B2.ts) AS mid_ts,
    LAST(B3.ts) AS last_ts
ALL ROWS PER MATCH
PATTERN (B1 {- B2 -} B3)
DEFINE
    B1 AS B1.button = 1,
    B2 AS B2.button = 2,
    B3 AS B3.button = 3
```

Result:

first_ts	mid_ts	last_ts	button	ts
100	200	300	1	100
100	200	300	3	300

AFTER MATCH SKIP

`AFTER MATCH SKIP` determines the method of transitioning from the found match to searching for the next one. In the `AFTER MATCH SKIP TO NEXT ROW` mode, the search for the next match starts after the first row of the previous one, while in the `AFTER MATCH SKIP PAST LAST ROW` mode it starts after the last row of the previous match. The default mode is `PAST LAST ROW`.

Examples

The input data for all examples are:

ts	button
100	1
200	1
300	2
400	3

Example 1

```
MEASURES
    FIRST(B1.ts) AS first_ts,
    LAST(B3.ts) AS last_ts
AFTER MATCH SKIP TO NEXT ROW
PATTERN (B1+ B2 B3)
DEFINE
    B1 AS B1.button = 1,
    B2 AS B2.button = 2,
    B3 AS B3.button = 3
```

Result:

first_ts	last_ts
100	400
200	400

Example 2

```
MEASURES
    FIRST(B1.ts) AS first_ts,
    LAST(B3.ts) AS last_ts
AFTER MATCH SKIP PAST LAST ROW
PATTERN (B1+ B2 B3)
DEFINE
    B1 AS B1.button = 1,
    B2 AS B2.button = 2,
    B3 AS B3.button = 3
```

Result:

first_ts	last_ts
100	400

ORDER BY

```
ORDER BY <sort_key_1> [ ... , <sort_key_N> ]

<sort_key> ::= { <column_names> | <expression> }
```

`ORDER BY` determines sorting of the input data. That is, before all pattern search operations are performed, the data will be pre-sorted according to the specified keys or expressions. The syntax is similar to the `ORDER BY` SQL expression.

Example

```
ORDER BY CAST(ts AS Timestamp)
```

PARTITION BY

```
PARTITION BY <partition_1> [ ... , <partition_N> ]

<partition> ::= { <column_names> | <expression> }
```

`PARTITION BY` partitions the source data into multiple non-overlapping groups, each used for an independent pattern search. If the expression is not specified, all data is processed as a single group. Records with the same values of the columns listed after `PARTITION BY` fall into the same group.

Example

```
PARTITION BY device_id, zone_id
```

Limitations

Our support for the `MATCH_RECOGNIZE` expression will eventually comply with SQL-2016; currently, however, the following limitations apply:

- `MEASURES`. Functions `PREV` / `NEXT` are not supported.
- `AFTER MATCH SKIP`. Only the `AFTER MATCH SKIP TO NEXT ROW` and `AFTER MATCH SKIP PAST LAST ROW` modes are supported.
- `PATTERN`. Union pattern variables are not implemented.
- `DEFINE`. Aggregation functions are not supported.

Example of usage

Here is a hands-on example of pattern recognizing in a data table produced by an IoT device, where pressing its buttons triggers certain events. Let's assume you need to find and process the following sequence of button clicks: `button 1`, `button 2`, and `button 3`.

The structure of the data to transmit is as follows:

ts	button	device_id	zone_id
600	3	17	3
500	3	4	2
400	2	17	3
300	2	4	2
200	1	17	3
100	1	4	2

The body of the SQL query looks like this:

```
PRAGMA FeatureR010="prototype"; -- pragma for enabling MATCH_RECOGNIZE

SELECT * FROM input MATCH_RECOGNIZE ( -- Performing pattern matching from input
    PARTITION BY device_id, zone_id -- Partitioning the input data into groups by columns device_id and zone_
id
    ORDER BY ts -- Viewing events based on the ts column data sorted ascending
    MEASURES
        LAST(B1.ts) AS b1, -- Going to get the latest timestamp of clicking button 1 in the query results
        LAST(B3.ts) AS b3  -- Going to get the latest timestamp of clicking button 3 in the query results
    ONE ROW PER MATCH            -- Going to get one result row per match hit
    AFTER MATCH SKIP TO NEXT ROW -- Going to move to the next row once the match is found
    PATTERN (B1 B2+ B3) -- Searching for a pattern that includes one button 1 click, one or more button 2 cli
cks, and one button 3 click
    DEFINE
        B1 AS B1.button = 1, -- Defining the B1 variable as event of clicking button 1 (the button field equa
ls 1)
        B2 AS B2.button = 2, -- Defining the B2 variable as event of clicking button 2 (the button field equa
ls 2)
        B3 AS B3.button = 3  -- Defining the B3 variable as event of clicking button 3 (the button field equa
ls 3)
);
```

Result:

b1	b3	device_id	zone_id
100	500	4	2
200	600	17	3

Streaming read from a topic

You can read data from a topic using a regular `SELECT` without creating a streaming query. To do this, specify `STREAMING = "TRUE"` in the `WITH` block and set a limit on the number of output rows via `LIMIT`; otherwise, the query will not complete.

Warning

This method is intended only for debugging and checking data in a topic. For production processes, create streaming queries using CREATE STREAMING QUERY.

Note

In the examples:

- `ext_source` is a pre-created external data source
- `input_topic` is a local or external topic.

For more information, see local and external topics.

Example

```
SELECT
    Data
FROM
    ext_source.input_topic -- or local topic input_topic
WITH (
    FORMAT = raw,
    SCHEMA = (
        Data String
    ),
    STREAMING = "TRUE"
)
LIMIT 1
```

See also

- Debug reads from a topic — recipe with additional examples
- Streaming queries
- CREATE STREAMING QUERY

SHOW CREATE

`SHOW CREATE` returns a query required to recreate the structure of the specified object: a table or view. The returned query may contain multiple DDL statements.

Syntax

```
SHOW CREATE [TABLE|VIEW] <name>;
```

Parameters

- `TABLE|VIEW` — The object type. Valid values are `TABLE` or `VIEW`.
- `<name>` — The object name. An absolute path may also be specified.

Result

The command always returns **exactly one row** with three columns:

Path	PathType	CreateQuery
Path	Table/View	DDL statements for creation

- **Path** — The path to the object (for example, `MyTable` or `MyView`).
- **PathType** — The type of object: `Table` or `View`.
- **CreateQuery** — The complete set of DDL statements needed to create the object:
  - For tables: the main CREATE TABLE statement (with the path relative to the database root), plus any additional statements describing the current configuration, such as:
    - ALTER TABLE ... ALTER INDEX — for index partitioning settings.
    - ALTER TABLE ... ADD CHANGEFEED — for adding a changefeed.
    - `ALTER SEQUENCE` — for restoring a `Sequence` state for `Serial` columns.
  - For views: the definition via CREATE VIEW, and, if necessary, the statements the view has captured from the creation context, for example, PRAGMA TablePathPrefix.

Examples

Row-oriented tables

```
SHOW CREATE TABLE my_table;
```

Path	PathType	CreateQuery
`my_table`	`Table`	`CREATE TABLE...` — see below

```
CREATE TABLE `my_table` (
    `Key1` Uint32 NOT NULL,
    `Key2` Utf8 NOT NULL,
    `Key3` Serial4 NOT NULL,
    `Value1` Utf8 FAMILY `my_family`,
    `Value2` Bool,
    `Value3` String,
    INDEX `my_index` GLOBAL SYNC ON (`Key2`, `Value1`, `Value2`),
    FAMILY `my_family` (COMPRESSION = 'lz4'),
    PRIMARY KEY (`Key1`, `Key2`, `Key3`)
)
WITH (
    AUTO_PARTITIONING_BY_SIZE = ENABLED,
    AUTO_PARTITIONING_PARTITION_SIZE_MB = 1000
);

ALTER TABLE `my_table`
    ADD CHANGEFEED `feed_3` WITH (MODE = 'KEYS_ONLY', FORMAT = 'JSON', RETENTION_PERIOD = INTERVAL('PT30M'),
TOPIC_MIN_ACTIVE_PARTITIONS = 3)
;

ALTER SEQUENCE `/Root/my_table/_serial_column_Key3` START WITH 101 INCREMENT BY 404 RESTART;

ALTER TABLE `my_table`
    ALTER INDEX `my_index` SET (AUTO_PARTITIONING_BY_LOAD = ENABLED, AUTO_PARTITIONING_MIN_PARTITIONS_COUNT =
1000, AUTO_PARTITIONING_MAX_PARTITIONS_COUNT = 5000)
;
```

Column-oriented tables

```
SHOW CREATE TABLE my_table;
```

Path	PathType	CreateQuery
`my_table`	`Table`	`CREATE TABLE...` — see below

```
CREATE TABLE `my_table` (
    `Key1` Uint64 NOT NULL,
    `Key2` Utf8 NOT NULL,
```

```
    `Key3` Int32 NOT NULL,
    `Value1` Utf8,
    `Value2` Int16,
    `Value3` String,
    PRIMARY KEY (`Key1`, `Key2`, `Key3`)
)
PARTITION BY HASH (`Key1`, `Key2`)
WITH (
    STORE = COLUMN,
    AUTO_PARTITIONING_MIN_PARTITIONS_COUNT = 100,
    TTL =
        INTERVAL('PT10S') TO EXTERNAL DATA SOURCE `/Root/tier1`,
        INTERVAL('PT1H') DELETE
    ON Key1 AS SECONDS
);
```

Views

```
SHOW CREATE VIEW my_view;
```

Path	PathType	CreateQuery
`my_view`	`View`	`PRAGMA TablePathPrefix...` — see below

```
PRAGMA TablePathPrefix = '/Root/DirA/DirB/DirC';

CREATE VIEW `my_view` WITH (security_invoker = TRUE) AS
SELECT
    *
FROM
    test_table
;
```

TRUNCATE TABLE

`TRUNCATE TABLE` removes all user data from the specified table and its indexes.

Syntax

```
TRUNCATE TABLE <table_name>;
```

Limitations

- Supported only for row-oriented tables.
- While the operation runs, the table is locked for reads and writes.
- The operation cannot be interrupted or canceled after it has started.
- The operation cannot be performed if the table has:
  - an asynchronous secondary index,
  - a changefeed,
  - asynchronous replication.

Examples

Removes all data from the table with the full path `/Root/test/my_table`.

```
TRUNCATE TABLE `/Root/test/my_table`;
```

Removes all data from table `my_table` in the current database.

```
TRUNCATE TABLE `my_table`;
```

See also

- CREATE TABLE
- ALTER TABLE
- DROP TABLE

UPDATE

> **Warning**
>
> Currently, mixing column-oriented tables and row-oriented tables in a single transaction is supported only if the transaction performs read operations; no writes are allowed. Support for read-write transactions involving both table types is under development.
>
> If a write transaction includes both types of tables, it fails with the following error: `Write transactions that use both row-oriented and column-oriented tables are disabled at current time`.

Updates the data in the table. After the `SET` keyword, enter the columns where you want to update values and the new values themselves. The list of rows is defined by the `WHERE` clause. If `WHERE` is omitted, the updates are applied to all the rows of the table.

`UPDATE` can't change the value of the primary key columns.

Example

```
UPDATE my_table
SET Value1 = YQL::ToString(Value2 + 1), Value2 = Value2 - 1
WHERE Key1 > 1;
```

UPDATE ON

Updates the data in the table based on the results of a subquery. The set of columns returned by the subquery must be a subset of the table's columns being updated, and all columns of the table's primary key must be present in the returned columns. The data types of the columns returned by the subquery must match the data types of the corresponding columns in the table.

The primary key value is used to search for the rows being updated. For each row found, the values of the non-key columns is replaced with the values returned in the corresponding row of the result of the subquery. The values of the table columns that are missing in the returned columns of the subquery remain unchanged.

Example

```
$to_update = (
    SELECT Key, SubKey, "Updated" AS Value FROM my_table
    WHERE Key = 1
);

UPDATE my_table ON
SELECT * FROM $to_update;
```

UPDATE ... RETURNING

Updates rows and returns their new values in a single operation. It allows to retrieve information about the updated rows in one query, eliminating the need for an additional SELECT statement.

Examples

- Return all values of modified rows

```
UPDATE orders
SET status = 'shipped'
WHERE order_date < '2023-01-01'
RETURNING *;
```

- Return specific columns

```
UPDATE products
SET price = price * 0.9
WHERE category = 'Electronics'
RETURNING product_id, name, price AS new_price;
```

See also

- BATCH UPDATE

UPSERT INTO

> **Warning**
>
> Currently, mixing column-oriented tables and row-oriented tables in a single transaction is supported only if the transaction performs read operations; no writes are allowed. Support for read-write transactions involving both table types is under development.
>
> If a write transaction includes both types of tables, it fails with the following error: `Write transactions that use both row-oriented and column-oriented tables are disabled at current time`.

UPSERT (which stands for UPDATE or INSERT) updates or inserts multiple rows to a table based on a comparison by the primary key. Missing rows are added. For the existing rows, the values of the specified columns are updated, but the values of the other columns are preserved.

`UPSERT` and `REPLACE` are data modification operations that don't require a prefetch and run faster and cheaper than other operations because of that.

Column mapping when using `UPSERT INTO ... SELECT` is done by names. Use `AS` to fetch a column with the desired name in `SELECT`.

Examples

```
UPSERT INTO my_table
SELECT pk_column, data_column1, col24 as data_column3 FROM other_table
```

```
UPSERT INTO my_table ( pk_column1, pk_column2, data_column2, data_column5 )
VALUES ( 1, 10, 'Some text', Date('2021-10-07')),
       ( 2, 10, 'Some text', Date('2021-10-08'))
```

UPSERT INTO ... RETURNING

Inserts or updates a row and returns their values in a single operation. It allows to retrieve information about the affected row in one query, eliminating the need for an additional SELECT statement.

Examples

- Return all values of modified row

```
UPSERT INTO orders (order_id, status, amount)
VALUES (1001, 'shipped', 500)
RETURNING *;
```

- Return specific columns

```
UPSERT INTO users (user_id, name, email)
VALUES (42, 'John Doe', 'john@example.com')
RETURNING user_id, email;
```

UPSERT OBJECT (TYPE SECRET)

> **Warning**
>
> The syntax for managing secrets will change in future YDB releases.

To change the contents of a secret, use the following statement:

```
UPSERT OBJECT `secret_name` (TYPE SECRET) WITH value = `secret_value`;
```

Where:

- `secret_name` — Name of the secret.
- `secret_value` — Secret payload.

Example

The following statement sets the secret named `MySecretName` to `MySecretData`:

```
UPSERT OBJECT `MySecretName` (TYPE SECRET) WITH value = `MySecretData`;
```

Basic VALUES syntax in YQL

VALUES as a top-level operator

It lets you create a table from specified values. For example, this statement creates a table of k columns and n rows:

```
VALUES (expr_11, expr_12, ..., expr_1k),
       (expr_21, expr_22, ..., expr_2k),
       ....
       (expr_n1, expr_n2, ..., expr_nk);
```

This statement is totally equivalent to the following one:

```
SELECT expr_11, expr_12, ..., expr_1k UNION ALL
SELECT expr_21, expr_22, ..., expr_2k UNION ALL
....
SELECT expr_n1, expr_n2, ..., expr_nk;
```

Example

```
VALUES (1,2), (3,4);
```

VALUES after FROM

VALUES can also be used in a subquery, after FROM. For example, the following two queries are equivalent:

```
VALUES (1,2), (3,4);
SELECT * FROM (VALUES (1,2), (3,4));
```

In all the examples above, column names are assigned by YQL and have the format `column0 ... columnN`. To assign arbitrary column names, you can use the following construct:

```
SELECT * FROM (VALUES (1,2), (3,4)) as t(x,y);
```

In this case, the columns will get the names `x`, `y`.

Built-in YQL functions

- Basic
- Aggregate
- Window
- For lists
- For dictionaries
- For structures
- For types
- For code generation
- For JSON
- C++ libraries

Basic built-in functions

Below are the general-purpose functions. For specialized functions, there are separate articles: aggregate functions, window functions, and functions for lists, dictionaries, structures, data types, and code generation.

COALESCE

Iterates through the arguments from left to right and returns the first non-empty argument found. To be sure that the result is non-empty (not of an optional type), the rightmost argument must be of this type (often a literal is used for this). With a single argument, returns this argument unchanged.

Lets you pass potentially empty values to functions that can't handle them by themselves.

A short format using the low-priority `??` operator is available (lower than the Boolean operations). You can use the `NVL` alias.

Examples

```
SELECT COALESCE(
  maybe_empty_column,
  "it's empty!"
) FROM my_table;
```

```
SELECT
  maybe_empty_column ?? "it's empty!"
FROM my_table;
```

```
SELECT NVL(
  maybe_empty_column,
  "it's empty!"
) FROM my_table;
```

All three examples above are equivalent.

LENGTH

Returns the length of the string in bytes. This function is also available under the `LEN` name .

Examples

```
SELECT LENGTH("foo");
```

```
SELECT LEN("bar");
```

> **Note**
>
> To calculate the length of a string in Unicode characters, you can use the function Unicode::GetLength.
>
> To get the number of elements in the list, use the function ListLength.

SUBSTRING

Returns a substring.

Required arguments:

- Source string;
- Position: The offset from the beginning of the string in bytes (integer) or `NULL` meaning "from the beginning".

Optional arguments:

- Substring length: The number of bytes starting from the specified position (an integer, or the default `NULL` meaning "up to the end of the source string").

Indexing starts from zero. If the specified position and length are beyond the string, returns an empty string.
If the input string is optional, the result is also optional.

Examples

```
SELECT SUBSTRING("abcdefg", 3, 1); -- d
```

```
SELECT SUBSTRING("abcdefg", 3); -- defg
```

```
SELECT SUBSTRING("abcdefg", NULL, 3); -- abc
```

FIND

Finding the position of a substring in a string.

Required arguments:

- Source string;
- The substring being searched for.

Optional arguments:

- A position in bytes to start the search with (an integer or `NULL` by default that means "from the beginning of the source string").

Returns the first substring position found or `NULL` (meaning that the desired substring hasn't been found starting from the specified position).

Examples

```
SELECT FIND("abcdefg_abcdefg", "abc"); -- 0
```

```
SELECT FIND("abcdefg_abcdefg", "abc", 1); -- 8
```

```
SELECT FIND("abcdefg_abcdefg", "abc", 9); -- null
```

RFIND

Reverse finding the position of a substring in a string, from the end to the beginning.

Required arguments:

- Source string;
- The substring being searched for.

Optional arguments:

- A position in bytes to start the search with (an integer or `NULL` by default, meaning "from the end of the source string").

Returns the first substring position found or `NULL` (meaning that the desired substring hasn't been found starting from the specified position).

Examples

```
SELECT RFIND("abcdefg_abcdefg", "bcd"); -- 9
```

```
SELECT RFIND("abcdefg_abcdefg", "bcd", 8); -- 1
```

```
SELECT RFIND("abcdefg_abcdefg", "bcd", 0); -- null
```

StartsWith, EndsWith

Checking for a prefix or suffix in a string.

Required arguments:

- Source string;
- The substring being searched for.

The arguments can be of the `String` or `Utf8` type and can be optional.

Examples

```
SELECT StartsWith("abc_efg", "abc") AND EndsWith("abc_efg", "efg"); -- true
```

```
SELECT StartsWith("abc_efg", "efg") OR EndsWith("abc_efg", "abc"); -- false
```

```
SELECT StartsWith("abcd", NULL); -- null
```

```
SELECT EndsWith(NULL, Utf8("")); -- null
```

IF

Checks the condition: `IF(condition_expression, then_expression, else_expression)`.

It's a simplified alternative for CASE WHEN ... THEN ... ELSE ... END.

You may omit the `else_expression` argument. In this case, if the condition is false (`condition_expression` returned `false`), an empty value is returned with the type corresponding to `then_expression` and allowing for `NULL`. Hence, the result will have an optional data type.

Examples

```
SELECT
  IF(foo > 0, bar, baz) AS bar_or_baz,
  IF(foo > 0, foo) AS only_positive_foo
FROM my_table;
```

NANVL

Replaces the values of `NaN` (not a number) in expressions like `Float`, `Double`, or Optional.

Arguments:

1. The expression where you want to make a replacement.
2. The value to replace `NaN`.

If one of the arguments is `Double`, the result is `Double`, otherwise, it's `Float`. If one of the arguments is `Optional`, then the result is `Optional`.

Examples

```
SELECT
  NANVL(double_column, 0.0)
FROM my_table;
```

Random...

Generates a pseudorandom number:

- `Random()`: A floating point number (Double) from 0 to 1.
- `RandomNumber()`: An integer from the complete Uint64 range.
- `RandomUuid()`: Uuid version 4.

Signatures

```
Random(T1[, T2, ...])->Double
RandomNumber(T1[, T2, ...])->Uint64
RandomUuid(T1[, T2, ...])->Uuid
```

No arguments are used for random number generation: they are only needed to control the time of the call. A new random number is returned at each call. Therefore:

- If Random is called again within a **same query** and with a same set of arguments does not guarantee getting the same sets of random numbers. The values will be equal if the Random calls fall into the same execution phase.
- Calling of Random with the same set of arguments in **different queries** returns different sets of random numbers.

> **Warning**
>
> If Random is used in named expressions, its one-time calculation is not guaranteed. Depending on the optimizers and runtime environment, it can be counted both once and multiple times. To make sure it's only counted once, materialize a named expression into a table.

Use cases:

- `SELECT RANDOM(1);`: Get one random value for the entire query and use it multiple times (to get multiple random values, you can pass various constants of any type).
- `SELECT RANDOM(1) FROM table;`: The same random number for each row in the table.
- `SELECT RANDOM(1), RANDOM(2) FROM table;`: Two random numbers for each row of the table, all the numbers in each of the columns are the same.
- `SELECT RANDOM(some_column) FROM table;`: Different random numbers for each row in the table.
- `SELECT RANDOM(some_column), RANDOM(some_column) FROM table;`: Different random numbers for each row of the table, but two identical numbers within the same row.
- `SELECT RANDOM(some_column), RANDOM(some_column + 1) FROM table;` or `SELECT RANDOM(some_column), RANDOM(other_column) FROM table;`: Two columns, with different numbers in both.

Examples

```
SELECT
    Random(key) -- [0, 1)
FROM my_table;
```

```
SELECT
    RandomNumber(key) -- [0, Max<Uint64>)
FROM my_table;
```

```
SELECT
    RandomUuid(key) -- Uuid version 4
FROM my_table;
```

```
SELECT
    RANDOM(column) AS rand1,
    RANDOM(column) AS rand2, -- same as rand1
    RANDOM(column, 1) AS randAnd1, -- different from rand1/2
    RANDOM(column, 2) AS randAnd2 -- different from randAnd1
FROM my_table;
```

CurrentUtc...

`CurrentUtcDate()`, `CurrentUtcDatetime()` and `CurrentUtcTimestamp()`: Getting the current date and/or time in UTC. The result data type is specified at the end of the function name.

The arguments are optional and work same as RANDOM.

Examples

```
SELECT CurrentUtcDate();
```

```
SELECT CurrentUtcTimestamp(TableRow()) FROM my_table;
```

CurrentTz...

`CurrentTzDate()`, `CurrentTzDatetime()`, and `CurrentTzTimestamp()`: Get the current date and/or time in the IANA time zone specified in the first argument. The result data type is specified at the end of the function name.

The arguments that follow are optional and work same as RANDOM.

Examples

```
SELECT CurrentTzDate("Europe/Moscow");
```

```
SELECT CurrentTzTimestamp("Europe/Moscow", TableRow()) FROM my_table;
```

AddTimezone

Adding the time zone information to the date/time in UTC. In the result of `SELECT` or after `CAST`, a `String` will be subject to the time zone rules used to calculate the time offset.

Arguments:

1. Date: the type is `Date` / `Datetime` / `Timestamp`.
2. The IANA name of the time zone.

Result type: `TzDate` / `TzDatetime` / `TzTimestamp`, depending on the input data type.

Examples

```
SELECT AddTimezone(Datetime("2018-02-01T12:00:00Z"), "Europe/Moscow");
```

RemoveTimezone

Removing the time zone data and converting the value to date/time in UTC.

Arguments:

1. Date: the type is `TzDate` / `TzDatetime` / `TzTimestamp`.

Result type: `Date` / `Datetime` / `Timestamp`, depending on the input data type.

Examples

```
SELECT RemoveTimezone(TzDatetime("2018-02-01T12:00:00,Europe/Moscow"));
```

Version

`Version()` returns a string describing the current version of the node processing the request. In some cases, such as during rolling upgrades, it might return different strings depending on which node processes the request. It does not accept any arguments.

Examples

```
SELECT Version();
```

MAX_OF, MIN_OF, GREATEST, and LEAST

Returns the minimum or maximum among N arguments. Those functions let you replace the SQL standard statement `CASE WHEN a < b THEN a ELSE b END` that would be too sophisticated for N more than two.

The argument types must be mutually castable and accept `NULL`.

`GREATEST` is a synonym for `MAX_OF` and `LEAST` is a synonym for `MIN_OF`.

Examples

```
SELECT MIN_OF(1, 2, 3);
```

AsTuple, AsStruct, AsList, AsDict, AsSet, AsListStrict, AsDictStrict and AsSetStrict

Creates containers of the applicable types. For container literals, operator notation is also supported.

Specifics:

- The container elements are passed in arguments. Hence, the number of elements in the resulting container is equal to the number of arguments passed, except when the dictionary keys repeat.
- `AsTuple` and `AsStruct` can be called without arguments, and also the arguments can have different types.
- The field names in `AsStruct` are set using `AsStruct(field_value AS field_name)`.
- Creating a list requires at least one argument if you need to output the element types. To create an empty list with the given type of elements, use the function ListCreate. You can create an empty list as an `AsList()` call without arguments. In this case, this

expression will have the `EmptyList` type.

- Creating a dictionary requires at least one argument if you need to output the element types. To create an empty dictionary with the given type of elements, use the function DictCreate. You can create an empty dictionary as an `AsDict()` call without arguments, in this case, this expression will have the `EmptyDict` type.
- Creating a set requires at least one argument if you need to output element types. To create an empty set with the given type of elements, use the function SetCreate. You can create an empty set as an `AsSet()` call without arguments, in this case, this expression will have the `EmptySet` type.
- `AsList` outputs the common type of elements in the list. A type error is raised in the case of incompatible types.
- `AsDict` separately outputs the common types for keys and values. A type error is raised in the case of incompatible types.
- `AsSet` outputs common types for keys. A type error is raised in the case of incompatible types.
- `AsListStrict`, `AsDictStrict`, `AsSetStrict` require the same type for their arguments.
- `AsDict` and `AsDictStrict` expect `Tuple` of two elements as arguments (key and value, respectively). If the keys repeat, only the value for the first key remains in the dictionary.
- `AsSet` and `AsSetStrict` expect keys as arguments.

Examples

```
SELECT
  AsTuple(1, 2, "3") AS `tuple`,
  AsStruct(
    1 AS a,
    2 AS b,
    "3" AS c
  ) AS `struct`,
  AsList(1, 2, 3) AS `list`,
  AsDict(
    AsTuple("a", 1),
    AsTuple("b", 2),
    AsTuple("c", 3)
  ) AS `dict`,
  AsSet(1, 2, 3) AS `set`
```

Container literals

Some containers support operator notation for their literal values:

- Tuple: `(value1, value2...)`;
- Structure: `<|name1: value1, name2: value2...|>`;
- List: `[value1, value2,...]`;
- Dictionary: `{key1: value1, key2: value2...}`;
- Set: `{key1, key2...}`.

In every case, you can use an insignificant trailing comma. For a tuple with one element, this comma is required: `(value1,)`.
For field names in the structure literal, you can use an expression that can be calculated at evaluation time, for example, string literals or identifiers (including those enclosed in backticks).

For nested lists, use AsList, for nested dictionaries, use AsDict, for nested sets, use AsSet, for nested tuples, use AsTuple, for nested structures, use AsStruct.

Examples

```
$name = "computed " || "member name";
SELECT
  (1, 2, "3") AS `tuple`,
  <|
    `complex member name`: 2.3,
    b: 2,
    $name: "3",
    "inline " || "computed member name": false
  |> AS `struct`,
  [1, 2, 3] AS `list`,
  {
    "a": 1,
    "b": 2,
    "c": 3,
  } AS `dict`,
  {1, 2, 3} AS `set`
```

Variant

`Variant()` creates a variant value over a tuple or structure.

Arguments:

- Value
- String with a field name or tuple index
- Variant type

Example

```
$var_type = Variant<foo: Int32, bar: Bool>;

SELECT
```

```
    Variant(6, "foo", $var_type) as Variant1Value,
    Variant(false, "bar", $var_type) as Variant2Value;
```

AsVariant

`AsVariant()` creates a value of a variant over a structure including one field. This value can be implicitly converted to any variant over a structure that has a matching data type for this field name and might include more fields with other names.

Arguments:

- Value
- A string with the field name

Example

```
SELECT
   AsVariant(6, "foo") as VariantValue
```

Visit, VisitOrDefault

Processes the possible values of a variant over a structure or tuple using the provided handler functions for each field/element of the variant.

Signature

```
Visit(Variant<key1: K1, key2: K2, ...>, K1->R AS key1, K2->R AS key2, ...)->R
Visit(Variant<K1, K2, ...>, K1->R, K2->R, ...)->R

VisitOrDefault(Variant<K1, K2, ...>{Flags:AutoMap}, R, [K1->R, [K2->R, ...]])->R
VisitOrDefault(Variant<key1: K1, key2: K2, ...>{Flags:AutoMap}, R, [K1->R AS key1, [K2->R AS key2, ...]])->R
```

Arguments

- For a variant over structure: accepts the variant as the positional argument and named arguments (handlers) corresponding to each field of the variant.
- For a variant over tuple: accepts the variant and handlers for each element of the variant as positional arguments.
- `VisitOrDefault` includes an additional positional argument (on the second place) for the default value, enabling the omission of certain handlers.

Example

```
$vartype = Variant<num: Int32, flag: Bool, str: String>;
$handle_num = ($x) -> { return 2 * $x; };
$handle_flag = ($x) -> { return If($x, 200, 10); };
$handle_str = ($x) -> { return Unwrap(CAST(LENGTH($x) AS Int32)); };

$visitor = ($var) -> { return Visit($var, $handle_num AS num, $handle_flag AS flag, $handle_str AS str); };
SELECT
    $visitor(Variant(5, "num", $vartype)),                -- 10
    $visitor(Just(Variant(True, "flag", $vartype))),      -- Just(200)
    $visitor(Just(Variant("somestr", "str", $vartype))),  -- Just(7)
    $visitor(Nothing(OptionalType($vartype))),            -- Nothing(Optional<Int32>)
    $visitor(NULL)                                        -- NULL
;
```

VariantItem

Returns the value of a homogeneous variant (i.e., a variant containing fields/elements of the same type).

Signature

```
VariantItem(Variant<key1: K, key2: K, ...>{Flags:AutoMap})->K
VariantItem(Variant<K, K, ...>{Flags:AutoMap})->K
```

Example

```
$vartype1 = Variant<num1: Int32, num2: Int32, num3: Int32>;
SELECT
    VariantItem(Variant(7, "num2", $vartype1)),          -- 7
    VariantItem(Just(Variant(5, "num1", $vartype1))),    -- Just(5)
    VariantItem(Nothing(OptionalType($vartype1))),       -- Nothing(Optional<Int32>)
    VariantItem(NULL)                                    -- NULL
;
```

Way

Returns an active field (active index) of a variant over a struct (tuple).

Signature

```
Way(Variant<key1: K1, key2: K2, ...>{Flags:AutoMap})->Utf8
Way(Variant<K1, K2, ...>{Flags:AutoMap})->Uint32
```

Example

```
$vr = Variant(1, "0", Variant<Int32, String>);
$vrs = Variant(1, "a", Variant<a:Int32, b:String>);
SELECT Way($vr);  -- 0
SELECT Way($vrs); -- "a"
```

DynamicVariant

Creates a homogeneous variant instance (i.e. containing fields/elements of the same type), where the variant index or field can be set dynamically. If the index or field name does not exist, `NULL` will be returned.
The inverse function is VariantItem.

Signature

```
DynamicVariant(item:T,index:Uint32?,Variant<T, T, ...>)->Optional<Variant<T, T, ...>>
DynamicVariant(item:T,index:Utf8?,Variant<key1: T, key2: T, ...>)->Optional<Variant<key1: T, key2: T, ...>>
```

Example

```
$dt = Int32;
$tvt = Variant<$dt,$dt>;
SELECT ListMap([(10,0u),(20,2u),(30,NULL)],($x)->(DynamicVariant($x.0,$x.1,$tvt))); -- [0: 10,NULL,NULL]

$dt = Int32;
$svt = Variant<x:$dt,y:$dt>;
SELECT ListMap([(10,'x'u),(20,'z'u),(30,NULL)],($x)->(DynamicVariant($x.0,$x.1,$svt))); -- [x: 10,NULL,NULL]
```

Enum

`Enum()` creates an enumeration value.

Arguments:

- A string with the field name
- Enumeration type

Example

```
$enum_type = Enum<Foo, Bar>;
SELECT
   Enum("Foo", $enum_type) as Enum1Value,
   Enum("Bar", $enum_type) as Enum2Value;
```

AsEnum

`AsEnum()` creates a value of enumeration including one element. This value can be implicitly cast to any enumeration containing such a name.

Arguments:

- A string with the name of an enumeration item

Example

```
SELECT
   AsEnum("Foo");
```

AsTagged, Untag

Wraps the value in the Tagged data type with the specified tag, preserving the physical data type. `Untag`: The reverse operation.

Required arguments:

1. Value of any type.
2. Tag name.

Returns a copy of the value from the first argument with the specified tag in the data type.

Examples of use cases:

- Returns to the client's web interface the media files from BASE64-encoded strings.
- Additional refinements at the level of returned columns types.

TableRow, JoinTableRow

Getting the entire table row as a structure. No arguments. `JoinTableRow` in case of `JOIN` always returns a structure with table prefixes.

Example

```
SELECT TableRow() FROM my_table;
```

Ensure...

Checking for the user conditions:

- `Ensure()`: Checking whether the predicate is true at query execution.
- `EnsureType()`: Checking that the expression type exactly matches the specified type.
- `EnsureConvertibleTo()`: A soft check of the expression type (with the same rules as for implicit type conversion).

If the check fails, the entire query fails.

Arguments:

1. An expression that will result from a function call if the check is successful. It's also checked for the data type in the corresponding functions.
2. Ensure uses a Boolean predicate that is checked for being `true`. The other functions use the data type that can be obtained using the relevant functions, or a string literal with a text description of the type.
3. An optional string with an error comment to be included in the overall error message when the query is complete. The data itself can't be used for type checks, since the data check is performed at query validation (or can be an arbitrary expression in the case of Ensure).

Examples

```
SELECT Ensure(
    value,
    value < 100,
    "value out or range"
) AS value FROM my_table;
```

```
SELECT EnsureType(
    value,
    TypeOf(other_value),
    "expected value and other_value to be of same type"
) AS value FROM my_table;
```

```
SELECT EnsureConvertibleTo(
    value,
    Double?,
    "expected value to be numeric"
) AS value FROM my_table;
```

EvaluateExpr, EvaluateAtom

Evaluate an expression before the start of the main calculation and input its result to the query as a literal (constant). In many contexts, where only a constant would be expected in standard SQL (for example, in table names, in the number of rows in LIMIT, and so on), this functionality is implicitly enabled automatically.

EvaluateExpr can be used where the grammar already expects an expression. For example, you can use it to:

- Round the current time to days, weeks, or months and insert it into the query to ensure correct query caching, although usually when functions are used to get the current time, query caching is completely disabled.
- Run a heavy calculation with a small result once per query instead of once per job.

The only argument for both functions is the expression for calculation and substitution.

Restrictions:

- The expression must not trigger MapReduce operations.
- This functionality is fully locked in YQL over YDB.

Examples

```
$now = CurrentUtcDate();
SELECT EvaluateExpr(
    DateTime::MakeDate(DateTime::StartOfWeek($now)
    )
);
```

Literals of primitive types

For primitive types, you can create literals based on string literals.

Syntax

`<Primitive type>( <string>[, <additional attributes>] )`

Unlike `CAST("myString" AS MyType)`:

- The check for literal's castability to the desired type occurs at validation.
- The result is non-optional.

The `<string>` argument must be a string literal. Expressions, including variables, columns, query parameters, and function results, are not supported. To convert a value at query execution time, use `CAST(expression AS <Primitive type>)`. For example, convert a computed `String` to `Uuid` using `CAST(expression AS Uuid)`, not `Uuid(expression)`.

```
$uuid = Uuid("550e8400-e29b-41d4-a716-446655440000");
$uuidAsString = CAST($uuid AS String);
```

```
SELECT CAST($uuidAsString AS Uuid) = $uuid; -- true
```

For the data types `Date`, `Datetime`, `Timestamp`, and `Interval`, literals are supported only in the format corresponding to ISO 8601. `Interval` has the following differences from the standard:

- It supports the negative sign for shifts to the past.
- Microseconds can be expressed as fractional parts of seconds.
- You can't use units of measurement exceeding one week.
- The options with the beginning/end of the interval and with repetitions, are not supported.

For the data types `TzDate`, `TzDatetime`, `TzTimestamp`, literals are also set in the format meeting ISO 8601, but instead of the optional Z suffix, they specify the IANA name of the time zone, separated by comma (for example, GMT or Europe/Moscow).

For the Decimal parametric data type, two additional arguments are specified:

- Total number of decimal places (up to 35, inclusive).
- Number of places after the decimal point (out of the total number, meaning it can't be larger than the previous argument).

Examples

```
SELECT
  Bool("true"),
  Uint8("0"),
  Int32("-1"),
  Uint32("2"),
  Int64("-3"),
  Uint64("4"),
  Float("-5"),
  Double("6"),
  Decimal("1.23", 5, 2), -- up to 5 decimal places, of which 2 are after the decimal point
  DyNumber("1"),
  String("foo"),
  Utf8("hello"),
  Yson("<a=1>[3;%false]"),
  Json(@@{"a":1,"b":null}@@),
  Date("2017-11-27"),
  Datetime("2017-11-27T13:24:00Z"),
  Timestamp("2017-11-27T13:24:00.123456Z"),
  Interval("P1W2DT2H3M4.567890S"),
  TzDate("2017-11-27,Europe/Moscow"),
  TzDatetime("2017-11-27T13:24:00,America/Los_Angeles"),
  TzTimestamp("2017-11-27T13:24:00.123456,GMT"),
  Uuid("f9d5cc3f-f1dc-4d9c-b97e-766e57ca4ccb");
```

ToBytes and FromBytes

Conversion of primitive data types to a string with their binary representation and back. Numbers are represented in the little endian format.

Examples

```
SELECT
    ToBytes(123), -- "{\u0000\u0000\u0000"
    String::HexEncode(ToBytes(123)) -- "7B000000"
    FromBytes(
        "\xd2\x02\x96\x49\x00\x00\x00\x00",
        Uint64
    ); -- 1234567890ul
```

ByteAt

Getting the byte value inside a string at an index counted from the beginning of the string. If an invalid index is specified, `NULL` is returned.

Arguments:

1. String: `String` or `Utf8`.
2. Index: `Uint32`.

Examples

```
SELECT
    ByteAt("foo", 0), -- 102
    ByteAt("foo", 1), -- 111
    ByteAt("foo", 9); -- NULL
```

...Bit

`TestBit()`, `ClearBit()`, `SetBit()` and `FlipBit()`: Test, clear, set, or flip a bit in an unsigned number using the specified bit sequence number.

Arguments:

1. An unsigned number that's subject to the operation. TestBit is also implemented for strings.
2. Number of the bit.

TestBit returns `true/false`. The other functions return a copy of their first argument with the corresponding conversion.

Examples

```
SELECT
    TestBit(1u, 0), -- true
    SetBit(8u, 0); -- 9
```

Abs

The absolute value of the number.

Examples

```
SELECT Abs(-123); -- 123
```

Just

`Just()`: Change the value's data type to optional from the current data type (i.e., `T` is converted to `T?`).

The reverse operation is Unwrap.

Examples

```
SELECT
  Just("my_string"); --  String?
```

Unwrap

`Unwrap()`: Converting the optional value of the data type to the relevant non-optional type, raising a runtime error if the data is `NULL`. This means that `T?` becomes `T`.

If the value isn't optional, then the function returns its first argument unchanged.

Arguments:

1. Value to be converted.
2. An optional string with a comment for the error text.

Reverse operation is Just.

Examples

```
$value = Just("value");

SELECT Unwrap($value, "Unexpected NULL for $value");
```

Nothing

`Nothing()`: Create an empty value for the specified Optional data type.

Examples

```
SELECT
  Nothing(String?); -- an empty (NULL) value with the String? type
```

Learn more about ParseType and other functions for data types.

Callable

Create a callable value with the specified signature from a lambda function. It's usually used to put callable values into containers.

Arguments:

1. Type.
2. Lambda function.

Examples

```
$lambda = ($x) -> {
    RETURN CAST($x as String)
};

$callables = AsTuple(
    Callable(Callable<(Int32)->String>, $lambda),
    Callable(Callable<(Bool)->String>, $lambda),
);

SELECT $callables.0(10), $callables.1(true);
```

Pickle, Unpickle

`Pickle()` and `StablePickle()` serialize an arbitrary object into a sequence of bytes, if possible. Typical non-serializable objects are Callable and Resource. The serialization format is not versioned and can be used within a single query. For the Dict type, the StablePickle function pre-sorts the keys, and for Pickle, the order of dictionary elements in the serialized representation isn't defined.

`Unpickle()` is the inverse operation (deserialization), where with the first argument being the data type of the result and the second argument is the string with the result of `Pickle()` or `StablePickle()`.

Examples

```
SELECT *
FROM my_table
WHERE Digest::MurMurHash32(
        Pickle(TableRow())
    ) % 10 == 0;

$buf = Pickle(123);
SELECT Unpickle(Int32, $buf);
```

StaticMap

Transforms a structure or tuple by applying a lambda function to each item.

Arguments:

- Structure or tuple.
- Lambda for processing items.

Result: a structure or tuple with the same number and naming of items as in the first argument, and with item data types determined by lambda results.

Examples

```
SELECT *
FROM (
    SELECT
        StaticMap(TableRow(), ($item) -> {
            return CAST($item AS String);
        })
    FROM my_table
) FLATTEN COLUMNS; -- converting all columns to rows
```

StaticZip

Merges structures or tuples element-by-element. All arguments (one or more) must be either structures with the same set of fields or tuples of the same length.
The result will be a structure or tuple, respectively.
Each item of the result is a tuple comprised of items taken from arguments.

Examples

```
$one = <|k1:1, k2:2.0|>;
$two = <|k1:3.0, k2:4|>;

-- Adding two structures item-by-item
SELECT StaticMap(StaticZip($one, $two), ($tuple)->($tuple.0 + $tuple.1)) AS sum;
```

StaticFold, StaticFold1

```
StaticFold(obj:Struct/Tuple, initVal, updateLambda)
StaticFold1(obj:Struct/Tuple, initLambda, updateLambda)
```

Left fold over struct/tuple elements.
The folding of tuples is done in order from the element with the lower index to the element with the larger one; for structures, the order is not guaranteed.

- `obj` - object to fold
- `initVal` - *(for StaticFold)* initial fold state
- `initLambda` - *(for StaticFold1)* lambda that produces initial fold state from the first element
- `updateLambda` - lambda that produces the new state (arguments are the next element and the previous state)

`StaticFold(<|key_1:$el_1, key_2:$el_2, ..., key_n:$el_n|>, $init, $f)` transforms into:

```
$f($el_n, ...$f($el_2, $f($init, el_1))...)
```

`StaticFold1(<|key_1:$el_1, key_2:$el_2, ..., key_n:$el_n|>, $f0, $f)`:

```
$f($el_n, ...$f($el_2, $f($f0($init), el_1))...)
```

`StaticFold1(<||>, $f0, $f)` returns `NULL`.

Works with tuples in the same way.

AggregationFactory

Create a factory for aggregation functions to separately describe the methods of aggregation and data types subject to aggregation.

Arguments:

1. A string in double quotes with the name of an aggregate function, for example "MIN".
2. Optional parameters of the aggregate function that are data-independent. For example, the percentile value in PERCENTILE.

The resulting factory can be used as the second parameter of the function AGGREGATE_BY.
If the aggregate function is applied to two columns instead of one, as, for example, MIN_BY, then in AGGREGATE_BY, the first argument passes a `Tuple` of two values. See more details in the description of the applicable aggregate function.

Examples

```
$factory = AggregationFactory("MIN");
SELECT
    AGGREGATE_BY (value, $factory) AS min_value -- apply the MIN aggregation to the "value" column
FROM my_table;
```

AggregateTransformInput

`AggregateTransformInput()` converts an aggregation factory, for example, obtained using the AggregationFactory function, to other factory, in which the specified transformation of input items is performed before starting aggregation.

Arguments:

1. Aggregation factory.
2. A lambda function with one argument that converts an input item.

Examples

```
$f = AggregationFactory("sum");
$g = AggregateTransformInput($f, ($x) -> (cast($x as Int32)));
$h = AggregateTransformInput($f, ($x) -> ($x * 2));
SELECT ListAggregate([1,2,3], $f); -- 6
SELECT ListAggregate(["1","2","3"], $g); -- 6
SELECT ListAggregate([1,2,3], $h); -- 12
```

AggregateTransformOutput

`AggregateTransformOutput()` converts an aggregation factory, for example, obtained using the AggregationFactory function, to other factory, in which the specified transformation of the result is performed after ending aggregation.

Arguments:

1. Aggregation factory.
2. A lambda function with one argument that converts the result.

Examples

```
$f = AggregationFactory("sum");
$g = AggregateTransformOutput($f, ($x) -> ($x * 2));
SELECT ListAggregate([1,2,3], $f); -- 6
SELECT ListAggregate([1,2,3], $g); -- 12
```

AggregateFlatten

Adapts a factory for aggregation functions, for example, obtained using the AggregationFactory function in a way that allows aggregation of list input items. This operation is similar to FLATTEN LIST BY: Each list item is aggregated.

Arguments:

1. Aggregation factory.

Examples

```
$i = AggregationFactory("AGGREGATE_LIST_DISTINCT");
$j = AggregateFlatten($i);
SELECT AggregateBy(x, $j) from (
   select [1,2] as x
   union all
   select [2,3] as x
); -- [1, 2, 3]
```

Fulltext search

The `FulltextMatch` and `FulltextScore` functions are intended for fulltext search in YDB **via a fulltext index**.

> **Note**
>
> These functions use a fulltext index only when the query is executed with `VIEW IndexName`. See VIEW (Fulltext index) and Fulltext Indexes.

FulltextMatch

`FulltextMatch(text, query)` filters rows by whether the text matches a fulltext query.

```
SELECT id, body
FROM articles VIEW ft_idx
WHERE FulltextMatch(body, "machine learning");
```

Only the first two arguments can be positional. Additional parameters must be passed as **named arguments**.

Supports named arguments:

- `Mode` (String): query mode:
  - `Keywords` (default) — the query is split into individual terms; how they are combined is determined by `DefaultOperator`
  - `Query` — extended syntax with logical operators: required terms via `+`, excluded terms via `-`, exact phrases in double quotes
  - `Wildcard` — wildcard search: `%` matches any substring, `_` matches a single character (similar to `LIKE`); requires an n-gram index
- `DefaultOperator` (String): term combination operator in `Keywords` mode:
  - `And` (default) — all query terms must be present in the text
  - `Or` — terms split into **required** (prefixed with `+`, must appear in every result) and **optional** (no prefix, count toward `MinimumShouldMatch`). Without any `+` prefix, all terms are optional and at least one must match
- `MinimumShouldMatch` (String): minimum number of **optional** terms that must match when `DefaultOperator = "Or"` — specified as an absolute number (for example, `"3"`) or a percentage of the optional terms (for example, `"50%"`). Required (`+`) terms are not counted

`Mode` and `DefaultOperator` values are case-insensitive.

`Wildcard` example

If the index is created with n-gram filtering, you can use `%` and `_` patterns similar to `LIKE`.

```
SELECT id, title
FROM articles VIEW ft_idx
WHERE FulltextMatch(body, "mach% learn%", "Wildcard" AS Mode)
LIMIT 20;
```

MinimumShouldMatch example

An 8-term query where at least half of the terms must match:

```
SELECT id
FROM articles VIEW ft_idx
WHERE FulltextMatch(
  body,
  "machine learning neural networks deep learning large scale data recommendations search",
  "Keywords" AS Mode,
  "Or" AS DefaultOperator,
  "50%" AS MinimumShouldMatch
);
```

In this example, `Mode` is set to `Keywords` (treating the query as a set of terms), `DefaultOperator` is set to `Or` (allowing matching any subset of terms), and `MinimumShouldMatch` is set to `"50%"` (requiring at least 4 out of 8 terms to match). This helps filter out documents that contain only one or two words from a long query.

Required term (`+`) example

Prefix a term with `+` in `Keywords` mode with `DefaultOperator = "Or"` to require it in every result. This follows Lucene's required term syntax.

```
SELECT id
FROM articles VIEW ft_idx
WHERE FulltextMatch(
  body,
  "+machine learning neural networks",
  "Or" AS DefaultOperator,
  "1" AS MinimumShouldMatch
);
```

This returns documents that contain "machine" (required) and at least 1 of "learning", "neural", "networks" (optional). `MinimumShouldMatch` counts only the optional terms. If all terms are prefixed with `+`, the query is a strict AND.

FulltextScore

`FulltextScore(text, query)` returns a relevance score based on the BM25 algorithm and can be used for ranking results.

Requires the `fulltext_relevance` index type.

```
SELECT id, FulltextScore(body, "machine learning") AS relevance
FROM articles VIEW ft_idx
WHERE FulltextScore(body, "machine learning") > 0
ORDER BY relevance DESC
LIMIT 10;
```

Only the first two arguments can be positional. Additional parameters must be passed as **named arguments**.

Supports named arguments:

- `DefaultOperator` (String): term combination operator — `And` (default, all terms must be present) or `Or` (terms split into **required** with `+` prefix and **optional** without; at least `MinimumShouldMatch` optional terms must match, and every required term must be present)
- `MinimumShouldMatch` (String): when `DefaultOperator = "Or"`, minimum number of **optional** terms that must match — specified as an absolute number (for example, `"2"`) or a percentage of the optional terms (for example, `"50%"`). Required (`+`) terms are not counted
- `K1` (Double): term frequency saturation parameter in BM25 — controls how strongly repeated occurrences of a term affect the score; typical range: 1.2–2.0
- `B` (Double): document length normalization parameter in BM25 — `0.0` disables normalization, `1.0` fully normalizes by document length; typical value: 0.75

Example:

```
SELECT id,
       FulltextScore(body, "machine learning", "Or" AS DefaultOperator, "1" AS MinimumShouldMatch, 0.75 AS K
1, 1.2 AS B) AS relevance
FROM articles VIEW ft_idx
WHERE FulltextScore(body, "machine learning", "Or" AS DefaultOperator, "1" AS MinimumShouldMatch, 0.75 AS K1,
1.2 AS B) > 0
ORDER BY relevance DESC;
```

> **Note**
>
> `MinimumShouldMatch` is supported only when `DefaultOperator` is set to `Or`. When `DefaultOperator` is set to `And`, use `FulltextScore` without `MinimumShouldMatch`.

Aggregate functions

COUNT

Counting the number of rows in the table (if `*` or constant is specified as the argument) or non-empty values in a table column (if the column name is specified as an argument).

Like other aggregate functions, it can be combined with GROUP BY to get statistics on the parts of the table that correspond to the values in the columns being grouped. Use the modifier DISTINCT to count distinct values.

Examples

```
SELECT COUNT(*) FROM my_table;
```

```
SELECT key, COUNT(value) FROM my_table GROUP BY key;
```

```
SELECT COUNT(DISTINCT value) FROM my_table;
```

MIN and MAX

Minimum or maximum value.

As an argument, you may use an arbitrary computable expression with a numeric result.

Examples

```
SELECT MIN(value), MAX(value) FROM my_table;
```

SUM

Sum of the numbers.

As an argument, you may use an arbitrary computable expression with a numeric result.

Integers are automatically expanded to 64 bits to reduce the risk of overflow.

```
SELECT SUM(value) FROM my_table;
```

AVG

Arithmetic average.

As an argument, you may use an arbitrary computable expression with a numeric result.

Integer values and time intervals are automatically converted to Double.

Examples

```
SELECT AVG(value) FROM my_table;
```

COUNT_IF

Number of rows for which the expression specified as the argument is true (the expression's calculation result is true).

The value `NULL` is equated to `false` (if the argument type is `Bool?`).

The function *does not* do the implicit type casting to Boolean for strings and numbers.

Examples

```
SELECT
  COUNT_IF(value % 2 == 1) AS odd_count
FROM my_table;
```

Note

To count distinct values in rows meeting the condition, unlike other aggregate functions, you can't use the modifier DISTINCT because arguments contain no values. To get such a result, use a query like this:

```
SELECT
    COUNT(DISTINCT IF(value % 2 == 1, value))
FROM my_table;
```

SUM_IF and AVG_IF

Sum or arithmetic average, but only for the rows that satisfy the condition passed by the second argument.

Therefore, `SUM_IF(value, condition)` is a slightly shorter notation for `SUM(IF(condition, value))`, same for `AVG`. The argument's data type expansion is similar to the same-name functions without a suffix.

Examples

```
SELECT
    SUM_IF(value, value % 2 == 1) AS odd_sum,
    AVG_IF(value, value % 2 == 1) AS odd_avg,
FROM my_table;
```

When you use aggregation factories, a `Tuple` containing a value and a predicate is passed as the first AGGREGATE_BY argument.

```
$sum_if_factory = AggregationFactory("SUM_IF");
$avg_if_factory = AggregationFactory("AVG_IF");

SELECT
    AGGREGATE_BY(AsTuple(value, value % 2 == 1), $sum_if_factory) AS odd_sum,
    AGGREGATE_BY(AsTuple(value, value % 2 == 1), $avg_if_factory) AS odd_avg
FROM my_table;
```

SOME

Get the value for an expression specified as an argument, for one of the table rows. Gives no guarantee of which row is used. It's similar to the any() function in ClickHouse.

Because of no guarantee, `SOME` is computationally cheaper than MIN / MAX often used in similar situations.

Examples

```
SELECT
  SOME(value)
FROM my_table;
```

Alert

When the aggregate function `SOME` is called multiple times, it's **not** guaranteed that all the resulting values are taken from the same row of the source table. To get this guarantee, pack the values into any container and pass it to `SOME`. For example, in the case of a structure, you can apply AsStruct

CountDistinctEstimate, HyperLogLog, and HLL

Approximating the number of unique values using the HyperLogLog algorithm. Logically, it does the same thing as COUNT(DISTINCT ...), but runs much faster at the cost of some error.

Arguments:

1. Estimated value
2. Accuracy (4 to 18 inclusive, 14 by default).

By selecting accuracy, you can trade added resource and RAM consumption for decreased error.

All the three functions are aliases at the moment, but `CountDistinctEstimate` may start using a different algorithm in the future.

Examples

```
SELECT
  CountDistinctEstimate(my_column)
FROM my_table;
```

```
SELECT
  HyperLogLog(my_column, 4)
FROM my_table;
```

AGGREGATE_LIST

Get all column values as a list. When combined with `DISTINCT,` it returns only distinct values. The optional second parameter sets the maximum number of values to be returned. A zero limit value means unlimited.

If you know already that you have few distinct values, use the `AGGREGATE_LIST_DISTINCT` aggregate function to build the same result in memory (that might not be enough for a large number of distinct values).

The order of elements in the result list depends on the implementation and can't be set externally. To return an ordered list, sort the result, for example, with ListSort.

To return a list of multiple values from one line, **DO NOT** use the `AGGREGATE_LIST` function several times, but add all the needed values to a container, for example, via AsList or AsTuple, then pass this container to a single `AGGREGATE_LIST` call.

For example, you can combine it with `DISTINCT` and the function String::JoinFromList (it's an equivalent of `','.join(list)` in Python) to output to a string all the values found in the column after GROUP BY.

Examples

```
SELECT
   AGGREGATE_LIST( region ),
   AGGREGATE_LIST( region, 5 ),
   AGGREGATE_LIST( DISTINCT region ),
   AGGREGATE_LIST_DISTINCT( region ),
```

```
   AGGREGATE_LIST_DISTINCT( region, 5 )
FROM users
```

```
-- An equivalent of GROUP_CONCAT in MySQL
SELECT
    String::JoinFromList(CAST(AGGREGATE_LIST(region, 2) AS List<String>), ",")
FROM users
```

These functions also have a short notation: `AGG_LIST` and `AGG_LIST_DISTINCT`.

Alert

Execution is **NOT** lazy, so when you use it, be sure that the list has a reasonable size (about a thousand items or less). To stay on the safe side, better use a second optional numeric argument that limits the number of items in the list.

MAX_BY and MIN_BY

Return the value of the first argument for the table row where the second argument is minimum/maximum.

You can optionally specify the third argument N that affects behavior if the table has multiple rows with the same minimum or maximum value:

- If N is omitted, the value of one of the rows is returned, and the other rows are discarded.
- If N is specified, the list is returned with all values, but their number can't exceed N. All values after the number are discarded.

When choosing N, we recommend that you don't exceed several hundreds or thousands to avoid issues with the limited memory available on YDB clusters.

If your task needs absolutely all values, and their number is measured in dozens of thousands or more, then instead of those aggregate functions better use `JOIN` on the source table with a subquery doing `GROUP BY + MIN/MAX` on the desired columns of this table.

Attention

If the second argument is always `NULL`, the aggregation result is `NULL`.

When you use aggregation factories, a `Tuple` containing a value and a key is passed as the first AGGREGATE_BY argument.

Examples

```
SELECT
  MIN_BY(value, LENGTH(value)),
  MAX_BY(value, key, 100)
FROM my_table;
```

```
$min_by_factory = AggregationFactory("MIN_BY");
$max_by_factory = AggregationFactory("MAX_BY", 100);

SELECT
    AGGREGATE_BY(AsTuple(value, LENGTH(value)), $min_by_factory),
    AGGREGATE_BY(AsTuple(value, key), $max_by_factory)
FROM my_table;
```

TOP and BOTTOM

Return a list of the maximum/minimum values of an expression. The first argument is an expression, the second argument limits the number of items.

Examples

```
SELECT
    TOP(key, 3),
    BOTTOM(value, 3)
FROM my_table;
```

```
$top_factory = AggregationFactory("TOP", 3);
$bottom_factory = AggregationFactory("BOTTOM", 3);

SELECT
    AGGREGATE_BY(key, $top_factory),
    AGGREGATE_BY(value, $bottom_factory)
FROM my_table;
```

TOP_BY and BOTTOM_BY

Return a list of values of the first argument for the rows containing the maximum/minimum values of the second argument. The third argument limits the number of items in the list.

When you use aggregation factories, a `Tuple` containing a value and a key is passed as the first AGGREGATE_BY argument. In this case, the limit for the number of items is passed by the second argument at factory creation.

Examples

```
SELECT
    TOP_BY(value, LENGTH(value), 3),
    BOTTOM_BY(value, key, 3)
FROM my_table;
```

```
$top_by_factory = AggregationFactory("TOP_BY", 3);
$bottom_by_factory = AggregationFactory("BOTTOM_BY", 3);

SELECT
    AGGREGATE_BY(AsTuple(value, LENGTH(value)), $top_by_factory),
    AGGREGATE_BY(AsTuple(value, key), $bottom_by_factory)
FROM my_table;
```

TOPFREQ and MODE

Getting an **approximate** list of the most common values in a column with an estimation of their count. Returns a list of structures with two fields:

- `Value`: the frequently occurring value that was found.
- `Frequency`: An estimated value occurrence in the table.

Required argument: the value itself.

Optional arguments:

1. For `TOPFREQ`, the desired number of items in the result. `MODE` is an alias to `TOPFREQ` with this argument set to 1. For `TOPFREQ`, this argument is also 1 by default.
2. The number of items in the buffer used: lets you trade memory consumption for accuracy. Default: 100.

Examples

```
SELECT
    MODE(my_column),
    TOPFREQ(my_column, 5, 1000)
FROM my_table;
```

STDDEV and VARIANCE

Standard deviation and variance in a column. Those functions use a single-pass parallel algorithm, whose result may differ from the more common methods requiring two passes through the data.

By default, the sample variance and standard deviation are calculated. Several write methods are available:

- with the `POPULATION` suffix/prefix, for example: `VARIANCE_POPULATION`, `POPULATION_VARIANCE` calculates the variance or standard deviation for the population.
- With the `SAMPLE` suffix/prefix or without a suffix, for example, `VARIANCE_SAMPLE`, `SAMPLE_VARIANCE`, `SAMPLE` calculate sample variance and standard deviation.

Several abbreviated aliases are also defined, for example, `VARPOP` or `STDDEVSAMP`.

If all the values passed are `NULL`, it returns `NULL`.

Examples

```
SELECT
  STDDEV(numeric_column),
  VARIANCE(numeric_column)
FROM my_table;
```

CORRELATION and COVARIANCE

Correlation and covariance between two columns.

Abbreviated versions are also available: `CORR` or `COVAR`. For covariance, there are also versions with the `SAMPLE` / `POPULATION` suffix that are similar to VARIANCE above.

Unlike most other aggregate functions, they don't skip `NULL`, but accept it as 0.

When you use aggregation factories, a `Tuple` containing two values is passed as the first AGGREGATE_BY argument.

Examples

```
SELECT
  CORRELATION(numeric_column, another_numeric_column),
  COVARIANCE(numeric_column, another_numeric_column)
FROM my_table;
```

```
$corr_factory = AggregationFactory("CORRELATION");

SELECT
    AGGREGATE_BY(AsTuple(numeric_column, another_numeric_column), $corr_factory)
FROM my_table;
```

PERCENTILE and MEDIAN

Calculating percentiles using the amortized version of the TDigest algorithm. `MEDIAN`: An alias for `PERCENTILE(N, 0.5)`.

> **Restriction**
>
> The first argument (N) must be a table column name. If you need to bypass this restriction, use a subquery. The restriction is introduced to simplify calculations, since the implementation merges the calls with the same first argument (N) into a single pass.

```
SELECT
    MEDIAN(numeric_column),
    PERCENTILE(numeric_column, 0.99)
FROM my_table;
```

HISTOGRAM

Plotting an approximate histogram based on a numeric expression with automatic selection of buckets.

Auxiliary functions

Basic settings

You can limit the number of buckets using an optional argument. The default value is 100. Keep in mind that added accuracy costs you more computing resources and may negatively affect the query execution time. In extreme cases, it may affect your query success.

Support for weights

You can specify a "weight" for each value used in the histogram. To do this, pass to the aggregate function the second argument with an expression for calculating the weight. The weight of `1.0` is always used by default. If you use non-standard weights, you may also use the third argument to limit the number of buckets.

If you pass two arguments, the meaning of the second argument is determined by its type (if it's an integer literal, it limits the number of buckets, otherwise it's used as a weight).

If you need an accurate histogram

1. You can use the aggregate functions described below with fixed bucket grids: LinearHistogram or LogarithmicHistogram.
2. You can calculate the bucket number for each row and apply to it GROUP BY.

When you use aggregation factories, a `Tuple` containing a value and a weight is passed as the first AGGREGATE_BY argument.

Examples

```
SELECT
    HISTOGRAM(numeric_column)
FROM my_table;
```

```
SELECT
    Histogram::Print(
        HISTOGRAM(numeric_column, 10),
        50
    )
FROM my_table;
```

```
$hist_factory = AggregationFactory("HISTOGRAM");

SELECT
    AGGREGATE_BY(AsTuple(numeric_column, 1.0), $hist_factory)
FROM my_table;
```

LinearHistogram, LogarithmicHistogram, and LogHistogram

Plotting a histogram based on an explicitly specified fixed bucket scale.

Arguments:

1. Expression used to plot the histogram. All the following arguments are optional.
2. Spacing between the `LinearHistogram` buckets or the logarithm base for `LogarithmicHistogram` / `LogHistogram` (those are aliases). In both cases, the default value is 10.
3. Minimum value. By default, it's minus infinity.
4. Maximum value. By default, it's plus infinity.

The format of the result is totally similar to adaptive histograms, so you can use the same set of auxiliary functions.

If the spread of input values is uncontrollably large, we recommend that you specify the minimum and maximum values to prevent potential failures due to high memory consumption.

Examples

```
SELECT
    LogarithmicHistogram(numeric_column, 2)
FROM my_table;
```

BOOL_AND, BOOL_OR and BOOL_XOR

Signature

```
BOOL_AND(Bool?)->Bool?
BOOL_OR(Bool?)->Bool?
BOOL_XOR(Bool?)->Bool?
```

Apply the relevant logical operation (`AND` / `OR` / `XOR`) to all values in a Boolean column or expression.

Unlike most other aggregate functions, these functions **don't skip** `NULL` during aggregation and use the following rules:

- `true AND null == null`
- `false OR null == null`

For `BOOL_AND` :

- If at least one `NULL` value is present, the result is `NULL` regardless of `true` values in the expression.
- If at least one `false` value is present, the result changes to `false` regardless of `NULL` values in the expression.

For `BOOL_OR` :

- If at least one `NULL` value is present, the result changes to `NULL` regardless of `false` values in the expression.
- If at least one `true` value is present, the result changes to `true` regardless of `NULL` values in the expression.

For `BOOL_XOR` :

- The result is `NULL` if any `NULL` is found.

Examples of such behavior can be found below.

To skip `NULL` values during aggregation, use the `MIN` / `MAX` or `BIT_AND` / `BIT_OR` / `BIT_XOR` functions.

Examples

```
$data = [
    <|nonNull: true, nonFalse: true, nonTrue: NULL, anyVal: true|>,
    <|nonNull: false, nonFalse: NULL, nonTrue: NULL, anyVal: NULL|>,
    <|nonNull: false, nonFalse: NULL, nonTrue: false, anyVal: false|>,
];

SELECT
    BOOL_AND(nonNull) as nonNullAnd,      -- false
    BOOL_AND(nonFalse) as nonFalseAnd,    -- NULL
    BOOL_AND(nonTrue) as nonTrueAnd,      -- false
    BOOL_AND(anyVal) as anyAnd,           -- false
    BOOL_OR(nonNull) as nonNullOr,        -- true
    BOOL_OR(nonFalse) as nonFalseOr,      -- true
    BOOL_OR(nonTrue) as nonTrueOr,        -- NULL
    BOOL_OR(anyVal) as anyOr,             -- true
    BOOL_XOR(nonNull) as nonNullXor,      -- true
    BOOL_XOR(nonFalse) as nonFalseXor,    -- NULL
    BOOL_XOR(nonTrue) as nonTrueXor,      -- NULL
    BOOL_XOR(anyVal) as anyXor,           -- NULL
FROM AS_TABLE($data);
```

BIT_AND, BIT_OR and BIT_XOR

Apply the relevant bitwise operation to all values of a numeric column or expression.

Examples

```
SELECT
    BIT_XOR(unsigned_numeric_value)
FROM my_table;
```

SessionStart

No arguments. It's allowed only if there is SessionWindow in GROUP BY / PARTITION BY.
Returns the value of the `SessionWindow` key column. If `SessionWindow` has two arguments, it returns the minimum value of the first argument within the group/section.
In the case of the expanded version `SessionWindow` , it returns the value of the second element from the tuple returned by `<calculate_lambda>` , for which the first tuple element is `True` .

AGGREGATE_BY and MULTI_AGGREGATE_BY

Applying an aggregation factory to all values of a column or expression. The `MULTI_AGGREGATE_BY` function requires that the value of a column or expression has a structure, tuple, or list, and applies the factory to each individual element, placing the result in a container of the same format. If different values of a column or expression contain lists of different length, the resulting list will have the smallest of the source lengths.

1. Column, `DISTINCT` column or expression.
2. Factory.

Examples

```
$count_factory = AggregationFactory("COUNT");
```

```
SELECT
    AGGREGATE_BY(DISTINCT column, $count_factory) as uniq_count
FROM my_table;

SELECT
    MULTI_AGGREGATE_BY(nums, AggregationFactory("count")) as count,
    MULTI_AGGREGATE_BY(nums, AggregationFactory("min")) as min,
    MULTI_AGGREGATE_BY(nums, AggregationFactory("max")) as max,
    MULTI_AGGREGATE_BY(nums, AggregationFactory("avg")) as avg,
    MULTI_AGGREGATE_BY(nums, AggregationFactory("percentile", 0.9)) as p90
FROM my_table;
```

List of window functions in YQL

The syntax for calling window functions is detailed in a separate article.

Aggregate functions

All aggregate functions can also be used as window functions.
In this case, each row includes an aggregation result obtained on a set of rows from the window frame.

Examples

```
SELECT
    SUM(int_column) OVER w1 AS running_total,
    SUM(int_column) OVER w2 AS total,
FROM my_table
WINDOW
    w1 AS (ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW),
    w2 AS ();
```

ROW_NUMBER

Row number within a partition. No arguments.

Signature

```
ROW_NUMBER()->Uint64
```

Examples

```
SELECT
    ROW_NUMBER() OVER w AS row_num
FROM my_table
WINDOW w AS (ORDER BY key);
```

LAG / LEAD

Accessing a value from a row in the section that lags behind (`LAG`) or leads (`LEAD`) the current row by a fixed number. The first argument specifies the expression to be accessed, and the second argument specifies the offset in rows. You may omit the offset. By default, the neighbor row is used: the previous or next, respectively (hence, 1 is assumed by default). For the rows having no neighbors at a given distance (for example, `LAG(expr, 3)` `NULL` is returned in the first and second rows of the section).

Signature

```
LEAD(T[,Int32])->T?
LAG(T[,Int32])->T?
```

Examples

```
SELECT
   int_value - LAG(int_value) OVER w AS int_value_diff
FROM my_table
WINDOW w AS (ORDER BY key);
```

```
SELECT item, odd, LAG(item, 1) OVER w as lag1 FROM (
    SELECT item, item % 2 as odd FROM (
        SELECT AsList(1, 2, 3, 4, 5, 6, 7) as item
    )
    FLATTEN BY item
)
WINDOW w As (
    PARTITION BY odd
    ORDER BY item
);

/* Output:
item  odd  lag1
--------------------
2  0  NULL
4  0  2
6  0  4
1  1  NULL
3  1  1
5  1  3
7  1  5
*/
```

FIRST_VALUE / LAST_VALUE

Access values from the first and last rows (using the `ORDER BY` clause for the window) of the window frame. The only argument is the expression that you need to access.

Optionally, `OVER` can be preceded by the additional modifier `IGNORE NULLS`. It changes the behavior of functions to the first or last **non-empty** (i.e., non-`NULL`) value among the window frame rows. The antonym of this modifier is `RESPECT NULLS`: it's the default behavior that can be omitted.

Signature

```
FIRST_VALUE(T)->T?
LAST_VALUE(T)->T?
```

Examples

```
SELECT
   FIRST_VALUE(my_column) OVER w
FROM my_table
WINDOW w AS (ORDER BY key);
```

```
SELECT
   LAST_VALUE(my_column) IGNORE NULLS OVER w
FROM my_table
WINDOW w AS (ORDER BY key);
```

NTH_VALUE

Access a value from a row specified by position in the window's `ORDER BY` order within window frame. Arguments - the expression to access and the row number, starting with 1.

Optionally, the `IGNORE NULLS` modifier can be specified before `OVER`, which causes rows with `NULL` in the first argument's value to be skipped. The antonym of this modifier is `RESPECT NULLS`, which is the default behavior and may be skipped.

Signature

```
NTH_VALUE(T,N)->T?
```

Examples

```
SELECT
   NTH_VALUE(my_column, 2) OVER w
FROM my_table
WINDOW w AS (ORDER BY key);
```

```
SELECT
   NTH_VALUE(my_column, 3) IGNORE NULLS OVER w
FROM my_table
WINDOW w AS (ORDER BY key);
```

RANK / DENSE_RANK / PERCENT_RANK

Number the groups of neighboring rows in the partition with the same expression value in the argument. `DENSE_RANK` numbers the groups one by one, and `RANK` skips `(N - 1)` values, with `N` being the number of rows in the previous group. `PERCENT_RANK` returns the relative rank of the current row: .

If there is no argument, it uses the order specified in the `ORDER BY` section in the window definition.
If the argument is omitted and `ORDER BY` is not specified, then all rows are considered equal to each other.

Note

Passing an argument to `RANK` / `DENSE_RANK` / `PERCENT_RANK` is a non-standard extension in YQL.

Signature

```
RANK([T])->Uint64
DENSE_RANK([T])->Uint64
PERCENT_RANK([T])->Double
```

Examples

```
SELECT
   RANK(my_column) OVER w
FROM my_table
WINDOW w AS (ORDER BY key);
```

```
SELECT
   DENSE_RANK() OVER w
FROM my_table
WINDOW w AS (ORDER BY my_column);
```

```
SELECT
   PERCENT_RANK() OVER w
FROM my_table
WINDOW w AS (ORDER BY my_column);
```

NTILE

Distributes the rows of an ordered partition into a specified number of groups. The groups are numbered starting with one. For each row, the `NTILE` function returns the number of the group to which the row belongs.

Signature

```
NTILE(Uint64)->Uint64
```

Examples

```
SELECT
    NTILE(10) OVER w AS group_num
FROM my_table
WINDOW w AS (ORDER BY key);
```

CUME_DIST

Returns the relative position (> 0 and <= 1) of a row within a partition. No arguments.

Signature

```
CUME_DIST()->Double
```

Examples

```
SELECT
    CUME_DIST() OVER w AS dist
FROM my_table
WINDOW w AS (ORDER BY key);
```

SessionState()

A non-standard window function `SessionState()` (without arguments) lets you get the session calculation status from SessionWindow for the current row.
It's allowed only if `SessionWindow()` is present in the `PARTITION BY` section in the window definition.

Functions for lists

ListCreate

Construct an empty list. The only argument specifies a string describing the data type of the list cell, or the type itself obtained using relevant functions. YQL doesn't support lists with an unknown cell type.

Documentation for the type definition format.

Examples

```
SELECT ListCreate(Tuple<String,Double?>);
```

```
SELECT ListCreate(OptionalType(DataType("String")));
```

AsList and AsListStrict

Construct a list based on one or more arguments. The argument types must be compatible in the case of `AsList` and strictly match in the case of `AsListStrict`.

Examples

```
SELECT AsList(1, 2, 3, 4, 5);
```

ListLength

The count of items in the list.

Examples

ListHasItems

Check that the list contains at least one item.

Examples

ListCollect

Convert a lazy list (it can be built by such functions as ListFilter, ListMap, ListFlatMap) to an eager list. In contrast to a lazy list, where each new pass re-calculates the list contents, in an eager list the content is built at once by consuming more memory.

Examples

ListSort, ListSortAsc, and ListSortDesc

Sort the list. By default, the ascending sorting order is applied (`ListSort` is an alias for `ListSortAsc`).

Arguments:

1. List.
2. An optional expression to get the sort key from a list element (it's the element itself by default).

Examples

```
$list = AsList(
    AsTuple("x", 3),
    AsTuple("xx", 1),
    AsTuple("a", 2)
);

SELECT ListSort($list, ($x) -> {
    RETURN $x.1;
});
```

Note

The example used a lambda function.

ListExtend and ListExtendStrict

Sequentially join lists (concatenation of lists). The arguments can be lists, optional lists, and `NULL`.
The types of list items must be compatible in the case of `ListExtend` and strictly match in the case of `ListExtendStrict`.
If at least one of the lists is optional, then the result is also optional.
If at least one argument is `NULL`, then the result type is `NULL`.

ListUnionAll

Sequentially join lists of structures (concatenation of lists). A field is added to the output list of structures if it exists in at least one source list, but if there is no such field in any list, it is added as NULL. In the case when a field is present in two or more lists, the output field is cast to the common type.

If at least one of the lists is optional, then the result is also optional.

ListZip and ListZipAll

Based on the input lists, build a list of pairs containing the list items with matching indexes (`List<Tuplefirst_list_element_type,second_list_element_type>`).

The length of the returned list is determined by the shortest list for ListZip and the longest list for ListZipAll.
When the shorter list is exhausted, a `NULL` value of a relevant optional type is paired with the elements of the longer list.

ListEnumerate

Build a list of pairs (Tuple) containing the element number and the element itself (`List<TupleUint64,list_element_type>`).

ListReverse

Reverse the list.

ListSkip

Returns a copy of the list, skipping the specified number of its first elements.

The first argument specifies the source list and the second argument specifies how many elements to skip.

ListTake

Returns a copy of the list containing a limited number of elements from the second list.

The first argument specifies the source list and the second argument specifies the maximum number of elements to be taken from the beginning of the list.

ListSample and ListSampleN

Returns a sample without replacement from the list.

- `ListSample` chooses elements independently with the specified probability.
- `ListSampleN` chooses a sample of the specified size (if the length of the list is less than the sample size, returns the original list).

If the probability/sample size is NULL, returns the original list.

An optional argument is used to control randomness, see documentation for `Random`.

Examples

```
ListSample(List<T>, Double?[, U])->List<T>
ListSample(List<T>?, Double?[, U])->List<T>?

ListSampleN(List<T>, Uint64?[, U])->List<T>
ListSampleN(List<T>?, Uint64?[, U])->List<T>?
```

```
$list = AsList(1, 2, 3, 4, 5);

SELECT ListSample($list, 0.5);  -- [1, 2, 5]
SELECT ListSampleN($list, 2);  -- [4, 2]
```

ListShuffle

Returns a shuffled copy of the list. An optional argument is used to control randomness, see documentation for `Random`.

Examples

```
ListShuffle(List<T>[, U])->List<T>
ListShuffle(List<T>?[, U])->List<T>?
```

```
$list = AsList(1, 2, 3, 4, 5);

SELECT ListShuffle($list);  -- [1, 3, 5, 2, 4]
```

ListIndexOf

Searches the list for an element with the specified value and returns its index at the first occurrence. Indexes count from 0. If such element is missing, it returns `NULL`.

ListMap, ListFilter, and ListFlatMap

Apply the function specified as the second argument to each list element. The functions differ in their returned result:

- `ListMap` returns a list with results.
- `ListFlatMap` returns a list with results, combining and expanding the first level of results (lists or optional values) for each item.
- `ListFilter` leaves only those elements where the function returned `true`.

> **Note**
>
> In `ListFlatMap`, using optional values in function results is deprecated, use the combination of `ListNotNull` and `ListMap` instead.

Arguments:

1. Source list.
2. Functions for processing list elements, such as:
   - Lambda function.
   - `Module::Function` - C++ UDF.

ListNotNull

Applies transformation to the source list, skipping empty optional items and strengthening the item type to non-optional. For a list with non-optional items, it returns the unchanged source list.

If the source list is optional, then the output list is also optional.

Examples

```
SELECT ListNotNull([1,2]),   -- [1,2]
    ListNotNull([3,null,4]); -- [3,4]
```

ListFlatten

Expands the list of lists into a flat list, preserving the order of items. As the top-level list item you can use an optional list that is interpreted as an empty list in the case of `NULL`.

If the source list is optional, then the output list is also optional.

Examples

```
SELECT ListFlatten([[1,2],[3,4]]),   -- [1,2,3,4]
    ListFlatten([null,[3,4],[5,6]]); -- [3,4,5,6]
```

ListUniq

Returns a copy of the list containing only distinct elements.

ListAny and ListAll

Returns `true` for a list of Boolean values, if:

- `ListAny`: At least one element is `true`.
- `ListAll`: All elements are `true`.

Otherwise, it returns false.

ListHas

Show whether the list contains the specified element. In this case, `NULL` values are considered equal to each other, and with a `NULL` input list, the result is always `false`.

ListHead, ListLast

Returns the first and last item of the list.

ListMin, ListMax, ListSum and ListAvg

Apply the appropriate aggregate function to all elements of the numeric list.

ListFold, ListFold1

Folding a list.

Arguments:

1. List
2. Initial state `U` for `ListFold`, `initLambda(item:T)->U` for `ListFold1`
3. `updateLambda(item:T, state:U)->U`

Type returned:
`U` for `ListFold`, `U?` for `ListFold1`.

```
$l = [1, 4, 7, 2];
$y = ($x, $y) -> { RETURN $x + $y; };
$z = ($x) -> { RETURN 4 * $x; };

SELECT
    ListFold($l, 6, $y) AS fold,                    -- 20
    ListFold([], 3, $y) AS fold_empty,              -- 3
    ListFold1($l, $z, $y) AS fold1,                 -- 17
    ListFold1([], $z, $y) AS fold1_empty;           -- Null
```

ListFoldMap, ListFold1Map

Converts each list item i by calling the handler(i, state).

Arguments:

1. List
2. Initial state `S` for `ListFoldMap`, `initLambda(item:T)->tuple (U S)` for `ListFold1Map`
3. `handler(item:T, state:S)->tuple (U S)`

Type returned: `List` of `U` items.

Examples

```
$l = [1, 4, 7, 2];
$x = ($i, $s) -> { RETURN ($i * $s, $i + $s); };
```

```
$t = ($i) -> { RETURN ($i + 1, $i + 2); };

SELECT
    ListFoldMap([], 1, $x),               -- []
    ListFoldMap($l, 1, $x),               -- [1, 8, 42, 26]
    ListFold1Map([], $t, $x),             -- []
    ListFold1Map($l, $t, $x);             -- [2, 12, 49, 28]
```

ListFromRange

Generate a sequence of numbers with the specified step. It's similar to `xrange` in Python 2, but additionally supports floats.

Arguments:

1. Start
2. End
3. Step (optional, 1 by default)

Specifics:

- The end is not included, i.e. `ListFromRange(1,3) == AsList(1,2)`.
- The type for the resulting elements is selected as the broadest from the argument types. For example, `ListFromRange(1, 2, 0.5)` results in a `Double` list.
- If the start and the end is one of the date representing type, the step has to be `Interval`.
- The list is "lazy", but if it's used incorrectly, it can still consume a lot of RAM.
- If the step is positive and the end is less than or equal to the start, the result list is empty.
- If the step is negative and the end is greater than or equal to the start, the result list is empty.
- If the step is neither positive nor negative (0 or NaN), the result list is empty.
- If any of the parameters is optional, the result list is optional.
- If any of the parameters is `NULL`, the result is `NULL`.

Examples

```
SELECT
    ListFromRange(-2, 2), -- [-2, -1, 0, 1]
    ListFromRange(2, 1, -0.5); -- [2.0, 1.5]
```

Signature

```
ListFromRange(T{Flags:AutoMap}, T{Flags:AutoMap}, T?)->LazyList<T> -- T – numeric type
ListFromRange(T{Flags:AutoMap}, T{Flags:AutoMap}, I?)->LazyList<T> -- T – type, representing date/time, I – i
nterval
```

ListReplicate

Creates a list containing multiple copies of the specified value.

Required arguments:

1. Value.
2. Number of copies.

Examples

```
SELECT ListReplicate(true, 3); -- [true, true, true]
```

ListConcat

Concatenates a list of strings into a single string.
You can set a separator as the second parameter.

ListExtract

For a list of structures, it returns a list of contained fields having the specified name.

ListTakeWhile, ListSkipWhile

`ListTakeWhile` returns a list from the beginning while the predicate is true, then the list ends.

`ListSkipWhile` skips the list segment from the beginning while the predicate is true, then returns the rest of the list ignoring the predicate.
`ListTakeWhileInclusive` returns a list from the beginning while the predicate is true. Then the list ends, but it also includes the item on which the stopping predicate triggered.
`ListSkipWhileInclusive` skips a list segment from the beginning while the predicate is true, then returns the rest of the list disregarding the predicate, but excluding the element that matched the predicate and starting with the next element after it.

Required arguments:

1. List.
2. Predicate.

If the input list is optional, then the result is also optional.

Examples

```
$data = AsList(1, 2, 5, 1, 2, 7);
```

```
SELECT
    ListTakeWhile($data, ($x) -> {return $x <= 3}), -- [1, 2]
    ListSkipWhile($data, ($x) -> {return $x <= 3}), -- [5, 1, 2, 7]
    ListTakeWhileInclusive($data, ($x) -> {return $x <= 3}), -- [1, 2, 5]
    ListSkipWhileInclusive($data, ($x) -> {return $x <= 3}); -- [1, 2, 7]
```

ListAggregate

Apply the aggregation factory to the passed list.
If the passed list is empty, the aggregation result is the same as for an empty table: 0 for the `COUNT` function and `NULL` for other functions.
If the passed list is optional and `NULL`, the result is also `NULL`.

Arguments:

1. List.
2. Aggregation factory.

Examples

```
SELECT ListAggregate(AsList(1, 2, 3), AggregationFactory("Sum")); -- 6
```

ToDict and ToMultiDict

Convert a list of tuples containing key-value pairs to a dictionary. In case of conflicting keys in the input list, `ToDict` leaves the first value and `ToMultiDict` builds a list of all the values.

It means that:

- `ToDict` converts `List<TupleK, V="">` to `Dict<K, V="">`
- `ToMultiDict` converts `List<TupleK, V>` to `Dict<K, List<V>>`

Optional lists are also supported, resulting in an optional dictionary.

ToSet

Converts a list to a dictionary where the keys are unique elements of this list, and values are omitted and have the type `Void`. For the `List<T>` list, the result type is `Dict<T, Void="">`.
An optional list is also supported, resulting in an optional dictionary.

Inverse function: get a list of keys for the DictKeys dictionary.

ListTop, ListTopAsc, ListTopDesc, ListTopSort, ListTopSortAsc и ListTopSortDesc

Select top values from the list. `ListTopSort*` additionally sorts the returned values. The smallest values are selected by default. Thus, the functions without a suffix are the aliases to `*Asc` functions, while `*Desc` functions return the largest values.

`ListTopSort` is more effective than consecutive `ListTop` and `ListSort` because `ListTop` can partially sort the list to find needed values. However, `ListTop` is more effective than `ListTopSort` when the result order is unimportant.

Arguments:

1. List.
2. Size of selection.
3. An optional expression to get the sort key from a list element (it's the element itself by default).

Examples

```
ListTop(List<T>{Flags:AutoMap}, N)->List<T>
ListTop(List<T>{Flags:AutoMap}, N, (T)->U)->List<T>
```

The signatures of other functions are the same.

Functions for dictionaries

DictCreate

Construct an empty dictionary. Two arguments are passed: for a key and a value. Each argument specifies a string with the data type declaration or the type itself built by type functions. There are no dictionaries with an unknown key or value type in YQL. As a key, you can set a primitive data type, except for `Yson` and `Json` that may be optional or a tuple of them of a length of at least two.

Documentation for the type definition format.

Examples

```
SELECT DictCreate(String, Tuple<String,Double?>);
```

```
SELECT DictCreate(Tuple<Int32?,String>, OptionalType(DataType("String")));
```

SetCreate

Construct an empty set. An argument is passed: the key type that can be built by type functions. There are no sets with an unknown key type in YQL. As a key, you can set a primitive data type, except for `Yson` and `Json` that may be optional or a tuple of them of a length of at least two.

Documentation for the type definition format.

Examples

```
SELECT SetCreate(String);
```

```
SELECT SetCreate(Tuple<Int32?,String>);
```

DictLength

The count of items in the dictionary.

Examples

```
SELECT DictLength(AsDict(AsTuple(1, AsList("foo", "bar"))));
```

DictHasItems

Check that the dictionary contains at least one item.

Examples

```
SELECT DictHasItems(AsDict(AsTuple(1, AsList("foo", "bar")))) FROM my_table;
```

DictItems

Get dictionary contents as a list of tuples including key-value pairs (`List<Tuplekey_type,value_type>`).

Examples

```
SELECT DictItems(AsDict(AsTuple(1, AsList("foo", "bar"))));
-- [ ( 1, [ "foo", "bar" ] ) ]
```

DictKeys

Get a list of dictionary keys.

Examples

```
SELECT DictKeys(AsDict(AsTuple(1, AsList("foo", "bar"))));
-- [ 1 ]
```

DictPayloads

Get a list of dictionary values.

Examples

```
SELECT DictPayloads(AsDict(AsTuple(1, AsList("foo", "bar"))));
-- [ [ "foo", "bar" ] ]
```

DictLookup

Get a dictionary element by its key.

Examples

```
SELECT DictLookup(AsDict(
    AsTuple(1, AsList("foo", "bar")),
```

```
    AsTuple(2, AsList("bar", "baz"))
), 1);
-- [ "foo", "bar" ]
```

DictContains

Checking if an element in the dictionary using its key. Returns true or false.

Examples

```
SELECT DictContains(AsDict(
    AsTuple(1, AsList("foo", "bar")),
    AsTuple(2, AsList("bar", "baz"))
), 42);
-- false
```

DictAggregate

Apply aggregation factory to the passed dictionary where each value is a list. The factory is applied separately inside each key.
If the list is empty, the aggregation result is the same as for an empty table: 0 for the `COUNT` function and `NULL` for other functions.
If the list under a certain key is empty in the passed dictionary, such a key is removed from the result.
If the passed dictionary is optional and contains `NULL`, the result is also `NULL`.

Arguments:

1. Dictionary.
2. Aggregation factory.

Examples

```
SELECT DictAggregate(AsDict(
    AsTuple(1, AsList("foo", "bar")),
    AsTuple(2, AsList("baz", "qwe"))),
    AggregationFactory("Max"));
-- {1 : "foo", 2 : "qwe" }
```

SetIsDisjoint

Check that the dictionary doesn't intersect by keys with a list or another dictionary.

So there are two options to make a call:

- With the `Dict<K,V1>` and `List<K>` arguments.
- With the `Dict<K,V1>` and `Dict<K,V2>` arguments.

Examples

```
SELECT SetIsDisjoint(ToSet(AsList(1, 2, 3)), AsList(7, 4)); -- true
SELECT SetIsDisjoint(ToSet(AsList(1, 2, 3)), ToSet(AsList(3, 4))); -- false
```

SetIntersection

Construct intersection between two dictionaries based on keys.

Arguments:

- Two dictionaries: `Dict<K,V1>` and `Dict<K,V2>`.
- An optional function that combines the values from the source dictionaries to construct the values of the output dictionary. If such a function has the `(K,V1,V2) -> U` type, the result type is `Dict<K,U>`. If the function is not specified, the result type is `Dict<K,Void>`, and the values from the source dictionaries are ignored.

Examples

```
SELECT SetIntersection(ToSet(AsList(1, 2, 3)), ToSet(AsList(3, 4))); -- { 3 }
SELECT SetIntersection(
    AsDict(AsTuple(1, "foo"), AsTuple(3, "bar")),
    AsDict(AsTuple(1, "baz"), AsTuple(2, "qwe")),
    ($k, $a, $b) -> { RETURN AsTuple($a, $b) });
-- { 1 : ("foo", "baz") }
```

SetIncludes

Checking that the keys of the specified dictionary include all the elements of the list or the keys of the second dictionary.

So there are two options to make a call:

- With the `Dict<K,V1>` and `List<K>` arguments.
- With the `Dict<K,V1>` and `Dict<K,V2>` arguments.

Examples

```
SELECT SetIncludes(ToSet(AsList(1, 2, 3)), AsList(3, 4)); -- false
SELECT SetIncludes(ToSet(AsList(1, 2, 3)), ToSet(AsList(2, 3))); -- true
```

SetUnion

Constructs a union of two dictionaries based on keys.

Arguments:

- Two dictionaries: `Dict<K,V1>` and `Dict<K,V2>`.
- An optional function that combines the values from the source dictionaries to construct the values of the output dictionary. If such a function has the `(K,V1?,V2?) -> U` type, the result type is `Dict<K,U>`. If the function is not specified, the result type is `Dict<K,Void>`, and the values from the source dictionaries are ignored.

Examples

```
SELECT SetUnion(ToSet(AsList(1, 2, 3)), ToSet(AsList(3, 4))); -- { 1, 2, 3, 4 }
SELECT SetUnion(
    AsDict(AsTuple(1, "foo"), AsTuple(3, "bar")),
    AsDict(AsTuple(1, "baz"), AsTuple(2, "qwe")),
    ($k, $a, $b) -> { RETURN AsTuple($a, $b) });
-- { 1 : ("foo", "baz"), 2 : (null, "qwe"), 3 : ("bar", null) }
```

SetDifference

Construct a dictionary containing all the keys with their values in the first dictionary with no matching key in the second dictionary.

Examples

```
SELECT SetDifference(ToSet(AsList(1, 2, 3)), ToSet(AsList(3, 4))); -- { 1, 2 }
SELECT SetDifference(
    AsDict(AsTuple(1, "foo"), AsTuple(2, "bar")),
    ToSet(AsList(2, 3)));
-- { 1 : "foo" }
```

SetSymmetricDifference

Construct a symmetric difference between two dictionaries based on keys.

Arguments:

- Two dictionaries: `Dict<K,V1>` and `Dict<K,V2>`.
- An optional function that combines the values from the source dictionaries to construct the values of the output dictionary. If such a function has the `(K,V1?,V2?) -> U` type, the result type is `Dict<K,U>`. If the function is not specified, the result type is `Dict<K,Void>`, and the values from the source dictionaries are ignored.

Examples

```
SELECT SetSymmetricDifference(ToSet(AsList(1, 2, 3)), ToSet(AsList(3, 4))); -- { 1, 2, 4 }
SELECT SetSymmetricDifference(
    AsDict(AsTuple(1, "foo"), AsTuple(3, "bar")),
    AsDict(AsTuple(1, "baz"), AsTuple(2, "qwe")),
    ($k, $a, $b) -> { RETURN AsTuple($a, $b) });
-- { 2 : (null, "qwe"), 3 : ("bar", null) }
```

Functions for structures

TryMember

Trying to get a field from the structure. If it's not found among the fields or null in the structure value, use the default value.

Arguments:

1. Structure.
2. Field name.
3. Default value.

```
$struct = <|a:1|>;
SELECT
  TryMember(
    $struct,
    "a",
    123
  ) AS a, -- 1
  TryMember(
    $struct,
    "b",
    123
  ) AS b; -- 123
```

ExpandStruct

Adding one or more new fields to the structure.

If the field set contains duplicate values, an error is returned.

Arguments:

- The first argument passes the source structure to be expanded.
- All the other arguments must be named, each argument adds a new field and the argument's name is used as the field's name (as in AsStruct).

Examples

```
$struct = <|a:1|>;
SELECT
  ExpandStruct(
    $struct,
    2 AS b,
    "3" AS c
  ) AS abc;
```

AddMember

Adding one new field to the structure. If you need to add multiple fields, better use ExpandStruct.

If the field set contains duplicate values, an error is returned.

Arguments:

1. Source structure.
2. Name of the new field.
3. Value of the new field.

Examples

```
$struct = <|a:1|>;
SELECT
  AddMember(
    $struct,
    "b",
    2
  ) AS ab;
```

RemoveMember

Removing a field from the structure.

If the entered field hasn't existed, an error is returned.

Arguments:

1. Source structure.
2. Field name.

Examples

```
$struct = <|a:1, b:2|>;
SELECT
  RemoveMember(
    $struct,
    "b"
  ) AS a;
```

ForceRemoveMember

Removing a field from the structure.

If the entered field hasn't existed, unlike RemoveMember, the error is not returned.

Arguments:

1. Source structure.
2. Field name.

Examples

```
$struct = <|a:1, b:2|>;
SELECT
  ForceRemoveMember(
    $struct,
    "c"
  ) AS ab;
```

ChooseMembers

Selecting fields with specified names from the structure.

If any of the fields haven't existed, an error is returned.

Arguments:

1. Source structure.
2. List of field names.

Examples

```
$struct = <|a:1, b:2, c:3|>;
SELECT
  ChooseMembers(
    $struct,
    ["a", "b"]
  ) AS ab;
```

RemoveMembers

Excluding fields with specified names from the structure.

If any of the fields haven't existed, an error is returned.

Arguments:

1. Source structure.
2. List of field names.

Examples

```
$struct = <|a:1, b:2, c:3|>;
SELECT
  RemoveMembers(
    $struct,
    ["a", "b"]
  ) AS c;
```

ForceRemoveMembers

Excluding fields with specified names from the structure.

If any of the fields haven't existed, it is ignored.

Arguments:

1. Source structure.
2. List of field names.

Examples

```
$struct = <|a:1, b:2, c:3|>;
SELECT
  ForceRemoveMembers(
    $struct,
    ["a", "b", "z"]
  ) AS c;
```

CombineMembers

Combining the fields from multiple structures into a new structure.

If the resulting field set contains duplicate values, an error is returned.

Arguments: two or more structures.

Examples

```
$struct1 = <|a:1, b:2|>;
$struct2 = <|c:3|>;
SELECT
  CombineMembers(
    $struct1,
    $struct2
  ) AS abc;
```

FlattenMembers

Combining the fields from multiple new structures into another new structure with prefix support.

If the resulting field set contains duplicate values, an error is returned.

Arguments: two or more tuples of two items: prefix and structure.

Examples

```
$struct1 = <|a:1, b:2|>;
$struct2 = <|c:3|>;
SELECT
  FlattenMembers(
    AsTuple("foo", $struct1), -- fooa, foob
    AsTuple("bar", $struct2)  -- barc
  ) AS abc;
```

StructMembers

Returns an unordered list of field names (possibly removing one Optional level) for a single argument that is a structure. For the `NULL` argument, an empty list of strings is returned.

Argument: structure

Examples

```
$struct = <|a:1, b:2|>;
SELECT
  StructMembers($struct); -- ['a', 'b']
```

RenameMembers

Renames the fields in the structure passed. In this case, you can rename a source field into multiple target fields. All fields not mentioned in the renaming as source names are moved to the result structure. If some source field is omitted in the rename list, an error is returned. For an Optional structure or `NULL`, the result has the same type.

Arguments:

1. Source structure.
2. A tuple of field names: the original name, the new name.

Examples

```
$struct = <|a:1, b:2|>;
SELECT
  RenameMembers($struct, [('a', 'c'), ('a', 'e')]); -- (b:2, c:1, e:1)
```

ForceRenameMembers

Renames the fields in the structure passed. In this case, you can rename a source field into multiple target fields. All fields not mentioned in the renaming as source names are moved to the result structure. If some source field is omitted in the rename list, the name is ignored. For an Optional structure or `NULL`, the result has the same type.

Arguments:

1. Source structure.
2. A tuple of field names: the original name, the new name.

Examples

```
$struct = <|a:1, b:2|>;
SELECT
  ForceRenameMembers($struct, [('a', 'c'), ('d', 'e')]); -- (b:2, c:1)
```

GatherMembers

Returns an unordered list of tuples including the field name and value. For the `NULL` argument, `EmptyList` is returned. It can be used only in the cases when the types of items in the structure are the same or compatible. Returns an optional list for an optional structure.

Argument: structure

Examples

```
$struct = <|a:1, b:2|>;
SELECT
```

```
  GatherMembers($struct); -- [('a', 1), ('b', 2)]
```

SpreadMembers

Creates a structure with a specified list of fields and applies a specified list of edits to it in the format (field name, field value). All types of fields in the resulting structure are the same and equal to the type of values in the update list with added Optional (unless they are optional already). If the field wasn't mentioned among the list of updated fields, it's returned as `NULL`. Among all updates for a field, the latest one is written. If the update list is Optional or `NULL`, the result has the same type. If the list of edits includes a field that is not in the list of expected fields, an error is returned.

Arguments:

1. List of tuples: field name, field value.
2. A list of all possible field names in the structure.

Examples

```
SELECT
  SpreadMembers([('a',1),('a',2)],['a','b']); -- (a: 2, b: null)
```

ForceSpreadMembers

Creates a structure with a specified list of fields and applies to it the specified list of updates in the format (field name, field value). All types of fields in the resulting structure are the same and equal to the type of values in the update list with added Optional (unless they are optional already). If the field wasn't mentioned among the list of updated fields, it's returned as `NULL`. Among all updates for a field, the latest one is written. If the update list is optional or equal to `NULL`, the result has the same type. If the list of updates includes a field that is not in the list of expected fields, this edit is ignored.

Arguments:

1. List of tuples: field name, field value.
2. A list of all possible field names in the structure.

Examples

```
SELECT
  ForceSpreadMembers([('a',1),('a',2),('c',100)],['a','b']); -- (a: 2, b: null)
```

StructUnion, StructIntersection, StructDifference, StructSymmetricDifference

Combine two structures using one of the four methods (using the provided lambda to merge fields with the same name):

- `StructUnion` adds all fields of both of the structures to the result.
- `StructIntersection` adds only the fields which are present in both of the structures.
- `StructDifference` adds only the fields of `left`, which are absent in `right`.
- `StructSymmetricDifference` adds all fields that are present in exactly one of the structures.

Signatures

```
StructUnion(left:Struct<...>, right:Struct<...>[, mergeLambda:(name:String, l:T1?, r:T2?)->T])->Struct<...>
StructIntersection(left:Struct<...>, right:Struct<...>[, mergeLambda:(name:String, l:T1?, r:T2?)->T])->Struct
<...>
StructDifference(left:Struct<...>, right:Struct<...>)->Struct<...>
StructSymmetricDifference(left:Struct<...>, right:Struct<...>)->Struct<...>
```

Arguments:

1. `left` - first structure.
2. `right` - second structure.
3. `mergeLambda` - *(optional)* function to merge fields with the same name (arguments: field name, `Optional` field value of the first struct, `Optional` field value of the second struct - arguments are `Nothing<T?>` in case of absence of the corresponding struct field). By default, if present, the first structure's field value is used; otherwise, the second one's value is used.

Examples

```
$merge = ($name, $l, $r) -> {
    return ($l ?? 0) + ($r ?? 0);
};
$left = <|a: 1, b: 2, c: 3|>;
$right = <|c: 1, d: 2, e: 3|>;

SELECT
    StructUnion($left, $right),                 -- <|a: 1, b: 2, c: 3, d: 2, e: 3|>
    StructUnion($left, $right, $merge),         -- <|a: 1, b: 2, c: 4, d: 2, e: 3|>
    StructIntersection($left, $right, $merge),  -- <|c: 4|>
    StructDifference($left, $right),            -- <|a: 1, b: 1|>
    StructSymmetricDifference($left, $right)    -- <|a: 1, b: 2, d: 2, e: 3|>
;
```

Functions for data types

FormatType

Serializing a type or a handle type to a human-readable string. This helps at debugging and will also be used in the next examples of this section. Documentation for the format.

ParseType

Building a type from a string with description. Documentation for its format.

Examples

```
SELECT FormatType(ParseType("List<Int32>"));  -- List<int32>
```

TypeOf

Getting the type of value passed to the argument.

Examples

```
SELECT FormatType(TypeOf("foo"));  -- String
```

```
SELECT FormatType(TypeOf(AsTuple(1, 1u))); -- Tuple<Int32,Uint32>
```

InstanceOf

Returns an instance of the specified type that can only be used to get the type of the result of an expression that uses this type.

If this instance remains in the computation graph by the end of optimization, the operation fails.

Examples

```
SELECT FormatType(TypeOf(
    InstanceOf(ParseType("Int32")) +
    InstanceOf(ParseType("Double"))
)); -- Double, because "Int32 + Double" returns Double
```

DataType

Returns a type for primitive data types based on type name.

Examples

```
SELECT FormatType(DataType("Bool")); -- Bool
SELECT FormatType(DataType("Decimal","5","1")); -- Decimal(5,1)
```

OptionalType

Adds the option to assign `NULL` to the passed type.

Examples

```
SELECT FormatType(OptionalType(DataType("Bool"))); -- Bool?
```

ListType and StreamType

Builds a list type or stream type based on the passed element type.

Examples

```
SELECT FormatType(ListType(DataType("Bool"))); -- List<Bool>
```

DictType

Builds a dictionary type based on the passed key types (first argument) and value types (second argument).

Examples

```
SELECT FormatType(DictType(
    DataType("String"),
    DataType("Double")
)); -- Dict<String,Double>
```

TupleType

Builds the tuple type from the passed element types.

Examples

```
SELECT FormatType(TupleType(
    DataType("String"),
    DataType("Double"),
```

```
    OptionalType(DataType("Bool"))
)); -- Tuple<String,Double,Bool?>
```

StructType

Builds the structure type based on the passed element types. The standard syntax of named arguments is used to specify the element names.

Examples

```
SELECT FormatType(StructType(
    DataType("Bool") AS MyBool,
    ListType(DataType("String")) AS StringList
)); -- Struct<'MyBool':Bool,'StringList':List<String>>
```

VariantType

Returns the type of a variant based on the underlying type (structure or tuple).

Examples

```
SELECT FormatType(VariantType(
  ParseType("Struct<foo:Int32,bar:Double>")
)); -- Variant<'bar':Double,'foo':Int32>
```

ResourceType

Returns the type of the resource based on the passed string label.

Examples

```
SELECT FormatType(ResourceType("Foo")); -- Resource<'Foo'>
```

CallableType

Constructs the type of the called value using the following arguments:

1. Number of optional arguments (if all arguments are required — 0).
2. Result type.
3. All the next arguments of CallableType are treated as types of arguments of the callable value, but with a shift for two required arguments (for example, the third argument of the CallableType describes the type of the first argument in the callable value).

Examples

```
SELECT FormatType(CallableType(
  1, -- optional args count
  DataType("Double"), -- result type
  DataType("String"), -- arg #1 type
  OptionalType(DataType("Int64")) -- arg #2 type
)); -- Callable<(String,[Int64?])->Double>
```

GenericType, UnitType, and VoidType

Return the same-name special data types. They have no arguments because they are not parameterized.

Examples

```
SELECT FormatType(VoidType()); -- Void
```

OptionalItemType, ListItemType and StreamItemType

If a type is passed to these functions, then they perform the action reverse to OptionalType, ListType, and StreamType: return the item type based on its container type.

If a type handle is passed to these functions, then they perform the action reverse to OptionalTypeHandle, ListTypeHandle, and StreamTypeHandle: they return the handle of the element type based on the type handle of its container.

Examples

```
SELECT FormatType(ListItemType(
  ParseType("List<Int32>")
)); -- Int32
```

```
SELECT FormatType(ListItemType(
  ParseTypeHandle("List<Int32>")
)); -- Int32
```

DictKeyType and DictPayloadType

Returns the type of the key or value based on the dictionary type.

Examples

```
SELECT FormatType(DictKeyType(
  ParseType("Dict<Int32,String>")
)); -- Int32
```

TupleElementType

Returns the tuple's element type based on the tuple type and the element index (index starts from zero).

Examples

```
SELECT FormatType(TupleElementType(
  ParseType("Tuple<Int32,Double>"), "1"
)); -- Double
```

StructMemberType

Returns the type of the structure element based on the structure type and element name.

Examples

```
SELECT FormatType(StructMemberType(
  ParseType("Struct<foo:Int32,bar:Double>"), "foo"
)); -- Int32
```

CallableResultType and CallableArgumentType

`CallableResultType` returns the result type based on the type of the called value. `CallableArgumentType` returns the argument type based on the called value type and its index (index starts from zero).

Examples

```
$callable_type = ParseType("(String,Bool)->Double");

SELECT FormatType(CallableResultType(
    $callable_type
)), -- Double
FormatType(CallableArgumentType(
    $callable_type, 1
)); -- Bool
```

VariantUnderlyingType

If a type is passed to this function, then it an action reverse to VariantType: it returns the underlying type based on the variant type.

If a type handle is passed to this function, it performs the action reverse to VariantTypeHandle: returns the handle of the underlying type based on the handle of the variant type.

Examples

```
SELECT FormatType(VariantUnderlyingType(
  ParseType("Variant<foo:Int32,bar:Double>")
)), -- Struct<'bar':Double,'foo':Int32>
FormatType(VariantUnderlyingType(
  ParseType("Variant<Int32,Double>")
)); -- Tuple<Int32,Double>
```

```
SELECT FormatType(VariantUnderlyingType(
  ParseTypeHandle("Variant<foo:Int32,bar:Double>")
)), -- Struct<'bar':Double,'foo':Int32>
FormatType(VariantUnderlyingType(
  ParseTypeHandle("Variant<Int32,Double>")
)); -- Tuple<Int32,Double>
```

Functions for data types during calculations

To work with data types during calculations, use handle types: these are resources that contain an opaque type definition. After constructing the type handle, you can revert to the regular type using the EvaluateType function. For debug purposes, you can convert a handle type to a string using the FormatType function.

TypeHandle

Getting a type handle from the type passed to the argument.

Examples

```
SELECT FormatType(TypeHandle(TypeOf("foo")));  -- String
```

EvaluateType

Getting the type from the type handle passed to the argument. The function is evaluated before the start of the main calculation, as well as EvaluateExpr.

Examples

```
SELECT FormatType(EvaluateType(TypeHandle(TypeOf("foo"))));  -- String
```

ParseTypeHandle

Building a type handle from a string with description. Documentation for its format.

Examples

```
SELECT FormatType(ParseTypeHandle("List<Int32>"));  -- List<int32>
```

TypeKind

Getting the top-level type name from the type handle passed to the argument.

Examples

```
SELECT TypeKind(TypeHandle(TypeOf("foo")));  -- Data
SELECT TypeKind(ParseTypeHandle("List<Int32>"));  -- List
```

DataTypeComponents

Getting the name and parameters for a primitive data type from the primitive type handle passed to the argument. Reverse function: DataTypeHandle.

Examples

```
SELECT DataTypeComponents(TypeHandle(TypeOf("foo")));  -- ["String"]
SELECT DataTypeComponents(ParseTypeHandle("Decimal(4,1)"));  -- ["Decimal", "4", "1"]
```

DataTypeHandle

Constructing a handle for a primitive data type from its name and parameters passed to the argument as a list. Reverse function: DataTypeComponents.

Examples

```
SELECT FormatType(DataTypeHandle(
    AsList("String")
)); -- String

SELECT FormatType(DataTypeHandle(
    AsList("Decimal", "4", "1")
)); -- Decimal(4,1)
```

OptionalTypeHandle

Adds the option to assign `NULL` to the passed type handle.

Examples

```
SELECT FormatType(OptionalTypeHandle(
    TypeHandle(DataType("Bool"))
)); -- Bool?
```

PgTypeName

Getting the name of the PostgreSQL type from the type handle passed to the argument. Inverse function: PgTypeHandle.

Examples

```
SELECT PgTypeName(ParseTypeHandle("pgint4")); -- int4
```

PgTypeHandle

Builds a type handle based on the passed name of the PostgreSQL type. Inverse function: PgTypeName.

Examples

```
SELECT FormatType(PgTypeHandle("int4")); -- pgint4
```

ListTypeHandle and StreamTypeHandle

Builds a list type handle or stream type handle based on the passed element type handle.

Examples

```
SELECT FormatType(ListTypeHandle(
    TypeHandle(DataType("Bool"))
)); -- List<Bool>
```

EmptyListTypeHandle and EmptyDictTypeHandle

Constructs a handle for an empty list or dictionary.

Examples

```
SELECT FormatType(EmptyListTypeHandle()); -- EmptyList
```

TupleTypeComponents

Getting a list of element type handles from the tuple type handle passed to the argument. Inverse function: TupleTypeHandle.

Examples

```
SELECT ListMap(
   TupleTypeComponents(
       ParseTypeHandle("Tuple<Int32, String>")
   ),
   ($x)->{
       return FormatType($x)
   }
); -- ["Int32", "String"]
```

TupleTypeHandle

Building a tuple type handle from handles of element types passed as a list to the argument. Inverse function: TupleTypeComponents.

Examples

```
SELECT FormatType(
    TupleTypeHandle(
        AsList(
            ParseTypeHandle("Int32"),
            ParseTypeHandle("String")
        )
    )
); -- Tuple<Int32,String>
```

StructTypeComponents

Getting a list of element type handles and their names from the structure type handle passed to the argument. Inverse function: StructTypeHandle.

Examples

```
SELECT ListMap(
    StructTypeComponents(
        ParseTypeHandle("Struct<a:Int32, b:String>")
    ),
    ($x) -> {
        return AsTuple(
            FormatType($x.Type),
            $x.Name
        )
    }
); -- [("Int32","a"), ("String","b")]
```

StructTypeHandle

Building a structure type handle from handles of element types and names passed as a list to the argument. Inverse function: StructTypeComponents.

Examples

```
SELECT FormatType(
    StructTypeHandle(
        AsList(
            AsStruct(ParseTypeHandle("Int32") as Type,"a" as Name),
            AsStruct(ParseTypeHandle("String") as Type, "b" as Name)
        )
    )
); -- Struct<'a':Int32,'b':String>
```

DictTypeComponents

Getting a key-type handle and a value-type handle from the dictionary-type handle passed to the argument. Inverse function: DictTypeHandle.

Examples

```
$d = DictTypeComponents(ParseTypeHandle("Dict<Int32,String>"));

SELECT
    FormatType($d.Key),     -- Int32
    FormatType($d.Payload); -- String
```

DictTypeHandle

Building a dictionary-type handle from a key-type handle and a value-type handle passed to arguments. Inverse function: DictTypeComponents.

Examples

```
SELECT FormatType(
    DictTypeHandle(
        ParseTypeHandle("Int32"),
        ParseTypeHandle("String")
    )
); -- Dict<Int32, String>
```

ResourceTypeTag

Getting the tag from the resource type handle passed to the argument. Inverse function: ResourceTypeHandle.

Examples

```
SELECT ResourceTypeTag(ParseTypeHandle("Resource<foo>")); -- foo
```

ResourceTypeHandle

Building a resource-type handle from the tag value passed to the argument. Inverse function: ResourceTypeTag.

Examples

```
SELECT FormatType(ResourceTypeHandle("foo")); -- Resource<'foo'>
```

TaggedTypeComponents

Getting the tag and the basic type from the decorated type handle passed to the argument. Inverse function: TaggedTypeHandle.

Examples

```
$t = TaggedTypeComponents(ParseTypeHandle("Tagged<Int32,foo>"));

SELECT FormatType($t.Base), $t.Tag; -- Int32, foo
```

TaggedTypeHandle

Constructing a decorated type handle based on the base type handle and the tag name passed in arguments. Inverse function: TaggedTypeComponents.

Examples

```
SELECT FormatType(TaggedTypeHandle(
    ParseTypeHandle("Int32"), "foo"
)); -- Tagged<Int32, 'foo'>
```

VariantTypeHandle

Building a variant-type handle from the handle of the underlying type passed to the argument. Inverse function: VariantUnderlyingType.

Examples

```
SELECT FormatType(VariantTypeHandle(
    ParseTypeHandle("Tuple<Int32, String>")
)); -- Variant<Int32, String>
```

VoidTypeHandle and NullTypeHandle

Constructing a handle for Void and Null types, respectively.

Examples

```
SELECT FormatType(VoidTypeHandle()); -- Void
SELECT FormatType(NullTypeHandle()); -- Null
```

CallableTypeComponents

Getting the handle description for the type of callable value passed to the argument. Inverse function: CallableTypeHandle.

Examples

```
$formatArgument = ($x) -> {
    return AsStruct(
        FormatType($x.Type) as Type,
        $x.Name as Name,
        $x.Flags as Flags
    )
};

$formatCallable = ($x) -> {
```

```
    return AsStruct(
        $x.OptionalArgumentsCount as OptionalArgumentsCount,
        $x.Payload as Payload,
        FormatType($x.Result) as Result,
        ListMap($x.Arguments, $formatArgument) as Arguments
    )
};

SELECT $formatCallable(
    CallableTypeComponents(
        ParseTypeHandle("(Int32,[bar:Double?{Flags:AutoMap}])->String")
    )
);  -- (OptionalArgumentsCount: 1, Payload: "", Result: "String", Arguments: [
    --   (Type: "Int32", Name: "", Flags: []),
    --   (Type: "Double?", Name: "bar", Flags: ["AutoMap"]),
    -- ])
```

CallableArgument

Packing the description of the argument of the callable value into the structure to be passed to the CallableTypeHandle function with the following arguments:

1. Argument type handle.
2. Optional argument name. The default value is an empty string.
3. A list of strings with optional argument flags. The default value is an empty list. Supported flags are "AutoMap".

CallableTypeHandle

Constructing the type handle of the called value using the following arguments:

1. Handle of the return value type.
2. List of descriptions of arguments received using the CallableArgument function.
3. Optional number of optional arguments in the callable value. The default value is 0.
4. An optional label for the called value type. The default value is an empty string.

Inverse function: CallableTypeComponents.

Examples

```
SELECT FormatType(
    CallableTypeHandle(
        ParseTypeHandle("String"),
        AsList(
            CallableArgument(ParseTypeHandle("Int32")),
            CallableArgument(ParseTypeHandle("Double?"), "bar", AsList("AutoMap"))
        ),
        1
    )
);  -- Callable<(Int32,['bar':Double?{Flags:AutoMap}])->String>
```

LambdaArgumentsCount

Getting the number of arguments in a lambda function.

Examples

```
SELECT LambdaArgumentsCount(($x, $y)->($x+$y))
; -- 2
```

Functions for code generation

When running calculations, you can generate the code including S-expressions nodes. This uses a mechanism for packing the code in the resource. After building the code, you can insert it into the main program using the EvaluateCode function. For debugging purposes, you can convert the code to a string using the FormatCode function.

Possible node types in S-expressions that can be used for code generation:

- An atom is an untyped string of zero or more characters.
- A list is a sequence of zero or more nodes. It corresponds to the `tuple` type in SQL.
- A call of a built-in function consists of a name expressed by an atom and a sequence of zero or more nodes that are arguments to this function.
- Lambda function declaration consists of declaring the names of arguments and a node that is the root of the body for this lambda function.
- The lambda function argument is a node that can only be used inside the body of the lambda function.
- World is a special node that labels I/O operations.

The S-expressions nodes form a directed graph. Atoms are always leaf nodes, because they cannot contain child nodes.

In the text representation, S-expressions have the following format:

- Atom: `'"foo"`. The apostrophe character (') denotes quoting of the next line that is usually enclosed in quotation marks.
- List: `'("foo" "bar")`. The apostrophe character (') denotes that there will be no function call in parentheses.
- Calling the built-in function: `(foo "bar")`. The first item inside the brackets is the mandatory name of the function followed by the function arguments.
- Declaring a lambda function: `(lambda '(x y) (+ x y))`. The `lambda` keyword is followed by a list of argument names and then by the body of the lambda function.
- The lambda function argument is `x`. Unlike an atom, a string without an apostrophe character (') references a name in the current scope. When declaring a lambda function, the names of arguments are added to the body's visibility scope, and, if needed, the name is hidden from the global scope.
- The `world`.

FormatCode

Serializing the code as S-expressions. The code must not contain free arguments of functions, hence, to serialize the lambda function code, you must pass it completely, avoiding passing individual expressions that might contain lambda function arguments.

Examples

```
SELECT FormatCode(AtomCode("foo"));
-- (
-- (return '"foo")
-- )
```

WorldCode

Build a code node with the `world` type.

Examples

```
SELECT FormatCode(WorldCode());
-- (
-- (return world)
-- )
```

AtomCode

Build a code node with the `atom` type from a string passed to the argument.

Examples

```
SELECT FormatCode(AtomCode("foo"));
-- (
-- (return '"foo")
-- )
```

ListCode

Build a code node with the `list` type from a set of nodes or lists of code nodes passed to arguments. In this case, lists of arguments are built in as separately listed code nodes.

Examples

```
SELECT FormatCode(ListCode(
    AtomCode("foo"),
    AtomCode("bar")));
-- (
-- (return '('"foo" '"bar"))
-- );

SELECT FormatCode(ListCode(AsList(
    AtomCode("foo"),
    AtomCode("bar"))));
-- (
```

```
-- (return '('"foo" '"bar"))
-- )
```

FuncCode

Build a code node with the `built-in function call` from a string with the function name and a set of nodes or lists of code nodes passed to arguments. In this case, lists of arguments are built in as separately listed code nodes.

Examples

```
SELECT FormatCode(FuncCode(
    "Baz",
    AtomCode("foo"),
    AtomCode("bar")));
-- (
-- (return (Baz '"foo" '"bar"))
-- )

SELECT FormatCode(FuncCode(
    "Baz",
    AsList(
        AtomCode("foo"),
        AtomCode("bar"))));
-- (
-- (return (Baz '"foo" '"bar"))
-- )
```

LambdaCode

You can build a code node with the `lambda function declaration` type from:

- a Lambda function, if you know the number of arguments in advance. In this case, the nodes of the `argument` type will be passed as arguments to this lambda function.
- The number of arguments and a lambda function with one argument. In this case, a list of nodes of the `argument` type will be passed as an argument to this lambda function.

Examples

```
SELECT FormatCode(LambdaCode(($x, $y) -> {
    RETURN FuncCode("+", $x, $y);
}));
-- (
-- (return (lambda '($1 $2) (+ $1 $2)))
-- )

SELECT FormatCode(LambdaCode(2, ($args) -> {
    RETURN FuncCode("*", Unwrap($args[0]), Unwrap($args[1]));
}));
-- (
-- (return (lambda '($1 $2) (* $1 $2)))
-- )
```

EvaluateCode

Substituting the code node passed in the argument, into the main program code.

Examples

```
SELECT EvaluateCode(FuncCode("Int32", AtomCode("1"))); -- 1

$lambda = EvaluateCode(LambdaCode(($x, $y) -> {
    RETURN FuncCode("+", $x, $y);
}));
SELECT $lambda(1, 2); -- 3
```

ReprCode

Substituting the code node representing the result of evaluating an expression passed in the argument, into the main program.

Examples

```
$add3 = EvaluateCode(LambdaCode(($x) -> {
    RETURN FuncCode("+", $x, ReprCode(1 + 2));
}));
SELECT $add3(1); -- 4
```

QuoteCode

Substituting into the main program the code node that represents an expression or a lambda function passed in the argument. If free arguments of lambda functions were found during the substitution, they are calculated and substituted into the code as in the ReprCode function.

Examples

```
$lambda = ($x, $y) -> { RETURN $x + $y };
$makeClosure = ($y) -> {
    RETURN EvaluateCode(LambdaCode(($x) -> {
```

```
        RETURN FuncCode("Apply", QuoteCode($lambda), $x, ReprCode($y))
    }))
};

$closure = $makeClosure(2);
SELECT $closure(1); -- 3
```

Functions for JSON

JSON is a lightweight data-interchange format. In YQL, it's represented by the `Json` type. Unlike relational tables, JSON can store data with no schema defined. Here is an example of a valid JSON object:

```
[
    {
        "name": "Jim Holden",
        "age": 30
    },
    {
        "name": "Naomi Nagata",
        "age": "twenty years old"
    }
]
```

Despite the fact that the `age` field in the first object is of the `Number` type (`"age": 21`) and in the second object its type is `String` (`"age": "twenty years old"`), this is a fully valid JSON object.

To work with JSON, YQL implements a subset of the SQL support for JavaScript Object Notation (JSON) standard, which is part of the common ANSI SQL standard.

JsonPath

Values inside JSON objects are accessed using a query language called JsonPath. All functions for JSON accept a JsonPath query as an argument.

Let's look at an example. Suppose we have a JSON object like:

```
{
    "comments": [
        {
            "id": 123,
            "text": "A whisper will do, if it's all that you can manage."
        },
        {
            "id": 456,
            "text": "My life has become a single, ongoing revelation that I haven't been cynical enough."
        }
    ]
}
```

Then, to get the text of the second comment, we can write the following JsonPath query:

```
$.comments[1].text
```

In this query:

1. `$` is a way to access the entire JSON object.
2. `$.comments` accesses the `comments` key of the JSON object.
3. `$.comments[1]` accesses the second element of the JSON array (element numbering starts from 0).
4. `$.comments[1].text` accesses the `text` key of the JSON object.
5. Query execution result: `"My life has become a single, ongoing revelation that I haven't been cynical enough."`

Quick reference

Operation	Example
Retrieving a JSON object key	`$.key`
Retrieving all JSON object keys	`$.*`
Accessing an array element	`$[25]`
Retrieving an array subsegment	`$[2 to 5]`
Accessing the last array element	`$[last]`
Accessing all array elements	`$[*]`
Unary operations	`- 1`
Binary operations	`(12 * 3) % 4 + 8`
Accessing a variable	`$variable`
Logical operations	`(1 > 2) || (3 <= 4) && ("string" == "another")|`
Matching a regular expression	`$.name like_regex "^[A-Za-z]+$"`
Checking the string prefix	`$.name starts with "Bobbie"`
Checking if a path exists	`exists ($.profile.name)`
Checking a Boolean expression for null	`($.age > 20) is unknown`
Filtering values	`$.friends ? (@.age >= 18 && @.gender == "male")`

Getting the value type	`$.name.type()`
Getting the array size	`$.friends.size()`
Converting a string to a number	`$.number.double()`
Rounding up a number	`$.number.ceiling()`
Rounding down a number	`$.number.floor()`
Returning the absolute value	`$.number.abs()`
Getting key-value pairs from an object	`$.profile.keyvalue()`

Data model

The result of executing all JsonPath expressions is a sequence of JSON values. For example:

- The result of executing the `"Bobbie"` expression is a sequence with the only element `"Bobbie"`. Its length is 1.
- The result of executing the `$` expression (that takes the entire JSON object) in JSON `[1, 2, 3]` is `[1, 2, 3]`. A sequence of 1 element of the array `[1, 2, 3]`
- The result of executing the `$[*]` expression (retrieving all array elements) in JSON `[1, 2, 3]` is `1, 2, 3`. A sequence of three items: `1`, `2`, and `3`

If the input sequence consists of multiple values, some operations are performed for each element (for example, accessing a JSON object key). However, other operations require a sequence of one element as input (for example, binary arithmetic operations).

The behavior of a specific operation is described in the corresponding section of the documentation.

Execution mode

JsonPath supports two execution modes, `lax` and `strict`. Setting the mode is optional. By default, `lax`. The mode is specified at the beginning of a query. For example, `strict $.key`.

The behavior for each mode is described in the corresponding sections with JsonPath operations.

Auto unpacking of arrays

When accessing a JSON object key in `lax` mode, arrays are automatically unpacked.

Example

```
[
    {
        "key": 123
    },
    {
        "key": 456
    }
]
```

The `lax $.key` query is successful and returns `123, 456`. As `$` is an array, it's automatically unpacked and accessing the key of the `$.key` JSON object is executed for each element in the array.

The `strict $.key` query returns an error. In `strict` mode, there is no support for auto unpacking of arrays. Since `$` is an array and not an object, accessing the `$.key` object key is impossible. You can fix this by writing `strict $[*].key`.

Unpacking is only 1 level deep. In the event of nested arrays, only the outermost one is unpacked.

Wrapping values in arrays

When accessing an array element in `lax` mode, JSON values are automatically wrapped in an array.

Example

```
{
    "name": "Avasarala"
}
```

The `lax $[0].name` query is successful and returns `"Avasarala"`. As `$` isn't an array, it's automatically wrapped in an array of length 1. Accessing the first element `$[0]` returns the source JSON object where the `name` key is taken.

The `strict $[0].name` query returns an error. In `strict` mode, values aren't wrapped in an array automatically. Since `$` is an object and not an array, accessing the `$[0]` element is impossible. You can fix this by writing `strict $.name`.

Handling errors

Some errors are converted to an empty result when a query is executed in `lax` mode.

Literals

Values of some types can be specified in a JsonPath query using literals:

Type	Example
Numbers	`42`, `-1.23e-5`
Boolean values	`false`, `true`
Null	`Null`

Stings	`"Belt"`

Accessing JSON object keys

JsonPath supports accessing JSON object keys, such as `$.session.user.name`.

> **Note**
>
> Accessing keys without quotes is only supported for keys that start with an English letter or underscore and only contain English letters, underscores, numbers, and a dollar sign. Use quotes for all other keys. For example, `$.profile."this string has spaces"` or `$.user."42 is the answer"`

For each value from the input sequence:

1. If the value is an array, it's automatically unpacked in `lax` mode.
2. If the value isn't a JSON object or if it is and the specified key is missing from this JSON object, a query executed in `strict` mode fails. In `lax` mode, an empty result is returned for this value.

The expression execution result is the concatenation of the results for each value from the input sequence.

Example

```
{
    "name": "Amos",
    "friends": [
        {
            "name": "Jim"
        },
        {
            "name": "Alex"
        }
    ]
}
```

	`lax`	`strict`
`$.name`	`"Amos"`	`"Amos"`
`$.surname`	Empty result	Error
`$.friends.name`	`"Jim", "Alex"`	Error

Accessing all JSON object keys

JsonPath supports accessing all JSON object keys at once: `$.*`.

For each value from the input sequence:

1. If the value is an array, it's automatically unpacked in `lax` mode.
2. If the value isn't a JSON object, a query executed in `strict` mode fails. In `lax` mode, an empty result is returned for this value.

The expression execution result is the concatenation of the results for each value from the input sequence.

Example

```
{
    "profile": {
        "id": 123,
        "name": "Amos"
    },
    "friends": [
        {
            "name": "Jim"
        },
        {
            "name": "Alex"
        }
    ]
}
```

	`lax`	`strict`
`$.profile.*`	`123, "Amos"`	`123, "Amos"`
`$.friends.*`	`"Jim", "Alex"`	Error

Accessing an array element

JsonPath supports accessing array elements: `$.friends[1, 3 to last - 1]`.

For each value from the input sequence:

1. If the value isn't an array, a query executed in `strict` mode fails. In `lax` mode, values are automatically wrapped in an array.
2. The `last` keyword is replaced with the array's last index. Using `last` outside of accessing the array is an error in both modes.
3. The specified indexes are calculated. Each of them must be a single number, otherwise the query fails in both modes.

4. If the index is a fractional number, it's rounded down.
5. If the index goes beyond the array boundaries, the query executed in `strict` mode fails. In `lax` mode, this index is ignored.
6. If a segment is specified and its start index is greater than the end index (for example, `$[20 to 1]`), the query fails in `strict` mode. In `lax` mode, this segment is ignored.
7. All elements by the specified indexes are added to the result. Segments include **both ends**.

Examples

```
[
    {
        "name": "Camina",
        "surname": "Drummer"
    },
    {
        "name": "Josephus",
        "surname": "Miller"
    },
    {
        "name": "Bobbie",
        "surname": "Draper"
    },
    {
        "name": "Julie",
        "surname": "Mao"
    }
]
```

	`lax`	`strict`
`$[0].name`	`"Camina"`	`"Camina"`
`$[1, 2 to 3].name`	`"Josephus", "Bobbie", "Julie"`	`"Josephus", "Bobbie", "Julie"`
`$[last - 2].name`	`"Josephus"`	`"Josephus"`
`$[2, last + 200 to 50].name`	`"Bobbie"`	Error
`$[50].name`	Empty result	Error

Accessing all array elements

JsonPath supports accessing all array elements at once: `$[*]`.

For each value from the input sequence:

1. If the value isn't an array, a query executed in `strict` mode fails. In `lax` mode, values are automatically wrapped in an array.
2. All elements of the current array are added to the result.

Examples

```
[
    {
        "class": "Station",
        "title": "Medina"
    },
    {
        "class": "Corvette",
        "title": "Rocinante"
    }
]
```

	`lax`	`strict`
`$[*].title`	`"Medina", "Rocinante"`	`"Medina", "Rocinante"`
`lax $[0][*].class`	`"Station"`	Error

Let's analyze the last example step by step:

1. `$[0]` returns the first element of the array, that is `{"class": "Station", "title": "Medina"}`
2. `$[0][*]` expects an array for input, but an object was input instead. It's automatically wrapped in an array as `[ {"class": "Station", "title": "Medina"} ]`
3. Now, `$[0][*]` can be executed and returns all elements of the array, that is `{"class": "Station", "title": "Medina"}`
4. `$[0][*].class` returns the `class` field value: `"Station"`.

Arithmetic operations

> **Note**
>
> All arithmetic operations work with numbers as with Double. Calculations are made with potential loss of accuracy.

Unary operations

JsonPath supports unary `+` and `-`.

A unary operation applies to all values from the input sequence. If a unary operation's input is a value that isn't a number, a query fails in both modes.

Example

```
[1, 2, 3, 4]
```

The `strict -$[*]` query is successful and returns `-1, -2, -3, -4`.

The `lax -$` query fails as `$` is an array and not a number.

Binary operations

JsonPath supports binary arithmetic operations (in descending order of priority):

1. Multiplication `*`, dividing floating-point numbers `/`, and taking the remainder `%` (works as the `MOD` function in `SQL`).
2. Addition `+`, subtraction `-`.

You can change the order of operations using parentheses.

If each argument of a binary operation is not a single number or a number is divided by 0, the query fails in both modes.

Examples

- `(1 + 2) * 3` returns `9`
- `1 / 2` returns `0.5`
- `5 % 2` returns `1`
- `1 / 0` fails
- If JSON is `[-32.4, 5.2]`, the `$[0] % $[1]` query returns `-1.2`
- If JSON is `[1, 2, 3, 4]`, the `lax $[*] + $[*]` query fails as the `$[*]` expression execution result is `1, 2, 3, 4`, that is multiple numbers. A binary operation only requires one number for each of its arguments.

Boolean values

Unlike some other programming languages, Boolean values in JsonPath are not only `true` and `false`, but also `null` (uncertainty).

JsonPath considers any values received from a JSON document to be non-Boolean. For example, a query like `! $.is_valid_user` (a logical negation applied to the `is_valid_user`) field is syntactically invalid because the `is_valid_user` field value is not Boolean (even when it actually stores `true` or `false`). The correct way to write this kind of query is to explicitly use a comparison with a Boolean value, such as `$.is_valid_user == false`.

Logical operations

JsonPath supports some logical operations for Boolean values.

The arguments of any logical operation must be a single Boolean value.
All logical operations return a Boolean value.

Logical negation, !

Truth table:

`x`	`!x`
`true`	`false`
`false`	`true`
`Null`	`Null`

Logical AND, &&

In the truth table, the first column is the left argument, the first row is the right argument, and each cell is the result of using the Logical AND both with the left and right arguments:

`&&`	`true`	`false`	`Null`
`true`	`true`	`false`	`Null`
`false`	`false`	`false`	`false`
`Null`	`Null`	`false`	`Null`

Logical OR, ||

In the truth table, the first column is the left argument, the first row is the right argument, and each cell is the result of using the logical OR with both the left and right arguments:

`||`	`true`	`false`	`Null`
`true`	`true`	`true`	`true`
`false`	`true`	`false`	`Null`
`Null`	`true`	`Null`	`Null`

Examples

- `! (true == true)`, the result is `false`

- `(true == true) && (true == false)`, the result is `false`
- `(true == true) || (true == false)`, the result is `true`

Comparison operators

JsonPath implements comparison operators for values:

- Equality, `==`
- Inequality, `!=` and `<>`
- Less than and less than or equal to, `<` and `=`
- Greater than and greater than or equal to, `>` and `>=`

All comparison operators return a Boolean value. Both operator arguments support multiple values.

If an error occurs when calculating the operator arguments, it returns `null`. In this case, the JsonPath query execution continues.

The arrays of each of the arguments are automatically unpacked. After that, for each pair where the first element is taken from the sequence of the left argument and the second one from the sequence of the right argument:

1. The elements of the pair are compared
2. If an error occurs during the comparison, the `ERROR` flag is set.
3. If the comparison result is true, the flag set is `FOUND`
4. If either the `ERROR` or `FOUND` flag is set and the query is executed in `lax` mode, no more pairs are analyzed.

If the pair analysis results in:

1. The `ERROR` flag is set, the operator returns `null`
2. The `FOUND` flag is set, the operator returns `true`
3. Otherwise, it returns `false`

We can say that this algorithm considers all pairs from the Cartesian product of the left and right arguments, trying to find the pair whose comparison returns true.

Elements in a pair are compared according to the following rules:

1. If the left or right argument is an array or object, the comparison fails.
2. `null == null` returns true
3. In all other cases, if one of the arguments is `null`, false is returned.
4. If the left and right arguments are of different types, the comparison fails.
5. Strings are compared byte by byte.
6. `true` is considered greater than `false`
7. Numbers are compared with the accuracy of `1e-20`

Example

Let's take a JSON document as an example

```
{
    "left": [1, 2],
    "right": [4, "Inaros"]
}
```

and analyze the steps for executing the `lax $.left < $.right` query:

1. Auto unpacking of arrays in the left and right arguments. As a result, the left argument is the sequence `1, 2` and the right argument is `4, "Iranos"`
2. Let's take the pair `(1, 4)`. The comparison is successful as `1 < 4` is true. Set the flag `FOUND`
3. Since the query is executed in `lax` mode and the `FOUND` flag is set, we aren't analyzing any more pairs.
4. Since we have the `FOUND` flag set, the operator returns true.

Let's take the same query in a different execution mode: `strict $.left < $.right`:

1. Auto unpacking of arrays in the left and right arguments. As a result, the left argument is the sequence `1, 2` and the right argument is `4, "Iranos"`
2. Let's take the pair `(1, 4)`. The comparison is successful as `1 < 4` is true. Set the flag `FOUND`
3. Let's take the pair `(2, 4)`. The comparison is successful as `2 < 4` is true. Set the flag `FOUND`
4. Let's take the pair `(1, "Iranos")`. The comparison fails as a number can't be compared with a string. Set the flag `ERROR`
5. Let's take the pair `(2, "Iranos")`. The comparison fails as a number can't be compared with a string. Set the flag `ERROR`
6. Since we have the `ERROR` flag set, the operator returns `null`

Predicates

JsonPath supports predicates which are expressions that return a Boolean value and check a certain condition. You can use them, for example, in filters.

`like_regex`

The `like_regex` predicate lets you check if a string matches a regular expression. The syntax of regular expressions is the same as in Hyperscan UDF and REGEXP.

Syntax

```
<expression> like_regex <regexp string> [flag <flag string>]
```

Where:

1. `<expression>` is a JsonPath expression with strings to be checked for matching the regular expression.
2. `<regexp string>` is a string with the regular expression.
3. `flag <flag string>` is an optional section where `<flag string>` is a string with regular expression execution flags.

Supported flags:

- `i` : Disable the case sensitivity.

Execution

Before the check, the input sequence arrays are automatically unpacked.

After that, for each element of the input sequence:

1. A check is made to find out if a string matches a regular expression.
2. If the element isn't a string, the `ERROR` flag is set.
3. If the check result is true, the `FOUND` flag is set.
4. If either the `ERROR` or `FOUND` flag is set and the query is executed in `lax` mode, no more pairs are analyzed.

If the pair analysis results in:

1. Setting the `ERROR` flag, the predicate returns `null`
2. Setting the `FOUND` flag, the predicate returns `true`
3. Otherwise, the predicate returns `false`

Examples

1. `"123456" like_regex "^[0-9]+$"` returns `true`
2. `"123abcd456" like_regex "^[0-9]+$"` returns `false`
3. `"Naomi Nagata" like_regex "nag"` returns `false`
4. `"Naomi Nagata" like_regex "nag" flag "i"` returns `true`

`starts with`

The `starts with` predicate lets you check if one string is a prefix of another.

Syntax

```
<string expression> starts with <prefix expression>
```

Where:

1. `<string expression>` is a JsonPath expression with the string to check.
2. `<prefix expression>` is a JsonPath expression with a prefix string.

This means that the predicate will check that the `<string expression>` starts with the `<prefix expression>` string.

Execution

The first argument of the predicate must be a single string.

The second argument of the predicate must be a sequence of (possibly, multiple) strings.

For each element in a sequence of prefix strings:

1. A check is made for whether "an element is a prefix of an input string"
2. If the element isn't a string, the `ERROR` flag is set.
3. If the check result is true, the `FOUND` flag is set.
4. If either the `ERROR` or `FOUND` flag is set and the query is executed in `lax` mode, no more pairs are analyzed.

If the pair analysis results in:

1. Setting the `ERROR` flag, the predicate returns `null`
2. Setting the `FOUND` flag, the predicate returns `true`
3. Otherwise, the predicate returns `false`

Examples

1. `"James Holden" starts with "James"` returns `true`
2. `"James Holden" starts with "Amos"` returns `false`

`exists`

The `exists` predicate lets you check whether a JsonPath expression returns at least one element.

Syntax

```
exists (<expression>)
```

Where `<expression>` is the JsonPath expression to be checked. Parentheses around the expression are required.

Execution

1. The passed JsonPath expression is executed
2. If an error occurs, the predicate returns `null`
3. If an empty sequence is obtained as a result of the execution, the predicate returns `false`
4. Otherwise, the predicate returns `true`

Examples

Let's take a JSON document:

```
{
    "profile": {
        "name": "Josephus",
        "surname": "Miller"
    }
}
```

1. `exists ($.profile.name)` returns `true`
2. `exists ($.friends.profile.name)` returns `false`
3. `strict exists ($.friends.profile.name)` returns `null`, because accessing non-existent object keys in `strict` mode is an error.

`is unknown`

The `is unknown` predicate lets you check if a Boolean value is `null`.

Syntax

```
(<expression>) is unknown
```

Where `<expression>` is the JsonPath expression to be checked. Only expressions that return a Boolean value are allowed. Parentheses around the expression are required.

Execution

1. If the passed expression returns `null`, the predicate returns `true`
2. Otherwise, the predicate returns `false`

Examples

1. `(1 == 2) is unknown` returns `false`. The `1 == 2` expression returned `false`, which is not `null`
2. `(1 == "string") is unknown` returns `true`. The `1 == "string"` expression returned `null`, because strings and numbers can't be compared in JsonPath.

Filters

JsonPath lets you filter values obtained during query execution.

An expression in a filter must return a Boolean value.
Before filtering, the input sequence arrays are automatically unpacked.

For each element of the input sequence:

1. The value of the current filtered `@` object becomes equal to the current element of the input sequence.
2. Executing the expression in the filter
3. If an error occurs during the expression execution, the current element of the input sequence is skipped.
4. If the expression execution result is the only `true` value, the current element is added to the filter result.

Example

Suppose we have a JSON document describing the user's friends

```
{
    "friends": [
        {
            "name": "James Holden",
            "age": 35,
            "money": 500
        },
        {
            "name": "Naomi Nagata",
            "age": 30,
            "money": 345
        }
    ]
}
```

and we want to get a list of friends who are over 32 years old using a JsonPath query. To do this, you can write the following query:

```
$.friends ? (@.age > 32)
```

Let's analyze the query in parts:

- `$.friends` accesses the `friends` array in the JSON document.
- `? ( ... )` is the filter syntax. An expression inside the parentheses is called a predicate.
- `` accesses the currently filtered object. In our example, it's the object describing a friend of the user.
- `.age` accesses the `age` field of the currently filtered object.
- `.age > 32` compares the `age` field with the value 32. As a result of the query, only the values for which this predicate returned true remain.

The query only returns the first friend from the array of user's friends.

Like many other JsonPath operators, filters can be arranged in chains. Let's take a more complex query that selects the names of friends who are older than 20 and have less than 400 currency units:

```
$.friends ? (@.age > 20) ? (@.money < 400) . name
```

Let's analyze the query in parts:

- `$.friends` accesses the `friends` array in the JSON document.
- `? (@.age > 20)` is the first filter. Since all friends are over 20, it just returns all the elements of the `friends` array.
- `? (@.money < 400)` is the second filter. It only returns the second element of the `friends` array, since only its `money` field value is less than 400.
- `.name` accesses the `name` field of filtered objects.

The query returns a sequence of a single element: `"Naomi Nagata"`.

In practice, it's recommended to combine multiple filters into one if possible. The above query is equivalent to `$.friends ? (@.age > 20 && @.money < 400) . name`.

Methods

JsonPath supports methods that are functions converting one sequence of values to another. The syntax for calling a method is similar to accessing the object key:

```
$.friends.size()
```

Just like in the case of accessing object keys, method calls can be arranged in chains:

```
$.numbers.double().floor()
```

`type`

The `type` method returns a string with the type of the passed value.

For each element of the input sequence, the method adds this string to the output sequence according to the table below:

Value type	String with type
Null	`"null"`
Boolean value	`"boolean"`
Number	`"number"`
String	`"string"`
Array	`"array"`
Object	`"object"`

Examples

1. `"Naomi".type()` returns `"string"`
2. `false.type()` returns `"boolean"`

`size`

The `size` method returns the size of the array.

For each element of the input sequence, the method adds the following to the output sequence:

1. The size of the array if the element type is an array.
2. For all other types (including objects), it adds `1`

Examples

Let's take a JSON document:

```
{
    "array": [1, 2, 3],
    "object": {
        "a": 1,
        "b": 2
    },
    "scalar": "string"
}
```

And queries to it:

1. `$.array.size()` returns `3`
2. `$.object.size()` returns `1`
3. `$.scalar.size()` returns `1`

`Double`

The `double` method converts strings to numbers.

Before its execution, the input sequence arrays are automatically unpacked.

All elements in the input sequence must be strings that contain decimal numbers. It's allowed to specify the fractional part and exponent.

Examples

1. `"125".double()` returns `125`
2. `"125.456".double()` returns `125.456`
3. `"125.456e-3".double()` returns `0.125456`

`ceiling`

The `ceiling` method rounds up a number.

Before its execution, the input sequence arrays are automatically unpacked.

All elements in the input sequence must be numbers.

Examples

1. `(1.3).ceiling()` returns `2`
2. `(1.8).ceiling()` returns `2`
3. `(1.5).ceiling()` returns `2`
4. `(1.0).ceiling()` returns `1`

`floor`

The `floor` method rounds down a number.

Before its execution, the input sequence arrays are automatically unpacked.

All elements in the input sequence must be numbers.

Examples

1. `(1.3).floor()` returns `1`
2. `(1.8).floor()` returns `1`
3. `(1.5).floor()` returns `1`
4. `(1.0).floor()` returns `1`

`abs`

The `abs` method calculates the absolute value of a number (removes the sign).

Before its execution, the input sequence arrays are automatically unpacked.

All elements in the input sequence must be numbers.

Examples

1. `(0.0).abs()` returns `0`
2. `(1.0).abs()` returns `1`
3. `(-1.0).abs()` returns `1`

`keyvalue`

The `keyvalue` method converts an object to a sequence of key-value pairs.

Before its execution, the input sequence arrays are automatically unpacked.

All elements in the input sequence must be objects.

For each element of the input sequence:

1. Each key-value pair in the element is analyzed.
2. For each key-value pair, an object is generated with the keys `name` and `value`.
3. `name` stores a string with the name of the key from the pair.
4. `value` stores the value from the pair.
5. All objects for this element are added to the output sequence.

Examples

Let's take a JSON document:

```
{
    "name": "Chrisjen",
    "surname": "Avasarala",
    "age": 70
}
```

The `$.keyvalue()` query returns the following sequence for it:

```
{
    "name": "age",
    "value": 70
},
{
    "name": "name",
    "value": "Chrisjen"
},
{
    "name": "surname",
```

```
    "value": "Avasarala"
}
```

Variables

Functions using JsonPath can pass values into a query. They are called variables. To access a variable, write the `$` character and the variable name: `$variable`.

Example

Let the `planet` variable be equal to

```
{
    "name": "Mars",
    "gravity": 0.376
}
```

Then the `strict $planet.name` query returns `"Mars"`.

Unlike many programming languages, JsonPath doesn't support creating new variables or modifying existing ones.

Common arguments

All functions for JSON accept:

1. A JSON value (can be an arbitrary `Json` or `Json?` expression)
2. A JsonPath query (must be explicitly specified with a string literal)
3. **(Optional)** `PASSING` section

PASSING section

Lets you pass values to a JsonPath query as variables.

Syntax

```
PASSING
    <expression 1> AS <variable name 1>,
    <expression 2> AS <variable name 2>,
    ...
```

`<expression>` can have the following types:

- Numbers, `Date`, `DateTime`, and `Timestamp` (a `CAST` into `Double` will be made before passing a value to JsonPath)
- `Utf8`, `Bool`, and `Json`

You can set a `<variable name>` in several ways:

- As an SQL name like `variable`
- In quotes, for example, `"variable"`

Example

```
JSON_VALUE(
    $json,
    "$.timestamp - $Now + $Hour"
    PASSING
        24 * 60 as Hour,
        CurrentUtcTimestamp() as "Now"
)
```

JSON_EXISTS

The `JSON_EXISTS` function checks if a JSON value meets the specified JsonPath.

Syntax

```
JSON_EXISTS(
    <JSON expression>,
    <JsonPath query>,
    [<PASSING clause>]
    [{TRUE | FALSE | UNKNOWN | ERROR} ON ERROR]
)
```

Return value: `Bool?`

Default value: If the `ON ERROR` section isn't specified, the used section is `FALSE ON ERROR`

Behavior

1. If `<JSON expression>` is `NULL` or an empty `Json?`, it returns an empty `Bool?`
2. If an error occurs during JsonPath execution, the returned value depends on the `ON ERROR` section:
   - `TRUE`: Return `True`
   - `FALSE`: Return `False`
   - `UNKNOWN`: Return an empty `Bool?`
   - `ERROR`: Abort the entire query
3. If the result of JsonPath execution is one or more values, the return value is `True`.

4. Otherwise, `False` is returned.

Examples

```
$json = CAST(@@{
    "title": "Rocinante",
    "crew": [
        "James Holden",
        "Naomi Nagata",
        "Alex Kamai",
        "Amos Burton"
    ]
}@@ as Json);

SELECT
    JSON_EXISTS($json, "$.title"), -- True
    JSON_EXISTS($json, "$.crew[*]"), -- True
    JSON_EXISTS($json, "$.nonexistent"); -- False, as JsonPath returns an empty result

SELECT
    -- JsonPath error, False is returned because the default section used is FALSE ON ERROR
    JSON_EXISTS($json, "strict $.nonexistent");

SELECT
    -- JsonPath error, the entire YQL query fails.
    JSON_EXISTS($json, "strict $.nonexistent" ERROR ON ERROR);
```

JSON_VALUE

The `JSON_VALUE` function retrieves a scalar value from JSON (anything that isn't an array or object).

Syntax

```
JSON_VALUE(
    <JSON expression>,
    <JsonPath query>,
    [<PASSING clause>]
    [RETURNING <type>]
    [{ERROR | NULL | DEFAULT <expr>} ON EMPTY]
    [{ERROR | NULL | DEFAULT <expr>} ON ERROR]
)
```

Return value: `<type>?`

Default values:

1. If the `ON EMPTY` section isn't specified, the section used is `NULL ON EMPTY`
2. If the `ON ERROR` section isn't specified, the section used is `NULL ON ERROR`
3. If the `RETURNING` section isn't specified, then for `<type>`, the type used is `Utf8`

Behavior:

1. If `<JSON expression>` is `NULL` or an empty `Json?`, it returns an empty `<type>?`
2. If an error occurs, the returned value depends on the `ON ERROR` section:
   - `NULL`: Return an empty `<type>?`
   - `ERROR`: Abort the entire query
   - `DEFAULT <expr>`: Return `<expr>` after running the `CAST` function to convert the data type to `<type>?`. If the `CAST` fails, the entire query fails, too.
3. If the JsonPath execution result is empty, the returned value depends on the `ON EMPTY` section:
   - `NULL`: Return an empty `<type>?`
   - `ERROR`: Abort the entire query
   - `DEFAULT <expr>`: Return `<expr>` after running the `CAST` function to convert the data type to `<type>?`. If the `CAST` fails, the behavior matches the `ON ERROR` section.
4. If the result of JsonPath execution is a single value, then:
   - If the `RETURNING` section isn't specified, the value is converted to `Utf8`.
   - Otherwise, the `CAST` function is run to convert the value to `<type>`. If the `CAST` fails, the behavior matches the `ON ERROR` section. In this case, the value from JSON must match the `<type>` type.
5. Return the result

Correlation between JSON and YQL types:

- JSON Number: Numeric types, `Date`, `DateTime`, and `Timestamp`
- JSON Bool: `Bool`
- JSON String: `Utf8` and `String`

Errors executing `JSON_VALUE` are as follows:

- Errors evaluating JsonPath
- The result of JsonPath execution is a number of values or a non-scalar value.
- The type of value returned by JSON doesn't match the expected one.

`The RETURNING` section supports such types as numbers, `Date`, `DateTime`, `Timestamp`, `Utf8`, `String`, and `Bool`.

Examples

```
$json = CAST(@@{
    "friends": [
        {
            "name": "James Holden",
            "age": 35
        },
        {
            "name": "Naomi Nagata",
            "age": 30
        }
    ]
}@@ as Json);

SELECT
    JSON_VALUE($json, "$.friends[0].age"), -- "35" (type Utf8?)
    JSON_VALUE($json, "$.friends[0].age" RETURNING Uint64), -- 35 (type Uint64?)
    JSON_VALUE($json, "$.friends[0].age" RETURNING Utf8); -- an empty Utf8? due to an error. The JSON's Numbe
r type doesn't match the string Utf8 type.

SELECT
    -- "empty" (type String?)
    JSON_VALUE(
        $json,
        "$.friends[50].name"
        RETURNING String
        DEFAULT "empty" ON EMPTY
    );

SELECT
    -- 20 (type Uint64?). The result of JsonPath execution is empty, but the
    -- default value from the ON EMPTY section can't be cast to Uint64.
    -- That's why the value from ON ERROR is used.
    JSON_VALUE(
        $json,
        "$.friends[50].age"
        RETURNING Uint64
        DEFAULT -1 ON EMPTY
        DEFAULT 20 ON ERROR
    );
```

JSON_QUERY

The `JSON_QUERY` function lets you retrieve arrays and objects from JSON.

Syntax

```
JSON_QUERY(
    <JSON expression>,
    <JsonPath query>,
    [<PASSING clause>]
    [WITHOUT [ARRAY] | WITH [CONDITIONAL | UNCONDITIONAL] [ARRAY] WRAPPER]
    [{ERROR | NULL | EMPTY ARRAY | EMPTY OBJECT} ON EMPTY]
    [{ERROR | NULL | EMPTY ARRAY | EMPTY OBJECT} ON ERROR]
)
```

Return value: `Json?`

Default values:

1. If the `ON EMPTY` section isn't specified, the section used is `NULL ON EMPTY`
2. If the `ON ERROR` section isn't specified, the section used is `NULL ON ERROR`
3. If the `WRAPPER` section isn't specified, the section used is `WITHOUT WRAPPER`
4. If the `WITH WRAPPER` section is specified but `CONDITIONAL` or `UNCONDITIONAL` is omitted, then the section used is `UNCONDITIONAL`

Behavior:

> **Note**
>
> You can't specify the `WITH ... WRAPPER` and `ON EMPTY` sections at the same time.

1. If `<JSON expression>` is `NULL` or an empty `Json?`, it returns an empty `Json?`
2. If the `WRAPPER` section is specified, then:
   - `WITHOUT WRAPPER` or `WITHOUT ARRAY WRAPPER`: Don't convert the result of JsonPath execution in any way.
   - `WITH UNCONDITIONAL WRAPPER` or `WITH UNCONDITIONAL ARRAY WRAPPER`: Wrap the result of JsonPath execution in an array.
   - `WITH CONDITIONAL WRAPPER` or `WITH CONDITIONAL ARRAY WRAPPER`: Wrap the result of JsonPath execution in an array if it isn't the only array or object.
3. If the JsonPath execution result is empty, the returned value depends on the `ON EMPTY` section:
   - `NULL`: Return an empty `Json?`
   - `ERROR`: Abort the entire query
   - `EMPTY ARRAY`: Return an empty JSON array, `[]`

    - `EMPTY OBJECT`: Return an empty JSON object, `{}`
4. If an error occurs, the returned value depends on the `ON ERROR` section:
    - `NULL`: Return an empty `Json?`
    - `ERROR`: Abort the entire query
    - `EMPTY ARRAY`: Return an empty JSON array, `[]`
    - `EMPTY OBJECT`: Return an empty JSON object, `{}`
5. Return the result

Errors running a `JSON_QUERY`:

- Errors evaluating JsonPath
- The result of JsonPath execution is a number of values (even after using the `WRAPPER` section) or a scalar value.

Examples

```
$json = CAST(@@{
    "friends": [
        {
            "name": "James Holden",
            "age": 35
        },
        {
            "name": "Naomi Nagata",
            "age": 30
        }
    ]
}@@ as Json);

SELECT
    JSON_QUERY($json, "$.friends[0]"); -- {"name": "James Holden", "age": 35}

SELECT
    JSON_QUERY($json, "$.friends.name" WITH UNCONDITIONAL WRAPPER); -- ["James Holden", "Naomi Nagata"]

SELECT
    JSON_QUERY($json, "$.friends[0]" WITH CONDITIONAL WRAPPER), -- {"name": "James Holden", "age": 35}
    JSON_QUERY($json, "$.friends.name" WITH CONDITIONAL WRAPPER); -- ["James Holden", "Naomi Nagata"]
```

See also

- Accessing values inside JSON with YQL
- Modifying JSON with YQL

Functions of built-in C++ libraries

Many application functions that on the one hand are too specific to become part of the YQL core, and on the other hand might be useful to a wide range of users, are available through built-in C++ libraries.

- DateTime
- Digest
- Histogram
- Hyperscan
- Ip
- Knn
- Math
- Pcre
- Pire
- Re2
- Roaring
- String
- Unicode
- Url
- Yson

DateTime

The DateTime module provides a set of functions for working with dates and times in YDB. These functions support all available timestamp and interval data types with both basic range of valid values and extended range. Additionally, these functions use their own internal representation of values, where each value is split into separate components. This representation is described below.

The simplest operations with timestamps and intervals can be performed without using the DateTime module. For example, conversion between strings and timestamps according to ISO 8601 is available through `CAST`, timestamp precision can also be changed using `CAST`, and arithmetic operators work between timestamps and intervals. The DateTime module is intended for tasks where these basic operations are insufficient and more complex transformations are required.

Internal representation format

Most functions in the DateTime module operate not on original timestamps and intervals, but on an internal representation based on a special resource data type. This represents a pointer to a data structure in memory where time components are stored separately. Working with resources happens exclusively through DateTime module functions. The typical workflow with this module (see examples at the end of the article) is as follows:

1. Call the `Split` function to get the internal representation from the original timestamp, or `Parse...` to get it from a string.
2. If necessary, call functions that modify this internal representation as needed, for example `Shift...`, `StartOf...` or `Update...`.
3. Call one of the functions to create a new value based on the internal representation, which can then be saved to tables or returned from a query:
   - `Make...` to create a new timestamp from the internal representation (not necessarily with the same precision as the original)
   - `Format` to get a string representation
   - `Get...` to get individual components

YDB supports two ranges of valid timestamp values: basic and extended. In the DateTime module, these correspond to two separate internal representations: `Resource<TM>` for basic data types and `Resource<TM64>` for data types with extended range of valid values.

`Resource<TM>` and `Resource<TM64>` contain the following date components:

- Year (for `Resource<TM>` 12 bits, unsigned; for `Resource<TM64>` 19 bits, signed)
- Month (4 bits)
- Day (5 bits)
- Hour (5 bits)
- Minute (6 bits)
- Second (6 bits)
- Microsecond (20 bits)
- TimezoneId (16 bits)
- DayOfYear (9 bits): Day since the beginning of the year
- WeekOfYear (6 bits): Week since the beginning of the year, January 1 is always in week 1
- WeekOfYearIso8601 (6 bits): Week of the year according to ISO 8601 (the first week is the one that includes January 4)
- DayOfWeek (3 bits): Day of the week

If the timezone is not GMT, the components store the local time for the relevant timezone.

Split

Conversion from a YDB primitive type to the internal representation of the DateTime module. Conversion is always successful when the required value is provided as input to the function.

List of functions

- `DateTime::Split(Date/TzDate/Datetime/TzDatetime/Timestamp/TzTimestamp{Flags:AutoMap}) -> Resource<TM>`
- `DateTime::Split(Date32/TzDate32/Datetime64/TzDatetime64/Timestamp64/TzTimestamp64{Flags:AutoMap}) -> Resource<TM64>`

Functions that accept `Resource<TM>` or `Resource<TM64>` as input can be called directly from the primitive date/time type. In this case, an implicit conversion will be made by calling the corresponding `Split` function.

Make...

Creating a YDB primitive type from the internal representation of the DateTime module. Conversion is always successful when the required value is provided as input to the function.

List of functions

- `DateTime::MakeDate(Resource<TM>{Flags:AutoMap}) -> Date`
- `DateTime::MakeDate32(Resource<TM64>{Flags:AutoMap}) -> Date32`
- `DateTime::MakeTzDate32(Resource<TM64>{Flags:AutoMap}) -> TzDate32`
- `DateTime::MakeDatetime(Resource<TM>{Flags:AutoMap}) -> Datetime`
- `DateTime::MakeTzDatetime(Resource<TM>{Flags:AutoMap}) -> TzDatetime`
- `DateTime::MakeDatetime64(Resource<TM64>{Flags:AutoMap}) -> Datetime64`
- `DateTime::MakeTzDatetime64(Resource<TM64>{Flags:AutoMap}) -> TzDatetime64`
- `DateTime::MakeTimestamp(Resource<TM>{Flags:AutoMap}) -> Timestamp`
- `DateTime::MakeTzTimestamp(Resource<TM>{Flags:AutoMap}) -> TzTimestamp`
- `DateTime::MakeTimestamp64(Resource<TM64>{Flags:AutoMap}) -> Timestamp64`
- `DateTime::MakeTzTimestamp64(Resource<TM64>{Flags:AutoMap}) -> TzTimestamp64`

Examples

```
SELECT
    DateTime::MakeTimestamp(DateTime::Split(Datetime("2019-01-01T15:30:00Z"))),
      -- 2019-01-01T15:30:00.000000Z
    DateTime::MakeDate(Datetime("2019-01-01T15:30:00Z")),
      -- 2019-01-01
    DateTime::MakeTimestamp(DateTime::Split(TzDatetime("2019-01-01T00:00:00,Europe/Moscow"))),
      -- 2018-12-31T21:00:00Z (conversion to UTC)
    DateTime::MakeDate(TzDatetime("2019-01-01T12:00:00,GMT"))
      -- 2019-01-01 (Datetime -> Date with implicit Split)
```

Get...

Extracting a component from the internal representation.

List of functions

- `DateTime::GetYear(Resource<TM>{Flags:AutoMap}) -> Uint16`
- `DateTime::GetYear(Resource<TM64>{Flags:AutoMap}) -> Int32`
- `DateTime::GetDayOfYear(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> Uint16`
- `DateTime::GetMonth(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetMonthName(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> String`
- `DateTime::GetWeekOfYear(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetWeekOfYearIso8601(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetDayOfMonth(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetDayOfWeek(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetDayOfWeekName(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> String`
- `DateTime::GetHour(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetMinute(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetSecond(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetMillisecondOfSecond(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> Uint32`
- `DateTime::GetMicrosecondOfSecond(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> Uint32`
- `DateTime::GetTimezoneId(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> Uint16`
- `DateTime::GetTimezoneName(Resource<TM>/Resource<TM64>{Flags:AutoMap}) -> String`

Examples

```
$tm = DateTime::Split(TzDatetime("2019-01-09T00:00:00,Europe/Moscow"));

SELECT
    DateTime::GetDayOfMonth($tm) as Day, -- 9
    DateTime::GetMonthName($tm) as Month, -- "January"
    DateTime::GetYear($tm) as Year, -- 2019
    DateTime::GetTimezoneName($tm) as TzName, -- "Europe/Moscow"
    DateTime::GetDayOfWeekName($tm) as WeekDay; -- "Wednesday"
```

Update

Updating one or more components in the internal representation. Returns either an updated copy or `NULL`, if an update produces an invalid date or other inconsistencies.

List of functions

```
DateTime::Update(Resource<TM>{Flags:AutoMap}, [ Year:Uint16?, Month:Uint8?, Day:Uint8?, Hour:Uint8?, Minute:U
int8?, Second:Uint8?, Microsecond:Uint32?, Timezone:String? ]) -> Resource<TM>?
DateTime::Update(Resource<TM64>{Flags:AutoMap}, [ Year:Int32?, Month:Uint8?, Day:Uint8?, Hour:Uint8?, Minute:
Uint8?, Second:Uint8?, Microsecond:Uint32?, Timezone:String? ]) -> Resource<TM64>?
```

Examples

```
$tm = DateTime::Split(Timestamp("2019-01-01T01:02:03.456789Z"));

SELECT
    DateTime::MakeDate(DateTime::Update($tm, 2012)), -- 2012-01-01
    DateTime::MakeDate(DateTime::Update($tm, 2000, 6, 6)), -- 2000-06-06
    DateTime::MakeDate(DateTime::Update($tm, NULL, 2, 30)), -- NULL (February 30)
    DateTime::MakeDatetime(DateTime::Update($tm, NULL, NULL, 31)), -- 2019-01-31T01:02:03Z
    DateTime::MakeDatetime(DateTime::Update($tm, 15 as Hour, 30 as Minute)), -- 2019-01-01T15:30:03Z
    DateTime::MakeTimestamp(DateTime::Update($tm, 999999 as Microsecond)), -- 2019-01-01T01:02:03.999999Z
    DateTime::MakeTimestamp(DateTime::Update($tm, "Europe/Moscow" as Timezone)), -- 2018-12-31T22:02:03.45678
9Z (conversion to UTC)
    DateTime::MakeTzTimestamp(DateTime::Update($tm, "Europe/Moscow" as Timezone)); -- 2019-01-01T01:02:03.456
789,Europe/Moscow
```

From...

Getting a Timestamp from the number of seconds/milliseconds/microseconds since the UTC epoch. When the Timestamp limits are exceeded, NULL is returned.

List of functions

- `DateTime::FromSeconds(Uint32{Flags:AutoMap}) -> Timestamp`
- `DateTime::FromSeconds64(Int64{Flags:AutoMap}) -> Timestamp64`
- `DateTime::FromMilliseconds(Uint64{Flags:AutoMap}) -> Timestamp`
- `DateTime::FromMilliseconds64(Int64{Flags:AutoMap}) -> Timestamp64`
- `DateTime::FromMicroseconds(Uint64{Flags:AutoMap}) -> Timestamp`
- `DateTime::FromMicroseconds64(Int64{Flags:AutoMap}) -> Timestamp64`

To...

Getting a number of seconds/milliseconds/microseconds since the UTC Epoch from a primitive type.

List of functions

- `DateTime::ToSeconds(Date/Datetime/Timestamp/TzDate/TzDatetime/TzTimestamp{Flags:AutoMap}) -> Uint32`
- `DateTime::ToSeconds(Date32/Datetime64/Timestamp64/TzDate32/TzDatetime64/TzTimestamp64{Flags:AutoMap}) -> Int64`
- `DateTime::ToMilliseconds(Date/Datetime/Timestamp/TzDate/TzDatetime/TzTimestamp{Flags:AutoMap}) -> Uint64`
- `DateTime::ToMilliseconds(Date32/Datetime64/Timestamp64/TzDate32/TzDatetime64/TzTimestamp64{Flags:AutoMap}) -> Int64`
- `DateTime::ToMicroseconds(Date/Datetime/Timestamp/TzDate/TzDatetime/TzTimestamp{Flags:AutoMap}) -> Uint64`
- `DateTime::ToMicroseconds(Date32/Datetime64/Timestamp64/TzDate32/TzDatetime64/TzTimestamp64{Flags:AutoMap}) -> Int64`

Examples

```
SELECT
    DateTime::FromSeconds(1546304523), -- 2019-01-01T01:02:03.000000Z
    DateTime::ToMicroseconds(Timestamp("2019-01-01T01:02:03.456789Z")); -- 1546304523456789
```

Interval...

Conversions between `Interval` and various time units.

List of functions

- `DateTime::ToDays(Interval{Flags:AutoMap}) -> Int32`
- `DateTime::ToDays(Interval64{Flags:AutoMap}) -> Int32`
- `DateTime::ToHours(Interval{Flags:AutoMap}) -> Int32`
- `DateTime::ToHours(Interval64{Flags:AutoMap}) -> Int64`
- `DateTime::ToMinutes(Interval{Flags:AutoMap}) -> Int32`
- `DateTime::ToMinutes(Interval64{Flags:AutoMap}) -> Int64`
- `DateTime::ToSeconds(Interval{Flags:AutoMap}) -> Int64`
- `DateTime::ToSeconds(Interval64{Flags:AutoMap}) -> Int64`
- `DateTime::ToMilliseconds(Interval{Flags:AutoMap}) -> Int64`
- `DateTime::ToMilliseconds(Interval64{Flags:AutoMap}) -> Int64`
- `DateTime::ToMicroseconds(Interval{Flags:AutoMap}) -> Int64`
- `DateTime::ToMicroseconds(Interval64{Flags:AutoMap}) -> Int64`
- `DateTime::IntervalFromDays(Int32{Flags:AutoMap}) -> Interval`
- `DateTime::Interval64FromDays(Int32{Flags:AutoMap}) -> Interval64`
- `DateTime::IntervalFromHours(Int32{Flags:AutoMap}) -> Interval`
- `DateTime::Interval64FromHours(Int64{Flags:AutoMap}) -> Interval64`
- `DateTime::IntervalFromMinutes(Int32{Flags:AutoMap}) -> Interval`
- `DateTime::Interval64FromMinutes(Int64{Flags:AutoMap}) -> Interval64`
- `DateTime::IntervalFromSeconds(Int64{Flags:AutoMap}) -> Interval`
- `DateTime::Interval64FromSeconds(Int64{Flags:AutoMap}) -> Interval64`
- `DateTime::IntervalFromMilliseconds(Int64{Flags:AutoMap}) -> Interval`
- `DateTime::Interval64FromMilliseconds(Int64{Flags:AutoMap}) -> Interval64`
- `DateTime::IntervalFromMicroseconds(Int64{Flags:AutoMap}) -> Interval`
- `DateTime::Interval64FromMicroseconds(Int64{Flags:AutoMap}) -> Interval64`

`AddTimezone` doesn't affect the output of `ToSeconds()` in any way, because `ToSeconds()` always returns GMT time.

Interval can also be created from a string literal in the format ISO 8601.

> **Warning**
>
> Currently, only literals for deterministic intervals are supported. All dimensions larger than a week — `M` (month) and `Y` (year) — are not fixed, since their duration is determined only by external context, the calendar.

Examples

```
SELECT
    DateTime::ToDays(Interval("PT3000M")), -- 2
```

```
    DateTime::IntervalFromSeconds(1000000), -- 11 days 13 hours 46 minutes 40 seconds
    DateTime::ToDays(CAST('2018-01-01' as date) - CAST('2017-12-31' as date)); --1
```

StartOf... / EndOf... / TimeOfDay

Get the start (end) of the period including the date/time. If the result is invalid, NULL is returned. If the timezone is different from GMT, then the period start (end) is in the specified time zone.

List of functions

- `DateTime::StartOfYear(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::StartOfYear(Resource<TM64>{Flags:AutoMap}) -> Resource<TM64>?`
- `DateTime::EndOfYear(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::EndOfYear(Resource<TM64>{Flags:AutoMap}) -> Resource<TM64>?`
- `DateTime::StartOfQuarter(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::StartOfQuarter(Resource<TM64>{Flags:AutoMap}) -> Resource<TM64>?`
- `DateTime::EndOfQuarter(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::EndOfQuarter(Resource<TM64>{Flags:AutoMap}) -> Resource<TM64>?`
- `DateTime::StartOfMonth(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::StartOfMonth(Resource<TM64>{Flags:AutoMap}) -> Resource<TM64>?`
- `DateTime::EndOfMonth(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::EndOfMonth(Resource<TM64>{Flags:AutoMap}) -> Resource<TM64>?`
- `DateTime::StartOfWeek(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::StartOfWeek(Resource<TM64>{Flags:AutoMap}) -> Resource<TM64>?`
- `DateTime::EndOfWeek(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::EndOfWeek(Resource<TM64>{Flags:AutoMap}) -> Resource<TM64>?`
- `DateTime::StartOfDay(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::StartOfDay(Resource<TM64>{Flags:AutoMap}) -> Resource<TM64>?`
- `DateTime::EndOfDay(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::EndOfDay(Resource<TM64>{Flags:AutoMap}) -> Resource<TM64>?`
- `DateTime::StartOf(Resource<TM>{Flags:AutoMap}, Interval{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::StartOf(Resource<TM64>{Flags:AutoMap}, Interval64{Flags:AutoMap}) -> Resource<TM64>?`
- `DateTime::EndOf(Resource<TM>{Flags:AutoMap}, Interval{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::EndOf(Resource<TM64>{Flags:AutoMap}, Interval64{Flags:AutoMap}) -> Resource<TM64>?`

The `StartOf` / `EndOf` functions are intended for grouping by an arbitrary period within a day. The result differs from the input value only by time components. A period exceeding one day is treated as a day (an equivalent of `StartOfDay` / `EndOfDay`). If a day doesn't include an integer number of periods, the number is rounded to the nearest time from the beginning of the day that is a multiple of the specified period. When the interval is zero, the output is same as the input. A negative interval is treated as a positive one.

The `EndOf...` functions are intended for obtaining the latest moment in the same period of time as the specified one.

The functions treat periods longer than one day in a different manner than the same-name functions in the old library. The time components are always reset to zero (this makes sense, because these functions are mainly used for grouping by the period). You can also specify a time period within a day:

- `DateTime::TimeOfDay(Resource<TM>{Flags:AutoMap}) -> Interval`
- `DateTime::TimeOfDay(Resource<TM64>{Flags:AutoMap}) -> Interval64`

Examples

```
SELECT
    DateTime::MakeDate(DateTime::StartOfYear(Date("2019-06-06"))),
      -- 2019-01-01 (implicit Split here and below)
    DateTime::MakeDatetime(DateTime::StartOfQuarter(Datetime("2019-06-06T01:02:03Z"))),
      -- 2019-04-01T00:00:00Z (time components are reset to zero)
    DateTime::MakeDate(DateTime::StartOfMonth(Timestamp("2019-06-06T01:02:03.456789Z"))),
      -- 2019-06-01
    DateTime::MakeDate(DateTime::StartOfWeek(Date("1970-01-01"))),
      -- NULL (the beginning of the epoch is Thursday, the beginning of the week is 1969-12-29 that is beyond
the limits)
    DateTime::MakeTimestamp(DateTime::StartOfWeek(Date("2019-01-01"))),
      -- 2018-12-31T00:00:00Z
    DateTime::MakeDatetime(DateTime::StartOfDay(Datetime("2019-06-06T01:02:03Z"))),
      -- 2019-06-06T00:00:00Z
    DateTime::MakeTzDatetime(DateTime::StartOfDay(TzDatetime("1970-01-01T05:00:00,Europe/Moscow"))),
      -- NULL (beyond the epoch in GMT)
    DateTime::MakeTzTimestamp(DateTime::StartOfDay(TzTimestamp("1970-01-02T05:00:00.000000,Europe/Moscow"))),
      -- 1970-01-02T00:00:00,Europe/Moscow (the beginning of the day in Moscow)
    DateTime::MakeDatetime(DateTime::StartOf(Datetime("2019-06-06T23:45:00Z"), Interval("PT7H"))),
      -- 2019-06-06T21:00:00Z
    DateTime::MakeDatetime(DateTime::StartOf(Datetime("2019-06-06T23:45:00Z"), Interval("PT20M"))),
      -- 2019-06-06T23:40:00Z
    DateTime::TimeOfDay(Timestamp("2019-02-14T01:02:03.456789Z"));
      -- 1 hour 2 minutes 3 seconds 456789 microseconds
```

Shift...

Add/subtract the specified number of units to/from the component in the internal representation and update the other fields. Returns either an updated copy or NULL, if an update produces an invalid date or other inconsistencies.

List of functions

- `DateTime::ShiftYears(Resource<TM>{Flags:AutoMap}, Int32) -> Resource<TM>?`
- `DateTime::ShiftYears(Resource<TM64>{Flags:AutoMap}, Int32) -> Resource<TM64>?`
- `DateTime::ShiftQuarters(Resource<TM>{Flags:AutoMap}, Int32) -> Resource<TM>?`
- `DateTime::ShiftQuarters(Resource<TM64>{Flags:AutoMap}, Int32) -> Resource<TM64>?`
- `DateTime::ShiftMonths(Resource<TM>{Flags:AutoMap}, Int32) -> Resource<TM>?`
- `DateTime::ShiftMonths(Resource<TM64>{Flags:AutoMap}, Int32) -> Resource<TM64>?`

If the resulting number of the day in the month exceeds the maximum allowed, then the `Day` field will accept the last day of the month without changing the time (see examples).

Examples

```
$tm1 = DateTime::Split(DateTime("2019-01-31T01:01:01Z"));
$tm2 = DateTime::Split(TzDatetime("2049-05-20T12:34:50,Europe/Moscow"));

SELECT
    DateTime::MakeDate(DateTime::ShiftYears($tm1, 10)), -- 2029-01-31T01:01:01
    DateTime::MakeDate(DateTime::ShiftYears($tm2, -10000)), -- NULL (beyond the limits)
    DateTime::MakeDate(DateTime::ShiftQuarters($tm2, 0)), -- 2049-05-20T12:34:50,Europe/Moscow
    DateTime::MakeDate(DateTime::ShiftQuarters($tm1, -3)), -- 2018-04-30T01:01:01
    DateTime::MakeDate(DateTime::ShiftMonths($tm1, 1)), -- 2019-02-28T01:01:01
    DateTime::MakeDate(DateTime::ShiftMonths($tm1, -35)), -- 2016-02-29T01:01:01
```

Format

Get a string representation of a time using an arbitrary formatting string.

List of functions

- `DateTime::Format(String, alwaysWriteFractionalSeconds:Bool?) -> (Resource<TM64>{Flags:AutoMap}) -> String`

A set of specifiers is implemented for the formatting string:

- `%%` : % character
- `%Y` : year, 1–6 digits and sign for BC dates
- `%m` : 2-digit month
- `%d` : 2-digit day
- `%H` : 2-digit hour
- `%M` : 2-digit minutes
- `%S` : 2-digit seconds or `XX.XXXXXX` in the case of non-empty microseconds (only if `alwaysWriteFractionalSeconds` is not set to `True`)
- `%z` : `+hhmm` or `-hhmm`
- `%Z` : IANA name of the timezone
- `%b` : A short three-letter English name of the month (Jan)
- `%B` : A full English name of the month (January)

All other characters in the format string are passed on without changes.

Examples

```
$format = DateTime::Format("%Y-%m-%d %H:%M:%S %Z");

SELECT
    $format(DateTime::Split(TzDatetime("2019-01-01T01:02:03,Europe/Moscow")));
      -- "2019-01-01 01:02:03 Europe/Moscow"
```

Parse/Parse64

Parse a string into an internal representation using an arbitrary formatting string. Default values are used for empty fields. If errors are raised, NULL is returned.

List of functions

- `DateTime::Parse(String) -> (String{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::Parse64(String) -> (String{Flags:AutoMap}) -> Resource<TM64>?`

Implemented specifiers:

- `%%` : the % character
- `%Y` : 4-digit year when using `Parse`, or 1–6 digits and sign for BC dates when using `Parse64`
- `%m` : 2-digit month
- `%d` : 2-digit day
- `%H` : 2-digit hour
- `%M` : 2-digit minutes
- `%S` : Seconds, can also accept microseconds in the formats from `XX` up to `XX.XXXXXX`
- `%Z` : The IANA name of the timezone (e.g., GMT)
- `%b` : A short three-letter case-insensitive English name of the month (e.g., Jan)
- `%B` : A full case-insensitive English name of the month (January)

Examples

```
$parse1 = DateTime::Parse("%H:%M:%S");
$parse2 = DateTime::Parse("%S");
$parse3 = DateTime::Parse("%m/%d/%Y");
$parse4 = DateTime::Parse("%Z");
$parse5 = DateTime::Parse64("%m/%d/%Y");

SELECT
    DateTime::MakeDatetime($parse1("01:02:03")), -- 1970-01-01T01:02:03Z
    DateTime::MakeTimestamp($parse2("12.3456")), -- 1970-01-01T00:00:12.345600Z
    DateTime::MakeTimestamp($parse3("02/30/2000")), -- NULL (Feb 30)
    DateTime::MakeTimestamp($parse4("Canada/Central")), -- 1970-01-01T06:00:00Z (conversion to UTC)
    DateTime::MakeTimestamp64($parse5("02/10/1931")); -- 1931-02-10T00:00:00Z (conversion to UTC)
```

Parse specific formats

For the common formats, wrappers around the corresponding util methods are supported. You can only get TM with components in the UTC timezone.

List of functions

- `DateTime::ParseRfc822(String{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::ParseIso8601(String{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::ParseHttp(String{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::ParseX509(String{Flags:AutoMap}) -> Resource<TM>?`

Examples

```
SELECT
    DateTime::MakeTimestamp(DateTime::ParseRfc822("Fri, 4 Mar 2005 19:34:45 EST")),
      -- 2005-03-05T00:34:45Z
    DateTime::MakeTimestamp(DateTime::ParseIso8601("2009-02-14T02:31:30+0300")),
      -- 2009-02-13T23:31:30Z
    DateTime::MakeTimestamp(DateTime::ParseHttp("Sunday, 06-Nov-94 08:49:37 GMT")),
      -- 1994-11-06T08:49:37Z
    DateTime::MakeTimestamp(DateTime::ParseX509("20091014165533Z"))
      -- 2009-10-14T16:55:33Z
```

Standard scenarios

Conversions between strings and seconds

Converting a string date (in the Moscow timezone) to seconds (in GMT timezone):

```
$datetime_parse = DateTime::Parse("%Y-%m-%d %H:%M:%S");
$datetime_parse_tz = DateTime::Parse("%Y-%m-%d %H:%M:%S %Z");

SELECT
    DateTime::ToSeconds(TzDatetime("2019-09-16T00:00:00,Europe/Moscow")) AS md_us1, -- 1568581200
    DateTime::ToSeconds(DateTime::MakeDatetime($datetime_parse_tz("2019-09-16 00:00:00" || " Europe/Mosco
w"))), -- 1568581200
    DateTime::ToSeconds(DateTime::MakeDatetime(DateTime::Update($datetime_parse("2019-09-16 00:00:00"), "Euro
pe/Moscow" AS Timezone))), -- 1568581200
    -- INCORRECT (Date imports time as GMT, but AddTimezone has no effect on ToSeconds that always returns GM
T time)
    DateTime::ToSeconds(AddTimezone(Date("2019-09-16"), 'Europe/Moscow')) AS md_us2, -- 1568592000
```

Converting a string date (in the Moscow timezone) to seconds (in the Moscow timezone). Since `DateTime::ToSeconds()` exports only GMT, we have to temporarily forget about timezones and work only in GMT (it looks like we temporarily assume that Moscow is in GMT timezone):

```
$date_parse = DateTime::Parse("%Y-%m-%d");
$datetime_parse = DateTime::Parse("%Y-%m-%d %H:%M:%S");
$datetime_parse_tz = DateTime::Parse("%Y-%m-%d %H:%M:%S %Z");

SELECT
    DateTime::ToSeconds(Datetime("2019-09-16T00:00:00Z")) AS md_ms1, -- 1568592000
    DateTime::ToSeconds(Date("2019-09-16")) AS md_ms2, -- 1568592000
    DateTime::ToSeconds(DateTime::MakeDatetime($date_parse("2019-09-16"))) AS md_ms3, -- 1568592000
    DateTime::ToSeconds(DateTime::MakeDatetime($datetime_parse("2019-09-16 00:00:00"))) AS md_ms4, -- 1568592
000
    DateTime::ToSeconds(DateTime::MakeDatetime($datetime_parse_tz("2019-09-16 00:00:00 GMT"))) AS md_ms5, --
1568592000

    -- INCORRECT (imports the time in the Moscow timezone, but RemoveTimezone doesn't affect ToSeconds in any
way)
    DateTime::ToSeconds(RemoveTimezone(TzDatetime("2019-09-16T00:00:00,Europe/Moscow"))) AS md_ms6, -- 156858
1200
    DateTime::ToSeconds(DateTime::MakeDatetime($datetime_parse_tz("2019-09-16 00:00:00 Europe/Moscow"))) AS m
d_ms7 -- 1568581200
```

Converting seconds (in the GMT timezone) to a string date (in the Moscow timezone):

```
$date_format = DateTime::Format("%Y-%m-%d %H:%M:%S %Z");
SELECT
```

```
    $date_format(AddTimezone(DateTime::FromSeconds(1568592000), 'Europe/Moscow')) -- "2019-09-16 03:00:00 Eur
ope/Moscow"
```

Converting seconds (in the Moscow timezone) to a string date (in the Moscow timezone). In this case, the `%Z` timezone is output for reference: usually, it's not needed because it's "GMT" and might mislead you.

```
$date_format = DateTime::Format("%Y-%m-%d %H:%M:%S %Z");
SELECT
    $date_format(DateTime::FromSeconds(1568592000)) -- "2019-09-16 00:00:00 GMT"
```

Converting seconds (in the GMT timezone) to three-letter days of the week (in the Moscow timezone):

```
SELECT
    SUBSTRING(DateTime::GetDayOfWeekName(AddTimezone(DateTime::FromSeconds(1568581200), "Europe/Moscow")), 0,
3) -- "Mon"
```

Date and time formatting

Usually a separate named expression is used to format time, but you can do without it:

```
$date_format = DateTime::Format("%Y-%m-%d %H:%M:%S %Z");

SELECT

   -- A variant with a named expression

   $date_format(AddTimezone(DateTime::FromSeconds(1568592000), 'Europe/Moscow')),

   -- A variant without a named expression

   DateTime::Format("%Y-%m-%d %H:%M:%S %Z")
       (AddTimezone(DateTime::FromSeconds(1568592000), 'Europe/Moscow'))
;
```

Converting types

This way, you can convert only constants:

```
SELECT
    TzDatetime("2019-09-16T00:00:00,Europe/Moscow"), -- 2019-09-16T00:00:00,Europe/Moscow
    Date("2019-09-16") -- 2019-09-16
```

But this way, you can convert a constant, a named expression, or a table field:

```
SELECT
CAST("2019-09-16T00:00:00,Europe/Moscow" AS TzDatetime), -- 2019-09-16T00:00:00,Europe/Moscow
    CAST("2019-09-16" AS Date) -- 2019-09-16
```

Converting time to date

A `CAST` to `Date` or `TzDate` outputs a GMT date for a midnight, local time (for example, for Moscow time 2019-10-22 00:00:00, the date 2019-10-21 is returned). To get a date in the local timezone, you can use DateTime::Format.

```
$x = Datetime("2019-10-21T21:00:00Z");
SELECT
    AddTimezone($x, "Europe/Moscow"), -- 2019-10-22T00:00:00,Europe/Moscow
    CAST($x as TzDate), -- 2019-10-21,GMT
    CAST(AddTimezone($x, "Europe/Moscow") as TzDate), -- 2019-10-21,Europe/Moscow
    CAST(AddTimezone($x, "Europe/Moscow") as Date), -- 2019-10-21
  DateTime::Format("%Y-%m-%d %Z")(AddTimezone($x, "Europe/Moscow")), -- 2019-10-22 Europe/Moscow
```

It's worth mentioning that several `TzDatetime` or `TzTimestamp` values with a positive timezone offset cannot be cast to `TzDate`. Consider the example below:

```
SELECT CAST(TzDatetime("1970-01-01T23:59:59,Europe/Moscow") as TzDate);
/* Fatal: Timestamp 1970-01-01T23:59:59.000000,Europe/Moscow cannot be casted to TzDate */
```

There is no valid value, starting from the Unix epoch, that can represent midnight on 01/01/1970 for the Moscow timezone. As a result, such a cast is considered impossible and generates a runtime error.

However, values with a negative timezone offset return a correct result:

```
SELECT CAST(TzDatetime("1970-01-01T23:59:59,America/Los_Angeles") as TzDate);
/* 1970-01-01,America/Los_Angeles */
```

Daylight saving time

Note that daylight saving time depends on the year:

```
SELECT
    RemoveTimezone(TzDatetime("2019-09-16T10:00:00,Europe/Moscow")) as DST1, -- 2019-09-16T07:00:00Z
```

```
    RemoveTimezone(TzDatetime("2008-12-03T10:00:00,Europe/Moscow")) as DST2, -- 2008-12-03T07:00:00Z
    RemoveTimezone(TzDatetime("2008-07-03T10:00:00,Europe/Moscow")) as DST3, -- 2008-07-03T06:00:00Z (DST)
```

Digest

A set of commonly used hash functions.

List of functions

- `Digest::Crc32c(String{Flags::AutoMap}) -> Uint32`
- `Digest::Crc64(String{Flags::AutoMap}, [Init:Uint64?]) -> Uint64`
- `Digest::Fnv32(String{Flags::AutoMap}, [Init:Uint32?]) -> Uint32`
- `Digest::Fnv64(String{Flags::AutoMap}, [Init:Uint64?]) -> Uint64`
- `Digest::MurMurHash(String{Flags:AutoMap}, [Init:Uint64?]) -> Uint64`
- `Digest::MurMurHash32(String{Flags:AutoMap}, [Init:Uint32?]) -> Uint32`
- `Digest::MurMurHash2A(String{Flags:AutoMap}, [Init:Uint64?]) -> Uint64`
- `Digest::MurMurHash2A32(String{Flags:AutoMap}, [Init:Uint32?]) -> Uint32`
- `Digest::CityHash(String{Flags:AutoMap}, [Init:Uint64?]) -> Uint64`
- `Digest::CityHash128(String{Flags:AutoMap}) -> Tuple<Uint64,Uint64>`
- `Digest::NumericHash(Uint64{Flags:AutoMap}) -> Uint64`
- `Digest::Md5Hex(String{Flags:AutoMap}) -> String`
- `Digest::Md5Raw(String{Flags:AutoMap}) -> String`
- `Digest::Md5HalfMix(String{Flags:AutoMap}) -> Uint64`: MD5 coarsening option (yabs_md5)
- `Digest::Argon2(String{Flags:AutoMap},String{Flags:AutoMap}) -> String`: The second argument is the salt
- `Digest::Blake2B(String{Flags:AutoMap},[String?]) -> String`: The second optional argument is the key
- `Digest::SipHash(Uint64,Uint64,String{Flags:AutoMap}) -> Uint64`
- `Digest::HighwayHash(Uint64,Uint64,Uint64,Uint64,String{Flags:AutoMap}) -> Uint64`
- `Digest::FarmHashFingerprint(Uint64{Flags:AutoMap}) -> Uint64`
- `Digest::FarmHashFingerprint2(Uint64{Flags:AutoMap}, Uint64{Flags:AutoMap}) -> Uint64`
- `Digest::FarmHashFingerprint32(String{Flags:AutoMap}) -> Uint32`
- `Digest::FarmHashFingerprint64(String{Flags:AutoMap}) -> Uint64`
- `Digest::FarmHashFingerprint128(String{Flags:AutoMap}) -> Tuple<Uint64,Uint64>`
- `Digest::SuperFastHash(String{Flags:AutoMap}) -> Uint32`
- `Digest::Sha1(String{Flags:AutoMap}) -> String`
- `Digest::Sha256(String{Flags:AutoMap}) -> String`
- `Digest::IntHash64(Uint64{Flags:AutoMap}) -> Uint64`
- `Digest::XXH3(String{Flags:AutoMap}) -> Uint64`
- `Digest::XXH3_128(String{Flags:AutoMap}) -> Tuple<Uint64,Uint64>`

The functions for the hashes that support the initialization parameter (seed) accept its value in the optional named argument `Init`.

Examples

```
SELECT Digest::Md5Hex("YQL");  -- "1a0c1b56e9d617688ee345da4030da3c"
SELECT Digest::NumericHash(123456789); -- 1734215268924325803
```

Histogram

Set of auxiliary functions for the HISTOGRAM aggregate function. In the signature description below, HistogramStruct refers to the result of the aggregate function `HISTOGRAM`, `LinearHistogram` or `LogarithmicHistogram` being a structure of a certain type.

List of functions

- `Histogram::Print(HistogramStruct{Flags:AutoMap}, Byte?) -> String`
- `Histogram::Normalize(HistogramStruct{Flags:AutoMap}, [Double?]) -> HistogramStruct`: The second argument specifies the desired area of the histogram, 100 by default.
- `Histogram::ToCumulativeDistributionFunction(HistogramStruct{Flags:AutoMap}) -> HistogramStruct`
- `Histogram::GetSumAboveBound(HistogramStruct{Flags:AutoMap}, Double) -> Double`
- `Histogram::GetSumBelowBound(HistogramStruct{Flags:AutoMap}, Double) -> Double`
- `Histogram::GetSumInRange(HistogramStruct{Flags:AutoMap}, Double, Double) -> Double`
- `Histogram::CalcUpperBound(HistogramStruct{Flags:AutoMap}, Double) -> Double`
- `Histogram::CalcLowerBound(HistogramStruct{Flags:AutoMap}, Double) -> Double`
- `Histogram::CalcUpperBoundSafe(HistogramStruct{Flags:AutoMap}, Double) -> Double`
- `Histogram::CalcLowerBoundSafe(HistogramStruct{Flags:AutoMap}, Double) -> Double`

`Histogram::Print` has an optional numeric argument that sets the maximum length of the histogram columns (the length is in characters, since the histogram is rendered in ASCII art). Default: 25. This function is primarily intended for viewing histograms in the console.

Hyperscan

Hyperscan is an opensource library for regular expression matching developed by Intel.

The library includes 4 implementations that use different sets of processor instructions (SSE3, SSE4.2, AVX2, and AVX512), with the needed instruction automatically selected based on the current processor.

By default, all functions work in the single-byte mode. However, if the regular expression is a valid UTF-8 string but is not a valid ASCII string, the UTF-8 mode is enabled automatically.

List of functions

- `Hyperscan::Grep(pattern:String) -> (string:String?) -> Bool`
- `Hyperscan::Match(pattern:String) -> (string:String?) -> Bool`
- `Hyperscan::BacktrackingGrep(pattern:String) -> (string:String?) -> Bool`
- `Hyperscan::BacktrackingMatch(pattern:String) -> (string:String?) -> Bool`
- `Hyperscan::MultiGrep(pattern:String) -> (string:String?) -> Tuple<Bool, Bool, ...>`
- `Hyperscan::MultiMatch(pattern:String) -> (string:String?) -> Tuple<Bool, Bool, ...>`
- `Hyperscan::Capture(pattern:String) -> (string:String?) -> String?`
- `Hyperscan::Replace(pattern:String) -> (string:String?, replacement:String) -> String?`

Call syntax

To avoid compiling a regular expression at each table row at direct call, wrap the function call by a named expression:

```
$re = Hyperscan::Grep("\\d+");      -- create a callable value to match a specific regular expression
SELECT * FROM table WHERE $re(key); -- use it to filter the table
```

Note

Please note escaping of special characters in regular expressions. Be sure to use the second slash, since all the standard string literals in SQL can accept C-escaped strings, and the `\d` sequence is not valid sequence (even if it were, it wouldn't search for numbers as intended).

You can enable the case-insensitive mode by specifying, at the beginning of the regular expression, the flag `(?i)`.

Grep

Matches the regular expression with a **part of the string** (arbitrary substring).

Match

Matches **the whole string** against the regular expression.

To get a result similar to `Grep` (where substring matching is included), enclose the regular expression in `.*` (`.*foo.*` instead of `foo`). However, in terms of code readability, it's usually better to change the function.

BacktrackingGrep/BacktrackingMatch

The functions are identical to the same-name functions without the `Backtracking` prefix. However, they support a broader range of regular expressions. This is due to the fact that if a specific regular expression is not fully supported by Hyperscan, the library switches to the prefilter mode. In this case, it responds not by "Yes" or "No", but by "Definitely not" or "Maybe yes". The "Maybe yes" responses are then automatically rechecked using a slower, but more functional, library libpcre.

MultiGrep/MultiMatch

Hyperscan lets you match against multiple regular expressions in a single pass through the text, and get a separate response for each match.

However, if you want to match a string against any of the listed expressions (the results would be joined with "or"), it would be more efficient to combine the query parts in a single regular expression with `|` and match it with regular `Grep` or `Match`.

When you call `MultiGrep` / `MultiMatch`, regular expressions are passed one per line using multiline string literals:

Example

```
$multi_match = Hyperscan::MultiMatch(@@a.*
.*x.*
.*axa.*@@);

SELECT
    $multi_match("a") AS a,     -- (true, false, false)
    $multi_match("axa") AS axa; -- (true, true, true)
```

Capture and Replace

`Hyperscan::Capture` if a string matches the specified regular expression, it returns the last substring matching the regular expression. `Hyperscan::Replace` replaces all occurrences of the specified regular expression with the specified string.

Hyperscan doesn't support advanced functionality for such operations. Although `Hyperscan::Capture` and `Hyperscan::Replace` are implemented for consistency, it's better to use the same-name functions from the Re2 library for any non-trivial capture and replace:

- Re2::Capture;
- Re2::Replace.

Usage example

```
$value = "xaaxaaXaa";

$match = Hyperscan::Match("a.*");
$grep = Hyperscan::Grep("axa");
$insensitive_grep = Hyperscan::Grep("(?i)axaa$");
$multi_match = Hyperscan::MultiMatch(@@a.*
.*a.*
.*a
.*axa.*@@);

$capture = Hyperscan::Capture(".*a{2}.*");
$capture_many = Hyperscan::Capture(".*x(a+).*");
$replace = Hyperscan::Replace("xa");

SELECT
    $match($value) AS match,                        -- false
    $grep($value) AS grep,                          -- true
    $insensitive_grep($value) AS insensitive_grep,  -- true
    $multi_match($value) AS multi_match,            -- (false, true, true, true)
    $multi_match($value).0 AS some_multi_match,     -- false
    $capture($value) AS capture,                    -- "xaa"
    $capture_many($value) AS capture_many,          -- "xa"
    $replace($value, "b") AS replace                -- "babaXaa"
;
```

Ip

The `Ip` module supports both the IPv4 and IPv6 addresses. By default, they are represented as binary strings of 4 and 16 bytes, respectively.

List of functions

- `Ip::FromString(String{Flags:AutoMap}) -> String?` - From a human-readable representation to a binary representation.
- `Ip::SubnetFromString(String{Flags:AutoMap}) -> String?` - From a human-readable representation of subnet to a binary representation.
- `Ip::ToString(String{Flags:AutoMap}) -> String?` - From a binary representation to a human-readable representation.
- `Ip::SubnetToString(String{Flags:AutoMap}) -> String?` - From a binary representation of subnet to a human-readable representation.
- `Ip::IsIPv4(String?) -> Bool`
- `Ip::IsIPv6(String?) -> Bool`
- `Ip::IsEmbeddedIPv4(String?) -> Bool`
- `Ip::ConvertToIPv6(String{Flags:AutoMap}) -> String`: IPv6 remains unchanged, and IPv4 becomes embedded in IPv6
- `Ip::GetSubnet(String{Flags:AutoMap}, [Uint8?]) -> String`: The second argument is the subnet size, by default it's 24 for IPv4 and 64 for IPv6
- `Ip::GetSubnetByMask(String{Flags:AutoMap}, String{Flags:AutoMap}) -> String`: The first argument is the base address, the second argument is the bit mask of a desired subnet.
- `Ip::SubnetMatch(String{Flags:AutoMap}, String{Flags:AutoMap}) -> Bool`: The first argument is a subnet, the second argument is a subnet or an address.

Examples

```
SELECT Ip::IsEmbeddedIPv4(
  Ip::FromString("::ffff:77.75.155.3")
); -- true

SELECT
  Ip::ToString(
    Ip::GetSubnet(
      Ip::FromString("213.180.193.3")
    )
  ); -- "213.180.193.0"

SELECT
  Ip::SubnetMatch(
    Ip::SubnetFromString("192.168.0.1/16"),
    Ip::FromString("192.168.1.14"),
  ); -- true

SELECT
  Ip::ToString(
    Ip::GetSubnetByMask(
      Ip::FromString("192.168.0.1"),
      Ip::FromString("255.255.0.0")
    )
  ); -- "192.168.0.0"
```

KNN

Introduction

One specific case of vector search is the k-NN problem, where it is required to find the `k` nearest points to the query point. This can be useful in various applications such as image classification, recommendation systems, etc.

The k-NN problem solution is divided into two major subclasses of methods: exact and approximate.

Exact method

The foundation of the exact method is the calculation of the distance from the query vector to all the vectors in the dataset. This algorithm, also known as the naive approach or brute force method, has a runtime of `O(dn)`, where `n` is the number of vectors in the dataset, and `d` is their dimensionality.

Exact vector search is best utilized if the complete enumeration of the vectors occurs within acceptable time limits. This includes cases where they can be pre-filtered based on some condition, such as a user identifier. In such instances, the exact method may perform faster than the current implementation of vector indexes.

Main advantages:

- No need for additional data structures, such as specialized vector indexes.
- Full support for transactions, including in strict consistency mode.
- Instant execution of data modification operations: insertion, update, deletion.

Approximate methods

Approximate methods do not perform a complete enumeration of the initial data. This allows significantly speeding up the search process, although it might lead to some reduction in the quality of the results.

Scalar Quantization is a method of reducing vector dimensionality, where a set of coordinates is mapped into a space of smaller dimensions.

YDB supports vector searching for vector types `Float`, `Int8`, `Uint8`, and `Bit`. Consequently, it is possible to apply scalar quantization to transform data from `Float` to any of these types.

Scalar quantization reduces the time required for reading and writing data by decreasing the number of bytes. For example, when quantizing from `Float` to `Bit`, each vector is reduced by 32 times.

Approximate vector search without an index uses a very simple additional data structure - a set of vectors with other quantization. This allows the use of a simple search algorithm: first, a rough preliminary search is performed on the compressed vectors, followed by refining the results on the original vectors.

Main advantages:

- Full support for transactions, including in strict consistency mode.
- Instant application of data modification operations: insertion, update, deletion.

> **Note**
>
> It is recommended to measure if such quantization provides sufficient accuracy/recall.

Data types

In mathematics, a vector of real or integer numbers is used to store points.
In this module, vectors are stored in the `String` data type, which is a binary serialized representation of a vector.

Functions

Vector functions are implemented as user-defined functions (UDF) in the `Knn` module.

Functions for converting between vector and binary representations

Conversion functions are needed to serialize vectors into an internal binary representation and vice versa.

All serialization functions wrap returned `String` data into Tagged types.

The binary representation of the vector can be stored in the YDB table column.

> **Note**
>
> Currently YDB does not support storing `Tagged`, so before storing binary representation vectors you must call Untag.

> **Note**
>
> Currently YDB does not support building an index for vectors with bit quantization `BitVector`.

Function signatures

```
Knn::ToBinaryStringFloat(List<Float>{Flags:AutoMap})->Tagged<String, "FloatVector">
Knn::ToBinaryStringUint8(List<Uint8>{Flags:AutoMap})->Tagged<String, "Uint8Vector">
Knn::ToBinaryStringInt8(List<Int8>{Flags:AutoMap})->Tagged<String, "Int8Vector">
Knn::ToBinaryStringBit(List<Double>{Flags:AutoMap})->Tagged<String, "BitVector">
Knn::ToBinaryStringBit(List<Float>{Flags:AutoMap})->Tagged<String, "BitVector">
Knn::ToBinaryStringBit(List<Uint8>{Flags:AutoMap})->Tagged<String, "BitVector">
Knn::ToBinaryStringBit(List<Int8>{Flags:AutoMap})->Tagged<String, "BitVector">
Knn::FloatFromBinaryString(String{Flags:AutoMap})->List<Float>?
```

Convert format

Conversion functions for vector data convert an array of elements into a byte string with the following format:

- **Main part** — a contiguous array of elements (knn-serializer.h)
- **Type** — 1 byte at the end of the string that specifies the data type (knn-defines.h):
  `1` — `Float` (4 bytes per element)
  `2` — `Uint8` (1 byte per element)
  `3` — `Int8` (1 byte per element)
  `10` — `Bit` (1 bit per element)

For example, a vector of 5 elements of type `Float` will be serialized into a 21-byte string:
4 bytes × 5 elements (main part) + 1 byte (type) = 21 bytes.

Implementation details

The `ToBinaryStringBit` function maps coordinates that are greater than `0` to `1`. All other coordinates are mapped to `0`.

Distance and similarity functions

The distance and similarity functions take two lists of real numbers as input and return the distance/similarity between them.

> **Note**
>
> Distance functions return small values for close vectors, while similarity functions return large values for close vectors. This should be taken into account when defining the sorting order.

Similarity functions:

- inner product `InnerProductSimilarity`, it's the dot product, also known as the scalar product (sum of products of coordinates)
- cosine similarity `CosineSimilarity` (dot product divided by product of vector lengths)

Distance functions:

- cosine distance `CosineDistance` (1 - cosine similarity)
- manhattan distance `ManhattanDistance`, also known as `L1 distance` (sum of modules of coordinate differences)
- euclidean distance `EuclideanDistance`, also known as `L2 distance` (square root of the sum of squares of coordinate differences)

Function signatures

```
Knn::InnerProductSimilarity(String{Flags:AutoMap}, String{Flags:AutoMap})->Float?
Knn::CosineSimilarity(String{Flags:AutoMap}, String{Flags:AutoMap})->Float?
Knn::CosineDistance(String{Flags:AutoMap}, String{Flags:AutoMap})->Float?
Knn::ManhattanDistance(String{Flags:AutoMap}, String{Flags:AutoMap})->Float?
Knn::EuclideanDistance(String{Flags:AutoMap}, String{Flags:AutoMap})->Float?
```

In case of mismatched lengths or formats, these functions return `NULL`.

> **Note**
>
> All distance and similarity functions support overloads when first or second arguments are `Tagged<String, "FloatVector">`, `Tagged<String, "Uint8Vector">`, `Tagged<String, "Int8Vector">`, `Tagged<String, "BitVector">`.
>
> If both arguments are `Tagged`, tag values should match, or the query will raise an error.
>
> Example:
>
> ```
> Error: Failed to find UDF function: Knn.CosineDistance, reason: Error: Module: Knn, function: Cosi
> neDistance, error: Arguments should have same tags, but 'FloatVector' is not equal to 'Uint8Vecto
> r'
> ```

Exact search examples

Creating a table

```
CREATE TABLE Facts (
    id Uint64,        -- Id of fact
    user Utf8,        -- User name
    fact Utf8,        -- Human-readable description of a user fact
    embedding String, -- Binary representation of embedding vector (result of Knn::ToBinaryStringFloat)
    PRIMARY KEY (id)
);
```

Adding vectors

```
$vector = [1.f, 2.f, 3.f, 4.f];
UPSERT INTO Facts (id, user, fact, embedding)
VALUES (123, "Williams", "Full name is John Williams", Untag(Knn::ToBinaryStringFloat($vector), "FloatVecto
r"));
```

Exact search of K nearest vectors

```
$K = 10;
$TargetEmbedding = Knn::ToBinaryStringFloat([1.2f, 2.3f, 3.4f, 4.5f]);

SELECT * FROM Facts
WHERE user="Williams"
ORDER BY Knn::CosineDistance(embedding, $TargetEmbedding)
LIMIT $K;
```

Exact search of vectors in radius R

```
$R = 0.1f;
$TargetEmbedding = Knn::ToBinaryStringFloat([1.2f, 2.3f, 3.4f, 4.5f]);

SELECT * FROM Facts
WHERE Knn::CosineDistance(embedding, $TargetEmbedding) < $R;
```

Approximate search examples

This example differs from the exact search example by using bit quantization.

This allows to first do a approximate preliminary search by the `embedding_bit` column, and then refine the results by the original vector column `embegging`.

Creating a table

```
CREATE TABLE Facts (
    id Uint64,            -- Id of fact
    user Utf8,            -- User name
    fact Utf8,            -- Human-readable description of a user fact
    embedding String,     -- Binary representation of embedding vector (result of Knn::ToBinaryStringFloat)
    embedding_bit String, -- Binary representation of embedding vector (result of Knn::ToBinaryStringBit)
    PRIMARY KEY (id)
);
```

Adding vectors

```
$vector = [1.f, 2.f, 3.f, 4.f];
UPSERT INTO Facts (id, user, fact, embedding, embedding_bit)
VALUES (123, "Williams", "Full name is John Williams", Untag(Knn::ToBinaryStringFloat($vector), "FloatVecto
r"), Untag(Knn::ToBinaryStringBit($vector), "BitVector"));
```

Scalar quantization

An ML model can do quantization, or it can be done manually with YQL.

Below there is a quantization example in YQL.

Float -> Int8

```
$MapInt8 = ($x) -> {
    $min = -5.0f;
    $max =  5.0f;
    $range = $max - $min;
  RETURN CAST(Math::Round(IF($x < $min, -127, IF($x > $max, 127, ($x / $range) * 255))) As Int8)
};

$FloatList = [-1.2f, 2.3f, 3.4f, -4.7f];
SELECT ListMap($FloatList, $MapInt8);
```

Approximate search of K nearest vectors: bit quantization

Approximate search algorithm:

- an approximate search is performed using bit quantization;
- an approximate list of vectors is obtained;
- we search this list without using quantization.

```
$K = 10;
$Target = [1.2f, 2.3f, 3.4f, 4.5f];
$TargetEmbeddingBit = Knn::ToBinaryStringBit($Target);
$TargetEmbeddingFloat = Knn::ToBinaryStringFloat($Target);

$Ids = SELECT id FROM Facts
ORDER BY Knn::CosineDistance(embedding_bit, $TargetEmbeddingBit)
LIMIT $K * 10;

SELECT * FROM Facts
WHERE id IN $Ids
ORDER BY Knn::CosineDistance(embedding, $TargetEmbeddingFloat)
LIMIT $K;
```

Math

A set of wrappers around the functions from the libm library and the Yandex utilities.

Constants

List of functions

- `Math::Pi() -> Double`
- `Math::E() -> Double`
- `Math::Eps() -> Double`

Examples

```
SELECT Math::Pi();  -- 3.141592654
SELECT Math::E();   -- 2.718281828
SELECT Math::Eps(); -- 2.220446049250313e-16
```

(Double) -> Bool

List of functions

- `Math::IsInf(Double{Flags:AutoMap}) -> Bool`
- `Math::IsNaN(Double{Flags:AutoMap}) -> Bool`
- `Math::IsFinite(Double{Flags:AutoMap}) -> Bool`

Examples

```
SELECT Math::IsNaN(0.0/0.0);    -- true
SELECT Math::IsFinite(1.0/0.0); -- false
```

(Double) -> Double

List of functions

- `Math::Abs(Double{Flags:AutoMap}) -> Double`
- `Math::Acos(Double{Flags:AutoMap}) -> Double`
- `Math::Asin(Double{Flags:AutoMap}) -> Double`
- `Math::Asinh(Double{Flags:AutoMap}) -> Double`
- `Math::Atan(Double{Flags:AutoMap}) -> Double`
- `Math::Cbrt(Double{Flags:AutoMap}) -> Double`
- `Math::Ceil(Double{Flags:AutoMap}) -> Double`
- `Math::Cos(Double{Flags:AutoMap}) -> Double`
- `Math::Cosh(Double{Flags:AutoMap}) -> Double`
- `Math::Erf(Double{Flags:AutoMap}) -> Double`
- `Math::ErfInv(Double{Flags:AutoMap}) -> Double`
- `Math::ErfcInv(Double{Flags:AutoMap}) -> Double`
- `Math::Exp(Double{Flags:AutoMap}) -> Double`
- `Math::Exp2(Double{Flags:AutoMap}) -> Double`
- `Math::Fabs(Double{Flags:AutoMap}) -> Double`
- `Math::Floor(Double{Flags:AutoMap}) -> Double`
- `Math::Lgamma(Double{Flags:AutoMap}) -> Double`
- `Math::Rint(Double{Flags:AutoMap}) -> Double`
- `Math::Sigmoid(Double{Flags:AutoMap}) -> Double`
- `Math::Sin(Double{Flags:AutoMap}) -> Double`
- `Math::Sinh(Double{Flags:AutoMap}) -> Double`
- `Math::Sqrt(Double{Flags:AutoMap}) -> Double`
- `Math::Tan(Double{Flags:AutoMap}) -> Double`
- `Math::Tanh(Double{Flags:AutoMap}) -> Double`
- `Math::Tgamma(Double{Flags:AutoMap}) -> Double`
- `Math::Trunc(Double{Flags:AutoMap}) -> Double`
- `Math::Log(Double{Flags:AutoMap}) -> Double`
- `Math::Log2(Double{Flags:AutoMap}) -> Double`
- `Math::Log10(Double{Flags:AutoMap}) -> Double`

Examples

```
SELECT Math::Sqrt(256);     -- 16
SELECT Math::Trunc(1.2345); -- 1
```

(Double, Double) -> Double

List of functions

- `Math::Atan2(Double{Flags:AutoMap}, Double{Flags:AutoMap}) -> Double`
- `Math::Fmod(Double{Flags:AutoMap}, Double{Flags:AutoMap}) -> Double`
- `Math::Hypot(Double{Flags:AutoMap}, Double{Flags:AutoMap}) -> Double`

- `Math::Pow(Double{Flags:AutoMap}, Double{Flags:AutoMap}) -> Double`
- `Math::Remainder(Double{Flags:AutoMap}, Double{Flags:AutoMap}) -> Double`

Examples

```
SELECT Math::Atan2(1, 0);       -- 1.570796327
SELECT Math::Remainder(2.1, 2); -- 0.1
```

(Double, Int32) -> Double

List of functions

- `Math::Ldexp(Double{Flags:AutoMap}, Int32{Flags:AutoMap}) -> Double`
- `Math::Round(Double{Flags:AutoMap}, [Int32?]) -> Double`: The second argument indicates the power of 10 to which we round (it's negative for decimal digits and positive for rounding to tens, thousands, or millions); the default value is 0

Examples

```
SELECT Math::Pow(2, 10);        -- 1024
SELECT Math::Round(1.2345, -2); -- 1.23
```

(Double, Double, [Double?]) -> Bool

List of functions

- `Math::FuzzyEquals(Double{Flags:AutoMap}, Double{Flags:AutoMap}, [Double?]) -> Bool`: Compares two Doubles for being inside the neighborhood specified by the third argument; the default value is 1.0e-13

Examples

```
SELECT Math::FuzzyEquals(1.01, 1.0, 0.05); -- true
```

Functions for computing remainders

List of functions

- `Math::Mod(Int64{Flags:AutoMap}, Int64) -> Int64?`
- `Math::Rem(Int64{Flags:AutoMap}, Int64) -> Int64?`

These functions behave similarly to the built-in % operator in the case of non-negative arguments. The differences are noticeable in the case of negative arguments:

- `Math::Mod` preserves the sign of the second argument (the denominator).
- `Math::Rem` preserves the sign of the first argument (the numerator).

Functions return null if the divisor is zero.

Examples

```
SELECT Math::Mod(-1, 7);        -- 6
SELECT Math::Rem(-1, 7);        -- -1
```

Pcre

The Pcre library is currently an alias to Hyperscan.

Currently available engines:

- Hyperscan (Intel)
- Pire (Yandex)
- Re2 (Google)

All three modules provide approximately the same set of functions with an identical interface. This lets you switch between them with minimal changes to a query.

Inside Hyperscan, there are several implementations that use different sets of processor instructions, with the relevant instruction automatically selected based on the current processor. In HyperScan, some functions support backtracking (referencing the previously found part of the string). Those functions are implemented through hybrid use of the two libraries: Hyperscan and libpcre.

Pire (Perl Incompatible Regular Expressions) is a very fast library of regular expressions developed by Yandex. At the lower level, it scans the input string once, without any lookaheads or rollbacks, spending 5 machine instructions per character (on x86 and x86_64). However, since the library almost hasn't been developed since 2011-2013 and its name says "Perl incompatible", you may need to adapt your regular expressions a bit.

Hyperscan and Pire are best-suited for Grep and Match.

The Re2 module uses google::RE2 that offers a wide range of features (see the official documentation). The main benefit of the Re2 is its advanced Capture and Replace functionality. Use this library, if you need those functions.

Pire

List of functions

- `Pire::Grep(pattern:String) -> (string:String?) -> Bool`
- `Pire::Match(pattern:String) -> (string:String?) -> Bool`
- `Pire::MultiGrep(pattern:String) -> (string:String?) -> Tuple<Bool, Bool, ...>`
- `Pire::MultiMatch(pattern:String) -> (string:String?) -> Tuple<Bool, Bool, ...>`
- `Pire::Capture(pattern:String) -> (string:String?) -> String?`
- `Pire::Replace(pattern:String) -> (string:String?, replacement:String) -> String?`

One of the options to match regular expressions in YQL is to use Pire (Perl Incompatible Regular Expressions). This is a very fast library of regular expressions developed at Yandex: at the lower level, it looks up the input string once, without any lookaheads or rollbacks, spending 5 machine instructions per character (on x86 and x86_64).

The speed is achieved by using the reasonable restrictions:

- Pire is primarily focused at checking whether a string matches a regular expression.
- The matching substring can also be returned (by Capture), but with restrictions (a match with only one group is returned).

By default, all functions work in the single-byte mode. However, if the regular expression is a valid UTF-8 string but is not a valid ASCII string, the UTF-8 mode is enabled automatically.

To enable the Unicode mode, you can put one character that's beyond ASCII with the `?` operator, for example: `\\w+я?`.

Call syntax

To avoid compiling a regular expression at each table row, wrap the function call by a named expression:

```
$re = Pire::Grep("\\d+"); -- create a callable value to match a specific regular expression
SELECT * FROM table WHERE $re(key); -- use it to filter the table
```

Alert

When escaping special characters in a regular expression, be sure to use the second slash, since all the standard string literals in SQL can accept C-escaped strings, and the `\d` sequence is not a valid sequence (even if it were, it wouldn't search for numbers as intended).

You can enable the case-insensitive mode by specifying, at the beginning of the regular expression, the flag `(?i)`.

Examples

```
$value = "xaaxaaxaa";
$match = Pire::Match("a.*");
$grep = Pire::Grep("axa");
$insensitive_grep = Pire::Grep("(?i)axa");
$multi_match = Pire::MultiMatch(@@a.*
.*a.*
.*a
.*axa.*@@);
$capture = Pire::Capture(".*x(a).*");
$capture_many = Pire::Capture(".*x(a+).*");
$replace = Pire::Replace(".*x(a).*");

SELECT
  $match($value) AS match,                        -- false
  $grep($value) AS grep,                          -- true
  $insensitive_grep($value) AS insensitive_grep,  -- true
  $multi_match($value) AS multi_match,            -- (false, true, true, true)
  $multi_match($value).0 AS some_multi_match,     -- false
  $capture($value) AS capture,                    -- "a"
  $capture_many($value) AS capture_many,          -- "aa"
  $replace($value, "b") AS replace;               -- "xaaxaaxba"
```

Grep

Matches the regular expression with a **part of the string** (arbitrary substring).

Match

Matches **the whole string** against the regular expression.
To get a result similar to `Grep` (where substring matching is included), enclose the regular expression in `.*`. For example, use `.*foo.*` instead of `foo`.

MultiGrep/MultiMatch

Pire lets you match against multiple regular expressions in a single pass through the text and get a separate response for each match. Use the MultiGrep/MultiMatch functions to optimize the query execution speed. Be sure to do it carefully, since the size of the state machine used for matching grows exponentially with the number of regular expressions:

- If you want to match a string against any of the listed expressions (the results are joined with "or"), it would be much more efficient to combine the query parts in a single regular expression with `|` and match it using regular Grep or Match.
- Pire has a limit on the size of the state machine (YQL uses the default value set in the library). If you exceed the limit, the error is raised at the start of the query: `Failed to glue up regexes, probably the finite state machine appeared to be too large`.

When you call MultiGrep/MultiMatch, regular expressions are passed one per line using multiline string literals:

Examples

```
$multi_match = Pire::MultiMatch(@@a.*
.*x.*
.*axa.*@@);

SELECT
    $multi_match("a") AS a,      -- (true, false, false)
    $multi_match("axa") AS axa;  -- (true, true, true)
```

Capture

If a string matches the specified regular expression, it returns a substring that matches the group enclosed in parentheses in the regular expression.
Capture is non-greedy: the shortest possible substring is returned.

> **Alert**
>
> The expression must contain only **one** group in parentheses. `NULL` (empty Optional) is returned in case of no match.

If the above limitations and features are unacceptable for some reason, we recommend that you consider Re2::Capture.

REPLACE

Pire doesn't support replace based on a regular expression. `Pire::Replace` implemented in YQL is a simplified emulation using `Capture`. It may run correctly, if the substring occurs more than once in the source string.

As a rule, it's better to use Re2::Replace instead.

Re2

List of functions

```
Re2::Grep(pattern:String, options:Struct<...>?) -> (string:String?) -> Bool
Re2::Match(pattern:String, options:Struct<...>?) -> (string:String?) -> Bool
Re2::Capture(pattern:String, options:Struct<...>?) -> (string:String?) -> Struct<_1:String?,foo:String?,...>
Re2::FindAndConsume(pattern:String, options:Struct<...>?) -> (string:String?) -> List<String>
Re2::Replace(pattern:String, options:Struct<...>?) -> (string:String?, replacement:String) -> String?
Re2::Count(pattern:String, options:Struct<...>?) -> (string:String?) -> Uint32
Re2::Options([CaseSensitive:Bool?,DotNl:Bool?,Literal:Bool?,LogErrors:Bool?,LongestMatch:Bool?,MaxMem:Uint6
4?,NeverCapture:Bool?,NeverNl:Bool?,OneLine:Bool?,PerlClasses:Bool?,PosixSyntax:Bool?,Utf8:Bool?,WordBoundar
y:Bool?]) -> Struct<CaseSensitive:Bool,DotNl:Bool,Literal:Bool,LogErrors:Bool,LongestMatch:Bool,MaxMem:Uint6
4,NeverCapture:Bool,NeverNl:Bool,OneLine:Bool,PerlClasses:Bool,PosixSyntax:Bool,Utf8:Bool,WordBoundary:Bool>
```

The Re2 module supports regular expressions based on google::RE2 with a wide range of features provided (see the official documentation).

By default, the UTF-8 mode is enabled automatically if the regular expression is a valid UTF-8-encoded string, but is not a valid ASCII string. You can manually control the settings of the re2 library, if you pass the result of the `Re2::Options` function as the second argument to other module functions, next to the regular expression.

Warning

Make sure to double all the backslashes in your regular expressions (if they are within a quoted string): standard string literals are treated as C-escaped strings in SQL. You can also format regular expressions as raw strings `@@regexp@@`: double slashes are not needed in this case.

Examples

```
$value = "xaaxaaxaa";
$options = Re2::Options(false AS CaseSensitive);
$match = Re2::Match("[ax]+\\d");
$grep = Re2::Grep("a.*");
$capture = Re2::Capture(".*(?P<foo>xa?)(a{2,}).*");
$replace = Re2::Replace("x(a+)x");
$count = Re2::Count("a", $options);

SELECT
  $match($value) AS match,                -- false
  $grep($value) AS grep,                  -- true
  $capture($value) AS capture,            -- (_0: 'xaaxaaxaa', _1: 'aa', foo: 'x')
  $capture($value)._1 AS capture_member,  -- "aa"
  $replace($value, "b\\1z") AS replace,   -- "baazaaxaa"
  $count($value) AS count;                -- 6
```

Re2::Grep / Re2::Match

If you leave out the details of implementation and syntax of regular expressions, those functions are totally similar to the same-name functions from the Pire module. With other things equal and no specific preferences, we recommend that you use `Pire::Grep or Pire::Match`.

You can call the `Re2::Grep` function by using a `REGEXP` expression (see the basic expression syntax).

For example, the following two queries are equivalent (also in terms of computing efficiency):

- `$grep = Re2::Grep("b+"); SELECT $grep("aaabccc");`
- `SELECT "aaabccc" REGEXP "b+";`

Re2::Capture

Unlike Pire::Capture, `Re2::Capture` supports multiple and named capturing groups.
Result type: a structure with the fields of the type `String?`.

- Each field corresponds to a capturing group with the applicable name.
- For unnamed groups, the following names are generated: `_1`, `_2`, etc.
- The result always includes the `_0` field containing the entire substring matching the regular expression.

For more information about working with structures in YQL, see the section on containers.

Re2::FindAndConsume

Searches for all occurrences of the regular expression in the passed text and returns a list of values corresponding to the parenthesized part of the regular expression for each occurrence.

Re2::Replace

Works as follows:

- In the input string (first argument), all the non-overlapping substrings matching the regular expression are replaced by the specified string (second argument).
- In the replacement string, you can use the contents of capturing groups from the regular expression using back-references in the format: `\\1`, `\\2` etc. The `\\0` back-reference stands for the whole substring that matches the regular expression.

Re2::Count

Returns the number of non-overlapping substrings of the input string that have matched the regular expression.

Re2::Options

Notes on Re2::Options from the official repository

Parameter	Default	Comments
CaseSensitive:Bool?	true	match is case-sensitive (regexp can override with (?i) unless in posix_syntax mode)
DotNl:Bool?	false	let `.` match `\n` (default)
Literal:Bool?	false	interpret string as literal, not regexp
LogErrors:Bool?	true	log syntax and execution errors to ERROR
LongestMatch:Bool?	false	search for longest match, not first match
MaxMem:Uint64?	-	(see below) approx. max memory footprint of RE2
NeverCapture:Bool?	false	parse all parents as non-capturing
NeverNl:Bool?	false	never match \n, even if it is in regexp
PosixSyntax:Bool?	false	restrict regexps to POSIX egrep syntax
Utf8:Bool?	true	text and pattern are UTF-8; otherwise Latin-1
The following options are only consulted when PosixSyntax == true. When PosixSyntax == false, these features are always enabled and cannot be turned off; to perform multi-line matching in that case, begin the regexp with (?m).		
PerlClasses:Bool?	false	allow Perl's \d \s \w \D \S \W
WordBoundary:Bool?	false	allow Perl's \b \B (word boundary and not)
OneLine:Bool?	false	^ and $ only match beginning and end of text

It is not recommended to use Re2::Options in the code. Most parameters can be replaced with regular expression flags.

Flag usage examples

```
$value = "Foo bar FOO"u;
-- enable case-insensitive mode
$capture = Re2::Capture(@@(?i)(foo)@@);

SELECT
    $capture($value) AS capture; -- ("_0": "Foo", "_1": "Foo")

$capture = Re2::Capture(@@(?i)(?P<foo>FOO).*(?P<bar>bar)@@);

SELECT
    $capture($value) AS capture; -- ("_0": "Foo bar", "bar": "bar", "foo": "Foo")
```

In both cases, the word FOO will be found. Using the raw string @@regexp@@ lets you avoid double slashes.

Roaring

Introduction

Bitsets, also called bitmaps, are commonly used as fast data structures. Unfortunately, they can use too much memory. To compensate, we often use compressed bitmaps.

Roaring bitmaps are compressed bitmaps which tend to outperform conventional compressed bitmaps such as WAH, EWAH or Concise. In some instances, roaring bitmaps can be hundreds of times faster and they often offer significantly better compression. They can even be faster than uncompressed bitmaps.

Implementation

You can work with Roaring bitmaps in YDB using a set of user-defined functions (UDFs) in the `Roaring` module. These functions provide the ability to work with 32-bit Roaring bitmaps. To do this, the data must be serialized in the format for 32-bit bitmaps described in the specification. This can be done using a function available in the Roaring bitmap library itself.

Such libraries exist for many programming languages, such as Go. If the serialization happened on the client side, the application can then save the serialized bitmap in a column with the `String` type.

To work with Roaring bitmaps in a query, data from the `String` type must be deserialized into the Resource<roaring_bitmap> type. To save the data, you need to perform the reverse operation. After that, the application can read the updated bitmap from YDB and deserialize it.

Available methods

```
Roaring::Deserialize(String{Flags:AutoMap})->Resource<roaring_bitmap>
Roaring::FromUint32List(List<Uint32>{Flags:AutoMap})->Resource<roaring_bitmap>
Roaring::Serialize(Resource<roaring_bitmap>{Flags:AutoMap})->String
Roaring::Uint32List(Resource<roaring_bitmap>{Flags:AutoMap})->List<Uint32>

Roaring::Cardinality(Resource<roaring_bitmap>{Flags:AutoMap})->Uint32

Roaring::Or(Resource<roaring_bitmap>{Flags:AutoMap}, Resource<roaring_bitmap>{Flags:AutoMap})->Resource<roari
ng_bitmap>
Roaring::OrWithBinary(Resource<roaring_bitmap>{Flags:AutoMap}, String{Flags:AutoMap})->Resource<roaring_bitma
p>

Roaring::And(Resource<roaring_bitmap>{Flags:AutoMap}, Resource<roaring_bitmap>{Flags:AutoMap})->Resource<roar
ing_bitmap>
Roaring::AndWithBinary(Resource<roaring_bitmap>{Flags:AutoMap}, String{Flags:AutoMap})->Resource<roaring_bitm
ap>

Roaring::AndNot(Resource<roaring_bitmap>{Flags:AutoMap}, Resource<roaring_bitmap>{Flags:AutoMap})->Resource<r
oaring_bitmap>
Roaring::AndNotWithBinary(Resource<roaring_bitmap>{Flags:AutoMap}, String{Flags:AutoMap})->Resource<roaring_b
itmap>

Roaring::RunOptimize(Resource<roaring_bitmap>{Flags:AutoMap})->Resource<roaring_bitmap>
```

Serialization and deserialization

Two functions, `Deserialize` and `FromUint32List`, are available for creating `Resource<roaring_bitmap>`. The second function allows creating a Roaring bitmap from a list of unsigned integers, i.e., without the need to use the Roaring bitmap library code to create a binary representation.

YDB does not store data with the `Resource` type, so the created bitmap must be converted to a binary representation using the `Serialize` method.

To use the resulting bitmap, for example, in a `WHERE` condition, the `Uint32List` method is provided. This method returns a list of unsigned integers from the `Resource<roaring_bitmap>`.

Bitwise operations

Currently, three modifying binary operations with bitmaps are supported:

- `Or`
- `And`
- `AndNot`

The operations are modifying, meaning that they modify the `Resource<roaring_bitmap>` passed as the first argument. Each of these operations has a version with the `WithBinary` suffix, which allows working with the binary representation without having to deserialize it into the `Resource<roaring_bitmap>` type. The implementation of these methods still has to deserialize the data to perform the operation, but it does not create an intermediate `Resource`, thereby saving resources.

Other operations

The `Cardinality` function is provided to obtain the number of bits set to 1 in the `Resource<roaring_bitmap>`.

After the bitmap has been constructed or modified, it can be optimized using the `RunOptimize` method. The internal format of a Roaring bitmap can use containers with better representations for different bit sequences.

Examples

```
$b = Roaring::FromUint32List(AsList(42));
$b = Roaring::Or($b, Roaring::FromUint32List(AsList(56)));


SELECT Roaring::Uint32List($b) AS `Or`; -- [42, 56]
```

```
$b1 = Roaring::FromUint32List(AsList(10, 567, 42));
$b2 = Roaring::FromUint32List(AsList(42));

$b2ser = Roaring::Serialize($b2); -- save this to String column

SELECT Roaring::Cardinality(Roaring::AndWithBinary($b1, $b2ser)) AS Cardinality; -- 1

SELECT Roaring::Uint32List(Roaring::And($b1, $b2)) AS `And`; -- [42]
SELECT Roaring::Uint32List(Roaring::AndWithBinary($b1, $b2ser)) AS AndWithBinary; -- [42]
```

```
$b1 = Roaring::FromUint32List(AsList(10, 567, 42));
$b2 = Roaring::FromUint32List(AsList(42));

$b2ser = Roaring::Serialize($b2); -- save this to String column

SELECT Roaring::Cardinality(Roaring::AndNotWithBinary($b1, $b2ser)) AS Cardinality; -- 2

SELECT Roaring::Uint32List(Roaring::AndNot($b1, $b2)) AS AndNot; -- [10,567]
SELECT Roaring::Uint32List(Roaring::AndNotWithBinary($b1, $b2ser)) AS AndNotWithBinary; -- [10,567]
```

String

Functions for ASCII strings:

List of functions

- `String::Base64Encode(String{Flags:AutoMap}) -> String`
- `String::Base64Decode(String) -> String?`
- `String::Base64StrictDecode(String) -> String?`
- `String::EscapeC(String{Flags:AutoMap}) -> String`
- `String::UnescapeC(String{Flags:AutoMap}) -> String`
- `String::HexEncode(String{Flags:AutoMap}) -> String`
- `String::HexDecode(String) -> String?`
- `String::EncodeHtml(String{Flags:AutoMap}) -> String`
- `String::DecodeHtml(String{Flags:AutoMap}) -> String`
- `String::CgiEscape(String{Flags:AutoMap}) -> String`
- `String::CgiUnescape(String{Flags:AutoMap}) -> String`
- `String::Strip(String{Flags:AutoMap}) -> String`
- `String::Collapse(String{Flags:AutoMap}) -> String`
- `String::CollapseText(String{Flags:AutoMap}, Uint64) -> String`
- `String::Contains(String?, String) -> Bool`
- `String::Find(String{Flags:AutoMap}, String, [Uint64?]) -> Int64`: Returns the first position found or -1. The optional argument is the offset from the beginning of the string.
- `String::ReverseFind(String{Flags:AutoMap}, String, [Uint64?]) -> Int64`: Returns the last position found or -1. The optional argument is the offset from the beginning of the string.
- `String::HasPrefix(String?, String) -> Bool`
- `String::HasPrefixIgnoreCase(String?, String) -> Bool`
- `String::StartsWith(String?, String) -> Bool`
- `String::StartsWithIgnoreCase(String?, String) -> Bool`
- `String::HasSuffix(String?, String) -> Bool`
- `String::HasSuffixIgnoreCase(String?, String) -> Bool`
- `String::EndsWith(String?, String) -> Bool`
- `String::EndsWithIgnoreCase(String?, String) -> Bool`
- `String::Substring(String{Flags:AutoMap}, [Uint64?, Uint64?]) -> String`
- `String::AsciiToLower(String{Flags:AutoMap}) -> String`: Changes only Latin characters. For working with other alphabets, see Unicode::ToLower
- `String::AsciiToUpper(String{Flags:AutoMap}) -> String`: Changes only Latin characters. For working with other alphabets, see Unicode::ToUpper
- `String::AsciiToTitle(String{Flags:AutoMap}) -> String`: Changes only Latin characters. For working with other alphabets, see Unicode::ToTitle
- `String::SplitToList( String?, String, [ DelimeterString:Bool?, SkipEmpty:Bool?, Limit:Uint64? ]) -> List<String>`

  The first argument is the source string
  The second argument is a delimiter
  The third argument includes the following parameters:

  - DelimeterString:Bool? — treating a delimiter as a string (true, by default) or a set of characters "any of" (false)
  - SkipEmpty:Bool? — whether to skip empty strings in the result, is false by default
  - Limit:Uint64? — Limits the number of fetched components (unlimited by default); if the limit is exceeded, the raw suffix of the source string is returned in the last item
- `String::JoinFromList(List<String>{Flags:AutoMap}, String) -> String`
- `String::ToByteList(List<String>{Flags:AutoMap}) -> List<Byte>`
- `String::FromByteList(List<Uint8>) -> String`
- `String::ReplaceAll(String{Flags:AutoMap}, String, String) -> String`: Arguments: input, find, replacement
- `String::ReplaceFirst(String{Flags:AutoMap}, String, String) -> String`: Arguments: input, find, replacement
- `String::ReplaceLast(String{Flags:AutoMap}, String, String) -> String`: Arguments: input, find, replacement
- `String::RemoveAll(String{Flags:AutoMap}, String) -> String`: The second argument is interpreted as an unordered set of characters to delete
- `String::RemoveFirst(String{Flags:AutoMap}, String) -> String`: An unordered set of characters in the second argument, only the first encountered character from set is deleted
- `String::RemoveLast(String{Flags:AutoMap}, String) -> String`: An unordered set of characters in the second argument, only the last encountered character from the set is deleted
- `String::IsAscii(String{Flags:AutoMap}) -> Bool`
- `String::IsAsciiSpace(String{Flags:AutoMap}) -> Bool`
- `String::IsAsciiUpper(String{Flags:AutoMap}) -> Bool`
- `String::IsAsciiLower(String{Flags:AutoMap}) -> Bool`
- `String::IsAsciiAlpha(String{Flags:AutoMap}) -> Bool`
- `String::IsAsciiAlnum(String{Flags:AutoMap}) -> Bool`
- `String::IsAsciiHex(String{Flags:AutoMap}) -> Bool`
- `String::LevensteinDistance(String{Flags:AutoMap}, String{Flags:AutoMap}) -> Uint64`
- `String::LeftPad(String{Flags:AutoMap}, Uint64, [String?]) -> String`
- `String::RightPad(String{Flags:AutoMap}, Uint64) -> String`
- `String::Hex(Uint64{Flags:AutoMap}) -> String`

- `String::SHex(Int64{Flags:AutoMap}) -> String`
- `String::Bin(Uint64{Flags:AutoMap}) -> String`
- `String::SBin(Int64{Flags:AutoMap}) -> String`
- `String::HexText(String{Flags:AutoMap}) -> String`
- `String::BinText(String{Flags:AutoMap}) -> String`
- `String::HumanReadableDuration(Uint64{Flags:AutoMap}) -> String`
- `String::HumanReadableQuantity(Uint64{Flags:AutoMap}) -> String`
- `String::HumanReadableBytes(Uint64{Flags:AutoMap}) -> String`
- `String::Prec(Double{Flags:AutoMap}, Uint64) -> String`
- `String::Reverse(String?) -> String?`

Alert

The functions from the String library don't support Cyrillic and can only work with ASCII characters. To work with UTF-8 encoded strings, use functions from Unicode.

Examples

```
SELECT String::Base64Encode("YQL"); -- "WVFM"
SELECT String::Base64Decode("WVFM"); -- "YQL"
SELECT String::Strip("YQL ");       -- "YQL"
SELECT String::SplitToList("1,2,3,4,5,6,7", ",", 3 as Limit); -- ["1", "2", "3", "4,5,6,7"]
```

Unicode

Functions for Unicode strings.

List of functions

- `Unicode::IsUtf(String) -> Bool`

  Checks whether a string is a valid UTF-8 sequence. For example, the string `"\xF0"` isn't a valid UTF-8 sequence, but the string `"\xF0\x9F\x90\xB1"` correctly describes a UTF-8 cat emoji.
- `Unicode::GetLength(Utf8{Flags:AutoMap}) -> Uint64`

  Returns the length of a utf-8 string in unicode code points. Surrogate pairs are counted as one character.

  ```
  SELECT Unicode::GetLength("жніўня"); -- 6
  ```
- `Unicode::Find(string:Utf8{Flags:AutoMap}, subString:Utf8, [pos:Uint64?]) -> Uint64?`
- `Unicode::RFind(string:Utf8{Flags:AutoMap}, subString:Utf8, [pos:Uint64?]) -> Uint64?`

  Finding the first (`RFind` - the last) occurrence of a substring in a string starting from the `pos` position. Returns the position of the first character from the found substring. In case of failure, returns Null.

  ```
  SELECT Unicode::Find("aaa", "bb"); -- Null
  ```
- `Unicode::Substring(string:Utf8{Flags:AutoMap}, from:Uint64?, len:Uint64?) -> Utf8`

  Returns a `string` substring starting with `from` that is `len` characters long. If the `len` argument is omitted, the substring is taken to the end of the source string.

  If `from` exceeds the length of the original string, an empty string `""` is returned.

  ```
  SELECT Unicode::Substring("0123456789abcdefghij", 10); -- "abcdefghij"
  ```
- The `Unicode::Normalize..` functions convert the passed UTF-8 string to a normalization form:
  - `Unicode::Normalize(Utf8{Flags:AutoMap}) -> Utf8` -- NFC
  - `Unicode::NormalizeNFD(Utf8{Flags:AutoMap}) -> Utf8`
  - `Unicode::NormalizeNFC(Utf8{Flags:AutoMap}) -> Utf8`
  - `Unicode::NormalizeNFKD(Utf8{Flags:AutoMap}) -> Utf8`
  - `Unicode::NormalizeNFKC(Utf8{Flags:AutoMap}) -> Utf8`
- `Unicode::Translit(string:Utf8{Flags:AutoMap}, [lang:String?]) -> Utf8`

  Transliterates with Latin letters the words from the passed string, consisting entirely of characters of the alphabet of the language passed by the second argument. If no language is specified, the words are transliterated from Russian. Available languages: "kaz", "rus", "tur", and "ukr".

  ```
  SELECT Unicode::Translit("Тот уголок земли, где я провел"); -- "Tot ugolok zemli, gde ya provel"
  ```
- `Unicode::LevensteinDistance(stringA:Utf8{Flags:AutoMap}, stringB:Utf8{Flags:AutoMap}) -> Uint64`

  Calculates the Levenshtein distance for the passed strings.
- `Unicode::Fold(Utf8{Flags:AutoMap}, [ Language:String?, DoLowerCase:Bool?, DoRenyxa:Bool?, DoSimpleCyr:Bool?, FillOffset:Bool? ]) -> Utf8`

  Performs case folding on the passed string.

  Parameters:

  - `Language` is set according to the same rules as in `Unicode::Translit()`.
  - `DoLowerCase` converts a string to lowercase letters, defaults to `true`.
  - `DoRenyxa` converts diacritical characters to similar Latin characters, defaults to `true`.
  - `DoSimpleCyr` converts diacritical Cyrillic characters to similar Latin characters, defaults to `true`.
  - `FillOffset` parameter is not used.

  ```
  SELECT Unicode::Fold("Kongreßstraße", false AS DoSimpleCyr, false AS DoRenyxa); -- "kongressstrasse"
  SELECT Unicode::Fold("сурт"); -- "сурт"
  SELECT Unicode::Fold("Eylül", "Turkish" AS Language); -- "eylul"
  ```
- `Unicode::ReplaceAll(input:Utf8{Flags:AutoMap}, find:Utf8, replacement:Utf8) -> Utf8`
- `Unicode::ReplaceFirst(input:Utf8{Flags:AutoMap}, find:Utf8, replacement:Utf8) -> Utf8`
- `Unicode::ReplaceLast(input:Utf8{Flags:AutoMap}, find:Utf8, replacement:Utf8) -> Utf8`

  Replaces all/first/last occurrences of the `find` string in the `input` with `replacement`.
- `Unicode::RemoveAll(input:Utf8{Flags:AutoMap}, symbols:Utf8) -> Utf8`
- `Unicode::RemoveFirst(input:Utf8{Flags:AutoMap}, symbols:Utf8) -> Utf8`
- `Unicode::RemoveLast(input:Utf8{Flags:AutoMap}, symbols:Utf8) -> Utf8`

  Deletes all/first/last occurrences of characters in the `symbols` set from the `input`. The second argument is interpreted as an unordered set of characters to be removed.

  ```
  SELECT Unicode::ReplaceLast("absence", "enc", ""); -- "abse"
  ```

```
SELECT Unicode::RemoveAll("abandon", "an"); -- "bdo"
```

- `Unicode::ToCodePointList(Utf8{Flags:AutoMap}) -> List<Uint32>`

  Splits a string into a Unicode sequence of codepoints.
- `Unicode::FromCodePointList(List<Uint32>{Flags:AutoMap}) -> Utf8`

  Generates a Unicode string from codepoints.

```
SELECT Unicode::ToCodePointList("Щавель"); -- [1065, 1072, 1074, 1077, 1083, 1100]
SELECT Unicode::FromCodePointList(AsList(99,111,100,101,32,112,111,105,110,116,115,32,99,111,110,118,10
1,114,116,101,114)); -- "code points converter"
```

- `Unicode::Reverse(Utf8{Flags:AutoMap}) -> Utf8`

  Reverses a string.
- `Unicode::ToLower(Utf8{Flags:AutoMap}) -> Utf8`
- `Unicode::ToUpper(Utf8{Flags:AutoMap}) -> Utf8`
- `Unicode::ToTitle(Utf8{Flags:AutoMap}) -> Utf8`

  Converts a string to UPPER, lower, or Title case.
- `Unicode::SplitToList(string:Utf8?, separator:Utf8, [DelimeterString:Bool?, SkipEmpty:Bool?, Limit:Uint64?
  ]) -> List<Utf8>`

  Splits a string into substrings by separator.
  `string` -- Source string. `separator` -- Separator. Parameters:

  - DelimeterString:Bool? — treating a delimiter as a string (true, by default) or a set of characters "any of" (false)
  - SkipEmpty:Bool? - whether to skip empty strings in the result, is false by default
  - Limit:Uint64? - Limits the number of fetched components (unlimited by default); if the limit is exceeded, the raw suffix of the source string is returned in the last item
- `Unicode::JoinFromList(List<Utf8>{Flags:AutoMap}, separator:Utf8) -> Utf8`

  Concatenates a list of strings via a `separator` into a single string.

```
SELECT Unicode::SplitToList("One, two, three, four, five", ", ", 2 AS Limit); -- ["One", "two", "three,
four, five"]
SELECT Unicode::JoinFromList(["One", "two", "three", "four", "five"], ";"); -- "One;two;three;four;five"
```

- `Unicode::ToUint64(string:Utf8{Flags:AutoMap}, [prefix:Uint16?]) -> Uint64`

  Converts a string to a number.

  The second optional argument sets the number system. By default, 0 (automatic detection by prefix).
  Supported prefixes: `0x(0X)` - base-16, `0` - base-8. Defaults to base-10.
  The `-` sign before a number is interpreted as in C unsigned arithmetic. For example, `-0x1` -> UI64_MAX.
  If there are incorrect characters in a string or a number goes beyond ui64, the function terminates with an error.
- `Unicode::TryToUint64(string:Utf8{Flags:AutoMap}, [prefix:Uint16?]) -> Uint64?`

  Similar to the `Unicode::ToUint64()` function, except that it returns `NULL` instead of an error.

```
SELECT Unicode::ToUint64("77741"); -- 77741
SELECT Unicode::ToUint64("-77741"); -- 18446744073709473875
SELECT Unicode::TryToUint64("asdh831"); -- Null
```

Url

Normalize

- `Url::Normalize(String) -> String?`

Normalizes the URL in a robot-friendly way: converts the hostname into lowercase, strips out certain fragments, and so on.
The normalization result only depends on the URL itself. The normalization **DOES NOT** include operations depending on the external data: transformation based on duplicates, mirrors, etc.

Returned value:

- Normalized URL.
- `NULL`, if the passed string argument can't be parsed as a URL.

Examples

```
SELECT Url::Normalize("hTTp://wWw.yDb.TECH/"); -- "http://www.ydb.tech/"
SELECT Url::Normalize("http://ydb.tech#foo");      -- "http://ydb.tech/"
```

NormalizeWithDefaultHttpScheme

- `Url::NormalizeWithDefaultHttpScheme(String?) -> String?`

Normalizes similarly to `Url::Normalize`, but inserts the `http://` schema in case there is no schema.

Returned value:

- Normalized URL.
- Source URL, if the normalization has failed.

Examples

```
SELECT Url::NormalizeWithDefaultHttpScheme("wWw.yDb.TECH");    -- "http://www.ydb.tech/"
SELECT Url::NormalizeWithDefaultHttpScheme("http://ydb.tech#foo"); -- "http://ydb.tech/"
```

Encode / Decode

Encode a UTF-8 string to the urlencoded format (`Url::Encode`) and back (`Url::Decode`).

List of functions

- `Url::Encode(String?) -> String?`
- `Url::Decode(String?) -> String?`

Examples

```
SELECT Url::Decode("http://ydb.tech/%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%86%D0%B0");
  -- "http://ydb.tech/page"
SELECT Url::Encode("http://ydb.tech/page");
  -- "http://ydb.tech/%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%86%D0%B0"
```

Parse

Parses the URL into parts.

```
Url::Parse(Parse{Flags:AutoMap}) -> Struct< Frag: String?, Host: String?, ParseError: String?, Pass: String?,
Path: String?, Port: String?, Query: String?, Scheme: String?, User: String? >
```

Examples

```
SELECT Url::Parse(
  "https://en.wikipedia.org/wiki/Isambard_Kingdom_Brunel?s=24&g=h-24#Great_Western_Railway");
/*
(
  "Frag": "Great_Western_Railway",
  "Host": "en.wikipedia.org",
  "ParseError": null,
  "Pass": null,
  "Path": "/wiki/Isambard_Kingdom_Brunel",
  "Port": null,
  "Query": "s=24&g=h-24",
  "Scheme": "https",
  "User": null
)
*/
```

Get...

Get a component of the URL.

List of functions

- `Url::GetScheme(String{Flags:AutoMap}) -> String`
- `Url::GetHost(String?) -> String?`
- `Url::GetHostPort(String?) -> String?`

- `Url::GetSchemeHost(String?) -> String?`
- `Url::GetSchemeHostPort(String?) -> String?`
- `Url::GetPort(String?) -> String?`
- `Url::GetTail(String?) -> String?` -- everything following the host: path + query + fragment
- `Url::GetPath(String?) -> String?`
- `Url::GetFragment(String?) -> String?`
- `Url::GetCGIParam(String?, String) -> String?` -- The second parameter is the name of the intended CGI parameter.
- `Url::GetDomain(String?, Uint8) -> String?` -- The second parameter is the required domain level.
- `Url::GetTLD(String{Flags:AutoMap}) -> String`
- `Url::IsKnownTLD(String{Flags:AutoMap}) -> Bool` -- Registered on iana.org.
- `Url::IsWellKnownTLD(String{Flags:AutoMap}) -> Bool` -- Belongs to a small whitelist of com, net, org, ru, and so on.
- `Url::GetDomainLevel(String{Flags:AutoMap}) -> Uint64`
- `Url::GetSignificantDomain(String{Flags:AutoMap}, [List<String>?]) -> String`

  Returns a second-level domain in most cases and a third-level domain for the hostnames like: `***.XXX.YY`, where `XXX` is com, net, org, co, gov, or edu. You can redefine this list using an optional second argument
- `Url::GetOwner(String{Flags:AutoMap}) -> String`

  Returns the domain that's most likely owned by an individual or organization. Unlike `Url::GetSignificantDomain`, it uses a special whitelist. Besides the `***.co.uk` domains, it can return a third-level domain used by free hosting sites and blogs (for example: something.livejournal.com)

Examples

```
SELECT Url::GetScheme("https://ydb.tech");       -- "https://"
SELECT Url::GetDomain("http://www.ydb.tech", 2); -- "ydb.tech"
```

Cut...

- `Url::CutScheme(String?) -> String?`

  Returns the passed URL without the schema (`http://`, `https://`, etc.).
- `Url::CutWWW(String?) -> String?`

  Returns the passed domain without the "www." prefix (if any).
- `Url::CutWWW2(String?) -> String?`

  Returns the passed domain without the prefixes like "www.", "www2.", "wwww777." (if any).
- `Url::CutQueryStringAndFragment(String{Flags:AutoMap}) -> String`

  Returns a copy of the passed URL, stripping out all the CGI parameters and fragments ("?foo=bar" and/or "#baz").

Examples

```
SELECT Url::CutScheme("http://www.ydb.tech"); -- "www.ydb.tech"
SELECT Url::CutWWW("www.ydb.tech");           -- "ydb.tech"
```

...Punycode...

Punycode transformations.

List of functions

- `Url::HostNameToPunycode(String{Flag:AutoMap}) -> String?`
- `Url::ForceHostNameToPunycode(String{Flag:AutoMap}) -> String`
- `Url::PunycodeToHostName(String{Flag:AutoMap}) -> String?`
- `Url::ForcePunycodeToHostName(String{Flag:AutoMap}) -> String`
- `Url::CanBePunycodeHostName(String{Flag:AutoMap}) -> Bool`

Examples

```
SELECT Url::PunycodeToHostName("xn--80aniges7g.xn--j1aef"); -- "example.com"
```

...Query...

Query transformations.

List of functions

```
Url::QueryStringToList(String{Flag:AutoMap}, [
  KeepBlankValues:Bool?,  -- Empty values in percent-encoded queries are interpreted as empty strings, defaul
ts to false.
  Strict:Bool?,           -- If false, parsing errors are ignored and incorrect fields are skipped, defaults
to true.
  MaxFields:Uint32?,      -- The maximum number of fields. If exceeded, an exception is thrown. Defaults to M
ax<Uint32>.
  Separator:String?       -- A key-value pair separator, defaults to '&'.
]) -> List<Tuple<String, String>>
Url::QueryStringToDict(String{Flag:AutoMap}, [
  KeepBlankValues:Bool?,  -- Empty values in percent-encoded queries are interpreted as empty strings, defaul
ts to false.
  Strict:Bool?,           -- If false, parsing errors are ignored and incorrect fields are skipped, defaults
```

```
to true.
  MaxFields:Uint32?,      -- The maximum number of fields. If exceeded, an exception is thrown. Defaults to M
ax<Uint32>.
  Separator:String?       -- A key-value pair separator, defaults to '&'.
]) -> Dict<String, List<String>>
Url::BuildQueryString(Dict<String, List<String?>>{Flag:AutoMap}, [
  Separator:String?       -- A key-value pair separator, defaults to '&'.
]) -> String
Url::BuildQueryString(Dict<String, String?>{Flag:AutoMap}, [
  Separator:String?       -- A key-value pair separator, defaults to '&'.
]) -> String
Url::BuildQueryString(List<Tuple<String, String?>>{Flag:AutoMap}, [
  Separator:String?       -- A key-value pair separator, defaults to '&'.
]) -> String
```

Examples

```
SELECT Url::QueryStringToList("a=1&b=2&a=3");                     -- [("a", "1"), ("b", "2"), ("a", "3")]
SELECT Url::QueryStringToDict("a=1&b=2&a=3");                     -- {"b" : ["2"], "a" : ["1", "3"]}
SELECT Url::BuildQueryString([("a", "1"), ("a", "3"), ("b", "2")]); -- "a=1&a=3&b=2"
SELECT Url::BuildQueryString({"a" : "1", "b" : "2"});             -- "b=2&a=1"
SELECT Url::BuildQueryString({"a" : ["1", "3"], "b" : ["2", "4"]}); -- "b=2&b=4&a=1&a=3"
```

Yson

YSON is a JSON-like data format developed at Yandex.

- Similarities with JSON:
  - Does not have a strict scheme.
  - Besides simple data types, it supports dictionaries and lists in arbitrary combinations.
- Some differences from JSON:
  - It also has a binary representation in addition to the text representation.
  - The text representation uses semicolons instead of commas and equal signs instead of colons.
- The concept of "attributes" is supported, that is, named properties that can be assigned to a node in the tree.

Implementation specifics and functionality of the module:

- Along with YSON, this module also supports standard JSON to expand the application scope in a way.
- It works with a DOM representation of YSON in memory that in YQL terms is passed between functions as a "resource" (see the description of special data types). Most of the module's functions have the semantics of a query to perform a specified operation with a resource and return an empty optional type if the operation failed because the actual data type mismatched the expected one.
- Provides several main classes of functions (find below a complete list and detailed description of functions):
  - `Yson::Parse***`: Getting a resource with a DOM object from serialized data, with all further operations performed on the obtained resource.
  - `Yson::From`: Getting a resource with a DOM object from simple YQL data types or containers (lists or dictionaries).
  - `Yson::ConvertTo***`: Converting a resource to primitive data types or containers.
  - `Yson::Lookup***`: Getting a single list item or a dictionary with optional conversion to the relevant data type.
  - `Yson::YPath***`: Getting one element from the document tree based on the relative path specified, optionally converting it to the relevant data type.
  - `Yson::Serialize***`: Getting a copy of data from the resource and serializing the data in one of the formats.
- For convenience, when serialized Yson and Json are passed to functions expecting a resource with a DOM object, implicit conversion using `Yson::Parse` or `Yson::ParseJson` is done automatically. In SQL syntax, the dot or square brackets operator automatically adds a `Yson::Lookup` call. To serialize a resource, you still need to call `Yson::ConvertTo***` or `Yson::Serialize***`. It means that, for example, to get the "foo" element as a string from the Yson column named mycolumn and serialized as a dictionary, you can write: `SELECT Yson::ConvertToString(mycolumn["foo"]) FROM mytable;` or `SELECT Yson::ConvertToString(mycolumn.foo) FROM mytable;`. In the variant with a dot, special characters can be escaped by general rules for IDs.

The module's functions must be considered as "building blocks" from which you can assemble different structures, for example:

- `Yson::Parse*** -> Yson::Serialize***`: Converting from one format to other.
- `Yson::Parse*** -> Yson::Lookup -> Yson::Serialize***`: Extracting the value of the specified subtree in the source YSON tree.
- `Yson::Parse*** -> Yson::ConvertToList -> ListMap -> Yson::Lookup***`: Extracting items by a key from the YSON list.

Examples

```
$node = Json(@@
  {"abc": {"def": 123, "ghi": "hello"}}
@@);
SELECT Yson::SerializeText($node.abc) AS `yson`;
-- {"def"=123;"ghi"="\xD0\xBF\xD1\x80\xD0\xB8\xD0\xB2\xD0\xB5\xD1\x82"}
```

```
$node = Yson(@@
  <a=z;x=y>[
    {abc=123; def=456};
    {abc=234; xyz=789};
  ]
@@);
$attrs = Yson::YPath($node, "/@");

SELECT
  ListMap(Yson::ConvertToList($node), ($x) -> { return Yson::LookupInt64($x, "abc") }) AS abcs,
  Yson::ConvertToStringDict($attrs) AS attrs,
  Yson::SerializePretty(Yson::Lookup($node, "7", Yson::Options(false AS Strict))) AS miss;

/*
- abcs: `[123; 234]`
- attrs: `{"a"="z";"x"="y"}`
- miss: `NULL`
*/
```

Yson::Parse...

```
Yson::Parse(Yson{Flags:AutoMap}) -> Resource<'Yson2.Node'>
Yson::ParseJson(Json{Flags:AutoMap}) -> Resource<'Yson2.Node'>
Yson::ParseJsonDecodeUtf8(Json{Flags:AutoMap}) -> Resource<'Yson2.Node'>

Yson::Parse(String{Flags:AutoMap}) -> Resource<'Yson2.Node'>? -- accepts YSON in any format
Yson::ParseJson(String{Flags:AutoMap}) -> Resource<'Yson2.Node'>?
Yson::ParseJsonDecodeUtf8(String{Flags:AutoMap}) -> Resource<'Yson2.Node'>?
```

The result of all three functions is non-serializable: it can only be passed as the input to other function from the Yson library. However, you can't save it to a table or return to the client as a result of the operation: such an attempt results in a typing error. You also can't

return it outside subqueries: if you need to do this, call Yson::Serialize, and the optimizer will remove unnecessary serialization and deserialization if materialization isn't needed in the end.

> **Note**
>
> The `Yson::ParseJsonDecodeUtf8` expects that characters outside the ASCII range must be additionally escaped.

Yson::From

```
Yson::From(T) -> Resource<'Yson2.Node'>
```

`Yson::From` is a polymorphic function that converts most primitive data types and containers (lists, dictionaries, tuples, structures, and so on) into a Yson resource. The source object type must be Yson-compatible. For example, in dictionary keys, you can only use the `String` or `Utf8` data types, but not `String?` or `Utf8?` .

Example

```
SELECT Yson::Serialize(Yson::From(TableRow())) FROM table1;
```

Yson::WithAttributes

```
Yson::WithAttributes(Resource<'Yson2.Node'>{Flags:AutoMap}, Resource<'Yson2.Node'>{Flags:AutoMap}) -> Resourc
e<'Yson2.Node'>?
```

Adds attributes (the second argument) to the Yson node (the first argument). The attributes must constitute a map node.

Yson::Equals

```
Yson::Equals(Resource<'Yson2.Node'>{Flags:AutoMap}, Resource<'Yson2.Node'>{Flags:AutoMap}) -> Bool
```

Checking trees in memory for equality. The operation is tolerant to the source serialization format and the order of keys in dictionaries.

Yson::GetHash

```
Yson::GetHash(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Uint64
```

Calculating a 64-bit hash from an object tree.

Yson::Is...

```
Yson::IsEntity(Resource<'Yson2.Node'>{Flags:AutoMap}) -> bool
Yson::IsString(Resource<'Yson2.Node'>{Flags:AutoMap}) -> bool
Yson::IsDouble(Resource<'Yson2.Node'>{Flags:AutoMap}) -> bool
Yson::IsUint64(Resource<'Yson2.Node'>{Flags:AutoMap}) -> bool
Yson::IsInt64(Resource<'Yson2.Node'>{Flags:AutoMap}) -> bool
Yson::IsBool(Resource<'Yson2.Node'>{Flags:AutoMap}) -> bool
Yson::IsList(Resource<'Yson2.Node'>{Flags:AutoMap}) -> bool
Yson::IsDict(Resource<'Yson2.Node'>{Flags:AutoMap}) -> bool
```

Checking that the current node has the appropriate type. The Entity is `#`.

Yson::GetLength

```
Yson::GetLength(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Uint64?
```

Getting the number of elements in a list or dictionary.

Yson::ConvertTo...

```
Yson::ConvertTo(Resource<'Yson2.Node'>{Flags:AutoMap}, Type<T>) -> T
Yson::ConvertToBool(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Bool?
Yson::ConvertToInt64(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Int64?
Yson::ConvertToUint64(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Uint64?
Yson::ConvertToDouble(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Double?
Yson::ConvertToString(Resource<'Yson2.Node'>{Flags:AutoMap}) -> String?
Yson::ConvertToList(Resource<'Yson2.Node'>{Flags:AutoMap}) -> List<Resource<'Yson2.Node'>>
Yson::ConvertToBoolList(Resource<'Yson2.Node'>{Flags:AutoMap}) -> List<Bool>
Yson::ConvertToInt64List(Resource<'Yson2.Node'>{Flags:AutoMap}) -> List<Int64>
Yson::ConvertToUint64List(Resource<'Yson2.Node'>{Flags:AutoMap}) -> List<Uint64>
Yson::ConvertToDoubleList(Resource<'Yson2.Node'>{Flags:AutoMap}) -> List<Double>
Yson::ConvertToStringList(Resource<'Yson2.Node'>{Flags:AutoMap}) -> List<String>
Yson::ConvertToDict(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Dict<String,Resource<'Yson2.Node'>>
Yson::ConvertToBoolDict(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Dict<String,Bool>
Yson::ConvertToInt64Dict(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Dict<String,Int64>
Yson::ConvertToUint64Dict(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Dict<String,Uint64>
Yson::ConvertToDoubleDict(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Dict<String,Double>
Yson::ConvertToStringDict(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Dict<String,String>
```

Warning

These functions do not do implicit type casting by default, that is, the value in the argument must exactly match the function called.

`Yson::ConvertTo` is a polymorphic function that converts the data type that is specified in the second argument and supports containers (lists, dictionaries, tuples, structures, and so on) into a Yson resource.

Example

```
$data = Yson(@@{
    "name" = "Anya";
    "age" = 15u;
    "params" = {
        "ip" = "95.106.17.32";
        "last_time_on_site" = 0.5;
        "region" = 213;
        "user_agent" = "Mozilla/5.0"
    }
}@@);
SELECT Yson::ConvertTo($data,
    Struct<
        name: String,
        age: Uint32,
        params: Dict<String,Yson>
    >
);
```

Yson::Contains

```
Yson::Contains(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> Bool?
```

Checks for a key in the dictionary. If the object type is a map, then it searches among the keys.
If the object type is a list, then the key must be a decimal number, i.e., an index in the list.

Yson::Lookup...

```
Yson::Lookup(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> Resource<'Yson2.Node'>?
Yson::LookupBool(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> Bool?
Yson::LookupInt64(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> Int64?
Yson::LookupUint64(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> Uint64?
Yson::LookupDouble(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> Double?
Yson::LookupString(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> String?
Yson::LookupDict(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> Dict<String,Resource<'Yson2.Node'>>?
Yson::LookupList(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> List<Resource<'Yson2.Node'>>?
```

The above functions are short notations for a typical use case: `Yson::YPath`: go to a level in the dictionary and then extract the value — `Yson::ConvertTo***`. For all the listed functions, the second argument is a key name from the dictionary (unlike YPath, it has no `/` prefix) or an index from the list (for example, `7`). They simplify the query and produce a small gain in speed.

Yson::YPath

```
Yson::YPath(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> Resource<'Yson2.Node'>?
Yson::YPathBool(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> Bool?
Yson::YPathInt64(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> Int64?
Yson::YPathUint64(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> Uint64?
Yson::YPathDouble(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> Double?
Yson::YPathString(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> String?
Yson::YPathDict(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> Dict<String,Resource<'Yson2.Node'>>?
Yson::YPathList(Resource<'Yson2.Node'>{Flags:AutoMap}, String) -> List<Resource<'Yson2.Node'>>?
```

Lets you get a part of the resource based on the source resource and the part's path in YPath format.

Yson::Attributes

```
Yson::Attributes(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Dict<String,Resource<'Yson2.Node'>>
```

Getting all node attributes as a dictionary.

Yson::Serialize...

```
Yson::Serialize(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Yson -- A binary representation
Yson::SerializeText(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Yson
Yson::SerializePretty(Resource<'Yson2.Node'>{Flags:AutoMap}) -> Yson -- To get a text result, wrap it in ToBy
tes(...)
```

Yson::SerializeJson

```
Yson::SerializeJson(Resource<'Yson2.Node'>{Flags:AutoMap}, [Resource<'Yson2.Options'>?, SkipMapEntity:Bool?,
EncodeUtf8:Bool?, WriteNanAsString:Bool?]) -> Json?
```

- `SkipMapEntity` serializes `#` values in dictionaries. The value of attributes is not affected by the flag. By default, `false`.

- `EncodeUtf8` responsible for escaping non-ASCII characters. By default, `false`.
- `WriteNanAsString` allows serializing `NaN` and `Inf` values as strings. By default, `false`.

The `Yson` and `Json` data types returned by serialization functions are special cases of a string that is known to contain data in the given format (Yson/Json).

Yson::Options

```
Yson::Options([AutoConvert:Bool?, Strict:Bool?]) -> Resource<'Yson2.Options'>
```

It's passed in the last optional argument (omitted for brevity) to the methods `Parse...`, `ConvertTo...`, `Contains`, `Lookup...`, and `YPath...` that accept the result of the `Yson::Options` call. By default, all the `Yson::Options` fields are false and when enabled (true), they modify the behavior as follows:

- **AutoConvert**: If the value passed to Yson doesn't match the result data type exactly, the value is converted where possible. For example, `Yson::ConvertToInt64` in this mode will convert even Double numbers to Int64.
- **Strict**: By default, all functions from the Yson library return an error in case of issues during query execution (for example, an attempt to parse a string that is not Yson/Json, or an attempt to search by a key in a scalar type, or when a conversion to an incompatible data type has been requested, and so on). If you disable the strict mode, `NULL` is returned instead of an error in most cases. When converting to a dictionary or list (`ConvertTo<Type>Dict` or `ConvertTo<Type>List`), improper items are excluded from the resulting collection.

Example

```
$yson = @@{y = true; x = 5.5}@@y;
SELECT Yson::LookupBool($yson, "z"); --- null
SELECT Yson::LookupBool($yson, "y"); --- true

SELECT Yson::LookupInt64($yson, "x"); --- Error
SELECT Yson::LookupInt64($yson, "x", Yson::Options(false as Strict)); --- null
SELECT Yson::LookupInt64($yson, "x", Yson::Options(true as AutoConvert)); --- 5

SELECT Yson::ConvertToBoolDict($yson); --- Error
SELECT Yson::ConvertToBoolDict($yson, Yson::Options(false as Strict)); --- { "y": true }
SELECT Yson::ConvertToDoubleDict($yson, Yson::Options(false as Strict)); --- { "x": 5.5 }
```

If you need to use the same Yson library settings throughout the query, it's more convenient to use PRAGMA yson.AutoConvert; and/or PRAGMA yson.Strict;. Only with these `PRAGMA` you can affect implicit calls to the Yson library occurring when you work with Yson/Json data types.

See also

- Accessing values inside JSON with YQL
- Modifying JSON with YQL

Query plan structure

To understand how a query will be executed, you can build and analyze its plan. The query plan structure in YDB is represented as a graph, where each node contains information about operations and tables.

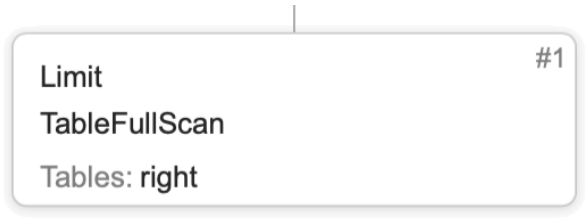
Below, you can find information about node types, and an example of analyzing a specific query plan can be found here.

Node types

Stage

Query execution stage.

UI representation



A stage can contain the following operations:

TableFullScan

Full table scan. This operation's resource consumption is proportional to the table size, so it should be avoided whenever possible.

Attribute	Description
Table	table name
ReadColumns	read columns list
ReadLimit	read rows limit
Reverse	flag indicating the order in which the rows will be read, by default the order is forward (ascending), but if the flag is set to `true`, the reading order will be reversed (descending).
Parallel	flag indicating that rows will be read from shards in parallel

TableRangeScan

Reading a table by a specific primary key range.

Attribute	Description
Table	table name
ReadColumns	read columns list
ReadRange	key range
ReadLimit	read rows limit
Reverse	flag indicating the order in which the rows will be read, by default the order is forward (ascending), but if the flag is set to `true`, the reading order will be reversed (descending).
Parallel	flag indicating that rows will be read from shards in parallel

TablePointLookup

Reading a table by specific primary key values. Note that for this operation, all components of the primary key should be specified. Reading by a key prefix is performed as a `TableRangeScan` operation.

Attribute	Description
Table	table name
ReadColumns	read columns list

Upsert

Updates or inserts multiple rows to a table based on a comparison by the primary key. The values of the specified columns are updated for the existing rows, but the values of the other columns are preserved.

Attribute	Description
Table	table name
Columns	columns contained in the row

Delete

Deleting rows from the table.

Attribute	Description
Table	table name

Join

Combine two data sources (subqueries or tables) by keys. The join strategy is specified in the operation description.

Filter

Filtering rows, keeping only those for which a predicate returns `true`.

Attribute	Description
Predicate	filtering condition
Limit	rows limit

Aggregate

Grouping rows by the values of the specified columns or expressions.

Attribute	Description
GroupBy	columns or expressions used for aggregation
Aggregation	aggregate function

Sort

Sorting rows.

Attribute	Description
SortBy	columns or expressions used for sorting

TopSort

Partial rows sorting with a specified limit.

Attribute	Description
TopSortBy	columns or expressions used for sorting
Limit	rows limit

Top

Returns the first N elements that are less or equal to the N+1 element if the entire sequence were sorted.

Attribute	Description
TopBy	columns or expressions by which the first N rows will be taken
Limit	rows limit

Limit

Limit on the number of rows.

Attribute	Description
Limit	limit value

Offset

Offset, allowing to skip the first N elements of a given set of rows.

Attribute	Description
Offset	offset value

Union

Concatenate the results of two or more subqueries into a single row set.

Iterator

Iterates through a given set of rows. Typically uses a precompute as an argument.

PartitionByKey

Partitioning by key. Typically uses a precompute.

Connection

Data dependencies between stages.

UI representation

Each stage is executed as a certain number of tasks. For example, a reading stage may be executed in N tasks, where N is the number of table shards. The method of transferring data between stages depends on the type of connection. Below is a description of different connections.

UnionAll

Combines the results of all producer stage tasks and sends them as a single result to a single consumer stage task.

Merge

This is a special case of `UnionAll`. The results of the producer stage are sorted by a specified set of columns and merged into a result that is also sorted.

Broadcast

Sends the result of a single producer stage task to all consumer stage tasks.

Map

Implements 1-to-1 relationships between tasks of stages, the producer and consumer stages should have the same number of tasks.

HashShuffle

Sends the results of producer stage tasks to consumer stage tasks based on a certain rule for specified columns. The rule is defined in the code, but the list of columns is specified in the connection.

ResultSet

The query execution result.

UI representation

Precompute

Materialized intermediate result.

UI representation

Stages that depend on precomputes should be executed after the precompute has been completed.

Using YDB UI

YDB provides tools for monitoring and determining system health:

- YDB Monitoring: The main monitor of the cluster. It shows the health of nodes and storage groups.
- Interconnect overview: The state of cluster interconnects.
- Logs: Each YDB component writes messages to logs of different levels. They can be used to detect severe issues or identify the root causes of issues.
- Charts: YDB collects a range of metrics that can be used to determine the health of the entire system or of a specific component.

Using YDB UI

YDB provides tools for monitoring and determining system health:

- YDB Monitoring: The main monitor of the cluster. It shows the health of nodes and storage groups.
- Interconnect overview: The state of cluster interconnects.
- Logs: Each YDB component writes messages to logs of different levels. They can be used to detect severe issues or identify the root causes of issues.
- Charts: YDB collects a range of metrics that can be used to determine the health of the entire system or of a specific component.

YDB Monitoring

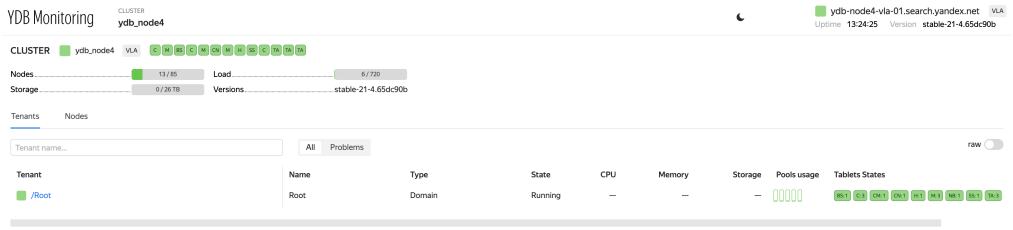
YDB Monitoring is a set of web pages that display the system health according to a range of different aspects. The pages contain lists of components and their current parameters. Many components have a health color indicator before the name.

Home page

The page is available at:

```
http://<endpoint>:8765/monitoring/cluster
```

An example of the page layout is shown in the screenshot below.



In the upper-right corner of the page, you can see details about the node that created the current page:

- host.
- node uptime.
- ydb version run by the node.

Below is the cluster summary:

- cluster name, list of running tablets.
- **Nodes**: The number of nodes.
- **Load**: The total CPU utilization level on all nodes.
- **Storage**: The used/total storage space.
- **Versions**: The list of YDB versions run on the cluster nodes.

Next, you will find lists of tenants and nodes on the Tenants and Nodes tabs, respectively.

Tenant list

YDB is a multi-tenant DBMS that lets you execute isolated queries against databases of different users called tenants. Each database belongs to one tenant.

The tenants list contains the following information on each tenant:

- **Tenant**: Tenant path.
- **Name**: Tenant name.
- **Type**: Tenant type.
- **State**: Tenant health.
- **CPU**: CPU utilization by the tenant's nodes.
- **Memory**: RAM consumption by the tenant's nodes.
- **Storage**: Estimated amount of data stored by the tenant.
- **Pools usage**: CPU usage by the nodes broken down by the internal stream pools.
- **Tablets States**: Tablets running in the given tenant.

`Domain` type tenants serve the system components needed to ensure all tenants can operate. This includes all storage nodes and system tablets. `Dedicated` type tenants perform database maintenance.

If you click on the tenant's path, you can go to the tenant page.

Node list

The list includes all nodes in the cluster. For each node, you can see:

- #: Node ID.
- **Host**: The host running the node.
- **Endpoints**: The ports being listened to.
- **Version**: The YDB version being run.
- **Uptime**: The node uptime.
- **Memory used**: The amount of RAM used.
- **Memory limit**: The RAM utilization limit set via cgroup.
- **Pools usage**: CPU utilization broken down by the internal stream pools.
- **Load average**: The average CPU utilization on the host.

To open the node page, click the host name.

Node page

The page is available at:

```
http://<endpoint>:8765/monitoring/node/<node-id>/
```

Information about the node is presented in the following sections:

- **Pools**: CPU utilization broken down by the internal stream pools, with roughly the following pool functions:

    - **System**: The tasks of critical system components.
    - **User**: User tasks, queries executed by tablets.
    - **Batch**: Long-running background tasks.
    - **IO**: Blocking I/O operations.
    - **IC**: Networking operations.

  High pool utilization might degrade performance and increase the system response time.
- **Common info**: Basic information about the node:
    - **Version**: The YDB version.
    - **Uptime**: The node uptime.
    - **DC**: The availability zone where the node resides.
    - **Rack**: The ID of the rack where the node resides.
- **Load average**: Average host CPU utilization for different time intervals:
    - 1 minute.
    - 5 minutes.
    - 15 minutes.

The node page has the Storage and Tablets tabs with a list of storage groups and a list of tablets, respectively.

List of storage groups on the node

For storage nodes, the list includes storage groups that store data on this node's disk. For dynamic nodes, the list shows the storage groups of the tenant that the node belongs to.

Storage groups are grouped by storage pools. Each pool can be expanded into a list of groups.

For each group, the following information is provided:

- The ID of the storage group.
- The performance indicator of the group.
- The number of VDisks in the group.
- The data storage topology.

Each storage group can also be expanded into a list of VDisks, with the following information provided for each VDisk:

- VDiskID.
- The unique (within the node) PDiskID where the VDisk resides.
- The ID of the node where the VDisk resides.
- Free/available space on the block store volume.
- Path used to access block storage.

This list can be used to identify storage groups that were affected by disk or node failure.

List of tablets residing on the node

Many YDB components are implemented as tablets. The system can move tablets between nodes. A certain number of tablets can be run on any node.

In the upper part of the list of tablets, there's a big indicator for tablets running on the given node. It shows the ratio of fully launched and running tablets.

Under the indicator, you can see a list of tablets, where each tablet is shown as a small color indicator icon. When you hover over the indicator, the tablet summary is shown:

- **Tablet**: The ID of the tablet.
- **NodeID**: The ID of the node where the tablet resides.
- **State**: The state of the tablet.
- **Type**: The type of tablet.
- **Uptime**: The uptime since the tablet was last launched.
- **Generation**: The tablet's generation (the number of the current launch attempt).

Tenant page

```
http://<endpoint>:8765/monitoring/tenant/healthcheck?name=<tenant-path>
```

Like the previous pages, this page includes the tenant summary, but unlike the other pages, this section is initially collapsed.

In the `Tenant Info` section, you can see the following information:

- **Pools**: Total CPU utilization by the tenant nodes broken down by internal stream pools (for more information about pools, see the tenant page).
- **Metrics**: Data about tablet utilization for this tenant:
    - **Memory**: The RAM utilized by tablets.
    - **CPU**: CPU utilized by tablets.
    - **Storage**: The amount of data stored by tablets.
    - **Network**: The estimated amount of data transferred between nodes.
    - **Read throughput**: The read stream created by the tenant's tablets.
    - **Write throughput**: The write stream created by the tenant's tablets.
- **Tablets/running**: The number of running tablets.

The tenant page also includes the following tabs:

- **HealthCheck**: The report regarding cluster issues, if any.
- **Storage**: List of storage groups showing the nodes and block store volumes hosting each VDisk.
- **Compute**: List of nodes showing the nodes and tablets running on them.

- **Schema**: Tenant schema that enables you to view tables, execute YQL queries, view a list of the slowest queries and the most loaded shards.
- **Network**: Cluster network health.

List of storage groups in the tenant

Similarly to the list of groups in the node, this page shows a list of storage groups in the given tenant. Storage groups are grouped by storage pools in the list. Each pool can be expanded into a list of groups.

List of nodes belonging to the tenant

Here you can see a list of nodes belonging to the current tenant. If the tenant has the domain type, the list includes storage nodes.

Each node is represented by the following parameters:

- #: Node ID.
- **Host**: The host running the node.
- **Uptime**: The node uptime.
- **Endpoints**: The ports being listened to.
- **Version**: The YDB version being run.
- **Pools usage**: CPU utilization broken down by the internal stream pools.
- **CPU**: CPU utilization by tablets.
- **Memory**: RAM consumption by tablets.
- **Network**: Network usage by tablets.
- **Tablets**: A list of tablets running on the node, grouped by type.

Tenant schema

The left part of the page shows a diagram of tenant objects where you can select specific tables and view their details.

The right part includes the tabs:

- **Info**: Information about the table and its schema.
- **Preview**: A preview of the first items in the table.
- **Graph**: A range of built-in graphs with information about the table status.
- **Describe**: The result of running the describe command.
- **Query**: The form for executing YQL queries.
- **Tablets**: The list of tablets serving the table.
- **ACL**: The Access Control List.
- **Top queries**: The top most time-consuming queries to the table.
- **Top shards**: The top most loaded table shards.

Network health

On the left side of the page, you can see the tenant's nodes as squares of health indicators.

Whenever you select a node, the right side of the screen shows details about the health of the network connection between this node and other nodes.

Whenever you select the ID and Racks checkboxes, you can also see the IDs of nodes and their location in racks.

Monitoring static groups

To perform a health check on a static group, go to the **Storage** panel. By default, it shows a list of groups with issues.

Enter `static` in the search bar. If the result is empty, no issues have been found for the static group. But if the panel shows a **static** group, check the VDisk health status in it. Acceptable health indicators are green (no issues) and blue (VDisk replication is in progress). Red indicator signals of an issue. Hover over the indicator to get a text description of the issue.

Health indicators

To the left of a component name, you might see a color indicating its health status.

The indicator colors have the following meaning:

- **Green**: There are no problems, the component is working normally.
- **Blue**: Data replication is in progress, no other issues observed.
- **Yellow**: There might be problems, the component is still running.
- **Red**: There are critical problems, the component is down (or runs with limitations).

If a component includes other components, then in the absence of its own issues, the state is determined by aggregating the states of its parts.

Hive web-viewer

The Hive web-viewer provides an interface for working with Hive.
Hive can be shared by a cluster or be tenant.
You can go to the Hive web-viewer from the YDB Monitoring.

Home page

The home page provides information about the distribution and usage of resources by tablets on each node in the form of a table.

The table is preceded by the following brief information:

- **Tenant**: The tenant that Hive is responsible for.
- **Tablets**: The percentage of tablets started and the number of running tablets to their total number.
- **Boot Queue**: The number of tablets in the boot queue.
- **Wait Queue**: The number of tablets that can't be started.
- **Resource Total**: Resource utilization by tablets (cpu, net).
- **Resource StDev**: Standard deviation of resource utilization (cnt, cpu, mem, net).

Then there is a table where each row represents a node managed by Hive. It has the following columns:

- **Node**: The node number.
- **Name**: The node FQDN and ic-port.
- **DC**: The datacenter where the node resides.
- **Domain**: The node tenant.
- **Uptime**: The node uptime.
- **Unknown**: The number of tablets whose state is unknown.
- **Starting**: The number of tablets being started.
- **Running**: The number of tablets running.
- **Types**: Tablet distribution by type.
- **Usage**: A normalized dominant resource.
- **Resources**:
  - **cnt**: The number of tablets that are not using any resources.
  - **cpu**: CPU usage by tablets.
  - **mem**: RAM usage by tablets.
  - **net**: Bandwidth usage by tablets.
- **Active**: Enables/disables the node to move tablets to this node.
- **Freeze**: Disables tablets to be deployed on other nodes.
- **Kick**: Moves all tablets from the node at once.
- **Drain**: Smoothly moves all tablets from the node.

Additional pages are presented below the table:

- **Bad tablets**: A list of tablets having issues or errors.
- **Heavy tablets**: A list of tablets utilizing a lot of resources.
- **Waiting tablets**: A list of tablets that can't be started.
- **Resources**: Resource utilization by each tablet.
- **Tenants**: A list of tenants indicating their local Hive tablets.
- **Nodes**: A list of nodes.
- **Storage**: A list of storage group pools.
- **Groups**: A list of storage groups for each tablet.
- **Settings** {#settings}: The Hive configuration page.
- **Reassign Groups**: The page for reassigning storage groups across tablets.

You can also view what tablets use a particular group and, vice versa, what groups are used on a particular tablet.

Reassign Groups

Click **Reassign Groups** to open the window with parameters for balancing:

- **Storage pool**: Pool of storage groups for balancing.
- **Storage group**: If the previous item is not specified, you can specify only one group separately.
- **Type**: Type of tablets that balancing will be performed for.
- **Channels**: Range of channels that balancing will be performed for.
- **Percent**: Percentage of the total number of tablet channels that will be moved as a result of balancing.
- **Inflight**: The number of tablets being moved to other groups at the same time.

After specifying all the parameters, click "Query" to get the number of channels moved and unlock the "Reassign" button.
Clicking this button starts reassigning.

Interconnect Overview

Overview of cluster node interconnects. Available at:

```
http://<endpoint>:8765/actors/interconnect/overview
```

Shows, for every other node:

- ping
- system clock difference
- connection availability
- last written error

Logs

Logging levels

Level	Numeric value	Value
TRACE	8	Very detailed debugging information.
DEBUG	7	Debugging information for developers.
INFO	6	Debugging information for collecting statistics.
NOTICE	5	An event essential for the system or the user has occurred.
WARN	4	This is a warning, it should be responded to and fixed unless it's temporary.
ERROR	3	A non-critical error.
CRIT	2	A critical state.
ALERT	1	System degradation is possible, system components may fail.
EMERG	0	System outage (for example, cluster failure) is possible.

The logging level for different YDB components can be configured individually. For each component, either an explicitly set value or a default logging level value can be applied. The default logging level value can also be changed.

Changing the logging level

To change the logging level:

1. Follow the link in the format

   ```
   http://<endpoint>:8765/cms
   ```

   The `Cluster Management System` page opens.
2. On the **Configs** tab, click on the `LogConfigItems` line. The `Create new item` button will show up along with a list of already created configuration elements.
3. Click `Create new item` to create a new configuration item (or click the pencil button to edit a previously created item).
4. To change the default logging level, select the desired logging level from the `Level` drop-down list under `Default log settings`. The default global setting is `NOTICE`.
5. To change the logging level for individual components, use the table under `Component log settings`. In the line with the name of the component whose logging level you want to change, in the `Component` column, select the desired logging level from the drop-down list in the `Log level` column.
6. To save changes, click `Submit`

Charts

To view charts, use Grafana.

The main metrics of the system are displayed on the dashboard:

- **CPU Usage**: The total CPU utilization on all nodes (1 000 000 = 1 CPU).
- **Memory Usage**: RAM utilization by nodes.
- **Disk Space Usage**: Disk space utilization by nodes.
- **SelfPing**: The highest actual delivery time of deferred messages in the actor system over the measurement interval. Measured for messages with a 10 ms delivery delay. If this value grows, it might indicate microbursts of workload, high CPU utilization, or displacement of the YDB process from CPU cores by other processes.

Integrations YDB

This section provides the main information about YDB integrations with third-party systems.

> **Note**
>
> In addition to its own native protocol, YDB has a compatibility layer that allows external systems to connect to databases via the Apache Kafka network protocol. Due to the compatibility layer, many tools designed to work with Kafka can also interact with YDB. The compatibility level of each specific application needs to be clarified separately.

Graphical user interfaces

Environment	Instruction	Compatibility level
YDB UI	Instruction	
DBeaver	Instruction	By JDBC-driver
JetBrains Database viewer	—	By JDBC-driver
JetBrains DataGrip	Instruction	By JDBC-driver
Other JDBC-compatible IDEs	—	By JDBC-driver
Jupyter Notebook	Instruction	By YDB-SQLAlchemy

Data visualization (Business intelligence, BI)

Environment	Compatibility Level	Instruction
Apache Superset	ydb-sqlalchemy	Instruction
DataLens	Full	Instruction
Grafana	Full	Instruction

Orchestration

System	Instruction
Apache Airflow™	Instruction

Data ingestion

Delivery system	Guide
FluentBit	Guide
LogStash	Guide
Kafka Connect Sink	Guide

Streaming data ingestion

Delivery System	Instruction
Apache Kafka API	Instruction

Data migrations

Source	Guide
Arbitrary JDBC data sources	Guide
MySQL	Guide

Schema migrations

Environment	Instruction
goose	Instruction
Liquibase	Instruction
Flyway	Instruction
dbt	Instruction

Query engines

Engine	Guide
Apache Spark™	Guide

Object–relational mapping (ORM)

System	Instructions
Hibernate	Instructions
Spring Data JDBC	Instructions
Kotlin Exposed	Instructions
JOOQ	Instructions
Dapper	Instructions
Entity Framework	Instructions
Linq To DB	Instructions
SQLAlchemy	Instructions
Django	Instructions
Drizzle ORM	Instructions

Vector search

Tool	Guide
LangChain	Guide
Mem0	Guide

See also

- Reference for YDB SDK
- Kafka API

Graphical user interfaces

Environment	Instruction	Compatibility level
YDB UI	Instruction	
DBeaver	Instruction	By JDBC-driver
JetBrains Database viewer	—	By JDBC-driver
JetBrains DataGrip	Instruction	By JDBC-driver
Other JDBC-compatible IDEs	—	By JDBC-driver
Jupyter Notebook	Instruction	By YDB-SQLAlchemy

Connecting to YDB with DBeaver

DBeaver is a free, cross-platform, open-source database management tool that provides a visual interface for connecting to various databases and executing SQL queries. It supports many database management systems, including MySQL, PostgreSQL, Oracle, and SQLite.

DBeaver allows you to work with YDB using the Java DataBase Connectivity (JDBC) protocol. This article demonstrates how to set up this integration.

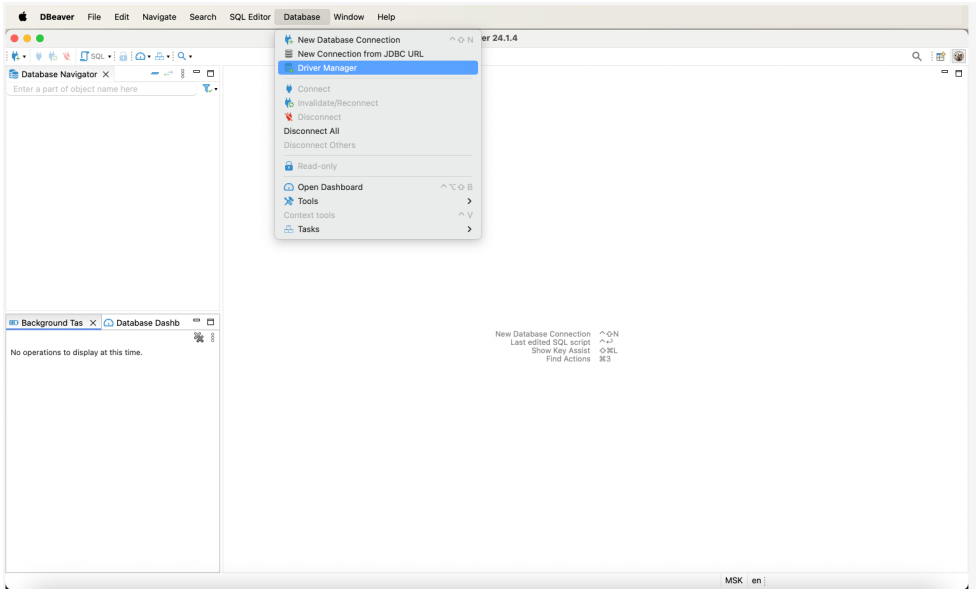
Connecting the YDB JDBC Driver to DBeaver

To connect to YDB from DBeaver, you will need the JDBC driver. Follow these steps to download the JDBC driver:

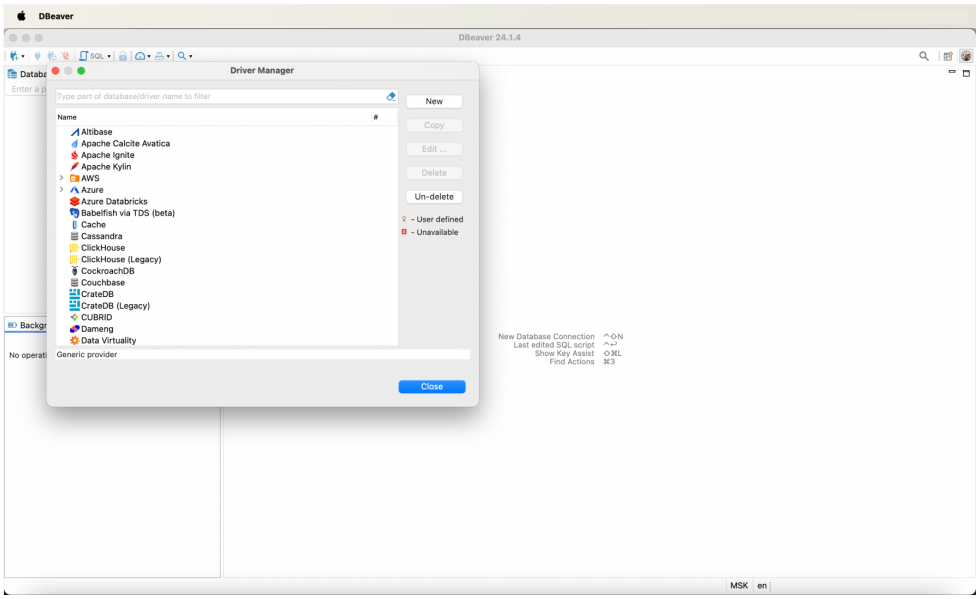
1. Go to the ydb-jdbc-driver repository.
2. Select the latest release (tagged as `Latest`) and save the `ydb-jdbc-driver-shaded-<driver-version>.jar` file.

Follow these steps to connect the downloaded JDBC driver:

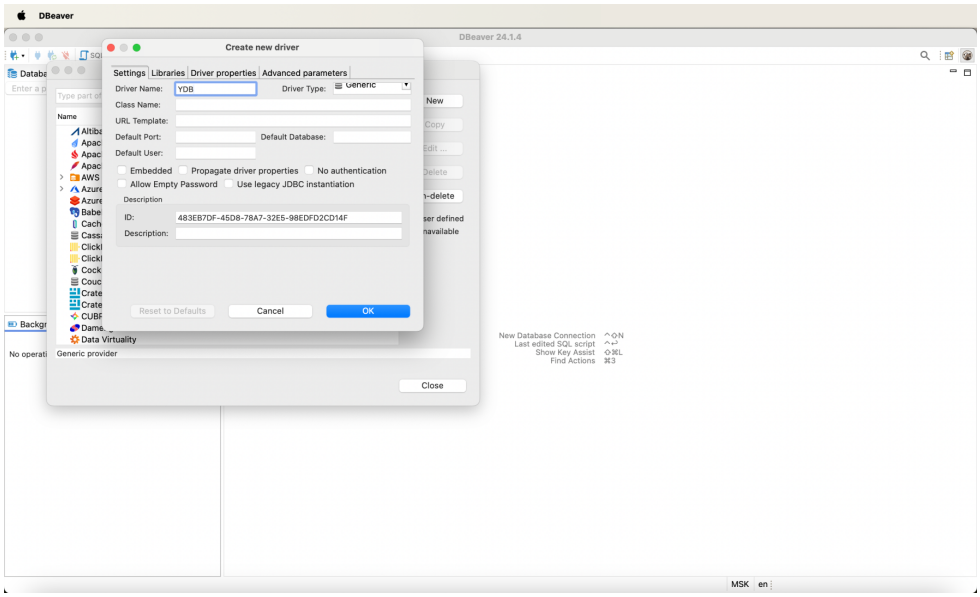
1. In the top menu of DBeaver, select the **Database** option, then select **Driver Manager**:



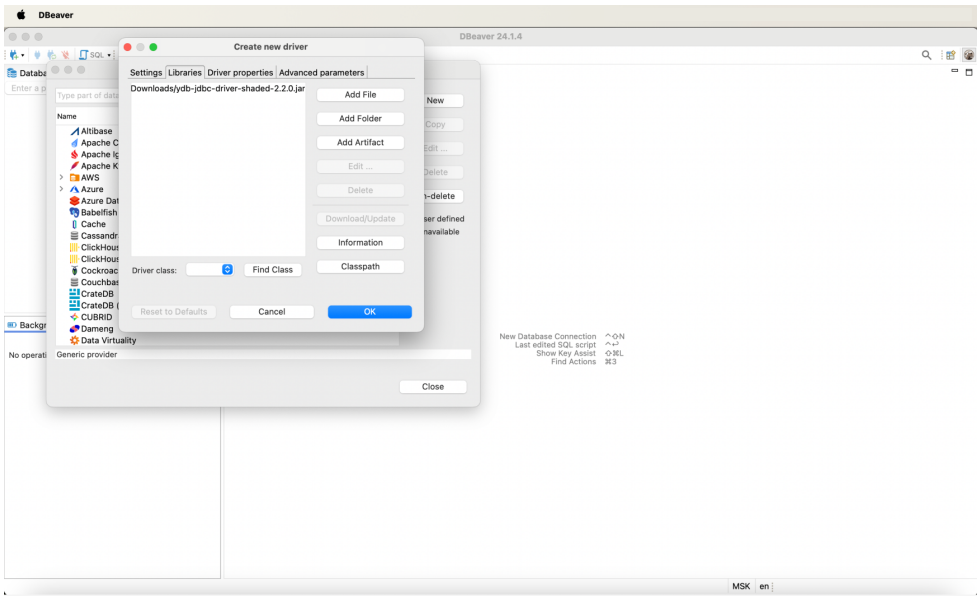
2. To create a new driver, click the **New** button in the **Driver Manager** window that opens



3. In the **Create Driver** window that opens, specify `YDB` in the **Driver Name** field:

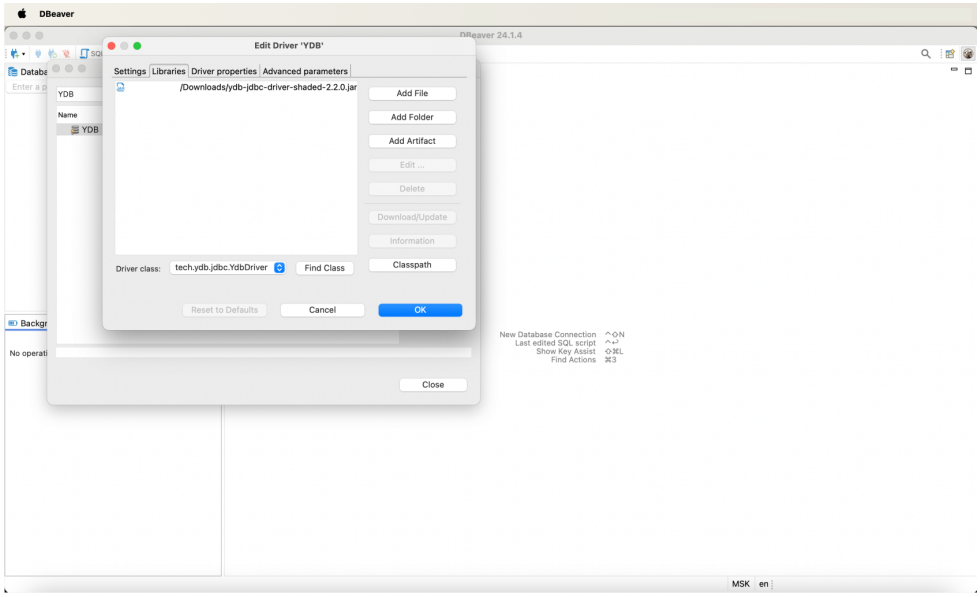


4. Go to the **Libraries** section, click **Add File**, specify the path to the previously downloaded YDB JDBC driver (the `ydb-jdbc-driver-shaded-<driver-version>.jar` file), and click **OK**:



5. The **YDB** item will appear in the list of drivers. Double-click the new driver and go to the **Libraries** tab, click **Find Class**, and select `tech.ydb.jdbc.YdbDriver` from the dropdown list.

> **Warning**
>
> Be sure to explicitly select the `tech.ydb.jdbc.YdbDriver` item from the dropdown list by clicking on it. Otherwise, DBeaver will consider that the driver has not been selected.



Creating a Connection to YDB

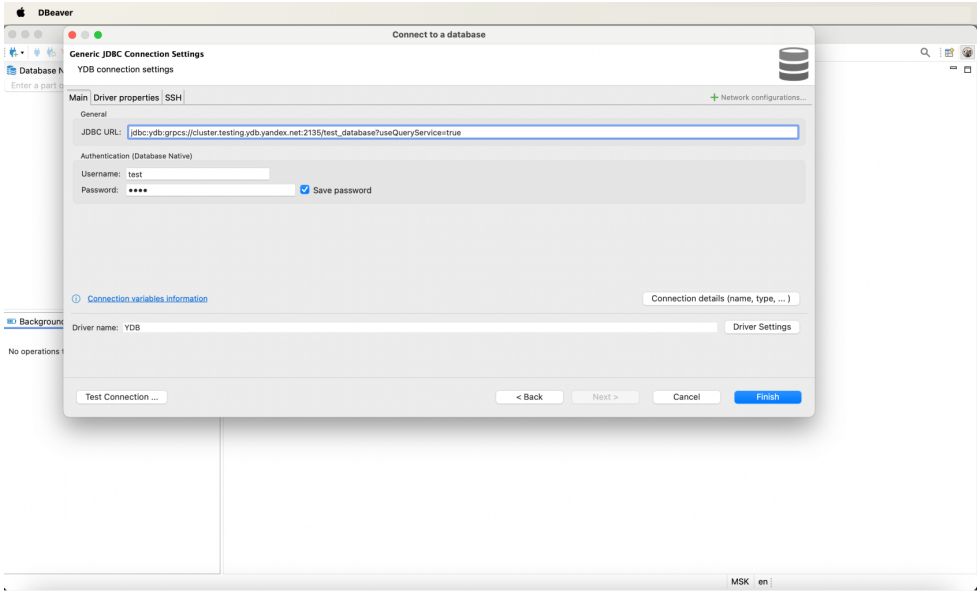
Perform the following steps to establish a connection:

1. In DBeaver, create a new connection, specifying the `YDB` connection type.
2. In the window that opens, go to the **Main** section.
3. In the **General** subsection, in the **JDBC URL** input field, specify the following connection string:

   ```
   jdbc:ydb:<ydb_endpoint>/<ydb_database>?useQueryService=true
   ```

   Where:

   - `ydb_endpoint` — the endpoint of the YDB cluster to which the connection will be made.
   - `ydb_database` — the path to the database in the YDB cluster to which queries will be made.

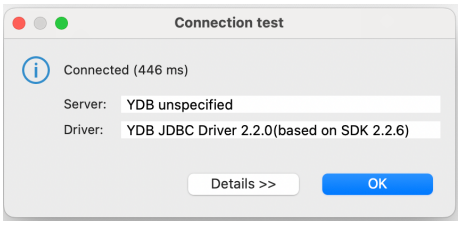


4. In the **User** and **Password** fields, enter the login and password for connecting to the database. A complete list of authentication methods and connection strings for YDB is provided in the JDBC driver description.

> **Note**
>
> In Managed installations of YDB login and password authentication is not available.

1. Click **Test Connection...** to verify the settings.

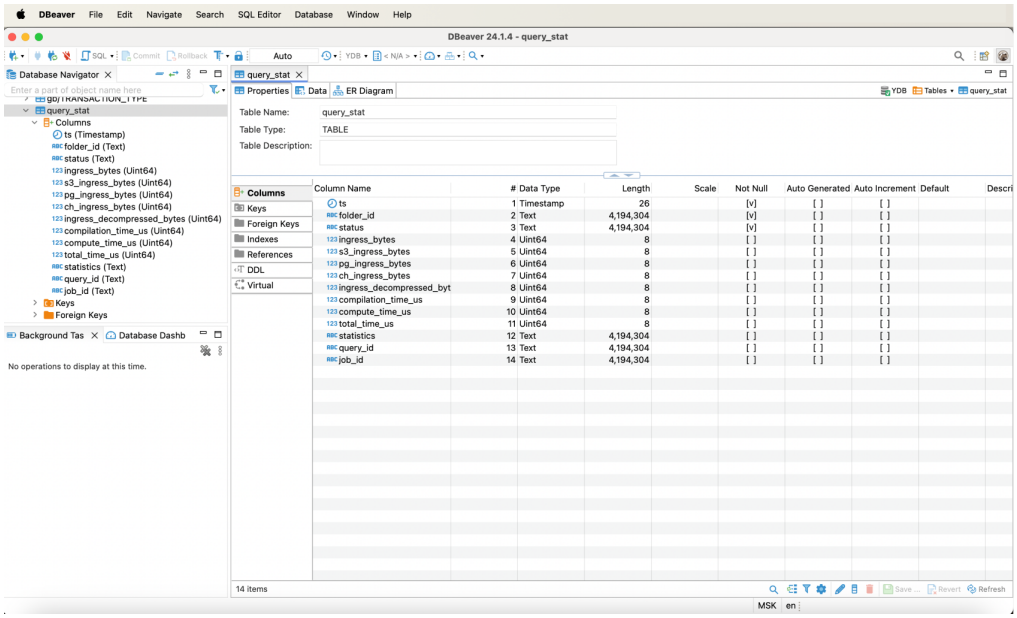
   If all settings are correct, a message indicating successful connection testing will appear:



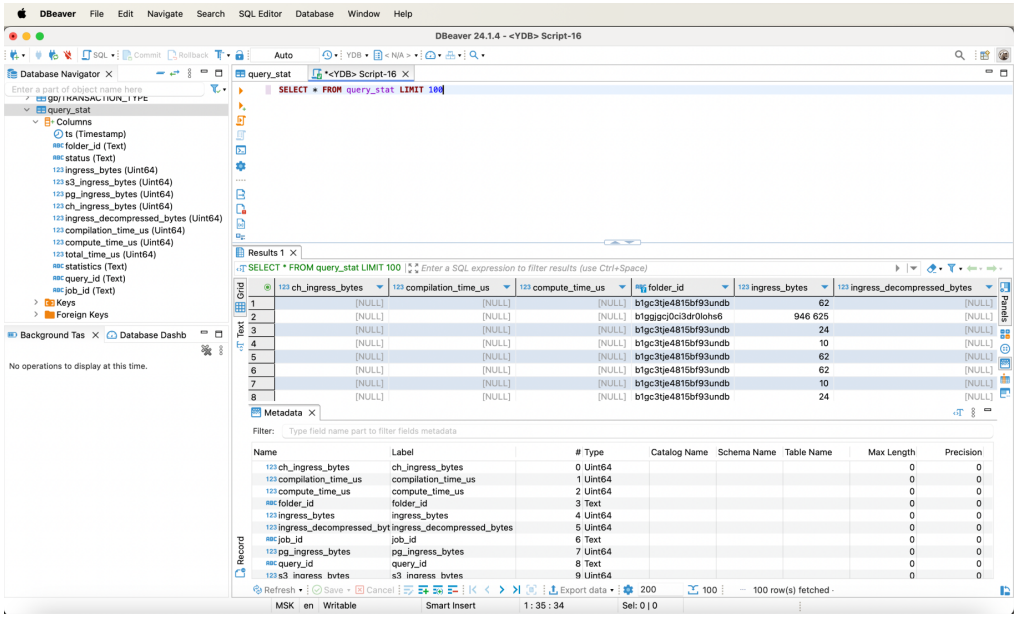
2. Click **Finish** to save the connection.

Working with YDB

With DBeaver, you can view the list and structure of tables:



As well as execute queries on the data:



Connecting to YDB Using the DBeaver Plugin

DBeaver is a free cross-platform open-source database management tool that provides a visual interface for connecting to various databases and executing SQL queries.

YDB DBeaver Plugin is a DBeaver extension with native support for YDB. Unlike connecting via a JDBC driver, the plugin provides a specialized interface for working with YDB objects: a hierarchical navigator for tables, topics, views, external data sources, support for all authentication methods, a YQL editor, query execution plan visualization, session and cluster monitoring, access control (ACL) management, and other features.

Key Features of the Plugin

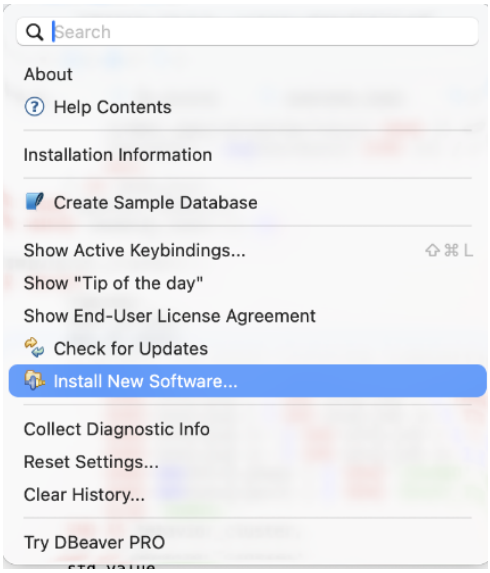
- Connecting to YDB with all authentication methods: anonymous, static, token-based, service account, metadata-based.
- Hierarchical object navigator: tables, topics, external data sources, external tables, views.
- System objects: system views (`.sys`), resource pools, and resource pool classifiers.
- YQL editor with syntax highlighting for over 150 keywords and built-in functions.
- Visualization of the query execution plan (`EXPLAIN` / `EXPLAIN ANALYZE`).
- Monitoring of active sessions via `.sys/query_sessions`.
- Cluster dashboard: CPU load, disk usage, memory usage, network traffic, node status (updated every 5 seconds).
- Access control (ACL) management: granting, revoking, and viewing permissions.
- Management of streaming queries: viewing, modifying, starting, stopping.
- Federated queries via external data sources (S3, databases).
- SQL Dialect Converter from other dialects (PostgreSQL, MySQL, ClickHouse, etc.) to YQL.
- Specialized editors for `JSON`, `JSONDOCUMENT`, `YSON` data types.

Requirements

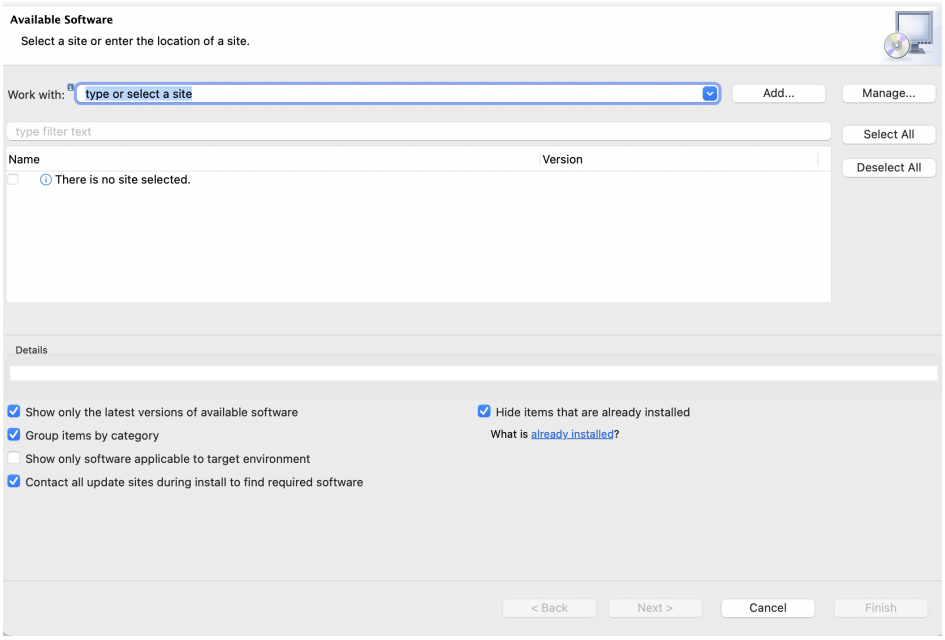
The plugin requires DBeaver Community Edition version 24.x or later.

Installing the Plugin

The plugin is installed via DBeaver's extension installation mechanism from a URL repository, which ensures automatic updates.

1. Open DBeaver. In the top menu, select **Help** → **Install New Software...**.
   Screenshot
   

2. Click the **Add...** button to the right of the **Work with:** field.
   Screenshot

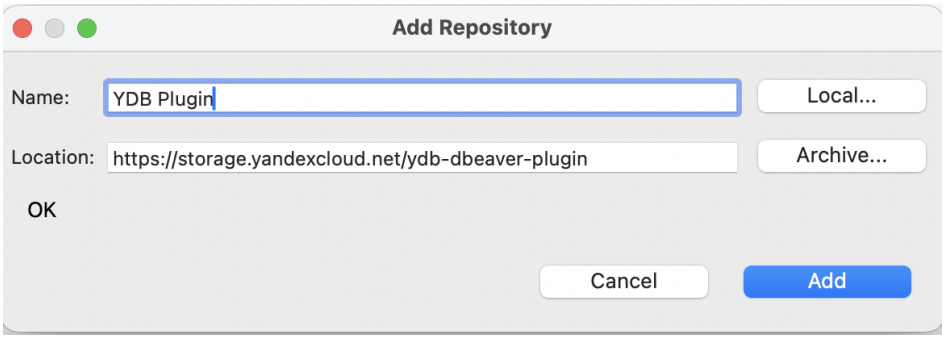


3. In the **Add Repository** window that opens, specify a repository name (for example, `YDB Plugin`) and paste the following URL into the **Location** field:

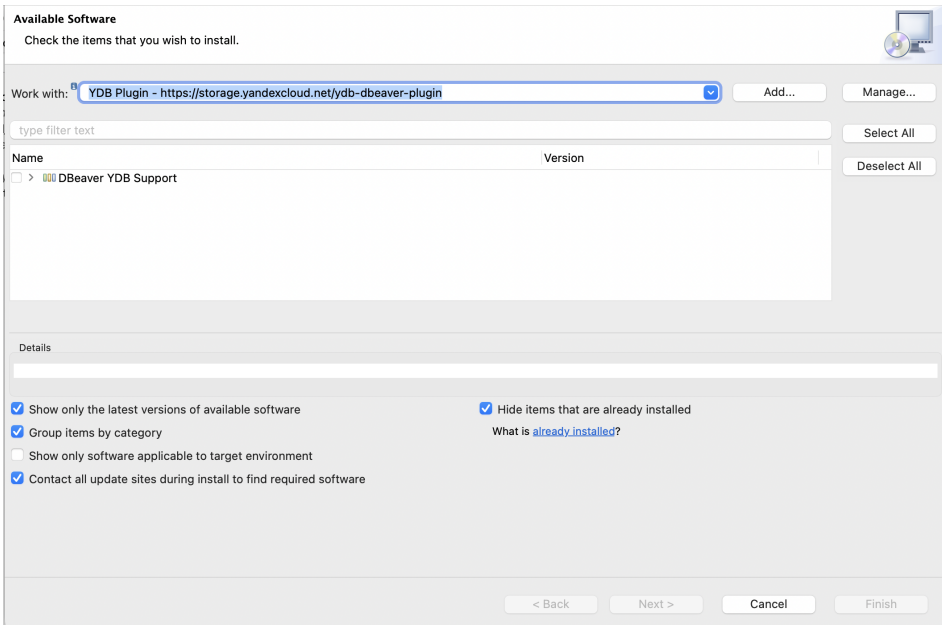
   ```
   https://storage.yandexcloud.net/ydb-dbeaver-plugin
   ```

   Click **Add**. DBeaver will load the repository metadata.

   Screenshot

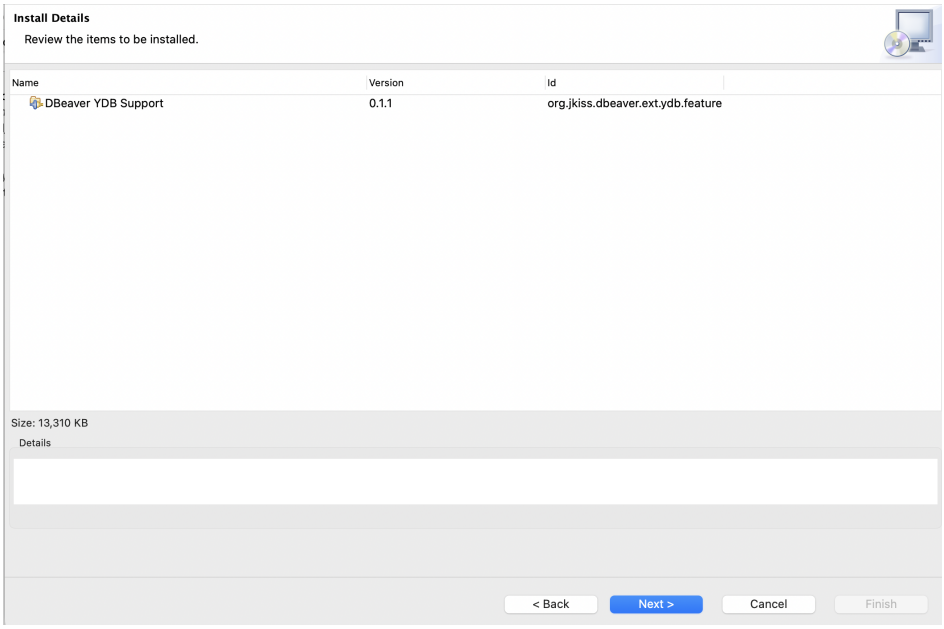


4. The **DBeaver YDB Support** category will appear in the component list. Select it and click **Next >**.
   Screenshot



5. On the **Install Details** screen, make sure both components (`org.jkiss.dbeaver.ext.ydb` and `org.jkiss.dbeaver.ext.ydb.ui`) are listed, then click **Next >**.

Screenshot

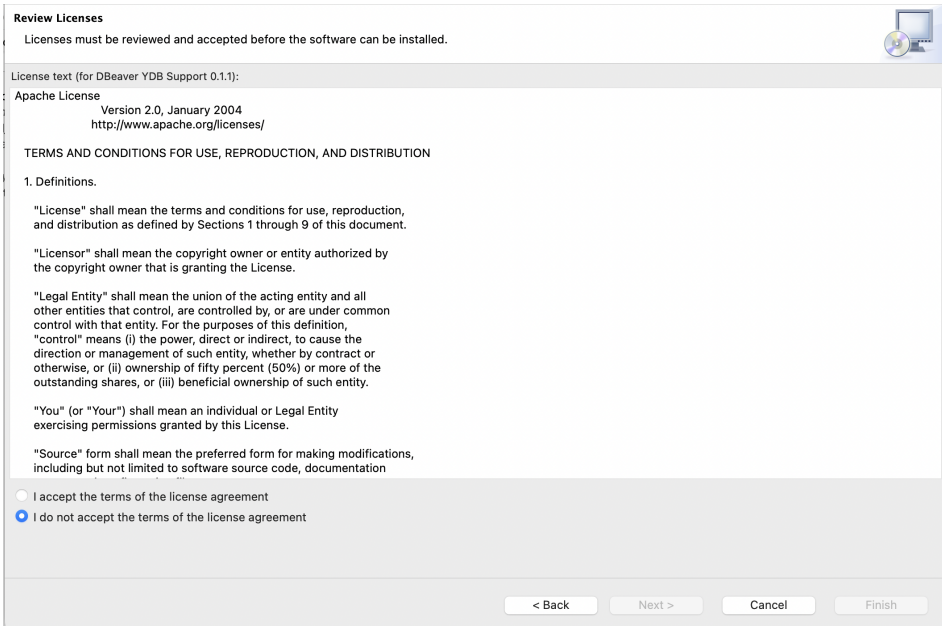


6. DBeaver may display a warning about unsigned content. This is expected behavior — the plugin's JAR files are not signed with a commercial certificate. Click **Install Anyway**.

> **Note**
>
> Eclipse, on which DBeaver is based, checks JAR file signatures to verify authenticity. This open-source plugin is distributed without a signature; the source code is available in the repository.

7. Review the license (Apache License 2.0), select **I accept the terms of the license agreements**, and click **Finish**.

   Screenshot



8. DBeaver will install the plugin and prompt you to restart. Click **Restart Now**. After restarting, the plugin will become active.

Creating a Connection to YDB

To create a connection to YDB, follow these steps:

1. In the top menu, select **Database** → **New Database Connection** (or press `Ctrl+Shift+N`).
2. In the search field, type `YDB`. Select **YDB** from the list and click **Next**.
3. The YDB connection setup page will open. Fill in the fields:

Field	Description	Example
Host	Host of the endpoint for the YDB cluster	`ydb.example.com`
Port	Port (default is `2135`)	`2135`
Database	Path to the database	`/Root/database`

Monitoring URL	URL of the YDB UI with the database path, used for the dashboard (optional)	`http://ydb.example.com:8765/monitoring/tenant?name=%2FRoot%2Fdatabase`
Use secure connection	Use a secure connection (`grpcs://`)	☑
Enable autocomplete API	Autocomplete via YDB API	☑

4. Select an authentication method from the **Auth type** dropdown list (see Authentication Methods).
5. Click the **Test Connection** button to verify the settings. If the connection is successful, a dialog will appear with the connection time in milliseconds.
6. Click the **Finish** button. The connection will appear in the **Database Navigator** panel.

Authentication Methods

The plugin supports all authentication methods available in YDB. The method is selected from the **Auth type** dropdown list on the connection setup page.

Anonymous

Connection without credentials. Used for local or test installations of YDB. No additional fields need to be filled out.

Static (Username and Password)

Authentication using a username and password. Enter the username in the **User** field and the password in the **Password** field. Used if YDB server has username and password authentication enabled.

> **Note**
>
> In managed installations of YDB, username and password authentication is disabled: managed services use the centralized access control system of the cloud platform (IAM).

Token

Authentication using an IAM or OAuth token. Enter the token in the **Token** field. The token is passed in the header of each request.

Service Account

Authentication using the key of a Yandex Cloud service account. Specify the path to the JSON file with the key in the **SA Key File** field (use the **...** button to select the file). For more information on how to create an authorized key, see the Yandex Cloud documentation.

Key file format:

```
{
  "id": "aje...",
  "service_account_id": "aje...",
  "private_key": "-----BEGIN RSA PRIVATE KEY-----\n..."
}
```

Metadata

Authentication via the Yandex Cloud metadata service. The plugin obtains an IAM token from the virtual machine's metadata service. Used only when DBeaver is running on a Yandex Cloud virtual machine.

Object Navigator

After connecting, the YDB object hierarchy is displayed in the **Database Navigator** panel. The root node is the connection, and inside it is the database path, which contains the following folders:

- **Tables** — tables organized by subdirectories according to the path in YDB (for example, a table with the path `folder1/subfolder/mytable` will be nested under `folder1 → subfolder`).
- **Topics** — topics.
- **Views** — views.
- **External Data Sources** — external data sources.
- **External Tables** — external tables.
- **System Views (.sys)** — system views, such as `partition_stats` , `query_sessions` .
- **Resource Pools** — resource pools.

Working with the Plugin

YQL Editor

Open the **SQL Editor** (`F3` or double-click on the connection). The editor supports:

- Syntax highlighting for YQL: keywords (`UPSERT` , `REPLACE` , `EVALUATE` , `PRAGMA` , `WINDOW` , and 145+ others), data types, built-in functions.
- Autocompletion of table, column, and function names.
- Query execution: `Ctrl+Enter` for the current query, `Ctrl+Shift+Enter` for the entire script.

Example YQL query:

```
UPSERT INTO `users` (id, name, created_at)
VALUES (1, "Alice", CurrentUtcDatetime());
```

EXPLAIN and Execution Plan

Click **Explain** (or `Ctrl+Shift+E`) to get the query execution plan. The plugin displays:

- **Text plan** — a tree of operations in text form.
- **Diagram** — a graphical representation as a DAG.
- **SVG plan** — an interactive visualization.

`EXPLAIN ANALYZE` additionally shows execution statistics (number of rows, execution time).

Session Manager

Right-click on the connection and select **Manage Sessions**, or use the **Database** → **Manage Sessions** menu item. The opened view displays all active sessions with the current query, state, and duration (data from the system view `.sys/query_sessions`). The **Hide Idle** checkbox hides sessions without an active query.

Cluster Dashboard

Open the **Dashboard** tab in the connection editor (requires the **Monitoring URL** field to be filled in during connection setup).

> **Warning**
>
> The dashboard is only available when working with self-hosted installations of YDB where access to the YDB UI is available. In Yandex Cloud Managed Service for YDB, the YDB UI is not published, so dashboard data is unavailable — use cloud platform tools for monitoring.

The dashboard displays real-time data (updated every 5 seconds):

- CPU load by nodes.
- Disk space usage.
- Memory usage.
- Network traffic.
- Number of executing queries.
- Status of cluster nodes.

Streaming Queries

In the navigator, expand the **Streaming Queries** folder. For each query, you can:

- View the original YQL.
- View errors (issues).
- View the execution plan.
- Perform actions: **Start**, **Stop**, **Alter**.

SQL Dialect Converter

The plugin allows you to convert an SQL query written in another dialect (PostgreSQL, MySQL, ClickHouse, etc.) to YQL. The converter is available on the **Convert Dialect** tab in the connection editor.

To convert a query:

1. In the **Source Dialect** dropdown list, select the source SQL dialect. The list of dialects is requested from the plugin's external service when the tab is first opened.
2. Paste the source SQL code into the **Input SQL** field.
3. Click **Convert**. The result will appear in the lower field.
4. Click **Copy** to copy the result to the clipboard.

For more information on how the converter works, supported dialects, and limitations, see the SQL Dialect Converter to YQL article.

> **Warning**
>
> The plugin sends the original query to an external HTTPS service for conversion. Do not use the converter for queries containing confidential data.

Creating Objects

Right-click on a folder or object and select **Create New**:

- **Create Table** — create a new table.
- **Create Topic** — create a new topic.
- **Create Resource Pool** — create a resource pool.

Updating the Plugin

DBeaver uses the Eclipse P2 mechanism to detect and install updates. When you install the plugin, DBeaver remembers the source — the repository URL. When a new version is published, DBeaver compares the installed version with the version in the repository and offers an update in one of two ways:

1. Automatically on the next DBeaver startup (if update checks are enabled in **Window** → **Preferences** → **Install/Update** → **Automatic Updates**).
2. Manually via **Help** → **Check for Updates**: select the available update and follow the same steps as during the initial installation (license → warning about unsigned content → restart).

Connecting to YDB with DataGrip

DataGrip is a powerful cross-platform tool for relational and NoSQL databases.

DataGrip allows you to work with YDB using the Java Database Connectivity (JDBC) protocol. This article demonstrates how to set up this integration.

Adding the YDB JDBC Driver to DataGrip

To connect to YDB from DataGrip, you need the YDB JDBC driver.

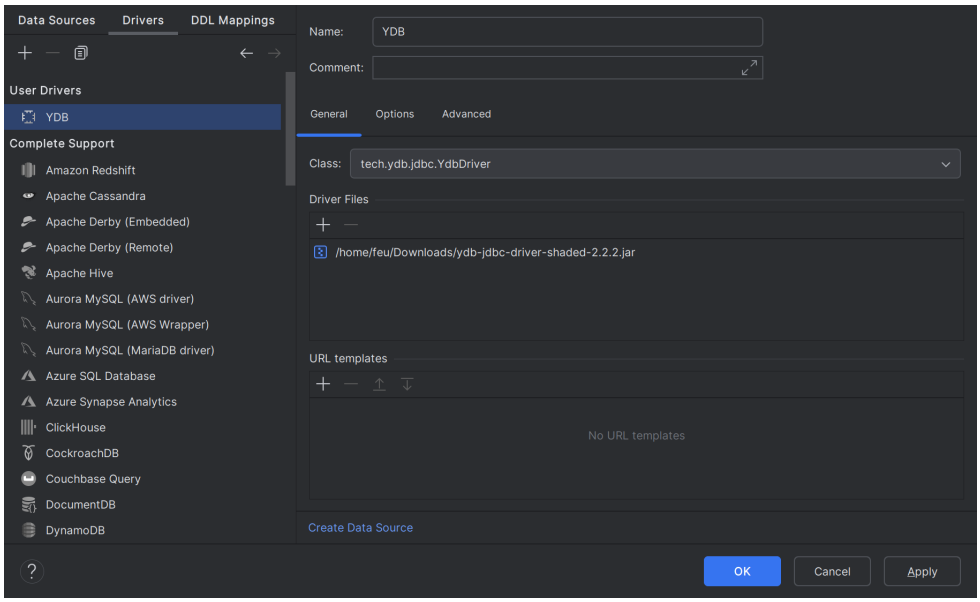
To download the YDB JDBC driver, follow these steps:

1. Go to the ydb-jdbc-driver repository.
2. Select the latest release (tagged as `Latest`) and save the `ydb-jdbc-driver-shaded-<driver-version>.jar` file.

To add the downloaded JDBC driver to DataGrip, follow these steps:

1. In DataGrip, go to **File | Data Sources...**.

   The **Data Sources and Drivers** dialog box appears.
2. In the **Data Sources and Drivers** dialog box, open the **Drivers** tab and click the **+** button.
3. In the **Name** field, specify `YDB`.
4. In the **Driver Files** section, click the **+** button, choose **Custom JARs...**, specify the path to the previously downloaded YDB JDBC driver (the `ydb-jdbc-driver-shaded-<driver-version>.jar` file), and click **OK**:
5. In the **Class** drop-down list, select `tech.ydb.jdbc.YdbDriver`.



6. Click **OK**.

Creating a Connection to YDB

To establish a connection, perform the following steps:

1. In DataGrip, go to **File | Data Sources...**.

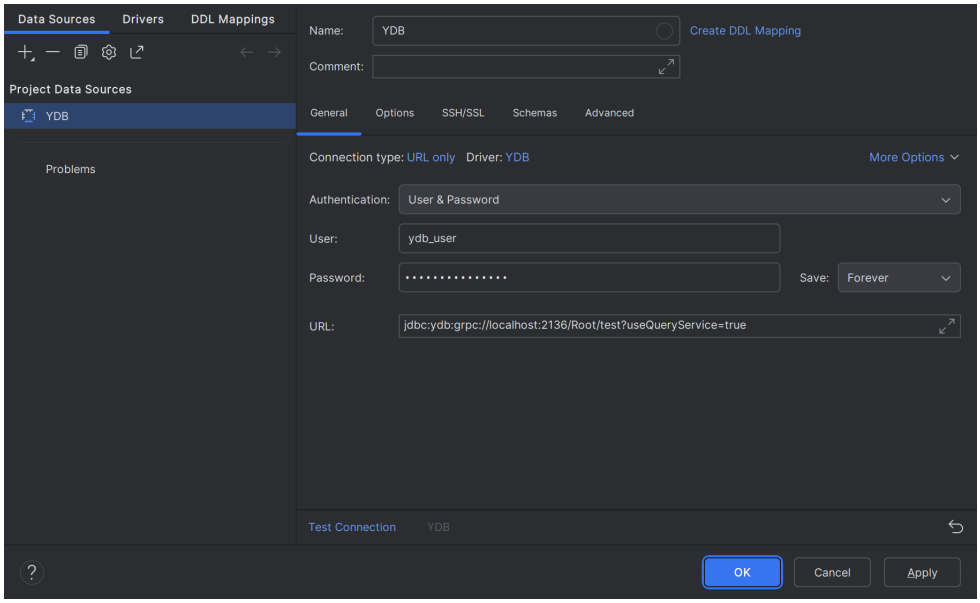
   The **Data Sources and Drivers** dialog box appears.
2. In the **Data Sources and Drivers** dialog box, on the **Data Sources** tab, click the **+** button and select `YDB`.
3. In the **Authentication** drop-down list, select an authentication method.
4. If you selected the `User & Password` authentication method, in the **User** and **Password** fields, enter your YDB login and password.
5. On the **General** tab, in the **URL** field, specify the following connection string:

   ```
   jdbc:ydb:<ydb_endpoint>/<ydb_database>?useQueryService=true
   ```

   Where:

   - `ydb_endpoint` — the endpoint of the YDB cluster.
   - `ydb_database` — the path to the database in the YDB cluster.

   A complete list of authentication methods and connection strings for YDB is provided in the JDBC driver description.

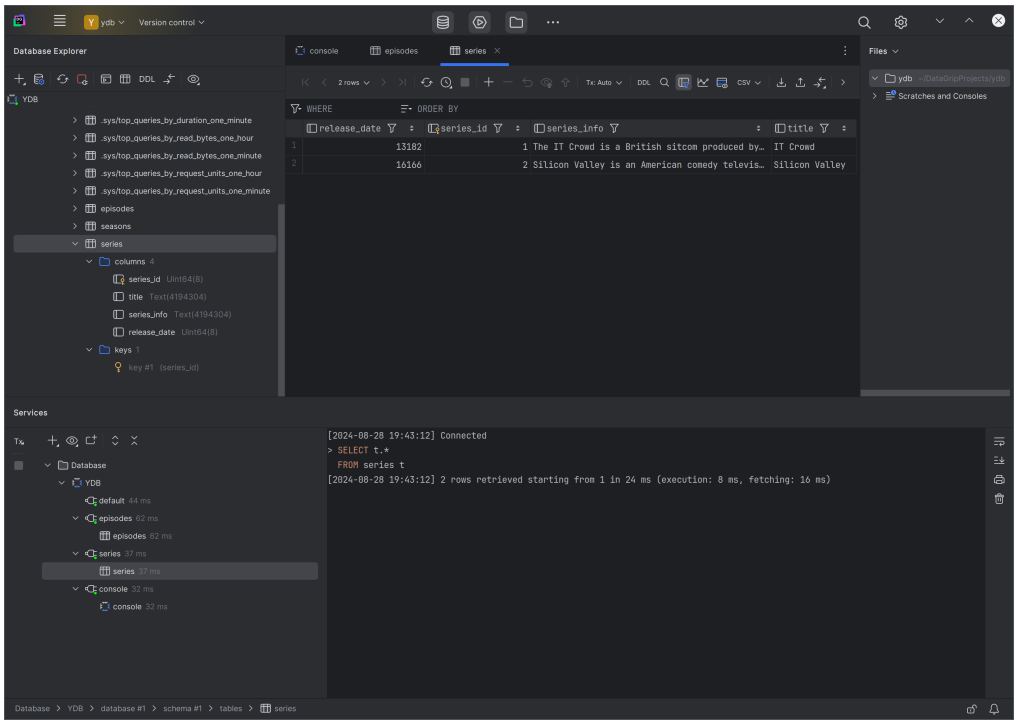


6. Click **Test Connection** to verify the settings.

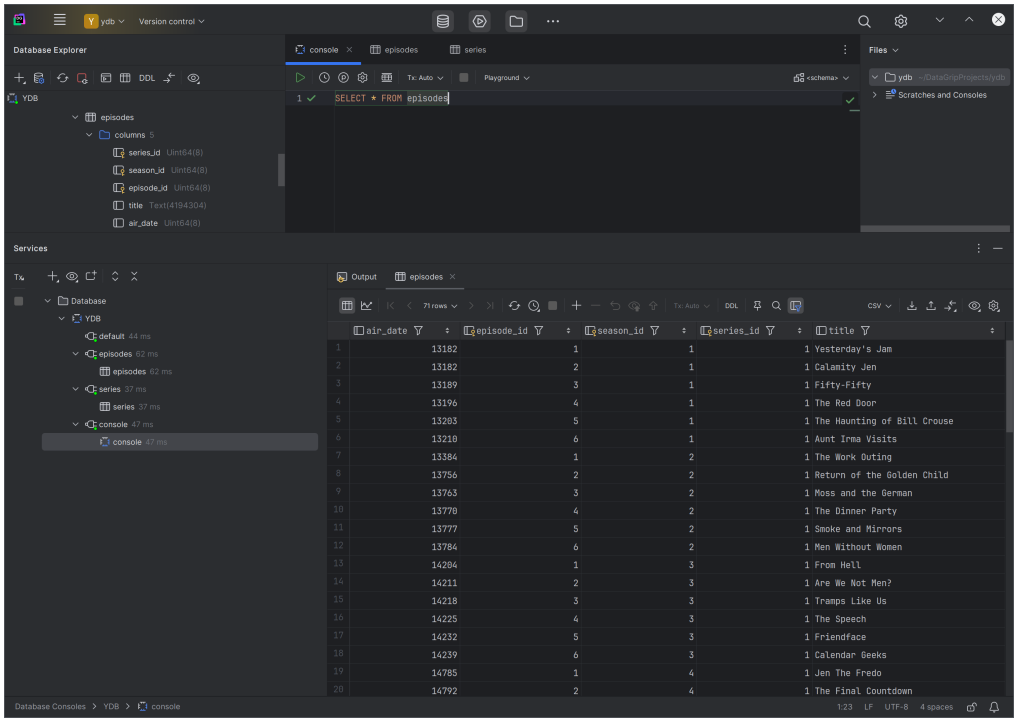
   If all the settings are correct, a message appears indicating a successful connection test.
7. Click **OK** to save the connection.

Working with YDB

With DataGrip you can view the list and structure of tables.



You can also execute queries on the data.


Database Explorer
console
episodes
series
SELECT * FROM episodes
Services
Output
episodes
air_date
episode_id
season_id
series_id
title
1 13182 1 1 1 Yesterday's Jam
2 13182 2 1 1 Calamity Jen
3 13189 3 1 1 Fifty-Fifty
4 13196 4 1 1 The Red Door
5 13203 5 1 1 The Haunting of Bill Crouse
6 13210 6 1 1 Aunt Irma Visits
7 13384 1 2 1 The Work Outing
8 13756 2 2 1 Return of the Golden Child
9 13763 3 2 1 Moss and the German
10 13770 4 2 1 The Dinner Party
11 13777 5 2 1 Smoke and Mirrors
12 13784 6 2 1 Men Without Women
13 14204 1 3 1 From Hell
14 14211 2 3 1 Are We Not Men?
15 14218 3 3 1 Tramps Like Us
16 14225 4 3 1 The Speech
17 14232 5 3 1 Friendface
18 14239 6 3 1 Calendar Geeks
19 14785 1 4 1 Jen The Fredo
20 14792 2 4 1 The Final Countdown

Work with YDB from Jupyter Notebook

Jupyter Notebook is an open-source tool for creating shareable documents that combine code, plain language descriptions, data, rich visualizations, and interactive controls.

The ydb-sqlalchemy dialect enables working with YDB from tools such as:

- Pandas
- JupySQL

Example

A detailed usage example is available as a notebook.

Prerequisites:

1. Python 3.8+
2. Jupyter Notebook
3. Existing YDB cluster, a single-node one from quickstart will suffice

To run the example, download the notebook file YDB SQLAlchemy - Jupyter Notebook Example.ipynb, open it in Jupyter, and follow each cell sequentially, executing code as necessary.

Data visualization (Business intelligence, BI)

Environment	Compatibility Level	Instruction
Apache Superset	ydb-sqlalchemy	Instruction
DataLens	Full	Instruction
Grafana	Full	Instruction

Apache Superset

Apache Superset is a modern data exploration and visualization platform. This article explains how to create visualizations using data stored in YDB.

Installation of dependencies

To connect to YDB from Superset, install the ydb-sqlalchemy driver.

The installation method depends on your Superset setup. For detailed instructions, see the official documentation.

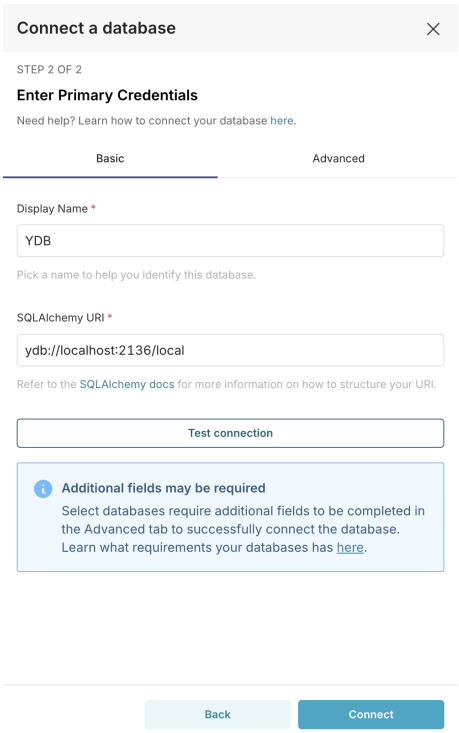
Adding a database connection to YDB

Native connection using SQLAlchemy driver

To connect to YDB from Apache Superset **version 5.0.0 and higher**, follow these steps:

1. In the Apache Superset toolbar, hover over **Settings** and select **Database Connections**.
2. Click the **+ DATABASE** button.

   The **Connect a database** wizard will appear.
3. In **Step 1** of the wizard, choose **YDB** from the **Supported databases** list. If the **YDB** option is not available, make sure that all the steps from prerequisites are completed.
4. In **Step 2** of the wizard, enter the YDB credentials in the corresponding fields:
   - **Display Name**. The YDB connection name in Apache Superset.
   - **SQLAlchemy URI**. A string in the format `ydb://{host}:{port}/{database_name}`, where **host** and **port** are parts of the endpoint of the YDB cluster to which the connection will be made, and **database_name** is the path to the database.



5. To enhance security, you can specify credentials parameters in the **Secure Extra** field under the **Advanced / Security** tab.

   Define the parameters as follows:

   Password

   ```
   {
       "credentials": {
           "username": "...",
           "password": "..."
       }
   }
   ```

   Access Token

   ```
   {
       "credentials": {
           "token": "...",
       }
   }
   ```

   Service Account

   ```
   {
       "credentials": {
           "service_account_json": {
   ```

```
            "id": "...",
            "service_account_id": "...",
            "created_at": "...",
            "key_algorithm": "...",
            "public_key": "...",
            "private_key": "..."
        }
    }
}
```

6. Click **CONNECT**.
7. To save the database connection, click **FINISH**.

For more information about configuring a YDB connection, refer to the YDB section in the official documentation.

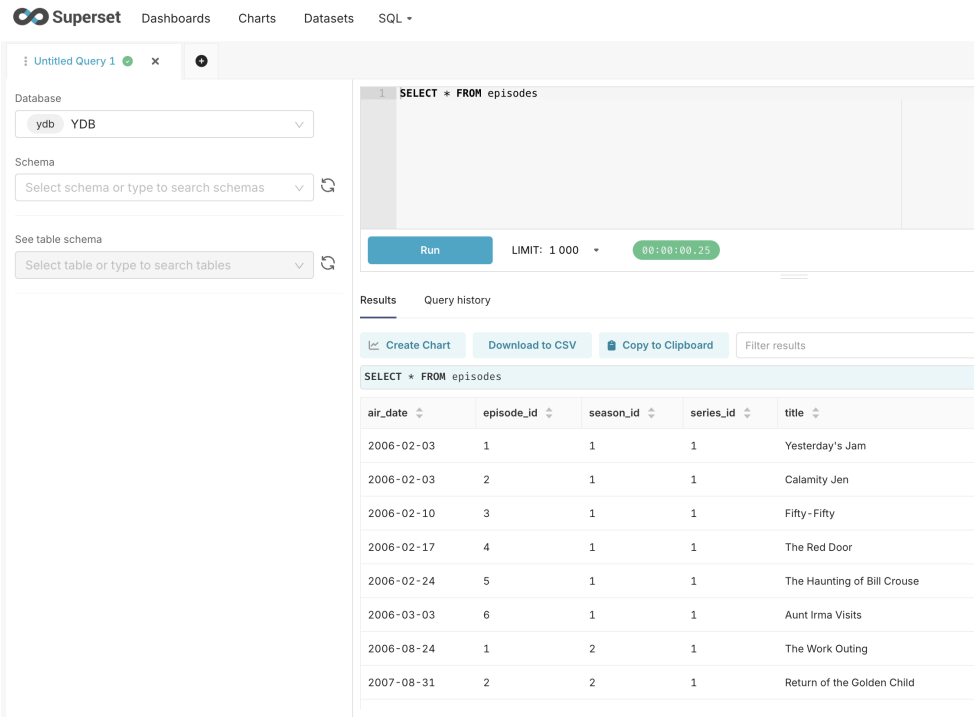
Creating a dataset

To create a dataset for a YDB table, follow these steps:

1. In the Apache Superset toolbar, hover over the **+** button and select **SQL query**.
2. In the **DATABASE** drop-down list, select the YDB database connection.
3. Enter the SQL query in the right section of the page. For example, `SELECT * FROM <ydb_table_name>`.

> **Tip**
>
> To create a dataset for a table located in a subdirectory of a YDB database, specify the table path in the table name. For example:
>
> ```
> SELECT * FROM "<path/to/subdirectory/table_name>";
> ```

1. Click **RUN** to test the SQL query.



2. Click the down arrow next to the **SAVE** button, then click **Save dataset**.

   The **Save or Overwrite Dataset** dialog box appears.
3. In the **Save or Overwrite Dataset** dialog box, select **Save as new**, enter the dataset name, and click **SAVE & EXPLORE**.

After creating datasets, you can use data from YDB to create charts in Apache Superset. For more information, refer to the Apache Superset documentation.

Creating a chart

Let's create a sample chart with the dataset from the `episodes` table that is described in the YQL tutorial.

The table contains the following columns:

- series_id
- season_id
- episode_id
- title
- air_date

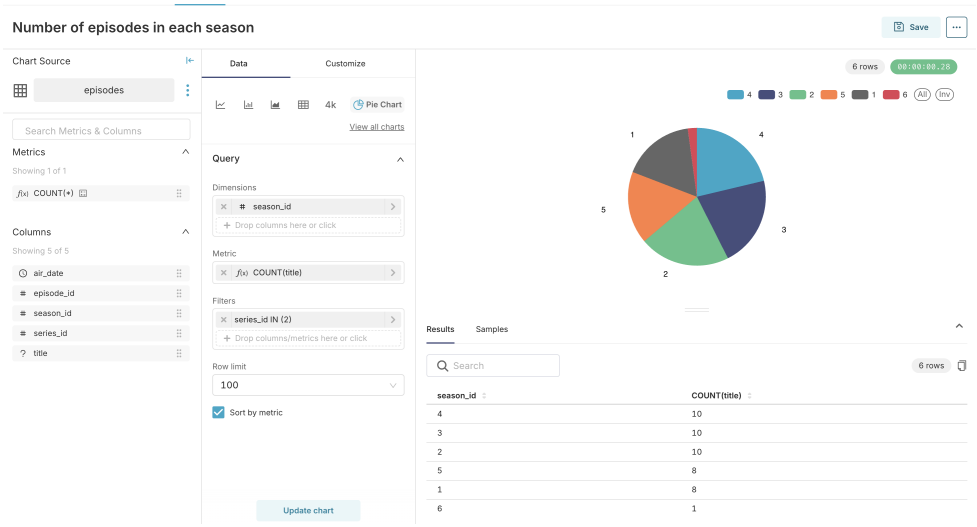
Let's say that we want to make a pie chart to show how many episodes each season contains.

To create a chart, follow these steps:

1. In the Apache Superset toolbar, hover over the **+** button and select **Chart**.

2. In the **Choose a dataset** drop-down list, select a dataset for the `episodes` table.
3. In the **Choose chart type** pane, select `Pie chart`.
4. Click **CREATE NEW CHART**.
5. In the **Query** pane, configure the chart:
    - In the **DIMENSIONS** drop-down list, select the `season_id` column.
    - In the **METRIC** field, specify the `COUNT(title)` function.
    - In the **FILTERS** field, specify the `series_id in (2)` filter.
6. Click **CREATE CHART**.

    The pie chart will appear in the preview pane on the right.



7. Click **SAVE**.

    The **Save chart** dialog box will appear.
8. In the **Save chart** dialog box, in the **CHART NAME** field, enter the chart name.
9. Click **SAVE**.

DataLens

DataLens is an open-source business intelligence (BI) and data visualization tool that enables users to analyze and display data from various sources, including YDB. DataLens allows you to describe data models, create charts and other visualizations, build dashboards, and provide collaborative access to analytics.

Prerequisites

DataLens must be deployed and configured.

> **Note**
>
> This article covers the integration of self-managed YDB and DataLens. For documentation on integrating the respective managed services, refer to the Yandex Cloud documentation.

Adding a database connection to YDB

To create a connection to YDB:

1. Go to the workbook page or create a new one.
2. In the top right corner, click **Create** → **Connection**.
3. Select the **YDB** connection.
4. Choose an authentication type:

   Anonymous

   - **Host name**. Specify the hostname for YDB connection.
   - **Port**. Specify the connection port for YDB. The default port is 2135.
   - **Database path**. Specify the name of the database to connect to.

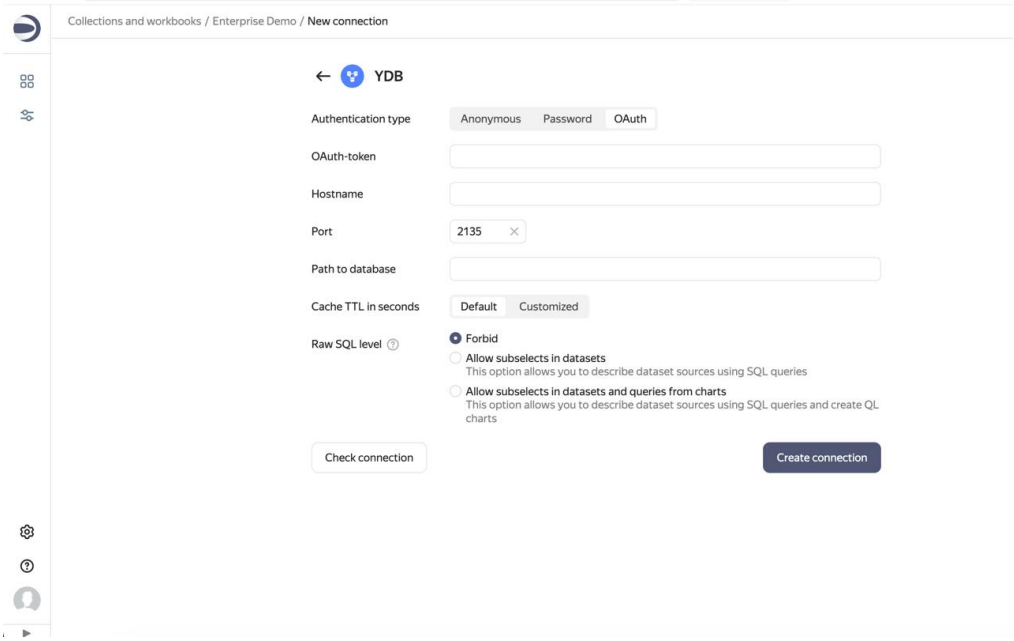
   Password

   - **Host name**. Specify the hostname for YDB connection.
   - **Port**. Specify the connection port for YDB. The default port is 2135.
   - **Database path**. Specify the name of the database to connect to.
   - **Username**. Enter the username to connect to YDB.
   - **Password**. Enter the user password.

   OAuth

   - **OAuth token**. Provide the OAuth token to access YDB.
   - **Host name**. Specify the hostname for YDB connection.
   - **Port**. Specify the connection port for YDB. The default port is 2135.
   - **Database path**. Specify the name of the database to connect to.
   - **Cache lifetime in seconds**. Set the cache lifetime or leave the default value. The recommended value is 300 seconds (5 minutes).
   - **SQL query access level**. Allows the use of custom SQL queries to create a dataset.
5. Click **Create connection**.
6. Specify a connection name and click **Create**.
7. Proceed to creating a dataset.

Example



YDB data source for Grafana

The YDB data source plugin allows you to use Grafana to query and visualize data from YDB.

Installation

Prerequisites: the plugin requires Grafana v9.2 or higher.

Follow the Grafana's plugin installation docs to install a plugin named `ydb-grafana-datasource-plugin`.

Configuration

YDB user for the data source

Set up an YDB user account with **read-only** permissions (more about permissions) and access to databases and tables you want to query.

> **Warning**
>
> Please note that Grafana does not validate that queries are safe. Queries can contain any SQL statements, including data modification instructions.

Data transfer protocol support

The plugin supports gRPC and gRPCS transport protocols.

Configuration via UI

Once the plugin is installed on your Grafana instance, follow these instructions to add a new YDB data source, and enter configuration options.

Configuration with provisioning system

Alternatively, Grafana's provisioning system allows you to configure data sources using configuration files. To read about how it works, including all the settings you can set for this data source, refer to the Provisioning Grafana data sources documentation.

Authentication

The Grafana plugin supports the following authentication methods: Anonymous, Access Token, Metadata, Service Account Key and Static Credentials.

Below is an example config for authenticating a YDB data source using username and password:

```
apiVersion: 1
datasources:
  - name: YDB
    type: ydbtech-ydb-datasource
    jsonData:
      authKind: '<password>'
      endpoint: 'grpcs://<hostname>:2135'
      dbLocation: '<location_to_db>'
      user: '<username>'
    secureJsonData:
      password: '<userpassword>'
      certificate: |
        <full content of *.pem file>
```

Here are fields that are supported in connection configuration:

Name	Description	Type
authKind	Authentication type	`"Anonymous"`, `"ServiceAccountKey"`, `"AccessToken"`, `"UserPassword"`, `"MetaData"`
endpoint	Database endpoint	`string`
dbLocation	Database location	`string`
user	User name	`string`
serviceAccAuthAccessKey	Service account access key	`string` (secured)
accessToken	Access token	`string` (secured)
password	User password	`string` (secured)
certificate	If self-signed certificates are used on your YDB cluster nodes, specify the Certificate Authority certificate used to issue them	`string` (secured)

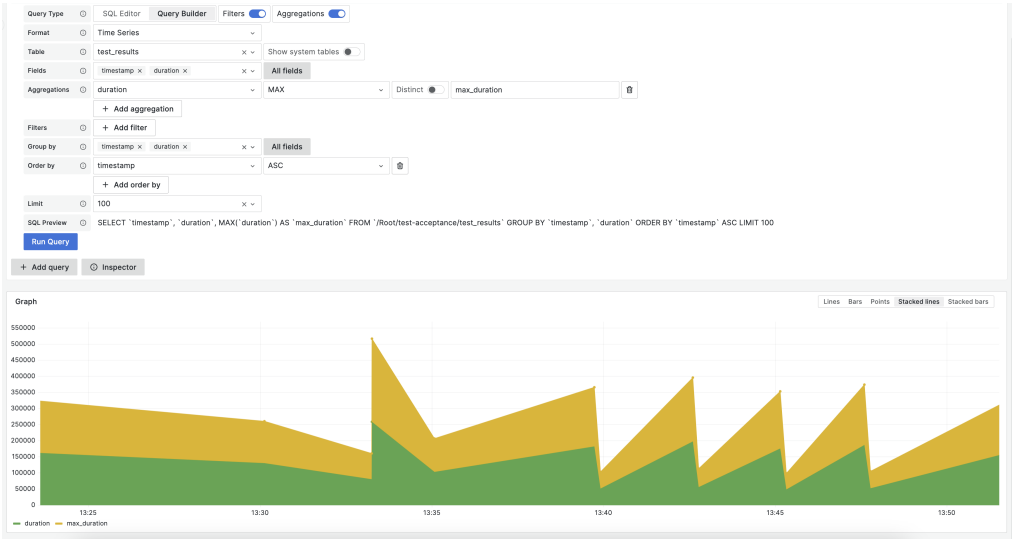
Building queries

YDB is queried with a SQL dialect named YQL.
The query editor allows to get data in different representations: time series, table, or logs.

Time series

Time series visualization options are selectable if the query returns at least one field with `Date`, `Datetime`, or `Timestamp` type (for now, working with time is supported only in UTC timezone) and at least one field with `Int64`, `Int32`, `Int16`, `Int8`, `Uint64`,

`Uint32`, `Uint16`, `Uint8`, `Double` or `Float` type. Then, you can select time series visualization options. Any other column is treated as a value column.



Multi-line time series

To create a multi-line time series, the query must return at least 3 fields:

- field with `Date`, `Datetime` or `Timestamp` type (for now, working with time is supported only in UTC timezone)
- metric - field with `Int64`, `Int32`, `Int16`, `Int8`, `Uint64`, `Uint32`, `Uint16`, `Uint8`, `Double` or `Float` type
- either metric or field with `String` or `Utf8` type - the value for splitting metrics into separate series.

For example:

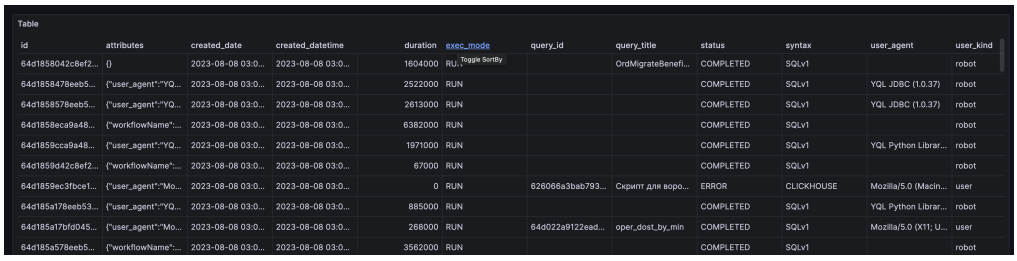
```
SELECT
    `timestamp`,
    `responseStatus`
    AVG(`requestTime`) AS `avgReqTime`
FROM `/database/endpoint/my-logs`
GROUP BY `responseStatus`, `timestamp`
ORDER BY `timestamp`
```

Tip

For this kind of queries, using column-oriented tables will likely be beneficial in terms of performance.

Tables

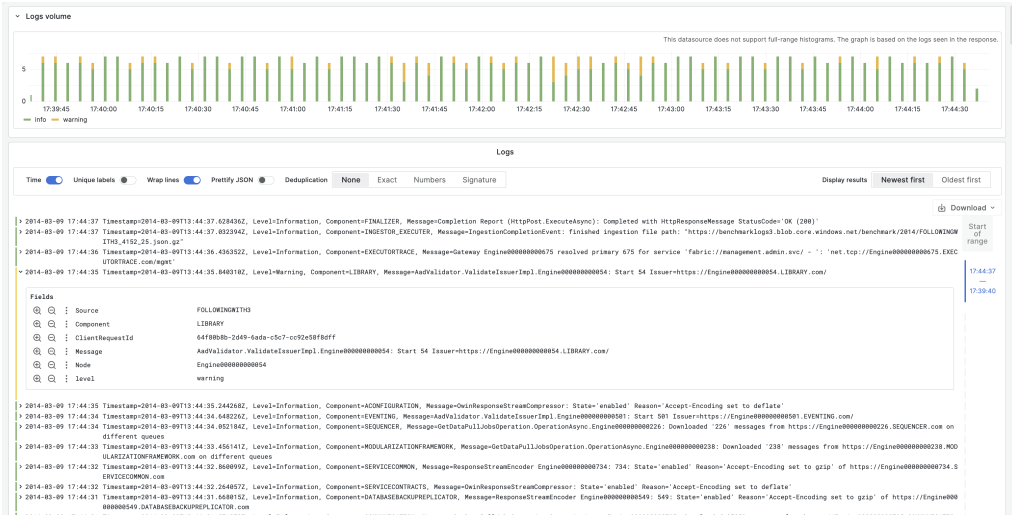
Table visualizations will always be available for any valid YDB query that returns exactly one result set.



Visualizing logs with the Logs Panel

To use the Logs panel, your query must return a `Date`, `Datetime`, or `Timestamp` value and a `String` value. You can select logs visualizations using the visualization options.

Only the first text field will be represented as a log line by default. This behavior can be customized using the query builder.



Macros

The query can contain macros, which simplify syntax and allow for dynamic parts, like date range filters.
There are two kinds of macros - Grafana-level and YDB-level. The plugin will parse query text and, before sending it to YDB, substitute variables and Grafana-level macros with particular values. After that YDB-level macroses will be treated by YDB server-side.

Here is an example of a query with a macro that will use Grafana's time filter:

```
SELECT `timeCol`
FROM `/database/endpoint/my-logs`
WHERE $__timeFilter(`timeCol`)
```

```
SELECT `timeCol`
FROM `/database/endpoint/my-logs`
WHERE $__timeFilter(`timeCol` + Interval("PT24H"))
```

Macro	Description	Output example
`$__timeFilter(expr)`	Replaced by a conditional that filters the data (using the provided column or expression) based on the time range of the panel in microseconds	`foo >= CAST(1636717526371000 AS Timestamp) AND foo <= CAST(1668253526371000 AS Timestamp)' )`
`$__fromTimestamp`	Replaced by the starting time of the range of the panel cast to Timestamp	`CAST(1636717526371000 AS Timestamp)`
`$__toTimestamp`	Replaced by the ending time of the range of the panel cast to Timestamp	`CAST(1636717526371000 AS Timestamp)`
`$__varFallback(condition, $templateVar)`	Replaced by the first parameter when the template variable in the second parameter is not provided.	

Templates and variables

To add a new YDB query variable, refer to Add a query variable.
After creating a variable, you can use it in your YDB queries by using Variable syntax.
For more information about variables, refer to Templates and variables.

Learn more

- Add Annotations.
- Configure and use Templates and variables.

- Add Transformations.
- Set up alerting; refer to Alerts overview.

Orchestration

System	Instruction
Apache Airflow™	Instruction

Apache Airflow™

Integration of YDB with Apache Airflow™ allows you to automate and manage complex workflows. Apache Airflow™ provides features for scheduling tasks, monitoring their execution, and managing dependencies between them, such as orchestration. Using Airflow to orchestrate tasks such as uploading data to YDB, executing queries, and managing transactions allows you to automate and optimize operational processes. This is especially important for ETL tasks, where large amounts of data require regular extraction, transformation, and loading.

YDB provider package apache-airflow-providers-ydb allows to work with YDB from Apache Airflow™. Apache Airflow™ tasks are Python applications consisting of a set of Apache Airflow™ operators and their dependencies, defining the order of execution.

Setup

Execute the following command on all Apache Airflow™ hosts to install the `apache-airflow-providers-ydb` package:

```
pip install ydb apache-airflow-providers-ydb
```

Python version 3.8 or higher is required.

Object model

The `airflow.providers.ydb` package contains a set of components for interacting with YDB:

- Operator `YDBExecuteQueryOperator` for integrating tasks into the Apache Airflow™ scheduler.
- Hook `YDBHook` for direct interaction with YDB.

YDBExecuteQueryOperator

To make requests to YDB, use the Apache Airflow™ operator `YDBExecuteQueryOperator`.

Required arguments

- `task_id` — the name of the Apache Airflow™ task.
- `sql` — the text of the SQL query to be executed in YDB.

Optional arguments

- `ydb_conn_id` — the connection identifier with the `YDB` type, containing the connection parameters for YDB. If omitted, a connection named `ydb_default` is used. The `ydb_default` connection is preinstalled as part of Apache Airflow™ and does not need to be configured separately.
- `is_ddl` — indicates that SQL DDL is running. If omitted or set to `False`, then SQL DML is running.
- `params` — a dictionary of query parameters.

Example:

```
ydb_operator = YDBExecuteQueryOperator(task_id="ydb_operator", sql="SELECT 'Hello, world!'")
```

In this example, a Apache Airflow™ task is created with the ID `ydb_operator`, which executes the query `SELECT 'Hello, world!'`.

YDBHook

The Apache Airflow™ class `YDBHook` is used to execute low-level commands in YDB.

Optional arguments

- `ydb_conn_id` — the connection identifier with the `YDB` type, containing the connection parameters for YDB. If omitted, a connection named `ydb_default` is used. The `ydb_default` connection is preinstalled as part of Apache Airflow™ and does not need to be configured separately.
- `is_ddl` — indicates that SQL DDL is running. If omitted or set to `False`, then SQL DML is running.

`YDBHook` supports the following methods:

- `bulk_upsert`
- `get_conn`

bulk_upsert

Performs batch data insertion into YDB tables.

Required arguments

- `table_name` — the name of the YDB table where the data will be inserted.
- `rows` — an array of rows to insert.
- `column_types` — a description of column types.

Example:

```
hook = YDBHook(ydb_conn_id=...)
column_types = (
        ydb.BulkUpsertColumns()
        .add_column("pet_id", ydb.OptionalType(ydb.PrimitiveType.Int32))
        .add_column("name", ydb.PrimitiveType.Utf8)
        .add_column("pet_type", ydb.PrimitiveType.Utf8)
        .add_column("birth_date", ydb.PrimitiveType.Utf8)
        .add_column("owner", ydb.PrimitiveType.Utf8)
    )

rows = [
    {"pet_id": 3, "name": "Lester", "pet_type": "Hamster", "birth_date": "2020-06-23", "owner": "Lily"},
    {"pet_id": 4, "name": "Quincy", "pet_type": "Parrot", "birth_date": "2013-08-11", "owner": "Anne"},
```

```
]
hook.bulk_upsert("pet", rows=rows, column_types=column_types)
```

In this example, a `YDBHook` object is created, through which the `bulk_upsert` batch data insertion operation is performed.

get_conn

Returns the `YDBConnection` object, which implements the `DbApiConnection` interface for working with data. The `DbApiConnection` class provides a standardized interface for interacting with the database, allowing operations such as establishing connections, executing SQL queries, and managing transactions, regardless of the specific database management system.

Example:

```
hook = YDBHook(ydb_conn_id=...)

# Execute the SQL query and get the cursor
connection = hook.get_conn()
cursor = connection.cursor()
cursor.execute("SELECT * from pet;")

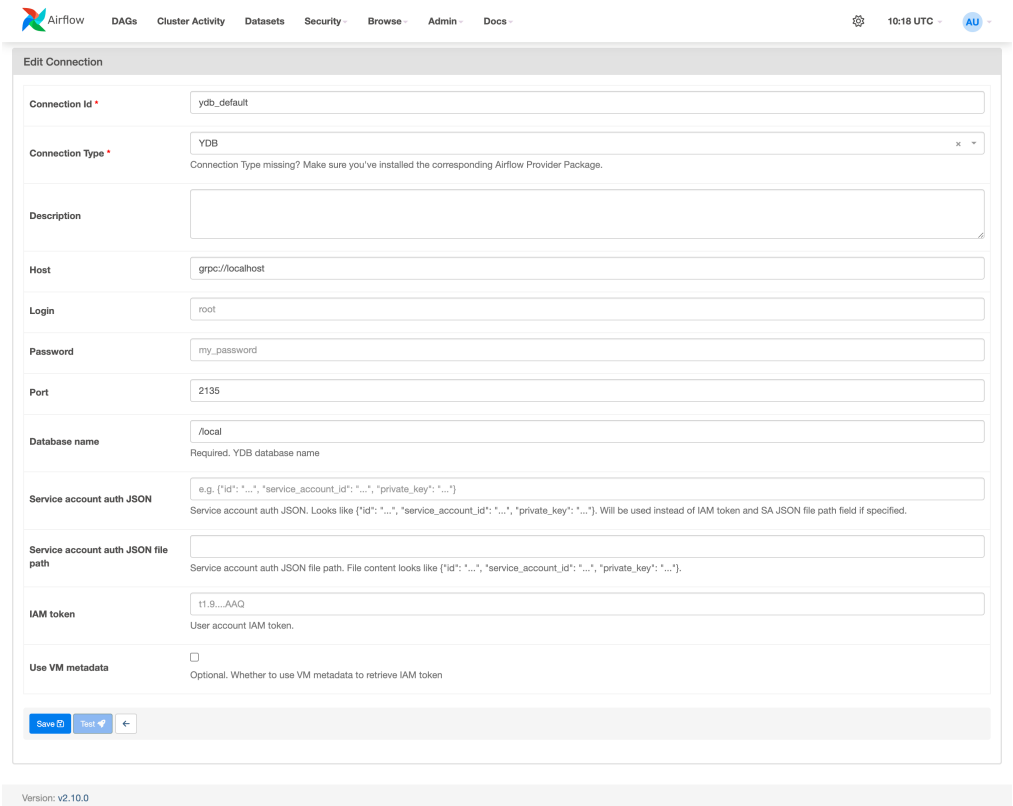
# Extract the result and column names
result = cursor.fetchall()
columns = [desc[0] for desc in cursor.description]

# Close cursor and connection
cursor.close()
connection.close()
```

In this example, a `YDBHook` object is created, and a `YDBConnection` object is requested from the created object. This connection is then used to read data and retrieve a list of columns.

Connection to YDB

To connect to YDB, you must create a new or edit an existing Apache Airflow™ connection with the `YDB` type.



Where:

- `Connection Id` — the Apache Airflow™ connection identifier.
- `Host` — the protocol and cluster address of YDB.
- `Port` — the port of YDB.
- `Database name` — the name of the YDB database.

Specify the details for one of the following authentication methods on the YDB cluster:

- `Login` and `Password` — specify user credentials for using static credentials.
- `Service account auth JSON` — specify the value of the `Service Account Key`.
- `Service account auth JSON file path` — specify the path to the `Service Account Key` file.
- `IAM token` — specify the IAM token.
- `Use VM metadata` — enable this option to use virtual machine metadata.

Matching between YQL and Python types

Below are the rules for converting YQL types to Python results. Types not listed below are not supported.

Scalar types

YQL type	Python type	Example in Python
`Int8`, `Int16`, `Int32`, `Uint8`, `Uint16`, `Uint32`, `Int64`, `Uint64`	`int`	`647713`
`Bool`	`bool`	`True`
`Float`, `float`	`float` NaN and Inf are represented as `None`	`7.88731023` `None`
`Decimal`	`Decimal`	`45.23410083`
`Utf8`	`str`	`"Text of string"`
`String`	`str`	`"Text of string"`

Complex types

YQL type	Python type	Example in Python
`Json`, `JsonDocument`	`str` (the entire node is inserted as a string)	`{"a":[1,2,3]}`
`Date`	`datetime.date`	`2022-02-09`
`Datetime`, `Timestamp`	`datetime.datetime`	`2022-02-09 10:13:11`

Optional types

YQL type	Python type	Example in Python
`Optional`	Original type or `None`	`1`

Containers

YQL type	Python type	Example in Python
`List<Type>`	`list`	`[1,2,3,4]`
`Dict<KeyType, ValueType>`	`dict`	`{"key1": "value1", "key2": "value2"}`
`Set<KeyType>`	`set`	`{"key_value1", "key_value2"}`
`Tuple<Type1, Type2>`	`tuple`	`(element1, element2)`
`Struct<Name:Utf8, Age:Int32>`	`dict`	`{"Name": "value1", "Age": value2}`

Special types

YQL type	Python type
`Void`, `Null`	`None`
`EmptyList`	`[]`
`EmptyDict`	`{}`

Example

To make requests to YDB, the package provides the Apache Airflow™ operator `YDBExecuteQueryOperator` and hook `YDBHook`.

In the example below, a `create_pet_table` task is launched to create a table in YDB. After the table is successfully created, the `populate_pet_table` task runs to populate the table with data using `UPSERT` commands. Additionally, the `populate_pet_table_via_bulk_upsert` task fills the table using `bulk_upsert`. After data insertion, a read operation is performed using the `get_all_pets` task and the `get_birth_date` task for parameterized data reading.

To execute queries in YDB, a pre-created connection of type YDB Connection named `test_ydb_connection` is used.

```
from __future__ import annotations

import datetime

import ydb
from airflow import DAG
from airflow.decorators import task
from airflow.providers.ydb.hooks.ydb import YDBHook
from airflow.providers.ydb.operators.ydb import YDBExecuteQueryOperator

@task
def populate_pet_table_via_bulk_upsert():
    hook = YDBHook(ydb_conn_id="test_ydb_connection")
```

```
    column_types = (
        ydb.BulkUpsertColumns()
        .add_column("pet_id", ydb.OptionalType(ydb.PrimitiveType.Int32))
        .add_column("name", ydb.PrimitiveType.Utf8)
        .add_column("pet_type", ydb.PrimitiveType.Utf8)
        .add_column("birth_date", ydb.PrimitiveType.Utf8)
        .add_column("owner", ydb.PrimitiveType.Utf8)
    )

    rows = [
        {"pet_id": 3, "name": "Lester", "pet_type": "Hamster", "birth_date": "2020-06-23", "owner": "Lily"},
        {"pet_id": 4, "name": "Quincy", "pet_type": "Parrot", "birth_date": "2013-08-11", "owner": "Anne"},
    ]
    hook.bulk_upsert("pet", rows=rows, column_types=column_types)


with DAG(
    dag_id="ydb_demo_dag",
    start_date=datetime.datetime(2020, 2, 2),
    schedule="@once",
    catchup=False,
) as dag:
    create_pet_table = YDBExecuteQueryOperator(
        task_id="create_pet_table",
        sql="""
            CREATE TABLE pet (
            pet_id INT,
            name TEXT NOT NULL,
            pet_type TEXT NOT NULL,
            birth_date TEXT NOT NULL,
            owner TEXT NOT NULL,
            PRIMARY KEY (pet_id)
            );
          """,
        is_ddl=True,  # must be specified for DDL queries
        ydb_conn_id="test_ydb_connection"
    )

    populate_pet_table = YDBExecuteQueryOperator(
        task_id="populate_pet_table",
        sql="""
              UPSERT INTO pet (pet_id, name, pet_type, birth_date, owner)
              VALUES (1, 'Max', 'Dog', '2018-07-05', 'Jane');

              UPSERT INTO pet (pet_id, name, pet_type, birth_date, owner)
              VALUES (2, 'Susie', 'Cat', '2019-05-01', 'Phil');
            """,
        ydb_conn_id="test_ydb_connection"
    )

    get_all_pets = YDBExecuteQueryOperator(task_id="get_all_pets", sql="SELECT * FROM pet;", ydb_conn_id="tes
t_ydb_connection")

    get_birth_date = YDBExecuteQueryOperator(
        task_id="get_birth_date",
        sql="SELECT * FROM pet WHERE birth_date BETWEEN '{{params.begin_date}}' AND '{{params.end_date}}'",
        params={"begin_date": "2020-01-01", "end_date": "2020-12-31"},
        ydb_conn_id="test_ydb_connection"
    )

    (
        create_pet_table
        >> populate_pet_table
        >> populate_pet_table_via_bulk_upsert()
        >> get_all_pets
        >> get_birth_date
    )
```

Data ingestion

Delivery system	Guide
FluentBit	Guide
LogStash	Guide
Kafka Connect Sink	Guide

Streaming data ingestion

Delivery System	Instruction
Apache Kafka API	Instruction

Log records collection using FluentBit

This section describes the integration between YDB and the log capture tool FluentBit to save and analyze the log records in YDB.

Overview

FluentBit is a tool that can collect text data, manipulate it (modify, transform, combine), and send it to various repositories for further processing. A custom plugin library for FluentBit has been developed to support saving the log records into YDB. The library's source code is available in the fluent-bit-ydb repository.

Deploying a log delivery scheme using FluentBit and YDB as the destination database includes the following steps:

1. Create YDB tables for the log data storage
2. Deploy FluentBit and YDB plugin for FluentBit
3. Configure FluentBit to collect and process the logs
4. Configure FluentBit to send the logs to YDB tables

Creating tables for log data

Tables for log data storage must be created in the chosen YDB database. The structure of the tables is determined by a set of fields of a specific log supplied using FluentBit. Depending on the requirements, different log types may be saved to different tables. Normally, the table for log data contains the following fields:

- timestamp
- log level
- hostname
- service name
- message text or its semi-structural representation as JSON document

YDB tables must have a primary key, uniquely identifying each table's row. A timestamp does not always uniquely identify messages coming from a particular source because messages might be generated simultaneously. To enforce the uniqueness of the primary key, a hash value can be added to the table. The hash value is computed using the CityHash64 algorithm over the log record data.

Row-based and columnar tables can both be used for log data storage. Columnar tables are recommended, as they support more efficient data scans for log data retrieval.

Example of the row-based table for log data storage:

```
CREATE TABLE `fluent-bit/log` (
    `timestamp`         Timestamp NOT NULL,
    `hostname`          Text NOT NULL,
    `input`             Text NOT NULL,
    `datahash`          Uint64 NOT NULL,
    `level`             Text NULL,
    `message`           Text NULL,
    `other`             JsonDocument NULL,
    PRIMARY KEY (
         `datahash`, `timestamp`, `hostname`, `input`
    )
);
```

Example of the columnar table for log data storage:

```
CREATE TABLE `fluent-bit/log` (
    `timestamp`         Timestamp NOT NULL,
    `hostname`          Text NOT NULL,
    `input`             Text NOT NULL,
    `datahash`          Uint64 NOT NULL,
    `level`             Text NULL,
    `message`           Text NULL,
    `other`             JsonDocument NULL,
    PRIMARY KEY (
         `timestamp`, `hostname`, `input`, `datahash`
    )
) PARTITION BY HASH(`timestamp`, `hostname`, `input`)
  WITH (STORE = COLUMN);
```

The command that creates the columnar table differs in the following details:

- it specifies the columnar storage type and the table's partitioning key in the last two lines;
- the `timestamp` column is the first column of the primary key, which is optimal and recommended for columnar, but not for row-based tables. See the specific guidelines for choosing the primary key for columnar tables and for row-based tables.

TTL configuration can be optionally applied to the table, limiting the data storage period and enabling the automatic removal of obsolete data. Enabling TTL requires an extra setting in the `WITH` section of the table creation command. For example, `TTL = Interval("P14D") ON timestamp` sets the storage period to 14 days, based on the `timestamp` field's value.

FluentBit deployment and configuration

FluentBit deployment should be performed according to its documentation.

YDB plugin for FluentBit is available in the source code form in the repository, along with the build instructions. A docker image is provided for container-based deployments: `ghcr.io/ydb-platform/fluent-bit-ydb`.

General logic, configuration syntax and procedures to set up the receiving, processing, and delivering logs in the FluentBit environment are defined in the corresponding FluentBit documentation.

Writing logs to YDB tables

Before using the YDB output plugin, it needs to be enabled in the FluentBit settings. The list of the enabled FluentBit plugins is configured in a file (for example, `plugins.conf`), which is referenced through the `plugins_file` parameter in the `SERVICE` section of the main FluentBit configuration file. Below is the example of such a file with YDB plugin enabled (plugin library path may be different depending on your setup):

```
# plugins.conf
[PLUGINS]
    Path /usr/lib/fluent-bit/out_ydb.so
```

The table below lists the configuration parameters supported by the YDB output plugin for FluentBit.

Parameter	Description
`Name`	Plugin type, should be value `ydb`
`Match`	(optional) Tag matching expression to select log records which should be routed to YDB
`ConnectionURL`	YDB connection URL, including the protocol, endpoint, and database path (see the documentation)
`TablePath`	Table path starting from database root (example: `fluent-bit/log`)
`Columns`	JSON structure mapping the fields of FluentBit record to the columns of the target YDB table. May include the pseudo-columns listed below
`CredentialsAnonymous`	Configured as `1` for anonymous YDB authentication
`CredentialsToken`	Token value, to use the token authentication YDB mode
`CredentialsYcMetadata`	Configure as `1` for virtual machine metadata YDB authentication
`CredentialsStatic`	Username and password for YDB authentication, specified in the following format: `username:password@`
`CredentialsYcServiceAccountKey`	Path of a file containing the service account (SA) key, to use the SA key YDB authentication
`CredentialsYcServiceAccountKeyJson`	JSON data of the service account key to be used instead of the filename (useful in K8s environment)
`Certificates`	Path to the certificate authority (CA) trusted certificates file, or the literal trusted CA certificate value
`LogLevel`	Plugin-specific logging level should be one of `disabled` (default), `trace`, `debug`, `info`, `warn`, `error`, `fatal` or `panic`

The following pseudo-columns are available, in addition to the actual FluentBit log record fields, to be used as source values in the column map (`Columns` parameter):

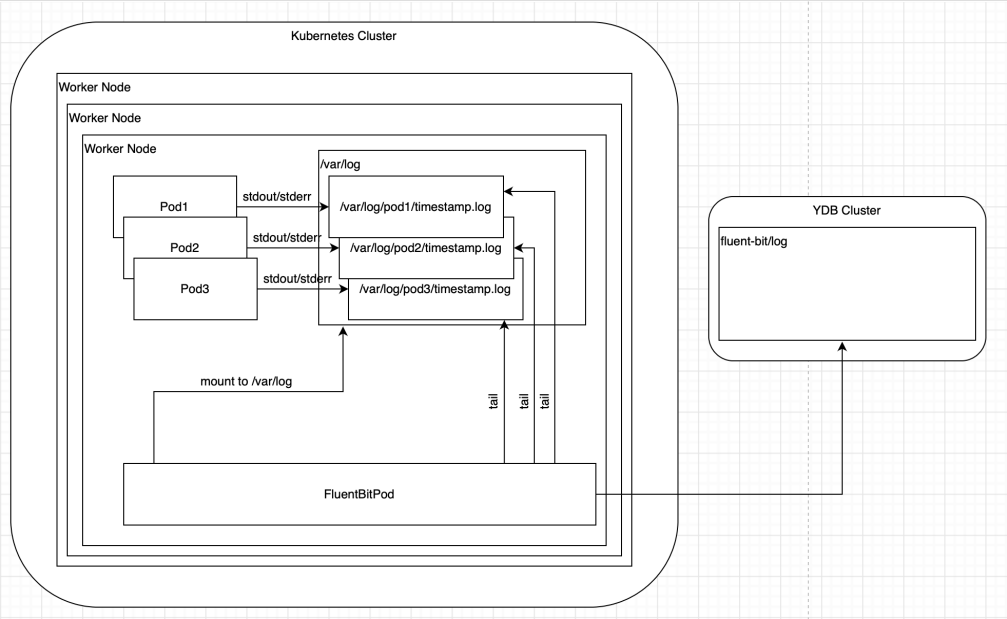
- `.timestamp` - log record timestamp (mandatory)
- `.input` - log input stream name (mandatory)
- `.hash` - uint64 hash code, computed over the log record fields (optional)
- `.other` - JSON document containing all log record fields that were not explicitly mapped to any table column (optional)

Example of `Columns` parameter value:

```
{".timestamp": "timestamp", ".input": "input", ".hash": "datahash", "log": "message", "level": "level", "hos
t": "hostname", ".other": "other"}
```

Collecting logs in a Kubernetes cluster

FluentBit is often used to collect logs in the Kubernetes environment. Below is the schema of the log delivery process, implemented using FluentBit and YDB, for applications running in the Kubernetes cluster:



In this diagram:

- Application pods write logs to stdout/stderr
- Text from stdout/stderr is saved as files on Kubernetes worker nodes
- Pod with FluentBit:
  - Mounts a folder with log files for itself
  - Reads the contents from the log files
  - Enriches log records with additional metadata
  - Saves records to YDB database

Table to store Kubernetes logs

Below is the YDB table structure to store the Kubernetes logs:

```
CREATE TABLE `fluent-bit/log` (
    `timestamp`         Timestamp NOT NULL,
    `file`              Text NOT NULL,
    `pipe`              Text NOT NULL,
    `message`           Text NULL,
    `datahash`          Uint64 NOT NULL,
    `message_parsed`    JSON NULL,
    `kubernetes`        JSON NULL,

    PRIMARY KEY (
         `timestamp`, `file`, `datahash`
    )
) PARTITION BY HASH(`timestamp`, `file`)
  WITH (STORE = COLUMN, TTL = Interval("P14D") ON `timestamp`);
```

Columns purpose:

- `timestamp` – the log record timestamp;
- `file` – the name of the source from which the log was read. In the case of Kubernetes, this will be the name of the file on the worker node in which the logs of a specific pod are written;
- `pipe` – stdout or stderr stream where application-level writing was done;
- `datahash` – hash code computed over the log record;
- `message` – the textual part of the log record;
- `message_parsed` – log record fields in the structured form, if it could be parsed using the configured FluentBit parsers from the textual part;
- `kubernetes` – information about the pod, including name, namespace, and annotations.

Optionally, TTL can be configured for the table, as shown in the example.

FluentBit configuration

In order to deploy FluentBit in the Kubernetes environment, a configuration file with the log collection and processing parameters must be prepared (typical file name: `values.yaml`). This section provides the necessary comments on this file's content with the examples.

It is necessary to replace the repository and image version of the FluentBit container:

```
image:
  repository: ghcr.io/ydb-platform/fluent-bit-ydb
  tag: latest
```

In this image, a plugin library has been added that implements YDB support.

The following lines define the rules for mounting log folders in FluentBit pods:

```
volumeMounts:
  - name: config
    mountPath: /fluent-bit/etc/conf

daemonSetVolumes:
  - name: varlog
    hostPath:
      path: /var/log
  - name: varlibcontainers
    hostPath:
      path: /var/lib/containerd/containers
  - name: etcmachineid
    hostPath:
      path: /etc/machine-id
      type: File

daemonSetVolumeMounts:
  - name: varlog
    mountPath: /var/log
  - name: varlibcontainers
    mountPath: /var/lib/containerd/containers
    readOnly: true
  - name: etcmachineid
    mountPath: /etc/machine-id
    readOnly: true
```

FluentBit startup parameters should be configured as shown below:

```
command:
  - /fluent-bit/bin/fluent-bit

args:
  - --workdir=/fluent-bit/etc
  - --plugin=/fluent-bit/lib/out_ydb.so
  - --config=/fluent-bit/etc/conf/fluent-bit.conf
```

The FluentBit pipeline for collecting, converting, and delivering logs should be defined according to the example:

```
config:
  inputs: |
    [INPUT]
        Name tail
        Path /var/log/containers/*.log
        multiline.parser cri
        Tag kube.*
        Mem_Buf_Limit 5MB
        Skip_Long_Lines On

  filters: |
    [FILTER]
        Name kubernetes
        Match kube.*
        Keep_Log On
        Merge_Log On
        Merge_Log_Key log_parsed
        K8S-Logging.Parser On
        K8S-Logging.Exclude On

    [FILTER]
        Name modify
        Match kube.*
        Remove time
        Remove _p

  outputs: |
    [OUTPUT]
        Name ydb
        Match kube.*
        TablePath fluent-bit/log
        Columns {".timestamp":"timestamp",".input":"file",".hash":"datahash","log":"message","log_parsed":"me
ssage_structured","stream":"pipe","kubernetes":"metadata"}
        ConnectionURL ${OUTPUT_YDB_CONNECTION_URL}
        CredentialsToken ${OUTPUT_YDB_CREDENTIALS_TOKEN}
```

Configuration blocks description:

- `inputs` - this block specifies where to read and how to parse logs. In this case, `*.log` files will be read from the `/var/log/containers/` folder, which is mounted from the host
- `filters` - this block specifies how the logs will be processed. In this case, for each log record, the corresponding metadata is added (using the Kubernetes filter), and unused fields (`_p` , `time`) are cut out
- `outputs` - this block specifies where the logs will be sent. In this case, logs are saved into the `fluent-bit/log` table in the YDB database. Database connection parameters (in the shown example, `ConnectionURL` and `CredentialsToken`) are defined using the environment variables – `OUTPUT_YDB_CONNECTION_URL` , `OUTPUT_YDB_CREDENTIALS_TOKEN` . Authentication parameters and the set of corresponding environment variables are updated depending on the configuration of the YDB cluster being used.

Environment variables are defined as shown below:

```
env:
  - name: OUTPUT_YDB_CONNECTION_URL
    value: grpc://ydb-endpoint:2135/path/to/database
  - name: OUTPUT_YDB_CREDENTIALS_TOKEN
    valueFrom:
      secretKeyRef:
        key: token
        name: fluent-bit-ydb-plugin-token
```

Authentication data should be stored as the secret object in the Kubernetes cluster configuration. Example command to create a Kubernetes secret:

```
kubectl create secret -n ydb-fluent-bit-integration generic fluent-bit-ydb-plugin-token --from-literal=token=
<YDB TOKEN>
```

Deploying FluentBit in a Kubernetes cluster

HELM is a tool to package and install applications in a Kubernetes cluster. To deploy FluentBit, the corresponding chart repository (containing the installation scenario) should be added using the following command:

```
helm repo add fluent https://fluent.github.io/helm-charts
```

After that, FluentBit can be deployed to a Kubernetes cluster with the following command:

```
helm upgrade --install fluent-bit fluent/fluent-bit \
  --version 0.37.1 \
  --namespace ydb-fluent-bit-integration \
  --create-namespace \
  --values values.yaml
```

The argument `--values` in the example command shown above references the file containing the FluentBit settings.

Installation verification

Check that FluentBit has started by reading its logs (there should be no `[error]` level entries):

```
kubectl logs -n ydb-fluent-bit-integration -l app.kubernetes.io/instance=fluent-bit
```

Check that there are records in the YDB table (they will appear approximately a few minutes after launching FluentBit):

```
SELECT * FROM `fluent-bit/log` LIMIT 10 ORDER BY `timestamp` DESC;
```

Resources cleanup

To remove FluentBit, it is sufficient to delete the Kubernetes namespace which was used for the installation:

```
kubectl delete namespace ydb-fluent-bit-integration
```

After uninstalling FluentBit, the log storage table can be dropped from the YDB database:

```
DROP TABLE `fluent-bit/log`;
```

Integrating Logstash and YDB

This section describes the integration options between YDB and Logstash, a server-side data collection and processing pipeline.

Introduction

Logstash dynamically ingests, transforms, and ships data regardless of format or complexity. A Logstash pipeline can contain different types of plugins: input, output, and filter. The YDB Logstash plugins repository is hosted on GitHub and contains the following plugins:

- **Storage Plugin** for persisting Logstash events in row-oriented or column-oriented YDB tables;
- **Input Topic Plugin** for reading Logstash events from YDB topics;
- **Output Topic Plugin** for sending Logstash events to YDB topics.

These plugins can be built from the source code or downloaded as pre-built artifacts for the two latest versions of Logstash.

> **Note**
>
> Further code snippets use the placeholder `<path-to-logstash>`, which must be replaced by a path to a directory with installed Logstash.

Execute the following command to install any Logstash plugin:

```
<path-to-logstash>/bin/logstash-plugin install </path/to/logstash-plugin.gem>
```

Check that the plugin has been installed:

```
<path-to-logstash>/bin/logstash-plugin list
```

The command will return a list of all installed plugins, which contain the plugin's name.

Configure YDB connection

All plugins use the same set of parameters to configure the connection to YDB. This set contains only one required parameter, `connection_string`. Other parameters are optional and allow configuring an authentication mode. An anonymous mode will be used if the configuration doesn't contain any of these parameters.

```
# This example demonstrates configuration for ydb_storage plugin.
# The plugins ydb_topics_output and ydb_topics_input configure the same way.
ydb_storage {
    # Database connection string contains a schema, hostname, port, and database path
    connection_string => "grpc://localhost:2136/local"
    # Authentication token (for using the "Access Token" mode)
    token_auth => "<token_value>"
    # Authentication token file path (for using the "Access Token" mode)
    token_file => "</path/to/token/file>"
    # Service account key file path (for using the "Service Account Key" mode)
    sa_key_file => "</path/to/key.json>"
    # Flag to use metadata authentication service (for using the "Metadata" mode)
    use_metadata => true
}
```

YDB Storage Plugin

This plugin allows storing the Logstash events stream in YDB tables for further analysis. This is especially useful with column-oriented tables optimized for handling Online Analytical Processing (OLAP) requests. Every field of a Logstash event will be stored in a column with a corresponding name. Fields that do not match any column will be ignored.

Configuration

The plugin configuration is done by adding a `ydb_storage` block in the `output` section of the Logstash pipeline config file. The plugin supports the standard set of connection parameters and a few additional options:

- `table_name` — the required name of the destination table.
- `uuid_column` — the optional name of the column that the plugin will use for storing an auto-generated identifier.
- `timestamp_column` — the optional name of the column that the plugin will use for storing the events' timestamp.

> **Warning**
>
> The Storage plugin doesn't check if the Logstash event has correct and unique values for the table's primary key. It uses the bulk upsert method for storing data in YDB, and if multiple events have the same primary keys, they will overwrite each other. The primary key is recommended to contain those event fields that will be present in every event and can uniquely identify each event. If no such set of fields exists, add an extra column to the primary key and fill it with a random value using the `uuid_column` parameter.

Usage example

Creating a table

Create a new column-oriented table with the necessary columns in any existing YDB database. For example, the table below uses a random value generated by the plugin.

```
CREATE TABLE `logstash_demo` (
    `uuid`     Text NOT NULL,      -- identifier
    `ts`       Timestamp NOT NULL, -- timestamp of event creation
    `message`  Text,
```

```
    `user`     Text,
    `level`    Int32,

    PRIMARY KEY (
        `uuid`
    )
)
WITH (STORE = COLUMN);
```

Setup plugin in Logstash pipeline config

To activate the plugin, add the `ydb_storage` block in the `output` section of the Logstash pipeline config. Additionally, add the `http` block in the `input` section for creating test messages via HTTP requests:

```
output {
  ydb_storage {
    connection_string => "..."    # YDB connection string
    table_name => "logstash_demo" # the table name
    uuid_column => "uuid"         # the primary key column with a random UUID
    timestamp_column => "ts"      # the column for storing the event timestamp
  }
}

input {
  http {
    port => 9876 # Any free port
  }
}
```

To apply these changes, restart Logstash.

Send test messages

Send a few test messages:

```
curl -H "content-type: application/json" -XPUT 'http://127.0.0.1:9876/http/ping' -d '{ "user" : "demo_user",
"message" : "demo message", "level": 4}'
curl -H "content-type: application/json" -XPUT 'http://127.0.0.1:9876/http/ping' -d '{ "user" : "test1", "lev
el": 1}'
curl -H "content-type: application/json" -XPUT 'http://127.0.0.1:9876/http/ping' -d '{ "message" : "error",
"level": -3}'
```

All commands return `ok` if the messages are sent.

Check that the messages are stored in YDB

To check that all sent messages are written to the table, run the query with ydb sql (the `ydb table query execute -t scan` command is deprecated):

```
ydb sql -s 'SELECT * FROM `logstash_demo`;'
```

The query will return a list of written events:

```
┌───────┬─────────────┬──────────────────────────────┬─────────────┬──────────────────────────────────────
──┐
| level | message     | ts                           |  user       | uuid
|
├───────┼─────────────┼──────────────────────────────┼─────────────┼──────────────────────────────────────
──┤
| -3    | "error"     | "2024-05-22T13:16:06.491000Z" |  null       | "74cd4048-0b61-4fb9-9385-308714e2188
1" |
|  1    | null        | "2024-05-22T13:15:56.591000Z" | "test1"     | "1df27d0a-9aa0-42c7-9ea2-ab69bc1f5d8
7" |
|  4    | "demo message" | "2024-05-22T13:15:38.760000Z" | "demo_user" | "b7468cb1-e1e3-46fa-965d-83e604e80a3
1" |
└───────┴─────────────┴──────────────────────────────┴─────────────┴──────────────────────────────────────
──┘
```

YDB Topic Input Plugin

This plugin allows reading messages from YDB topics and transforming them into `Logstash` events.

Configuration

The plugin configuration is done by adding a `ydb_topic` block to the `input` section of the Logstash pipeline config. The plugin supports the standard set of connection parameters and a few additional options:

- `topic_path` — the required the full path of the topic for reading.
- `consumer_name` — the required the name of the topic consumer.
- `schema` — the optional mode for processing of the YDB events. By default, the plugin reads and sends messages as binary data, but if you specify the `JSON` mode, each message will be parsed as a JSON object.

Usage example

Create a topic

Create a topic and add a consumer to it in any existing YDB database:

```
ydb -e grpc://localhost:2136 -d /local topic create /local/logstash_demo_topic
ydb -e grpc://localhost:2136 -d /local topic consumer add --consumer logstash-consumer /local/logstash_demo_t
opic
```

Setup plugin in Logstash pipeline config

To activate the plugin, add the `ydb_topic` block in the `input` section of the Logstash pipeline config. Additionally, add the `stdout` plugin in the `output` section for logging all `Logstash` events:

```
input {
  ydb_topic {
    connection_string => "grpc://localhost:2136/local" # YDB connection string
    topic_path => "/local/logstash_demo_topic"         # The full path of the topic to read
    consumer_name => "logstash-consumer"               # The consumer name
    schema => "JSON"                                   # Use JSON mode
  }
}

output {
  stdout { }
}
```

To apply these changes, restart Logstash.

Write a test message to the topic

Send a few test messages to the topic:

```
echo '{"message":"test"}' | ydb -e grpc://localhost:2136 -d /local topic write /local/logstash_demo_topic
echo '{"user":123}' | ydb -e grpc://localhost:2136 -d /local topic write /local/logstash_demo_topic
```

Check if Logstash processed the messages

The `stdout` plugin writes the messages to the Logstash logs:

```
{
      "message" => "test",
      "@timestamp" => 2024-05-23T10:31:47.712896899Z,
      "@version" => "1"
}
{
      "user" => 123.0,
      "@timestamp" => 2024-05-23T10:34:08.574599108Z,
      "@version" => "1"
}
```

YDB Topic Output Plugin

This plugin allows writing `Logstash` events to a YDB topic.

Configuration

The plugin configuration is done by adding a `ydb_topic` block to the `output` section of the Logstash pipeline config. The plugin supports the standard set of connection parameters and a few additional options:

- `topic_path` — the required the full path of the topic for writing.

Usage example

Create a topic

Create a topic in any existing YDB database:

```
ydb -e grpc://localhost:2136 -d /local topic create /local/logstash_demo_topic
```

Setup plugin in Logstash pipeline config

To activate the plugin, add the `ydb_topic` block in the `output` section of the Logstash pipeline config. Additionally, add the `http` block in the `input` section for creating test messages via HTTP requests:

```
output {
  ydb_topic {
    connection_string => "grpc://localhost:2136/local" # YDB connection string
    topic_path => "/local/logstash_demo_topic"         # The topic name for writing
  }
}

input {
  http {
    port => 9876 # Any free port
  }
}
```

To apply these changes, restart Logstash.

Send a test message

Send a test message via the `http` plugin:

```
curl -H "content-type: application/json" -XPUT 'http://127.0.0.1:9876/http/ping' -d '{ "user" : "demo_user",
"message" : "demo message", "level": 4}'
```

The command returns `ok` if the message has been sent successfully.

Reading the message from the topic

Check that the plugin wrote the message to the topic successfully by reading this message via CLI:

```
ydb -e grpc://localhost:2136 -d /local topic consumer add --consumer logstash-consumer /local/logstash_demo_t
opic
ydb -e grpc://localhost:2136 -d /local topic read /local/logstash_demo_topic --consumer logstash-consumer --c
ommit true
```

The latter command will return the message content:

```
{"level":4,"message":"demo message","timestamp":1716470292640,"user":"demo_user"}
```

Data migrations

Source	Guide
Arbitrary JDBC data sources	Guide
MySQL	Guide

Importing table structures and data from JDBC data sources to YDB

Introduction

Table structures and data can be imported into YDB from data sources accessible via JDBC drivers using the JDBC import tool. This can be useful for migrating from other database management systems to YDB or just importing your real data to run some benchmarks on YDB.

Data import is performed using the following process:

1. The tool connects to the source database and determines the list of tables and custom SQL queries to be imported.
2. The structure of target YDB tables is generated and optionally saved as YQL script.
3. The target database is checked for the existence of the target tables. Missing tables are created, and already existing ones may be left as is or re-created.
4. Row data is imported by reading from the source database using the SELECT statements and written into the target YDB database using the Bulk Upsert mechanism.

All operations, including metadata extraction from the source database, table creation (and re-creation) in the target database, and row data import, are performed in parallel mode (thread-per-table), using multiple concurrent threads and multiple open connections to both source and target databases. The maximum degree of parallelism is configurable, although the number of concurrent operations can't exceed the number of imported tables or custom SQL queries.

Import tool installation

To run the import tool, JDK 8 or later is required, with the `java` executable available in the program search path. The tool has been built and tested using OpenJDK 8.

Import tool binary distribution is available on the Releases page. The import tool is provided as the ZIP archive, which needs to be unpacked into the local directory on the server where the tool is required to run.

The import tool distribution archive contains the configuration file examples as `*.xml` files, sample tool startup script `ydb-importer.sh`, and the executable code for the tool and its dependencies (including the YDB Java SDK) as `lib/*.jar` files.

Before running the import tool, the JDBC drivers for the source databases should be put (as `*.jar` files) into the `lib` subdirectory of the tool installation directory.

Using the import tool

The tool reads the XML configuration file on startup. When running the tool, the file name must be specified as a free command line argument. Settings in the configuration file define:

- source database type and connection preferences;
- YDB target database connection preferences;
- (optional) file name to save the generated YQL script with YDB target tables structure;
- the rules to filter source tables' names;
- the list of custom SQL queries to fetch data from the data source (can be used to define custom queries to get data from all or some of the tables);
- the rules to generate target tables' names based on the names of the source tables;
- the degree of parallelism (which determines the worker pool size, plus the connection pool sizes for both source and target databases).

The configuration file based on one of the provided samples needs to be prepared before running the import tool.

The example command to run the tool is provided in the ydb-importer.sh file, which can also be used to run the tool as shown below:

```
./ydb-importer.sh my-import-config.xml
```

Import tool limitations

The tested data sources include the following database management systems:

- PostgreSQL
- MySQL
- Oracle Database
- Microsoft SQL Server
- IBM Db2
- IBM Informix

Other data sources will likely work, too, as the tool uses the generic JDBC APIs to retrieve data and metadata.

Secondary indexes are not imported.

Some data types and table structures are known to be unsupported:

- the embedded tables for Oracle Database
- spatial data type for Microsoft SQL Server
- object data types for Informix

Handling tables without the primary key

Each YDB table must have a primary key. If a primary key (or at least a unique index) is defined on the source table, the tool creates the primary key for the target YDB table with the columns and order determined by the original primary key. When multiple unique indexes are defined on the source table, the tool prefers the index with the smallest number of columns.

In case the source table has no primary key and no unique indexes, primary key columns can be configured using the import settings in the configuration file as a sequence of `key-column` elements in the `table-ref` section (see below the example in the description of the configuration file format).

When no primary key is defined anywhere, the tool automatically adds the column `ydb_synth_key` to the target table and uses it as the primary key. The values of the synthetic primary key are computed as "SHA-256" hash code over the values of all row cells except the columns of the `BLOB` type.

If the input table contains several rows with completely identical values, the destination table will have only one row per each set of duplicate input rows. This also means that the tool cannot import the table in which all columns are of `BLOB` type.

Importing large objects (BLOB, XML)

For each BLOB field in the source database, the import tool creates an additional YDB target table (BLOB supplemental table) with the following schema:

```
CREATE TABLE `blob_table`(
    `id` Int64,
    `pos` Int32,
    `val` String,
    PRIMARY KEY(`id`, `pos`)
)
```

The name of the BLOB supplemental table is generated based on the `table-options` / `blob-name-format` setting in the configuration file.

Each BLOB field value is saved as a sequence of records in the BLOB supplemental table in YDB. Actual data is stored in the `val` field, storing no more than 64K bytes. The order of blocks stored is defined by the values of the `pos` column, containing the integer values `1..N`.

For each source BLOB value, a unique value of type `Int64` is generated and stored in the `id` field of the BLOB supplemental table. This identifier is also stored in the "main" target table in a field that has the same name as the BLOB field in the source table.

For PostgreSQL, working with the `EXTENSION lo` BLOBS may require extra permissions for the account used to connect to the data source. An alternative option is to enable the compatibility mode on the database level using the following statement:

```
ALTER DATABASE dbname SET lo_compat_privileges TO on;
```

Configuration file format

Sample configuration files:

- for PostgreSQL
- for MySQL
- for Oracle Database
- for Microsoft SQL Server
- for IBM Db2
- for IBM Informix

Below is the explanation of the configuration file structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<ydb-importer>
    <workers>
        <!-- Number of worker threads (integer starting with 1).
             This setting also defines the maximum number of source and target database sessions.
         -->
        <pool size="4"/>
    </workers>
    <!-- Source database connection parameters.
         type - the required attribute defining the type of the data source
      -->
    <source type="generic|postgresql|mysql|oracle|mssql|db2|informix">
        <!-- JDBC driver class name to be used. Typical values:
              org.postgresql.Driver
              com.mysql.cj.jdbc.Driver
              org.mariadb.jdbc.Driver
              oracle.jdbc.driver.OracleDriver
              com.microsoft.sqlserver.jdbc.SQLServerDriver
              com.ibm.db2.jcc.DB2Driver
              com.informix.jdbc.IfxDriver
        -->
        <jdbc-class>driver-class-name</jdbc-class>
        <!-- JDBC driver URL. Value templates:
              jdbc:postgresql://hostname:5432/dbname
              jdbc:mysql://hostname:3306/dbname
              jdbc:mariadb://hostname:3306/dbname
              jdbc:oracle:thin:@//hostname:1521/serviceName
              jdbc:sqlserver://localhost;encrypt=true;trustServerCertificate=true;database=AdventureWorks202
2;
              jdbc:db2://localhost:50000/SAMPLE
              jdbc:informix-sqli://localhost:9088/stores_demo:INFORMIXSERVER=informix
        -->
        <jdbc-url>jdbc-url</jdbc-url>
        <username>username</username>
        <password>password</password>
    </source>
    <!-- Target YDB database connection parameters. -->
    <target type="ydb">
        <!-- If the following tag is defined, the tool will import
             The YQL script is used to generate YDB tables in the file specified.
             It can also be used without specifying the connection string,
             if the actual target schema creation is not required.
        -->
        <script-file>sample-database.yql.tmp</script-file>
        <!-- Connection string: protocol + endpoint + database. Typical values:
```

```
            grpcs://ydb.serverless.yandexcloud.net:2135?database=/ru-central1/b1gfvslmokutuvt2g019/etn63999hr
inbapmef6g
            grpcs://localhost:2135?database=/local
            grpc://localhost:2136?database=/Root/testdb
         -->
        <connection-string>ydb-connection-string</connection-string>
        <!-- Authentication mode:
          ENV      Configure authentication via environment variables
          NONE     Anonymous access, or static credentials in the connection string
          STATIC   Static credentials defined as login and password properties (see below)
          SAKEY    Service account key file authentication for YDB Managed Service
          METADATA Service account metadata authentication for YDB Managed Service
        -->
        <auth-mode>ENV</auth-mode>
        <!--
         For managed YDB in Yandex Cloud, authentication parameters can be specified
         in the environment variables, as specified in the documentation:
            https://ydb.tech/en/docs/reference/ydb-sdk/auth#env
         In case the Service Account authentication is used, either explicitly
         or through the YDB_SERVICE_ACCOUNT_KEY_FILE_CREDENTIALS env, the key file
         must be generated as written in the following document:
            https://yandex.cloud/en/docs/iam/operations/authorized-key/create
        -->
        <!-- Custom TLS certificates, if needed -->
        <tls-certificate-file>ca.crt</tls-certificate-file>
        <!-- For auth-mode: STATIC -->
        <static-login>username</static-login>
        <static-password>password</static-password>
        <!-- For auth-mode: SAKEY -->
        <sa-key-file>ydb-sa-keyfile.json</sa-key-file>
        <!-- Drop the already existing tables before loading the data -->
        <replace-existing>false</replace-existing>
        <!-- Should the tool actually load the data after creating tables? -->
        <load-data>true</load-data>
        <!-- Maximum rows per bulk upsert operation
             (used for writing into the main table) -->
        <max-batch-rows>1000</max-batch-rows>
        <!-- Maximum rows per blob bulk upsert operation
             (used for writing into the supplemental BLOB fields tables) -->
        <max-blob-rows>200</max-blob-rows>
    </target>
    <!-- Table name and structure conversion rules.
         Each rule is defined under a distinct name, and later referenced in the table mappings.
     -->
    <table-options name="default">
        <!--  Table name case conversion mode: ASIS (no changes, used by default), LOWER, UPPER. -->
        <case-mode>ASIS</case-mode>
        <!-- The template used to generate the full destination YDB table name,
             including the directory to put the table in.
             The values ${schema} and ${table} are used to substitute
             the schema and table name of the source table. -->
        <table-name-format>oraimp1/${schema}/${table}</table-name-format>
        <!-- The template used to generate the name of the supplemental table
             to store source BLOB field's data in YDB.
             The values ${schema}, ${table} and ${field}
             are used for source schema, table and BLOB field names. -->
        <blob-name-format>oraimp1/${schema}/${table}_${field}</blob-name-format>
        <!-- Date and timestamp data type values conversion mode.
             Possible values: DATE (use YDB Date datatype, default), INT, STR.
             DATE does not support input values before January, 1, 1970,
                 and there will be import errors for such values in the source
             INT saves date as 32-bit integer YYYYMMDD for dates,
                 and as a 64-bit milliseconds since epoch for timestamps.
             STR saves dates as character strings (Utf8) in format "YYYY-MM-DD",
                 and in "YYYY-MM-DD hh:mm:ss.xxx" for timestamps.
         -->
        <conv-date>INT</conv-date>
        <conv-timestamp>STR</conv-timestamp>
        <!-- If true, columns with unsupported types are skipped with warning,
             otherwise import error is generated, and the whole table is skipped. -->
        <skip-unknown-types>true</skip-unknown-types>
    </table-options>
    <!-- Table map filters the source tables and defines the conversion modes for them. -->
    <table-map options="default">
        <!-- Schemas to include, can be specified as regular expression or literal value -->
        <include-schemas regexp="true">.*</include-schemas>
        <!-- Schemas to exclude (of those included), as literal or regular expression -->
        <exclude-schemas>SOMESCHEMA</exclude-schemas>
        <!-- Particular tables can be included or excluded using the
             "include-tables" and "exclude-tables" tags,
             using literals or regular expressions as well. -->
    </table-map>
    <!-- Table reference may refer to a particular table in a particular schema
         and optionally specify the query to execute. The latter option allows to import
         virtual tables that do not actually exist on the data source. -->
    <table-ref options="default">
        <schema-name>ora$sys</schema-name>
        <table-name>all_tables</table-name>
        <!-- If the query text is defined, it is executed as shown. -->
        <query-text>SELECT * FROM all_tables</query-text>
        <!-- Primary key column names should be defined for the query result.  -->
```

```
        <key-column>OWNER</key-column>
        <key-column>TABLE_NAME</key-column>
    </table-ref>
</ydb-importer>
```

Importing table schemas and data from MySQL

Introduction

Migrating schemas and data from **MySQL** to YDB — including **large tables** whose size exceeds available RAM — is performed using the mysql2ydb utility.

The utility's goal is to get an exact one-to-one copy of the source database. Table and column names are preserved as in MySQL; data types are mapped to the nearest equivalents in YDB (`INT` → `Int32`, `VARCHAR` → `Text`, `AUTO_INCREMENT` → `BigSerial`, etc.). Since names are transferred unchanged, they must be valid identifiers in YDB (characters, length, reserved words). The result should be a YDB database that replicates the original structure and allows queries with minimal "translation" from MySQL.

Each data fragment is read from MySQL by a single bounded query `SELECT`, written to YDB in a batch (`BulkUpsert` or transactional `UPSERT`), and then freed from memory. The process repeats until the entire table is processed — no more than one batch is stored in memory at a time (no more than two with pipeline reading).

The import is performed in the following order:

1. The utility connects to MySQL and, based on the `information_schema` metadata, determines the list of tables and their structure.
2. Target tables with mapped data types are created in YDB.
3. Data is transferred in batches: reading from MySQL and writing to YDB.
4. After each batch is successfully written, the cursor position is saved to the state table — in case of failure, migration can be resumed from the last successful fragment.

MySQL schema mapping

When transferring the schema, mysql2ydb aims for **structural correspondence**: the same table and column names, indexes where YDB supports them, and types that preserve values without loss (unsigned integers remain unsigned, boolean types remain boolean, etc.).

The utility reads MySQL metadata from `information_schema` and generates YDB DDL directly. The main type mappings are:

MySQL type	Type in YDB
`INT`, `MEDIUMINT`, `SMALLINT`, `TINYINT`	`Int32`
`BIGINT`	`Int64`
`INT UNSIGNED`, `MEDIUMINT UNSIGNED`, `SMALLINT UNSIGNED`, `TINYINT UNSIGNED`	`Uint32`
`BIGINT UNSIGNED`	`Uint64`
`TINYINT(1)`	`Bool`
`FLOAT`	`Float`
`DOUBLE`, `REAL`	`Double`
`BIT`	`Uint64` (including `BIT(1)` — wrapped in `Uint64`, not in `Bool`)
`CHAR`, `VARCHAR`, `TEXT`, `MEDIUMTEXT`, `LONGTEXT`	`Text`
`ENUM`, `SET`, `JSON`	`Text`
`BINARY`, `VARBINARY`, `BLOB`, `MEDIUMBLOB`, `LONGBLOB`	`String` (bytes inline in the same column)
`DATE`	`Date`
`DATETIME`, `TIMESTAMP`	`Timestamp` — the value is stored as UTC. For `DATETIME` (a type without time zone), the interpretation depends on the `time_zone` MySQL setting during reading; keep this in mind when migrating historical data.
`DECIMAL(p, s)`, `NUMERIC(p, s)`	`Decimal(22, 9)` for all — the original `p` / `s` are not preserved; values that do not fit in 22 digits will be lost
`YEAR`	`Uint16`
Other types	`Text` (fallback)

Structural decisions

MySQL schema element	mysql2ydb decision
Table and column names	As in MySQL (`users`, `orders`, ...); must be valid YDB identifiers
`AUTO_INCREMENT`	`BigSerial` + `ALTER SEQUENCE … START WITH` from `TABLES.AUTO_INCREMENT`
Secondary `KEY` / `UNIQUE KEY`	`INDEX … GLOBAL ASYNC` / `GLOBAL UNIQUE SYNC` in `CREATE TABLE`
Table without `PRIMARY KEY`	Not supported when automatically creating the schema — see Tables without a primary key
Partitioning	Only `AUTO_PARTITIONING_BY_LOAD`

Secondary indexes

Non-unique MySQL keys become `INDEX … GLOBAL ASYNC`, unique ones (except the primary key) become `INDEX … GLOBAL UNIQUE SYNC`. Tables with synchronous unique indexes automatically switch to transactional `UPSERT` during data loading, because `BulkUpsert` only supports asynchronous indexes.

Examples

`AUTO_INCREMENT` — the MySQL column `id BIGINT AUTO_INCREMENT` becomes:

```
`id` BigSerial NOT NULL,
PRIMARY KEY (`id`)
```

After `CREATE TABLE`, if the value of `information_schema.TABLES.AUTO_INCREMENT` is known:

```
ALTER SEQUENCE `<db>/<table>/_serial_column_id` START WITH <next_value> RESTART
```

Tables without a primary key

In YDB, each table must have a `PRIMARY KEY`. mysql2ydb takes the key **only** from MySQL columns with `COLUMN_KEY = 'PRI'`; `UNIQUE KEY` indexes become secondary `GLOBAL UNIQUE SYNC`, but are **not** substituted for the primary key. If a table has no `PRIMARY KEY`, schema creation fails with an error (`PRIMARY KEY ()` — invalid DDL), and data transfer does not start.

Before migration, you need to set a key in MySQL for such tables, for example:

```
ALTER TABLE my_table ADD COLUMN id BIGINT AUTO_INCREMENT PRIMARY KEY FIRST;
```

or assign `PRIMARY KEY` to an existing column (or set of columns) if it is truly unique.

If you cannot add `PRIMARY KEY` in MySQL, but you can prepare the schema in YDB manually, create the table with the required key yourself and run the utility with the `-data-only` flag. Reading from MySQL without `PRIMARY KEY` in the metadata will go through `LIMIT/OFFSET`: on each page, MySQL rescans all previous rows, so migration time grows quadratically with table size — for large tables, it is better to add a key in MySQL beforehand. The key columns in YDB must be present among the MySQL columns.

If you need automatic import without a key in MySQL, consider YDB Importer — it adds a synthetic column `ydb_synth_key` (SHA-256 of the row). mysql2ydb does not create synthetic keys.

Efficient access to MySQL

The utility minimizes accesses to MySQL and limits the size of each query:

1. **Page-by-page reading by primary key** — `WHERE (pk…) > (?) ORDER BY pk LIMIT n`. Without full table scan into memory.
2. **Read → write pipeline** — one goroutine reads the next batch from MySQL while another writes the previous one to YDB. No more than two batches are in flight at the same time per table.
3. **Adaptive batch size** — the number of rows is limited by available RAM and the average row size from `information_schema`.
4. **Progress checkpoint** — after each successfully written batch, the cursor position and row counter are saved in the YDB service table; a restart continues from the last batch.

```
MySQL: SELECT batch N   →  BulkUpsert(batch N)   →  YDB
MySQL: SELECT batch N+1 →  BulkUpsert(batch N+1) →  YDB
...
```

Features

- **One-to-one schema copy** — original table and column names; mapping of MySQL types to the nearest YDB types (`AUTO_INCREMENT` → `BigSerial`, `UNSIGNED` → `Uint32` / `Uint64`, secondary indexes, `TINYINT(1)` → `Bool`; see the MySQL schema mapping section).
- **Schema creation** — tables in YDB are created from MySQL metadata (`information_schema`).
- **Fragment-by-fragment data migration**: the table is processed in fixed-size chunks; the data volume may exceed available RAM.
- **Upsert by key** — data is written via `BulkUpsert` or transactional `UPSERT`; re-running does not create duplicate rows.
- **Resumption** — progress for each table is stored in YDB; an interrupted migration can be resumed from the last batch.

Installation

Source code and build instructions are in the mysql-ydb-importer repository on GitHub.

Building requires Go version not lower than specified in `go.mod` of the repository:

```
git clone https://github.com/ydb-platform/mysql-ydb-importer.git
cd mysql-ydb-importer
go build -o mysql2ydb ./cmd/mysql2ydb
```

Usage procedure

MySQL connection parameters are read by default from the **~/.my.cnf** file (like the `mysql` client), section `[client]`. The `-mysql` flag overrides the configuration file.

Tables without primary key

Tables without `PRIMARY KEY` in MySQL **are not supported** during automatic schema creation. Before running, add a primary key in MySQL or prepare the schema manually and use `-data-only` (see the section Tables without a primary key).

Example `~/.my.cnf`:

```
[client]
user = myuser
password = mypass
host = localhost
port = 3306
database = mydb
```

If the server requires a secure connection (`require_secure_transport=ON`), add to `[client]`:

```
ssl-mode = REQUIRED
```

For a self-signed certificate (without verification):

```
ssl-mode = REQUIRED
ssl-verify = 0
```

Alternative option — `ssl=1` and `ssl-verify=0`.

Minimum startup when `~/.my.cnf` is present (only `-ydb` is required):

```
./mysql2ydb -ydb "grpc://localhost:2136"
```

With explicit MySQL DSN (ignores `~/.my.cnf`):

```
./mysql2ydb -mysql "user:$MYSQL_PASSWORD@tcp(localhost:3306)/mydb" -ydb "grpc://localhost:2136" -batch-size 1
0000
```

Command line flags

Flag	Description
`-mysql`	DSN MySQL (if specified, overrides `~/.my.cnf`)
`-ydb`	Endpoint YDB (required)
`-ydb-database`	Path to YDB database (default `local`)
`-schema-only`	Only create the schema, without data transfer
`-data-only`	Only transfer data (the schema in YDB must already exist — including one created manually)
`-batch-size`	Target batch size in rows (default 10,000)
`-max-chunk-rows`	Hard upper limit of rows in a batch (default 25,000)
`-parallel-tables`	Number of tables for parallel transfer (default 1)
`-tables`	Comma-separated list of tables (default – all)
`-force`	Migrate all tables from scratch, ignoring the saved state
`-force-recreate`	Delete all tables in YDB and recreate the schema from scratch

Examples

Schema only:

```
./mysql2ydb -mysql "..." -ydb "grpc://localhost:2136" -schema-only
```

Only data (after schema creation):

```
./mysql2ydb -mysql "..." -ydb "grpc://localhost:2136" -data-only -batch-size 5000
```

Selected tables:

```
./mysql2ydb -mysql "..." -ydb "grpc://localhost:2136" -tables "users,orders"
```

Fragment-based reading

- For tables with a **primary key** in MySQL, cursor-based pagination is used: `WHERE (pk > ?) ORDER BY pk LIMIT batch_size`. No more than one batch is kept in memory at a time.
- If there is no `PRIMARY KEY` in MySQL, but the schema in YDB has already been created manually (mode `-data-only`), `LIMIT batch_size OFFSET offset` is applied — see the section Tables without a primary key.

Writing to YDB

By default, data is written via `BulkUpsert`. For tables with synchronous unique indexes, the utility automatically switches to transactional `UPSERT`, since `BulkUpsert` only supports asynchronous indexes.

Queries to YDB are marked as idempotent, so the SDK automatically retries them on network failures. The absence of duplicates when restarting the migration is ensured by overwriting rows by key and saving progress in a service table (see the Features section, the "Resumption" item).

Comparison with YDB Importer

There are other tools for loading data into YDB. YDB Importer is a universal JDBC importer for PostgreSQL, Oracle, SQL Server, and other sources: XML configuration, parallel import, partitioning configuration.

mysql2ydb solves a more specific task — one-to-one migration of a MySQL database using a single binary file, without JVM. For migration only from MySQL with large tables, `~/.my.cnf` and the `-ydb` flag are sufficient; progress is saved in the service table YDB.

	YDB Importer	**mysql2ydb**
Runtime	Java + JDBC drivers + XML configuration	One binary file in Go, native MySQL protocol
Scope	Multiple JDBC sources	Only MySQL
Default batch size	1,000 rows (`max-batch-rows`)	10,000 rows (`-batch-size`), with auto-tuning
Large tables	Parallel ranges, partition buffers	Cursor-based paginated reading, batch size based on RAM
Resumption after failure	Re-running the import	Checkpoint per batch in the service table YDB
Setting	JDBC JAR files, XML configuration	`~/.my.cnf` and two flags

YDB Importer is designed for importing from many JDBC sources into a **configurable** YDB structure (`table-name-format` , `blob-name-format` , date conversion modes, optional synthetic keys). Below are the practical differences when importing the same MySQL database with two tools:

MySQL feature	mysql2ydb (one-to-one copy)	ydb-importer
Table names	As in MySQL (`users` , `orders` , …)	Rename by pattern, e.g., `mysql1/${schema}/${table}` → `mysql1/mydb/users` (configuration example)
Column names	As in MySQL	Sanitization: spaces, `.` , `/` , `` ` `` → `_`
`AUTO_INCREMENT`	`BigSerial` + `ALTER SEQUENCE … START WITH` from `TABLES.AUTO_INCREMENT`	Regular `Int32` / `Int64` ; no `BigSerial` or sequence reset
`INT UNSIGNED` , `BIGINT UNSIGNED` , …	`Uint32` / `Uint64`	JDBC maps integers to signed `Int32` / `Int64`
`TINYINT(1)`	`Bool`	`TINYINT` → `Int32` (`BOOLEAN` → `Bool`)
`BIT`	`Uint64`	`Bool` (JDBC `Types.BIT`)
Secondary `KEY` / `UNIQUE KEY`	`INDEX … GLOBAL ASYNC` / `GLOBAL UNIQUE SYNC` in `CREATE TABLE`	Only `PRIMARY KEY` in DDL; secondary indexes are not created
`ENUM` , `SET` , `JSON`	`Text`	`Text` (similar)
`BLOB` / `BINARY`	Same column, value inline in `String` (bytes)	Column becomes `Int64` (blob identifier); data is moved to a **separate table** `${schema}/${table}_${field}` with `(id, pos, val)` rows in 64 KB blocks
`TEXT` / `CLOB` (large text)	Inline `Text` in the main table	Optionally a separate CLOB table (32 K character blocks) or inline `Text` depending on settings
Table **without a primary key**	**Not supported** — `CREATE TABLE` fails with an error; add `PRIMARY KEY` in MySQL before migration	A column `ydb_synth_key Text` is added as PK (SHA-256 of the row); **duplicates are collapsed** into one row
`DATE`	`Date`	`Date32` by default (`conv-date=DATE_NEW`)
`DATETIME` / `TIMESTAMP`	`Timestamp`	`Datetime64` / `Timestamp64` by default
`TIME`	`Text` (fallback)	`Int32` (seconds since midnight)
`DECIMAL(p,s)`	`Decimal(22, 9)` for all	`Decimal(p,s)` when `allow-custom-decimal=true`
`YEAR`	`Uint16`	Not covered in MySQL type tests
Partitioning	Only `AUTO_PARTITIONING_BY_LOAD`	`PARTITION_AT_KEYS` , splitting by source partitions, `HASH` for column store, etc.

Working with BLOBs

MySQL table `attachments (id INT, data MEDIUMBLOB)` :

	Column `data` in the main table	Where bytes are stored
mysql2ydb	`data String`	In the same row
ydb-importer	`data Int64` (reference)	Separate table `…/attachments_data` with chunked rows

With mysql2ydb, you can perform `SELECT data FROM attachments WHERE id = 1` the same way as in MySQL. With ydb-importer, you need to join the main table with an additional blob table.

Schema migrations

Environment	Instruction
goose	Instruction
Liquibase	Instruction
Flyway	Instruction
dbt	Instruction

Integration of dbt with YDB

Introduction

dbt (data build tool) is a tool for data transformation in analytical data platforms, designed to organize data model development using software engineering practices. It covers the transformation (T) stage in ETL pipelines and enables structured modeling of data logic — from raw data to data marts — with support for testing, documentation, and data lineage, providing end-to-end traceability of data from source to final model.

Models in dbt are defined uniformly as `SELECT` statements, simplifying their maintenance and composability. The tool handles their materialization in the target system and tracks dependencies between models, including the evolution of data across transformation stages.

To integrate dbt with YDB, the dbt-ydb connector is used. It enables connecting to YDB as a target platform and supports model materialization as tables and views, incremental data processing, loading test datasets (seeds), as well as running data tests and generating documentation.

This section describes the connector's capabilities and provides steps to set up and start using it.

Warning

The dbt-ydb connector is in the Preview stage and does not currently support all dbt features. The sections below list the supported features and known limitations.

Features

Models and Their Materialization

A core concept in dbt is a data model. A model is essentially a SQL query that can reference any data source in your data warehouse, including other models. The dbt-ydb connector supports the following approaches to materializing models in YDB:

1. **View** — stored as a YDB view.
2. **Table** — stored as a table in YDB and re-created by dbt on each model update.

   The dbt-ydb connector allows you to specify the following table parameters using model configuration:

Parameter	Required	Default Value	Description
primary_key	Yes		Primary key of the table
store_type	No	`row`	Table type. `row` for row-oriented table or `column` for column-oriented table
auto_partitioning_by_size	No		Automatic partitioning by size
auto_partitioning_partition_size_mb	No		Partition size threshold
ttl	No		Time-To-Live rule

   Example of a model materialized as a table based on another model (using ref).
   Configured with a primary key, TTL, and automatic partitioning by size.

   ```
   {{ config(
      primary_key='id, created_at',
      store_type='row',
      auto_partitioning_by_size='ENABLED',
      auto_partitioning_partition_size_mb=256,
      ttl='Interval("P30D") on created_at'
   ) }}

   select
      id,
      name,
      created_at
   from {{ ref('source_table') }}
   ```

3. Incremental model — created as a table inside YDB, but instead of being recreated, it is updated with changed and new rows when refreshed by dbt.

   The dbt-ydb connector supports the same parameters as for table materialization, plus unique parameters for the incremental model:

Parameter	Required	Default Value	Description
incremental_strategy	No	`MERGE`	Incremental materialization strategy. The `MERGE` strategy is supported, using the YDB `UPSERT` operation. `APPEND` strategy support is under development.

Note

Another materialization type, ephemeral model, is not currently supported by the connector.

Snapshots

The snapshot mechanism is not currently supported by dbt-ydb.

Seeds (CSV-based reference/test data)

The dbt-ydb connector supports dbt seeds to upload reference and test data from CSV files into your project and use them in other models.

Data Testing

dbt-ydb supports standard dbt data tests, as well as singular tests within the capabilities of YQL.

Documentation Generation

dbt-ydb supports generating documentation from dbt projects for YDB.

Getting Started

Requirements

To start working with dbt on YDB, you will need:

- Python 3.10+;
- dbt Core 1.8+;
- An existing YDB cluster (a single-node installation from the quickstart is sufficient).

> **Note**
>
> dbt Fusion 2.0 is not supported at this time.

Installation

To install dbt-ydb, run:

```
pip install dbt-ydb
```

Connecting dbt to a YDB Cluster

dbt connects to YDB via the dbt-ydb connector using the standard way for YDB. To connect successfully, specify the endpoint, database path, and authentication parameters in the dbt profiles file.

Example profile file with possible authentication options and default values (in square brackets):

```
profile_name:
target: dev
outputs:
   dev:
      type: ydb
      host: [localhost] # YDB host
      port: [2136] # YDB port
      database: [/local] # YDB database
      schema: [<empty string>] # Optional subfolder for DBT models
      secure: [False] # If enabled, grpcs protocol will be used
      root_certificates_path: [<empty string>] # Optional path to root certificates file

      # Static Credentials
      username: [<empty string>]
      password: [<empty string>]

      # Access Token Credentials
      token: [<empty string>]

      # Service Account Credentials
      service_account_credentials_file: [<empty string>]
```

Creating a Project from Scratch via dbt init

1. Initialize a project:

   ```
   dbt init
   ```

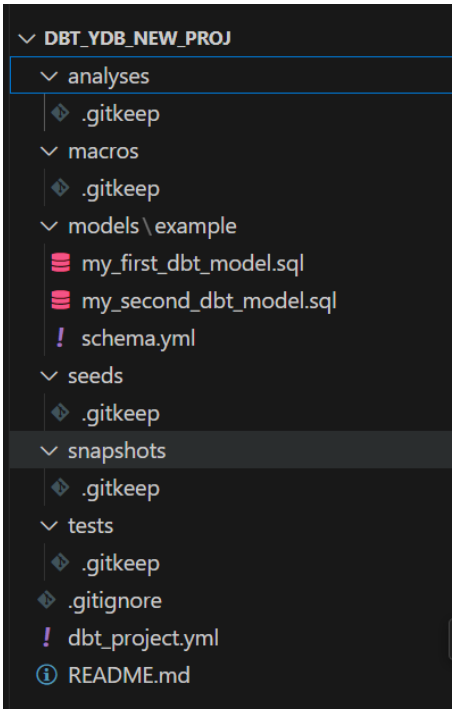
2. Follow dbt's interactive prompts to select the dbt-ydb connector and authentication settings for your YDB cluster.
3. As a result, your project directory will be created along with a dbt profiles file in your home directory, containing a new connection to YDB:

   ```
   ~/.dbt/profiles.yml
   ```

4. Run `dbt debug` to verify the connection:

   ```
   dbt debug
   ```

5. Inside your project directory, you will find the following structure:



6. Adapt the model `my_first_dbt_model`.

   Currently, dbt does not support customizing the auto-generated example per connector. Therefore, to run this model with dbt-ydb, you need to update it as follows:

```
/*
   Welcome to your first dbt model!
   Did you know that you can also configure models directly within SQL files?
   This will override configurations stated in dbt_project.yml

   Try changing "table" to "view" below
*/

{{ config(materialized='table', primary_key='id') }}

select *
from (
   select 1 as id
   union all
   select null as id
)
```

7. Now you can run your project:

```
dbt run
```

Running the Example Project

The dbt-ydb connector comes with an example you can use to quickly test dbt functionality with YDB.

1. Clone the repository:

```
git clone https://github.com/ydb-platform/dbt-ydb.git
cd dbt-ydb/examples/jaffle_shop
```

2. Configure the connection profile in the `profiles.yml` file. For a single-node installation from the quickstart, the file should look like this:

```
profile_name:
   target: dev
   outputs:
      dev:
      type: ydb
      host: localhost # YDB host
      port: 2136 # YDB port
      database: /local # YDB database
      schema: jaffle_shop
```

3. Verify the connection:

```
dbt debug
```

4. Load test data (via seeds):

   This command will load CSV files from `data/` into `raw_*` tables in YDB.

```
dbt seed
```

5. Run models:

   This will create tables and views based on the project's example models.

   ```
   dbt run
   ```

6. Test model data:

   This will run standard data tests described in the example — such as checks for `null`, allowed values lists, and others.

   ```
   dbt test
   ```

7. Generate documentation and start a local web server to view it:

   The project documentation will be available in your browser at http://localhost:8080.

   ```
   dbt docs generate
   dbt docs serve --port 8080
   ```

Next Steps

You can find the official dbt documentation here.
Additionally, you can explore the connector's source code and contribute to its development in the public dbt-ydb repository on GitHub.

Migrating YDB data schemas with the Flyway migration tool

Introduction

Flyway is an open-source database migration tool. It strongly favors simplicity and convention over configuration. It has extensions for various database management systems (DBMS), including YDB.

Install

To use Flyway with YDB in a Java / Kotlin application or a Gradle / Maven plugin, you need to add dependencies for the Flyway core, the Flyway extension for YDB, and the YDB JDBC Driver:

Maven

```
<!-- Set actual versions -->
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
    <version>${flyway.core.version}</version>
</dependency>

<dependency>
    <groupId>tech.ydb.jdbc</groupId>
    <artifactId>ydb-jdbc-driver</artifactId>
    <version>${ydb.jdbc.version}</version>
</dependency>

<dependency>
    <groupId>tech.ydb.dialects</groupId>
    <artifactId>flyway-ydb-dialect</artifactId>
    <version>${flyway.ydb.dialect.version}</version>
</dependency>
```

Gradle

```
dependencies {
    // Set actual versions
    implementation "org.flywaydb:flyway-core:$flywayCoreVersion"
    implementation "tech.ydb.dialects:flyway-ydb-dialect:$flywayYdbDialecVersion"
    implementation "tech.ydb.jdbc:ydb-jdbc-driver:$ydbJdbcVersion"
}
```

To work with YDB via Flyway CLI, you need to install the `flyway` utility itself using one of the recommended methods.

Then the utility must be extended with the YDB dialect and the JDBC driver:

```
# install flyway
# cd $(which flyway) // prepare this command for your environment

cd libexec
# set actual versions of .jar files

cd drivers && curl -L -o ydb-jdbc-driver-shaded-2.1.0.jar https://repo.maven.apache.org/maven2/tech/ydb/jdbc/
ydb-jdbc-driver-shaded/2.1.0/ydb-jdbc-driver-shaded-2.1.0.jar

cd ..

cd lib && curl -L -o flyway-ydb-dialect.jar https://repo.maven.apache.org/maven2/tech/ydb/dialects/flyway-ydb
-dialect/1.0.0-RC0/flyway-ydb-dialect-1.0.0-RC0.jar
```

Note

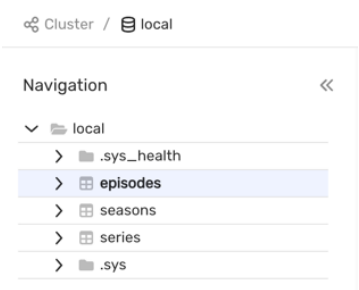
Flyway Desktop is currently not supported.

Migration management using Flyway

baseline

Command baseline initializes Flyway in an existing database, excluding all migrations up to and including the `baselineVersion`.

Suppose we have an existing project with the current database schema:



Let's write down our existing migrations as follows:

```
db/migration:
  V1__create_series.sql
  V2__create_seasons.sql
  V3__create_episodes.sql
```

Contents of SQL files:

V1__create_series.sql

```
CREATE TABLE series -- series is the table name.
(                           -- Must be unique within the folder.
    series_id    Int64,
    title        Text,
    series_info  Text,
    release_date Int64,
    PRIMARY KEY (series_id) -- The primary key is a column or
    -- combination of columns that uniquely identifies
    -- each table row (contains only
    -- non-repeating values). A table can have
    -- only one primary key. For every table
    -- in YDB, the primary key is required.
);
```

V2__create_seasons.sql

```
CREATE TABLE seasons
(
    series_id Uint64,
    season_id Uint64,
    title Utf8,
    first_aired Uint64,
    last_aired Uint64,
    PRIMARY KEY (series_id, season_id)
);
```

V3__create_episodes.sql

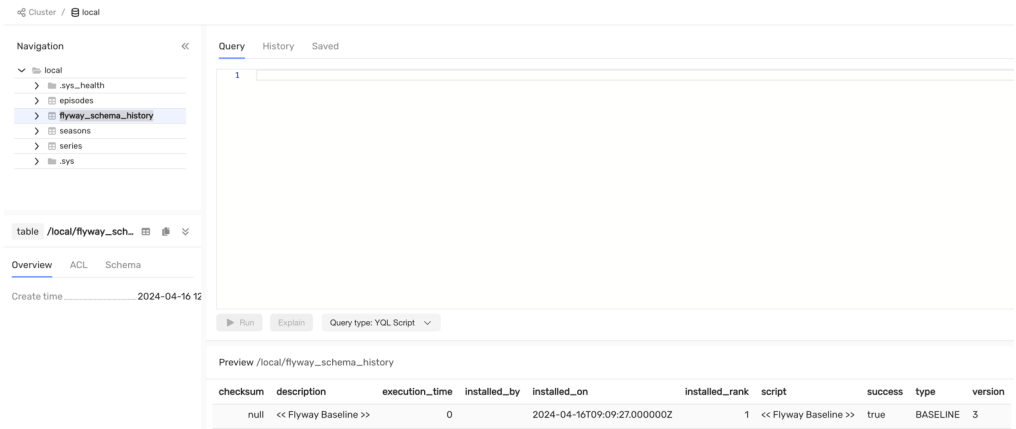
```
CREATE TABLE episodes
(
    series_id Uint64,
    season_id Uint64,
    episode_id Uint64,
    title Utf8,
    air_date Uint64,
    PRIMARY KEY (series_id, season_id, episode_id)
);
```

Set `baselineVersion = 3`, then run the following command:

```
flyway -url=jdbc:ydb:grpc://localhost:2136/local -locations=db/migration -baselineVersion=3 baseline
```

> **Note**
>
> All examples use a Docker container, which does not require any additional authentication parameters.
>
> You can see how to connect to YDB in the next section.

As a result, a table named `flyway_schema_history` will be created, and it will contain a `baseline` record:



migrate

Command migrate evolves the database schema to the latest version. Flyway will create the schema history table automatically if it doesn't exist.

Let's add the migration of data downloads to the previous example:

```
db/migration:
  V1__create_series.sql
  V2__create_seasons.sql
  V3__create_episodes.sql
  V4__load_data.sql
```

The contents of `V4__load_data.sql`

```
INSERT INTO series (series_id, title, release_date, series_info)
VALUES

    -- By default, numeric literals have type Int32
    -- if the value is within the range.
    -- Otherwise, they automatically expand to Int64.
    (1,
     "IT Crowd",
     CAST(Date ("2006-02-03") AS Uint64), -- CAST converts one datatype into another.
        -- You can convert a string
        -- literal into a primitive literal.
        -- The Date() function converts a string
        -- literal in ISO 8601 format into a date.

     "The IT Crowd is a British sitcom produced by Channel 4, written by Graham Linehan, produced by Ash
Atalla and starring Chris O'Dowd, Richard Ayoade, Katherine Parkinson, and Matt Berry."),
    (2,
     "Silicon Valley",
     CAST(Date ("2014-04-06") AS Uint64),
     "Silicon Valley is an American comedy television series created by Mike Judge, John Altschuler and D
ave Krinsky. The series focuses on five young men who founded a startup company in Silicon Valley.")
;

INSERT INTO seasons (series_id, season_id, title, first_aired, last_aired)
VALUES (1, 1, "Season 1", CAST(Date ("2006-02-03") AS Uint64), CAST(Date ("2006-03-03") AS Uint64)),
       (1, 2, "Season 2", CAST(Date ("2007-08-24") AS Uint64), CAST(Date ("2007-09-28") AS Uint64)),
       (1, 3, "Season 3", CAST(Date ("2008-11-21") AS Uint64), CAST(Date ("2008-12-26") AS Uint64)),
       (1, 4, "Season 4", CAST(Date ("2010-06-25") AS Uint64), CAST(Date ("2010-07-30") AS Uint64)),
       (2, 1, "Season 1", CAST(Date ("2014-04-06") AS Uint64), CAST(Date ("2014-06-01") AS Uint64)),
       (2, 2, "Season 2", CAST(Date ("2015-04-12") AS Uint64), CAST(Date ("2015-06-14") AS Uint64)),
       (2, 3, "Season 3", CAST(Date ("2016-04-24") AS Uint64), CAST(Date ("2016-06-26") AS Uint64)),
       (2, 4, "Season 4", CAST(Date ("2017-04-23") AS Uint64), CAST(Date ("2017-06-25") AS Uint64)),
       (2, 5, "Season 5", CAST(Date ("2018-03-25") AS Uint64), CAST(Date ("2018-05-13") AS Uint64))
;

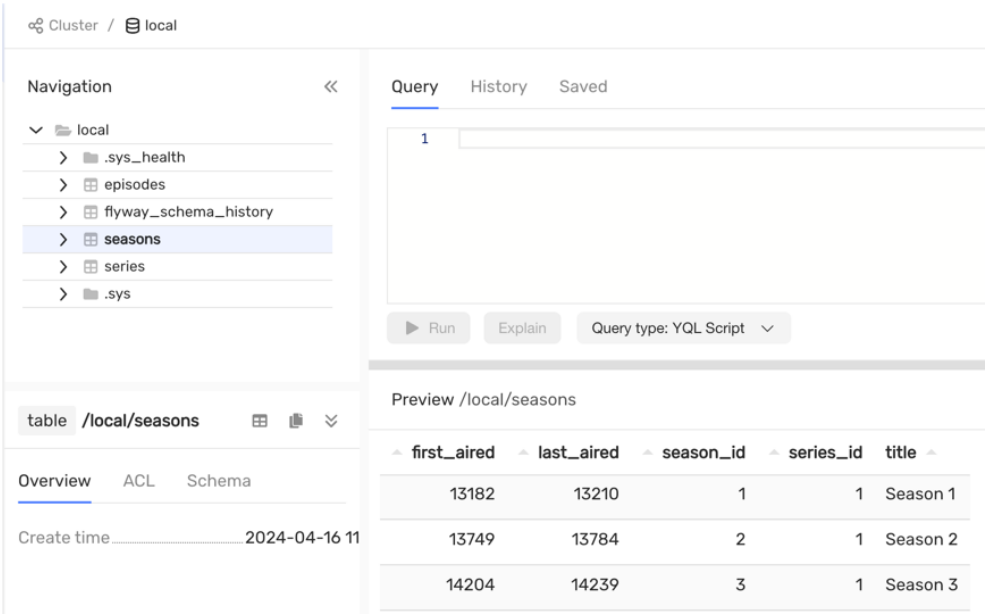
INSERT INTO episodes (series_id, season_id, episode_id, title, air_date)
VALUES (1, 1, 1, "Yesterday's Jam", CAST(Date ("2006-02-03") AS Uint64)),
       (1, 1, 2, "Calamity Jen", CAST(Date ("2006-02-03") AS Uint64)),
       (1, 1, 3, "Fifty-Fifty", CAST(Date ("2006-02-10") AS Uint64)),
       (1, 1, 4, "The Red Door", CAST(Date ("2006-02-17") AS Uint64)),
       (1, 1, 5, "The Haunting of Bill Crouse", CAST(Date ("2006-02-24") AS Uint64)),
       (1, 1, 6, "Aunt Irma Visits", CAST(Date ("2006-03-03") AS Uint64)),
       (1, 2, 1, "The Work Outing", CAST(Date ("2006-08-24") AS Uint64)),
       (1, 2, 2, "Return of the Golden Child", CAST(Date ("2007-08-31") AS Uint64)),
       (1, 2, 3, "Moss and the German", CAST(Date ("2007-09-07") AS Uint64)),
       (1, 2, 4, "The Dinner Party", CAST(Date ("2007-09-14") AS Uint64)),
       (1, 2, 5, "Smoke and Mirrors", CAST(Date ("2007-09-21") AS Uint64)),
       (1, 2, 6, "Men Without Women", CAST(Date ("2007-09-28") AS Uint64)),
       (1, 3, 1, "From Hell", CAST(Date ("2008-11-21") AS Uint64)),
       (1, 3, 2, "Are We Not Men?", CAST(Date ("2008-11-28") AS Uint64)),
       (1, 3, 3, "Tramps Like Us", CAST(Date ("2008-12-05") AS Uint64)),
       (1, 3, 4, "The Speech", CAST(Date ("2008-12-12") AS Uint64)),
       (1, 3, 5, "Friendface", CAST(Date ("2008-12-19") AS Uint64)),
       (1, 3, 6, "Calendar Geeks", CAST(Date ("2008-12-26") AS Uint64)),
       (1, 4, 1, "Jen The Fredo", CAST(Date ("2010-06-25") AS Uint64)),
       (1, 4, 2, "The Final Countdown", CAST(Date ("2010-07-02") AS Uint64)),
       (1, 4, 3, "Something Happened", CAST(Date ("2010-07-09") AS Uint64)),
       (1, 4, 4, "Italian For Beginners", CAST(Date ("2010-07-16") AS Uint64)),
       (1, 4, 5, "Bad Boys", CAST(Date ("2010-07-23") AS Uint64)),
       (1, 4, 6, "Reynholm vs Reynholm", CAST(Date ("2010-07-30") AS Uint64)),
       (2, 1, 1, "Minimum Viable Product", CAST(Date ("2014-04-06") AS Uint64)),
       (2, 1, 2, "The Cap Table", CAST(Date ("2014-04-13") AS Uint64)),
       (2, 1, 3, "Articles of Incorporation", CAST(Date ("2014-04-20") AS Uint64)),
       (2, 1, 4, "Fiduciary Duties", CAST(Date ("2014-04-27") AS Uint64)),
       (2, 1, 5, "Signaling Risk", CAST(Date ("2014-05-04") AS Uint64)),
       (2, 1, 6, "Third Party Insourcing", CAST(Date ("2014-05-11") AS Uint64)),
       (2, 1, 7, "Proof of Concept", CAST(Date ("2014-05-18") AS Uint64)),
       (2, 1, 8, "Optimal Tip-to-Tip Efficiency", CAST(Date ("2014-06-01") AS Uint64)),
       (2, 2, 1, "Sand Hill Shuffle", CAST(Date ("2015-04-12") AS Uint64)),
       (2, 2, 2, "Runaway Devaluation", CAST(Date ("2015-04-19") AS Uint64)),
       (2, 2, 3, "Bad Money", CAST(Date ("2015-04-26") AS Uint64)),
       (2, 2, 4, "The Lady", CAST(Date ("2015-05-03") AS Uint64)),
       (2, 2, 5, "Server Space", CAST(Date ("2015-05-10") AS Uint64)),
       (2, 2, 6, "Homicide", CAST(Date ("2015-05-17") AS Uint64)),
       (2, 2, 7, "Adult Content", CAST(Date ("2015-05-24") AS Uint64)),
       (2, 2, 8, "White Hat/Black Hat", CAST(Date ("2015-05-31") AS Uint64)),
       (2, 2, 9, "Binding Arbitration", CAST(Date ("2015-06-07") AS Uint64)),
       (2, 2, 10, "Two Days of the Condor", CAST(Date ("2015-06-14") AS Uint64)),
       (2, 3, 1, "Founder Friendly", CAST(Date ("2016-04-24") AS Uint64)),
       (2, 3, 2, "Two in the Box", CAST(Date ("2016-05-01") AS Uint64)),
```

```
        (2, 3, 3, "Meinertzhagen's Haversack", CAST(Date ("2016-05-08") AS Uint64)),
        (2, 3, 4, "Maleant Data Systems Solutions", CAST(Date ("2016-05-15") AS Uint64)),
        (2, 3, 5, "The Empty Chair", CAST(Date ("2016-05-22") AS Uint64)),
        (2, 3, 6, "Bachmanity Insanity", CAST(Date ("2016-05-29") AS Uint64)),
        (2, 3, 7, "To Build a Better Beta", CAST(Date ("2016-06-05") AS Uint64)),
        (2, 3, 8, "Bachman's Earnings Over-Ride", CAST(Date ("2016-06-12") AS Uint64)),
        (2, 3, 9, "Daily Active Users", CAST(Date ("2016-06-19") AS Uint64)),
        (2, 3, 10, "The Uptick", CAST(Date ("2016-06-26") AS Uint64)),
        (2, 4, 1, "Success Failure", CAST(Date ("2017-04-23") AS Uint64)),
        (2, 4, 2, "Terms of Service", CAST(Date ("2017-04-30") AS Uint64)),
        (2, 4, 3, "Intellectual Property", CAST(Date ("2017-05-07") AS Uint64)),
        (2, 4, 4, "Teambuilding Exercise", CAST(Date ("2017-05-14") AS Uint64)),
        (2, 4, 5, "The Blood Boy", CAST(Date ("2017-05-21") AS Uint64)),
        (2, 4, 6, "Customer Service", CAST(Date ("2017-05-28") AS Uint64)),
        (2, 4, 7, "The Patent Troll", CAST(Date ("2017-06-04") AS Uint64)),
        (2, 4, 8, "The Keenan Vortex", CAST(Date ("2017-06-11") AS Uint64)),
        (2, 4, 9, "Hooli-Con", CAST(Date ("2017-06-18") AS Uint64)),
        (2, 4, 10, "Server Error", CAST(Date ("2017-06-25") AS Uint64)),
        (2, 5, 1, "Grow Fast or Die Slow", CAST(Date ("2018-03-25") AS Uint64)),
        (2, 5, 2, "Reorientation", CAST(Date ("2018-04-01") AS Uint64)),
        (2, 5, 3, "Chief Operating Officer", CAST(Date ("2018-04-08") AS Uint64)),
        (2, 5, 4, "Tech Evangelist", CAST(Date ("2018-04-15") AS Uint64)),
        (2, 5, 5, "Facial Recognition", CAST(Date ("2018-04-22") AS Uint64)),
        (2, 5, 6, "Artificial Emotional Intelligence", CAST(Date ("2018-04-29") AS Uint64)),
        (2, 5, 7, "Initial Coin Offering", CAST(Date ("2018-05-06") AS Uint64)),
        (2, 5, 8, "Fifty-One Percent", CAST(Date ("2018-05-13") AS Uint64));
```

Let's apply the latest migration using the following command:

```
flyway -url=jdbc:ydb:grpc://localhost:2136/local -locations=db/migration migrate
```

As a result, `series`, `season`, and `episode` tables will be created and filled with data:



Then, we evolve the schema by adding a secondary index to the `series` table:

```
db/migration:
  V1__create_series.sql
  V2__create_seasons.sql
  V3__create_episodes.sql
  V4__load_data.sql
  V5__create_series_title_index.sql
```

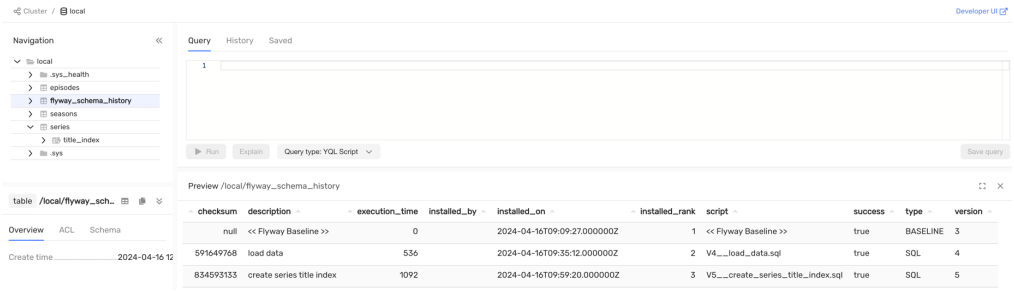
The contents of `V5__create_series_title_index.sql`

```
ALTER TABLE `series` ADD INDEX `title_index` GLOBAL ON (`title`);
```

Let's apply the latest migration using the following command:

```
flyway -url=jdbc:ydb:grpc://localhost:2136/local -locations=db/migration migrate
```

As a result, a secondary index for the `series` table will be created:



info

Command info prints the details and status information about all the migrations.

Let's add another migration that renames the previously added secondary index:

```
db/migration:
  V1__create_series.sql
  V2__create_seasons.sql
  V3__create_episodes.sql
  V4__load_data.sql
  V5__create_series_title_index.sql
  V6__rename_series_title_index.sql
```

The contents of `V6__rename_series_title_index.sql`

```
ALTER TABLE `series` RENAME INDEX `title_index` TO `title_index_new`;
```

The result of executing the `flyway -url=jdbc:ydb:grpc://localhost:2136/local -locations=db/migration info` will provide detailed information about the status of the migrations:

```
+-----------+---------+---------------------------+----------+---------------------+--------------------+----
------+
| Category  | Version | Description               | Type     | Installed On        | State              | Und
oable |
+-----------+---------+---------------------------+----------+---------------------+--------------------+----
------+
| Versioned | 1       | create series             | SQL      |                     | Below Baseline     | No
|
| Versioned | 2       | create seasons            | SQL      |                     | Below Baseline     | No
|
| Versioned | 3       | create episodes           | SQL      |                     | Ignored (Baseline) | No
|
|           | 3       | << Flyway Baseline >>     | BASELINE | 2024-04-16 12:09:27 | Baseline           | No
|
| Versioned | 4       | load data                 | SQL      | 2024-04-16 12:35:12 | Success            | No
|
| Versioned | 5       | create series title index | SQL      | 2024-04-16 12:59:20 | Success            | No
|
| Versioned | 6       | rename series title index | SQL      |                     | Pending            | No
|
+-----------+---------+---------------------------+----------+---------------------+--------------------+----
------+
```

validate

Command validate validates the applied migrations against the available ones.

After applying the `flyway -url=jdbc:ydb:grpc://localhost:2136/local -locations=db/migration validate` command with current migrations, the logs will show that the latest migration was not applied to our database:

```
ERROR: Validate failed: Migrations have failed validation
Detected resolved migration not applied to database: 6.
To fix this error, either run migrate, or set -ignoreMigrationPatterns='*:pending'.
```

Let's apply this error by executing `flyaway .. migrate` again. The validation will now be successful, and the secondary index will be renamed.

Next, we will modify the file of the previously applied migration `V4__load_date.sql` by deleting comments from the SQL script.

After executing the validation command, we got a logical error because the `checksum` differed in the modified migration:

```
ERROR: Validate failed: Migrations have failed validation
Migration checksum mismatch for migration version 4
-> Applied to database : 591649768
-> Resolved locally    : 1923849782
```

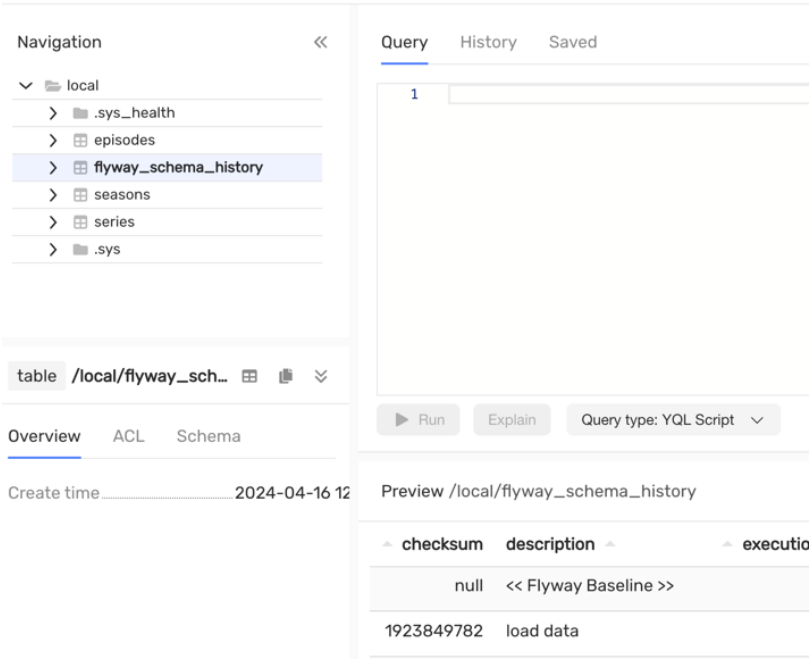
repair

Command repair tries to fix the identified errors and discrepancies from the database schema history table.

Fix the problem with different `checksum`'s by running the following command:

```
flyway -url=jdbc:ydb:grpc://localhost:2136/local -locations=db/migration repair
```

The result will be an update to the `checksum` column in the `flyway_schema_history` table for the migration entry `V4__load_data.sql`:



After restoring the log table, validation is successful.

Also, using the `repair` command, you can delete a failed DDL script.

clean

Command clean drops all objects in the configured schemas.

> **Warning**
>
> Unlike other database management systems, YDB does not have a concept of `SCHEMA`. Thus, the `clean` command **drops all the user tables** in a given database.

Let's delete all the tables in our database using the following command:

```
flyway -url=jdbc:ydb:grpc://localhost:2136/local -locations=db/migration -cleanDisabled=false clean
```

The result will be an empty database:

Data schema versioning and migration in YDB using "goose"

Introduction

Goose is an open-source tool that helps to version the data schema in the database and manage migrations between these versions. Goose supports many different database management systems, including YDB. Goose uses migration files and stores the state of migrations directly in the database in a special table.

Install goose

Goose installation options are described in its documentation.

Launch arguments goose

After installation, the `goose` command line utility can be called:

```
$ goose <DB> <CONNECTION_STRING> <COMMAND> <COMMAND_ARGUMENTS>
```

Where:

- `<DB>` - database engine, for YDB you should write `goose ydb`
- `<CONNECTION_STRING>` - database connection string.
- `<COMMAND>` - the command to be executed. A complete list of commands is available in the built-in help (`goose help`).
- `<COMMAND_ARGUMENTS>` - command arguments.

YDB connection string

To connect to YDB you should use a connection string like:

```
<protocol>://<host>:<port>/<database_path>?go_query_mode=scripting&go_fake_tx=scripting&go_query_bind=declar
e,numeric
```

Where:

- `<protocol>` - connection protocol (`grpc` for an unsecured connection or `grpcs` for a secure TLS connection). The secure connection requires certificates. You should declare certificates like this: `export YDB_SSL_ROOT_CERTIFICATES_FILE=/path/to/ydb/certs/CA.pem` .
- `<host>` - hostname for connecting to YDB cluster.
- `<port>` - port for connecting to YDB cluster.
- `<database_path>` - database in the YDB cluster.
- `go_query_mode=scripting` - special `scripting` mode for executing queries by default in the YDB driver. In this mode, all requests from goose are sent to the YDB `scripting` service, which allows processing of both DDL and DML SQL statements.
- `go_fake_tx=scripting` - support for transaction emulation in query execution mode through the YDB `scripting` service. The fact is that in YDB, executing `DDL` `SQL` statements in a transaction is impossible (or incurs significant overhead). In particular, the `scripting` service does not allow interactive transactions (with explicit `Begin` + `Commit` / `Rollback`). Accordingly, the transaction emulation mode does not actually do anything (`noop`) on the `Begin` + `Commit` / `Rollback` calls from `goose` . This trick can, in rare cases, cause an individual migration step to end up in an intermediate state. The YDB team is working on a new `query` service that should eliminate this risk.
- `go_query_bind=declare,numeric` - support for bindings of auto-inference of YQL types from query parameters (`declare`) and support for bindings of numbered parameters (`numeric`). YQL is a strongly typed language that requires you to explicitly specify the types of query parameters in the body of the `SQL` query itself using the special `DECLARE` statement. Also, YQL only supports named query parameters (for example, `$my_arg`), while the goose core generates SQL queries with numbered parameters (`$1` , `$2` , etc.) . The `declare` and `numeric` bindings modify the original queries from `goose` at the YDB driver level.

If connecting to a local YDB docker container, the connection string could look like:

```
grpc://localhost:2136/local?go_query_mode=scripting&go_fake_tx=scripting&go_query_bind=declare,numeric
```

Let's store this connection string to an environment variable to re-use it later:

```
export YDB_CONNECTION_STRING="grpc://localhost:2136/local?go_query_mode=scripting&go_fake_tx=scripting&go_que
ry_bind=declare,numeric"
```

Further examples of calling `goose` commands will contain exactly this connection string.

Directory with migration files

Let's create a migrations directory and then all `goose` commands should be executed in this directory:

```
$ mkdir migrations && cd migrations
```

Managing migrations using goose

Creating migration files and applying to database

The migration file can be generated using the `goose create` command:

```
$ goose ydb $YDB_CONNECTION_STRING create 00001_create_first_table sql
2024/01/12 11:52:29 Created new file: 20240112115229_00001_create_first_table.sql
```

This means that the tool has created a new migration file `<timestamp>_00001_create_table_users.sql` :

```
$ ls
20231215052248_00001_create_table_users.sql
```

After executing the `goose create` command, a migration file `<timestamp>_00001_create_table_users.sql` will be created with the following content :

```
-- +goose Up
-- +goose StatementBegin
SELECT 'up SQL query';
-- +goose StatementEnd

-- +goose Down
-- +goose StatementBegin
SELECT 'down SQL query';
-- +goose StatementEnd
```

This migration file structure helps keep the instructions that lead to the next version of the database in context. It is also easy, without unnecessary distractions, to write instructions that roll back a database change.

The migration file consists of two sections:

1. `+goose Up` is an area where we can record the migration steps.
2. `+goose Down` is an area where we can write queries to revert changes of the `+goose Up` steps.

Goose carefully inserted placeholder queries:

```
SELECT 'up SQL query';
```

And

```
SELECT 'down SQL query';
```

So that we can replace them with the real migration SQL queries for change the schema for the YDB database, which is accessible through the corresponding connection string.

Let's edit the migration file `<timestamp>_00001_create_table_users.sql` so that when applying the migration we create a table with necessary schema, and when rolling back the migration we delete the created table:

```
-- +goose Up
-- +goose StatementBegin
CREATE TABLE users (
     id Uint64,
     username Text,
     created_at Timestamp,
     PRIMARY KEY (id)
);
-- +goose StatementEnd

-- +goose Down
-- +goose StatementBegin
DROP TABLE users;
-- +goose StatementEnd
```

We can check status of migrations:

```
$ goose ydb $YDB_CONNECTION_STRING status
2024/01/12 11:53:50     Applied At                  Migration
2024/01/12 11:53:50     =======================================
2024/01/12 11:53:50     Pending                  -- 20240112115229_00001_create_first_table.sql
```

Status `Pending` of migration means that migration has not been applied yet. You can apply such migrations with commands `goose up` or `goose up-by-one`.

Let's apply migration using `goose up`:

```
$ goose ydb $YDB_CONNECTION_STRING up
2024/01/12 11:55:18 OK   20240112115229_00001_create_first_table.sql (93.58ms)
2024/01/12 11:55:18 goose: successfully migrated database to version: 20240112115229
```

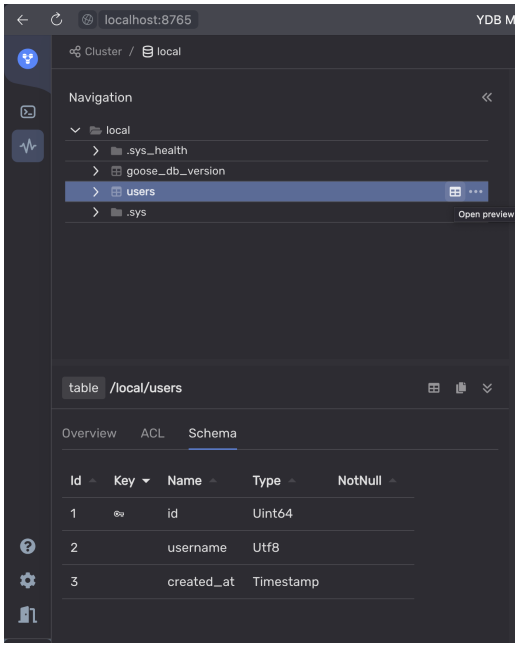
Let's check the status of migration through `goose status`:

```
$ goose ydb $YDB_CONNECTION_STRING status
2024/01/12 11:56:00     Applied At                  Migration
2024/01/12 11:56:00     =======================================
2024/01/12 11:56:00     Fri Jan 12 11:55:18 2024 -- 20240112115229_00001_create_first_table.sql
```

Status `Pending` changed to timestamp `Fri Jan 12 11:55:18 2024` - this means that migration applied.

There are alternative options to see the applied changes:

Using YDB UI on `http://localhost:8765`



Using YDB CLI

```
$ ydb -e grpc://localhost:2136 -d /local scheme describe users
<table> users

Columns:
┌────────────┬────────────┬────────┬─────┐
| Name       | Type       | Family | Key |
├────────────┼────────────┼────────┼─────┤
| id         | Uint64?    |        | K0  |
| username   | Utf8?      |        |     |
| created_at | Timestamp? |        |     |
└────────────┴────────────┴────────┴─────┘

Storage settings:
Store large values in "external blobs": false

Column families:
┌─────────┬──────┬─────────────┬──────────────────┐
| Name    | Data | Compression | Keep in memory |
├─────────┼──────┼─────────────┼──────────────────┤
| default |      | None        |                  |
└─────────┴──────┴─────────────┴──────────────────┘

Auto partitioning settings:
Partitioning by size: true
Partitioning by load: false
Preferred partition size (Mb): 2048
Min partitions count: 1
```

Let's make the second migration that adds a column `password_hash` to the `users` table:

```
$ goose ydb $YDB_CONNECTION_STRING create 00002_add_column_password_hash_into_table_users sql
2024/01/12 12:00:57 Created new file: 20240112120057_00002_add_column_password_hash_into_table_users.sql
```

Let's edit the migration file `<timestamp>_00002_add_column_password_hash_into_table_users.sql`:

```
-- +goose Up
-- +goose StatementBegin
ALTER TABLE users ADD COLUMN password_hash Text;
-- +goose StatementEnd

-- +goose Down
-- +goose StatementBegin
ALTER TABLE users DROP COLUMN password_hash;
-- +goose StatementEnd
```

We can check the migration statuses again:

```
$ goose ydb $YDB_CONNECTION_STRING status
2024/01/12 12:02:40     Applied At                  Migration
2024/01/12 12:02:40     =======================================
2024/01/12 12:02:40     Fri Jan 12 11:55:18 2024 -- 20240112115229_00001_create_first_table.sql
2024/01/12 12:02:40     Pending                  -- 20240112120057_00002_add_column_password_hash_into_table_
users.sql
```

Now we see the first migration in applied status and the second in pending status.

Let's apply the second migration using `goose up-by-one`:

```
$ goose ydb $YDB_CONNECTION_STRING up-by-one
2024/01/12 12:04:56 OK   20240112120057_00002_add_column_password_hash_into_table_users.sql (59.93ms)
```

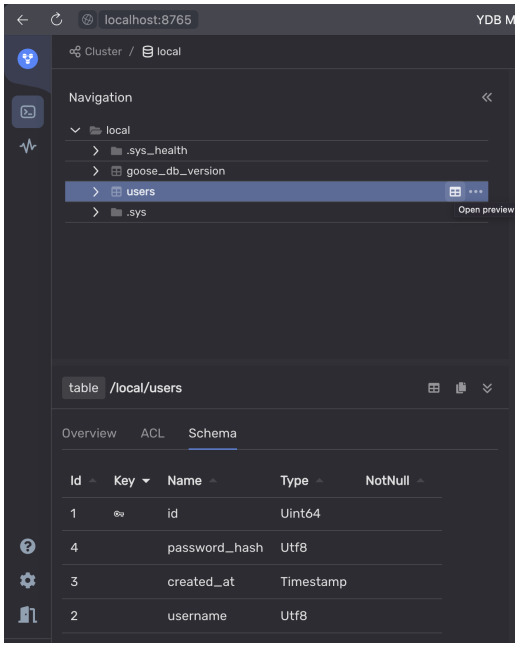
Let's check the migration status through `goose status`:

```
$ goose ydb $YDB_CONNECTION_STRING status
2024/01/12 12:05:17     Applied At                  Migration
2024/01/12 12:05:17     =======================================
2024/01/12 12:05:17     Fri Jan 12 11:55:18 2024 -- 20240112115229_00001_create_first_table.sql
2024/01/12 12:05:17     Fri Jan 12 12:04:56 2024 -- 20240112120057_00002_add_column_password_hash_into_table_
users.sql
```

Both migration are fully applied.

Let's use the same methods to see the new changes:

Using YDB UI on `http://localhost:8765`



Using YDB CLI

```
$ ydb -e grpc://localhost:2136 -d /local scheme describe users
<table> users

Columns:
┌───────────────┬──────────────┬────────┬─────┐
| Name          | Type         | Family | Key |
├───────────────┼──────────────┼────────┼─────┤
| id            | Uint64?      |        | K0  |
| username      | Utf8?        |        |     |
| created_at    | Timestamp?   |        |     |
| password_hash | Utf8?        |        |     |
└───────────────┴──────────────┴────────┴─────┘

Storage settings:
Store large values in "external blobs": false

Column families:
┌─────────┬──────┬─────────────┬──────────────────┐
| Name    | Data | Compression | Keep in memory |
├─────────┼──────┼─────────────┼──────────────────┤
| default |      | None        |                  |
└─────────┴──────┴─────────────┴──────────────────┘

Auto partitioning settings:
Partitioning by size: true
Partitioning by load: false
Preferred partition size (Mb): 2048
Min partitions count: 1
```

All subsequent migration files should be created in the same way.

Downgrading migrations

Let's downgrade (revert) the latest migration using `goose down`:

```
$ goose ydb $YDB_CONNECTION_STRING down
2024/01/12 13:07:18 OK   20240112120057_00002_add_column_password_hash_into_table_users.sql (43ms)
```

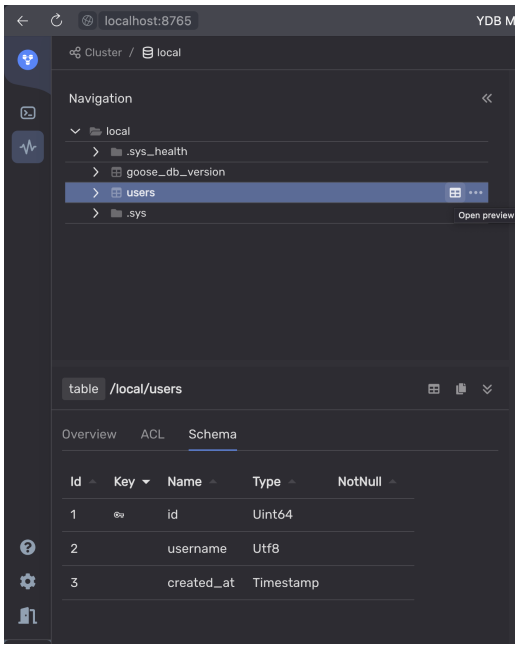
Let's check the migration statuses through `goose status`:

```
$ goose ydb $YDB_CONNECTION_STRING status
2024/01/12 13:07:36     Applied At                  Migration
2024/01/12 13:07:36     =======================================
2024/01/12 13:07:36     Fri Jan 12 11:55:18 2024 -- 20240112115229_00001_create_first_table.sql
2024/01/12 13:07:36     Pending                  -- 20240112120057_00002_add_column_password_hash_into_table_
users.sql
```

Status `Fri Jan 12 12:04:56 2024` changed to `Pending` - this means that the latest migration is no longer applied.

Let's check the changes again:

Using YDB UI on `http://localhost:8765`



Using YDB CLI

```
$ ydb -e grpc://localhost:2136 -d /local scheme describe users
<table> users

Columns:
┌────────────┬────────────┬────────┬─────┐
| Name       | Type       | Family | Key |
├────────────┼────────────┼────────┼─────┤
| id         | Uint64?    |        | K0  |
| username   | Utf8?      |        |     |
| created_at | Timestamp? |        |     |
└────────────┴────────────┴────────┴─────┘

Storage settings:
Store large values in "external blobs": false

Column families:
┌─────────┬──────┬─────────────┬──────────────────┐
| Name    | Data | Compression | Keep in memory |
├─────────┼──────┼─────────────┼──────────────────┤
| default |      | None        |                  |
└─────────┴──────┴─────────────┴──────────────────┘

Auto partitioning settings:
Partitioning by size: true
Partitioning by load: false
Preferred partition size (Mb): 2048
Min partitions count: 1
```

Short list of "goose" commands

The `goose` utility allows you to manage migrations via the command line:

- `goose status` - view the status of applying migrations. For example, `goose ydb $YDB_CONNECTION_STRING status`.
- `goose up` - apply all known migrations. For example, `goose ydb $YDB_CONNECTION_STRING up`.
- `goose up-by-one` - apply exactly one “next” migration. For example, `goose ydb $YDB_CONNECTION_STRING up-by-one`.
- `goose redo` - re-apply the latest migration. For example, `goose ydb $YDB_CONNECTION_STRING redo`.
- `goose down` - rollback the last migration. For example, `goose ydb $YDB_CONNECTION_STRING down`.

- `goose reset` - rollback all migrations. For example, `goose ydb $YDB_CONNECTION_STRING reset`.

> **Warning**
>
> Be careful: the `goose reset` command will revert all your migrations using your statements in blocks `+goose Down`. In many cases it might lead to all data in the database being erased. Make sure you regularly do backups and check that they can be restored to minimize impact of this risk.

Migrating YDB data schemas with the Liquibase migration tool

Introduction

Liquibase is an open-source library for tracking, managing, and applying changes to database schemas. It is extended with dialects for different database management systems (DBMS), including YDB.

Dialect is the main component in the Liquibase framework, which assists in creating SQL queries for a database, considering the specific features of a given DBMS.

Features of the YDB Dialect

Liquibase's main functionality is the abstract description of database schemas in a `.xml`, `.json`, or `.yaml` format. This ensures portability when switching between different DBMSs.

The YDB dialect supports the following basic constructs in the standard migration description (changeset).

Creating a table

The `createTable` changeset is responsible for creating a table. The descriptions of types from the SQL standard are mapped to primitive types in YDB. For example, the `bigint` type will be converted to `Int64`.

> **Note**
>
> You can also explicitly specify the original type name, such as `Int32`, `Json`, `JsonDocument`, `Bytes`, or `Interval`. However, in this case, the schema won't be portable.

Table of comparison of Liquibase types descriptions with YDB types:

Liquibase types	YDB type
`boolean`, `java.sql.Types.BOOLEAN`, `java.lang.Boolean`, `bit`, `bool`	`Bool`
`blob`, `longblob`, `longvarbinary`, `String`, `java.sql.Types.BLOB`, `java.sql.Types.LONGBLOB`, `java.sql.Types.LONGVARBINARY`, `java.sql.Types.VARBINARY`, `java.sql.Types.BINARY`, `varbinary`, `binary`, `image`, `tinyblob`, `mediumblob`, `long binary`, `long varbinary`	`Bytes` (synonym `String`)
`java.sql.Types.DATE`, `smalldatetime`, `date`	`Date`
`decimal`, `java.sql.Types.DECIMAL`, `java.math.BigDecimal`	`Decimal(22,9)`
`double`, `java.sql.Types.DOUBLE`, `java.lang.Double`	`Double`
`float`, `java.sql.Types.FLOAT`, `java.lang.Float`, `real`, `java.sql.Types.REAL`	`Float`
`int`, `integer`, `java.sql.Types.INTEGER`, `java.lang.Integer`, `int4`, `int32`	`Int32`
`bigint`, `java.sql.Types.BIGINT`, `java.math.BigInteger`, `java.lang.Long`, `integer8`, `bigserial`, `long`	`Int64`
`java.sql.Types.SMALLINT`, `int2`, `smallserial`, `smallint`	`Int16`
`java.sql.Types.TINYINT`, `tinyint`	`Int8`
`char`, `java.sql.Types.CHAR`, `bpchar`, `character`, `nchar`, `java.sql.Types.NCHAR`, `nchar2`, `text`, `varchar`, `java.sql.Types.VARCHAR`, `java.lang.String`, `varchar2`, `character varying`, `nvarchar`, `java.sql.Types.NVARCHAR`, `nvarchar2`, `national`, `clob`, `longvarchar`, `longtext`, `java.sql.Types.LONGVARCHAR`, `java.sql.Types.CLOB`, `nclob`, `longnvarchar`, `ntext`, `java.sql.Types.LONGNVARCHAR`, `java.sql.Types.NCLOB`, `tinytext`, `mediumtext`, `long varchar`, `long nvarchar`	`Text` (synonym `Utf8`)
`timestamp`, `java.sql.Types.TIMESTAMP`, `java.sql.Timestamp`	`Timestamp`
`datetime`, `time`, `java.sql.Types.TIME`, `java.sql.Time`	`Datetime`

> **Warning**
>
> In YDB, the `Timestamp` data type stores dates with microsecond precision. The `java.sql.Timestamp` or `java.time.Instant` store timestamps with nanosecond precision, so you should be aware of this when using these data types.
>
> The type names are case insensitive.

The `dropTable` changeset - delete a table. For example: `<dropTable tableName="episodes"/>`

Changing the table structure

`addColumn` - add a column to a table. For example:

xml

```
<addColumn tableName="seasons">
    <column name="is_deleted" type="bool"/>
</addColumn>
```

json

```
"changes": [
    {
      "addColumn": {
        "tableName": "seasons",
        "columns": [
          {
            "column": {
              "name": "is_deleted",
              "type": "bool"
            }
          }
        ]
      }
    }
  ]
```

yaml

```
changes:
- addColumn:
    tableName: seasons
    columns:
    - column:
        name: is_deleted
        type: bool
```

`createIndex` - create a secondary index. For example:

xml

```
<createIndex tableName="episodes" indexName="episodes_index">
    <column name="title"/>
</createIndex>
```

json

```
"changes": [
    {
      "createIndex": {
        "tableName": "episodes",
        "indexName": "episodes_index",
        "columns": {
          "column": {
            "name": "title"
          }
        }
      }
    }
```

yaml

```
changes:
- createIndex:
    tableName: episodes
    indexName: episodes_index
    columns:
    - column:
        name: title
```

Warning

YDB doesn't support unique secondary indexes.

Note

Asynchronous indexes should be created using native SQL migrations.

`dropIndex` - drop a secondary index. For example:

xml

```
<dropIndex tableName="series" indexName="series_index"/>
```

json

```
"changes": [
  {
    "dropIndex": {
```

```
      "tableName": "series",
      "indexName": "series_index"
    }
  }
]
```

yaml

```
changes:
- dropIndex:
    tableName: series
    indexName: series_index
```

Ingesting data into a table

`loadData`, `loadUpdateData` - upload data from a `CSV` file into a table. `loadUpdateData` loads data using the UPSERT INTO command.

`insert` is a changeset that performs a single insert into a table using the INSERT INTO command. For example:

xml

```
<insert tableName="episodes">
    <column name="series_id" valueNumeric="1"/>
    <column name="season_id" valueNumeric="1"/>
    <column name="episode_id" valueNumeric="1"/>
    <column name="title" value="Yesterday's Jam"/>
    <column name="air_date" valueDate="2023-04-03T08:46:23.456"/>
</insert>
```

json

```
"changes": [
  {
    "insert": {
      "tableName": "episodes",
      "columns": [
        {
          "column": {
            "name": "series_id",
            "valueNumeric": "1"
          }
        },
        {
          "column": {
            "name": "season_id",
            "valueNumeric": "1"
          }
        },
        {
          "column": {
            "name": "episode_id",
            "valueNumeric": "1"
          }
        },
        {
          "column": {
            "name": "title",
            "value": "Yesterday's Jam"
          }
        },
        {
          "column": {
            "name": "air_date",
            "valueDate": "2023-04-03T08:46:23.456"
          }
        }
      ]
    }
  }
]
```

yaml

```
changes:
- insert:
    tableName: episodes
    columns:
      - column:
          name: series_id
          valueNumeric: 1
      - column:
          name: season_id
          valueNumeric: 1
      - column:
          name: episode_id
          valueNumeric: 1
      - column:
```

```
            name: title
            value: Yesterday's Jam
        - column:
            name: air_date
            valueDate: 2023-04-03T08:46:23.456
```

You can also specify any value in the `value` field. Data from the `value` field in `insert` changeset or `CSV` files will be automatically converted to the required types, taking into account strict typing in YDB.

The type formatting table to load into the table:

YDB type	Description format
`Bool`	`true` or `false`
`Int8`, `Int16`, `Int32`, `Int64`	A signed integer
`Uint8`, `Uint16`, `Uint32`, `Uint64`	An unsigned integer
`Text`, `Bytes`, `Json`, `JsonDocument`	Represent as text
`Float`, `Double`, `Decimal(22, 9)`	A real number
`Interval`	ISO-8601, corresponds to the `java.time.Duration` in Java.
`Date`	Pattern `YYYY-MM-DD` calendar date from standard ISO-8601
`Datetime`	Pattern `YYYY-MM-DDThh:mm:ss`, timezone will be set to `UTC`
`Timestamp`	The timestamp from the ISO-8601 standard corresponds to the `java.time.Instant` in Java, timezone will be set to `UTC` (precision in microseconds - `Timestamp` type YDB restriction)

Example `CSV` file:

```
id,bool,bigint,smallint,tinyint,float,double,decimal,uint8,uint16,uint32,uint64,text,binary,json,jsondocumen
t,date,datetime,timestamp,interval
2,true,123123,13000,112,1.123,1.123123,1.123123,12,13,14,15,kurdyukov-kir,binary,{"asd": "asd"},{"asd": "as
d"},2014-04-06,2023-09-16T12:30,2023-07-31T17:00:00.00Z,PT10S
```

Warning

To understand which SQL statements YDB can perform and what are the restrictions on data types, read the documentation for the query language YQL.

Note

It is important to note that custom YQL instructions can be applied via native SQL queries.

How to use it?

There are two ways:

Programmatically from Java / Kotlin applications

The project's README describes how to use it from Java or Kotlin in detail. There is also an example of a Spring Boot application using it.

Liquibase CLI

First, you need to install Liquibase itself using one of the recommended methods. Then you need to place the `.jar` archives of YDB JDBC driver and Liquibase YDB dialect into the `internal/lib` folder.

```
# $(which liquibase)
cd ./internal/lib/

# you may need to sudo
# set actual versions of .jar files
curl -L -o ydb-jdbc-driver.jar https://repo.maven.apache.org/maven2/tech/ydb/jdbc/ydb-jdbc-driver-shaded/2.0.
7/ydb-jdbc-driver-shaded-2.0.7.jar
curl -L -o liquibase-ydb-dialect.jar https://repo.maven.apache.org/maven2/tech/ydb/dialects/liquibase-ydb-dia
lect/1.0.0/liquibase-ydb-dialect-1.0.0.jar
```

For a more detailed description, see the Manual library management in Liquibase documentation.

Now the `liquibase` command line utility can be used with YDB.

Liquibase usage scenarios

Initializing Liquibase on an empty YDB cluster

The main command is `liquibase update`, which applies migrations if the current schema in YDB lags behind the user-defined description.

Let's apply this changeset to an empty database:

```
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
        xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
                      http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.8.xsd">

    <changeSet id="episodes" author="kurdyukov-kir">
        <comment>Table episodes.</comment>

        <createTable tableName="episodes">
            <column name="series_id" type="bigint">
                <constraints primaryKey="true"/>
            </column>
            <column name="season_id" type="bigint">
                <constraints primaryKey="true"/>
            </column>
            <column name="episode_id" type="bigint">
                <constraints primaryKey="true"/>
            </column>

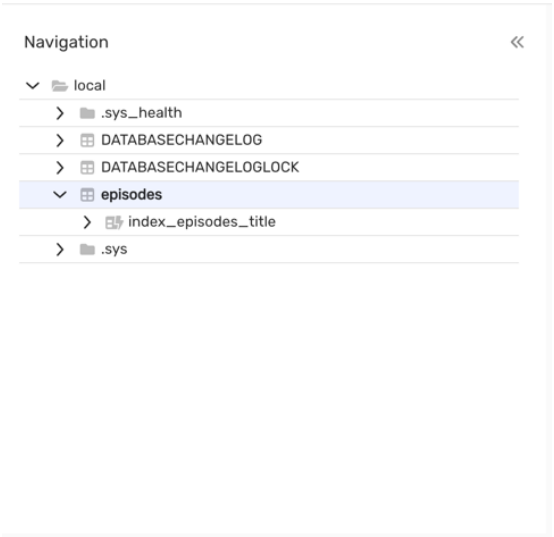
            <column name="title" type="text"/>
            <column name="air_date" type="timestamp"/>
        </createTable>
        <rollback>
            <dropTable tableName="episodes"/>
        </rollback>
    </changeSet>
    <changeSet id="index_episodes_title" author="kurdyukov-kir">
        <createIndex tableName="episodes" indexName="index_episodes_title">
            <column name="title"/>
        </createIndex>
    </changeSet>
</databaseChangeLog>
```

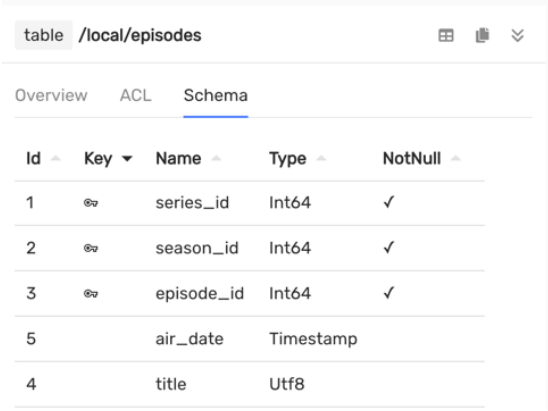
After executing the `liquibase update` command, Liquibase will print the following log:

```
UPDATE SUMMARY
Run:                          2
Previously run:               0
Filtered out:                 0
-------------------------------
Total change sets:            2

Liquibase: Update has been successful. Rows affected: 2
Liquibase command 'update' was executed successfully.
```

After applying migrations, the data schema now looks like this:





You can see that Liquibase has created two service tables: `DATABASECHANGELOG`, which is the migration log, and `DATABASECHANGELOGLOCK`, which is a table for acquiring a distributed lock.

Example contents of the `DATABASECHANGELOG` table:

AUTHOR	COMMENTS	CONTEXTS	DATEEXECUTED	DEPLOYMENT_ID	DESCRIPTION	EXECTYPE	FILENAME
kurdyukov-kir	Table episodes.		12:53:27	1544007500	createTable tableName=episodes	EXECUTED	migration/episodes.xml
kurdyukov-kir	""		12:53:28	1544007500	createIndex indexName=index_episodes_title, tableName=episodes	EXECUTED	migration/episodes.xml

Database schema evolution

Let's say we need to create a YDB topic and turn off the param `AUTO_PARTITIONING_BY_SIZE` of the table. This can be done with a native SQL script:

```
--liquibase formatted sql

--changeset kurdyukov-kir:create-a-topic
CREATE TOPIC `my_topic` (
    CONSUMER my_consumer
) WITH (
     retention_period = Interval('P1D')
);

--changeset kurdyukov-kir:auto-partitioning-disabled
ALTER TABLE episodes SET (AUTO_PARTITIONING_BY_SIZE = DISABLED);
```

Also, let's add a new column `is_deleted` and remove the `index_episodes_title` index:

```
<changeSet id="alter-episodes" author="kurdyukov-kir">
    <comment>Alter table episodes.</comment>

    <dropIndex tableName="episodes" indexName="index_episodes_title"/>

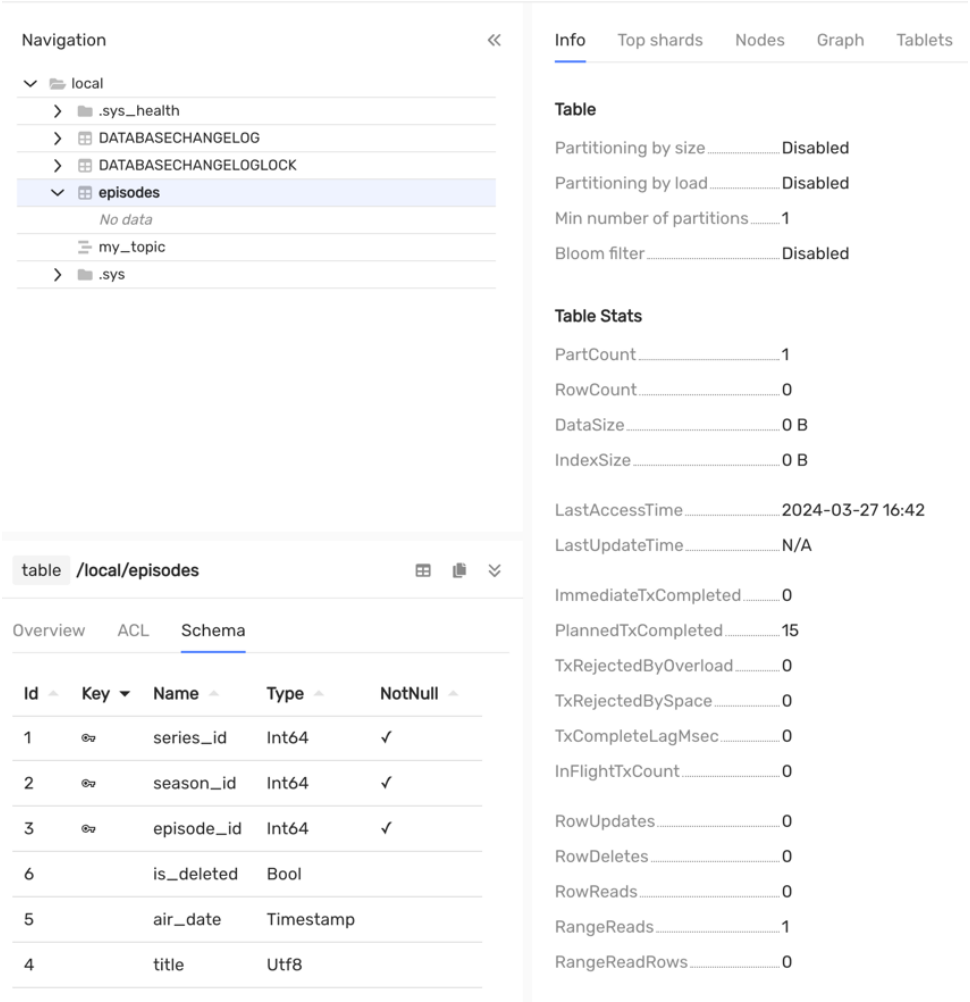
    <addColumn tableName="episodes">
        <column name="is_deleted" type="bool"/>
    </addColumn>
</changeSet>
<include file="/migration/sql/yql.sql" relativeToChangelogFile="true"/>
```

After executing `liquibase update`, the database schema will be successfully updated with all of these changes:

```
UPDATE SUMMARY
Run:                          3
Previously run:               2
Filtered out:                 0
-------------------------------
Total change sets:            5

Liquibase: Update has been successful. Rows affected: 3
Liquibase command 'update' was executed successfully.
```

The result will be deleting the index, adding the `is_deleted` column, disabling the auto partitioning setting, and creating a topic:



Initializing liquibase in a project with a non-empty data schema

Let's suppose there's an existing project with the following database schema:

table /local/all_types_table

Overview ACL Schema

Id	Key	Name	Type
9		uint8_column	Uint8
10		uint16_column	Uint16
11		unit32_column	Uint32
12		unit64_column	Uint64
13		text_column	Utf8
14		binary_column	String
15		json_column	Json
16		jsondocument_column	JsonDocument
17		date_column	Date
18		datetime_column	Datetime
19		timestamp_column	Timestamp
20		interval_column	Interval

In this case to start using Liquibase, you need to run:

```
liquibase generate-changelog --changelog-file=changelog.xml
```

The contents of the generated changelog.xml:

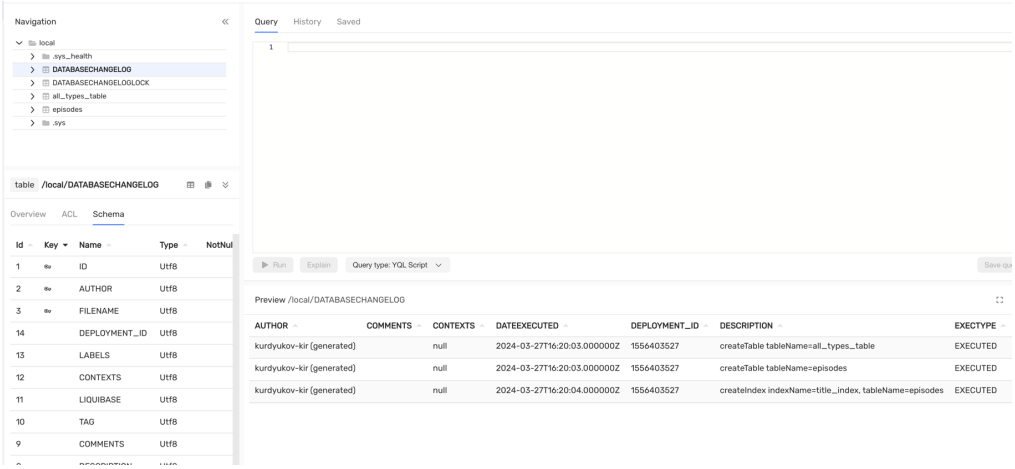
```
<changeSet author="kurdyukov-kir (generated)" id="1711556283305-1">
    <createTable tableName="all_types_table">
        <column name="id" type="INT32">
            <constraints nullable="false" primaryKey="true"/>
        </column>
        <column name="bool_column" type="BOOL"/>
        <column name="bigint_column" type="INT64"/>
        <column name="smallint_column" type="INT16"/>
        <column name="tinyint_column" type="INT8"/>
        <column name="float_column" type="FLOAT"/>
        <column name="double_column" type="DOUBLE"/>
        <column name="decimal_column" type="DECIMAL(22, 9)"/>
        <column name="uint8_column" type="UINT8"/>
        <column name="uint16_column" type="UINT16"/>
        <column name="unit32_column" type="UINT32"/>
        <column name="unit64_column" type="UINT64"/>
        <column name="text_column" type="TEXT"/>
        <column name="binary_column" type="BYTES"/>
        <column name="json_column" type="JSON"/>
        <column name="jsondocument_column" type="JSONDOCUMENT"/>
        <column name="date_column" type="DATE"/>
        <column name="datetime_column" type="DATETIME"/>
        <column name="timestamp_column" type="TIMESTAMP"/>
        <column name="interval_column" type="INTERVAL"/>
    </createTable>
</changeSet>
<changeSet author="kurdyukov-kir (generated)" id="1711556283305-2">
    <createTable tableName="episodes">
        <column name="series_id" type="INT64">
            <constraints nullable="false" primaryKey="true"/>
        </column>
        <column name="season_id" type="INT64">
            <constraints nullable="false" primaryKey="true"/>
        </column>
        <column name="episode_id" type="INT64">
            <constraints nullable="false" primaryKey="true"/>
        </column>
        <column name="title" type="TEXT"/>
        <column name="air_date" type="DATE"/>
```

```
    </createTable>
</changeSet>
<changeSet author="kurdyukov-kir (generated)" id="1711556283305-3">
    <createIndex indexName="title_index" tableName="episodes">
        <column name="title"/>
    </createIndex>
</changeSet>
```

Then you need to synchronize the generated changelog.xml the file, this is done by the command:

```
liquibase changelog-sync --changelog-file=changelog.xml
```

The result will be liquibase synchronization in your project:



Connecting to YDB

In the examples above, a Docker container was used, which didn't require any additional authentication settings.

List of different authentication options through URL parameters:

- Local or remote Docker (anonymous authentication):
  `jdbc:ydb:grpc://localhost:2136/local`
- Self-hosted cluster:
  `jdbc:ydb:grpcs://<host>:2135/Root/testdb?secureConnectionCertificate=file:~/myca.cer`
- Connect with token to the cloud instance:
  `jdbc:ydb:grpcs://<host>:2135/path/to/database?token=file:~/my_token`
- Connect with service account to the cloud instance:
  `jdbc:ydb:grpcs://<host>:2135/path/to/database?saFile=file:~/sa_key.json`

Also, if your cluster is configured using username and password, authentication is done through Liquibase parameters.

For more info about different authentication settings, refer to the section.

Query engines

Engine	Guide
Apache Spark™	Guide

Apache Spark™

Apache Spark™ is an open-source, distributed processing system used for big data workloads. It utilizes in-memory caching and optimized query execution for fast analytic queries against data of any size. It provides development APIs in Java, Scala, Python, and R, and supports code reuse across multiple workloads—batch processing, interactive queries, real-time analytics, machine learning, and graph processing. Apache Spark™ can work with YDB using the YDB Spark Connector, a special module that implements core Apache Spark™ primitives. It supports:

- Distribution of operations across YDB table partitions
- Scalable YDB table readings and writing
- Automatic creation of tables if they do not exist

> **Note**
>
> The connector may require additional memory on the Apache Spark™ executor side to work with better speed and performance. 4 GB or more memory per executor is highly recommended.

How to Use

To work with YDB in Apache Spark™, you need to add the YDB Spark Connector to your Apache Spark™ driver. This can be done in several ways:

- Download the connector dependency directly from Maven Central using the `--packages` option. It's recommended to use the latest published version:

  Spark Shell

  ```
  ~ $ spark-shell --master <master-url> --packages tech.ydb.spark:ydb-spark-connector-shaded:2.0.1 --conf
  spark.executor.memory=4g
  ```

  PySpark

  ```
  ~ $ pyspark --master <master-url> --packages tech.ydb.spark:ydb-spark-connector-shaded:2.0.1 --conf spar
  k.executor.memory=4g
  ```

  Spark SQL

  ```
  ~ $ spark-sql --master <master-url> --packages tech.ydb.spark:ydb-spark-connector-shaded:2.0.1 --conf sp
  ark.executor.memory=4g
  ```

- Download the latest version of the shaded connector (a connector build that includes all dependencies) from GitHub or Maven Central and specify the downloaded artifact in the `--jars` option:

  Spark Shell

  ```
  ~ $ spark-shell --master <master-url> --jars ~/Download/ydb-spark-connector-shaded-2.0.1.jar --conf spar
  k.executor.memory=4g
  ```

  PySpark

  ```
  ~ $ pyspark --master <master-url> --jars ~/Download/ydb-spark-connector-shaded-2.0.1.jar  --conf spark.e
  xecutor.memory=4g
  ```

  Spark SQL

  ```
  ~ $ spark-sql --master <master-url> --jars ~/Download/ydb-spark-connector-shaded-2.0.1.jar  --conf spar
  k.executor.memory=4g
  ```

- You can also copy the downloaded shaded artifact to the `jars` folder of your Apache Spark™ distribution. In this case, no additional options need to be specified.

Use DataFrame API

The DataFrame API allows you to work with YDB in an interactive `spark-shell` or `pyspark` session, as well as when writing code in `Java`, `Scala`, or `Python` for `spark-submit`.

To create a `DataFrame`, you need to specify the `ydb` format, pass a set of connection options and the path to the YDB table:

Scala

```
val ydb_df = spark.read.format("ydb").options(<options>).load("<table-path>")
```

Python

```
ydb_df = spark.read.format("ydb").options(<options>).load("<table-path>")
```

To save any `DataFrame` in the table YDB, you similarly need to specify the `ydb` format, connection options and the path to the table:

Scala

```
any_dataframe.write.format("ydb").options(<options>).mode("append").save("<table-path>")
```

Python

```
any_dataframe.write.format("ydb").options(<options>).mode("append").save("<table-path>")
```

> **Note**
>
> For writing data to YDB it is recommended to use the `append` mode, which uses batch data loading. If the table specified in the `save()` method does not exist, it will be created automatically according to the table autocreation options.

A more detailed example is provided in the Spark-shell example.

Use Catalog API

Catalogs allow you to work with YDB in interactive `spark-sql` sessions or execute SQL queries via the `spark.sql` method.
To access YDB, you need to add a catalog by specifying the following Apache Spark™ properties. You can define multiple catalogs with different names to access to different YDB databases:

```
# Mandatory catalog's driver name
spark.sql.catalog.<catalog_name>=tech.ydb.spark.connector.YdbCatalog
# Mandatory option, the url of database
spark.sql.catalog.<catalog_name>.url=<ydb-connection-url>
# Other options are not mandatory and may be specified as necessary
spark.sql.catalog.<catalog_name>.<param-name>=<param-value>
```

After that, you can work with YDB tables through standard Apache Spark™ SQL queries.
Note that you should use a dot `.` as a separator in the table path.

```
SELECT * FROM <catalog_name>.<table-path> LIMIT 10;
```

A more detailed example is provided in the Spark SQL example.

YDB Spark Connector Options

The behavior of the YDB Spark Connector is configured using options that can be passed as one set with the `options` method or specified individually with the `option` method. Each `DataFrame` and each individual operation on a `DataFrame` can have its own configuration of options.

Connection Options

- `url` — a required parameter with the YDB connection string in the following format:
  `grpc[s]://<endpoint>:<port>/<database>[?<options>]`
  Examples:
  - Local Docker container with anonymous authentication and without TLS:
    `grpc://localhost:2136/local`
  - Remote self-hosted cluster:
    `grpcs://my-private-cluster:2135/Root/my-database?secureConnectionCertificate=~/myCertificate.cer`
  - Cloud database instance with a token:
    `grpcs://ydb.my-cloud.com:2135/my_folder/test_database?tokenFile=~/my_token`
  - Cloud database instance with a service account key:
    `grpcs://ydb.my-cloud.com:2135/my_folder/test_database?saKeyFile=~/sa_key.json`
- `ca.text` — specifies the certificate value for establishing a TLS connection.
- `ca.file` — specifies the path to the certificate for establishing a TLS connection. You can specify it directly in `url` as the `secureConnectionCertificate` option.
- `auth.use_env` — if set to `true`, authentication based on environment variables will be used.
- `auth.use_metadata` — if set to `true`, metadata-based authentication mode will be used. You can specify it directly in `url` as the `useMetadata` option.
- `auth.login` and `auth.password` — login and password for static authentication.
- `auth.token` — authentication using the specified Access Token.
- `auth.token.file` — authentication using Access Token from the specified file. You can specify it directly in `url` as the `tokenFile` option.
- `auth.sakey.text` — used to specify the key content for authentication with a service account key.
- `auth.sakey.file` — used to specify the path to the key file for authentication with a service account key. You can specify it directly in `url` as the `saKeyFile` option.

Table Autocreation Options

> **Tip**
>
> If you need the table to have some custom settings configured, create it manually beforehand with CREATE TABLE or modify it afterward with ALTER TABLE.

- `table.autocreate` — if set to `true`, then when writing to a non-existent table, it will be created automatically. Enabled by default;

- `table.type` — the type of automatically created table. Possible values:
  - `row` for creating a row-oriented table (default).
  - `column` for creating a column-oriented table.
- `table.primary_keys` — a comma-separated list of columns to use as the primary key. If this option is not provided, a new column with random content will be used for the key.
- `table.auto_pk_name` — the name of the column for the randomly created key. This column will be created with the type `Utf8` and will be filled with random UUID v4 values. Default value is `_spark_key`.

Spark Shell and PySpark Example

As an example, we'll show how to load a list of all Stack Overflow posts from 2020 into YDB. This data can be downloaded from the following link: https://datasets-documentation.s3.eu-west-3.amazonaws.com/stackoverflow/parquet/posts/2020.parquet

Spark Shell

```
~ $ spark-shell --master <master-url> --packages tech.ydb.spark:ydb-spark-connector-shaded:2.0.1 --conf spar
k.executor.memory=4g
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.5.4
      /_/

Using Scala version 2.12.18 (OpenJDK 64-Bit Server VM, Java 17.0.15)
Type in expressions to have them evaluated.
Type :help for more information.
```

PySpark

```
~ $ pyspark --master <master-url> --packages tech.ydb.spark:ydb-spark-connector-shaded:2.0.1 --conf spark.exe
cutor.memory=4g
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.5.4
      /_/

Using Python version 3.10.12 (main, May 27 2025 17:12:29)
SparkSession available as 'spark'.
```

Let's display the schema of the Parquet file and count the number of rows it contains:

Spark Shell

```
scala> val so_posts2020 = spark.read.format("parquet").load("/home/username/2020.parquet")
so_posts2020: org.apache.spark.sql.DataFrame = [Id: bigint, PostTypeId: bigint ... 20 more fields]

scala> so_posts2020.printSchema
root
|-- Id: long (nullable = true)
|-- PostTypeId: long (nullable = true)
|-- AcceptedAnswerId: long (nullable = true)
|-- CreationDate: timestamp (nullable = true)
|-- Score: long (nullable = true)
|-- ViewCount: long (nullable = true)
|-- Body: binary (nullable = true)
|-- OwnerUserId: long (nullable = true)
|-- OwnerDisplayName: binary (nullable = true)
|-- LastEditorUserId: long (nullable = true)
|-- LastEditorDisplayName: binary (nullable = true)
|-- LastEditDate: timestamp (nullable = true)
|-- LastActivityDate: timestamp (nullable = true)
|-- Title: binary (nullable = true)
|-- Tags: binary (nullable = true)
|-- AnswerCount: long (nullable = true)
|-- CommentCount: long (nullable = true)
|-- FavoriteCount: long (nullable = true)
|-- ContentLicense: binary (nullable = true)
|-- ParentId: binary (nullable = true)
|-- CommunityOwnedDate: timestamp (nullable = true)
|-- ClosedDate: timestamp (nullable = true)

scala> so_posts2020.count
res1: Long = 4304021
```

PySpark

```
>>> so_posts2020 = spark.read.format("parquet").load("/home/username/2020.parquet")
>>> so_posts2020.printSchema()
root
|-- Id: long (nullable = true)
|-- PostTypeId: long (nullable = true)
|-- AcceptedAnswerId: long (nullable = true)
```

```
|-- CreationDate: timestamp (nullable = true)
|-- Score: long (nullable = true)
|-- ViewCount: long (nullable = true)
|-- Body: binary (nullable = true)
|-- OwnerUserId: long (nullable = true)
|-- OwnerDisplayName: binary (nullable = true)
|-- LastEditorUserId: long (nullable = true)
|-- LastEditorDisplayName: binary (nullable = true)
|-- LastEditDate: timestamp (nullable = true)
|-- LastActivityDate: timestamp (nullable = true)
|-- Title: binary (nullable = true)
|-- Tags: binary (nullable = true)
|-- AnswerCount: long (nullable = true)
|-- CommentCount: long (nullable = true)
|-- FavoriteCount: long (nullable = true)
|-- ContentLicense: binary (nullable = true)
|-- ParentId: binary (nullable = true)
|-- CommunityOwnedDate: timestamp (nullable = true)
|-- ClosedDate: timestamp (nullable = true)

>>> so_posts2020.count()
4304021
```

Then add a new column with the year to this DataFrame and store it all to a column-oriented YDB table:

Spark Shell

```
scala> val my_ydb = Map("url" -> "grpcs://ydb.my-host.net:2135/preprod/spark-test?tokenFile=~/.token")
my_ydb: scala.collection.immutable.Map[String,String] = Map(url -> grpcs://ydb.my-host.net:2135/preprod/spark
-test?tokenFile=~/.token)

scala> so_posts2020.withColumn("Year", lit(2020)).write.format("ydb").options(my_ydb).option("table.type", "c
olumn").option("table.primary_keys", "Id").mode("append").save("stackoverflow/posts");
```

PySpark

```
>>> from pyspark.sql.functions import col,lit
>>> my_ydb = {"url": "grpcs://ydb.my-host.net:2135/preprod/spark-test?tokenFile=~/.token"}
>>> so_posts2020.withColumn("Year", lit(2020)).write.format("ydb").options(**my_ydb).option("table.type", "co
lumn").option("table.primary_keys", "Id").mode("append").save("stackoverflow/posts")
```

As a result, you can read the stored data from the YDB table and, for example, count the number of posts that have an accepted answer:

Spark Shell

```
scala> val ydb_posts2020 = spark.read.format("ydb").options(my_ydb).load("stackoverflow/posts")
ydb_posts2020: org.apache.spark.sql.DataFrame = [Id: bigint, PostTypeId: bigint ... 21 more fields]

scala> ydb_posts2020.printSchema
root
|-- Id: long (nullable = false)
|-- PostTypeId: long (nullable = true)
|-- AcceptedAnswerId: long (nullable = true)
|-- CreationDate: timestamp (nullable = true)
|-- Score: long (nullable = true)
|-- ViewCount: long (nullable = true)
|-- Body: binary (nullable = true)
|-- OwnerUserId: long (nullable = true)
|-- OwnerDisplayName: binary (nullable = true)
|-- LastEditorUserId: long (nullable = true)
|-- LastEditorDisplayName: binary (nullable = true)
|-- LastEditDate: timestamp (nullable = true)
|-- LastActivityDate: timestamp (nullable = true)
|-- Title: binary (nullable = true)
|-- Tags: binary (nullable = true)
|-- AnswerCount: long (nullable = true)
|-- CommentCount: long (nullable = true)
|-- FavoriteCount: long (nullable = true)
|-- ContentLicense: binary (nullable = true)
|-- ParentId: binary (nullable = true)
|-- CommunityOwnedDate: timestamp (nullable = true)
|-- ClosedDate: timestamp (nullable = true)
|-- Year: integer (nullable = true)

scala> ydb_posts2020.count
res3: Long = 4304021

scala> ydb_posts2020.filter(col("AcceptedAnswerId") > 0).count
res4: Long = 843780
```

PySpark

```
>>> ydb_posts2020 = spark.read.format("ydb").options(**my_ydb).load("stackoverflow/posts")
>>> ydb_posts2020.printSchema()
root
|-- Id: long (nullable = true)
```

```
|-- PostTypeId: long (nullable = true)
|-- AcceptedAnswerId: long (nullable = true)
|-- CreationDate: timestamp (nullable = true)
|-- Score: long (nullable = true)
|-- ViewCount: long (nullable = true)
|-- Body: binary (nullable = true)
|-- OwnerUserId: long (nullable = true)
|-- OwnerDisplayName: binary (nullable = true)
|-- LastEditorUserId: long (nullable = true)
|-- LastEditorDisplayName: binary (nullable = true)
|-- LastEditDate: timestamp (nullable = true)
|-- LastActivityDate: timestamp (nullable = true)
|-- Title: binary (nullable = true)
|-- Tags: binary (nullable = true)
|-- AnswerCount: long (nullable = true)
|-- CommentCount: long (nullable = true)
|-- FavoriteCount: long (nullable = true)
|-- ContentLicense: binary (nullable = true)
|-- ParentId: binary (nullable = true)
|-- CommunityOwnedDate: timestamp (nullable = true)
|-- ClosedDate: timestamp (nullable = true)
|-- Year: integer (nullable = true)

>>> ydb_posts2020.count()
4304021
>>> ydb_posts2020.filter(col("AcceptedAnswerId") > 0).count()
843780
```

Spark SQL example

As an example, we'll show how to load a list of all Stack Overflow posts from 2020 into YDB. This data can be downloaded from the following link: https://datasets-documentation.s3.eu-west-3.amazonaws.com/stackoverflow/parquet/posts/2020.parquet

First, let's run `spark-sql` with the configured `my_ydb` catalog:

```
~ $ spark-sql --master <master-url> --packages tech.ydb.spark:ydb-spark-connector-shaded:2.0.1 \
     --conf spark.sql.catalog.my_ydb=tech.ydb.spark.connector.YdbCatalog \
     --conf spark.sql.catalog.my_ydb.url=grpcs://ydb.my-host.net:2135/preprod/spark-test \
     --conf spark.sql.catalog.my_ydb.auth.token.file=~/.token \
     --conf spark.executor.memory=4g
spark-sql (default)>
```

Let's validate the current state of the connected database and confirm the absence of the `stackoverflow/posts` table:

```
spark-sql (default)> SHOW NAMESPACES FROM my_ydb;
stackoverflow
Time taken: 0.11 seconds, Fetched 1 row(s)
spark-sql (default)> SHOW TABLES FROM my_ydb.stackoverflow;
Time taken: 0.041 seconds
```

Let's count the number of rows in the original parquet file:

```
spark-sql (default)> SELECT COUNT(*) FROM parquet.`/home/username/2020.parquet`;
4304021
```

Let's add a new column with the year and copy it all to a new YDB table:

```
spark-sql (default)> CREATE TABLE my_ydb.stackoverflow.posts OPTIONS(table.primary_keys='Id') AS SELECT *, 20
20 as Year FROM parquet.`/home/username/2020.parquet`;
Time taken: 85.225 seconds
```

Let's verify that the new table has appeared in the YDB database:

```
spark-sql (default)> SHOW TABLES FROM my_ydb.stackoverflow;
posts
Time taken: 0.07 seconds, Fetched 1 row(s)
spark-sql (default)> DESCRIBE TABLE my_ydb.stackoverflow.posts;
Id                    bigint
PostTypeId            bigint
AcceptedAnswerId      bigint
CreationDate          timestamp
Score                 bigint
ViewCount             bigint
Body                  binary
OwnerUserId           bigint
OwnerDisplayName      binary
LastEditorUserId      bigint
LastEditorDisplayName binary
LastEditDate          timestamp
LastActivityDate      timestamp
Title                 binary
Tags                  binary
AnswerCount           bigint
CommentCount          bigint
FavoriteCount         bigint
ContentLicense        binary
```

```
ParentId              binary
CommunityOwnedDate    timestamp
ClosedDate            timestamp
Year                  int
```

As a result, we can read the stored data from YDB table and, for example, count the number of posts that have an accepted answer:

```
spark-sql (default)> SELECT COUNT(*) FROM my_ydb.stackoverflow.posts;
4304021
Time taken: 19.726 seconds, Fetched 1 row(s)
spark-sql (default)> SELECT COUNT(*) FROM my_ydb.stackoverflow.posts WHERE AcceptedAnswerId > 0;
843780
Time taken: 6.599 seconds, Fetched 1 row(s)
```

Object–relational mapping

System	Instructions
Hibernate	Instructions
Spring Data JDBC	Instructions
Kotlin Exposed	Instructions
JOOQ	Instructions
Dapper	Instructions
Entity Framework	Instructions
Linq To DB	Instructions
SQLAlchemy	Instructions
Django	Instructions
Drizzle ORM	Instructions

YDB dialect for Hibernate

Overview

This is a guide to using Hibernate with YDB.

Hibernate is an Object-Relational Mapping (ORM) framework for Java that facilitates the mapping of object-oriented models to SQL.

Installation

Add the following dependency to your project:

Maven

```
<!-- Set actual versions -->
<dependency>
    <groupId>tech.ydb.jdbc</groupId>
    <artifactId>ydb-jdbc-driver</artifactId>
    <version>${ydb.jdbc.version}</version>
</dependency>

<dependency>
    <groupId>tech.ydb.dialects</groupId>
    <artifactId>hibernate-ydb-dialect</artifactId>
    <version>${hibernate.ydb.dialect.version}</version>
</dependency>
```

Gradle

```
dependencies {
    // Set actual versions
    implementation "tech.ydb.dialects:hibernate-ydb-dialect:$ydbDialectVersion"
    implementation "tech.ydb.jdbc:ydb-jdbc-driver:$ydbJdbcVersion"
}
```

If you use Hibernate version 5, you need `<artifactId>hibernate-ydb-dialect-v5</artifactId>` for Maven or `implementation 'tech.ydb.dialects:hibernate-ydb-dialect-v5:$version` for Gradle instead of the similar package without the `-v5` suffix.

Configuration

Configure Hibernate to use the custom YDB dialect by updating your persistence.xml file:

```
<property name="hibernate.dialect">tech.ydb.hibernate.dialect.YdbDialect</property>
```

Or, if you are using programmatic configuration:

Java

```
import org.hibernate.cfg.AvailableSettings;
import org.hibernate.cfg.Configuration;

public static Configuration basedConfiguration() {
    return new Configuration()
            .setProperty(AvailableSettings.JAKARTA_JDBC_DRIVER, YdbDriver.class.getName())
            .setProperty(AvailableSettings.DIALECT, YdbDialect.class.getName());
}
```

Kotlin

```
import org.hibernate.cfg.AvailableSettings
import org.hibernate.cfg.Configuration

fun basedConfiguration(): Configuration = Configuration().apply {
    setProperty(AvailableSettings.JAKARTA_JDBC_DRIVER, YdbDriver::class.name)
    setProperty(AvailableSettings.DIALECT, YdbDialect::class.name)
}
```

Usage

Use this custom dialect just like any other Hibernate dialect. Map your entity classes to database tables and use Hibernate's session factory to perform database operations.

Table of comparison of Java types descriptions with YDB types:

Java type	YDB type
`bool`, `Boolean`	`Bool`
`String`, enum with annotation `@Enumerated(EnumType.STRING)`	`Text` (synonym `Utf8`)
`java.time.LocalDate`	`Date`
`java.math.BigDecimal`, `java.math.BigInteger`	`Decimal(22,9)`
`double`, `Double`	`Double`

`float`, `Float`	`Float`
`int`, `java.lang.Integer`	`Int32`
`long`, `java.lang.Long`	`Int64`
`short`, `java.lang.Short`	`Int16`
`byte`, `java.lang.Byte`, enum with annotation `@Enumerated(EnumType.ORDINAL)`	`Int8`
`[]byte`	`Bytes` (synonym `String`)
`java.time.LocalDateTime` (timezone will be set to `UTC`)	`Datetime`
`java.time.Instant` (timezone will be set to `UTC`)	`Timestamp`

YDB dialect supports database schema generation based on Hibernate entities.

For example, for the `Group` class:

Java

```
@Getter
@Setter
@Entity
@Table(name = "Groups", indexes = @Index(name = "group_name_index", columnList = "GroupName"))
public class Group {

    @Id
    @Column(name = "GroupId")
    private int id;

    @Column(name = "GroupName")
    private String name;

    @OneToMany(mappedBy = "group")
    private List<Student> students;
}
```

Kotlin

```
@Entity
@Table(name = "Groups", indexes = [Index(name = "group_name_index", columnList = "GroupName")])
data class Group(
    @Id
    @Column(name = "GroupId")
    val id: Int,

    @Column(name = "GroupName")
    val name: String,

    @OneToMany(mappedBy = "group")
    val students: List<Student>
)
```

The following `Groups` table will be created, and the `GroupName` will be indexed by a global secondary index named `group_name_index`:

```
CREATE TABLE Groups (
    GroupId Int32 NOT NULL,
    GroupName Text,
    PRIMARY KEY (GroupId)
);

ALTER TABLE Groups
  ADD INDEX group_name_index GLOBAL
    ON (GroupName);
```

If you evolve the Group entity by adding the `deparment` field:

Java

```
@Column
private String department;
```

Kotlin

```
@Column
val department: String
```

At the start of the application, Hibernate will update the database schema if the `update` mode is set in properties:

```
jakarta.persistence.schema-generation.database.action=update
```

The result of changing the schema is:

```
ALTER TABLE Groups
   ADD COLUMN department Text
```

> **Warning**
>
> Hibernate is not designed to manage database schemas. You can manage your database schema using Liquibase or Flyway.

YDB dialect supports `@OneToMany`, `@ManyToOne` and `@ManyToMany` relationships.

For example, for `@OneToMany` generates a SQL script:

FetchType.LAZY

```
SELECT
    g1_0.GroupId,
    g1_0.GroupName
FROM
    Groups g1_0
WHERE
    g1_0.GroupName='M3439'

SELECT
    s1_0.GroupId,
    s1_0.StudentId,
    s1_0.StudentName
FROM
    Students s1_0
WHERE
    s1_0.GroupId=?
```

FetchType.EAGER

```
SELECT
    g1_0.GroupId,
    g1_0.GroupName,
    s1_0.GroupId,
    s1_0.StudentId,
    s1_0.StudentName
FROM
    Groups g1_0
JOIN
    Students s1_0
        on g1_0.GroupId=s1_0.GroupId
WHERE
    g1_0.GroupName='M3439'
```

Example with Spring Data JPA

Configure Spring Data JPA with Hibernate to use custom YDB dialect by updating your `application.properties`:

```
spring.jpa.properties.hibernate.dialect=tech.ydb.hibernate.dialect.YdbDialect

spring.datasource.driver-class-name=tech.ydb.jdbc.YdbDriver
spring.datasource.url=jdbc:ydb:<grpc/grpcs>://<host>:<2135/2136>/path/to/database[?saFile=file:~/sa_key.json]
```

Create a simple entity and repository:

Java

```
@Data
@Entity
@Table(name = "employee")
public class Employee {
    @Id
    private long id;

    @Column(name = "full_name")
    private String fullName;

    @Column
    private String email;

    @Column(name = "hire_date")
    private LocalDate hireDate;

    @Column
    private java.math.BigDecimal salary;

    @Column(name = "is_active")
    private boolean isActive;

    @Column
```

```
    private String department;

    @Column
    private int age;
}

public interface EmployeeRepository implements CrudRepository<Employee, Long> {}
```

Kotlin

```
@Entity
@Table(name = "employee")
data class Employee(
    @Id
    val id: Long,

    @Column(name = "full_name")
    val fullName: String,

    @Column
    val email: String,

    @Column(name = "hire_date")
    val hireDate: LocalDate,

    @Column
    val salary: java.math.BigDecimal,

    @Column(name = "is_active")
    val isActive: Boolean,

    @Column
    val department: String,

    @Column
    val age: Int,
)

interface EmployeeRepository : CrudRepository<Employee, Long>

fun EmployeeRepository.findByIdOrNull(id: Long): Employee? = this.findById(id).orElse(null)
```

Usage example:

Java

```
Employee employee = new Employee(
    1,
    "Example",
    "example@bk.com",
    LocalDate.parse("2023-12-20"),
    BigDecimal("500000.000000000"),
    true,
    "YDB AppTeam",
    23
);

/* The following SQL will be executed:
INSERT INTO employee (age,department,email,full_name,hire_date,is_active,limit_date_password,salary,id)
VALUES (?,?,?,?,?,?,?,?,?)
*/
employeeRepository.save(employee);

assertEquals(employee, employeeRepository.findById(employee.getId()).get());

/* The following SQL will be executed:
DELETE FROM employee WHERE id=?
 */
employeeRepository.delete(employee);

assertNull(employeeRepository.findById(employee.getId()).orElse(null));
```

Kotlin

```
val employee = Employee(
    1,
    "Example",
    "example@bk.com",
    LocalDate.parse("2023-12-20"),
    BigDecimal("500000.000000000"),
    true,
    "YDB AppTeam",
    23
)

/* The following SQL will be executed:
INSERT INTO employee (age,department,email,full_name,hire_date,is_active,limit_date_password,salary,id)
VALUES (?,?,?,?,?,?,?,?,?)
*/
```

```
employeeRepository.save(employee)

assertEquals(employee, employeeRepository.findByIdOrNull(employee.id))

/* The following SQL will be executed:
DELETE FROM employee WHERE id=?
 */
employeeRepository.delete(employee)

assertNull(employeeRepository.findByIdOrNull(employee.id))
```

An example of a simple Spring Data JPA repository can be found at the following link.

YDB Dialect for Spring Data JDBC

This guide is intended for use with Spring Data JDBC and YDB.

Spring Data JDBC is part of the Spring Data ecosystem, providing a simplified way to interact with relational databases using SQL and Java objects. Unlike Spring JPA, which relies on JPA (Java Persistence API), Spring Data offers a direct approach to working with databases, bypassing complex ORM (Object-Relational Mapping) for simpler methods.

Installing the YDB dialect

To integrate YDB with a Spring Data JDBC project, it needs two dependencies: the YDB JDBC driver and the Spring Data JDBC extension for YDB.

Examples for different build systems:

Maven

```
<!-- Set actual versions -->
<dependency>
    <groupId>tech.ydb.jdbc</groupId>
    <artifactId>ydb-jdbc-driver</artifactId>
    <version>${ydb.jdbc.version}</version>
</dependency>

<dependency>
    <groupId>tech.ydb.dialects</groupId>
    <artifactId>spring-data-jdbc-ydb</artifactId>
    <version>${spring.data.jdbc.ydb}</version>
</dependency>
```

Gradle

```
dependencies {
    // Set actual versions
    implementation "tech.ydb.dialects:spring-data-jdbc-ydb:$ydbDialectVersion"
    implementation "tech.ydb.jdbc:ydb-jdbc-driver:$ydbJdbcVersion"
}
```

Usage

After importing all the necessary dependencies, the dialect is ready for use. Below is a simple example of a Spring Data JDBC application.

```
spring.datasource.driver-class-name=tech.ydb.jdbc.YdbDriver
spring.datasource.url=jdbc:ydb:<grpc/grpcs>://<host>:<2135/2136>/path/to/database[?saFile=file:~/sa_key.json]
```

```
@Table(name = "Users")
public class User implements Persistable<Long> {
    @Id
    private Long id = ThreadLocalRandom.current().nextLong();

    private String login;
    private String firstname;
    private String lastname;

    @Transient
    private boolean isNew;

    // Constructors, getters and setters

    @Override
    public Long getId() {
      return id;
    }

    @Override
    public boolean isNew() {
      return isNew;
    }

    public void setNew(boolean isNew) {
      this.isNew = isNew;
    }
}
```

Creating a repository for the `User` entity in the `Users` table:

```
public interface SimpleUserRepository extends CrudRepository<User, Long> {
}
```

Saving a new user and verifying that it has been successfully saved:

```
@Component
public class UserRepositoryCommandLineRunner implements CommandLineRunner {
```

```
    @Autowired
    private SimpleUserRepository repository;

    @Override
    public void run(String... args) {
        User user = new User();
        user.setLogin("johndoe");
        user.setFirstname("John");
        user.setLastname("Doe");
        user.setNew(true);  // Setting the flag for the new entity

        // Save user
        User savedUser = repository.save(user);

        // Check saved user
        assertThat(repository.findById(savedUser.getId())).contains(savedUser);

        System.out.println("User saved with ID: " + savedUser.getId());
    }
}
```

View Index

To generate `VIEW INDEX` statements from repository methods, use the `@ViewIndex` annotation. The `@ViewIndex` annotation has two fields:

- `indexName`: The index name.
- `tableName`: The table name to which the `VIEW INDEX` is bound, which is particularly useful when using the `@MappedCollection` annotation.

Here is an example of an index on the Users table by the `login` field:

```
public interface SimpleUserRepository extends CrudRepository<User, Long> {

    @ViewIndex(indexName = "login_index")
    User findByLogin(String login);
}
```

The query generated by this method will look as follows:

```
SELECT
    `Users`.`id` AS `id`,
    `Users`.`login` AS `login`,
    `Users`.`lastname` AS `lastname`,
    `Users`.`firstname` AS `firstname`
FROM `Users` VIEW login_index AS `Users` WHERE `Users`.`login` = ?
```

YdbType

The `@YdbType` annotation allows you to declare a specific YDB data type for an entity field. Here is an example of its usage:

```
    @YdbType("Json")
    private String jsonColumn;
    @YdbType("JsonDocument")
    private String jsonDocumentColumn;
    @YdbType("Uint8")
    private byte uint8Column;
    @YdbType("Uint16")
    private short uint16Column;
    @YdbType("Uint32")
    private int uint32Column;
    @YdbType("Uint64")
    private long uint64Column;
```

Using the `@YdbType` annotation allows you to accurately specify the data types supported by YDB, ensuring proper interaction with the database.

A complete example of a simple Spring Data JDBC repository is available on GitHub.

JOOQ extension for YDB

This guide explains how to use JOOQ with YDB.

JOOQ is a Java library that allows you to create type-safe SQL queries by generating Java classes from a database schema and providing convenient query builders.

Generating Java Classes

You can generate Java classes using any of the tools provided on the official JOOQ website. Two dependencies are required: the YDB JDBC driver and the JOOQ extension for YDB, along with two parameters:

- `database.name` : `tech.ydb.jooq.codegen.YdbDatabase` (mandatory setting)
- `strategy.name` : `tech.ydb.jooq.codegen.YdbGeneratorStrategy` (recommended setting)

An example using the `maven` plugin:

```
<plugin>
    <groupId>org.jooq</groupId>
    <artifactId>jooq-codegen-maven</artifactId>
    <version>3.19.11</version>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>tech.ydb.jdbc</groupId>
            <artifactId>ydb-jdbc-driver</artifactId>
            <version>${ydb.jdbc.version}</version>
        </dependency>
        <dependency>
            <groupId>tech.ydb.dialects</groupId>
            <artifactId>jooq-ydb-dialect</artifactId>
            <version>${jooq.ydb.version}</version>
        </dependency>
    </dependencies>
    <configuration>
        <jdbc>
            <driver>tech.ydb.jdbc.YdbDriver</driver>
            <url>jdbc:ydb:grpc://localhost:2136/local</url>
        </jdbc>
        <generator>
            <strategy>
                <name>tech.ydb.jooq.codegen.YdbGeneratorStrategy</name>
            </strategy>
            <database>
                <name>tech.ydb.jooq.codegen.YdbDatabase</name>
                <!-- excluding system tables -->
                <excludes>.sys.*</excludes>
            </database>
            <target>
                <packageName>ydb</packageName>
                <directory>./src/main/java</directory>
            </target>
        </generator>
    </configuration>
</plugin>
```

Example of generated classes from YQL tutorial (full file contents are available on GitHub):

```
ydb/DefaultCatalog.java
ydb/default_schema
ydb/default_schema/tables
ydb/default_schema/tables/Seasons.java
ydb/default_schema/tables/records
ydb/default_schema/tables/records/SeriesRecord.java
ydb/default_schema/tables/records/EpisodesRecord.java
ydb/default_schema/tables/records/SeasonsRecord.java
ydb/default_schema/tables/Series.java
ydb/default_schema/tables/Episodes.java
ydb/default_schema/Indexes.java
ydb/default_schema/Keys.java
ydb/default_schema/Tables.java
ydb/default_schema/DefaultSchema.java
```

Usage

To integrate YDB with JOOQ into your project, you need to add two dependencies: YDB JDBC Driver and the JOOQ extension for YDB.

Examples for different build systems:

Maven

```
<!-- Set actual versions -->
<dependency>
```

```
    <groupId>tech.ydb.jdbc</groupId>
    <artifactId>ydb-jdbc-driver</artifactId>
    <version>${ydb.jdbc.version}</version>
</dependency>

<dependency>
    <groupId>tech.ydb.dialects</groupId>
    <artifactId>jooq-ydb-dialect</artifactId>
    <version>${jooq.ydb.dialect.version}</version>
</dependency>
```

Gradle

```
dependencies {
    // Set actual versions
    implementation "tech.ydb.dialects:jooq-ydb-dialect:$jooqYdbDialectVersion"
    implementation "tech.ydb.jdbc:ydb-jdbc-driver:$ydbJdbcVersion"
}
```

To obtain a `YdbDSLContext` class instance (an extension of `org.jooq.DSLContext`), use the `tech.ydb.jooq.YDB` class. For example:

```
String url = "jdbc:ydb:<schema>://<host>:<port>/path/to/database[?saFile=file:~/sa_key.json]";
Connection conn = DriverManager.getConnection(url);

YdbDSLContext dsl = YDB.using(conn);
```

or

```
String url = "jdbc:ydb:<schema>://<host>:<port>/path/to/database[?saFile=file:~/sa_key.json]";
try(CloseableYdbDSLContext dsl = YDB.using(url)) {
    // ...
}
```

`YdbDSLContext` is ready to use.

YQL statements

The following statements are available from the YQL syntax in `YdbDSLContext`:

- `UPSERT`:

```
// generated SQL:
// upsert into `episodes` (`series_id`, `season_id`, `episode_id`, `title`, `air_date`)
// values (?, ?, ?, ?, ?)
public void upsert(YdbDSLContext context) {
    context.upsertInto(EPISODES)
            .set(record)
            .execute();
}
```

- `REPLACE`:

```
// generated SQL:
// replace into `episodes` (`series_id`, `season_id`, `episode_id`, `title`, `air_date`)
// values (?, ?, ?, ?, ?)
public void replace(YdbDSLContext context) {
    ydbDSLContext.replaceInto(EPISODES)
            .set(record)
            .execute();
}
```

- `VIEW index_name`:

```
// generated SQL:
// select `series`.`series_id`, `series`.`title`, `series`.`series_info`, `series`.`release_date`
// from `series` view `title_name` where `series`.`title` = ?
var record = ydbDSLContext.selectFrom(SERIES.useIndex(Indexes.TITLE_NAME.name))
        .where(SERIES.TITLE.eq(title))
        .fetchOne();
```

In all other respects, the YDB dialect follows the JOOQ documentation.

Spring Boot Configuration

Extend `JooqAutoConfiguration.DslContextConfiguration` with your own `YdbDSLContext`. For example:

```
@Configuration
public class YdbJooqConfiguration extends JooqAutoConfiguration.DslContextConfiguration {

    @Override
    public YdbDSLContextImpl dslContext(org.jooq.Configuration configuration) {
        return YdbDSLContextImpl(configuration);
```

```
    }
}
```

```
spring.datasource.driver-class-name=tech.ydb.jdbc.YdbDriver
spring.datasource.url=jdbc:ydb:<schema>://<host>:<port>/path/to/database[?saFile=file:~/sa_key.json]
```

A complete example of a simple Spring Boot application can be found on GitHub.

Using Dapper

Dapper is a micro ORM (Object-Relational Mapping) tool that provides a simple and flexible way to interact with databases. It operates on top of the ADO.NET standard and offers various features that simplify database operations.

Getting started

To get started, you need an additional dependency Dapper.

Let's consider a complete example:

```
using Dapper;
using Ydb.Sdk.Ado;

await using var connection = await new YdbDataSource().OpenConnectionAsync();

await connection.ExecuteAsync("""
                              CREATE TABLE Users(
                                  Id Int32,
                                  Name Text,
                                  Email Text,
                                  PRIMARY KEY (Id)
                              );
                              """);

await connection.ExecuteAsync("INSERT INTO Users(Id, Name, Email) VALUES (@Id, @Name, @Email)",
    new User { Id = 1, Name = "Name", Email = "Email" });

Console.WriteLine(await connection.QuerySingleAsync<User>("SELECT * FROM Users WHERE Id = @Id", new { Id = 1
}));

await connection.ExecuteAsync("DROP TABLE Users");

internal class User
{
    public int Id { get; init; }
    public string Name { get; init; } = null!;
    public string Email { get; init; } = null!;

    public override string ToString()
    {
        return $"Id: {Id}, Name: {Name}, Email: {Email}";
    }
}
```

For more information, refer to the official documentation.

Important aspects

For Dapper to interpret `DateTime` values as the YDB type `DateTime`, execute the following code:

```
SqlMapper.AddTypeMap(typeof(DateTime), DbType.DateTime);
```

By default, `DateTime` is interpreted as `Timestamp`.

YDB Entity Framework Core Provider

YDB has an Entity Framework (EF) Core provider, an object-relational mapper (ORM) that enables .NET developers to work with a YDB database using .NET objects. It behaves like other EF Core providers (for example, SQL Server). If you're just getting started with EF Core, the EF Core documentation is the best place to start.

EF Core Provider is being developed in the open GitHub repository, all problems should be reported there.

Set Up the YDB Entity Framework Core Provider

To get started, you need to add the necessary NuGet packages to your project:

```
dotnet add package EntityFrameworkCore.Ydb
```

Defining a Model and a DbContext

Let's say you want to store blogs and their posts in a database. You can model these as .NET types as follows:

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

You then define a `DbContext` type, which you'll use to interact with the database:

OnConfiguring

Using `OnConfiguring()` to configure your context is the easiest way to get started but is discouraged for most production applications:

```
public class BloggingContext : DbContext
  {
  public DbSet<Blog> Blogs { get; set; }
  public DbSet<Post> Posts { get; set; }

      protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
          => optionsBuilder.UseYdb("<connection string>");
  }

  // At the point where you need to perform a database operation:
  using var context = new BloggingContext();
  // Use the context...
```

DbContext pooling

```
var dbContextFactory = new PooledDbContextFactory<BloggingContext>(
  new DbContextOptionsBuilder<BloggingContext>()
      .UseYdb("<connection string>")
      .Options);

// At the point where you need to perform a database operation:
using var context = dbContextFactory.CreateDbContext();
// Use the context...
```

ASP.NET / DI

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContextPool<BloggingContext>(opt =>
    opt.UseYdb(builder.Configuration.GetConnectionString("BloggingContext")));

public class BloggingContext(DbContextOptions<BloggingContext> options) : DbContext(options)
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}
```

For more information on getting started with EF, see the Getting Started guide.

Additional YDB Configuration

The Entity Framework (EF) Core provider for YDB has its own additional configuration parameters.

Connection with Yandex Cloud

You can find more information about different authentication methods for Yandex Cloud in the ADO.NET documentation.

Below is an example of how to specify the necessary parameters for connecting to Yandex Cloud using Entity Framework:

```
.UseYdb(cmd.ConnectionString, builder => builder
    .WithCredentialsProvider(saProvider)
    .WithServerCertificates(YcCerts.GetYcServerCertificates())
)
```

Schema Migration

To ensure that database schema migrations are executed correctly, disable the automatic retry strategy (`ExecutionStrategy`), which is enabled by default in `EntityFrameworkCore.Ydb`.

To do this, explicitly override the `IDesignTimeDbContextFactory` interface and use the `DisableRetryOnFailure()` method.

An example implementation of a context factory for migrations is shown below:

```
internal class BloggingContextFactory : IDesignTimeDbContextFactory<BloggingContext>
{
    public BloggingContext CreateDbContext(string[] args)
    {
        var optionsBuilder = new DbContextOptionsBuilder<BloggingContext>();

        return new BloggingContext(
            optionsBuilder.UseYdb("Host=localhost;Port=2136;Database=/local",
                builder => builder.DisableRetryOnFailure()
            ).Options
        );
    }
}
```

Note

This factory class is required to execute migration commands. For example:

```
dotnet ef migrations add MyMigration
dotnet ef database update
```

Examples

You can find complete usage examples on GitHub.

YDB Provider for LinqToDB

Overview

This guide explains how to use LinqToDB with YDB.

linq2db is a lightweight and fast ORM for .NET that provides type-safe LINQ queries and precise SQL control. The YDB provider generates correct YQL, supports YDB types, schema generation, and bulk operations (Bulk Copy).

Installing the YDB provider

Add dependencies:

dotnet CLI

```
dotnet add package Community.Ydb.Linq2db
dotnet add package linq2db
```

csproj (PackageReference)

```
<ItemGroup>
    <PackageReference Include="Community.Ydb.Linq2db" Version="$(CommunityYdbLinqToDbVersion)" />
    <PackageReference Include="linq2db" Version="$(LinqToDbVersion)" />
</ItemGroup>
```

Provider Configuration

Configure LinqToDB to use YDB in code:

C#

```
DataConnection.AddProviderDetector(YdbTools.ProviderDetector);

// Option 1: local YDB via connection string
//
// Example: local YDB running on localhost:2136 with database "/local"
using var localDb = new DataConnection(
    new DataOptions().UseConnectionString(
        "YDB",
        "Host=localhost;Port=2136;Database=/local;UseTls=false"
    )
);

// Option 2: managed YDB in cloud only via YdbConnectionStringBuilder
//
// Example: connection built from explicit host/port/database settings.
static async Task<DataConnection> BuildYdbDataConnection()
{
    var ydbConnectionBuilder = new YdbConnectionStringBuilder
    {
        Host     = "server",
        Port     = 2135,
        Database = "/ru-prestable/my-table",
        UseTls   = true
    };

    await using var ydbConnection = new YdbConnection(ydbConnectionBuilder);
    return YdbTools.CreateDataConnection(ydbConnection);
}
```

Usage

Use the provider like any other Linq To DB provider: map your entity classes to tables and run queries via `DataConnection` / `ITable<T>`. Below you'll find the type mapping table and schema generation examples.

Type Mapping between .NET and YDB

.NET type(s)	Linq To DB `DataType`	YDB type	Notes
`bool`	`Boolean`	`Bool`	—
`string`	`NVarChar` / `VarChar` / `Char` / `NChar`	`Text`	UTF-8 text. Text and Utf8 are the same in YDB; the DDL generator may output either
`byte[]`	`VarBinary` / `Binary` / `Blob`	`Bytes`	Binary data. Bytes and String are equivalent in YDB; DDL may show either
`Guid`	`Guid`	`Uuid`	128-bit RFC 4122 UUID. No version enforcement (v1/v4/v7). Generate the desired version in the app
`DateOnly` / `DateTime`	`Date`	`Date`	Date-only value stored as YDB `Date` in UTC, range `1970-01-01` .. `2106-01-01`. .NET `DateOnly` / `DateTime` support a wider range (`0001` .. `9999`), but values outside the YDB range cannot be stored.

`DateTime`	`DateTime`	`Datetime`	YDB `Datetime` with second precision, UTC instant in the range `1970-01-01` .. `2106-01-01`. .NET `DateTime` supports a wider range; values outside the YDB range are not supported. `DateTime.Kind` is not stored; the provider expects UTC values.
`DateTime` / `DateTimeOffset`	`DateTime2`	`Timestamp`	YDB `Timestamp` with microsecond precision, UTC instant `1970-01-01` .. `2106-01-01`. When writing `DateTimeOffset`, the value is converted to UTC; the offset/time zone is not persisted.
`TimeSpan`	`Interval`	`Interval`	Duration with microsecond precision. .NET `TimeSpan` range is wider than YDB `Interval`; values outside the YDB range are not supported. Sub-microsecond ticks are truncated.
`decimal`	`Decimal`	`Decimal(p,s)`	Default: Decimal(22,9). To use custom precision/scale, specify Decimal(p,s). Example
`float`	`Single`	`Float`	—
`double`	`Double`	`Double`	—
`sbyte` / `byte`	`SByte` / `Byte`	`Int8` / `Uint8`	—
`short` / `ushort`	`Int16` / `UInt16`	`Int16` / `Uint16`	—
`int` / `uint`	`Int32` / `UInt32`	`Int32` / `Uint32`	—
`long` / `ulong`	`Int64` / `UInt64`	`Int64` / `Uint64`	—
`string`	`Json`	`Json`	Text JSON.
`byte[]`	`BinaryJson`	`JsonDocument`	Binary JSON.
`DateOnly` / `DateTime`	`Date`	`Date32`	Wider date range. Use `DbType = "Date32"`. Example
`DateTime`	`DateTime`	`Datetime64`	Second precision, wider range. `DbType = "Datetime64"`. Example
`DateTime` / `DateTimeOffset`	`DateTime2`	`Timestamp64`	Microseconds, wider range. `DbType = "Timestamp64"`. Example
`TimeSpan`	`Interval`	`Interval64`	Wider interval range. `DbType = "Interval64"`. Example

Tip

You can set exact `Precision` / `Scale` with attributes: `[Column(DataType = DataType.Decimal, Precision = 22, Scale = 9)]`.

Note

By default (when `DbType` is not specified), the provider uses legacy YDB temporal types: `Date`, `Datetime`, `Timestamp`, `Interval`. To opt in per column to extended types, set `DbType`, e.g. `[Column(DbType = "Date32")]`. Both families can coexist in the same table.

Custom precision scale example

C#

```
[Table("amounts")]
public sealed class AmountRow
{
    [PrimaryKey] public long Id { get; set; }

    // Custom precision & scale: Decimal(25,10)
    [Column("amount", DataType = DataType.Decimal, Precision = 25, Scale = 10), NotNull]
    public decimal Amount { get; set; }
}
```

DbType override example

C#

```
[Table("events")]
public sealed class EventRow
{
[PrimaryKey] public long Id { get; set; }

    // Extended-range timestamp
    [Column("happened_at", DbType = "Timestamp64"), NotNull]
    public DateTime HappenedAt { get; set; }

    // Extended-range date
    [Column("due_on", DbType = "Date32"), NotNull]
```

```
    public DateTime DueOn { get; set; }

    // Second precision date
    [Column("made_at", DbType = "Datetime64"), NotNull]
    public DateTime MadeAt { get; set; }

    // Extended-range interval
    [Column("duration", DbType = "Interval64"), NotNull]
    public TimeSpan Duration { get; set; }
}
```

Schema Generation from Attributes

Describe an entity using Linq To DB attributes, then explicitly create the schema by calling db.CreateTable(). At that moment the table is created, and any [Index] attributes are applied as database indexes.

C#

```
using LinqToDB.Mapping;

[Table(Name = "Groups")]
[Index("GroupName", Name = "group_name_index")]
public class Group
{
    [PrimaryKey, Column("GroupId")]
    public int Id { get; set; }

    [Column("GroupName")]
    public string? Name { get; set; }
}
```

Generated DDL (YDB):

```
CREATE TABLE Groups (
    GroupId Int32 NOT NULL,
    GroupName Utf8,
    PRIMARY KEY (GroupId)
);

ALTER TABLE Groups
  ADD INDEX group_name_index GLOBAL
       ON (GroupName);
```

Schema evolution (adding a column)

Add the `Department` column to the `Group` entity:

C#

```
[Column] public string? Department { get; set; }
```

Schema change (DDL):

```
ALTER TABLE Groups
   ADD COLUMN Department Utf8;
```

> **Note**
>
> Linq To DB doesn't manage migrations. The DDL below is illustrative—apply it with Liquibase/Flyway (recommended). For quick local changes you can also run it directly with db.Execute(...) or the YDB CLI.

C#

```
using var db = new DataConnection("YDB", connectionString);

// Apply the schema change (adds the column):
db.Execute(@"ALTER TABLE Groups
   ADD COLUMN Department Utf8;");
```

YDB Indexes how to set parameters

With the [Index] attribute you can set name and columns. The provider creates a GLOBAL secondary index.
Parameters like ASYNC/SYNC and COVER(...) are not supported via the attribute; add them using a separate DDL statement after table creation.

Option A — attribute (name + columns)

C#

```
[Table(Name = "Groups", IsColumnAttributeRequired = false)]
[Index("GroupName", Name = "group_name_index")]
public class Group
{
    [PrimaryKey, Column("GroupId")] public int Id { get; set; }
```

```
    [Column("GroupName")] public string? Name { get; set; }

    // A column you may include into COVER via separate DDL
    [Column] public string? Department { get; set; }
}

// When you call db.CreateTable<Group>(), a GLOBAL index on GroupName is created.
```

The generated DDL is effectively

```
CREATE TABLE Groups (
    GroupId Int32 NOT NULL,
    GroupName Utf8,
    Department Utf8,
    PRIMARY KEY (GroupId)
);

ALTER TABLE Groups
  ADD INDEX group_name_index GLOBAL
       ON (GroupName);
```

Option B — extended parameters (ASYNC, COVER) via separate DDL

C#

```
[Table(Name = "Groups", IsColumnAttributeRequired = false)]
public class Group
{
    [PrimaryKey, Column("GroupId")] public int Id { get; set; }
    [Column("GroupName")] public string? Name { get; set; }
    [Column] public string? Department { get; set; }
}

public static class Demo
{
    public static void Main()
    {
        using var db = new DataConnection("YDB", connectionString);

        // 1) Create table from attributes (avoid duplicate indexes if you plan to add them via DDL)
        db.CreateTable<Group>();

        // 2) Add an index with ASYNC and COVER parameters
        db.Execute(@"
ALTER TABLE Groups
  ADD INDEX group_name_index GLOBAL ASYNC
       ON (GroupName)
       COVER (Department);
");
    }
}
```

Relationships between entities

Describe navigations using `[Association]` attributes. Example of one-to-many between `Group` and `Student`:

C#

```
[Table("Students")]
public class Student
{
    [PrimaryKey, Column("StudentId")]
    public int Id { get; set; }

    [Column("StudentName")]
    public string Name { get; set; } = null!;

    [Column]
    public int GroupId { get; set; }

    // many-to-one
    [Association(ThisKey = nameof(GroupId), OtherKey = nameof(Group.Id), CanBeNull = false)]
    public Group Group { get; set; } = null!;
}

[Table("Groups")]
public class Group
{
    [PrimaryKey, Column("GroupId")]
    public int Id { get; set; }

    [Column("GroupName")]
    public string? Name { get; set; }

    // one-to-many
    [Association(ThisKey = nameof(Id), OtherKey = nameof(Student.GroupId))]
    public IEnumerable<Student> Students { get; set; } = null!;
}
```

Created tables:

```
CREATE TABLE Groups (
    GroupId Int32 NOT NULL,
    GroupName Utf8,
    PRIMARY KEY (GroupId)
);

CREATE TABLE Students (
    StudentId Int32 NOT NULL,
    StudentName Utf8,
    GroupId Int32,
    PRIMARY KEY (StudentId)
);
```

Example generated YQL for relationship queries

C#: “Lazy” loading (two queries)

```
  // 1) SELECT g.GroupId, g.GroupName FROM Groups AS g WHERE g.GroupName = 'M3439';
  var grp = db.GetTable<Group>()
  .Where(g => g.Name == "M3439")
  .Select(g => new { g.Id, g.Name })
  .FirstOrDefault();

  // 2) SELECT s.StudentId, s.StudentName, s.GroupId FROM Students AS s WHERE s.GroupId = ?;
  var students = grp == null
  ? new List<Student>()
  : db.GetTable<Student>()
  .Where(s => s.GroupId == grp.Id)
  .Select(s => new { s.Id, s.Name, s.GroupId })
  .ToList();
```

C#: “Eager” loading (JOIN)

```
  var joined =
  (from g in db.GetTable<Group>()
  join s in db.GetTable<Student>() on g.Id equals s.GroupId
  where g.Name == "M3439"
  select new { GroupId = g.Id, GroupName = g.Name, StudentId = s.Id, StudentName = s.Name, s.GroupId })
  .ToList();
```

YQL: “Lazy” loading (two queries)

```
SELECT
    g.GroupId,
    g.GroupName
FROM
    Groups AS g
WHERE
    g.GroupName = 'M3439';

SELECT
    s.StudentId,
    s.StudentName,
    s.GroupId
FROM
    Students AS s
WHERE
    s.GroupId = ?;
```

YQL: “Eager” loading (JOIN)

```
SELECT
    g.GroupId,
    g.GroupName,
    s.StudentId,
    s.StudentName,
    s.GroupId
FROM
    Groups AS g
JOIN
    Students AS s
  ON g.GroupId = s.GroupId
WHERE
    g.GroupName = 'M3439';
```

Bulk operations: insert, update and delete

BulkCopy:

In the YDB provider, `BulkCopy` uses the native `BulkUpsert API` and does not generate textual YQL. Rows are sent to YDB as a stream of strongly-typed values over the SDK's binary protocol, so there are no DECLARE statements or ? placeholders

C#

```
var now  = DateTime.UtcNow;
var data = Enumerable.Range(0, 15_000).Select(i => new SimpleEntity
{
    Id      = i,
    IntVal  = i,
    DecVal  = 0m,
    StrVal  = $"Name {i}",
    BoolVal = (i & 1) == 0,
    DtVal   = now,
});

  db.BulkCopy(data)
```

Massive Update (WHERE IN):

C#

```
var ids = Enumerable.Range(0, 15_000).ToArray();

table.Where(t => ids.Contains(t.Id))
    .Set(_ => _.DecVal, 1.23m)
    .Set(_ => _.StrVal, "updated")
    .Set(_ => _.BoolVal, true)
    .Update();
```

YQL

```
DECLARE $Gen_List_Primitive_1 AS List<Int32>;
UPDATE
    SimpleEntity
SET
    DecVal = Decimal('1.23', 22, 9),
    StrVal = 'updated'u,
    BoolVal = true
WHERE
    SimpleEntity.Id IN $Gen_List_Primitive_1
```

Mass deletion (WHERE IN):

C#

```
table.Delete(t => ids.Contains(t.Id));
```

YQL

```
DECLARE $Gen_List_Primitive_1 AS List<Int32>;
DELETE FROM
    SimpleEntity
WHERE
    SimpleEntity.Id IN $Gen_List_Primitive_1
```

A complete example

This section shows a practical, production-style entity from a typical domain. It demonstrates:

- a realistic column set (name, email, hire date, salary, flags, ints);
- precise types (e.g., Decimal(22,9) for money-like values, Date for dates, Utf8/Bool/Int32/Int64);
- a GLOBAL secondary index on full_name for lookups and ordering;
- end-to-end flow: mapping → table creation → CRUD with generated YQL/DDL.

When you call `db.CreateTable<Employee>()`, Linq To DB creates the table and applies the [Index] attribute as a YDB GLOBAL index.

C#

```
using System;
using LinqToDB.Mapping;

[Table("employee")]
[Index("full_name", Name = "employee_full_name_idx")]
public class Employee
{
    [PrimaryKey] public long Id { get; set; }

    [Column("full_name"), NotNull] public string FullName { get; set; } = null!;

    [Column, NotNull] public string Email { get; set; } = null!;

    [Column("hire_date", DataType = DataType.Date), NotNull]
    public DateTime HireDate { get; set; }

    [Column(DataType = DataType.Decimal, Precision = 22, Scale = 9), NotNull]
    public decimal Salary { get; set; }
```

```
    [Column("is_active"), NotNull] public bool IsActive { get; set; }

    [Column, NotNull] public string Department { get; set; } = null!;

    [Column, NotNull] public int Age { get; set; }
}
```

Generated DDL:

```
CREATE TABLE employee (
    Id Int64 NOT NULL,
    full_name Utf8 NOT NULL,
    Email Utf8 NOT NULL,
    hire_date Date NOT NULL,
    Salary Decimal(22,9) NOT NULL,
    is_active Bool NOT NULL,
    Department Utf8 NOT NULL,
    Age Int32 NOT NULL,
    PRIMARY KEY (Id)
);

ALTER TABLE employee
  ADD INDEX employee_full_name_idx GLOBAL
       ON (full_name);
```

Usage example

C#

```
using System;
using System.Linq;
using LinqToDB;
using LinqToDB.Data;

// This example uses a direct connection string and is intended for local / test YDB instances.
// If you need to connect to managed YDB in Yandex Cloud , use YdbConnectionStringBuilder
// and YdbConnection instead (see the provider configuration section)

using var db = new DataConnection(
  new DataOptions().UseConnectionString(
    "YDB",
    "Host=localhost;Port=2136;Database=/local;UseTls=false"
  )
);

// INSERT
var employee = new Employee
{
    Id         = 1L,
    FullName   = "Example",
    Email      = "example@example.com",
    HireDate   = new DateTime(2023, 12, 20),
    Salary     = 500000.000000000m,
    IsActive   = true,
    Department = "YDB AppTeam",
    Age        = 23,
};
db.Insert(employee);

// SELECT by primary key
var loaded = db.GetTable<Employee>()
              .FirstOrDefault(e => e.Id == employee.Id);

// UPDATE Email/Department/Salary by primary key
db.GetTable<Employee>()
  .Where(e => e.Id == employee.Id)
  .Set(e => e.Email,      "example+updated@example.com")
  .Set(e => e.Department, "Analytics")
  .Set(e => e.Salary,     550000.000000000m)
  .Update();

// DELETE by primary key
db.GetTable<Employee>()
  .Where(e => e.Id == employee.Id)
  .Delete();
```

YQL samples generated by the provider for simple operations:

- Insert a single row

```
INSERT INTO employee (Age,Department,Email,full_name,hire_date,is_active,Salary,Id)
VALUES (?,?,?,?,?,?,?,?);
```

- Read by primary key

```
SELECT
    e.Id, e.full_name, e.Email, e.hire_date, e.Salary, e.is_active, e.Department, e.Age
```

```
FROM employee AS e
WHERE e.Id = ?;
```

- Update by primary key

```
UPDATE employee
SET
    Email      = ?,
    Department = ?,
    Salary     = ?
WHERE Id = ?;
```

> **Note**
>
> The provider emits parameters (?) because values and types are bound via the driver, not declared in the query text. When YQL requires typed parameters, the provider adds the necessary DECLARE statements automatically. For non-standard patterns such as upsert-style writes, use YDB's UPSERT with parameter binding—the provider generates these statements as regular YQL with parameters.

- Delete by primary key

```
DELETE FROM employee WHERE Id = ?;
```

SQLAlchemy

SQLAlchemy is a popular Python library for working with databases, providing both ORM (Object-Relational Mapping) and Core API for executing SQL queries.

YDB supports integration with SQLAlchemy through a special `ydb-sqlalchemy` dialect, which provides full compatibility with SQLAlchemy 2.0 and partial support for SQLAlchemy 1.4.

Installation

Install the `ydb-sqlalchemy` package using pip:

```
pip install ydb-sqlalchemy
```

Quick Start

Synchronous Connection

```
import sqlalchemy as sa

# Create engine
engine = sa.create_engine("yql+ydb://localhost:2136/local")

# Execute query
with engine.connect() as conn:
    result = conn.execute(sa.text("SELECT 1 AS value"))
    print(result.fetchone())
```

Asynchronous Connection

```
import asyncio
import sqlalchemy as sa
from sqlalchemy.ext.asyncio import create_async_engine

async def main():
    # Create async engine
    engine = create_async_engine("yql+ydb_async://localhost:2136/local")

    # Execute query
    async with engine.connect() as conn:
        result = await conn.execute(sa.text("SELECT 1 AS value"))
        print(await result.fetchone())

asyncio.run(main())
```

Connection Configuration

Authentication Methods

Anonymous Access

For local development or testing:

```
import sqlalchemy as sa

engine = sa.create_engine("yql+ydb://localhost:2136/local")
```

Static Credentials

Use username and password authentication:

```
engine = sa.create_engine(
    "yql+ydb://localhost:2136/local",
    connect_args={
        "credentials": {
            "username": "your_username",
            "password": "your_password"
        }
    }
)
```

Token Authentication

Use access token for authentication:

```
engine = sa.create_engine(
    "yql+ydb://localhost:2136/local",
    connect_args={
        "credentials": {
            "token": "your_access_token"
        }
    }
)
```

Service Account Authentication

Use service account JSON key:

```
import json

# Load from file
with open('service_account_key.json', 'r') as f:
    service_account_json = json.load(f)

engine = sa.create_engine(
    "yql+ydb://localhost:2136/local",
    connect_args={
        "credentials": {
            "service_account_json": service_account_json
        }
    }
)

# Or pass JSON directly
engine = sa.create_engine(
    "yql+ydb://localhost:2136/local",
    connect_args={
        "credentials": {
            "service_account_json": {
                "id": "your_key_id",
                "service_account_id": "your_service_account_id",
                "created_at": "2023-01-01T00:00:00Z",
                "key_algorithm": "RSA_2048",
                "public_key": "-----BEGIN PUBLIC KEY-----\n...\n-----END PUBLIC KEY-----",
                "private_key": "-----BEGIN PRIVATE KEY-----\n...\n-----END PRIVATE KEY-----"
            }
        }
    }
)
```

Using YDB SDK Credentials

You can use any available authentication methods from YDB Python SDK:

```
import ydb.iam

# Metadata service
engine = sa.create_engine(
    "yql+ydb://localhost:2136/local",
    connect_args={
        "credentials": ydb.iam.MetadataUrlCredentials()
    }
)

# OAuth token
engine = sa.create_engine(
    "yql+ydb://localhost:2136/local",
    connect_args={
        "credentials": ydb.iam.OAuthCredentials("your_oauth_token")
    }
)

# Static credentials
engine = sa.create_engine(
    "yql+ydb://localhost:2136/local",
    connect_args={
        "credentials": ydb.iam.StaticCredentials("username", "password")
    }
)
```

TLS Configuration

For secure connections to YDB:

```
engine = sa.create_engine(
    "yql+ydb://ydb.example.com:2135/prod",
    connect_args={
        "credentials": {"token": "your_token"},
        "protocol": "grpc",
        "root_certificates_path": "/path/to/ca-certificates.crt",
        # "root_certificates": crt_string,  # Alternative - certificate string
    }
)
```

Supported Data Types

YDB SQLAlchemy provides comprehensive support for YDB data types through custom SQLAlchemy types. For detailed information about YDB data types, see the YDB Data Types Documentation.

Type Mapping Summary

YDB Type	YDB SQLAlchemy Type	Standard SQLAlchemy Type	Python Type	Notes
`Bool`	`Boolean`	`Boolean`	`bool`	
`Int8`	`Int8`		`int`	-2^7 to 2^7-1
`Int16`	`Int16`		`int`	-2^15 to 2^15-1
`Int32`	`Int32`		`int`	-2^31 to 2^31-1
`Int64`	`Int64`	`Integer`	`int`	-2^63 to 2^63-1
`Uint8`	`UInt8`		`int`	0 to 2^8-1
`Uint16`	`UInt16`		`int`	0 to 2^16-1
`Uint32`	`UInt32`		`int`	0 to 2^32-1
`Uint64`	`UInt64`		`int`	0 to 2^64-1
`Float`	`Float`	`Float`	`float`	
`Double`	`Double`		`float`	Available in SQLAlchemy 2.0+
`Decimal(p,s)`	`Decimal`	`DECIMAL`	`decimal.Decimal`	
`String`		`BINARY`	`bytes`	
`Utf8`		`String/Text`	`str`	
`Date`	`YqlDate`	`Date`	`datetime.date`	
`Date32`	`YqlDate32`		`datetime.date`	Extended date range support
`Datetime`	`YqlDateTime`	`DATETIME`	`datetime.datetime`	
`Datetime64`	`YqlDateTime64`		`datetime.datetime`	Extended range
`Timestamp`	`YqlTimestamp`	`TIMESTAMP`	`datetime.datetime`	
`Timestamp64`	`YqlTimestamp64`		`datetime.datetime`	Extended range
`Json`	`YqlJSON`	`JSON`	`dict/list`	
`List<T>`	`ListType`	`ARRAY`	`list`	
`Struct<...>`	`StructType`		`dict`	
`Optional<T>`	`nullable=True`		`None + base type`	

Migrations with Alembic

YDB-Specific Configuration

YDB requires special configuration in `env.py` due to its unique characteristics:

```
# migrations/env.py
from logging.config import fileConfig
import sqlalchemy as sa
from sqlalchemy import engine_from_config, pool
from alembic import context
from alembic.ddl.impl import DefaultImpl

# Import your models
from myapp.models import Base

config = context.config

if config.config_file_name is not None:
    fileConfig(config.config_file_name)

target_metadata = Base.metadata

# YDB-specific implementation
class YDBImpl(DefaultImpl):
    __dialect__ = "yql"

def run_migrations_offline() -> None:
    """Run migrations in 'offline' mode."""
    url = config.get_main_option("sqlalchemy.url")
    context.configure(
        url=url,
        target_metadata=target_metadata,
        literal_binds=True,
        dialect_opts={"paramstyle": "named"},
    )
```

```
    with context.begin_transaction():
        context.run_migrations()

def run_migrations_online() -> None:
    """Run migrations in 'online' mode."""
    connectable = engine_from_config(
        config.get_section(config.config_ini_section, {}),
        prefix="sqlalchemy.",
        poolclass=pool.NullPool,
    )

    with connectable.connect() as connection:
        context.configure(
            connection=connection,
            target_metadata=target_metadata
        )

        with context.begin_transaction():
            context.run_migrations()

if context.is_offline_mode():
    run_migrations_offline()
else:
    run_migrations_online()
```

Useful Links

- Examples on GitHub
- PyPI Package

Django

Django is a popular Python web framework with a powerful ORM for working with databases.

YDB supports integration with Django through a special `django-ydb-backend` backend, which provides full Django ORM support for working with YDB.

Quick Start

Installation

Install the `django-ydb-backend` package using pip:

```
pip install django-ydb-backend
```

Connection Setup

Add YDB to your Django settings in `settings.py`:

```
DATABASES = {
    "default": {
        "NAME": "ydb_db",
        "ENGINE": "ydb_backend.backend",
        "HOST": "localhost",
        "PORT": "2136",
        "DATABASE": "/local",
    }
}
```

Configuration

DATABASES

Required parameters:

- **NAME** (required): traditional Django database name
- **ENGINE** (required): must be set to `ydb_backend.backend`
- **HOST** (required): hostname or IP address of the YDB server (e.g., "localhost")
- **PORT** (required): gRPC port YDB is running on (default is 2136)
- **DATABASE** (required): full path to your YDB database (e.g., "/local" for local testing or "/my_production_db")

```
DATABASES = {
    "default": {
        "NAME": "ydb_db",
        "ENGINE": "ydb_backend.backend",
        "HOST": "localhost",
        "PORT": "2136",
        "DATABASE": "/local",
    }
}
```

Authentication Methods

Anonymous Authentication

To use anonymous authentication, you don't need to pass any additional parameters.

Static Authentication

To use static credentials, you should provide `username` and `password`:

```
DATABASES = {
    "default": {
        "ENGINE": "ydb_backend.backend",
        "CREDENTIALS": {
            "username": "...",
            "password": "..."
        }
    }
}
```

Token Authentication

To use access token credentials, you should provide `token`:

```
DATABASES = {
    "default": {
        "ENGINE": "ydb_backend.backend",
        "CREDENTIALS": {
            "token": "..."
        },
    }
}
```

Service Account Authentication

To use service account credentials, you should provide `service_account_json`:

```
DATABASES = {
    "default": {
        "ENGINE": "ydb_backend.backend",
        "CREDENTIALS": {
            "service_account_json": {
                "id": "...",
                "service_account_id": "...",
                "created_at": "...",
                "key_algorithm": "...",
                "public_key": "...",
                "private_key": "..."
            }
        }
    }
}
```

Django Fields

YDB backend supports Django builtin fields.

Note: ForeignKey, ManyToManyField or even OneToOneField could be used with YDB backend. However, it's important to note that these relationships won't enforce database-level constraints, which may lead to potential data consistency issues.

Supported Django Fields

Class	YDB Type	Python Type	Comments
`SmallAutoField`	`Int16`	`int`	YDB type SmallSerial will generate value automatically. Range -32768 to 32767
`AutoField`	`Int32`	`int`	YDB type Serial will generate value automatically. Range -2147483648 to 2147483647
`BigAutoField`	`Int64`	`int`	YDB type BigSerial will generate value automatically. Range -9223372036854775808 to 9223372036854775807
`CharField`	`UTF-8`	`str`	
`TextField`	`UTF-8`	`str`	
`BinaryField`	`String`	`bytes`	
`SlugField`	`UTF-8`	`str`	
`FileField`	`String`	`bytes`	
`FilePathField`	`UTF-8`	`str`	
`DateField`	`Date`	`datetime.date`	Range of values for all time types except Interval: From 00:00 01.01.1970 to 00:00 01.01.2106. Internal Date representation: Unsigned 16-bit integer
`DateTimeField`	`DateTime`	`datetime.datetime`	Internal representation: Unsigned 32-bit integer
`DurationField`	`Interval`	`int`	The range of values is from -136 years to +136 years. The internal representation is a 64–bit signed integer. Cannot be used in the primary key
`SmallIntegerField`	`Int16`	`int`	Range -32768 to 32767
`IntegerField`	`Int32`	`int`	Range -2147483648 to 2147483647
`BigIntegerField`	`Int64`	`int`	Range -9223372036854775808 to 9223372036854775807
`PositiveSmallIntegerField`	`UInt16`	`int`	Range 0 to 65535
`PositiveIntegerField`	`UInt32`	`int`	Range 0 to 4294967295
`PositiveBigIntegerField`	`UInt64`	`int`	Range 0 to 18446744073709551615
`FloatField`	`Float`	`float`	A real number with variable precision, 4 bytes in size. Can't be used in the primary key
`DecimalField`	`Decimal`	`Decimal`	Pythonic values are rounded to fit the scale of the database field. Supports only Decimal(22,9)
`UUIDField`	`UUID`	`uuid.UUID`	
`IPAddressField`	`UTF-8`	`str`	
`GenericIPAddressField`	`UTF-8`	`str`	
`BooleanField`	`Bool`	`boolean`	
`EmailField`	`UTF-8`	`str`	

Useful Links

- Examples on GitHub
- PyPI Package

Dialect YDB for Kotlin Exposed

Introduction

This guide describes using JetBrains Exposed with YDB.

Exposed is a Kotlin library for working with a database via SQL DSL and DAO API on top of JDBC. The YDB dialect adds support for YDB-compatible SQL, YDB data type mapping, retry-aware transactions for optimistic concurrency control, and YDB-specific schema features.

The guide covers two practical scenarios:

- Main JDBC dialect for Exposed
- Example of custom configuration for Spring Boot 3 applications.

Installing the YDB dialect

Examples for various build systems:

Maven

```
<!-- Укажите актуальные версии -->
<dependency>
    <groupId>tech.ydb.jdbc</groupId>
    <artifactId>ydb-jdbc-driver</artifactId>
    <version>${ydb.jdbc.version}</version>
</dependency>

<dependency>
    <groupId>tech.ydb.dialects</groupId>
    <artifactId>kotlin-exposed-ydb-dialect</artifactId>
    <version>${exposed.ydb.dialect.version}</version>
</dependency>
```

Gradle

```
dependencies {
    // Specify the current versions
    implementation "tech.ydb.jdbc:ydb-jdbc-driver:$ydbJdbcVersion"
    implementation "tech.ydb.dialects:kotlin-exposed-ydb-dialect:$ydbDialectVersion"
}
```

Current versions of artifacts:

- `tech.ydb.dialects:kotlin-exposed-ydb-dialect` on Maven Central
- `tech.ydb.jdbc:ydb-jdbc-driver` — same as above.

If the application uses DAO, JSON DSL, or other Exposed modules, connect them in the standard way together with the main dialect.

Connection and configuration

First, register the dialect and driver, then open `Database` through a regular `Database.connect(...)` and pass the recommended values `DatabaseConfig`:

```
import org.jetbrains.exposed.v1.core.DatabaseConfig
import org.jetbrains.exposed.v1.jdbc.Database
import tech.ydb.exposed.dialect.registerYdbDialect
import java.sql.Connection

registerYdbDialect()

val db = Database.connect(
    url = "jdbc:ydb:grpc://localhost:2136/local",
    driver = "tech.ydb.jdbc.YdbDriver",
    databaseConfig = DatabaseConfig {
        useNestedTransactions = false
    }
)
```

`registerYdbDialect()`:

- Registers the JDBC driver `tech.ydb.jdbc.YdbDriver` for URLs with the prefix `jdbc:ydb:`.
- registers `YdbDialect` in Exposed
- registers a metadata bridge for reading existing secondary indexes via JDBC metadata.

Recommended Exposed configuration for YDB includes:

- `useNestedTransactions = false` — a setting on the Exposed side that prevents Exposed from emulating nested transactions. YDB does not support nested transactions, so such emulation would lead to errors.
- registration of `YdbDialect` through `registerYdbDialect(...)` before opening `Database`.

DDL features in YDB

Basic usage of Exposed — defining tables, DSL/DAO queries, transactions, etc. — is described in the official JetBrains Exposed documentation. This section only covers YDB-specific features of DDL dialect generation.

Creating tables

In YDB `CREATE TABLE` must contain a table `PRIMARY KEY (...)`. The standard Exposed 1.3.0 DDL path for tables with a single-column PK can generate an inline form:

```
id Int32 PRIMARY KEY
```

YDB does not accept this form. Therefore, for tables that should be created via Exposed DDL, you need to override `createStatement()` and call `createYdbStatement()`:

```
import org.jetbrains.exposed.v1.core.PrimaryKey
import org.jetbrains.exposed.v1.core.Table
import tech.ydb.exposed.dialect.createYdbStatement
import tech.ydb.exposed.dialect.javatime.ydbTimestamp64
import tech.ydb.exposed.dialect.ydbDecimal

object Products : Table("products") {
    val id = integer("id")
    val sku = varchar("sku", 64)
    val name = varchar("name", 255)
    val category = varchar("category", 128)
    val price = ydbDecimal("price", precision = 10, scale = 2)
    val expiresAt = ydbTimestamp64("expires_at")

    override val primaryKey = PrimaryKey(id)

    init {
        index(false, sku)
    }

    override fun createStatement(): List<String> = createYdbStatement()
}
```

`createYdbStatement()`:

- renders all columns without inline `PRIMARY KEY`
- Adds a table `PRIMARY KEY (...)`
- saves `NOT NULL` and `DEFAULT`
- preserves `storageParameters`, therefore YDB-specific `WITH (...)` can be described through the standard Exposed API.

Schema creation is performed using standard Exposed tools:

```
import org.jetbrains.exposed.v1.jdbc.SchemaUtils
import tech.ydb.exposed.dialect.ydbTransaction

ydbTransaction(db) {
    SchemaUtils.create(Products)
}
```

YDB-specific table parameters

TTL and other YDB-specific table options are set via `storageParameters`:

```
import org.jetbrains.exposed.v1.core.PrimaryKey
import org.jetbrains.exposed.v1.core.RawTableStorageParameter
import org.jetbrains.exposed.v1.core.Table
import org.jetbrains.exposed.v1.core.TableStorageParameter
import tech.ydb.exposed.dialect.createYdbStatement
import tech.ydb.exposed.dialect.javatime.ydbTimestamp64

object Sessions : Table("sessions") {
    val id = integer("id")
    val expireAt = ydbTimestamp64("expire_at")

    override val primaryKey = PrimaryKey(id)

    override val storageParameters: List<TableStorageParameter> =
        listOf(RawTableStorageParameter("TTL = Interval(\"PT1H\") ON expire_at"))

    override fun createStatement(): List<String> = createYdbStatement()
}
```

Secondary indexes declared via the standard `Table.index(...)` are created separately via:

```
ALTER TABLE ... ADD INDEX ... GLOBAL
```

Arbitrary YQL via `createYdbStatement()`

`createYdbStatement()` returns `List<String>`, so you can add arbitrary YQL statements to the generated DDL — for example, `CREATE TABLE` of a columnar table, `ALTER TABLE ADD INDEX ... LOCAL`, or any other DDL that the dialect does not cover first-class.

```
import org.jetbrains.exposed.v1.core.PrimaryKey
import org.jetbrains.exposed.v1.core.Table
```

```
import tech.ydb.exposed.dialect.createYdbStatement

object EventsColumn : Table("events_column") {
    val id = integer("id")
    val name = varchar("name", 255)

    override val primaryKey = PrimaryKey(id)

    override fun createStatement(): List<String> = createYdbStatement() + listOf(
        """
        CREATE TABLE events_column_olap (
            id Int32 NOT NULL,
            payload Json,
            PRIMARY KEY (id)
        ) WITH (STORE = COLUMN)
        """.trimIndent()
    )
}
```

This is a convenient way to describe tables for which the Exposed DSL does not cover the required YDB syntax. To manage the schema as a whole, it is recommended to use Flyway or Liquibase.

Read and write operations

Standard Exposed DSL operations (`select` , `insert` , `update` , `deleteWhere`) work without additional modifications.

The dialect also uses native operations YDB `UPSERT` and `REPLACE` through extensions `Table.upsert(...)` , `Table.batchUpsert(...)` , and `Table.replace(...)` :

```
import java.math.BigDecimal

Products.upsert {
    it[id] = 1
    it[sku] = "BOOK-001"
    it[name] = "Kotlin in Action"
    it[category] = "books"
    it[price] = BigDecimal("39.90")
}

Products.replace {
    it[id] = 1
    it[sku] = "BOOK-001"
    it[name] = "Kotlin in Action, 2nd edition"
    it[category] = "books"
    it[price] = BigDecimal("44.90")
}
```

The semantics correspond to the YQL statements `UPSERT INTO` and `REPLACE INTO` .
Exposed-specific parameters `onUpdate` and `keyColumns` for `upsert(...)` are ignored, and `upsert(where)` and PostgreSQL-like logic `ON CONFLICT DO UPDATE` are not supported.

> **Warning**
>
> The ANSI `MERGE` dialect is intentionally not supported. In YDB scenarios, its role is performed by `UPSERT` and `REPLACE` .

Repeatable transactions

YDB uses optimistic concurrency control, so a transaction may end with a retryable error and require re-execution. For this, the dialect provides `ydbTransaction(...)` :

```
import tech.ydb.exposed.dialect.YdbRetryConfig
import tech.ydb.exposed.dialect.ydbTransaction

ydbTransaction(db) {
    // regular read-write transaction
}

ydbTransaction(db, retry = YdbRetryConfig.IDEMPOTENT) {
    // safely repeatable operation
}

ydbTransaction(db, readOnly = true, retry = YdbRetryConfig.IDEMPOTENT) {
    // read-only scenario
}
```

`YdbRetryConfig.IDEMPOTENT` should only be used when the transaction body can be safely re-executed.

Data types

Table mapping standard Exposed types to YDB types:

Exposed	YDB
`byte` / `ubyte`	`Int8` / `Uint8`
`short` / `ushort`	`Int16` / `Uint16`

`integer` / `uinteger`	`Int32` / `Uint32`
`long`	`Int64`
`float` / `double`	`Float` / `Double`
`bool`	`Bool`
`varchar` / `text`	`Text`
`binary` / `blob`	`Bytes`
`uuid`	`Uuid`
`date`	`Date` or `Date32`
`datetime`	`Datetime` or `Datetime64`
`timestamp`	`Timestamp` or `Timestamp64`
`json`	`Json`
`jsonb`	`JsonDocument`

Features:

- `varchar(n)` maps to `Text`; the length constraint is not encoded in YDB DDL;
- `jsonb` corresponds to `JsonDocument` — this is a binary JSON representation optimized for efficient retrieval via `JSON_VALUE` / `JSON_EXISTS` / `JSON_QUERY`. `JsonDocument` itself cannot be used as a B-tree index key (in YDB, JSON is not a comparable type);
- `autoIncrement()` maps to `Serial` or `BigSerial`.

YDB-specific types

The dialect adds a set of extensions for columns:

```
import tech.ydb.exposed.dialect.ydbDecimal
import tech.ydb.exposed.dialect.ydbInterval
import tech.ydb.exposed.dialect.ydbInterval64
import tech.ydb.exposed.dialect.ydbJson
import tech.ydb.exposed.dialect.ydbJsonDocument
import tech.ydb.exposed.dialect.ydbUbyte
import tech.ydb.exposed.dialect.ydbUint32
import tech.ydb.exposed.dialect.ydbUint64
import tech.ydb.exposed.dialect.ydbUshort
import tech.ydb.exposed.dialect.ydbUlong
import tech.ydb.exposed.dialect.ydbUuid

val amount = ydbDecimal("amount", precision = 22, scale = 9)
val payload = ydbJson("payload")
val indexedPayload = ydbJsonDocument("indexed_payload")
val uid = ydbUuid("uid")
val flags = ydbUbyte("flags")
val counter = ydbUint32("counter")
val total = ydbUint64("total")
val someField = ydbUshort("some_field")
val totalFullRange = ydbUlong("total_full_range")
val duration = ydbInterval("duration")
val duration64 = ydbInterval64("duration64")
```

Mapping of YDB types and Kotlin runtime types:

Function	YDB type	Kotlin type
`ydbDecimal(name, p, s)`	`Decimal(p, s)`	`java.math.BigDecimal`
`ydbInterval(name)`	`Interval`	`java.time.Duration`
`ydbInterval64(name)`	`Interval64`	`java.time.Duration`
`ydbJson(name)`	`Json`	`String`
`ydbJsonDocument(name)`	`JsonDocument`	`String`
`ydbUuid(name)`	`Uuid`	`java.util.UUID`
`ydbUbyte(name)`	`Uint8`	`kotlin.UByte`
`ydbUshort(name)`	`Uint16`	`kotlin.UShort`
`ydbUint32(name)`	`Uint32`	`kotlin.UInt`
`ydbUint64(name)`	`Uint64`	`kotlin.Long` (`0..Long.MAX_VALUE`)
`ydbUlong(name)`	`Uint64`	`kotlin.ULong`

For literal values of `Decimal` in update expressions, `ydbDecimalLiteral(...)` is available.

Explicit selection of temporal type at the column level

```
import org.jetbrains.exposed.v1.core.PrimaryKey
import org.jetbrains.exposed.v1.core.Table
import tech.ydb.exposed.dialect.createYdbStatement
import tech.ydb.exposed.dialect.javatime.ydbDate
import tech.ydb.exposed.dialect.javatime.ydbDate32
import tech.ydb.exposed.dialect.javatime.ydbDatetime
import tech.ydb.exposed.dialect.javatime.ydbDatetime64
import tech.ydb.exposed.dialect.javatime.ydbTimestamp
import tech.ydb.exposed.dialect.javatime.ydbTimestamp64

object Events : Table("events") {
    val id = integer("id")
    val legacyDate = ydbDate("legacy_date")
    val signedDate = ydbDate32("signed_date")
    val legacyDatetime = ydbDatetime("legacy_datetime")
    val signedDatetime = ydbDatetime64("signed_datetime")
    val legacyTimestamp = ydbTimestamp("legacy_timestamp")
    val signedTimestamp = ydbTimestamp64("signed_timestamp")

    override val primaryKey = PrimaryKey(id)
    override fun createStatement(): List<String> = createYdbStatement()
}
```

Note

`registerYdbDialect(enableSignedDatetimes = true)` changes DDL names only for standard Exposed `date`, `datetime`, and `timestamp`. Explicit `ydbDate(...)`, `ydbDate32(...)`, `ydbDatetime(...)`, `ydbDatetime64(...)`, `ydbTimestamp(...)`, and `ydbTimestamp64(...)` always generate the type reflected in the function name.

Warning

`ydbUint64(...)` stores a value in Kotlin `Long`, so only the `0..Long.MAX_VALUE` range is supported. For values above `Long.MAX_VALUE`, use `ydbUlong(...)` — it is represented by the Kotlin type `ULong` and covers the entire YQL `Uint64` range.

Integration with Spring Boot 3

For Spring Boot 3, it is recommended to use the official Exposed scenario from the Spring Boot 3 integration documentation and add a small YDB-specific configuration to the application.

The logic of the previously existing starter module can serve as a reference for such manual configuration: in the application, you need to explicitly register the dialect, explicitly specify the YDB JDBC driver, match `forceSignedDatetimes=...` in the URL, and create `Database` from Spring-managed `DataSource`.

Dependencies

Maven

```
<!-- Укажите актуальную версию -->
<dependency>
    <groupId>org.jetbrains.exposed</groupId>
    <artifactId>exposed-spring-boot-starter</artifactId>
    <version>${exposed.version}</version>
</dependency>
<dependency>
    <groupId>tech.ydb.dialects</groupId>
    <artifactId>kotlin-exposed-ydb-dialect</artifactId>
    <version>${exposed.ydb.dialect.version}</version>
</dependency>
<dependency>
    <groupId>tech.ydb.jdbc</groupId>
    <artifactId>ydb-jdbc-driver</artifactId>
    <version>${ydb.jdbc.version}</version>
</dependency>
```

Gradle

```
dependencies {
    // Specify the current version
    implementation "org.jetbrains.exposed:exposed-spring-boot-starter:$exposedVersion"
    implementation "tech.ydb.dialects:kotlin-exposed-ydb-dialect:$ydbDialectVersion"
    implementation "tech.ydb.jdbc:ydb-jdbc-driver:$ydbJdbcVersion"
}
```

Basic Spring Boot configuration

```
spring:
  datasource:
    url: jdbc:ydb:grpc://localhost:2136/local?forceSignedDatetimes=false
    driver-class-name: tech.ydb.jdbc.YdbDriver
  exposed:
```

```
    generate-ddl: false
    show-sql: false
    ydb:
      enable-signed-datetimes: false
```

The standard properties of the official Exposed starter remain available, including:

- `spring.exposed.generate-ddl`
- `spring.exposed.excluded-packages`
- `spring.exposed.show-sql`

The example above uses a YDB-specific property:

- `spring.exposed.ydb.enable-signed-datetimes`

In this variant, the property name `spring.exposed.ydb.enable-signed-datetimes` is chosen by the application itself by convention with the rest of the configuration. It can be replaced with any other name if more convenient. If signed mode is enabled, the value in this property must be aligned with the JDBC URL, i.e., use `forceSignedDatetimes=true`.

Custom configuration example

Below is a minimal example of `@Configuration` that adds YDB-specific steps to the standard Spring Boot Exposed integration:

- Calls `registerYdbDialect(...)`.
- Sets the recommended values of `DatabaseConfig`.
- Creates a bean `Database` from Spring-managed `DataSource`.

```
import org.jetbrains.exposed.v1.core.DatabaseConfig
import org.jetbrains.exposed.v1.jdbc.Database
import org.springframework.beans.factory.InitializingBean
import org.springframework.beans.factory.annotation.Value
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.DependsOn
import tech.ydb.exposed.dialect.registerYdbDialect
import java.sql.Connection
import javax.sql.DataSource

@Configuration
class YdbExposedConfiguration(
    @Value("\${spring.exposed.ydb.enable-signed-datetimes:false}")
    private val enableSignedDatetimes: Boolean
) {
    @Bean
    fun ydbDialectRegistration(): InitializingBean = InitializingBean {
        registerYdbDialect(enableSignedDatetimes = enableSignedDatetimes)
    }

    @Bean
    fun databaseConfig(): DatabaseConfig = DatabaseConfig {
        defaultIsolationLevel = Connection.TRANSACTION_SERIALIZABLE
        defaultReadOnly = false
        useNestedTransactions = false
    }

    @Bean
    @DependsOn("ydbDialectRegistration")
    fun database(dataSource: DataSource, databaseConfig: DatabaseConfig): Database =
        Database.connect(
            datasource = dataSource,
            databaseConfig = databaseConfig
        )
}
```

If the application already has its own logic for creating `DataSource` or `Database`, the same steps can be moved into it. The key YDB-specific actions here remain the same: register the dialect, manually align `forceSignedDatetimes` in the JDBC URL, explicitly specify `tech.ydb.jdbc.YdbDriver`, and pass the recommended `DatabaseConfig` to Exposed.

Thus, in a typical case, it is sufficient to:

1. Add `exposed-spring-boot-starter`, `kotlin-exposed-ydb-dialect`, and `ydb-jdbc-driver`.
2. Specify `spring.datasource.url` starting with `jdbc:ydb:` and containing `forceSignedDatetimes=...`.
3. Add a custom `@Configuration` similar to the example above.
4. Enable `spring.exposed.ydb.enable-signed-datetimes=true` if necessary.

That is, the YDB integration for Spring Boot is described here as an explicit application configuration, not as a separate starter artifact.

Automatic schema generation

`spring.exposed.generate-ddl=true` works the same as in the official Exposed starter: on application startup, Exposed creates a schema based on the discovered `Table` classes.

For YDB, there is an important nuance here: tables that should be created via DDL must be declared using a regular `Table` with an override:

```
override fun createStatement(): List<String> = createYdbStatement()
```

> **Warning**
>
> If you enable `spring.exposed.generate-ddl=true` and leave tables on plain `Table` or `IdTable` without this override, Exposed will generate an inline `PRIMARY KEY` that YDB does not accept.

Transactions in Spring

For standard Spring integration, use `@Transactional`:

```
import org.springframework.stereotype.Service
import org.springframework.transaction.annotation.Transactional

@Service
class ProductService {
    @Transactional
    fun renameProduct(id: Int, newName: String) {
        // Exposed DSL
    }
}
```

`@Transactional` provides the usual Spring Exposed integration, but does not add a YDB retry policy for optimistic concurrency conflicts.

If a retry-aware path with automatic retries on YDB retryable errors is required, in your Spring service you can call `ydbTransaction(...)` directly, passing the bean `Database`:

```
import org.springframework.stereotype.Service
import org.jetbrains.exposed.v1.jdbc.Database
import tech.ydb.exposed.dialect.YdbRetryConfig
import tech.ydb.exposed.dialect.ydbTransaction

@Service
class RetryableProductService(
    private val database: Database
) {
    fun saveOrUpdate() =
        ydbTransaction(database, retry = YdbRetryConfig.IDEMPOTENT) {
            // Exposed DSL
        }
}
```

Typically, for one piece of code, one of two paths is chosen:

- Either `@Transactional`.
- Or `ydbTransaction(database, ...)` if automatic retries are needed.

Schema management

For production scenarios, it is recommended to consider Exposed DDL generation as a supplementary, not primary, path. Typically, the schema:

1. Is described by separate SQL scripts.
2. Is migrated by an external versioning tool — for example, Flyway or Liquibase.
3. Is reconciled with the Exposed model on the application side.

If the application uses schema validation or migration diff generation via Exposed, add:

Maven

```
<dependency>
    <groupId>org.jetbrains.exposed</groupId>
    <artifactId>exposed-migration-core</artifactId>
    <version>${exposed.version}</version>
</dependency>
<dependency>
    <groupId>org.jetbrains.exposed</groupId>
    <artifactId>exposed-migration-jdbc</artifactId>
    <version>${exposed.version}</version>
</dependency>
```

Gradle

```
dependencies {
    implementation "org.jetbrains.exposed:exposed-migration-core:$exposedVersion"
    implementation "org.jetbrains.exposed:exposed-migration-jdbc:$exposedVersion"
}
```

In Exposed 1.3.0, the full path `MigrationUtils.statementsRequiredForDatabaseMigration(...)` unconditionally reads metadata about `CHECK` constraints from `INFORMATION_SCHEMA.CHECK_CONSTRAINTS`. The current JDBC driver YDB does not provide this part of the metadata, so the full generic Exposed diff for YDB does not work completely.

A practical YDB-compatible path:

- Apply the schema using external migrations.
- Use Exposed table definitions as a description of the client model.

- Validate drift through compatible Exposed metadata paths:
  - Existing columns.
  - Existing secondary indexes.

The dialect repository has integration coverage for a scenario where the schema is created with raw SQL and then reconciled with the Exposed model without failing on unsupported metadata.

> **Note**
>
> If the schema was just changed by raw SQL, perform validation in a new transaction so that Exposed does not work with stale metadata cache.

Signed temporal types in DDL

`enableSignedDatetimes` refers exclusively to schema generation via Exposed DDL: it changes DDL names for the standard Exposed types `date()`, `datetime()`, and `timestamp()`. This flag itself does not change anything for existing tables or for runtime binding of individual columns.

> **Note**
>
> Since Exposed DDL is a test/dev path, the `enableSignedDatetimes` flag also primarily relates to tests and local development. In production, temporal types are fixed in versioned SQL migrations.

By default (`enableSignedDatetimes = false`), the standard Exposed temporal types appear in the generated `CREATE TABLE` as:

- `Date`
- `Datetime`
- `Timestamp`

If signed mode is needed in Exposed DDL, configure it in two places: enable `enableSignedDatetimes` on the dialect (this changes the DDL of standard temporal types) and add `forceSignedDatetimes=true` to the JDBC URL (this changes the binding on the driver side):

```
registerYdbDialect(enableSignedDatetimes = true)

val db = Database.connect(
    url = "jdbc:ydb:grpc://localhost:2136/local?forceSignedDatetimes=true",
    driver = "tech.ydb.jdbc.YdbDriver",
    databaseConfig = DatabaseConfig {
        defaultIsolationLevel = Connection.TRANSACTION_SERIALIZABLE
        defaultReadOnly = false
        useNestedTransactions = false
    }
)
```

In this mode, the standard Exposed types `date`, `datetime`, and `timestamp` will appear in the generated `CREATE TABLE` as `Date32`, `Datetime64`, and `Timestamp64`.

If you need explicit control of an individual column type, use the extensions `ydbDate(...)`, `ydbDate32(...)`, `ydbDatetime(...)`, `ydbDatetime64(...)`, `ydbTimestamp(...)`, and `ydbTimestamp64(...)`. They always generate DDL and bind the parameter via JDBC vendor code of exactly the type specified in the function name, regardless of `enableSignedDatetimes`.

If the JDBC URL already has other parameters, the flag must be added via `&forceSignedDatetimes=true`.

Limitations and specifics

- Exposed 1.3.0 does not provide a dialect hook for rendering a single-column `PRIMARY KEY` inside `CREATE TABLE`.
- therefore, the DDL path for YDB is implemented as a workaround via `createStatement()` and `createYdbStatement()`.
- each table created via Exposed DDL in YDB must have `PRIMARY KEY`.
- `SchemaUtils.createMissingTablesAndColumns(...)` and `spring.exposed.generate-ddl=true` require the same override `createStatement()`.
- ANSI `MERGE` is not supported.
- `upsert(where)` and PostgreSQL-like logic `ON CONFLICT DO UPDATE` are not supported.
- functional indexes are not supported.
- `GLOBAL UNIQUE` via `ALTER TABLE ... ADD INDEX ...` depends on the limitations of the specific YDB version.
- `LOCAL` indexes and column-oriented tables (`STORE = COLUMN`) are not supported first-class in the dialect; if necessary, they can be attached via arbitrary YQL in `createYdbStatement()` (test-only) — the dialect is oriented towards row-oriented OLTP tables.
- full generic diff via `MigrationUtils.statementsRequiredForDatabaseMigration(...)` in Exposed 1.3.0 for YDB does not work completely due to unsupported metadata on `CHECK` constraints.

Drizzle ORM

Drizzle ORM is a lightweight TypeScript ORM for Node.js with a type-safe query builder and a schema definition DSL.

YDB supports integration with Drizzle ORM through the `@ydbjs/drizzle-adapter` adapter, which is part of the YDB JavaScript/TypeScript SDK. The adapter provides the familiar Drizzle API and accounts for YDB specifics: primary keys, secondary and vector indexes, TTL, partitioning, and column families.

This page is a brief overview of the integration. The detailed guide covering all features (query builders, relational API, migrations, DDL, vector search) is available at ydb.js.org.

Requirements

- Node.js 20.19 or newer
- The `drizzle-orm` and `@ydbjs/drizzle-adapter` packages
- A running YDB instance

Installation

```
npm install @ydbjs/drizzle-adapter drizzle-orm
```

Connecting

Connection string

The simplest way is to pass a connection string to `createDrizzle()`. The adapter creates and manages the YDB driver itself.

```
import { createDrizzle } from '@ydbjs/drizzle-adapter'

const db = createDrizzle({
  connectionString: process.env['YDB_CONNECTION_STRING']!,
})
```

The connection string has the form `grpc://localhost:2136/local` (or `grpcs://` for a secure connection), specifying the server address, gRPC port, and the database path.

Existing SDK driver

If your application already uses a driver from `@ydbjs/core`, you can wrap it in `YdbDriver` and reuse the shared connection.

```
import { Driver } from '@ydbjs/core'
import { YdbDriver, createDrizzle } from '@ydbjs/drizzle-adapter'

const driver = new Driver('grpc://localhost:2136/local')
await driver.ready()
const db = createDrizzle({
  client: new YdbDriver(driver),
})
```

When the work is done, release the resources if the driver was created by the adapter:

```
db.$client.close?.()
```

Schema definition

Tables are described with `ydbTable()`. A primary key is required for every table.

```
import { integer, text, timestamp, ydbTable } from '@ydbjs/drizzle-adapter/schema'

export const users = ydbTable('users', {
  id: integer('id').primaryKey(),
  email: text('email').notNull().unique(),
  name: text('name'),
  createdAt: timestamp('created_at')
    .notNull()
    .$defaultFn(() => new Date()),
})
```

The adapter supports all the main YDB data types (`Bool`, integer and floating-point types, `Utf8`, `String`, `Uuid`, `Json`, date and time types, and others), as well as YDB-specific schema features: secondary and vector indexes, TTL, partitioning, and column families. The full list of types and options is provided in the schema guide.

CRUD operations

```
import { and, eq, like } from 'drizzle-orm'

// Insert
await db.insert(users).values({ id: 1, email: 'alice@example.com', name: 'Alice' }).execute()

// Select with filtering
const filteredUsers = await db
  .select({ userId: users.id, userName: users.name })
  .from(users)
  .where(and(eq(users.id, 1), like(users.email, '%@example.com')))
  .execute()
```

```
// Update
await db.update(users).set({ name: 'Alice Cooper' }).where(eq(users.id, 1)).execute()

// Delete
await db.delete(users).where(eq(users.id, 1)).execute()
```

The "insert or update" by primary key operation uses `onDuplicateKeyUpdate()`:

```
await db
  .insert(users)
  .values({ id: 1, email: 'alice_new@example.com', name: 'Alice Updated' })
  .onDuplicateKeyUpdate({ set: { name: 'Alice Updated' } })
  .execute()
```

Transactions

Transactions are started via `db.transaction()`. YDB access modes and isolation levels are supported, along with an idempotency flag for automatic retries on network errors.

```
import { eq } from 'drizzle-orm'
import { TransactionRollbackError } from 'drizzle-orm/errors'

try {
  await db.transaction(
    async (tx) => {
      // Check a precondition before modifying data
      const inviter = await tx.select().from(users).where(eq(users.id, 1)).execute()
      if (inviter.length === 0) {
        tx.rollback()
      }

      await tx.insert(users).values({ id: 4, email: 'delta@example.com' }).execute()
    },
    {
      accessMode: 'read write',
      isolationLevel: 'serializableReadWrite',
      idempotent: true,
    }
  )
} catch (error) {
  if (error instanceof TransactionRollbackError) {
    console.log('transaction was rolled back')
  }
}
```

Limitations

Keep in mind the following YDB and adapter specifics:

- The adapter is distributed in ESM format only.
- Nested transactions are not supported.
- Only YDB isolation levels are allowed (`serializableReadWrite`, `snapshotReadOnly`); other levels are not emulated.
- `references()` is stored as metadata for the relational API — YDB does not enforce foreign keys at the database level.

See also

- Detailed adapter guide
- YDB JavaScript/TypeScript SDK on GitHub
- The `@ydbjs/drizzle-adapter` npm package

Vector search

Tool	Guide
LangChain	Guide
Mem0	Guide

LangChain

Integration of YDB with LangChain allows using YDB as a vector store for RAG applications.

This integration allows developers to efficiently manage, query, and retrieve vectorized data, which is the foundation for modern applications related to natural language processing, search, and data analysis. Using embedding models, users can create sophisticated systems capable of understanding and retrieving information based on semantic similarity.

The integration is available for Python and JavaScript.

Installation

To use this integration, install a local YDB. For more information, see Install and start YDB.

Also install the LangChain packages and an embedding model for the required language:

Python

```
pip install -qU langchain-ydb
pip install -qU langchain-huggingface
```

JavaScript

```
npm install @ydbjs/langchain @langchain/core
npm install @langchain/community @huggingface/transformers
```

Initialization

To create a vector store YDB, you need to specify an embedding model and connection parameters:

Python

```
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_ydb.vectorstores import YDB, YDBSearchStrategy, YDBSettings

embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-mpnet-base-v2")

settings = YDBSettings(
    host="localhost",
    port=2136,
    database="/local",
    table="ydb_example",
    strategy=YDBSearchStrategy.COSINE_SIMILARITY,
)
vector_store = YDB(embeddings, config=settings)
```

JavaScript

```
import { HuggingFaceTransformersEmbeddings } from "@langchain/community/embeddings/huggingface_transformers";
import { YDBSearchStrategy, YDBVectorStore } from "@ydbjs/langchain";

const embeddings = new HuggingFaceTransformersEmbeddings({
  model: "sentence-transformers/all-mpnet-base-v2",
});

const vectorStore = new YDBVectorStore(embeddings, {
  connectionString: "grpc://localhost:2136/local",
  table: "ydb_example",
  strategy: YDBSearchStrategy.CosineSimilarity,
});
```

Managing the Vector Store

Once you have created a vector store, you can interact with it by adding and removing various items.

Adding Items

Prepare the documents for processing:

Python

```
from uuid import uuid4

from langchain_core.documents import Document

uuids = [str(uuid4()) for _ in range(10)]
documents = [
    Document(
        page_content="I had chocolate chip pancakes and scrambled eggs for breakfast this morning.",
        metadata={"source": "tweet"},
        id=uuids[0],
    ),
    Document(
        page_content="The weather forecast for tomorrow is cloudy and overcast, with a high of 62 degrees.",
        metadata={"source": "news"},
        id=uuids[1],
    ),
```

```
    Document(
        page_content="Building an exciting new project with LangChain - come check it out!",
        metadata={"source": "tweet"},
        id=uuids[2],
    ),
    Document(
        page_content="Robbers broke into the city bank and stole $1 million in cash.",
        metadata={"source": "news"},
        id=uuids[3],
    ),
    Document(
        page_content="Wow! That was an amazing movie. I can't wait to see it again.",
        metadata={"source": "tweet"},
        id=uuids[4],
    ),
    Document(
        page_content="Is the new iPhone worth the price? Read this review to find out.",
        metadata={"source": "website"},
        id=uuids[5],
    ),
    Document(
        page_content="The top 10 soccer players in the world right now.",
        metadata={"source": "website"},
        id=uuids[6],
    ),
    Document(
        page_content="LangGraph is the best framework for building stateful, agentic applications!",
        metadata={"source": "tweet"},
        id=uuids[7],
    ),
    Document(
        page_content="The stock market is down 500 points today due to fears of a recession.",
        metadata={"source": "news"},
        id=uuids[8],
    ),
    Document(
        page_content="I have a bad feeling I am going to get deleted :(",
        metadata={"source": "tweet"},
        id=uuids[9],
    ),
]
```

Add the documents to the vector store:

```
ids = vector_store.add_documents(documents=documents)
```

JavaScript

```
import { Document } from "@langchain/core/documents";
import { randomUUID } from "node:crypto";

const uuids = Array.from({ length: 10 }, () => randomUUID());
const documents = [
  new Document({
    pageContent: "I had chocolate chip pancakes and scrambled eggs for breakfast this morning.",
    metadata: { source: "tweet" },
    id: uuids[0],
  }),
  new Document({
    pageContent: "The weather forecast for tomorrow is cloudy and overcast, with a high of 62 degrees.",
    metadata: { source: "news" },
    id: uuids[1],
  }),
  new Document({
    pageContent: "Building an exciting new project with LangChain - come check it out!",
    metadata: { source: "tweet" },
    id: uuids[2],
  }),
  new Document({
    pageContent: "Robbers broke into the city bank and stole $1 million in cash.",
    metadata: { source: "news" },
    id: uuids[3],
  }),
  new Document({
    pageContent: "Wow! That was an amazing movie. I can't wait to see it again.",
    metadata: { source: "tweet" },
    id: uuids[4],
  }),
  new Document({
    pageContent: "Is the new iPhone worth the price? Read this review to find out.",
    metadata: { source: "website" },
    id: uuids[5],
  }),
  new Document({
    pageContent: "The top 10 soccer players in the world right now.",
    metadata: { source: "website" },
    id: uuids[6],
  }),
  new Document({
    pageContent: "LangGraph is the best framework for building stateful, agentic applications!",
    metadata: { source: "tweet" },
```

```
    id: uuids[7],
  }),
  new Document({
    pageContent: "The stock market is down 500 points today due to fears of a recession.",
    metadata: { source: "news" },
    id: uuids[8],
  }),
  new Document({
    pageContent: "I have a bad feeling I am going to get deleted :(",
    metadata: { source: "tweet" },
    id: uuids[9],
  }),
];
```

Add the documents to the vector store:

```
const ids = await vectorStore.addDocuments(documents);
```

Deleting Items

Items are deleted from the vector store by ID using the `delete` function:

Python

```
vector_store.delete(ids=[ids[-1]])
```

JavaScript

```
await vectorStore.delete({ ids: [ids.at(-1)] });
```

Querying the Vector Store

After creating the vector store and adding the necessary documents, you can perform search queries during chain or agent execution.

Direct Query

Similarity Search

A simple similarity search can be performed as follows:

Python

```
results = vector_store.similarity_search(
    "LangChain provides abstractions to make working with LLMs easy",
    k=2,
)
for res in results:
    print(f"* {res.page_content} [{res.metadata}]")
```

JavaScript

```
const results = await vectorStore.similaritySearch(
  "LangChain provides abstractions to make working with LLMs easy",
  2,
);
for (const res of results) {
  console.log(`* ${res.pageContent} [${JSON.stringify(res.metadata)}]`);
}
```

Result:

```
* Building an exciting new project with LangChain - come check it out! [{'source': 'tweet'}]
* LangGraph is the best framework for building stateful, agentic applications! [{'source': 'tweet'}]
```

Similarity Search with Score

You can also perform a search with a score:

Python

```
results = vector_store.similarity_search_with_score(
    "Will it be hot tomorrow?",
    k=3,
)
for res, score in results:
    print(f"* [SIM={score:.3f}] {res.page_content} [{res.metadata}]")
```

JavaScript

```
const results = await vectorStore.similaritySearchWithScore(
  "Will it be hot tomorrow?",
  3,
);
```

```
for (const [res, score] of results) {
  console.log(`* [SIM=${score.toFixed(3)}] ${res.pageContent} [${JSON.stringify(res.metadata)}]`);
}
```

Result:

```
* [SIM=0.595] The weather forecast for tomorrow is cloudy and overcast, with a high of 62 degrees. [{'sourc
e': 'news'}]
* [SIM=0.212] I had chocolate chip pancakes and scrambled eggs for breakfast this morning. [{'source': 'twee
t'}]
* [SIM=0.118] Wow! That was an amazing movie. I can't wait to see it again. [{'source': 'tweet'}]
```

Filtering

Search using filters is performed as follows:

Python

```
results = vector_store.similarity_search_with_score(
    "What did I eat for breakfast?",
    k=4,
    filter={"source": "tweet"},
)
for res, _ in results:
    print(f"* {res.page_content} [{res.metadata}]")
```

JavaScript

```
const results = await vectorStore.similaritySearchWithScore(
  "What did I eat for breakfast?",
  4,
  { source: "tweet" },
);
for (const [res] of results) {
  console.log(`* ${res.pageContent} [${JSON.stringify(res.metadata)}]`);
}
```

Result:

```
* I had chocolate chip pancakes and scrambled eggs for breakfast this morning. [{'source': 'tweet'}]
* Wow! That was an amazing movie. I can't wait to see it again. [{'source': 'tweet'}]
* Building an exciting new project with LangChain - come check it out! [{'source': 'tweet'}]
* LangGraph is the best framework for building stateful, agentic applications! [{'source': 'tweet'}]
```

Query via Retriever Transformation

The vector store can be transformed into a retriever for simplified use in chains.

An example is shown below:

Python

```
retriever = vector_store.as_retriever(
    search_kwargs={
        "k": 2,
        "filter": {"source": "news"},
    },
)
results = retriever.invoke("Stealing from the bank is a crime")
for res in results:
    print(f"* {res.page_content} [{res.metadata}]")
```

JavaScript

```
const retriever = vectorStore.asRetriever({
  k: 2,
  filter: { source: "news" },
});
const results = await retriever.invoke("Stealing from the bank is a crime");
for (const res of results) {
  console.log(`* ${res.pageContent} [${JSON.stringify(res.metadata)}]`);
}
```

Result:

```
* Robbers broke into the city bank and stole $1 million in cash. [{'source': 'news'}]
* The stock market is down 500 points today due to fears of a recession. [{'source': 'news'}]
```

Mem0

Mem0 is a long-term memory layer for AI agents and LLM applications. Mem0 extracts meaningful facts about the user from conversations, stores them as embeddings, and on subsequent queries finds relevant memories by semantic similarity. This allows the agent to "remember" the user's preferences and context across sessions.

Integration with YDB allows using YDB as a vector store for Mem0 memory — without a separate specialized vector database. The integration works on top of LangChain: Mem0 supports LangChain vector stores as a backend, and YDB provides a compatible store via the `langchain-ydb` package.

> **Note**
>
> In addition to the vector store, Mem0 uses a language model (LLM) to extract facts from the conversation and an embedding model to vectorize them. The examples below use OpenAI models (requires the `OPENAI_API_KEY` environment variable), but any Mem0-supported provider will work.

Installation

To use this integration, install a local YDB. For more information, see Install and start YDB.

Install Mem0 and the YDB integration package with LangChain:

```
pip install -qU mem0ai langchain-ydb langchain-openai
```

Initialization

Create a YDB vector store using `langchain-ydb` and pass it to the Mem0 configuration with the `langchain` provider:

The key to OpenAI models is expected in the `OPENAI_API_KEY` environment variable — set it in advance in your environment or secret manager.

```
from langchain_openai import OpenAIEmbeddings
from langchain_ydb.vectorstores import YDB, YDBSearchStrategy, YDBSettings
from mem0 import Memory

embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

vector_store = YDB(
    embeddings,
    config=YDBSettings(
        host="localhost",
        port=2136,
        database="/local",
        table="mem0_memories",
        strategy=YDBSearchStrategy.COSINE_SIMILARITY,
    ),
)

memory = Memory.from_config({
    "vector_store": {
        "provider": "langchain",
        "config": {"client": vector_store},
    },
    "embedder": {
        "provider": "langchain",
        "config": {"model": embeddings},
    },
})
```

Saving facts

Pass the conversation messages to Mem0 — the model will extract facts from them and save them to the YDB table. Memory is shared between users using `user_id`:

```
memory.add(
    [
        {"role": "user", "content": "Привет! Меня зовут Алиса, я работаю архитектором."},
        {"role": "assistant", "content": "Рада знакомству, Алиса!"},
        {"role": "user", "content": "Предпочитаю примеры на Python."},
    ],
    user_id="alice",
)
```

Memory search

Before forming a response, the agent searches for relevant memories — Mem0 performs a vector search in YDB and returns facts ordered by similarity:

```
results = memory.search("На каком языке показывать примеры?", user_id="alice")
for item in results["results"]:
    print(f"{item['score']:.3f}  {item['memory']}")
```

Result:

```
0.512  Prefers examples in Python
0.231  Works as an architect
```

Viewing and deletion

User memories can be retrieved or deleted entirely:

```python
# all user memories
for item in memory.get_all(user_id="alice")["results"]:
    print(item["memory"])

# delete all user memories
memory.delete_all(user_id="alice")
```

SQL Dialect Converter to YQL

The SQL Dialect Converter is a service that transforms SQL queries written in other dialects (PostgreSQL, MySQL, ClickHouse, and others) into YQL syntax. The service is designed to simplify the migration of existing queries to YDB and lower the entry barrier for developers already familiar with other DBMS.

The converter is used in the DBeaver plugin for YDB on the **Convert Dialect** tab. The user pastes a query in the source dialect, selects the dialect from the list, and receives a version of the same query in YQL.

Principle of Operation

The transformation is performed in three stages:

1. **Parsing the Source SQL.** The source query is parsed by a dialect-specific parser into an abstract syntax tree (AST) — a structure that describes the meaning of the query independently of the syntax.
2. **AST Transformation.** Constructions specific to the source dialect are replaced with their equivalents in YQL. For example, type casting functions, work with dates and strings, `LIMIT` / `OFFSET` operators, and `WITH` constructions are adapted to YQL rules.
3. **YQL Generation.** A textual representation of the query in YQL is generated from the modified AST.

The stages are performed using the SQLGlot library, which includes a separate dialect module for YDB. SQLGlot is an open-source SQL parser and transpiler that supports over twenty dialects. It acts as both the parser for the source query and the generator for YQL.

> **Warning**
>
> The source query is sent to an external HTTPS service. Do not send queries containing confidential data (personal data, trade secrets, identifiers of real objects in production) to the converter.

Supported Dialects

The list of source dialects is determined by the set supported by SQLGlot. Among the main ones:

- PostgreSQL;
- MySQL;
- ClickHouse;
- Microsoft SQL Server (T-SQL);
- Oracle;
- Snowflake;
- BigQuery;
- Presto / Trino;
- Spark SQL / Databricks;
- SQLite.

The up-to-date full list of dialects can be obtained via the converter's API or viewed in the SQLGlot sources. The target dialect is always YQL.

Supported Constructions

The converter covers typical constructions of analytical and OLTP queries:

- Selection operators: `SELECT`, `JOIN` (all types), `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT`, `OFFSET`.
- Window functions (`OVER`, `PARTITION BY`, `ROWS BETWEEN`).
- Subqueries and CTE (`WITH`).
- Type casting (`CAST`, `::`), arithmetic and logical operations.
- Functions for strings, numbers, dates, and times — mapped to the closest equivalent in YQL.
- Aggregate functions (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`, and others).
- DML operations: `INSERT`, `UPDATE`, `DELETE` (taking into account YDB transaction features).

Limitations

Full automatic conversion is not always possible, as YQL and other SQL dialects differ in data models and operation semantics. In particular:

- **Dialect-specific functions.** Functions that do not have a direct equivalent in YQL (for example, PostgreSQL arrays, MySQL `JSON_EXTRACT` with special path syntax) are translated into approximate equivalents or left for manual refinement.
- **Stored procedures and triggers.** Not supported, as YDB uses a different code execution model in the database.
- **Specific DDL.** `CREATE TABLE` for complex types may require manual adjustments taking into account columnar and row tables in YDB.
- **Optimizer hints.** Ignored: the YDB optimizer has its own control mechanisms.

The conversion result should be considered a draft: complex queries require manual verification and adaptation before execution.

See Also

- DBeaver Plugin for YDB
- YQL Reference
- SQLGlot

YDB CLI

The YDB CLI provides software for managing your data in YDB.

To use the YDB CLI, first install it and then set up the connection and authentication.

For a full description of YDB CLI commands, see the following articles of this section:

- List of objects.
- Getting information about schema objects.
- Working with directories.
- Query execution.
- Streaming table reads.
- Working with secondary indexes.
- Getting a list of DB endpoints.
- Load testing.

YDB CLI

The YDB CLI provides software for managing your data in YDB.

To use the YDB CLI, first install it and then set up the connection and authentication.

For a full description of YDB CLI commands, see the following articles of this section:

- List of objects.
- Getting information about schema objects.
- Working with directories.
- Query execution.
- Streaming table reads.
- Working with secondary indexes.
- Getting a list of DB endpoints.
- Load testing.

Installing the YDB CLI

Linux

To install the YDB CLI, run the command:

```
curl -sSL https://install.ydb.tech/cli | bash
```

The script will install the YDB CLI and add the executable file path to the `PATH` environment variable. It will also generate shell completion files and print instructions on how to enable tab completion for commands and options.

> **Note**
>
> The script will update the `PATH` variable only if you run it in the bash or zsh command shell. If you run the script in a different shell, add the path to the CLI to the `PATH` variable yourself.
>
> Shell completion for bash requires the `bash-completion` package to be installed.

> **Tip**
>
> If you use a non-standard shell configuration, you can source the generated completion files from any rc file. The files are located at `~/.local/share/ydb/completion.bash.inc` and `~/.local/share/ydb/completion.zsh.inc`, and are kept up to date automatically by `ydb update`.

To update the environment variables, restart the command shell.

macOS

To install the YDB CLI, run the command:

```
curl -sSL https://install.ydb.tech/cli | bash
```

The script will install the YDB CLI and add the executable file path to the `PATH` environment variable. It will also generate shell completion files and print instructions on how to enable tab completion for commands and options.

To update the environment variables, restart the command shell.

Windows

You can install the YDB CLI using:

PowerShell. To do this, run the command:

```
iex (New-Object System.Net.WebClient).DownloadString('https://install.ydb.tech/cli-windows')
```

Specify whether to add the executable file path to the `PATH` environment variable:

```
Add ydb installation dir to your PATH? [Y/n]
```

cmd. To do this, run the command:

```
@"%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe" -Command "iex ((New-Object System.Net.WebClien
t).DownloadString('https://install.ydb.tech/cli-windows'))"
```

Specify whether to add the executable file path to the `PATH` environment variable:

```
Add ydb installation dir to your PATH? [Y/n]
```

To update the environment variables, restart the command shell.

> **Note**
>
> The YDB CLI uses Unicode characters in the output of some commands. If these characters aren't displayed correctly in the Windows console, switch the encoding to UTF-8:
>
> ```
> chcp 65001
> ```

Connecting the CLI to and authenticating with a database

Most of the YDB CLI commands relate to operations on a YDB database and require establishing a connection to it to be executed.

The YDB CLI uses the following sources to determine the database to connect to and the authentication mode to use with it (listed in descending priority):

1. The command line.
2. The profile set in the `--profile` command-line option.
3. Environment variables.
4. The activated profile.

For the YDB CLI to try connecting to the database, these steps must result in the endpoint and database path.

If all the steps are completed, but the YDB CLI did not determine the authentication mode, requests will be sent to the YDB server without adding authentication data. This may let you successfully work with locally deployed YDB clusters that require no authentication. For all databases available over the network, such requests will be rejected by the server with an authentication error returned.

To learn about potential situations where the YDB CLI won't try to connect to the database, see the Error messages below.

Command line parameters

DB connection options in the command line are specified before defining the command and its parameters:

```
ydb <connection_options> <command> <command_options>
```

- `-e, --endpoint <endpoint>` is the endpoint, that is, the main connection parameter that allows finding a YDB server on the network. If no port is specified, port 2135 is used. If no protocol is specified, gRPCs (with encryption) is used in YDB CLI public builds.
- `-d, --database <database>` is the database path.
- `--no-discovery` means do not perform discovery (client balancing) for ydb cluster connection. If this option is set the user provided endpoint (by `-e` option) will be used to setup a connections.

The authentication mode and parameters are selected by setting one of the following options:

- `--token-file <filepath>`: Enables the **Access Token** authentication mode using the contents of the file specified in this option.
- `--yc-token-file <filepath>`: Enables the **Refresh Token** authentication mode using the contents of the file specified in this option.
- `--use-metadata-credentials` : The **Metadata** authentication mode is used.
- `--sa-key-file <filepath>`: Enables the **Service Account Key** authentication mode, where the key and other parameters are taken from the JSON file specified in this option.
- `--user <username>` : The username and password based authentication mode is used with the username set in this option value. Additionally , you can specify:
  - `--password-file <filename>` : The password is read from the specified file.
  - `--no-password` : Defines an empty password. The password will be requested interactively if none of the password identification options listed above are specified in the command line parameters.
- `--oauth2-key-file <filepath>` : Enables the **OAuth 2.0 token exchange** authentication mode, where the key and other parameters are taken from the JSON file specified in this option. The `--iam-endpoint` option sets the token exchange endpoint in the `<schema>://<host>:<port>/<path>` format (via the YDB CLI option, profile, or environment variable).

If several of the above options are set simultaneously in the command line, the CLI returns an error asking you to specify only one:

```
$ ydb --use-metadata-credentials --iam-token-file ~/.ydb/token scheme ls
More than one auth method were provided via options. Choose exactly one of them
Try "--help" option for more info.
```

When using authentication modes that involve token rotation along with regularly re-requesting them from IAM (**Refresh Token**, **Service Account Key**, or **OAuth 2.0 token exchange**), a special parameter can be set to indicate where the IAM service is located:

- `--iam-endpoint <URL>` : Sets the URL of the IAM service to request new tokens in authentication modes with token rotation. The default value is `"iam.api.cloud.yandex.net"`.

Parameters from the profile set by the command-line option

If a certain connection parameter is not specified in the command line when calling the YDB CLI, it tries to determine it by the profile set in the `--profile` command-line option.

In the profile, you can define most of the variables that have counterparts in the Command line parameters section. Their values are processed in the same way as command line parameters.

Parameters from environment variables

If you did not explicitly specify a profile or authentication parameters at the command line, the YDB CLI attempts to determine the authentication mode and parameters from the YDB CLI environment as follows:

- If the value of the `IAM_TOKEN` environment variable is set, the **Access Token** authentication mode is used, where this variable value is passed.
- Otherwise, if the value of the `YC_TOKEN` environment variable is set, the **Refresh Token** authentication mode is used and the token to transfer to the IAM endpoint is taken from this variable value when repeating the request.
- Otherwise, if the value of the `USE_METADATA_CREDENTIALS` environment variable is set to 1, the **Metadata** authentication mode is used.
- Otherwise, if the value of the `SA_KEY_FILE` environment variable is set, the **Service Account Key** authentication mode is used and the key is taken from the file whose name is specified in this variable.

- If the value of the `YDB_USER` or `YDB_PASSWORD` environment variable is set, the username and password based authentication mode is used. The username is read from the `YDB_USER` variable. If it is not set, an error saying `User password was provided without user name` is returned. The password is read from the `YDB_PASSWORD` variable. If it is not set, then, depending on whether the `--no-password` command-line option is specified, either an empty password is used or a password is requested interactively.
- Otherwise, if the `YDB_OAUTH2_KEY_FILE` environment variable value is set, the **OAuth 2.0 token exchange** authentication mode is used, and the token exchange parameters are taken from a JSON file specified in this variable. The `--iam-endpoint` option is used to set the token exchange endpoint in the `<schema>://<host>:<port>/<path>` format (via a YDB CLI option, profile, or environment variable).

Parameters from the activated profile

If some connection parameter could not be determined in the previous steps, and you did not explicitly specify a profile at the command line with the `--profile` option, the YDB CLI attempts to use the connection parameters from the activated profile.

Error messages

Errors before attempting to establish a DB connection

If the CLI completed all the steps listed at the beginning of this article but failed to determine the endpoint, the command terminates with the error `Missing required option 'endpoint'`.

If the CLI completed all the steps listed at the beginning of this article but failed to determine the database path, the command terminates with the error message `Missing required option 'database'`.

If the authentication mode is known, but the necessary additional parameters are not, the command is aborted and an error message describing the issue is returned:

- `(No such file or directory) util/system/file.cpp:857: can't open "<filepath>" with mode RdOnly|Seq (0x00000028)`: Couldn't open and read the file `<filepath>` specified in a parameter passing the file name and path.

Additional parameters

When using gRPCs (with encryption), you may need to select a root certificate.

- `--ca-file <filepath>`: Root certificate PEM file for a TLS connection.

Authentication

The YDB CLI `discovery whoami` auxiliary command lets you check the account that you actually used to authenticate with the server.

YDB CLI commands

General syntax for calling YDB CLI commands:

```
ydb [global options] <command> [<subcommand> ...] [command options]
```

where:

- `ydb` is the command to run the YDB CLI from the OS command line.
- `[global options]` are global options that are common for all YDB CLI commands.
- `<command>` is the command.
- `[<subcomand> ...]` are subcommands specified if the selected command contains subcommands.
- `[command options]` are command options specific to each command and subcommands.

Commands

You can learn about the necessary commands by selecting the subject section in the menu on the left or using the alphabetical list below.

Any command can be run from the command line with the `--help` option to get help on it. You can get a list of all commands supported by the YDB CLI by running the YDB CLI with the `--help` option, but without any command.

Command / subcommand	Brief description
admin cluster bridge failover	Emergency switchover when pile is unavailable
admin cluster bridge list	List state of each pile in bridge mode
admin cluster bridge rejoin	Returning pile to the cluster after maintenance or recovery
admin cluster bridge switchover	Planned `PRIMARY` pile change
admin cluster bridge takedown	Taking pile out of the cluster for maintenance
admin cluster config fetch	Getting the current dynamic configuration of the cluster
admin cluster config generate	Generating a dynamic configuration from a static startup configuration
admin cluster config replace	Replacing the cluster dynamic configuration
admin cluster config resolve	Computing the final cluster dynamic configuration based on the base configuration and override selectors
admin cluster config version	Displaying the cluster configuration version on nodes
admin cluster dump	Dumping cluster' metadata to the file system
admin cluster restore	Restoring cluster' metadata from the file system
admin database config fetch	Getting the current dynamic configuration of the database
admin database config generate	Generating a database dynamic configuration from a static startup configuration
admin database config replace	Replacing the database dynamic configuration
admin database config resolve	Computing the final database dynamic configuration based on the base configuration and override selectors
admin database config version	Displaying the database configuration version
admin database dump	Dumping database' data and metadata to the file system
admin database restore	Restoring database' data and metadata from the file system
admin node config init	Initializing node configuration
auth get-token	Getting an authentication token from authentication parameters
config info	Viewing connection parameters
	config profile activate
	config profile create
	config profile delete
	config profile deactivate
	config profile get
	config profile list
	config profile replace
	config profile set
	config profile update
	debug latency

	debug ping
	discovery list
	discovery whoami
	export s3
	import file csv
	import file json
	import file parquet
	import file tsv
	import s3
	init
	monitoring healthcheck
	operation cancel
	operation forget
	operation get
	operation list
	scheme describe
	scheme ls
	scheme mkdir
	scheme permissions chown
	scheme permissions clear
	scheme permissions grant
	scheme permissions revoke
	scheme permissions set
	scheme permissions list
	scheme permissions clear-inheritance
	scheme permissions set-inheritance
	scheme rmdir
	scripting yql
	sql
	table attribute add
	table attribute drop
	table drop
	table index add global-async
	table index add global-sync
	table index drop
	table index rename
	table query execute
	table query explain
	table read
	table ttl set
	table ttl reset
	tools copy
	tools dump
	tools infer csv
	tools rename
	tools restore
	topic create

	topic alter
	topic drop
	topic consumer add
	topic consumer describe
	topic consumer drop
	topic consumer offset commit
	topic read
	topic write
	update
	version
	workload clickbench init
	workload clickbench import files
	workload clickbench run
	workload clickbench clean
	workload kv init
	workload kv run upsert
	workload kv run insert
	workload kv run mixed
	workload kv run read-rows
	workload kv run select
	workload kv clean
	workload log init
	workload log import generator
	workload log run bulk-upsert
	workload log run delete
	workload log run insert
	workload log run upsert
	workload log run select
	workload log clean
	workload mixed init
	workload mixed run bulk-upsert
	workload mixed run insert
	workload mixed run upsert
	workload mixed run select
	workload mixed clean
	workload query init
	workload query import
	workload query run
	workload query clean
	workload stock init
	workload stock run add-rand-order
	workload stock run put-rand-order
	workload stock run put-same-order
	workload stock run rand-user-hist
	workload stock run user-hist
	workload stock clean
	workload topic init

	workload topic run full
	workload topic run read
	workload topic run write
	workload topic clean
	workload tpcc init
	workload tpcc import
	workload tpcc check
	workload tpcc run
	workload tpcc clean
	workload tpcds init
	workload tpcds import generator
	workload tpcds run
	workload tpcds clean
	workload tpch init
	workload tpch import generator
	workload tpch run
	workload tpch clean
	workload transfer topic-to-table init
	workload transfer topic-to-table run
	workload transfer topic-to-table clean
	workload vector init
	workload vector run select
	workload vector run upsert
	workload vector clean
	workload fulltext init
	workload fulltext import files
	workload fulltext import generator
	workload fulltext run select
	workload fulltext run upsert
	workload fulltext clean
	workload fulltext model
	yql

Service commands

These commands have to do with the YDB CLI client itself and do not involve establishing a DB connection. They can be expressed either as a parameter or as an option.

Name	Description
`-?`, `-h`, `--help`	Output the YDB CLI syntax help
`version`	Output the YDB CLI version (for public builds)
`update`	Update the YDB CLI to the latest version (for public builds)
`config info`	Displaying connection parameters
`--license`	Show the license (for public builds)
`--credits`	Show third-party product licenses (for public builds)

If it is not known whether the used YDB CLI build is public, you can find out if a particular service command is supported through help output.

Global parameters

DB connection options

DB connection options are described in Connecting to and authenticating with a database.

Service options

- `--profile <name>`: Indicates the use of the DB connection profile with the specified name when executing a YDB CLI command. Most connection parameters can be stored in the profile.
- `-v, --verbose`: Prints detailed information about all operations being executed. Specifying this option is helpful when locating DB connection issues.
- `--profile-file`: Use profiles from the specified file. By default, profiles from the `~/.ydb/config/config.yaml` file are used.

table attribute add

With the `table attribute add` command, you can add a custom attribute to your table.

General format of the command:

```
ydb [global options...] table attribute add [options...] <table path>
```

- `global options`: Global parameters.
- `options`: Parameters of the subcommand.
- `table path`: The table path.

Look up the description of the command to add a custom attribute:

```
ydb table attribute add --help
```

Parameters of the subcommand

Name	Description
`--attribute`	The custom attribute in the `<key>=<value>` format. You can use `--attribute` many times to add multiple attributes by a single command.

Examples

Add the custom attributes with the keys `attr_key1`, `attr_key2` and the respective values `attr_value1`, `attr_value2` to the `my-table` table:

```
ydb table attribute add --attribute attr_key1=attr_value1 --attribute attr_key2=attr_value2 my-table
```

table attribute drop

With the `table attribute drop` command, you can drop a custom attribute from your table.

General format of the command:

```
ydb [global options...] table attribute drop [options...] <table path>
```

- `global options`: Global parameters.
- `options`: Parameters of the subcommand.
- `table path`: The table path.

Look up the description of the command to add a custom attribute:

```
ydb table attribute drop --help
```

Parameters of the subcommand

Name	Description
`--attributes`	The key of the custom attribute to be dropped. You can list multiple keys separated by a comma (`,`).

Examples

Drop the custom attributes with the keys `attr_key1` and `attr_key2` from the `my-table` table:

```
ydb table attribute drop --attributes attr_key1,attr_key2 my-table
```

List of objects

The `scheme ls` command lets you get a list of scheme objects in the database:

```
ydb [connection options] scheme ls [path] [-lR1]
```

where [connection options] are database connection options

Executing the command without parameters produces a compressed list of object names in the database's root directory.

In the `path` parameter, you can specify the directory you want to list objects in.

The following options are available for the command:

- `-l`: Full details about attributes of each object.
- `-R`: Recursive traversal of all subdirectories.
- `-1`: Output a single schema object per row (for example, to be later handled in a script).

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

- Getting objects from the root database directory in a compressed format

```
ydb --profile quickstart scheme ls
```

- Getting objects in all database directories in a compressed format

```
ydb --profile quickstart scheme ls -R
```

- Getting objects from the given database directory in a compressed format

```
ydb --profile quickstart scheme ls dir1
ydb --profile quickstart scheme ls dir1/dir2
```

- Getting objects in all subdirectories in the given directory in a compressed format

```
ydb --profile quickstart scheme ls dir1 -R
ydb --profile quickstart scheme ls dir1/dir2 -R
```

- Getting complete information on objects in the root database directory

```
ydb --profile quickstart scheme ls -l
```

- Getting complete information about objects in a given database directory

```
ydb --profile quickstart scheme ls dir1 -l
ydb --profile quickstart scheme ls dir2/dir3 -l
```

- Getting complete information about objects in all database directories

```
ydb --profile quickstart scheme ls -lR
```

- Getting complete information on objects in all subdirectories of a given database directory

```
ydb --profile quickstart scheme ls dir1 -lR
ydb --profile quickstart scheme ls dir2/dir3 -lR
```

Getting information about schema objects

Get information about a schema object:

```
ydb scheme describe episodes --stats
```

Result:

```
<table> episodes

┌─────────────┬──────────┬────────┬─────┐
| Name        | Type     | Family | Key |
├─────────────┼──────────┼────────┼─────┤
| air_date    | Uint64? |        |     |
| episode_id | Uint64? |        | K2  |
| season_id   | Uint64? |        | K1  |
| series_id   | Uint64? |        | K0  |
| title       | Utf8?    |        |     |
└─────────────┴──────────┴────────┴─────┘

Storage settings:
Internal channel 0 commit log storage pool: ssd
Internal channel 1 commit log storage pool: ssd
Store large values in "external blobs": false

Column families:
┌──────────┬──────┬──────────────┬──────────────────┐
| Name     | Data | Compression | Keep in memory |
├──────────┼──────┼──────────────┼──────────────────┤
| default | ssd  | None         |                  |
└──────────┴──────┴──────────────┴──────────────────┘

Auto partitioning settings:
Partitioning by size: true
Partitioning by load: false
Preferred partition size (Mb): 2048

Table stats:
Partitions count: 1
Approximate number of rows: 70
Approximate size of table: 11.05 Kb
Last modified: Thu, 17 Jun 2021 11:01:06 UTC
Created: Thu, 17 Jun 2021 11:00:29 UTC
```

Permissions

General list of commands

You can get a list of available commands via interactive help:

```
ydb scheme permissions --help
```

```
Usage: ydb [global options...] scheme permissions [options...] <subcommand>

Description: Modify permissions

Subcommands:
permissions                Modify permissions
├─ chown                   Change owner
├─ clear                   Clear permissions
├─ grant                   Grant permission (aliases: add)
├─ list                    List permissions
├─ revoke                  Revoke permission (aliases: remove)
├─ set                     Set permissions
├─ clear-inheritance       Do not inherit permissions from the parent
└─ set-inheritance         Inherit permissions from the parent
```

All commands have an additional parameter, which is not critical for them:
`--timeout ms` - a technical parameter that sets the server response timeout.

grant, revoke

The `grant` and `revoke` commands allow you to establish and revoke, respectively, access rights to schema objects for a user or group of users. Essentially, they are analogues of the corresponding YQL GRANT and REVOKE commands.

The syntax of the YDB CLI commands is as follows:

```
ydb [connection options] scheme permissions grant  [options...] <path> <subject>
ydb [connection options] scheme permissions revoke [options...] <path> <subject>
```

Parameters:

`<path>` — the full path from the root of the cluster to the object whose rights need to be modified.
`<subject>` — the name of the user or group whose access rights are being changed.

Additional parameters `[options...]`:

`{-p|--permission} NAME` — the list of rights that need to be granted (`grant`) or revoked (`revoke`) for the user.

Each right must be passed as a separate parameter, for example:

```
ydb scheme permissions grant -p "ydb.access.grant" -p "ydb.generic.read" '/Root/db1/MyApp/Orders' testuser
```

set

The `set` command allows you to set access rights to schema objects for a user or group of users.

Command syntax:

```
ydb [connection options] scheme permissions set  [options...] <path> <subject>
```

The values of all parameters are identical to the `grant` , `revoke` commands. However, the key difference of the `set` command from `grant` and `revoke` is that it sets exactly those access rights to the specified object that are listed in the `-p (--permission)` parameters. Other rights for the specified user or group will be revoked.

For example, previously the user `testuser` was granted rights to the object `'/Root/db1'` such as `"ydb.granular.select_row"` , `"ydb.granular.update_row"` , `"ydb.granular.erase_row"` , `"ydb.granular.read_attributes"` , `"ydb.granular.write_attributes"` , `"ydb.granular.create_directory"` .
Then, as a result of executing the command, all rights to the specified object will be revoked (as if `revoke` was called for each of the rights) and only the right `"ydb.granular.select_row"` specified in the `set` command will remain:

```
ydb scheme permissions set -p "ydb.granular.select_row" '/Root/db1' testuser
```

list

The `list` command allows you to obtain the current list of access rights to schema objects.

Command syntax:

```
ydb [connection options] scheme permissions list [options...] <path>
```

Parameters:
`<path>` — the full path from the cluster's root to the object you want to get rights for.

Example result of executing `list` :

```
ydb scheme permissions list '/Root/db1/MyApp'
```

```
Owner: root

Permissions:
user1:ydb.generic.read

Effective permissions:
USERS:ydb.database.connect
METADATA-READERS:ydb.generic.list
DATA-READERS:ydb.granular.select_row
DATA-WRITERS:ydb.tables.modify
DDL-ADMINS:ydb.granular.create_directory,ydb.granular.write_attributes,ydb.granular.create_table,ydb.granula
r.remove_schema,ydb.granular.alter_schema
ACCESS-ADMINS:ydb.access.grant
DATABASE-ADMINS:ydb.generic.manage
user1:ydb.generic.read
```

The result structure consists of three blocks:

- `Owner` — shows the owner of the schema object.
- `Permissions` — displays a list of rights directly given to this object.
- `Effective permissions` — displays a list of rights that are effectively applied to this schema object, taking into account the rules of rights inheritance. This list also includes all the rights displayed in the `Permissions` section.

clear

The `clear` command allows you to revoke all previously granted rights to the schema object. Rights that apply to it by inheritance rules will continue to apply.

```
ydb [global options...] scheme permissions clear [options...] <path>
```

Parameters:
`<path>` — the full path from the root of the cluster to the object whose permissions need to be revoked.

For example, if you execute the command over the database state from the previous example `list`:

```
ydb scheme permissions clear '/Root/db1/MyApp'
```

And then execute the `list` command again on the object `/Root/db1/MyApp`, you will get the following result:

```
Owner: root

Permissions:
none

Effective permissions:
USERS:ydb.database.connect
METADATA-READERS:ydb.generic.list
DATA-READERS:ydb.granular.select_row
DATA-WRITERS:ydb.tables.modify
DDL-ADMINS:ydb.granular.create_directory,ydb.granular.write_attributes,ydb.granular.create_table,ydb.granula
r.remove_schema,ydb.granular.alter_schema
ACCESS-ADMINS:ydb.access.grant
DATABASE-ADMINS:ydb.generic.manage
```

Note that the `Permissions` section is now empty. This means all permissions for this object have been revoked. Also, there have been changes in the contents of the `Effective permissions` section: it no longer lists the permissions that were granted directly to the object `/Root/db1/MyApp`.

chown

The `chown` command allows you to change the owner of a schema object.

Command syntax:

```
ydb [connection options] scheme permissions chown [options...] <path> <owner>
```

Parameters:
`<path>` — the full path from the root of the cluster to the object whose permissions need to be modified.
`<owner>` — the name of the new owner (a user or a group) of the specified object.

Example of a `chown` command:

```
ydb scheme permissions chown '/Root/db1' testuser
```

> **Note**
>
> In the current version of YDB, there is a restriction that only the user who is the current owner of the schema object can change the owner.

clear-inheritance

The `clear-inheritance` command allows you to prohibit the inheritance of permissions for a schema object.

Command syntax:

```
ydb [connection options] scheme permissions clear-inheritance [options...] <path>
```

Parameters:
`<path>` — the full path from the cluster' root to the object whose permissions need to be modified.

Example of a `clear-inheritance` command:

```
ydb scheme permissions clear-inheritance '/Root/db1'
```

set-inheritance

The `set-inheritance` command allows you to enable permission inheritance for a schema object.

Command syntax:

```
ydb [connection options] scheme permissions set-inheritance [options...] <path>
```

Parameters:
`<path>` — the full path from the cluster's root to the object whose permissions need to be modified.

Example of a `set-inheritance` command:

```
ydb scheme permissions set-inheritance '/Root/db1'
```

Directories

The YDB database supports an internal directory structure that can host database objects.

YDB CLI supports operations to change the directory structure and to access schema objects by their directory name.

Creating a directory

The `scheme mkdir` command creates the directories:

```
ydb [connection options] scheme mkdir <path>
```

where [connection options] are database connection options

In the `path` parameter, specify the relative path to the directory being created, from the root database directory. This command creates all the directories that didn't exist at the path when the command was called.

If the destination directory had already existed at the path, then the command execution will be completed successfully (result code 0) with a warning that no changes have been made:

```
Status: SUCCESS
Issues:
<main>: Error: dst path fail checks, path: /<database>/<path>: path exist, request accepts it,
pathId: [OwnerId: <some>, LocalPathId: <some>], path type: EPathTypeDir, path state: EPathStateNoChanges
```

It also supports the syntax of a full path beginning with `/`. The full path must begin with the path to the database specified in the connection parameters or allowed by the current connection to the cluster.

Examples:

- Creating a directory at the database root

  ```
  ydb --profile quickstart scheme mkdir dir1
  ```

- Creating directories at the specified path from the database root

  ```
  ydb --profile quickstart scheme mkdir dir1/dir2/dir3
  ```

Deleting a directory

The `scheme rmdir` command deletes a directory:

```
ydb [global options...] scheme rmdir [options...] <path>
```

- `global options`: Global parameters.
- `options`: Parameters of the subcommand.
- `path`: Path to the deleted directory.

Look up the description of the directory deletion command:

```
ydb scheme rmdir --help
```

Parameters of the subcommand

Name	Description
`-r`, `--recursive`	This option deletes the directory recursively, which all its child objects (subdirectories, tables, topics). If you use this option, the confirmation prompt is shown by default.
`-f`, `--force`	Do not prompt for confirmation.
`-i`	Prompt for deletion confirmation on each object.
`-I`	Show a single confirmation prompt.
`--timeout <value>`	Operation timeout, ms.

If you try to delete a non-empty directory without the `-r` or `--recursive` option, the command fails with an error.

```
Status: SCHEME_ERROR
Issues:
<main>: Error: path table fail checks, path: /<database>/<path>: path has children, request
doesn't accept it, pathId: [OwnerId: <some>, LocalPathId: <some>], path type:
EPathTypeDir, path state: EPathStateNoChanges, alive children: <count>
```

Examples

- Deleting an empty directory:

  ```
  ydb scheme rmdir dir1
  ```

- Deleting an empty directory with a confirmation prompt:

```
ydb scheme rmdir -I dir1
```

- Recursively deleting a non-empty directory with a confirmation prompt:

```
ydb scheme rmdir -r dir1
```

- Recursively deleting a non-empty directory without a confirmation prompt:

```
ydb scheme rmdir -rf dir1
```

- Recursively deleting a non-empty directory, showing a confirmation prompt on each object:

```
ydb scheme rmdir -ri dir1
```

Using directories in other CLI commands

In all CLI commands to which the object name is passed by the parameter, it can be specified with a directory, for example, in `scheme describe`:

```
ydb --profile quickstart scheme describe dir1/table_a
```

The `scheme ls` command supports passing the path to the directory as a parameter:

```
ydb --profile quickstart scheme ls dir1/dir2
```

Using directories in YQL

Names of objects used in YQL queries may contain paths to directories hosting such objects. This path will be concatenated with the path prefix from the `TablePathPrefix` pragma. If the pragma is omitted, the object name is resolved relative to the database root.

Implicit creation of directories during import

The data import command creates a directory tree mirroring the original imported catalog.

Creating and deleting secondary indexes

By using the `table index` command, you can create and delete secondary indexes:

```
ydb [connection options] table index [subcommand] [options]
```

where [connection options] are database connection options

You can also add or delete a secondary index with the ADD INDEX and DROP INDEX directives of YQL ALTER TABLE.

To learn about secondary indexes and their use in application development, see Secondary indexes under "Recommendations".

Creating a secondary index

Secondary indexes are created with the `table index add` command:

```
ydb [connection options] table index add <sync-async> <table> \
  --index-name STR --columns STR [--cover STR]
```

Parameters:

`<sync-async>`: The type of the secondary index. Use `global-sync` to build an index updated synchronously or `global-async` to build an index updated asynchronously.

`<table>`: The path and name of the table you are building an index for

`--index-name STR`: A mandatory parameter that sets the name of the index. It is recommended that you specify the names of such indexes, so that the columns included in them can be identified. Index names are unique in the context of the table.

`--columns STR`: A required parameter that defines the columns used in the index and their order in the index key. Column names are separated by a comma, with no spaces. The index key will include both the columns listed and the columns from the table's primary key.

`--cover STR`: An optional parameter that defines the covering columns of the index. Their values won't be added to the index key, but will be written to the index. This enables you to retrieve the values when searching the index without accessing the table.

When the command is executed, the DBMS starts building the index in the background, and the pseudographics-formatted `id` field shows the operation ID, so you can retrieve its status by `operation get`. When the index is being built, you can abort the process using `operation cancel`.

To forget an index-building operation (either completed or terminated), use `operation forget`.

To retrieve the status of all index-building operations, use `operation list buildindex`.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Adding a synchronous index built on the `air_date` column to the `episodes` table created previously:

```
ydb -p quickstart table index add global-sync episodes \
  --index-name idx_aired --columns air_date
```

Adding to the previously created `series` table an asynchronous index built on the `release_date` and `title` columns, copying to the index the `series_info` column value:

```
ydb -p quickstart table index add global-async series \
  --index-name idx_rel_title --columns release_date,title --cover series_info
```

Result (the actual operation id might differ):

```
┌──────────────────────────────────────┬───────┬────────┐
| id                                   | ready | status |
├──────────────────────────────────────┼───────┼────────┤
| ydb://buildindex/7?id=2814749869 | false |        |
└──────────────────────────────────────┴───────┴────────┘
```

Getting the operation status (use the actual operation id):

```
ydb -p quickstart operation get ydb://buildindex/7?id=281474976866869
```

Returned value:

```
┌──────────────────────────────────────┬───────┬─────────┬───────┬──────────┬──────────────────┬────────────┐
| id                                   | ready | status  | state | progress | table            | index      |
├──────────────────────────────────────┼───────┼─────────┼───────┼──────────┼──────────────────┼────────────┤
| ydb://buildindex/7?id=2814749869 | true  | SUCCESS | Done  | 100.00%  | /local/episodes | idx_aired |
└──────────────────────────────────────┴───────┴─────────┴───────┴──────────┴──────────────────┴────────────┘
```

Deleting the index-building details (use the actual operation id):

```
ydb -p quickstart operation forget ydb://buildindex/7?id=2814749869
```

Deleting a index

Indexes are deleted by the `table index drop` command:

```
ydb [connection options] table index drop <table> --index-name STR
```

Example

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Deleting the `idx_aired` index from the episodes table (see the index-building example above):

```
ydb -p quickstart table index drop episodes --index-name idx_aired
```

Renaming a secondary index

To rename secondary indexes, use the `table index rename` command:

```
ydb [connection options] table index rename <table> --index-name STR --to STR
```

If an index with the new name exists, the command returns an error.

To replace your existing index atomically, execute the rename command with the `--replace` option:

```
ydb [connection options] table index rename <table> --index-name STR --to STR --replace
```

Example

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Renaming the `idx_aired` index built on the episodes table (see the example of index creation above):

```
ydb -p quickstart table index rename episodes --index-name idx_aired --to idx_aired_renamed
```

Copying tables

Using the `tools copy` subcommand, you can create a copy of one or more DB tables. The copy operation leaves the source table unchanged while the copy contains all the source table data.

General format of the command:

```
ydb [global options...] tools copy [options...]
```

- `global options`: Global parameters.
- `options`: Parameters of the subcommand.

View a description of the command to copy a table:

```
ydb tools copy --help
```

Parameters of the subcommand

Parameter name	Parameter description
`--timeout`	The time within which the operation should be completed on the server.
`--item <property>=<value>,...`	Operation properties. You can specify the parameter more than once to copy several tables in a single transaction. Required properties: • `destination`, `dst`, `d`: Path to target table. If the destination path contains folders, they must be created in advance. No table with the destination name should exist. • `source`, `src`, `s`: Path to source table. Optional properties: • `omit-indexes`: If `true`, indexes are not copied (default: `false`).

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Create the `backup` folder in the DB:

```
ydb -p quickstart scheme mkdir backup
```

Copy the `series` table to a table called `series-v1`, the `seasons` table to a table called `seasons-v1`, and `episodes` to `episodes-v1` in the `backup` folder:

```
ydb -p quickstart tools copy --item destination=backup/series-v1,source=series --item destination=backup/seas
ons-v1,source=seasons --item destination=backup/episodes-v1,source=episodes
```

View the listing of objects in the `backup` folder:

```
ydb -p quickstart scheme ls backup
```

Result:

```
episodes-v1  seasons-v1  series-v1
```

Renaming a table

Using the `tools rename` subcommand, you can rename one or more tables at the same time, move a table to another directory within the same database, replace one table with another one within the same transaction.

General command format:

```
ydb [global options...] tools rename [options...]
```

- `global options` : Global parameters.
- `options` : Subcommand parameters.

View a description of the command to rename a table:

```
ydb tools rename --help
```

Subcommand parameters

A single run of the `tools rename` command executes a single rename transaction that may include one or more operations to rename different tables.

Parameter name	Parameter description
`--item <property>=<value>,...`	Description of the rename operation. Can be specified multiple times if multiple rename operations need to be executed within a single transaction. Required properties: • `source`, `src`, and `s`: Path to the source table. • `destination`, `dst`, and `d`: Path to the destination table. If the destination path contains folders, they must be created in advance. Advanced properties: • `replace`, `force`: Overwrite the destination table. If `True`, the destination table is overwritten with its data deleted. `False`: If the destination table exists, an error is returned and the entire rename transaction is canceled. Default value: `False`.
`--timeout <value>`	Operation timeout, ms.

When including multiple rename operations in a single `tools rename` call, they're executed in the specified order, but within a single transaction. This lets you rotate the table under load without data loss: the first operation is renaming the working table to the backup one and the second is renaming the new table to the working one.

Examples

- Renaming a single table:

  ```
  ydb tools rename --item src=old_name,dst=new_name
  ```

- Renaming multiple tables within a single transaction:

  ```
  ydb tools rename \
    --item source=new-project/main_table,destination=new-project/episodes \
    --item source=new-project/second_table,destination=new-project/seasons \
    --item source=new-project/third_table,destination=new-project/series
  ```

- Moving tables to a different directory:

  ```
  ydb tools rename \
    --item source=new-project/main_table,destination=cinema/main_table \
    --item source=new-project/second_table,destination=cinema/second_table \
    --item source=new-project/third_table,destination=cinema/third_table
  ```

- Replacing a table

  ```
  ydb tools rename \
    --item replace=True,source=pre-prod-project/main_table,destination=prod-project/main_table
  ```

- Rotating a table

  ```
  ydb tools rename \
    --item source=prod-project/main_table,destination=prod-project/main_table.backup \
    --item source=pre-prod-project/main_table,destination=prod-project/main_table
  ```

Setting TTL parameters

Use the `table ttl set` subcommand to set TTL for the specified table.

General format of the command:

```
ydb [global options...] table ttl set [options...] <table path>
```

- `global options`: Global parameters.
- `options`: Parameters of the subcommand.
- `table path`: The table path.

View a description of the TTL set command:

```
ydb table ttl set --help
```

Parameters of the subcommand

Name	Description
`--column`	The name of the column that will be used to calculate the lifetime of the rows. The column must have the numeric or date and time type. In case of the numeric type, the value will be interpreted as the time elapsed since the beginning of the Unix epoch. Measurement units must be specified in the `--unit` parameter.
`--expire-after`	Additional time before deleting that must elapse after the lifetime of the row has expired. Specified in seconds. The default value is `0`.
`--unit`	The value measurement units of the column specified in the `--column` parameter. It is mandatory if the column has the numeric type. Possible values: • `seconds (s, sec)`: Seconds. • `milliseconds (ms, msec)`: Milliseconds. • `microseconds (us, usec)`: Microseconds. • `nanoseconds (ns, nsec)`: Nanoseconds.
`--run-interval`	The interval for running the operation to delete rows with expired TTL. Specified in seconds. The minimum allowed interval is 15 minutes (900 seconds). The default value (`3600`) is a good starting point for working with a TTL-enabled table. When choosing `--run-interval`, balance the delay before expired data is physically removed against the extra load on the cluster. Too small an interval triggers deletion background tasks frequently and can noticeably increase CPU and I/O load, especially with large data volumes. Too large an interval, conversely, delays physical deletion of rows: expired data occupies disk space longer and may continue to participate in scans. Regardless of the `--run-interval` value, queries to the table may return rows with expired TTL.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Set TTL for the `series` table

```
ydb -p quickstart table ttl set \
  --column createtime \
  --expire-after 3600 \
  --run-interval 1200 \
  series
```

Resetting TTL parameters

Use the `table ttl reset` subcommand to reset TTL for the specified table.

General format of the command:

```
ydb [global options...] table ttl reset <table path>
```

- `global options`: Global parameters.
- `table path`: The table path.

View the description of the TTL reset command:

```
ydb table ttl reset --help
```

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Reset TTL for the `series` table:

```
ydb -p quickstart table ttl reset \
  series
```

Deleting a table

Using the `table drop` subcommand, you can delete a specified table.

General format of the command:

```
ydb [global options...] table drop [options...] <table path>
```

- `global options`: Global parameters.
- `options`: Parameters of the subcommand.
- `table path`: The table path.

To view a description of the table delete command:

```
ydb table drop --help
```

Parameters of the subcommand

Name	Description
`--timeout`	The time within which the operation should be completed on the server.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

To delete the table `series`:

```
ydb -p quickstart table drop series
```

Table schema inference from data files

You can use the `ydb tools infer csv` command to generate a `CREATE TABLE` statement from a CSV data file. This can be helpful when you want to import data into a database and the table has not been created yet.

Command syntax:

```
ydb [global options...] tools infer csv [options...] <input files...>
```

- `global options` – global options.
- `options` – subcommand options.

To get the most up-to-date information about the command, use the `--help` option:

```
ydb tools infer csv --help
```

Subcommand options

Option Name	Description
`-p, --path`	Database path to the table that should be created. Default: `table`.
`--columns`	Explicitly specifies table column names, as a comma-separated list.
`--gen-columns`	Explicitly indicates that table column names should be generated automatically (column1, column2, ...).
`--header`	Explicitly indicates that the first row in the CSV contains column names.
`--rows-to-analyze`	Number of rows to analyze. 0 means unlimited. Reading will stop as soon as this number of rows is read. Default: `500000`.
`--execute`	Execute the `CREATE TABLE` request immediately after generation.

Note

If none of the `--columns`, `--gen-columns`, or `--header` options are explicitly specified, the following algorithm is used:

The values of the first row in the file are checked for the following conditions:

- The values meet the requirements for column names.
- The types of the values in the first row are different from the data types in the other rows of the file.

If both conditions are met, the values from the first row are used as the table's column names. Otherwise, column names are generated automatically (as `column1`, `column2`, etc.). See the example below for more details.

Column type inference algorithm

For each column, the command determines the least general type that fits all its values. The most general type is `Text`: if any value in a column is a string (for example, `abc`), the entire column will be inferred as `Text`.

All integer values are inferred as `Int64` if they fit within the `Int64` range. If any value exceeds this range, the type is set to `Double`.

Floating-point numbers are always inferred as `Double`.

Current Limitation

The first column is always chosen as the primary key. You may need to change the primary key to one that is more appropriate for your use case. For recommendations, see Choosing a primary key.

Examples

Note

The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Column names in the first row, no options specified

The `key` and `value` values in the first row match the table column name requirements and do not match the data types in the other rows (`Int64` and `Text`).
So the command uses the first row of the file as column names.

```
$ cat data_with_header.csv
key,value
123,abc
456,def

ydb tools infer csv data_with_header.csv
CREATE TABLE table (
    key Int64,
    value Text,
    PRIMARY KEY (key) -- First column is chosen. Probably need to change this.
)
WITH (
```

```
    STORE = ROW -- or COLUMN
    -- Other useful table options to consider:
    --, AUTO_PARTITIONING_BY_SIZE = ENABLED
    --, AUTO_PARTITIONING_BY_LOAD = ENABLED
    --, UNIFORM_PARTITIONS = 100 -- Initial number of partitions
    --, AUTO_PARTITIONING_MIN_PARTITIONS_COUNT = 100
    --, AUTO_PARTITIONING_MAX_PARTITIONS_COUNT = 1000
);
```

Note

The `WITH` block lists some useful table options. Uncomment the ones you need and remove the rest as appropriate.

Column names in the first row, using `--header` option

In this example, the `key` and `value` values in the first row match the data types (`Text`) in the other rows.
In this case, without the `--header` option, the command would not use the first row of the file as column names but would generate column names automatically.
To use the first row as column names in this situation, use the `--header` option explicitly.

```
$ cat data_with_header_text.csv
key,value
aaa,bbb
ccc,ddd

ydb tools infer csv data_with_header_text.csv --header
CREATE TABLE table (
    key Text,
    value Text,
    PRIMARY KEY (key) -- First column is chosen. Probably need to change this.
)
WITH (
    STORE = ROW -- or COLUMN
    -- Other useful table options to consider:
    --, AUTO_PARTITIONING_BY_SIZE = ENABLED
    --, AUTO_PARTITIONING_BY_LOAD = ENABLED
    --, UNIFORM_PARTITIONS = 100 -- Initial number of partitions
    --, AUTO_PARTITIONING_MIN_PARTITIONS_COUNT = 100
    --, AUTO_PARTITIONING_MAX_PARTITIONS_COUNT = 1000
);
```

Explicit column list

```
cat ~/data_no_header.csv
123,abc
456,def

ydb tools infer csv -p newtable ~/data_no_header.csv --columns my_key,my_value
CREATE TABLE newtable (
    my_key Int64,
    my_value Text,
    PRIMARY KEY (my_key)
)
WITH (
    STORE = ROW -- or COLUMN
    -- Other useful table options to consider:
    --, AUTO_PARTITIONING_BY_SIZE = ENABLED
    --, AUTO_PARTITIONING_BY_LOAD = ENABLED
    --, UNIFORM_PARTITIONS = 100 -- Initial number of partitions
    --, AUTO_PARTITIONING_MIN_PARTITIONS_COUNT = 100
    --, AUTO_PARTITIONING_MAX_PARTITIONS_COUNT = 1000
);
```

Automatically generate column names

```
cat ~/data_no_header.csv
123,abc
456,def

ydb tools infer csv -p newtable ~/data_no_header.csv --gen-columns
CREATE TABLE newtable (
    column1 Int64,
    column2 Text,
    PRIMARY KEY (f0) -- First column is chosen. Probably need to change this.
)
WITH (
    STORE = ROW -- or COLUMN
    -- Other useful table options to consider:
    --, AUTO_PARTITIONING_BY_SIZE = ENABLED
    --, AUTO_PARTITIONING_BY_LOAD = ENABLED
    --, UNIFORM_PARTITIONS = 100 -- Initial number of partitions
    --, AUTO_PARTITIONING_MIN_PARTITIONS_COUNT = 100
    --, AUTO_PARTITIONING_MAX_PARTITIONS_COUNT = 1000
);
```

Executing generated statement using the `--execute` option

In this example, the `CREATE TABLE` statement is actually executed right after generation.

```
$ cat data_with_header.csv
key,value
123,abc
456,def

ydb -p quickstart tools infer csv data_with_header.csv --execute
Executing request:

CREATE TABLE table (
    key Int64,
    value Text,
    PRIMARY KEY (key) -- First column is chosen. Probably need to change this.
)
WITH (
    STORE = ROW -- or COLUMN
    -- Other useful table options to consider:
    --, AUTO_PARTITIONING_BY_SIZE = ENABLED
    --, AUTO_PARTITIONING_BY_LOAD = ENABLED
    --, UNIFORM_PARTITIONS = 100 -- Initial number of partitions
    --, AUTO_PARTITIONING_MIN_PARTITIONS_COUNT = 100
    --, AUTO_PARTITIONING_MAX_PARTITIONS_COUNT = 1000
);

Query executed successfully.
```

Query execution plan and AST

YDB provides two types of query plans: a logical plan and an execution plan. The logical plan is better suited for analyzing complex queries with many join operators. The execution plan is more detailed: it additionally shows the stages of the distributed plan and the connectors between them, which makes it more convenient for analyzing simple OLTP queries.

Query execution plans

You can display the execution plan via YDB CLI with the `explain` command:

```
ydb -p <profile_name> table query explain \
-q "SELECT season_id, episode_id, title
FROM episodes
WHERE series_id = 1
AND season_id > 1
ORDER BY season_id, episode_id
LIMIT 3"
```

The returned execution plan is shown below:

```
Query Plan:
ResultSet
└──Limit (Limit: 3)
└──<Merge>
└──TopSort (Limit: 3, TopSortBy: )
└──Filter (Predicate: Exist(item.season_id) And Exist(item.series_id))
└──TablePointLookup (ReadRange: [series_id (1), season_id (1, +∞), episode_id (-∞, +∞)], ReadColumns: [episod
e_id, season_id, title, series_id], Table: episodes)
Tables: [episodes]
```

In addition to the query execution plan, you can get an AST (abstract syntax tree).
The AST section contains a representation in the internal miniKQL language.

To display the AST, call `explain` with the `--ast` flag:

```
ydb -p <profile_name> table query explain \
-q "SELECT season_id, episode_id, title
FROM episodes
WHERE series_id = 1
AND season_id > 1
ORDER BY season_id, episode_id
LIMIT 3" --ast
```

The resulting AST:

```
Query AST:
(
(let $1 (KqpTable '"episodes" '"72075186224045943:83859" '"" '1))
(let $2 '('"episode_id" '"season_id" '"title" '"series_id"))
(let $3 (Uint64 '1))
(let $4 (KqpRowsSourceSettings $1 $2 '() '((KqlKeyExc $3 $3) (KqlKeyInc $3))))
(let $5 (Uint64 '"3"))
(let $6 (DqPhyStage '((DqSource (DataSource '"KqpReadRangesSource") $4)) (lambda '($12) (block '(
  (let $13 (Bool 'true))
  (return (FromFlow (TopSort (OrderedMap (OrderedFilter (ToFlow $12) (lambda '($14) (And (Exists (Member $14
'"season_id")) (Exists (Member $14 '"series_id"))))) (lambda '($15) (AsStruct '('"episode_id" (Member $15 '"e
pisode_id")) '('"season_id" (Member $15 '"season_id")) '('"title" (Member $15 '"title"))))) $5 '($13 $13) (la
mbda '($16) '((Member $16 '"season_id") (Member $16 '"episode_id"))))))
))) '('('"_logical_id" '842) '('"_id" '"14388682-e03b28dd-60f91f84-d7cd002f"))))
(let $7 (DqCnMerge (TDqOutput $6 '0) '('('"season_id" '"Asc") '('"episode_id" '"Asc"))))
(let $8 (DqPhyStage '($7) (lambda '($17) (FromFlow (Take (ToFlow $17) $5))) '('('"_logical_id" '855) '('"_id"
'"295c0459-34d4b026-b467e71f-35402430"))))
(let $9 '('"season_id" '"episode_id" '"title"))
(let $10 (DqCnResult (TDqOutput $8 '0) $9))
(let $11 (OptionalType (DataType 'Uint64)))
(return (KqpPhysicalQuery '((KqpPhysicalTx '($6 $8) '($10) '() '('('"type" '"data")))) '((KqpTxResultBinding
(ListType (StructType '('"episode_id" $11) '('"season_id" $11) '('"title" (OptionalType (DataType 'Strin
g))))) '0 '0)) '('('"type" '"data_query"))))
)
```

Logical query plan

Let's consider the analytical query Q18 from the TPCH benchmark:

```
$p = (
  select
    p.p_brand as p_brand,
    p.p_type as p_type,
    p.p_size as p_size,
    ps.ps_suppkey as ps_suppkey
  from
    part as p
  join
    partsupp as ps
on
    p.p_partkey = ps.ps_partkey
where
```

```
    p.p_brand <> 'Brand#45'
    and p.p_type not like 'MEDIUM POLISHED%'
    and (p.p_size = 49 or p.p_size = 14 or p.p_size = 23 or p.p_size = 45 or p.p_size = 19 or p.p_size = 3 or
p.p_size = 36 or p.p_size = 9)
);

$s = (
  select
    s_suppkey
  from
    supplier
  where
    s_comment like "%Customer%Complaints%"
);

$j = (
  select
    p.p_brand as p_brand,
    p.p_type as p_type,
    p.p_size as p_size,
    p.ps_suppkey as ps_suppkey
  from
    $p as p
  left only join
    $s as s
  on
    p.ps_suppkey = s.s_suppkey
);

select
    j.p_brand as p_brand,
    j.p_type as p_type,
    j.p_size as p_size,
    count(distinct j.ps_suppkey) as supplier_cnt
from
    $j as j
group by
    j.p_brand,
    j.p_type,
    j.p_size
order by
    supplier_cnt desc,
    p_brand,
    p_type,
    p_size
;
```

You can get the logical execution plan with the command:

```
ydb -p <profile_name> sql --explain -f q18.sql
```

This will result in a logical query plan consisting of the plan's operators and various optimizer predictions:

```
Query Plan:
┌────────────┬────────────┬────────────┬──────────────────────────────────────────────────────────┐
│ E-Cost     │ E-Rows     │ E-Size     │ Operation                                                │
├────────────┼────────────┼────────────┼──────────────────────────────────────────────────────────┤
│            │            │            │ ┌> ResultSet                                             │
│            │            │            │ └─┬> Sort ([row.supplier_cnt,row.p_brand,row.p_type,r    │
│            │            │            │ ow.p_size])                                              │
│            │            │            │   └─┬> Aggregate (Phase: Final)                          │
│            │            │            │     └─┬> HashShuffle (KeyColumns: ["part.p_brand","pa    │
│            │            │            │ rt.p_size","part.p_type"], HashFunc: "HashV2")           │
│            │            │            │       └─┬> Aggregate (GroupBy: [part.p_brand,part.p_s    │
│            │            │            │ ize,part.p_type], Aggregation: {Inc(_yql_agg_0)}, Pha    │
│            │            │            │ se: Intermediate)                                        │
│            │            │            │         └─┬> Aggregate (Phase: Final)                    │
│            │            │            │           └─┬> HashShuffle (KeyColumns: ["part.p_bran    │
│            │            │            │ d","part.p_size","part.p_type","partsupp.ps_suppkey"]    │
│            │            │            │ , HashFunc: "HashV2")                                    │
│            │            │            │             └─┬> Aggregate (GroupBy: [part.p_brand,pa    │
│            │            │            │ rt.p_size,part.p_type,partsupp.ps_suppkey], Aggregati    │
│            │            │            │ on: state, Phase: Intermediate)                          │
│ 4.350e+08  │ 40000000   │ 3.421e+09  │               └─┬> LeftOnlyJoin (Grace) (partsupp.ps_    │
│            │            │            │ suppkey = supplier.s_suppkey)                            │
│            │            │            │                 ├─┬> HashShuffle (KeyColumns: ["parts    │
│            │            │            │ upp.ps_suppkey"], HashFunc: "ColumnShardHashV1")         │
│ 3.135e+08  │ 40000000   │ 2.331e+09  │                 │ └─┬> InnerJoin (Grace) (part.p_part    │
│            │            │            │ key = partsupp.ps_partkey)                               │
│ 0          │ 9000000    │ 2.869e+08  │                 │   ├─┬> Filter (Contains)               │
│ 0          │ 9000000    │ 2.869e+08  │                 │   │ └─┬> Filter ((p_brand != "Brand    │
│            │            │            │ #33") AND (NOT p_type LIKE "PROMO POLISHED%"), Pushdo    │
│            │            │            │ wn: True)                                                │
│ 0          │ 20000000   │ 6.375e+08  │                 │   │   └──> TableFullScan (Table: pa    │
│            │            │            │ rt, ReadColumns: ["p_partkey (-∞, +∞)","p_brand","p_s    │
│            │            │            │ ize","p_type"])                                          │
│            │            │            │                 │   └─┬> HashShuffle (KeyColumns: ["p    │
│            │            │            │ s_partkey"], HashFunc: "ColumnShardHashV1")              │
│ 0          │ 80000000   │ 2.113e+09  │                 │     └──> TableFullScan (Table: part    │
│            │            │            │ supp, ReadColumns: ["ps_partkey (-∞, +∞)","ps_suppkey    │
```

```
|           |          |           |  (-∞, +∞)"])                                                  |
| 0         | 500000   | 13621431  |                 └─┬> Filter (Apply, Pushdown: True)   |
| 0         | 1000000  | 27242862  |                   └──> TableFullScan (Table: supplier |
|           |          |           | , ReadColumns: ["s_suppkey (-∞, +∞)","s_comment"])    |
└───────────┴──────────┴───────────┴──────────────────────────────────────────────────────┘
```

This plan shows the operator tree YDB and three optimizer predictions for each operator:

- `E-Cost`: estimated cost of the current operator and all its inputs;
- `E-Rows`: estimated number of records at the output of the current operator;
- `E-Size`: estimated size of the operator's result in bytes.

In addition to optimizer predictions, you can also get actual statistics when executing the query:

- `A-Cpu`: total CPU time for the current operator;
- `A-Rows`: the actual number of records in the output of the current operator.

Execution statistics are not always available for each operator, as they are currently collected by execution stages. Several operators can be placed into a single stage.

To get a logical query plan with the execution statistics, run the following command:

```
ydb -p <profile_name> sql --explain-analyze --format pretty-table --analyze -f q18.sql
```

You will get the following logical query plan:

```
┌─────────┬────────────┬────────────┬────────────┬────────────┬──────────────────────────────────────────────────────────────────────────────────────────────────────────────
─────────────┐
| A-Cpu | A-Rows   | E-Cost     | E-Rows   | E-Size     | Operation
|
├─────────┼────────────┼────────────┼────────────┼────────────┼──────────────────────────────────────────────────────────────────────────────────────────────────────────────
─────────────┤
|       |          |            |          |            | ┌> ResultSet
|
| 934   | 27840    |            |          |            | └─┬> Sort (A-SelfCpu: 15.706, A-Size: 936768, [row.su
pplier_cnt, |
|       |          |            |          |            | row.p_brand,row.p_type,row.p_size])
|
|       | 27840    |            |          |            |   └─┬> Aggregate (Phase: Final)
|
|       |          |            |          |            |     └─┬> HashShuffle (KeyColumns: ["part.p_brand","pa
rt.p_size", |
|       |          |            |          |            | "part.p_type"], HashFunc: "HashV2")
|
| 918   | 3431671  |            |          |            |       └─┬> Aggregate (GroupBy: [part.p_brand,part.p_s
ize,part.p_ |
|       |          |            |          |            | type], Aggregation: {Inc(_yql_agg_0)}, A-SelfCpu: 13
9.139, Phase |
|       |          |            |          |            | : Intermediate, A-Size: 112058009)
|
|       |          |            |          |            |         └─┬> Aggregate (Phase: Final)
|
|       |          |            |          |            |           └─┬> HashShuffle (KeyColumns: ["part.p_bran
d","part.p_ |
|       |          |            |          |            | size","part.p_type","partsupp.ps_suppkey"], HashFunc:
"HashV2")  |
| 779   | 11865156 |            |          |            |             └─┬> Aggregate (GroupBy: [part.p_brand,pa
rt.p_size,p |
|       |          |            |          |            | art.p_type,partsupp.ps_suppkey], Aggregation: state,
A-SelfCpu:  |
|       |          |            |          |            | 194.29, Phase: Intermediate, A-Size: 410786277)
|
|       | 11867631 | 4.350e+08  | 40000000 | 3.421e+09  |               └─┬> LeftOnlyJoin (Grace) (partsupp.ps_
suppkey = s |
|       |          |            |          |            | upplier.s_suppkey)
|
|       |          |            |          |            |                 ├─┬> HashShuffle (KeyColumns: ["parts
upp.ps_supp |
|       |          |            |          |            | key"], HashFunc: "ColumnShardHashV1")
|
| 584   | 11873200 | 3.135e+08  | 40000000 | 2.331e+09  |                 | └─┬> InnerJoin (Grace) (A-SelfCpu:
308.14, par |
|       |          |            |          |            | t.p_partkey = partsupp.ps_partkey, A-Size: 505055811)
|
| 65    | 2968300  | 0          | 9000000  | 2.869e+08  |                 |   ├─┬> Filter (Contains, A-SelfCpu:
65.311, A- |
|       |          |            |          |            | Size: 105580755)
|
|       |          | 0          | 9000000  | 2.869e+08  |                 |   | └─┬> Filter ((p_brand != "Brand
#33") AND ( |
|       |          |            |          |            | NOT p_type LIKE "PROMO POLISHED%"), Pushdown: True)
|
|       | 18560313 | 0          | 20000000 | 6.375e+08  |                 |   |   └──> TableFullScan (Table: pa
rt, ReadCol |
|       |          |            |          |            | umns: ["p_partkey (-∞, +∞)","p_brand","p_size","p_typ
e"], A-Size |
|       |          |            |          |            | : 1060176372)
|
|       |          |            |          |            |                 |   └─┬> HashShuffle (KeyColumns: ["p
s_partkey"] |
```

```
|       |          |           |          |           | , HashFunc: "ColumnShardHashV1")
|
| 211   | 80000000 | 0         | 80000000 | 2.113e+09 |                 |     └──> TableFullScan (Table: part
supp, A-Sel |
|       |          |           |          |           | fCpu: 210.822, ReadColumns: ["ps_partkey (-∞, +∞)","p
s_suppkey ( |
|       |          |           |          |           | -∞, +∞)"], A-Size: 1283276800)
|
| 0     | 479      | 0         | 500000   | 13621431  |                 └─┬> Filter (Apply, A-SelfCpu: 0.27,
Pushdown: T |
|       |          |           |          |           | rue, A-Size: 4937)
|
|       | 479      | 0         | 1000000  | 27242862  |                   └──> TableFullScan (Table: supplie
r, ReadColum |
|       |          |           |          |           | ns: ["s_suppkey (-∞, +∞)","s_comment"], A-Size: 7885)
|
└───────┴──────────┴───────────┴──────────┴───────────┴──────────────────────────────────────────────────────────
──────────┘
```

This plan contains actual execution statistics - `A-Cpu` and `A-Rows` in addition to optimizer estimates.

Streaming table reads

To read an entire table snapshot, use the `read` subcommand. Data is transferred as a stream, which enables you to read any size table.

Read data:

```
ydb table read episodes \
  --ordered \
  --limit 5 \
  --columns series_id,season_id,episode_id,title
```

Where:

- `--ordered`: Order read entries by key.
- `--limit`: Limit the number of entries to read.
- `--columns`: Columns whose values should be read (all by default) in CSV format.

Result:

```
┌───────────┬───────────┬────────────┬──────────────────────────────────┐
| series_id | season_id | episode_id | title                            |
├───────────┼───────────┼────────────┼──────────────────────────────────┤
| 1         | 1         | 1          | "Yesterday's Jam"                |
├───────────┼───────────┼────────────┼──────────────────────────────────┤
| 1         | 1         | 2          | "Calamity Jen"                   |
├───────────┼───────────┼────────────┼──────────────────────────────────┤
| 1         | 1         | 3          | "Fifty-Fifty"                    |
├───────────┼───────────┼────────────┼──────────────────────────────────┤
| 1         | 1         | 4          | "The Red Door"                   |
├───────────┼───────────┼────────────┼──────────────────────────────────┤
| 1         | 1         | 5          | "The Haunting of Bill Crouse"    |
└───────────┴───────────┴────────────┴──────────────────────────────────┘
```

To only get the number of read entries, use the `--count-only` parameter:

```
ydb table read episodes \
  --columns series_id \
  --count-only
```

Result:

```
70
```

Data export and import

YDB CLI contains a set of commands for exporting and importing data and descriptions of data schema objects. Data export can be used both for creating backups for subsequent recovery and for other purposes.

- Export file structure, used both when exporting to a file system and when exporting to an S3-compatible object storage.
- Exporting cluster metadata to a file system using `admin cluster dump`
- Importing cluster metadata from a file system using `admin cluster restore`
- Exporting database data and metadata to a file system using `admin database dump`
- Importing database data and metadata from a file system using `admin database restore`
- Exporting individual schema objects to a file system using `tools dump`
- Importing individual schema objects from a file system using `tools restore`
- Connection and authentication when working with an S3-compatible object storage
- Exporting to an S3-compatible object storage `export s3`
- Importing from an S3-compatible object storage `import s3`
- Configuring NFS for backup
- Exporting to NFS `export nfs`
- Importing from NFS `import nfs`

File structure of an export

The file structure outlined below is used to export data both to the file system and an S3-compatible object storage. When working with S3, the file path is added to the object key, and the key's prefix specifies the export directory. If a file is encrypted, additional `.enc` extension is added.

Cluster

> **Note**
>
> A cluster can be exported only to the file system.

A cluster corresponds to a directory in the file structure, which contains:

- Directories describing databases in the cluster, except:
  - Database schema objects
  - Database users and groups that are not administrators
- The `permissions.pb` file, which describes the cluster root ACL and its owner in text protobuf format
- The `create_user.sql` file, which describes the cluster users in YQL format
- The `create_group.sql` file, which describes the cluster groups in YQL format
- The `alter_group.sql` file, which describes user membership in the cluster groups in YQL format

Database

> **Note**
>
> You can export only database schema objects to an S3-compatible object storage.

A database corresponds to a directory in the file structure, which contains:

- Directories describing the database schema objects, for example, tables
- The `database.pb` file, which describes the database settings in text protobuf format
- The `permissions.pb` file, which describes the cluster root ACL and its owner in text protobuf format
- The `create_user.sql` file, which describes the cluster users in YQL format
- The `create_group.sql` file, which describes the cluster groups in YQL format
- The `alter_group.sql` file, which describes user membership in the cluster groups in YQL format

Schema mapping

When exporting data to S3-compatible object storage with a common prefix, a `SchemaMapping/mapping.json` file is generated. This file contains a list of schema objects in the export, mapped to the paths in the S3 storage.

For encrypted exports, this mapping serves two purposes: it anonymizes the original object names and stores additional metadata required for decryption.

The example of `SchemaMapping/mapping.json` file content:

```
{
    "exportedObjects": {
        "dir/table1": {
            "exportPrefix": "dir/table1"
        },
        "table2": {
            "exportPrefix": "table2"
        }
    }
}
```

Directories

Each database directory has a corresponding directory in the file structure. Each of them includes a `permissions.pb` file, which describes the directory ACL and owner in the text protobuf format. The directory hierarchy in the file structure mirrors the hierarchy in the database. If a database directory contains no items (neither tables nor subdirectories), directory in the file structure includes an empty file named `empty_dir`.

Tables

For each table in the database, there's a same-name directory in the file structure's directory hierarchy that includes:

- The `scheme.pb` file describing the table structure and parameters in the text protobuf format
- The `permissions.pb` file describes the table ACL and owner in the text protobuf format
- One or more `data_XX.csv` files with the table data in `csv` format, where `XX` is the file's sequence number. The export starts with the `data_00.csv` file, with a next file created whenever the current file exceeds 100 MB
- Directories describing the changefeeds. Directory names match the names of the changefeeds. Each directory contains the following files:
  - The `changefeed_description.pb` file describing the changefeed in the text protobuf format
  - The `topic_description.pb` file describing the underlying topic in the text protobuf format

Topics

For each topic in the database, there's a same-name directory in the file structure's directory hierarchy that includes the `create_topic.pb` file. This file provides information about the topic parameters, partitioning settings, and consumers in the text protobuf format.

Files with data

The format of data files is `.csv`, where each row corresponds to a record in the table (except the row with column headings). The urlencoded format is used for rows. For example, the file row for the table with the uint64 and utf8 columns that includes the number 1 and the Russian string "Привет" (translates to English as "Hi"), would look like this:

```
1,"%D0%9F%D1%80%D0%B8%D0%B2%D0%B5%D1%82"
```

Checksums

> **Note**
>
> File checksums are only generated when exporting to S3-compatible object storage.

YDB generates a checksum for each export file and saves it to a corresponding file with the `.sha256` suffix.

The file checksum can be validated using the `sha256sum` console utility:

```
$ sha256sum -c scheme.pb.sha256
scheme.pb: OK
```

Examples

Cluster

When you export a cluster containing a database named `/Root/db1` and a database named `/Root/db2`, the system will create the following file structure:

```
backup
├─ Root
│  ├─ db1
│  │  ├─ alter_group.sql
│  │  ├─ create_group.sql
│  │  ├─ create_user.sql
│  │  ├─ permissions.pb
│  │  └─ database.pb
│  └─ db2
│     ├─ alter_group.sql
│     ├─ create_group.sql
│     ├─ create_user.sql
│     ├─ permissions.pb
│     └─ database.pb
├─ alter_group.sql
├─ create_group.sql
├─ create_user.sql
└─ permissions.pb
```

Database

When you export a database containing a table named `episodes`, the system will create the following file structure:

```
├─ episodes
│     ├─ data00.csv
│     ├─ scheme.pb
│     └─ permissions.pb
├─ alter_group.sql
├─ create_group.sql
├─ create_user.sql
├─ permissions.sql
└─ database.pb
```

Tables

When you export the tables created under YDB Quick Start in Getting started, the system will create the following file structure:

```
├── episodes
│    ├── data_00.csv
│    ├── permissions.pb
│    └── scheme.pb
├── seasons
│    ├── data_00.csv
│    ├── permissions.pb
│    └── scheme.pb
└── series
     ├── data_00.csv
     ├── permissions.pb
     └── scheme.pb
     └── updates_feed
          └── changefeed_description.pb
          └── topic_description.pb
```

Contents of the `series/scheme.pb` file:

```
columns {
  name: "series_id"
  type {
    optional_type {
      item {
        type_id: UINT64
      }
    }
  }
}
columns {
  name: "title"
  type {
    optional_type {
      item {
        type_id: UTF8
      }
    }
  }
}
columns {
  name: "series_info"
  type {
    optional_type {
      item {
        type_id: UTF8
      }
    }
  }
}
columns {
  name: "release_date"
  type {
    optional_type {
      item {
        type_id: UINT64
      }
    }
  }
}
primary_key: "series_id"
storage_settings {
  store_external_blobs: DISABLED
}
column_families {
  name: "default"
  compression: COMPRESSION_NONE
}
```

Contents of the `series/update_feed/topic_description.pb` file:

```
retention_period {
  seconds: 86400
}
consumers {
  name: "my_consumer"
  read_from {
  }
  attributes {
    key: "_service_type"
    value: "data-streams"
  }
}
```

Directories

When you export an empty directory `series`, the system will create the following file structure:

```
└── series
    ├── permissions.pb
    └── empty_dir
```

When you export a directory `series` with the nested table `episodes`, the system will create the following file structure:

```
└── series
    ├── permissions.pb
    └── episodes
        ├── data_00.csv
        ├── permissions.pb
        └── scheme.pb
```

Exporting data to the file system

Cluster

The `admin cluster dump` command dumps the cluster' metadata to the client file system in the format described in the File structure of an export article:

```
ydb [connection options] admin cluster dump [options]
```

where [connection options] are database connection options

`[options]` – command parameters:

- `-o <PATH>` or `--output <PATH>` : Path to the directory in the client file system where the data will be dumped.
  If the directory doesn't exist, it will be created. However, the entire path to the directory must already exist.

  If the specified directory exists, it must be empty.

  If the parameter is omitted, the `backup_YYYYDDMMTHHMMSS` directory will be created in the current directory, where `YYYYDDMM` is the date and `HHMMSS` is the time when the dump process began, accroding to the system clock.

A cluster configuration is dumped separately using the `ydb admin cluster config fetch` command.

Database

The `admin database dump` command dumps the database' data and metadata to the client file system in the format described in File structure of an export:

> **Warning**
>
> Currently, this command does not process column tables. To export column tables, use the `export s3` command.

> **Warning**
>
> Currently, this command does not process secrets. You must create secrets manually during restore before restoring objects that use them, such as data transfers.

```
ydb [connection options] admin database dump [options]
```

where [connection options] are database connection options

`[options]` – command parameters:

- `-o <PATH>` or `--output <PATH>` : Path to the directory in the client file system where the data will be dumped.
  If the directory doesn't exist, it will be created. However, the entire path to the directory must already exist.

  If the specified directory exists, it must be empty.

  If the parameter is omitted, the `backup_YYYYDDMMTHHMMSS` directory will be created in the current directory, where `YYYYDDMM` is the date and `HHMMSS` is the time when the dump process began, accroding to the system clock.

A database configuration is dumped separately using the `ydb admin database config fetch` command.

Schema objects

The `tools dump` command dumps the schema objects to the client file system in the format described in File structure of an export:

> **Warning**
>
> Currently, this command does not process column tables. To export column tables, use the `export s3` command.

> **Warning**
>
> Currently, this command does not process secrets. You must create secrets manually during restore before restoring objects that use them, such as data transfers.

```
ydb [connection options] tools dump [options]
```

where [connection options] are database connection options

`[options]` – command parameters:

- `-o <PATH>` or `--output <PATH>` : Path to the directory in the client file system where the data will be dumped.
  If the directory doesn't exist, it will be created. However, the entire path to the directory must already exist.

  If the specified directory exists, it must be empty.

  If the parameter is omitted, the `backup_YYYYDDMMTHHMMSS` directory will be created in the current directory, where `YYYYDDMM` is the date and `HHMMSS` is the time when the dump process began, accroding to the system clock.

- `-p <PATH>` or `--path <PATH>` : Path to the database directory with objects or a path to the table to be dumped. The root database directory is used by default. The dump includes all subdirectories whose names don't begin with a dot and the tables in them whose names don't begin with a dot. To dump such tables or the contents of such directories, you can specify their names explicitly in this parameter.

- `--exclude <STRING>`: Template (PCRE) to exclude paths from export. Specify this parameter multiple times to exclude more than one template simultaneously.
- `--scheme-only`: Dump only the details of the database schema objects without dumping their data.
- `--consistency-level <VAL>`: The consistency level. Possible options:
  - `database`: A fully consistent dump, with one snapshot taken before starting the dump. Applied by default.
  - `table`: Consistency within each dumped table, taking individual independent snapshots for each table. Might run faster and have less impact on the current workload processing in the database.
- `--avoid-copy`: Do not create a snapshot before dumping. The default consistency snapshot might be inapplicable in some cases (for example, for tables with external blobs).
- `--save-partial-result`: Retain the result of a partial dump. Without this option, dumps that terminate with an error are deleted.
- `--preserve-pool-kinds`: If enabled, the `tools dump` command saves the storage device types specified for column groups of the tables to the dump (see the `DATA` parameter in Column groups for reference). To import such a dump, the same storage pools must be present in the database. If at least one storage pool is missing, the import procedure will end with an error. By default, this option is disabled, and the import procedure uses the default storage pool specified at the time of database creation (see Create a Database for reference).
- `--ordered`: Sorts rows in the exported tables by the primary key.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Exporting a cluster

With automatic creation of the `backup_...` directory in the current directory:

```
ydb -e <endpoint> admin cluster dump
```

To a specific directory:

```
ydb -e <endpoint> admin cluster dump -o ~/backup_cluster
```

Exporting a database

To an automatically created `backup_...` directory in the current directory:

```
ydb -e <endpoint> -d <database> admin database dump
```

To a specific directory:

```
ydb -e <endpoint> -d <database> admin database dump -o ~/backup_db
```

Exporting a database schema objects

To an automatically created `backup_...` directory in the current directory:

```
ydb --profile quickstart tools dump
```

To a specific directory:

```
ydb --profile quickstart tools dump -o ~/backup_quickstart
```

Dumping the table structure within a specified database directory (including subdirectories)

```
ydb --profile quickstart tools dump -p dir1 --scheme-only
```

Importing data from the file system

Cluster

The `admin cluster restore` command restores a cluster from a backup on the file system. The backup must have been previously exported or prepared manually as described in the File structure of an export article:

```
ydb [connection options] admin cluster restore -i <PATH> [options]
```

where [connection options] are database connection options

The destination cluster must be running and initialized before it can be restored.

When restoring a cluster' metadata, databases and their administrators are created. Refer to Database for further details on restoring databases.

The restore operation requires that for each database to be restored, its database nodes must be available. You can start database nodes before running the restore command or while the restore operation is waiting for available nodes. If you encounter problems with available database nodes, you can restart the restore operation.

A cluster configuration is restored separately using the following steps:

1. Load the saved configuration using the `ydb admin cluster config replace` command.
2. Restart the cluster nodes.

Required parameters

`-i <PATH>` or `--input <PATH>`: Path to the directory in the client system from which the data will be imported.

Optional parameters

`[options]` – optional parameters of the command:

`--wait-nodes-duration <DURATION>`: The period of time that the restore command waits for available database nodes. Example: `10s`, `5m`, `1h`, `1.5d`, `30`. Duration can be expressed in weeks, days, hours, minutes, seconds, microseconds, nanoseconds. If no suffix is specified, the duration is seconds. The duration can be fractional. Combined duration like `1h30m` is not supported. If the duration is `0`, the restore command does not wait for available nodes.

Database

The `admin database restore` command restores the database from a backup on the file system. The backup must have been previously exported with the `admin database dump` command or prepared manually as described in the File structure of an export article:

> **Warning**
>
> Currently, this command does not restore column table data. To import column tables, use the `import s3` command with a backup created by `export s3`.

> **Warning**
>
> Currently, this command does not restore secrets (this data is not included in the backup). You must create secrets manually during restore before restoring objects that use them, such as data transfers.

```
ydb [connection options] admin database restore -i <PATH> [options]
```

where [connection options] are database connection options

The restore operation requires that for each database to be restored, its database nodes must be available. You can start database nodes before running the restore command or while the restore operation is waiting for available nodes. If you encounter problems with available database nodes, you can restart the restore operation.

Restoring database schema objects follows the same process described in Schema objects.

Database configuration is restored separately using the following steps:

1. Load the saved configuration using the `ydb admin database config replace` command.
2. Restart the database nodes.

Required parameters

`-i <PATH>` or `--input <PATH>`: Path to the directory in the client system from which the data will be imported.

Optional parameters

`[options]` – optional parameters of the command:

`--wait-nodes-duration <DURATION>`: The period of time that the restore command waits for available database nodes. Example: `10s`, `5m`, `1h`, `1.5d`, `30`. Duration can be expressed in weeks, days, hours, minutes, seconds, microseconds, nanoseconds. If no suffix is specified, the duration is seconds. The duration can be fractional. Combined duration like `1h30m` is not supported. If the duration is `0`, the restore command does not wait for available nodes.

Schema objets

The `tools restore` command creates the items of the database schema in the database, and populates them with the data previously exported there with the `tools dump` command or prepared manually as per the rules from the File structure of an export article:

Warning

Currently, this command does not restore column table data. To import column tables, use the `import s3` command with a backup created by `export s3`.

Warning

Currently, this command does not restore secrets (this data is not included in the backup). You must create secrets manually during restore before restoring objects that use them, such as data transfers.

```
ydb [connection options] tools restore -p <PATH> -i <PATH> [options]
```

where [connection options] are database connection options

If the table or directory already exists in the database, no changes will be made to its schema and ACL. If some columns present in the imported files are missing in the database or have mismatching types, this may lead to the data import operation failing.

To import data to the table, use the YQL `REPLACE` command. If the table included any records before the import, the records whose keys are present in the imported files are replaced by the data from the file. The records whose keys are absent in the imported files aren't affected.

Required parameters

- `-p <PATH>` or `--path <PATH>`: Path to the database directory the data will be imported to. To import data to the root directory, specify `.`. All the missing directories along the path will be created.
- `-i <PATH>` or `--input <PATH>`: Path to the directory in the client system from which the data will be imported.

Optional parameters

`[options]` – optional parameters of the command:

- `--restore-data <VAL>`: Enables/disables data import, 1 (yes) or 0 (no), defaults to 1. If set to 0, the import only creates items in the schema without populating them with data. If there's no data in the file system (only the schema has been exported), it doesn't make sense to change this option.
- `--restore-indexes <VAL>`: Enables/disables import of indexes, 1 (yes) or 0 (no), defaults to 1. If set to 0, the import won't either register indexes in the data schema or populate them with data.
- `--restore-acl <VAL>`: Enables/disables import of ACL, 1 (yes) or 0 (no), defaults to 1. If set to 0, the import creates items in the schema with an empty ACL, and their owner will be the user who started the import.
- `--replace-sys-acl <VAL>`: Enables/disables replacement of ACL for system objects, 1 (yes) or 0 (no), defaults to 1.
- `--dry-run`: Matching the data schemas in the database and file system without updating the database, 1 (yes) or 0 (no), defaults to 0. When enabled, the system checks that:
  - All tables in the file system are present in the database
  - These items are based on the same schema, both in the file system and in the database
- `--save-partial-result`: Save the partial import result. If disabled, an import error results in reverting to the database state before the import.
- `--import-data`: Use ImportData, a more efficient method for uploading data than the default approach. This method sends data to the server partitioned by the client and in a lighter format. However, it returns an error when attempting to import exported data into an existing table that already has indexes or is in the process of building them. To restore a table with indexes, ensure they are not already present in the schema (for example, using the `ydb scheme ls` command). By default, ImportData is disabled.
- `--replace`: Remove existing objects from the database that match those in the backup before restoration. Objects present in the backup but missing in the database are restored as usual; removal is skipped. If both `--replace` and `--verify-existence` are specified, restoration stops with an error when the first such object is found.
- `--verify-existence`: Use with `--replace` option to report an error if an object in the backup is missing from the database instead of silently skipping its removal.
- `--retries`: Number of retries for every upload data request. By default: `10`.

Workload restriction parameters

Using the below parameters, you can limit the import workload against the database.

Attention!

Some of the below parameters have default values. This means that the workload will be limited even if none of them is mentioned in `tools restore`.

- `--rps <VAL>`: Limits the number of queries used to upload batches to the database per second, the default value is 30.
- `--bandwidth <VAL>`: Limit the workload per second, defaults to 0 (not set). `<VAL>` specifies the data amount with a unit, for example, 2MiB. If this value is set, the `--rps` limit (see above) is not applied.
- `--in-flight <VAL>`: Limits the number of queries that can be run in parallel, the default value is 10. To achieve maximum parallelism, set the parameter value to the number of cores allocated for the restore process.
- `--upload-batch-rows <VAL>`: Limits the number of records in the uploaded batch, the default value is 0 (unlimited). `<VAL>` determines the number of records and is set as a number with an optional unit, for example, 1K.
- `--upload-batch-bytes <VAL>`: Limits the batch size of uploaded data, the default value is 512KB. `<VAL>` specifies the data amount with a unit, for example, 1MiB. Maximum value is 16 MiB.
- `--upload-batch-rus <VAL>`: Applies only to Serverless databases to limit Request Units (RU) that can be consumed to upload one batch, defaults to 30 RU. The batch size is selected to match the specified value. `<VAL>` determines the number of RU and is set as a number with an optional unit, for example, 100 or 1K.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Restoring cluster

From the current file system directory:

```
ydb -e <endpoint> admin cluster restore -i .
```

From the specified file system directory:

```
ydb -e <endpoint> admin cluster restore -i ~/backup_cluster
```

Restoring database

From the current file system directory:

```
ydb -e <endpoint> -d <database> admin database restore -i .
```

From the specified file system directory:

```
ydb -e <endpoint> -d <database> admin database restore -i ~/backup_db
```

Importing schema objets to the database root

From the current file system directory:

```
ydb -p quickstart tools restore -p . -i .
```

From the current file system directory:

```
ydb -p quickstart tools restore -p . -i ~/backup_quickstart
```

Uploading data to the specified directory in the database

From the current file system directory:

```
ydb -p quickstart tools restore -p dir1/dir2 -i .
```

From the current file system directory:

```
ydb -p quickstart tools restore -p dir1/dir2 -i ~/backup_quickstart
```

Matching schemas between the database and file system:

```
ydb -p quickstart tools restore -p dir1/dir2 -i ~/backup_quickstart --dry-run
```

Example options for better performance

```
ydb -p quickstart tools restore -p . -i . --import-data --bandwidth=10GiB --in-flight=16 --upload-batch-bytes
=16MiB
```

Connecting to S3-compatible object storages

The commands used to export data from S3-compatible storages, `export s3` and `import s3`, use the same parameters for S3 connection and authentication. To learn how to get these parameters for certain cloud providers, see Getting S3 connection parameters below.

Connecting

To connect with S3, you need to specify an endpoint and bucket:

`--s3-endpoint HOST`: An S3 endpoint. `HOST`: A valid host name, such as: `storage.yandexcloud.net`

`--bucket STR`: An S3 bucket. `STR`: A string containing the bucket name

Authentication

Except when you import data from a public bucket, to connect, log in with an account that has write access to the bucket (for export to it) and read access to the bucket (for import from it).

You need two parameters to authenticate with S3:

- Access key ID (`--access-key`).
- Secret access key (`--secret-key`).

The YDB CLI takes values of these parameters from the following sources (listed in descending priority):

1. The command line.
2. Environment variables.
3. The `~/.aws/credentials` file.

Command line parameters

- `--access-key`: Access key ID.
- `--secret-key`: Secret access key.
- `--aws-profile`: Profile name from the `~/.aws/credentials` file. The default value is `default`.

Environment variables

If a certain authentication parameter is omitted in the command line, the YDB CLI tries to retrieve it from the following environment variables:

- `AWS_ACCESS_KEY_ID`: Access key ID.
- `AWS_SECRET_ACCESS_KEY`: Secret access key.
- `AWS_PROFILE`: Profile name from the `~/.aws/credentials` file.

AWS authentication file

If a certain authentication parameter is omitted in the command line and cannot be retrieved from an environment variable, the YDB CLI tries to get it from the specified profile or the default profile in the `~/.aws/credentials` file used for authenticating the AWS CLI. You can create this file with the `aws configure` AWS CLI command.

Getting the S3 connection parameters

Yandex.Cloud

Below is an example of getting access keys for the Yandex.Cloud Object Storage using the Yandex.Cloud CLI.

1. Install and set up the Yandex.Cloud CLI.
2. Use the following command to get the ID of your cloud folder (`folder-id`) (you'll need to add it to the commands below):

   ```
   yc config list
   ```

   The ID of your cloud folder is in the `folder-id:` line in the result:

   ```
   folder-id: b2ge70qdcff4bo9q6t19
   ```

3. To create a service account, run the command:

   ```
   yc iam service-account create --name s3account
   ```

   You can indicate any account name instead of `s3account`, or use your existing account name (be sure to replace it when copying the commands below).

   Account id will be printed on creation.

   To get the id of an existing account, use this command:

   ```
   yc iam service-account get --name <account-name>
   ```

4. Grant roles to your service account according to your intended S3 access level by running the command:

   Read (to import data to the YDB database)

   ```
   yc resource-manager folder add-access-binding <folder-id> \
     --role storage.viewer --subject serviceAccount:<s3-account-id>
   ```

Write (to export data from the YDB database)

```
yc resource-manager folder add-access-binding <folder-id> \
  --role storage.editor --subject serviceAccount:<s3-account-id>
```

Where `<folder-id>` is the cloud folder ID that you retrieved at step 2 and `<s3-account-id>` is the id of the account you created at step 3.

You can also read a full list of Yandex.Cloud roles.

5. Get static access keys by running the command:

```
yc iam access-key create --service-account-name s3account
```

If successful, the command will return the access_key attributes and the secret value:

```
access_key:
  id: aje6t3vsbj8lp9r4vk2u
  service_account_id: ajepg0mjt06siuj65usm
  created_at: "2018-11-22T14:37:51Z"
  key_id: 0n8X6WY6S24N7OjXQ0YQ
secret: JyTRFdqw8t1kh2-OJNz4JX5ZTz9Dj1rI9hxtzMP1
```

In this result:

- `access_key.key_id` is the access key ID (`--access-key`).
- `secret` is the secret access key (`--secret-key`).

Export to S3-compatible storage

The `export s3` command starts a server-side process of exporting data and schema object information to an S3-compatible storage in the format described in the article File structure:

```
ydb [connection options] export s3 [options]
```

where [connection options] are database connection options

Warning

Export is available only for objects of the following types:

- directory.
- row table.
- column table.
- secondary index.
- vector index.
- full-text index.
- topic (schema only).
- view.
- asynchronous replication.
- transfer.
- external data source.
- external table.

Note

As an alternative, you can export individual tables to an S3-compatible data storage using external data sources. For more details, see the article Exporting data to S3 object storage.

Command line parameters

`[options]` — command parameters:

S3 parameters

The export to S3 command requires specifying S3 connection parameters. Since the export is performed asynchronously by the YDB server, the specified endpoint must be accessible for establishing a connection from the server side.

`--destination-prefix PREFIX`: Key prefix in the S3 bucket.

List of exported objects

`--root-path PATH`: Root directory for exported objects. If not specified, the root directory of the database is used.

`--include PATH`: Data schema objects to include in the export. Directories are traversed recursively. Paths are specified relative to `root-path`. This parameter can be specified multiple times to include multiple objects. If not specified, all non-system objects in `root-path` are exported.

`--exclude STRING`: Pattern (PCRE) to exclude paths from the export. Paths are specified relative to `root-path`. This parameter can be specified multiple times for different patterns.

Alternative method

An alternative way to specify the list of objects is supported:

`--item STRING`: Description of the export object. The `--item` parameter can be specified multiple times if you need to export several objects. `STRING` is specified in the `<property>=<value>,...` format, with the following required properties:

- `source`, `src`, or `s` — path to the exported directory or table; `.` points to the root directory of the database. When specifying a directory, all non-system objects in it are exported, as well as all non-system subdirectories recursively.
- `destination`, `dst`, or `d` — path (key prefix) in S3 for placing the exported objects.

`--exclude STRING`: Pattern (PCRE) to exclude paths from export. This parameter can be specified multiple times for different patterns.

Warning

Exports made using alternative syntax will not contain a list of objects in the backup, so some features may be unavailable for them (in particular, encrypted backups), and import is only possible using the corresponding alternative import syntax.

Additional parameters

Parameter	Description
`--description STRING`	Text description of the operation, saved in the operation history.
`--retries NUM`	Number of retry attempts for export that the server will make. Default value: `10`.

`--compression STRING`	Compress exported data. With the default compression level for the Zstandard algorithm, data can be compressed by 5-10 times. Data compression uses CPU resources and may affect the speed of other database operations. Allowed values: • `zstd` — compression using the Zstandard algorithm with the default compression level (`3`); • `zstd-N` — compression using the Zstandard algorithm, `N` — compression level (`1` — `22`).
`--encryption-algorithm ALGORITHM`	Encrypt exported data using the specified algorithm. Supported values: `AES-128-GCM` , `AES-256-GCM` , `ChaCha20-Poly1305` .
`--encryption-key-file PATH`	Path to the file containing the encryption key (only for encrypted exports). This file is binary and must contain the exact number of bytes corresponding to the key length in the selected encryption algorithm (16 bytes for `AES-128-GCM` , 32 bytes for `AES-256-GCM` and `ChaCha20-Poly1305`). The key can also be passed via the `YDB_ENCRYPTION_KEY` environment variable, in hexadecimal string representation.
`--format STRING`	Output format. Allowed values: • `pretty` — human-readable format (default); • `proto-json-base64` — Protocol Buffers in JSON format, binary strings encoded in Base64.

Running the export

Server export operation flow

1. A server asynchronous export operation is created.
2. In the root directory of the database, a service directory `export-{id}` is created, where `{id}` is the numeric operation ID.
3. A consistent copy of the tables is created in this directory using the `CopyTables` mechanism.
4. The data of each table is written from the copy to the storage in the export file structure format, with parallel writing from different cluster nodes.
5. After successful completion, the service directory `export-{id}` and the table copies are deleted.

Launch result

Upon successful execution, the `export s3` command outputs summary information about the queued export operation to S3, in the format specified by the `--format` option. The actual export is performed asynchronously by the server. The summary information includes the operation ID, which can be used later to check the status and perform actions on the operation:

- In the `pretty` output mode (default), the operation ID is shown in the id field highlighted with pseudographics:

```
┌──────────────────────────────────────────────┬───────┬─────...
| id                                           | ready | stat...
├──────────────────────────────────────────────┼───────┼─────...
| ydb://export/6?id=281474976788395&kind=s3 | true  | SUCC...
├──────────────────────────────────────────────┴───────┴────...
| StorageClass: NOT_SET
| Items:
...
```

- In the `proto-json-base64` output mode, the identifier is in the "id" attribute:

```
{"id":"ydb://export/6?id=281474976788395&kind=s3","ready":true, ... }
```

Export status

Data export runs in the background. You can get information about the status and progress of the export by calling the `operation get` command, which must be passed the **quoted** operation ID, for example:

```
ydb -p quickstart operation get "ydb://export/6?id=281474976788395&kind=s3"
```

The output format `operation get` is also set by the `--format` option.

Although the operation ID is in URL format, it is not guaranteed to be preserved in the future. It should be interpreted only as a string.

Completion of the export is tracked by changes in the "progress" attribute:

- In the `pretty` output mode (default), a successfully completed operation is reflected by the value "Done" in the `progress` field highlighted with pseudographics:

```
┌─────── ... ──┬───────┬─────────┬──────────┬──...
| id          | ready | status  | progress | ...
├───────── ... ──┼───────┼─────────┼──────────┼──...
| ydb://...   | true  | SUCCESS | Done     | ...
├─────── ... ──┴───────┴─────────┴──────────┴──...
...
```

- In the `proto-json-base64` output mode, a completed operation is reflected by the value `PROGRESS_DONE` of the `progress` attribute:

```
{"id":"ydb://...", ...,"progress":"PROGRESS_DONE",... }
```

Completing the export operation

After completing the export, use the `operation forget` command to mark the export as completed (removed from the list of operations):

```
ydb -p quickstart operation forget "ydb://export/6?id=2814749767883 95&kind=s3"
```

List of export operations

To get the list of export operations, use the `operation list export/s3` command:

```
ydb -p quickstart operation list export/s3
```

The output format `operation list` is also set by the `--format` option.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Exporting a database

Export all non-system database objects to the `export1` directory in the `mybucket` bucket using S3 authentication parameters from environment variables or the `~/.aws/credentials` file:

```
ydb -p quickstart export s3 \
  --s3-endpoint storage.yandexcloud.net --bucket mybucket \
  --destination-prefix export1
```

Exporting multiple directories

Export objects from the `dir1` and `dir2` directories of the database to the `export1` directory in the `mybucket` bucket, using explicitly specified S3 authentication parameters:

```
ydb -p quickstart export s3 \
  --s3-endpoint storage.yandexcloud.net --bucket mybucket \
  --access-key <access-key> --secret-key <secret-key> \
  --destination-prefix export1 --include dir1 --include dir2
```

Or using an alternative method:

```
ydb -p quickstart export s3 \
  --s3-endpoint storage.yandexcloud.net --bucket mybucket \
  --access-key <access-key> --secret-key <secret-key> \
  --item src=dir1,dst=export1/dir1 --item src=dir2,dst=export1/dir2
```

Export with encryption

Export the entire database with encryption:

- Using the `AES-128-GCM` encryption algorithm
- With generation of a random key by the `openssl` utility into the `~/my_secret_key` file
- With reading the generated key from the `~/my_secret_key` file
- To the path prefix `export1` in the S3 bucket `mybucket`
- Using S3 authentication parameters from environment variables or the `~/.aws/credentials` file

```
openssl rand -out ~/my_secret_key 16
ydb -p quickstart export s3 \
  --s3-endpoint storage.yandexcloud.net --bucket mybucket --destination-prefix export1 \
  --encryption-algorithm AES-128-GCM --encryption-key-file ~/my_secret_key
```

Exporting the `dir1` directory of the database with encryption:

- Using the `AES-256-GCM` encryption algorithm
- By generating a random key with the `openssl` utility into the `YDB_ENCRYPTION_KEY` environment variable
- By reading the generated key from the `YDB_ENCRYPTION_KEY` environment variable
- Into the path prefix `export1` in the S3 bucket `mybucket`
- Using S3 authentication parameters from environment variables or the `~/.aws/credentials` file

```
export YDB_ENCRYPTION_KEY=$(openssl rand -hex 32)
ydb -p quickstart export s3 \
  --root-path dir1 \
  --s3-endpoint storage.yandexcloud.net --bucket mybucket --destination-prefix export1 \
  --encryption-algorithm AES-256-GCM
```

Getting operation IDs

To get a list of export operation IDs in a format convenient for processing in bash scripts, you can use the jq utility:

```
ydb -p quickstart operation list export/s3 --format proto-json-base64 | jq -r ".operations[].id"
```

You will get output where each new line contains an operation ID, for example:

```
ydb://export/6?id=281474976789577&kind=s3
ydb://export/6?id=281474976789526&kind=s3
ydb://export/6?id=281474976788779&kind=s3
```

Using these IDs, you can, for example, run a loop to terminate all current operations:

```
ydb -p quickstart operation list export/s3 --format proto-json-base64 | jq -r ".operations[].id" | while read
line; do ydb -p quickstart operation forget $line;done
```

Importing data from an S3 compatible storage

The `import s3` command starts, on the server side, the process of importing data and schema objects details from an S3-compatible storage, in the format described in the File structure section:

```
ydb [connection options] import s3 [options]
```

where [connection options] are database connection options

As opposed to the `tools restore` command, the `import s3` command always creates objects in entirety, so none of the imported objects (directories or tables) should already exist.

If you need to load additional data into existing tables from S3, you can copy the S3 contents to the file system (for example, using S3cmd) and use the `tools restore` command.

> **Warning**
>
> The `tools restore` command loads data only into row tables. To append data to existing column tables, use external data sources. For more information, see Data import.

Command line parameters

`[options]`: Command parameters:

S3 parameters

To run the command to import data from an S3 storage, specify the S3 connection parameters. As data is imported by the YDB server asynchronously, the specified endpoint must be available so that a connection can be established from the server side.

`--source-prefix PREFIX`: Source prefix for export in the bucket.

Imported schema objects

`--destination-path PATH`: Destination folder for the objects being imported; defaults to the database root if not provided.

`--include PATH`: Schema objects to be included in the import. Directories are traversed recursively. You may specify this parameter multiple times to include several objects. If not specified, all objects in export are imported.

`--exclude STRING`: Template (PCRE) to exclude paths from import. Paths are relative to the `root-path`. You may specify this parameter multiple times for different templates.

Alternate syntax

There's an alternate syntax to specify the list of imported objects, supported for backward compatibility.

`--item STRING`: Description of the item to import. You can specify the `--item` parameter multiple times to import multiple items. If no `--item` or `--include` parameters are specified, all objects from the source prefix will be imported. `STRING` is specified in the `<property>=<value>,...` format with the following properties:

- `source`, `src`, or `s` is the key prefix in S3 that contains the imported directory or table.
- `destination`, `dst`, or `d` is the database path to host the imported directory or table. The destination of the path must not exist. All the directories along the path will be created if missing.

Some features may not be available using the alternate syntax (like encryption and listing).

Additional parameters

Parameter	Description
`--description STRING`	A text description of the operation saved in the operation history.
`--retries NUM`	The number of import retries to be made by the server. The default value is 10.
`--skip-checksum-validation`	Skip the validating imported objects' checksums step.
`--encryption-key-file PATH`	File path containing the encryption key (only for encrypted exports). The file is binary and must contain exactly the number of bytes matching the key length for the chosen encryption algorithm (16 bytes for `AES-128-GCM`, 32 bytes for `AES-256-GCM` and `ChaCha20-Poly1305`). The key can also be provided using the `YDB_ENCRYPTION_KEY` environment variable, in hexadecimal string representation.
`--list`	List objects in an existing export.
`--format STRING`	Result format. Possible values: • `pretty`: Human-readable format (default). • `proto-json-base64`: Protocol Buffers in JSON format, binary strings are Base64-encoded.

Importing

Import result

If successful, the `import s3` command prints summary information about the enqueued operation to import data from S3 in the format specified in the `--format` option. The import itself is performed by the server asynchronously. The summary shows the operation ID that you can use later to check the operation status and perform actions on it:

- In the default `pretty` mode, the operation ID is displayed in the id field with semigraphics formatting:

```
┌──────────────────────────────────────────┬───────┬─────...
| id                                       | ready | stat...
├──────────────────────────────────────────┼───────┼─────...
| ydb://import/8?id=281474976788395&kind=s3 | true  | SUCC...
├------------------------------------------┴-------┴-----...
| Items:
...
```

- In the `proto-json-base64` mode, the operation ID is in the "id" attribute:

```
{"id":"ydb://export/8?id=281474976788395&kind=s3","ready":true, ... }
```

Import status

Data is imported in the background. To get information on import status, use the `operation get` command with the operation ID **enclosed in quotation marks** and passed as a command parameter. For example:

```
ydb -p quickstart operation get "ydb://import/8?id=281474976788395&kind=s3"
```

The `operation get` format is also set by the `--format` option.

Although the operation ID is in URL format, there is no guarantee that it is maintained in the future. It should only be interpreted as a string.

You can track the import by changes in the "progress" attribute:

- In the default `pretty` mode, successfully completed export operations are displayed as "Done" in the `progress` field with semigraphics formatting:

```
┌──────── ... ──┬─────────┬──────────┬────────────┬...
| id            | ready | status   | progress | ...
├──────────... ──┼─────────┼──────────┼────────────┼...
| ydb:/...      | true  | SUCCESS | Done       | ...
├------ ... ---┴-------┴---------┴----------┴ ...
...
```

- In the `proto-json-base64` mode, the completed export operation is indicated with the `PROGRESS_DONE` value of the `progress` attribute:

```
{"id":"ydb://...", ...,"progress":"PROGRESS_DONE",... }
```

Completing the import operation

When the import is complete, use `operation forget` to delete the import from the operation list:

```
ydb -p quickstart operation forget "ydb://import/8?id=281474976788395&kind=s3"
```

List of import operations

To get a list of import operations, run the `operation list import/s3` command:

```
ydb -p quickstart operation list import/s3
```

The `operation list` format is also set by the `--format` option.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Importing to the database root

Importing to the database root the contents of the `export1` directory in the `mybucket` bucket using the S3 authentication parameters taken from the environment variables or the `~/.aws/credentials` file:

```
ydb -p quickstart import s3 \
  --s3-endpoint storage.yandexcloud.net --bucket mybucket \
  --source-prefix export1
```

Importing multiple directories

Importing items from the `dir1` and `dir2` directories of an export located in `export1` in the `mybucket` S3 bucket to the same-name database directories using explicitly specified S3 authentication parameters:

```
ydb -p quickstart import s3 \
  --s3-endpoint storage.yandexcloud.net --bucket mybucket \
  --access-key <access-key> --secret-key <secret-key> \
  --source-prefix export1
  --include dir1 --include dir2
```

List objects in existing encrypted export

Listing all object paths in existing encrypted export located in `export1` in the `mybucket` S3 bucket, using the secret key stored in the `~/my_secret_key` file.

```
ydb -p quickstart import s3 \
  --s3-endpoint storage.yandexcloud.net --bucket mybucket \
  --access-key <access-key> --secret-key <secret-key> \
  --source-prefix export1
  --encryption-key-file ~/my_secret_key
  --list
```

Importing encrypted export

Importing one table that was exported using the `dir/my_table` path to the `dir1/dir/my_table` path from an encrypted export located in `export1` in the `mybucket` S3 bucket, using the secret key stored in the `~/my_secret_key` file.

```
ydb -p quickstart import s3 \
  --s3-endpoint storage.yandexcloud.net --bucket mybucket \
  --access-key <access-key> --secret-key <secret-key> \
  --source-prefix export1 --destination-path dir1 \
  --include dir/my_table \
  --encryption-key-file ~/my_secret_key
```

Getting operation IDs

To get a list of import operation IDs in a bash-friendly format, use the jq utility:

```
ydb -p quickstart operation list import/s3 --format proto-json-base64 | jq -r ".operations[].id"
```

You'll get a result where each new line shows an operation's ID. For example:

```
ydb://import/8?id=281474976789577&kind=s3
ydb://import/8?id=281474976789526&kind=s3
ydb://import/8?id=281474976788779&kind=s3
```

You can use these IDs, for example, to run a loop to end all the current operations:

```
ydb -p quickstart operation list import/s3 --format proto-json-base64 | jq -r ".operations[].id" | while read
line; do ydb -p quickstart operation forget $line;done
```

Export to NFS

The `export nfs` command starts a server-side export process to the network file system (Network File System, NFS) of the YDB cluster hosts, exporting data and schema object information in the format described in the File structure article:

```
ydb [connection options] export nfs [options]
```

where [connection options] are database connection options

Warning

Export is available only for objects of the following types:

- directory.
- row table.
- secondary index.
- vector index.
- full-text index.
- local Bloom skip index.
- topic (schema only).
- view.
- asynchronous replication.
- transfer.
- external data source.
- external table.

Warning

Currently, this command does not process column tables. To export column tables, use the `export s3` command.

Command line parameters

`[options]` — command parameters:

NFS parameters

The NFS export command requires specifying a mounted directory (or subdirectory) common to all objects involved in the export. Since the export is performed asynchronously on all YDB hosts, the specified directory must exist on each YDB host and be mounted in NFS.

`--fs-path PATH` : path to the mounted directory (or subdirectory).

List of exported objects

`--root-path PATH` : Root directory for exported objects. If not specified, the root directory of the database is used.

`--include PATH` : Data schema objects to include in the export. Directories are traversed recursively. Paths are specified relative to `root-path` . This parameter can be specified multiple times to include multiple objects. If not specified, all non-system objects in `root-path` are exported.

`--exclude STRING` : Pattern (PCRE) to exclude paths from the export. Paths are specified relative to `root-path` . This parameter can be specified multiple times for different patterns.

Alternative method

An alternative way to specify the list of objects is supported:

`--item STRING` : Description of the export object. The `--item` parameter can be specified multiple times if you need to export several objects. `STRING` is specified in the `<property>=<value>,...` format, with the following required properties:

- `source` , `src` , or `s` — path to the exported directory or table; `.` points to the root directory of the database. When specifying a directory, all non-system objects in it are exported, as well as all non-system subdirectories recursively.
- `destination` , `dst` , or `d` — path in NFS (relative to `--fs-path`).

`--exclude STRING` : Pattern (PCRE) to exclude paths from export. This parameter can be specified multiple times for different patterns.

Warning

Exports made using alternative syntax will not contain a list of objects in the backup, so some features may be unavailable for them (in particular, encrypted backups), and import is only possible using the corresponding alternative import syntax.

Additional parameters

Parameter	Description
`--description STRING`	Text description of the operation, saved in the operation history.
`--retries NUM`	Number of retry attempts for export that the server will make. Default value: `10` .

`--compression STRING`	Compress exported data. With the default compression level for the Zstandard algorithm, data can be compressed by 5-10 times. Data compression uses CPU resources and may affect the speed of other database operations. Allowed values: • `zstd` — compression using the Zstandard algorithm with the default compression level (`3`); • `zstd-N` — compression using the Zstandard algorithm, `N` — compression level (`1` — `22`).
`--encryption-algorithm ALGORITHM`	Encrypt exported data using the specified algorithm. Supported values: `AES-128-GCM` , `AES-256-GCM` , `ChaCha20-Poly1305` .
`--encryption-key-file PATH`	Path to the file containing the encryption key (only for encrypted exports). This file is binary and must contain the exact number of bytes corresponding to the key length in the selected encryption algorithm (16 bytes for `AES-128-GCM` , 32 bytes for `AES-256-GCM` and `ChaCha20-Poly1305`). The key can also be passed via the `YDB_ENCRYPTION_KEY` environment variable, in hexadecimal string representation.
`--format STRING`	Output format. Allowed values: • `pretty` — human-readable format (default); • `proto-json-base64` — Protocol Buffers in JSON format, binary strings encoded in Base64.

Running the export

Server export operation flow

1. A server asynchronous export operation is created.
2. In the root directory of the database, a service directory `export-{id}` is created, where `{id}` is the numeric operation ID.
3. A consistent copy of the tables is created in this directory using the `CopyTables` mechanism.
4. The data of each table is written from the copy to the storage in the export file structure format, with parallel writing from different cluster nodes.
5. After successful completion, the service directory `export-{id}` and the table copies are deleted.

Launch result

Upon successful execution, the `export nfs` command outputs summary information about the queued NFS export operation, in the format specified by the `--format` option. The actual export is performed asynchronously by the server. The summary information includes the operation ID, which can be used later to check the status and perform actions on the operation:

- In the `pretty` output mode (default), the operation ID is shown in the id field highlighted with pseudographics:

```
┌──────────────────────────────────────────────┬───────┬─────...
| id                                           | ready | stat...
├──────────────────────────────────────────────┼───────┼─────...
| ydb://export/6?id=281474976788395&kind=fs | true  | SUCC...
├----------------------------------------------┴-------┴-----...
| Include index data: false
| Items:
...
```

- In the `proto-json-base64` output mode, the identifier is in the "id" attribute:

```
{"id":"ydb://export/6?id=281474976788395&kind=fs","ready":true, ... }
```

Export status

Data export runs in the background. You can get information about the status and progress of the export by calling the `operation get` command, which must be passed the **quoted** operation ID, for example:

```
ydb -p quickstart operation get "ydb://export/6?id=281474976788395&kind=fs"
```

The output format `operation get` is also set by the `--format` option.

Although the operation ID is in URL format, it is not guaranteed to be preserved in the future. It should be interpreted only as a string.

Completion of the export is tracked by changes in the "progress" attribute:

- In the `pretty` output mode (default), a successfully completed operation is reflected by the value "Done" in the `progress` field highlighted with pseudographics:

```
┌──────  ...  ──┬───────┬──────────┬────────────┬...
| id            | ready | status   | progress   | ...
├──────...  ──┼───────┼──────────┼────────────┼...
| ydb://...     | true  | SUCCESS | Done       | ...
├-----  ...  --┴-------┴---------┴------------┴ ...
...
```

- In the `proto-json-base64` output mode, a completed operation is reflected by the value `PROGRESS_DONE` of the `progress` attribute:

```
{"id":"ydb://...", ...,"progress":"PROGRESS_DONE",... }
```

Completing the export operation

After completing the export, use the `operation forget` command to mark the export as completed (removed from the list of operations):

```
ydb -p quickstart operation forget "ydb://export/6?id=281474976788395&kind=fs"
```

List of export operations

To get the list of export operations, use the `operation list export/nfs` command:

```
ydb -p quickstart operation list export/nfs
```

The output format `operation list` is also set by the `--format` option.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Exporting a database

Export all non-system database objects to the `/mnt/nfs/backups/export1` directory on the file system:

```
ydb -p quickstart export nfs \
  --fs-path /mnt/nfs/backups/export1
```

Exporting multiple directories

Export objects from the `dir1` and `dir2` directories of the database to the `/mnt/nfs/backups/export1` directory on the file system:

```
ydb -p quickstart export nfs \
  --fs-path /mnt/nfs/backups/export1 \
  --include dir1 --include dir2
```

Or using an alternative method:

```
ydb -p quickstart export nfs \
  --fs-path /mnt/nfs/backups \
  --item src=dir1,dst=export1/dir1 --item src=dir2,dst=export1/dir2
```

Export with encryption

Export the entire database with encryption:

- Using the `AES-128-GCM` encryption algorithm
- With generation of a random key by the `openssl` utility into the `~/my_secret_key` file
- With reading the generated key from the `~/my_secret_key` file
- To the `/mnt/nfs/backups/export1` directory on the file system

```
openssl rand -out ~/my_secret_key 16
ydb -p quickstart export nfs \
  --fs-path /mnt/nfs/backups/export1 \
  --encryption-algorithm AES-128-GCM --encryption-key-file ~/my_secret_key
```

Exporting the `dir1` directory of the database with encryption:

- Using the `AES-256-GCM` encryption algorithm
- By generating a random key with the `openssl` utility into the `YDB_ENCRYPTION_KEY` environment variable
- By reading the generated key from the `YDB_ENCRYPTION_KEY` environment variable
- To the `/mnt/nfs/backups/export1` directory on the file system

```
export YDB_ENCRYPTION_KEY=$(openssl rand -hex 32)
ydb -p quickstart export nfs \
  --root-path dir1 \
  --fs-path /mnt/nfs/backups/export1 \
  --encryption-algorithm AES-256-GCM
```

Getting operation IDs

To get a list of export operation IDs in a format convenient for processing in bash scripts, you can use the jq utility:

```
ydb -p quickstart operation list export/nfs --format proto-json-base64 | jq -r ".operations[].id"
```

You will get output where each new line contains an operation ID, for example:

```
ydb://export/6?id=281474976789577&kind=fs
ydb://export/6?id=281474976789526&kind=fs
ydb://export/6?id=281474976788779&kind=fs
```

Using these IDs, you can, for example, run a loop to terminate all current operations:

```
ydb -p quickstart operation list export/nfs --format proto-json-base64 | jq -r ".operations[].id" | while rea
d line; do ydb -p quickstart operation forget $line;done
```

Import from NFS

The `import nfs` command starts a server-side import process from the network file system (Network File System, NFS) of the YDB cluster hosts, importing data and schema object information in the format described in the File structure article:

```
ydb [connection options] import nfs [options]
```

where [connection options] are database connection options

> **Warning**
>
> Currently, this command does not restore column table data. To import column tables, use the `import s3` command with a backup created by `export s3`.

Unlike the `tools restore` command, the `import nfs` command always creates objects entirely, so for it to succeed, none of the imported objects (neither directories nor tables) must exist.

If you need to load additional data into existing tables, use the `tools restore` command directly on the mounted NFS directory.

Command line parameters

`[options]` — command parameters:

NFS parameters

The import from NFS command requires specifying a mounted directory (or subdirectory) common to all objects involved in the import. Since the import is performed asynchronously on all YDB hosts, the specified directory must be present on each YDB host and mounted in NFS.

`--fs-path PATH`: path to the mounted directory (or subdirectory).

Imported database schema objects

`--destination-path PATH`: Target directory for imported objects; the default value is the database root.

`--include PATH`: Data schema objects to include in the import. Directories are traversed recursively. To include multiple objects, the parameter can be specified multiple times. If not specified, all exported objects are imported.

`--exclude STRING`: Pattern (PCRE) to exclude paths from the import. This parameter can be specified multiple times for different patterns.

Alternative method

For backward compatibility, an alternative way to specify the list of objects is supported:

`--item STRING`: Description of the object to load. The `--item` parameter can be specified multiple times if you need to load several objects. If the `--item` or `--include` parameters are not specified, all objects present in the specified dump will be loaded. `STRING` is specified in the `<property>=<value>,...` format with the following required properties:

- `source`, `src`, or `s` — path in NFS (relative to `fs-path`) with the imported directory or table.
- `destination`, `dst`, or `d` — the path in the database for placing the imported directory or table. The final path element must not exist. All directories on the path will be created if they do not exist.

Some features may be unavailable when using alternative syntax (in particular, encrypted backups or listing of export objects).

Additional parameters

Parameter	Description
`--description STRING`	Text description of the operation, saved in the operation history.
`--retries NUM`	Number of retry attempts the server will make. Default value: `10`.
`--skip-checksum-validation`	Skip the validation stage of checksums of imported objects.
`--encryption-key-file PATH`	Path to the file containing the encryption key (only for encrypted exports). This file is binary and must contain the exact number of bytes corresponding to the key length in the selected encryption algorithm (16 bytes for `AES-128-GCM`, 32 bytes for `AES-256-GCM` and `ChaCha20-Poly1305`). The key can also be passed via the `YDB_ENCRYPTION_KEY` environment variable, in hexadecimal string representation.
`--format STRING`	Result output format. Valid values: • `pretty` — human-readable format (default); • `proto-json-base64` — Protocol Buffers in JSON format, binary strings encoded in Base64.

Running the import

Server import operation flow

1. A server asynchronous import operation is created.
2. The server reads object metadata from files (`scheme.pb`, `metadata.json`, etc.) from the storage.
3. For each object, a new table is created in the database. Target paths **must not exist** — import cannot overwrite existing tables.
4. Data is loaded from files in parallel on all cluster nodes.

Launch result

Upon successful execution, the `import nfs` command outputs summary information about the queued import from NFS operation, in the format specified by the `--format` option. The actual import is performed asynchronously by the server. The summary information includes the operation ID, which can later be used to check the status and perform actions on the operation:

- In the `pretty` output mode (default), the operation ID is shown in the id field highlighted with pseudographics:

```
┌──────────────────────────────────────────────┬──────┬────...
| id                                           | ready | stat...
├──────────────────────────────────────────────┼──────┼────...
| ydb://import/8?id=281474976788395&kind=fs | true  | SUCC...
├-----------------------------------------------┴------┴----...
| Items:
...
```

- In the `proto-json-base64` output mode, the identifier is in the "id" attribute:

```
{"id":"ydb://import/8?id=281474976788395&kind=fs","ready":true, ... }
```

Import status

Data loading is performed in the background. You can get information about the status and progress of the loading by calling the `operation get` command, whose parameter must be the **quoted** operation identifier, for example:

```
ydb -p quickstart operation get "ydb://import/8?id=281474976788395&kind=fs"
```

The output format `operation get` is also set by the `--format` option.

Although the operation ID is in URL format, it is not guaranteed to be preserved in the future. It should be interpreted only as a string.

The completion of the upload is tracked by the change of the "progress" attribute:

- In the `pretty` output mode (default), a successfully completed operation is reflected by the value "Done" in the `progress` field highlighted with pseudographics:

```
┌─────── ... ──┬───────┬─────────┬──────────┬──...
| id          | ready | status  | progress | ...
├──────── ... ──┼───────┼─────────┼──────────┼──...
| ydb://...   | true  | SUCCESS | Done     | ...
├----- ... --┴-------┴---------┴----------┴ ...
...
```

- In the `proto-json-base64` output mode, a completed operation is reflected by the value `PROGRESS_DONE` of the `progress` attribute:

```
{"id":"ydb://...", ...,"progress":"PROGRESS_DONE",... }
```

Completing an import operation

After completing the import, use the `operation forget` command to remove the import from the list of operations:

```
ydb -p quickstart operation forget "ydb://import/8?id=281474976788395&kind=fs"
```

List of import operations

To get a list of import operations, use the `operation list import/nfs` command:

```
ydb -p quickstart operation list import/nfs
```

The output format `operation list` is also set by the `--format` option.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Importing to the database root

Importing the contents of the `/mnt/nfs/backups/export1` directory on the file system into the database root:

```
ydb -p quickstart import nfs \
  --fs-path /mnt/nfs/backups/export1
```

Importing multiple directories

Importing objects from the `dir1` and `dir2` directories of the export located in `/mnt/nfs/backups/export1` on the file system into the database directories with the same names:

```
ydb -p quickstart import nfs \
  --fs-path /mnt/nfs/backups/export1 \
  --include dir1 --include dir2
```

Importing an encrypted export

Importing a single table that was exported along the `dir/my_table` path into the `dir1/dir/my_table` path from an encrypted export located in `/mnt/nfs/backups/export1` on the file system, using a secret key from the `~/my_secret_key` file.

```
ydb -p quickstart import nfs \
  --fs-path /mnt/nfs/backups/export1 --destination-path dir1 \
  --include dir/my_table \
  --encryption-key-file ~/my_secret_key
```

Getting operation IDs

To get a list of import operation IDs in a format convenient for processing in bash scripts, you can use the jq utility:

```
ydb -p quickstart operation list import/nfs --format proto-json-base64 | jq -r ".operations[].id"
```

You will get output where each new line contains an operation ID, for example:

```
ydb://import/8?id=281474976789577&kind=fs
ydb://import/8?id=281474976789526&kind=fs
ydb://import/8?id=281474976788779&kind=fs
```

Using these IDs, you can, for example, run a loop to terminate all current operations:

```
ydb -p quickstart operation list import/nfs --format proto-json-base64 | jq -r ".operations[].id" | while rea
d line; do ydb -p quickstart operation forget $line;done
```

Importing data from a file to an existing table

With the `import file` command, you can import data from CSV or TSV files to an existing table.

Data from an imported file is read in batches whose size is set in the `--batch-bytes` option. An independent query is used to write each batch to the database. The queries are executed asynchronously. When the number of executed queries reaches `--max-in-flight`, reading from the file pauses. You can import data from multiple files using a single command. In this case, data from the files will be read asynchronously.

The command implements the `BulkUpsert` method, which ensures high efficiency of multi-row bulk upserts with no atomicity guarantees. The upsert process is split into multiple independent parallel transactions, each covering a single partition. When completed successfully, it guarantees that all data is upserted.

If the table already includes data, it's replaced by imported data on primary key match.

The imported file must be in the UTF-8 encoding. Line feeds aren't supported in the data field.

Note

If the table doesn't exist yet, you can use the ydb tools infer csv command to generate the `CREATE TABLE` statement based on an existing CSV file.

You can also try to import into a non-existent table. In that case, the command will suggest running ydb tools infer csv with the correct options.

General format of the command:

```
ydb [connection options] import file csv|json|parquet|tsv [options] <input files...>
```

where [connection options] are database connection options

`<input files>`: Paths to local file system files you want to import.

Subcommand options

Required options

- `-p, --path STRING`: A path to the table in the database.

Additional options

- `--timeout VAL`: Time within which the operation should be completed on the server. Default: `300s`.
- `--skip-rows NUM`: A number of rows from the beginning of the file that will be skipped at import. The default value is `0`.
- `--header`: Use this option if the first row (excluding the rows skipped by `--skip-rows`) includes names of data columns to be mapped to table columns. If the header row is missing, the data is mapped according to the order in the table schema.
- `--delimiter STRING`: The data column delimiter character. You can't use the tabulation character as a delimiter in this option. For tab-delimited import, use the `import file tsv` subcommand. Default value: `,`.
- `--null-value STRING`: The value to be imported as `NULL`. Default value: `""`.
- `--batch-bytes VAL`: Split the imported file into batches of specified sizes. If a row fails to fit into a batch completely, it's discarded and added to the next batch. Whatever the batch size is, the batch must include at least one row. Default value: `1 MiB`.
- `--max-in-flight VAL`: The number of data batches imported in parallel. You can increase this option value to import large files faster. The default value is `100`.
- `--threads VAL`: Maximum number of threads used to import data. Default value: Number of logical processors.
- `--columns`: List of data columns in the file delimited by a `comma` (for `csv` format) or by a tab character (for `tsv` format). If you use the `--header` option, the column names in it will be replaced by column names from the list. If the number of columns in the list mismatches the number of data columns, you will get an error.
- `--newline-delimited`: This flag guarantees that there will be no line breaks in records. If this flag is set, and the data is loaded from a file, then different upload streams will process different parts of the source file. This way you can distribute the workload across all partitions, ensuring the maximum performance when uploading sorted datasets to partitioned tables.

Examples

Note

The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Before performing the examples, create a table named `series`.

Import file

The file includes data without any additional information. The `,` character is used as a delimiter.

```
1,IT Crowd,The IT Crowd is a British sitcom.,13182
2,Silicon Valley,Silicon Valley is an American comedy television series.,16166
```

Note

The `release_date` column in the `series` table has the Date type, so the release date in the imported file has a numeric format. To import values in the timestamp format, use string-type table columns for them. Alternatively, you can import them to a temporary table and convert them to a relevant type.

To import such a file, use the command:

```
ydb import file csv -p series series.csv
```

The following data will be imported:

```
┌──────────────┬───────────┬──────────────────────────────────────────────────────────┬──────────────┐
| release_date | series_id | series_info                                              | title        |
├──────────────┼───────────┼──────────────────────────────────────────────────────────┼──────────────┤
| "2006-02-03" | 1         | "The IT Crowd is a British sitcom."                      | "IT Crowd"   |
├──────────────┼───────────┼──────────────────────────────────────────────────────────┼──────────────┤
| "2014-04-06" | 2         | "Silicon Valley is an American comedy television series." | "Silicon Valley" |
└──────────────┴───────────┴──────────────────────────────────────────────────────────┴──────────────┘
```

Importing multiple files

The following files include CSV data without additional information:

- series1.csv:

  ```
  1,IT Crowd,The IT Crowd is a British sitcom.,131822
  ```

- series2.csv:

  ```
  2,Silicon Valley,Silicon Valley is an American comedy television series., 16166
  ```

To import such files, run the command:

```
ydb import file csv -p series series1.csv series2.csv
```

Import file with the `|` delimiter

The file includes data without any additional information. The `|` character is used as a delimiter.

```
1|IT Crowd|The IT Crowd is a British sitcom.|13182
2|Silicon Valley|Silicon Valley is an American comedy television series.|16166
```

To import such a file, use `|` in the `--delimiter` option:

```
ydb import file csv -p series --delimiter "|" series.csv
```

Skip rows and read column headers

The file includes additional information in the first and second row, as well as column headers in the third row. The order of data in the file rows mismatches the order of columns in the table:

```
#The file contains data about the series.
#
series_id,title,release_date,series_info
1,IT Crowd,13182,The IT Crowd is a British sitcom.
2,Silicon Valley,16166,Silicon Valley is an American comedy television series.
```

To skip comments in the first and second rows, use `--skip-rows 2`. To process the third row as headers and map the file data to table columns, use the `--header` option:

```
ydb import file csv -p series --skip-rows 2 --header series.csv
```

Replace values to `NULL`

The file includes the `\N` sequence often used for `NULL`, as well as empty strings.

```
1,IT Crowd,The IT Crowd is a British sitcom.,13182
2,Silicon Valley,"",\N
3,Lost,,\N
```

Use `--null-value "\N"` so that `\N` is interpreted as `NULL`:

```
ydb import file csv -p series --null-value "\N" series.csv
```

The following data will be imported:

```
┌──────────────┬───────────┬──────────────────────────────┬──────────────┐
| release_date | series_id | series_info                  | title        |
├──────────────┼───────────┼──────────────────────────────┼──────────────┤
| "2006-02-03" | 1         | "The IT Crowd is a British sitcom." | "IT Crowd"   |
├──────────────┼───────────┼──────────────────────────────┼──────────────┤
| null         | 2         | ""                           | "Silicon Valley" |
├──────────────┼───────────┼──────────────────────────────┼──────────────┤
```

```
| null         | 3         | ""                                   | "Lost"           |
└──────────────┴───────────┴──────────────────────────────────────┴──────────────────┘
```

Backup and recovery

YDB is initially designed for data safety in case of hardware failures: redundancy options are available for different numbers of availability zones, racks, hosts, disks, and other components (see cluster operating modes). This reduces the risk of data loss due to hardware failures.

However, data can be lost or corrupted **logically**: errors or malicious actions can lead to mass deletion/corruption of data by operations legitimate for the DBMS. In such cases, cluster fault tolerance does not replace a **separate copy** of data outside the cluster.

Backup is used to protect against such scenarios and allows restoring data from a backup. It is recommended to store copies on a separate medium or in cloud object storage (for example, via export to files or export to S3).

YDB provides several solutions for performing backup and recovery. For conceptual information and comparison of backup methods, see backup concepts.

- Backup to files and recovery using the YDB CLI.
- Backup to S3-compatible storage and recovery using the YDB CLI.
- Backup of system tablets to the local file system of cluster hosts.

YDB CLI

Files

The following commands are used for backup to files:

- `ydb admin cluster dump` — for backing up cluster metadata
- `ydb admin database dump` — for backing up the database
- `ydb tools dump` — for backing up individual schema objects or directories

Learn more about these commands in Exporting data to the file system.

The following commands are used for restoration from a file backup:

- `ydb admin cluster restore` — for restoring cluster metadata from a backup
- `ydb admin database restore` — for restoring the database from a backup
- `ydb tools restore` — for restoring individual schema objects or directories from a backup

Learn more about these commands in Importing data from the file system.

S3-compatible storage

To perform backup to an S3-compatible storage (for example, AWS S3), the `ydb export s3` command is used. Follow the link to the YDB CLI reference for information about this command.

To perform restoration from a backup created in an S3-compatible storage, the `ydb import s3` command is used. Follow the link to the YDB CLI reference for information about this command.

NFS

To perform backup to NFS on hosts where YDB is running, the `ydb export nfs` command is used. Follow the link to the YDB CLI reference for information about this command.

To perform restoration from a backup created in NFS on hosts where YDB is running, the `ydb import nfs` command is used. Follow the link to the YDB CLI reference for information about this command.

For more details on configuring NFS for backup and recovery, see the recipe Backup and recovery via NFS.

> **Note**
>
> The speed of backup and restore operations to/from S3-compatible storage or NFS is tuned to minimize the impact on user load. To control the operation speed, configure limits for the corresponding queue of the resource broker.

> **Note**
>
> When performing an export, a directory named `export_*` is created in the root directory of the database, where `*` is the numeric part of the export identifier. This directory contains tables that hold a consistent snapshot of the exported data at the time the export started. After the backup operation completes successfully, the `export_*` directory along with its contents is deleted.

Backup collections

Backup collections provide incremental backup and point-in-time recovery for production workloads. For conceptual information and architecture details, see Backup collections.

Backup collections are recommended for production environments with a regular backup schedule and large datasets where incremental changes are significantly smaller than the total volume. For simpler scenarios (one-time migrations, development environments, small databases), consider using export/import or dump/restore.

For step-by-step instructions on configuring and using backup collections, see:

- Backup collections — architecture, concepts, and limitations
- Recipes for backup collections — typical scenarios and examples

Backup of system tablets

The system tablet backup mechanism provides incremental copying of cluster metadata to the local file system of the cluster hosts.

For conceptual information and how it works, see Backup concepts.

For step-by-step instructions on enabling and restoring, see Recipes for system tablet backup.

Commands for topics

Using YDB CLI commands, you can perform the following operations:

- Creating a topic.
- Updating a topic.
- Deleting a topic.
- Adding a topic consumer.
- Deleting a topic consumer.
- Saving a consumer offset.
- Reading messages from a topic.
- Writing messages to a topic.

Creating a topic

You can use the `topic create` subcommand to create a new topic.

General format of the command:

```
ydb [global options...] topic create [options...] <topic-path>
```

- `global options` : Global parameters.
- `options` : Parameters of the subcommand.
- `topic-path` : Topic path.

View the description of the create topic command:

```
ydb topic create --help
```

Parameters of the subcommand

Name	Description
`--partitions-count`	The number of topic partitions. The default value is `1`.
`--retention-period`	Data retention time in a topic. A positive number followed by a unit of time. The following units are supported: • `s` – seconds; • `m` – minutes; • `h` – hours; • `d` – days. The default value is `24h`.
`--partition-write-speed-kbps`	The maximum write speed to a partition, specified in KB/s. The default value is `1024`.
`--retention-storage-mb`	The maximum storage size, specified in MB. When the limit is reached, the oldest data will be deleted. The consumed space may exceed the set value when autopartitioning is enabled. The default value is `0` (no limit).
`--auto-partitioning-strategy`	Auto-partitioning modes. Possible values: • `up` — the number of partitions can increase under high load, but cannot decrease; • `paused` — automatic scaling of the number of partitions is paused; • `disabled` — automatic scaling of the number of partitions is disabled. The default value is `disabled`.
`--supported-codecs`	Supported data compression methods. Set with a comma. The default value is `raw`. Possible values: • `RAW` : No compression. • `ZSTD` : zstd compression. • `GZIP` : gzip compression. • `LZOP` : lzop compression.
`--metering-mode`	The topic pricing method for a serverless database. Possible values: • `request-units` : Based on actual usage. • `reserved-capacity` : Based on dedicated resources.

Examples

Note

The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Create a topic with 2 partitions, `RAW` and `GZIP` compression methods, message retention time of 2 hours, and the `my-topic` path:

```
ydb -p quickstart topic create \
  --partitions-count 2 \
  --supported-codecs raw,gzip \
  --retention-period 2h \
  my-topic
```

View parameters of the created topic:

```
ydb -p quickstart scheme describe my-topic
```

Result:

```
RetentionPeriod: 2h
PartitionsCount: 2
```

```
SupportedCodecs: RAW, GZIP
```

Updating a topic

You can use the `topic alter` subcommand to update a previously created topic.

General format of the command:

```
ydb [global options...] topic alter [options...] <topic-path>
```

- `global options` : Global parameters.
- `options` : Parameters of the subcommand.
- `topic-path` : Topic path.

View the description of the update topic command:

```
ydb topic alter --help
```

Parameters of the subcommand

The command changes the values of parameters specified in the command line. The other parameter values remain unchanged.

Name	Description
`--partitions-count`	The number of topic partitions. You can only increase the number of partitions.
`--retention-period`	Data retention time in a topic. A positive number followed by a unit of time. The following units are supported: • `s` – seconds; • `m` – minutes; • `h` – hours; • `d` – days.
`--partition-write-speed-kbps`	The maximum write speed to a partition, specified in KB/s. The default value is `1024`.
`--retention-storage-mb`	The maximum storage size, specified in MB. When the limit is reached, the oldest data will be deleted. The consumed space may exceed the set value when autopartitioning is enabled. The default value is `0` (no limit).
`--auto-partitioning-strategy`	Auto-partitioning modes. Possible values: • `up` — the number of partitions can increase under high load, but cannot decrease; • `paused` — automatic scaling of the number of partitions is paused.
`--supported-codecs`	Supported data compression methods. Possible values: • `RAW` : No compression. • `ZSTD` : zstd compression. • `GZIP` : gzip compression. • `LZOP` : lzop compression.
`--metering-mode`	The topic pricing method for a serverless database. Possible values: • `request-units` : Based on actual usage. • `reserved-capacity` : Based on dedicated resources.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Add a partition and the `lzop` compression method to the previously created topic:

```
ydb -p quickstart topic alter \
  --partitions-count 3 \
  --supported-codecs raw,gzip,lzop \
  my-topic
```

Make sure that the topic parameters have been updated:

```
ydb -p quickstart scheme describe my-topic
```

Result:

```
RetentionPeriod: 2h
PartitionsCount: 3
SupportedCodecs: RAW, GZIP, LZOP
```

Deleting a topic

You can use the `topic drop` subcommand to delete a previously created topic.

> **Note**
>
> Deleting a topic also deletes all the consumers added for it.

General format of the command:

```
ydb [global options...] topic drop <topic-path>
```

- `global options`: Global parameters.
- `topic-path`: Topic path.

View the description of the delete topic command:

```
ydb topic drop --help
```

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Delete the previously created topic:

```
ydb -p quickstart topic drop my-topic
```

Adding a topic consumer

You can use the `topic consumer add` command to add a consumer for a previously created topic.

General format of the command:

```
ydb [global options...] topic consumer add [options...] <topic-path>
```

- `global options`: Global parameters.
- `options`: Parameters of the subcommand.
- `topic-path`: Topic path.

View the description of the add consumer command:

```
ydb topic consumer add --help
```

Parameters of the subcommand

Name	Description
`--consumer VAL`	Name of the consumer to be added.
`--starting-message-timestamp VAL`	Time in UNIX timestamp format. Consumption starts as soon as the first message is received after the specified time. If the time is not specified, consumption will start from the oldest message in the topic.
`--supported-codecs`	Supported data compression methods. The default value is `raw`. Possible values: • `RAW`: No compression. • `ZSTD`: zstd compression. • `GZIP`: gzip compression. • `LZOP`: lzop compression.

Examples

Note

The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Create a consumer with the `my-consumer` name for the previously created `my-topic` topic. Consumption will start as soon as the first message is received after August 15, 2022 13:00:00 GMT:

```
ydb -p quickstart topic consumer add \
  --consumer my-consumer \
  --starting-message-timestamp 1660568400 \
  my-topic
```

Make sure the consumer was created:

```
ydb -p quickstart scheme describe my-topic
```

Result:

```
RetentionPeriod: 2h
PartitionsCount: 2
SupportedCodecs: RAW, GZIP

Consumers:
┌──────────────┬─────────────────┬──────────────────────────────┬─────────────┐
| ConsumerName | SupportedCodecs | ReadFrom                     | Important |
├──────────────┼─────────────────┼──────────────────────────────┼─────────────┤
| my-consumer  | RAW, GZIP       | Mon, 15 Aug 2022 16:00:00 MSK | 0         |
└──────────────┴─────────────────┴──────────────────────────────┴─────────────┘
```

Deleting a topic consumer

You can use the `topic consumer drop` command to delete a previously added consumer.

General format of the command:

```
ydb [global options...] topic consumer drop <topic-path>
```

- `global options`: Global parameters.
- `topic-path`: Topic path.

View the description of the delete consumer command:

```
ydb topic consumer drop --help
```

Parameters of the subcommand

Name	Description
`--consumer VAL`	Name of the consumer to be deleted.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Delete the previously created consumer with the `my-consumer` name for the `my-topic` topic:

```
ydb -p quickstart topic consumer drop \
  --consumer my-consumer \
  my-topic
```

Saving a consumer offset

Each topic consumer has a consumer offset.

You can use the `topic consumer offset commit` command to save the consumer offset for the consumer that you added.

General format of the command:

```
ydb [global options...] topic consumer offset commit [options...] <topic-path>
```

- `global options` : Global parameters.
- `options` : Parameters of the subcommand.
- `topic-path` : Topic path.

Viewing the command description:

```
ydb topic consumer offset commit --help
```

Parameters of the subcommand

Name	Description
`--consumer <value>`	Consumer name.
`--partition <value>`	Partition number.
`--offset <value>`	Offset value that you want to set.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

For `my-consumer`, set the offset of 123456789 in `my-topic` and partition `1`:

```
ydb -p db1 topic consumer offset commit \
  --consumer my-consumer \
  --partition 1 \
  --offset 123456789 \
  my-topic
```

Reading messages from a topic

The `topic read` command reads messages from a topic and outputs them to a file or the command-line terminal:

```
ydb [connection options] topic read <topic-path> --consumer STR \
  [--format STR] [--wait] [--limit INT] \
  [--transform STR] [--file STR] [--commit BOOL] \
  [additional parameters...]
```

where [connection options] are database connection options

Three command modes are supported:

1. **Single message**. No more than one message is read from a topic.
2. **Batch mode**. Messages are read from a topic until it runs out of messages for processing or their number exceeds the limit that must be set.
3. **Streaming mode**. Messages are read from a topic as they appear while waiting for new messages to arrive until you terminate the command with `Ctrl+C` or the number of messages exceeds the limit that is set optionally.

Parameters

Required parameters

`<topic-path>`: Topic path

Basic optional parameters

`-c VAL`, `--consumer VAL`: Topic consumer name.

- If not set, then you need to specify partitions through --partition-ids to read without consumer
- Message consumption starts from the current offset for this consumer (if the `--timestamp` parameter is not specified).
  If consumer name is not specified, message consumption will start from the first message in partition.

`--format STR`: Output format.

- Specifies how to format messages at the output. Some formats don't support streaming mode.
- List of supported formats:

Name	Description	Is streaming mode supported?
`single-message` (default)	The contents of no more than one message are output without formatting.	-
`pretty`	Output to a pseudo-graphic table with columns containing message metadata. The message itself is output to the `body` column.	No
`newline-delimited`	Messages are output with a delimiter (`0x0A` newline character) added after each message.	Yes
`concatenated`	Messages are output one after another with no delimiter added.	Yes

`--wait` (`-w`): Waiting for new messages to arrive.

- Enables waiting for the first message to appear in a topic. If not set and the topic has no messages to handle, the command is terminated once started. If the flag is set, the started read message command waits for the first message to arrive to be processed.
- Enables streaming selection mode for the formats that support it, or else batch mode is used.

`--limit INT`: The maximum number of messages that can be consumed from a topic.

- The default and acceptable values depend on the selected output format:

Does the format support streaming selection mode?	Default limit value	Acceptable values
No	10	1-500
Yes	0 (no limit)	0-500

`--transform VAL`: Method for transforming messages.

- Defaults to `none`.
- Possible values:
  `base64`: A message is transformed into Base64
  `none`: The contents of a message are output byte by byte without transforming them.

`--file VAL` (`-f VAL`): Write the messages read to the specified file. If not set, messages are output to `stdout`.

`--commit BOOL`: Commit message reads. Default value - `false`.

- Possible values: `true` or `false`.
- If `true`, a consumer's current offset is shifted as topic messages are consumed.
- If the value is set to `false`, messages will be read, but the reading progress won't be saved, and upon restart, the messages will be read again. This functionality is useful for debugging: allowing messages to be read without affecting the production system (without offset commit).

Other optional parameters

Name	Description
`--idle-timeout VAL`	Timeout for deciding if a topic is empty, meaning that it contains no messages for processing. The time is counted from the point when a connection is established once the command is run or when the last message is received. If no new messages arrive from the server during the specified timeout, the topic is considered to be empty. Defaults to `1s` (1 second).
`--timestamp VAL`	Message consumption starts from the point in time specified in UNIX timestamp format. If not set, messages are consumed starting from the consumer's current offset in the topic. If set, consumption starts from the first message received after the specified time.
`--metadata-fields VAL`	List of message attributes whose values should be output in columns with metadata in `pretty` format. If not set, columns with all attributes are output. Possible values: • `write_time`: The time a message is written to the server in UNIX timestamp format. • `meta`: Message metadata. • `create_time`: The time a message is created by the source in UNIX timestamp format. • `seq_no`: Message sequence number. • `offset`: Message sequence number within a partition. • `message_group_id`: Message group ID. • `body`: Message body.
`--partition-ids VAL`	Comma-separated list of partition identifiers to read from. If not specified, messages are read from all partitions.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

In all the examples below, a topic named `topic1` and a consumer named `c1` are used.

- Reading a single message with output to the terminal: If the topic doesn't contain new messages for this consumer, the command terminates with no data output:

```
ydb -p quickstart topic read topic1 -c c1
```

- Waiting for and reading a single message written to a file named `message.bin`. The command keeps running until new messages appear in the topic for this consumer. However, you can terminate it with `Ctrl+C`:

```
ydb -p quickstart topic read topic1 -c c1 -w -f message.bin
```

- Viewing information about messages waiting to be handled by the consumer without committing them. Up to 10 first messages are output:

```
ydb -p quickstart topic read topic1 -c c1 --format pretty --commit false
```

- Output messages to the terminal as they appear, using newline delimiter characters and transforming messages into Base64. The command will be running until you terminate it with `Ctrl+C`:

```
ydb -p quickstart topic read topic1 -c c1 -w --format newline-delimited --transform base64
```

- Track when new messages with the `ERROR` text appear in the topic and output them to the terminal once they arrive:

```
ydb -p quickstart topic read topic1 -c c1 --format newline-delimited -w | grep ERROR
```

- Receive another non-empty batch of no more than 150 messages transformed into base64, delimited with newline characters, and written to the `batch.txt` file:

```
ydb -p quickstart topic read topic1 -c c1 \
  --format newline-delimited -w --limit 150 \
  --transform base64 -f batch.txt
```

- Examples of YDB CLI command integration

Writing messages to a topic

The `topic write` command writes messages to a topic from a file or `stdin`:

```
ydb [connection options] topic write <topic-path> \
  [--file STR] [--format STR] [--transform STR] \
  [additional parameters...]
```

where [connection options] are database connection options

Parameters

Basic parameters

`<topic-path>`: Topic path, the only required parameter.

`--file VAL` (`-f VAL`): Read a stream of incoming messages and write them to a topic from the specified file. If not set, messages are read from `stdin`.

`--format STR`: Format of the incoming message stream. Supported formats:

Name	Description
`single-message` (default)	The entire input stream is treated as a single message to be written to the topic.
`newline-delimited`	A stream at the input contains multiple messages delimited with the `0x0A` newline character.

`--transform VAL`: Method for transforming messages.

- Defaults to `none`.
- Possible values:
  - `base64`: Decode each message in the input stream from Base64 and write the output to the topic. If decoding fails, the command is aborted with an error.
  - `none`: Write the contents of a message from the input stream to the topic byte by byte without transforming them.

Additional parameters

Name	Description
`--delimiter STR`	Delimiter byte. The input stream is delimited into messages with the specified byte. Specified only if no `--format` is set. Specified as an escaped string.
`--message-group-id STR`	Message group string ID. If not set, all messages generated from the input stream are assigned the same ID value as a hexadecimal string representation of a random three-byte integer.
`--codec STR`	Codec used for message compression on the client before sending them to the server. Possible values: `RAW` (no compression, default), `GZIP`, and `ZSTD`. Compression causes higher CPU utilization on the client when reading and writing messages, but usually lets you reduce the volume of data transferred over the network and stored. When consumers read messages, they're automatically decompressed with the codec used when writing them, without specifying any special options. Make sure the specified codec is listed in the topic parameters as supported.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

All the examples given below use a topic named `topic1`.

- Writing a terminal input to a single message Once the command is run, you can type any multi-line text and press `Ctrl+D` to input it.

  ```
  ydb -p quickstart topic write topic1
  ```

- Writing the contents of the `message.bin` file to a single message compressed with the GZIP codec

  ```
  ydb -p quickstart topic write topic1 -f message.bin --codec GZIP
  ```

- Writing the contents of the `example.txt` file delimited into messages line by line

  ```
  ydb -p quickstart topic write topic1 -f example.txt --format newline-delimited
  ```

- Writing a resource downloaded via HTTP and delimited into messages with tab characters

  ```
  curl http://example.com/resource | ydb -p quickstart topic write topic1 --delimiter "\t"
  ```

- Examples of YDB CLI command integration

Message pipeline processing

The use of the `topic read` and `topic write` commands with standard I/O devices and support for reading messages in streaming mode lets you build full-featured integration scenarios with message transfer across topics and their conversion. This section describes a number of these scenarios.

- Transferring a single message from `topic1` in the `quickstart` database to `topic2` in `db2`, waiting for it to appear in the source topic

```
ydb -p quickstart topic read topic1 -c c1 -w | ydb -p db2 topic write topic2
```

- Transferring all one-line messages that appear in `topic1` in the `quickstart` database to `topic2` in `db2` in background mode. You can use this scenario if it's guaranteed that there are no `0x0A` bytes (newline) in source messages.

```
ydb -p quickstart topic read topic1 -c c1 --format newline-delimited -w | \
ydb -p db2 topic write topic2 --format newline-delimited
```

- Transferring an exact binary copy of all messages that appear in `topic1` in the `quickstart` database to `topic2` in `db2` in background mode with base64-encoding of messages in the transfer stream.

```
ydb -p quickstart topic read topic1 -c c1 --format newline-delimited -w --transform base64 | \
ydb -p quickstart topic write topic2 --format newline-delimited --transform base64
```

- Transferring a limited batch of one-line messages filtered by the `ERROR` substring

```
ydb -p quickstart topic read topic1 -c c1 --format newline-delimited | \
grep ERROR | \
ydb -p db2 topic write topic2 --format newline-delimited
```

- Writing YQL query results as messages to `topic1`

```
ydb -p quickstart yql -s "select * from series" --format json-unicode | \
ydb -p quickstart topic write topic1 --format newline-delimited
```

Running an SQL query with the transmission of messages from the topic as parameters

- Running a YQL, passing each message read from `topic1` as a parameter

```
ydb -p quickstart topic read topic1 -c c1 --format newline-delimited -w | \
ydb -p quickstart sql -s 'declare $s as String;select Len($s) as Bytes' \
--input-framing newline-delimited --input-param-name s --input-format raw
```

- Running a YQL query involving adaptive batching of parameters from messages read from `topic1`

```
ydb -p quickstart topic read topic1 -c c1 --format newline-delimited -w | \
ydb -p quickstart sql \
-s 'declare $s as List<String>;select ListLength($s) as Count, $s as Items' \
--input-framing newline-delimited --input-param-name s --input-format raw \
--input-batch adaptive
```

Overview

You can use the following YDB CLI commands to run queries:

- ydb sql: A single command to execute any SQL query supported by YDB.
- ydb: Switches the console to interactive mode to execute queries.

Query parameterization is explained separately.

Query execution

You can use the `ydb sql` subcommand to execute an SQL query. The query can be of any type (DDL, DML, etc.) and can consist of several subqueries. The `ydb sql` subcommand establishes a streaming connection and retrieves data through it. With in-stream query execution, no limit is imposed on the amount of data read. Data can also be written using this command, which is more efficient when executing repeated queries with data passed through parameters.

General format of the command:

```
ydb [global options...] sql [options...]
```

- `global options` : Global parameters.
- `options` : Subcommand parameters.

View the description of the query execution command:

```
ydb sql --help
```

Parameters of the subcommand

Name	Description
`-h`, `--help`	Print general usage help.
`-hh`	Print complete usage help, including specific commands not shown with `--help`.
`-s`, `--script`	Script (query) text to execute.
`-f`, `--file`	Path to a file with query text to execute. Path `-` means reading query text from `stdin` which disables passing parameters via `stdin`.
`--stats`	Statistics mode. Available options: • `none` (default): Do not collect statistics. • `basic` : Collect aggregated statistics for updates and deletes per table. • `full` : Include execution statistics and plan in addition to `basic`. • `profile` : Collect detailed execution statistics, including statistics for individual tasks and channels.
`--explain`	Execute an explain request for the query. Displays the query's logical plan. The query is not actually executed and does not affect database data.
`--explain-ast`	Same as `--explain`, but in addition to the query's logical plan, an abstract syntax tree (AST) is printed. The AST section contains a representation in the internal miniKQL language.
`--explain-analyze`	Execute the query in `EXPLAIN ANALYZE` mode. Displays the query execution plan. Query results are ignored. **Important note: The query is actually executed, so any changes will be applied to the database**.
`--format`	Output format. Available options: • `pretty` (default): Human-readable format. • `json-unicode` : JSON output with binary strings Unicode-encoded and each JSON string in a separate line. • `json-unicode-array` : JSON output with binary strings Unicode-encoded and the result output as an array of JSON strings with each JSON string in a separate line. • `json-base64` : JSON output with binary strings Base64-encoded and each JSON string in a separate line. • `json-base64-array` : JSON output with binary strings Base64-encoded and the result output as an array of JSON strings with each JSON string in a separate line; • `parquet` : Output in Apache Parquet format. • `csv` : Output in CSV format. • `tsv` : Output in TSV format.

Working with parameterized queries

For a detailed description with examples on how to use parameterized queries, see Executing parameterized queries.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Running DDL and DML together in a single query is not supported.

```
# Creating a table
ydb -p quickstart sql -s '
  CREATE TABLE series (
    series_id Uint64,
    title Utf8,
    series_info Utf8,
    release_date Date,
    PRIMARY KEY (series_id)
  );
```

```
'

# Populating with data and retrieving a sample
ydb -p quickstart sql -s '
    UPSERT INTO series (series_id, title, series_info, release_date)
    VALUES (1, "Title1", "Info1", Cast("2023-04-20" as Date));
    SELECT * FROM series;
'

# Adding an index
ydb -p quickstart sql -s '
    ALTER TABLE series
    ADD INDEX title_idx GLOBAL ON (title);
'
```

Command output:

```
┌──────────────┬────────────┬──────────────┬────────────┐
| release_date | series_id | series_info | title    |
├──────────────┼────────────┼──────────────┼────────────┤
| "2023-04-20" | 1         | "Info1"     | "Title1" |
└──────────────┴────────────┴──────────────┴────────────┘
```

To run a query from a file (for example, script1.yql) with JSON output

```
ydb -p quickstart sql -f script1.yql --format json-unicode
```

Command output:

```
{"release_date":"2023-04-20","series_id":1,"series_info":"Info1","title":"Title1"}
```

Examples of passing parameters to scripts are given in the article on passing parameters to query execution commands.

Executing parameterized queries

Overview

YDB CLI supports executing parameterized queries. To work with parameters, the query text must contain their definitions using the YQL `DECLARE` command.

The main tool for executing parameterized queries in YDB CLI is the ydb sql command.

Why use parameterized queries?

Using parameterized queries provides several important benefits:

- **Improved performance:** Parameterized queries significantly improve performance when executing many similar operations that differ only in input values. This is due to query plan caching. After the first compilation of a query, its plan is stored in the LRU cache of the YDB node. All subsequent calls of the same query with new parameters skip the compilation stage and start executing immediately, saving time and resources.
- **Protection against SQL injection:** Another important advantage of using parameterized queries is protection against SQL injection. This security feature ensures proper handling of input data, reducing the risk of malicious code execution.

Single query execution

This command supports passing parameters via command-line options, a file, and also via `stdin`. When passing parameters via `stdin` or a file, repeated streaming execution of the query with different parameter values and batching capability is supported. For this purpose, the ydb sql command provides the following parameters:

Name	Description
`-p, --param`	Value of a single query parameter in `$name=value` or `name=value` format, where `name` is the parameter name and `value` is its value (a valid JSON value).
`--input-file`	Name of a file in JSON format encoded in UTF-8, which contains parameter values mapped to query parameters by key names. At most one parameter file can be used.
`--input-format`	Format for representing parameter values. Applies to all methods of passing them (via a command parameter, file, or `stdin`). Possible values: • `json` (default): JSON format. • `csv` — CSV format. By default, parameter names must be in the CSV file header. For a single query execution, only one row is allowed in the file, not counting the header. • `tsv` — TSV format. • `raw`: The input stream from `stdin` or `--input-file` contains only the parameter value as binary data. The parameter name must be specified with the `--input-param-name` option.
`--input-binary-strings`	Encoding format for parameter values of the 'binary string' type (`DECLARE $par AS String`). Specifies how binary strings from the input stream should be interpreted. Possible values: • `unicode`: Each byte in a binary string that is not a printable ASCII character (codes 32-126) must be encoded in UTF-8. • `base64`: Binary strings are represented in Base64 encoding. This allows passing binary data, which YDB CLI will decode from Base64.

If values are specified for all parameters that do not allow `NULL` (i.e., with the NOT NULL type), as part of the `DECLARE` statement, the query will be executed on the server. If a value is missing for at least one such parameter, the command will fail with an error and the message "Parameter value not set".

More specific options for using input parameters

The following options are not shown in the `--help` output. Their description can only be seen in the `-hh` output.

Name	Description
`--input-framing`	Sets the framing for the input stream (`stdin` or `--input-file`). Determines how the input stream is split into separate parameter sets. Possible values: • `no-framing` (default): One parameter set is expected from the input stream, the query is executed once. • `newline-delimited`: A newline character marks the end of one parameter set in the input stream, separating it from the next. A parameter set is considered complete each time a newline character is received in the input stream.
`--input-param-name`	The name of the parameter whose value is passed to the input stream. Specified without the `$` character. Required when using the `raw` format in `--input-format`. When used with the JSON format, the input stream is interpreted not as a JSON document, but as a JSON value, with the value passed to the parameter with the specified name.
`--input-columns`	A string with column names replacing the header of the CSV/TSV document read from the input stream. When this option is specified, the header is considered absent. The option is only valid with CSV and TSV formats.
`--input-skip-rows`	The number of lines from the beginning of the data read from stdin to skip, not including the header line if present. The option is only valid with stdin formats CSV and TSV.

`--input-batch`	Batching mode for parameter set values received from the input stream (`stdin` or `--input-file`). Possible values: • `iterative` (default): Batching is disabled. The query is executed for each parameter set (exactly once if option `--input-framing` uses `no-framing`) • `full` : Full batch. The query is executed once after reading the input stream is complete; all received parameter sets are wrapped into `List<...>` , the parameter name is set by option `--input-param-name` • `adaptive` : Adaptive batching. The query is executed each time a limit on the number of parameter sets in a single query (`--input-batch-max-rows`) or a processing delay limit (`--input-batch-max-delay`) is triggered. All parameter sets received up to that point are wrapped into `List<...>` , the parameter name is set by option `--input-param-name` .
`--input-batch-max-rows`	Maximum number of parameter sets in a batch for adaptive batching mode. The next batch will be sent for execution together with the query if the number of data sets in it reaches the specified value. Setting to `0` removes the limit. Default value — `1000` . Parameters are passed to the query without streaming, and the total size of a single gRPC request that includes parameter values has an upper limit of about 5 MB.
`--input-batch-max-delay`	Maximum delay for sending a received parameter set for processing in adaptive batching mode. Specified as a number with a time dimension - `s` (seconds), `ms` (milliseconds), `m` (minutes), etc. Default value — `1s` (1 second). YDB CLI will count time from the moment the first parameter set for a batch is received and will send the accumulated batch for execution as soon as the time exceeds the specified value. This parameter allows efficient batching in case of an unpredictable rate of new parameter sets arriving at `stdin` .

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Passing a single parameter value

On the command line, via the `--param` option:

```
ydb -p quickstart sql -s 'DECLARE $a AS Int64; SELECT $a' --param '$a=10'
```

Via a file in JSON format (which is used by default):

```
echo '{"a":10}' > p1.json
ydb -p quickstart sql -s 'DECLARE $a AS Int64; SELECT $a' --input-file p1.json
```

Via `stdin` , passing a JSON string as a single parameter set:

```
echo '{"a":10}' | ydb -p quickstart sql -s 'DECLARE $a AS Int64; SELECT $a'
```

Via `stdin` , passing only the parameter value and specifying the parameter name via the `--input-param-name` option:

```
echo '10' | ydb -p quickstart sql -s 'DECLARE $a AS Int64; SELECT $a' --input-param-name a
```

Passing parameter values of different types from multiple sources

```
# Create a JSON file with fields 'a', 'b', and 'x', where 'x' will be ignored in the query
echo '{ "a":10, "b":"Some text", "x":"Ignore me" }' > p1.json

# Run the query using ydb-cli, passing in 'a' and 'b' from the input file, and 'c' as a direct parameter
ydb -p quickstart sql \
  -s 'DECLARE $a AS Int64;
      DECLARE $b AS Utf8;
      DECLARE $c AS Int64;

      SELECT $a, $b, $c' \
  --input-file p1.json \
  --param '$c=30'
```

Command output:

```
┌─────────┬─────────────┬─────────┐
| column0 | column1     | column2 |
├─────────┼─────────────┼─────────┤
| 10      | "Some text" | 30      |
└─────────┴─────────────┴─────────┘
```

Passing binary strings in Base64 encoding

```
ydb -p quickstart sql \
  -s 'DECLARE $a AS String;
```

```
    SELECT $a' \
  --input-format json \
  --input-binary-strings base64 \
  --param '$a="SGVsbG8sIHdvcmxkCg=="'
```

Command output:

```
┌──────────────────┐
| column0          |
├──────────────────┤
| "Hello, world\n" |
└──────────────────┘
```

Direct binary content passing

```
curl -Ls http://ydb.tech/docs/en | ydb -p quickstart sql \
  -s 'DECLARE $a AS String;
      SELECT LEN($a)' \
  --input-format raw \
  --input-param-name a
```

Command output (exact number of bytes may differ):

```
┌──────────┐
| column0 |
├──────────┤
| 66426   |
└──────────┘
```

Passing a file in CSV format

```
echo '10,Some text' | ydb -p quickstart sql \
  -s 'DECLARE $a AS Int32;
      DECLARE $b AS String;
      SELECT $a, $b' \
  --input-format csv \
  --input-columns 'a,b'
```

Command output:

```
┌──────────┬──────────────┐
| column0 | column1     |
├──────────┼──────────────┤
| 10      | "Some text" |
└──────────┴──────────────┘
```

Iterative streaming processing

YDB CLI supports the ability to execute a query multiple times with different sets of parameter values, when they are passed via `stdin` **or** an input file (not simultaneously). In this case, the database connection is established once, and the query execution plan is cached, which significantly improves the performance of this approach compared to separate CLI invocations.

To use this feature, you need to pass different sets of values for the same parameters one after another to the command input, telling YDB CLI the rule by which these sets can be separated from each other.

The query is executed as many times as the number of parameter value sets received. Each set received via the input stream (`stdin` or `--input-file`) is merged with the parameter values defined via the `--param` options. Command execution will finish after the input stream ends. Each query is executed in its own transaction.

The rule for separating parameter sets from each other (framing) complements the description of the parameter representation format on the input stream, specified by the `--input--framing` parameter:

Name	Description
`--input-framing`	Specifies framing for the input stream (file or `stdin`). Possible values: • `no-framing` (default) — one parameter set is expected at the input, the query is executed once after reading the stream completes. • `newline-delimited` — a newline character marks the end of one parameter set at the input, separating it from the next. The query is executed each time a newline character is received.

Warning

When using a newline character as a separator for parameter sets, you must ensure it is absent inside the parameter sets. Quoting text does not make a newline inside such text valid. Multiline JSON documents are not allowed.

Example

Streaming processing of multiple parameter sets

JSON

Suppose we need to execute the query three times, with the following sets of parameter values `a` and `b`:

1. `a` = 10, `b` = 20

2. `a` = 15, `b` = 25
3. `a` = 35, `b` = 48

Let's create a file that will contain lines with JSON representation of these sets:

```
echo -e '{"a":10,"b":20}\n{"a":15,"b":25}\n{"a":35,"b":48}' | tee par1.txt
```

Command output:

```
{"a":10,"b":20}
{"a":15,"b":25}
{"a":35,"b":48}
```

Execute the query by passing the contents of this file to `stdin`, with output formatting in JSON:

```
cat par1.txt | \
ydb -p quickstart sql \
  -s 'DECLARE $a AS Int64;
      DECLARE $b AS Int64;
      SELECT $a+$b' \
  --input-framing newline-delimited \
  --format json-unicode
```

Command output:

```
{"column0":30}
{"column0":40}
{"column0":83}
```

Or simply by passing the file name to the `--input-file` option:

```
ydb -p quickstart sql \
  -s 'DECLARE $a AS Int64;
      DECLARE $b AS Int64;
      SELECT $a + $b' \
  --input-file par1.txt \
  --input-framing newline-delimited \
  --format json-unicode
```

Command output:

```
{"column0":30}
{"column0":40}
{"column0":83}
```

The result obtained in this format can be used as input to the command executing the next query.

CSV

Suppose we need to execute the query three times, with the following sets of parameter values `a` and `b`:

1. `a` = 10, `b` = 20
2. `a` = 15, `b` = 25
3. `a` = 35, `b` = 48

Let's create a file with sets of parameter values in CSV format:

```
echo -e 'a,b\n10,20\n15,25\n35,48' | tee par1.txt
```

Command output:

```
a,b
10,20
15,25
35,48
```

Let's execute the query by passing the contents of this file to `stdin`, with output formatting in CSV:

```
cat par1.txt | \
ydb -p quickstart sql \
  -s 'DECLARE $a AS Int64;
      DECLARE $b AS Int64;
      SELECT $a + $b' \
  --input-format csv \
  --input-framing newline-delimited \
  --format csv
```

Command output:

```
30
40
```

```
83
```

Or simply by passing the file name to the `--input-file` option:

```
ydb -p quickstart sql \
  -s 'DECLARE $a AS Int64;
      DECLARE $b AS Int64;
      SELECT $a + $b' \
  --input-file par1.txt \
  --input-format csv \
  --input-framing newline-delimited \
  --format csv
```

Command output:

```
30
40
83
```

The result obtained in this format can be used as input to the command executing the next query, by specifying the header data in CSV format with the `--input-columns` option.

TSV

Suppose we need to execute the query three times, with the following sets of parameter values `a` and `b`:

1. `a` = 10, `b` = row1
2. `a` = 15, `b` = row 2
3. `a` = 35, `b` = "row"\n3

Let's create a file with sets of parameter values in TSV format:

```
echo -e 'a\tb\n10\t20\n15\t25\n35\t48' | tee par1.txt
```

Command output:

```
a  b
10 20
15 25
35 48
```

Let's execute the query by passing the contents of this file to `stdin`, with output formatting in TSV:

```
cat par1.txt | \
ydb -p quickstart sql \
  -s 'DECLARE $a AS Int64;
      DECLARE $b AS Utf8;
      SELECT $a, $b' \
  --input-framing newline-delimited \
  --input-format tsv \
  --format tsv
```

Command output:

```
30
40
83
```

Or simply by passing the file name to the `--input-file` option:

```
ydb -p quickstart sql \
  -s 'DECLARE $a AS Int64;
      DECLARE $b AS Int64;
      SELECT $a + $b' \
  --input-file par1.txt \
  --input-format tsv \
  --input-framing newline-delimited \
  --format tsv
```

Command output:

```
30
40
83
```

The result obtained in this format can be used as input to the command executing the next query, by specifying the header data in TSV format with the `--input-columns` option.

Stream processing with merging parameter values from different sources

Suppose we need to execute the query three times, with the following sets of parameter values `a` and `b`:

1. `a` = 10, `b` = 100
2. `a` = 15, `b` = 100
3. `a` = 35, `b` = 100

```
echo -e '10\n15\n35' | \
ydb -p quickstart sql \
  -s 'DECLARE $a AS Int64;
      DECLARE $b AS Int64;
      SELECT $a + $b AS sum1' \
  --param '$b=100' \
  --input-framing newline-delimited \
  --input-param-name a \
  --format json-unicode
```

Command output:

```
{"sum1":110}
{"sum1":115}
{"sum1":135}
```

Batch stream processing

The YDB CLI supports automatic conversion of parameter sets into `List<...>`, allowing a single server request to process multiple parameter sets in one transaction, which can further significantly improve performance compared to executing queries "one by one".

Two batching modes are supported:

- Full (`full`)
- Adaptive (`adaptive`)

Full batching mode

The full (`full`) mode is a simplified batching option where the query is executed once, wrapping all parameter sets received from the input stream into `List<...>`. If the query size turns out to be too large, an error will be raised.

This batching option is necessary when atomicity must be guaranteed by applying all parameters in a single transaction.

Adaptive batching mode

When operating in adaptive (`adaptive`) mode, the input stream processing is split into multiple transactions, with automatic selection of the batch size for each of them.

This mode allows efficient processing of a wide range of input loads, with unpredictable or infinite amounts of data, as well as unpredictable or rapidly changing input rates. In particular, this profile is typical when feeding the output of another command to `stdin` via the `|` operator.

Adaptive mode solves two main problems of dynamic stream processing:

1. Limiting the maximum batch size.
2. Limiting the maximum data processing delay.

Syntax

To use the batching feature, you need to declare a parameter of type `List<...>` or `List<Struct<...>>` in the DECLARE section of the query, and select the mode with the following parameter:

Name	Description
`--input-batch`	Mode for batching values of parameter sets obtained from the input stream (`stdin` or `--input-file`). Possible values: • `iterative` (default): Batching is disabled. The query is executed for each parameter set (exactly once if the `--input-framing` option uses `no-framing`) • `full`: Full batch. The query will run once after the input stream is read, all received parameter sets are wrapped in `List<...>`, the parameter name is set by the `--input-param-name` option • `adaptive`: Adaptive batching. The query is executed each time the limit on the number of parameter sets in one query (`--input-batch-max-rows`) or the processing delay (`--input-batch-max-delay`) is triggered. All parameter sets received by this point are wrapped in `List<...>`, the parameter name is set by the `--input-param-name` option.

Additional parameters are available in adaptive batching mode:

Name	Description
`--input-batch-max-rows`	Maximum number of parameter sets in a batch for adaptive batching mode. The next batch will be sent for execution together with the query if the number of data sets in it has reached the specified value. Setting to `0` removes the limit. Default value — `1000`. Parameters are passed to the query without streaming, and the total size of a single gRPC request that includes parameter values has an upper limit of about 5 MB.

`--input-batch-max-delay`	Maximum delay for sending a received parameter set for processing in adaptive batching mode. Specified as a number with a time dimension - `s` (seconds), `ms` (milliseconds), `m` (minutes), etc. Default value — `1s` (1 second). The YDB CLI will count time from the moment the first parameter set for a batch is received and will send the accumulated batch for execution as soon as the time exceeds the specified value. This parameter allows efficient batching in case of an unpredictable rate of new parameter sets arriving at `stdin`.

Examples - full batch processing

```
echo -e '{"a":10,"b":20}\n{"a":15,"b":25}\n{"a":35,"b":48}' | \
ydb -p quickstart sql \
  -s 'DECLARE $x AS List<Struct<a:Int64,b:Int64>>;
      SELECT ListLength($x), $x' \
  --input-framing newline-delimited \
  --input-param-name x \
  --input-batch full
```

Command output:

```
┌─────────┬───────────────────────────────────────────────────┐
| column0 | column1                                           |
├─────────┼───────────────────────────────────────────────────┤
| 3       | [{"a":10,"b":20},{"a":15,"b":25},{"a":35,"b":48}] |
└─────────┴───────────────────────────────────────────────────┘
```

Examples - adaptive batch processing

Limiting the maximum processing delay

To demonstrate adaptive batching with the processing delay limit trigger, the first line of the command below generates 1000 lines with a 0.2 second delay in `stdout`, which are passed to the `stdin` query execution command. The query execution command, in turn, displays parameter batches in each subsequent query call.

```
for i in $(seq 1 1000); do echo "Line$i"; sleep 0.2; done | \
ydb -p quickstart sql \
  -s 'DECLARE $x AS List<Utf8>;
      SELECT ListLength($x), $x' \
  --input-framing newline-delimited \
  --input-format raw \
  --input-param-name x \
  --input-batch adaptive
```

Command output (exact values may differ):

```
┌─────────┬──────────────────────────────────────────────────────────────────────────────────────────
──────────────────────┐
| column0 | column1
|
├─────────┼──────────────────────────────────────────────────────────────────────────────────────────
──────────────────────┤
| 14      | ["Line1","Line2","Line3","Line4","Line5","Line6","Line7","Line8","Line9","Line10","Line11","Line1
2","Line13","Line14"] |
└─────────┴──────────────────────────────────────────────────────────────────────────────────────────
──────────────────────┘

┌─────────┬─────────────────────────────────────────────────────┐
| column0 | column1                                             |
├─────────┼─────────────────────────────────────────────────────┤
| 6       | ["Line15","Line16","Line17","Line18","Line19","Line20"] |
└─────────┴─────────────────────────────────────────────────────┘

┌─────────┬─────────────────────────────────────────────────────┐
| column0 | column1                                             |
├─────────┼─────────────────────────────────────────────────────┤
| 6       | ["Line21","Line22","Line23","Line24","Line25","Line26"] |
└─────────┴─────────────────────────────────────────────────────┘
^C
```

The first batch includes all lines accumulated at the input during the database connection opening time, and therefore it is larger than subsequent ones.

You can interrupt the command execution with Ctrl+C, or wait until 200 seconds have passed, during which the input is generated.

Limiting by number of records

To demonstrate adaptive batching with the trigger by the number of parameter sets, the first line of the command below generates 200 lines. The command will display parameter batches in each subsequent query call, taking into account the specified limit `--input-batch-max-rows` equal to 20 (default 1000).

This example also shows the ability to combine parameters from different sources and generate JSON output.

```
for i in $(seq 1 200); do echo "Line$i"; done | \
ydb -p quickstart sql \
  -s 'DECLARE $x AS List<Utf8>;
      DECLARE $p2 AS Int64;
      SELECT ListLength($x) AS count, $p2 AS p2, $x AS items' \
  --input-framing newline-delimited \
  --input-format raw \
```

```
  --input-param-name x \
  --input-batch adaptive \
  --input-batch-max-rows 20 \
  --param '$p2=10' \
  --format json-unicode
```

Command output:

```
{"count":20,"p2":10,"items":["Line1","Line2","Line3","Line4","Line5","Line6","Line7","Line8","Line9","Line1
0","Line11","Line12","Line13","Line14","Line15","Line16","Line17","Line18","Line19","Line20"]}
{"count":20,"p2":10,"items":["Line21","Line22","Line23","Line24","Line25","Line26","Line27","Line28","Line2
9","Line30","Line31","Line32","Line33","Line34","Line35","Line36","Line37","Line38","Line39","Line40"]}
...
{"count":20,"p2":10,"items":["Line161","Line162","Line163","Line164","Line165","Line166","Line167","Line16
8","Line169","Line170","Line171","Line172","Line173","Line174","Line175","Line176","Line177","Line178","Line1
79","Line180"]}
{"count":20,"p2":10,"items":["Line181","Line182","Line183","Line184","Line185","Line186","Line187","Line18
8","Line189","Line190","Line191","Line192","Line193","Line194","Line195","Line196","Line197","Line198","Line1
99","Line200"]}
```

Deleting multiple records from a row table YDB by primary keys

If you try to delete a large number of rows from a large table using a simple `DELETE FROM large_table WHERE id > 10;` query, you may encounter an error due to exceeding the limit on the number of records in a transaction. This example shows how to delete an unlimited number of records from YDB tables without violating this limit.
Let's create a test row table:

```
ydb -p quickstart sql -s 'CREATE TABLE test_delete_1(id UInt64 NOT NULL, PRIMARY KEY (id))'
```

Let's insert 100,000 records into it:

```
for i in $(seq 1 100000); do echo "$i";done | \
ydb -p quickstart import file csv -p test_delete_1
```

Let's delete all records with `id` values greater than 10:

```
ydb -p quickstart sql \
  -s 'SELECT t.id FROM test_delete_1 AS t WHERE t.id > 10' \
  --format json-unicode | \
ydb -p quickstart sql \
  -s 'DECLARE $lines AS List<Struct<id:UInt64>>;
      DELETE FROM test_delete_1 WHERE id IN (SELECT tl.id FROM AS_TABLE($lines) AS tl)' \
  --input-framing newline-delimited \
  --input-param-name lines \
  --input-batch adaptive \
  --input-batch-max-rows 10000
```

Processing messages read from a topic

Examples of processing messages read from a topic are given in the article Running an SQL query with the transmission of messages from the topic as parameters.

See also

- Parameterized queries in YDB SDK

Interactive query execution mode

Overview

Executing the `ydb` command without subcommands launches the interactive query execution mode. After that, you can enter queries directly into the console or terminal. When you enter a newline character, the query text is considered complete, and query execution begins. The query text can be either a YQL query or a special command.

General format of the command:

```
ydb [global options...]
```

- `global options` — global parameters.

> **Note**
>
> Note that the command still requires connection parameters to be set. You can supply them by having the default profile, via an explicitly specified profile, or by passing a set of connection parameters.

Example usage:

The interactive query execution mode in YDB CLI offers the following features:

- Syntax highlighting
- Hotkeys
- Query history
- Auto completion
- Special commands
- AI assistant mode

Syntax highlighting

Interactive mode supports YQL syntax highlighting, which helps to better understand the query structure. Different colors are used for the following groups of elements:

- YQL keywords (`SELECT`, `FROM`, `WHERE`, `INSERT`, `UPDATE`, and others)
- Table and column names
- String literals (text in quotes)
- Numeric literals
- Operators (`=`, `<`, `>`, `+`, `-`, and others)
- Special characters (brackets, commas, dots)
- Comments

Hotkeys

You can use these hotkeys while working in the interactive mode:

Hotkey	Description
`Up arrow`	Shows the previous query from history.
`Down arrow`	Shows the next query from history.
`TAB`	Completes the current word based on YQL syntax.

`CTRL + R`	Searches for a query in history containing a specified substring.
`CTRL + T`	Switches between YQL and AI modes.
`CTRL + D`	Exits interactive mode.

Query history

You can navigate through the query history using the up and down arrow keys:

History is stored locally and persists between CLI launches.

A query search function (`CTRL + R`) is also supported:

Auto completion

Auto completion helps you write queries more efficiently. While typing, it suggests possible completions for the current word based on YQL syntax.

It also searches for schema object names in the database where possible.

There are two types of suggestions: auto completion by pressing the `TAB` key and interactive hints.

Auto completion by pressing the `TAB` key

While in interactive mode, pressing the `TAB` key shows a list of suggestions for completing the current word according to the YQL syntax.

Continue typing to narrow down the list of suitable candidates.

If there is only one available option, pressing `TAB` will automatically complete the current word.

If all available options share a common prefix, pressing `TAB` will automatically insert it.

Interactive hints

While typing in interactive mode, a list of hints appears under the cursor, showing the first four suggestions for completing the current word according to the YQL grammar.

This feature provides quick guidance without overwhelming you with all possible options, helping you stay on track while writing queries.

Special commands

Special commands are CLI-specific commands and are not part of the YQL syntax. Their purpose is to perform various functions that cannot be accomplished through a YQL query.

Command	Description
`SET param = value`	Sets the value of the internal variable `param` to `value`.
`EXPLAIN query-text`	Outputs the query plan for `query-text`. Equivalent to the command ydb table query explain.
`EXPLAIN AST query-text`	Outputs the query plan for `query-text` along with the AST. Equivalent to the command ydb table query explain --ast.

List of internal variables

Internal variables determine the behavior of commands and are set using the special command `SET`.

Variable	Description
`stats`	The statistics collection mode for subsequent queries. Acceptable values: • `none` (default): Do not collect. • `basic`: Collect statistics. • `full`: Collect statistics and query plan.

Example

Executing a query in the `full` statistics collection mode:

```
$ ydb
ydb> SET stats = full
ydb> select * from table1 limit 1
┌────┬─────┬───────┐
| id | key | value |
├────┼─────┼───────┤
| 10 | 0   | ""    |
└────┴─────┴───────┘

Statistics:
query_phases {
  duration_us: 14987
  table_access {
    name: "/ru-central1/a1v7bqj3vtf10qjleyow/laebarufb61tguph3g22/table1"
    reads {
      rows: 9937
      bytes: 248426
    }
  }
  cpu_time_us: 2925
  affected_shards: 1
}
process_cpu_time_us: 3816
total_duration_us: 79530
total_cpu_time_us: 6741


Full statistics:
Query 0:
ResultSet
└──Limit (Limit: 1)
   TotalCpuTimeUs: 175
   TotalTasks: 1
   TotalInputBytes: 6
   TotalInputRows: 1
   TotalOutputBytes: 16
   TotalDurationMs: 0
   TotalOutputRows: 1
   └──<UnionAll>
      └──Limit (Limit: 1)
      └──TableFullScan (ReadColumns: ["id","key","value"], ReadRanges: ["key (-∞, +∞)"], Table: impex_table)
         Tables: ["table1"]
         TotalCpuTimeUs: 154
         TotalTasks: 1
         TotalInputBytes: 0
         TotalInputRows: 0
         TotalOutputBytes: 16
         TotalDurationMs: 0
         TotalOutputRows: 1
```

AI assistant mode

The interactive mode includes an AI assistant mode — a conversational agent powered by a large language model that helps you explore the database structure, write and run YQL queries, search the documentation, and invoke YDB CLI commands. To switch between YQL and AI modes, press `Ctrl + T` or use the `/switch` command.

For more details, see AI assistant mode.

AI assistant mode

The YDB CLI interactive query execution mode includes an AI assistant mode — a conversational agent powered by a large language model (LLM). The assistant understands natural-language requests and helps you work with the database: explore its structure, write and run YQL queries, explain query plans, search the documentation, and invoke YDB CLI commands.

> **Warning**
>
> The assistant calls an external large language model API that you specify yourself. The conversation contents, including the schema structure and query results that the assistant chooses to pass to the model, are sent to that API. Do not use this mode with sensitive data if your policies prohibit it. The YDB CLI does not provide a language model or an access key — you must obtain them from the provider of your choice.

Switching to AI mode

The interactive mode runs in one of two submodes:

- **YQL** — entering and running YQL queries (used by default on the first launch);
- **AI** — a dialog with the AI assistant.

Start the interactive mode by running the `ydb` command without a subcommand and with the required connection parameters:

```
ydb [global options...]
```

The active submode is shown in the command-line prompt (`YQL>` or `AI>`). To switch between submodes, press `Ctrl + T` or use the `/switch` command. The YDB CLI remembers the last used submode and opens it on the next launch.

Setting up an AI profile

The assistant requires an AI profile — a set of connection parameters for the language model API. The first time you switch to AI mode, if no profiles exist yet, the YDB CLI offers to create one and guides you through the setup step by step:

Parameter	Description
API endpoint	The URL of the language model API. It must start with `http://` or `https://` (for example, `https://api.openai.com/v1/`). The YDB CLI verifies the connection to the specified address.
API type	The API type: • **OpenAI-compatible** — for models with an OpenAI-compatible API; • **Anthropic** — for the Anthropic API.
API token	The API access token. It is stored in the configuration file. You can leave the field empty and pass the token via the `YDB_CLI_AI_TOKEN` environment variable.
Model name	The model name. If the API allows it, the YDB CLI requests the list of available models; the name can also be entered manually. An empty value means the model name is not passed in requests.
Profile display name	The display name under which the profile is saved and shown during selection.

After setup, the profile is saved to the configuration file and becomes active. To switch the active profile or create a new one, use the `/model` command.

Working with the assistant

In AI mode, type a natural-language message and press `Enter` to send it (`Ctrl + Enter` inserts a newline). The assistant analyzes your request and, when needed, calls tools to work with the database:

Capability	Modifies data	Confirmation
Listing schema objects under a given path	No	Not required
Describing a schema object (table, topic, and so on)	No	Not required
Getting a YQL query plan (`EXPLAIN`)	No	Not required
Searching the YDB documentation	No	Not required
Getting help on YDB CLI commands	No	Not required
Running a YQL query	Yes	Required
Running a shell command	Yes	Required

The assistant keeps the conversation context: subsequent messages take previous ones into account. You can clear the context with the `/config` command.

Action confirmation

Before performing an action that may modify data — running a YQL query or a shell command — the assistant asks for confirmation and offers the following options:

- **Approve execution** — run the query or command as is;
- **Edit query** / **Edit command** — edit the query or command text before running it;
- **Abort operation** — cancel the execution (and continue working in the same context).

Commands

The following commands are available in AI mode:

Command	Description
`/help`	Show help for AI mode and the list of hotkeys.
`/model`	Select another AI profile or create a new one.
`/config`	Change session settings: clear the conversation context, switch or edit the active profile, remove a profile.
`/switch`	Switch to YQL mode (equivalent to `Ctrl + T`).

Configuration file

AI profiles and AI mode settings are stored in a YAML file. By default it is `~/.config/ydb/ai_profiles.yaml`; the path can be overridden with the `YDB_CLI_AI_PROFILE_FILE` environment variable. The file contains the list of profiles, the ID of the active profile, and the current interactive submode. You usually do not need to edit it manually — manage profiles via the `/model` and `/config` commands.

Note

A token entered directly during profile setup is stored in the configuration file. The YDB CLI restricts the file's permissions so that only the owner — the user who runs the CLI — can read and modify it (on Unix systems, mode `0600`). If you prefer not to store the token on disk, leave the token field empty and pass it via the `YDB_CLI_AI_TOKEN` environment variable.

Environment variables

Variable	Description
`YDB_CLI_AI_TOKEN`	The access token for the language model API. Used if the token is not set in the active profile.
`YDB_CLI_AI_PROFILE_FILE`	The path to the AI profiles configuration file. Defaults to `~/.config/ydb/ai_profiles.yaml`. Lets you use a separate profile set or a non-default file location, for example in automated scenarios.
`YDB_CLI_AI_DISABLE_HISTORY`	If set, the AI mode message history is not stored on disk between launches. Use it when the conversation with the assistant should not be persisted to disk.

Running scan queries

> **Warning**
>
> Using `ScanQuery` is not recommended for new scenarios. Use standard mechanisms to execute queries.
>
> For more details, see the Executing queries section.
>
> The exception is executing long-running (over 5 minutes) queries against row tables. In this case, we still recommend using `ScanQuery`, as standard query execution mechanisms do not yet fully support this scenario.

To run a query via Scan Queries using YDB CLI, add the `-t scan` flag to the `ydb table query execute` command.

Run a query against the data:

```
ydb table query execute -t scan \
 --query "SELECT season_id, episode_id, title \
 FROM episodes \
 WHERE series_id = 1 AND season_id > 1 \
 ORDER BY season_id, episode_id \
 LIMIT 3"
```

Where:

- `--query` — query text.

Result:

```
┌───────────┬────────────┬──────────────────────────────────┐
| season_id | episode_id | title |
├───────────┼────────────┼──────────────────────────────────┤
| 2 | 1 | "The Work Outing" |
├───────────┼────────────┼──────────────────────────────────┤
| 2 | 2 | "Return of the Golden Child" |
├───────────┼────────────┼──────────────────────────────────┤
| 2 | 3 | "Moss and the German" |
└───────────┴────────────┴──────────────────────────────────┘
```

Getting a list of background operations

Using the `ydb operation list` subcommand, you can get a list of background operations of the specified type.

General command format:

```
ydb [global options...] operation list [options...] <kind>
```

- `global options` — global parameters.
- `options` — subcommand parameters.
- `kind` — operation type. Possible values:
  - `analyze` — ANALYZE statistics collection operations
  - `buildindex` — index building operations
  - `compaction` — table compaction operations
  - `export/s3` — export to S3 operations
  - `export/nfs` — export to NFS operations
  - `import/s3` — import from S3 operations
  - `import/nfs` — import from NFS operations
  - `scriptexec` — script execution operations
  - `incbackup` — incremental backup operations
  - `restore` — restore from backup operations
  - `setnotnull` — operations for setting the `NOT NULL` limit.

View the description of the command for getting a list of background operations:

```
ydb operation list --help
```

Subcommand parameters

Name	Description
`-s`, `--page-size`	Number of operations per page. If the operation list contains more rows than specified in the `--page-size` parameter, the output will be split into multiple pages. To get the next page, specify the `--page-token` parameter.
`-t`, `--page-token`	Page token.
`--format`	Output format. Default value — `pretty`. Possible values: • `pretty` — human-readable format • `proto-json-base64` — Protobuf output in JSON format, binary strings are encoded in Base64.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Get a list of background index building operations for the `series` table:

```
ydb -p quickstart operation list \
  buildindex
```

Result:

```
┌──────────────────────────────────────────┬───────┬──────────┬───────┬──────────┬──────────────────────┬──────────
────┐
| id                                       | ready | status   | state | progress | table                | index
|
├──────────────────────────────────────────┼───────┼──────────┼───────┼──────────┼──────────────────────┼──────────
────┤
| ydb://buildindex/7?id=281489389055514 | true  | SUCCESS | Done  | 100.00%  | /my-database/series | idx_rele
ase |
└──────────────────────────────────────────┴───────┴──────────┴───────┴──────────┴──────────────────────┴──────────
────┘

Next page token: 0
```

Obtaining the status of long-running operations

Use the `ydb operation get` subcommand to obtain the status of the specified long-running operation.

General format of the command:

```
ydb [global options...] operation get [options...] <id>
```

- `global options`: Global parameters.
- `options`: Parameters of the subcommand.
- `id`: The ID of the long-running operation. The ID contains characters that can be interpreted by your command shell. If necessary, use shielding, for example, `'<id>'` for bash.

View a description of the command to obtain the status of a long-running operation:

```
ydb operation get --help
```

Parameters of the subcommand

Name	Description
`--format`	Input format. Default value: `pretty`. Acceptable values: • `pretty`: A human-readable format. • `proto-json-base64`: Protobuf result in JSON format, binary strings are encoded in Base64.

Examples

Note

The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Obtain the status of the long-running operation with the `ydb://buildindex/7?id=281489389055514` ID:

```
ydb -p quickstart operation get \
  'ydb://buildindex/7?id=281489389055514'
```

Result:

```
┌──────────────────────────────────────────┬───────┬──────────┬───────┬──────────┬──────────────────────┬──────────
────┐
| id                                       | ready | status   | state | progress | table                | index
|
├──────────────────────────────────────────┼───────┼──────────┼───────┼──────────┼──────────────────────┼──────────
────┤
| ydb://buildindex/7?id=281489389055514 | true  | SUCCESS | Done  | 100.00%  | /my-database/series | idx_rele
ase |
└──────────────────────────────────────────┴───────┴──────────┴───────┴──────────┴──────────────────────┴──────────
────┘
```

Canceling long-running operations

Use the `ydb operation cancel` subcommand to cancel the specified long-running operation. Only an incomplete operation can be canceled.

General format of the command:

```
ydb [global options...] operation cancel <id>
```

- `global options`: Global parameters.
- `id`: The ID of the long-running operation. The ID contains characters that can be interpreted by your command shell. If necessary, use shielding, for example, `'<id>'` for bash.

View a description of the command to obtain the status of a long-running operation:

```
ydb operation cancel --help
```

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Deleting long running operations from the list

Use the `ydb operation forget` subcommand to delete information about the specified long running operation from the list. The operation must be complete.

General format of the command:

```
ydb [global options...] operation forget <id>
```

- `global options` : Global parameters.
- `id` : The ID of the long running operation. The ID contains characters that can be interpreted by your command shell. If necessary, use shielding, for example, `'<id>'` for bash.

View a description of the command to delete information about the specified long running operation:

```
ydb operation forget --help
```

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Delete the long running operation with the `ydb://buildindex/7?id=281489389055514` ID from the list:

```
ydb -p db1 operation forget \
  'ydb://buildindex/7?id=281489389055514'
```

Managing profiles

A profile is a named set of DB connection parameters stored in a configuration file in the local file system. With profiles, you can reuse data about DB location and authentication parameters, making a CLI call much shorter:

- Calling the `scheme ls` command without a profile:

```
ydb \
-e grpsc://some.host.in.some.domain:2136 \
-d /some_long_identifier1/some_long_identifier2/database_name \
--yc-token-file ~/secrets/token_database1 \
scheme ls
```

- Calling the same `scheme ls` command using a profile:

```
ydb -p quickstart scheme ls
```

Profile management commands

- Creating a profile
- Using a profile
- Getting a list of profiles and profile parameters
- Deleting a profile
- Activating a profile and using the activated profile

Where profiles are stored

Profiles are stored locally in a file named `~/.ydb/config/config.yaml`.

Creating and updating profiles

You can set connection parameter values for the profile being created or updated through the command line or request them in interactive mode in the console.

Command line

To create or update a profile, the command line uses the `profile create`, `profile update`, and `profile replace` commands.

They only use the values entered directly on the command line without accessing environment variables or the activated profile.

Profile create

`Profile create` creates a new profile with the specified parameter values:

```
ydb config profile create <profile_name> <connection_options>
```

Where:

- `<profile_name>` is the required profile name.
- `<connection options>` are connection parameters to be written to the profile. You need to specify at least one connection parameter; otherwise the command will run in interactive mode.

If a profile with the specified name exists, the command will return an error.

Profile replace

`Profile replace` creates or replaces a profile with the specified parameter values:

```
ydb config profile replace <profile_name> [connection_options]
```

Where:

- `<profile_name>` is the required profile name.
- `<connection options>` are optional connection parameters to be written to the profile.

If a profile with the specified name already exists, it will be overwritten with a new one with the passed-in parameters. If you specify no connection parameters, the profile will be empty once the command completes.

Profile update

`Profile update` modifies the properties of an existing profile:

```
ydb config profile update <profile_name> [connection_options] [reset-options]
```

Where:

- `<profile_name>` is the required profile name.
- `<connection options>` are optional connection parameters to be written to the profile.
- `<reset options>` are optional settings for deleting parameters from an existing profile. Possible values:
  - `--no-endpoint`: Delete an endpoint from the profile
  - `--no-database`: Delete the database path from the profile
  - `--no-auth`: Delete authentication information from the profile
  - `--no-iam-endpoint`: Delete the IAM server URL

The profile will update with the parameters entered on the command line. Any properties not listed on the command line will remain unchanged.

Examples

Creating a profile to connect to a test database

To connect to a DB in a single-node YDB cluster, you can use the `quickstart` profile:

```
ydb config profile create quickstart --endpoint grpc://localhost:2136 --database <path_database>
```

- `path_database`: Database path. Specify one of these values:
  - `/Root/test`: If you used an executable to deploy your cluster.
  - `/local`: If you deployed your cluster from a Docker image.

Creating a profile from previous connection settings

Any command with explicit connection settings performing a YDB database transaction can be converted to a profile create command by moving connection properties from global options to options specific to the `config profile create` command.

For instance, if you successfully ran the `scheme ls` command with the following properties:

```
ydb \
  -e grpcs://example.com:2135 -d /Root/somedatabase --sa-key-file ~/sa_key.json \
  scheme ls
```

You can create a profile to connect to the accessed database using the following command:

```
ydb \
  config profile create db1 \
```

```
  -e grpcs://example.com:2135 -d /Root/somedatabase --sa-key-file ~/sa_key.json
```

You can now use much shorter syntax to re-write the original command:

```
ydb -p db1 scheme ls
```

Profile to connect to a local database

Creating/replacing a `local` profile to connect to a local YDB database deployed using quick start:

```
ydb config profile replace local --endpoint grpc://localhost:2136 --database /Root/local
```

Defining the login and password authentication method in the `local` profile:

```
ydb config profile update local --user user1 --password-file ~/pwd.txt
```

Interactive mode

You can use the commands below to create and update profiles in interactive mode:

```
ydb init
```

or

```
ydb config profile create [profile_name] [connection_options]
```

Where:

- `[profile_name]` is an optional name of the profile to create or update.
- `[connection_options]` are optional connection settings to write to the profile.

The `init` command always runs in interactive mode while `config profile create` launches in interactive mode unless you specify a profile name or none of the connection settings on the command line.

The interactive scenario starts differently for the `init` and the `profile create` commands:

Init

1. Prints a list of existing profiles (if any) and prompts you to make a choice: Create a new or update the configuration of an existing profile:

   ```
   Please choose profile to configure:
   [1] Create a new profile
   [2] test
   [3] local
   ```

2. If no profiles exist or you select option `1` in the previous step, the name of a profile to create is requested:

   ```
   Please enter name for a new profile:
   ```

3. If you enter the name of an existing profile at this point, the YDB CLI proceeds to updating its parameters as if an option with the name of this profile was selected at once.

Profile Create

If no profile name is specified on the command line, it is requested:

```
Please enter configuration profile name to create or re-configure:
```

Next, you'll be prompted to sequentially perform the following actions with each connection parameter that can be saved in the profile:

- Don't save
- Set a new value or Use
- Use current value (this option is available when updating an existing profile)

Example

Creating a new `mydb1` profile:

1. Run this command:

   ```
   ydb config profile create mydb1
   ```

2. Enter the endpoint or don't save this parameter for the profile:

   ```
   Pick desired action to configure endpoint:
    [1] Set a new endpoint value
    [2] Don't save endpoint for profile "mydb1"
   Please enter your numeric choice:
   ```

3. Enter the database name or don't save this parameter for the profile:

```
Pick desired action to configure database:
 [1] Set a new database value
 [2] Don't save database for profile "mydb1"
Please enter your numeric choice:
```

4. Select the authentication mode or don't save this parameter for the profile:

```
Pick desired action to configure authentication method:
  [1] Use static credentials (user & password)
  [2] Use IAM token (iam-token) yandex.cloud/docs/iam/concepts/authorization/iam-token
  [3] Use OAuth token of a Yandex Passport user (yc-token). Doesn't work with federative accounts. yande
x.cloud/docs/iam/concepts/authorization/oauth-token
  [4] Use metadata service on a virtual machine (use-metadata-credentials) yandex.cloud/docs/compute/ope
rations/vm-connect/auth-inside-vm
  [5] Use service account key file (sa-key-file) yandex.cloud/docs/iam/operations/iam-token/create-for-s
a
  [6] Set new access token (ydb-token)
  [7] Don't save authentication data for profile "mydb1"
Please enter your numeric choice:
```

   All the available authentication methods are described in Authentication. The set of methods and text of the hints may differ from those given in this example.

   If the method you choose involves specifying an additional parameter, you'll be prompted to enter it. For example, if you select `4` (Use service account key file):

```
Please enter Path to service account key file (sa-key-file):
```

5. In the last step, you'll be prompted to activate the created profile to be used by default. Choose 'n' (No) until you read the article about Activating a profile and using the activated profile:

```
Activate profile "mydb1" to use by default? (current active profile is not set) y/n: n
```

Using a profile

Connection based on a selected profile

A profile can be applied when running a YDB CLI command with the `--profile` or the `-p` option:

```
ydb -p <profile_name> <command and command options>
```

For example:

```
ydb -p mydb1 scheme ls -l
```

In this case, all DB connection parameters are taken from the profile. At the same time, if the authentication parameters are not specified in the profile, the YDB CLI will try to define them based on environment variables, as described in Connecting to and authenticating with a database — Environment variable.

Connection based on a selected profile and specified command line parameters

The `--profile` (`-p`) option doesn't need to be the only connection setting specified on the command line. For example:

```
ydb -p mydb1 -d /local2 scheme ls -l
```

```
ydb -p mydb1 --user alex scheme ls -l
```

In this case, the connection parameters specified on the command line have priority over those stored in the profile. This format lets you reuse profiles to connect to different databases or under different accounts. In addition, specifying the authentication parameter on the command line (such as `--user alex` in the example above) disables environment variable checks regardless of their presence in the profile.

Connection based on an activated profile

If the `--profile` (`-p`) option is not specified on the command line, the YDB CLI will attempt to take all the connection parameters that it couldn't otherwise determine (from the command-line options or environment variables, as described in Connecting to and authenticating with a database) from the currently activated profile.

Implicit use of the activated profile may cause errors, so we recommend that you read the Activated profile article before using this mode.

Getting profile information

Getting a list of profiles

Getting a list of profiles:

```
ydb config profile list
```

If there is a currently activated profile, it will be marked as `(active)` in the output list, for example:

```
prod
test (active)
local
```

Getting detailed profile information

Getting parameters saved in the specified profile:

```
ydb config profile get <profile_name>
```

For example:

```
$ ydb config profile get local1
  endpoint: grpcs://ydb.serverless.yandexcloud.net:2135
  database: /rul1/b1g8skp/etn02099
  sa-key-file: /Users/username/secrets/sa_key_test.json
```

Getting profiles with content

Full information on all profiles and parameters stored in them:

```
ydb config profile list --with-content
```

The output of this command combines the output of the command to get a list of profiles (with the active profile marked) and the parameters of each profile in the lines following its name.

Deleting a profile

Currently, you can only delete profiles interactively with the following command:

```
ydb config profile delete <profile_name>
```

where `<profile_name>` is the profile name.

The YDB CLI will request confirmation to delete the profile:

```
Profile "<profile_name>" will be permanently removed. Continue? (y/n):
```

Choose `y` (Yes) to delete the profile.

Example

Deleting the `mydb1` profile:

```
$ ydb config profile delete mydb1
Profile "mydb1" will be permanently removed. Continue? (y/n): y
Profile "mydb1" was removed.
```

Deleting a profile without interactive input

Although this mode is not supported by the YDB CLI, if necessary, you can use input redirection in your OS to automatically respond `y` to the request to confirm the deletion:

```
echo y | ydb config profile delete my_profile
```

The efficiency of this method is not guaranteed in any way.

Activated profile

Executing YDB CLI commands on a database require establishing a connection to the database. If the YDB CLI couldn't identify a certain connection parameter by command-line parameters and environment variables, it's taken from the activated profile.

Profile activation is an easy way to get started with the YDB CLI, since the connection parameters that are set once will be applied automatically to any command, without the need to specify any connection parameters in the command line.

However, this simplicity may lead to undesirable behavior in further operation, as soon as you need to work with multiple databases:

- The activated profile is applied implicitly, meaning that it can be applied by mistake when a certain connection parameter is missing in the command line.
- The activated profile is applied implicitly, meaning that it can be applied by mistake when a typo is made in the name of an environment variable.
- The activated profile cannot be used in scripts, since it is saved in a file and its change in one terminal window will affect all other windows, possibly leading to an unexpected change of the DB in the middle of the loop being executed in the script.

When you need to connect to any new database other than the initial one for the first time, we recommend that you deactivate the profile and always select it explicitly using the `--profile` option.

Activating a profile with a command

Profile activation is performed by running the command

```
ydb config profile activate [profile_name]
```

where `[profile_name]` is an optional profile name.

If the profile name is specified, it is activated. If a profile with the specified name does not exist, an error is returned prompting you to view the list of available profiles:

```
No existing profile "<profile_name>". Run "ydb config profile list" without arguments to see existing profile
s
```

If the profile name is not specified, you'll be asked to choose between the following options in interactive mode:

```
Please choose profile to activate:
 [1] Don't do anything, just exit
 [2] Deactivate current active profile (if any)
 [3] <profile_name_1> (active)
 [4] <profile_name_2>
 ...
Please enter your numeric choice:
```

- `1` terminates the command execution and keeps the currently activated profile activated. It's marked as `(active)` in the list of existing profiles starting from item 3.
- `2` deactivates the currently activated profile. If no profile has been activated before, nothing changes.
- `3` and so on activates the selected profile. The currently activated profile is marked as `(active)`.

If the profile is successfully activated, the execution ends with a message saying

```
Profile "<profile_name>" was activated.
```

Example

Activating a profile named `mydb1`:

```
$ ydb config profile activate mydb1
Profile "mydb1" was activated.
```

Activating a profile during its initialization

As the last step during the interactive execution of the command to create or update the profile `ydb config profile create`, you're prompted to activate the created (or updated) profile:

```
Activate profile "<profile_name>" to use by default? (current active profile is not set) y/n:
```

Choose `y` (Yes) to activate the profile.

Deactivating a profile

Currently, the YDB CLI only supports profile deactivation in interactive mode, when calling the activation command without specifying the profile (choosing item `2` in the above activation command).

If necessary, you can use input redirection in your OS to automatically select option `2` in interactive input:

```
echo 2 | ydb config profile activate
```

The efficiency of this method is not guaranteed in any way.

List of endpoints

Using the `discovery list` information command, you can get a list of YDB cluster endponts that you can connect to in order to access your database:

```
ydb [connection options] discovery list
```

where [connection options] are database connection options

The output rows in the response contain the following information:

1. Endpoint, including protocol and port
2. Availability zone (in square brackets)
3. The `#` character is used for the list of YDB services available on this endpoint

An endpoint discovery request to the YDB cluster is executed in the YDB SDK at driver initialization so that you can use the `discovery list` CLI command to localize connection issues.

Example

```
$ ydb -p quickstart discovery list
grpcs://vm-etn01q5-ysor.etn01q5k.ydb.mdb.yandexcloud.net:2135 [sas] #table_service #scripting #discovery #rate_limiter #locking #kesus
grpcs://vm-etn01q5-arum.etn01ftr.ydb.mdb.yandexcloud.net:2135 [vla] #table_service #scripting #discovery #rate_limiter #locking #kesus
grpcs://vm-etn01q5beftr.ydb.mdb.yandexcloud.net:2135 [myt] #table_service #scripting #discovery #rate_limiter #locking #kesus
```

Authentication

The `discovery whoami` information command lets you check the account on behalf of which the server actually accepts requests:

```
ydb [connection options] discovery whoami [-g|--groups] [-l|--access-list] [-a|--all]
```

where [connection options] are database connection options

The response includes the account name (User SID). Additional information can be requested using the following options:

- `-g`, `--groups` — Show the groups to which the account belongs
- `-l`, `--access-list` — Show the access levels granted to the account (administration, monitoring, viewer, database, register node, bootstrap)
- `-a`, `--all` — Show all additional information (equivalent to specifying both `-g` and `-l`)

The access levels displayed with the `-l` or `-a` option reflect the hierarchical access control configuration. For detailed information about access level lists and their hierarchy, see Access level lists in the Authorization documentation.

Only access levels that are granted to the user are listed:

- **Database** (presence in `database_allowed_sids`) — Grants the right to access the YDB UI only as "database users": they can open the UI and see database-scoped data, but not cluster-wide data or cluster-level operations.
- **Viewer** (presence in `viewer_allowed_sids`) — Grants the right to access the YDB UI, without the ability to make changes.
- **Monitoring** (presence in `monitoring_allowed_sids`) — Grants the right to perform actions in the YDB UI that change the system state.
- **Administration** (presence in `administration_allowed_sids`) — Grants the right to perform administrative actions on databases or the cluster.
- **Register node** (presence in `register_dynamic_node_allowed_sids`) — Grants the right to register dynamic nodes with the cluster.
- **Bootstrap** (presence in `bootstrap_allowed_sids`) — Grants the right to perform bootstrap operations.

If authentication is not enabled on the YDB server (for example, in the case of an independent local deployment), the command will fail with an error.

Support for the `-g` option depends on the server configuration. If disabled, you'll receive `User has no groups` in response, regardless of the actual inclusion of your account in any groups.

Examples

Basic usage

```
$ ydb -p quickstart discovery whoami
User SID: aje5kkjdgs0puc18976co@as
```

With groups

```
$ ydb -p quickstart discovery whoami -g
User SID: aje5kkjdgs0puc18976co@as

User has no groups
```

With access list

```
$ ydb -p quickstart discovery whoami -l
User SID: user1@builtin

Access levels:
Database
Viewer
```

With all information

```
$ ydb -p quickstart discovery whoami -a
User SID: admin@builtin

Group SIDs:
all-users@well-known
ADMINS

Access levels:
Database
Viewer
Monitoring
Administration
```

Displaying connection parameters

`config info` is a service command for debugging various issues with connection and authentication. The command displays the final connection parameters, such as the endpoint and database path, obtained by considering parameters from all possible sources. With these parameters, the CLI would connect when executing a command that implies a connection to the database. When specifying the global option `-v`, in addition to the final connection parameters, values of all connection parameters from all sources that the CLI managed to detect will be displayed, along with the names of those sources in order of priority

General format of the command:

```
ydb [global options...] config info
```

- `global options` — global parameters.

Connection parameter list

- endpoint — URL of database cluster.
- database — Database path.
- Authentication parameters:
  - token — Access Token.
  - yc-token — Refresh Token.
  - sa-key-file — Service Account Key.
  - use-metadata-credentials — Metadata.
  - user
  - password
- ca-file — Root certificate.
- iam-endpoint — URL of IAM service.

Examples

Display final connection parameters

```
$ ydb -e grpcs://another.endpoint:2135 --ca-file some_certs.crt -p db123 config info
endpoint: another.endpoint:2135
yc-token: SOME_A12****************21_TOKEN
iam-endpoint: iam.api.cloud.yandex.net
ca-file: some_certs.crt
```

Display all connection parameters along with their sources

```
$ ydb -e grpcs://another.endpoint:2135 --ca-file some_certs.crt -p db123 -v config info
Using Yandex.Cloud Passport token from YC_TOKEN env variable

endpoint: another.endpoint:2135
yc-token: SOME_A12****************21_TOKEN
iam-endpoint: iam.api.cloud.yandex.net
ca-file: some_certs.crt
current auth method: yc-token

"ca-file" sources:
  1. Value: some_certs.crt. Got from: explicit --ca-file option

"database" sources:
  1. Value: /some/path. Got from: active profile "test_config_info"

"endpoint" sources:
  1. Value: another.endpoint:2135. Got from: explicit --endpoint option
  2. Value: db123.endpoint:2135. Got from: profile "db123" from explicit --profile option
  3. Value: some.endpoint:2135. Got from: active profile "test_config_info"

"iam-endpoint" sources:
  1. Value: iam.api.cloud.yandex.net. Got from: default value

"sa-key-file" sources:
  1. Value: /Users/username/some-sa-key-file. Got from: active profile "test_config_info"

"yc-token" sources:
  1. Value: SOME_A12****************21_TOKEN. Got from: YC_TOKEN enviroment variable
```

Getting YDB CLI version

Use the `version` subcommand to find out the version of the YDB CLI installed and manage new version availability auto checks.

New version availability auto checks are made when you run any YDB CLI command, except `ydb version --enable-checks` and `ydb version --disable-checks`, but only once in 24 hours. The result and time of the last check are saved to the YDB CLI configuration file.

General format of the command:

```
ydb [global options...] version [options...]
```

- `global options`: Global parameters.
- `options`: Parameters of the subcommand.

View a description of the command:

```
ydb version --help
```

Parameters of the subcommand

Parameter	Description
`--semantic`	Get only the version number.
`--check`	Check if a new version is available.
`--disable-checks`	Disable new version availability checks.
`--enable-checks`	Enable new version availability checks.

Examples

Disable new version availability checks

When running YDB CLI commands, the system automatically checks if a new version is available. If the host where the command is run doesn't have internet access, this causes a delay and the corresponding warning appears during command execution. To disable auto checks for updates, run:

```
ydb version --disable-checks
```

Result:

```
Latest version checks disabled
```

Getting only the version number

To facilitate data handling in scripts, you can limit result to the YDB CLI version number:

```
ydb version --semantic
```

Result:

```
1.9.1
```

Health check

YDB has a built-in self-diagnostic system that provides a brief report on the cluster status and information about existing issues. This report can be obtained via YDB CLI using the command explained below.

General command format:

```
ydb [global options...] monitoring healthcheck [options...]
```

- `global options` — global options,
- `options` — subcommand options.

Subcommand options

Name	Description
`--timeout`	The time, in milliseconds, within which the operation should be completed on the server.
`--format`	Output format. Available options: • `pretty` — overall database status. Possible values are provided in the table. • `json` — a detailed JSON response containing a hierarchical list of detected problems. Possible issues are listed in the Healthcheck API documentation. Default: `pretty`.
`--no-merge`	Do not merge entries of health check result.
`--no-cache`	Do not use cached result.

Examples

Health check result in pretty format

```
ydb --profile quickstart monitoring healthcheck --format pretty
```

Database is in good condition:

```
Healthcheck status: GOOD
```

Database is degraded:

```
Healthcheck status: DEGRADED
```

Health check result in JSON format

```
ydb --profile quickstart monitoring healthcheck --format json
```

Database is in good condition:

```
{
 "self_check_result": "GOOD",
 "location": {
  "id": 51059,
  "host": "my-host.net",
  "port": 19001
 }
}
```

Database is degraded:

```
{
 "self_check_result": "DEGRADED",
 "issue_log": [
  {
   "id": "YELLOW-b3c0-70fb",
   "status": "YELLOW",
   "message": "Database has multiple issues",
   "location": {
    "database": {
     "name": "/my-cluster/my-database"
    }
   },
   "reason": [
    "YELLOW-b3c0-1ba8",
    "YELLOW-b3c0-1c83"
   ],
   "type": "DATABASE",
   "level": 1
  },
  {
   "id": "YELLOW-b3c0-1ba8",
   "status": "YELLOW",
   "message": "Compute is overloaded",
```

```
    "location": {
     "database": {
      "name": "/my-cluster/my-database"
     }
    },
    "reason": [
     "YELLOW-b3c0-343a-51059-User"
    ],
    "type": "COMPUTE",
    "level": 2
   },
   {
    "id": "YELLOW-b3c0-343a-51059-User",
    "status": "YELLOW",
    "message": "Pool usage is over than 99%",
    "location": {
     "compute": {
      "node": {
       "id": 51059,
       "host": "my-host.net",
       "port": 31043
      },
      "pool": {
       "name": "User"
      }
     },
     "database": {
      "name": "/my-cluster/my-database"
     }
    },
    "type": "COMPUTE_POOL",
    "level": 4
   },
   {
    "id": "YELLOW-b3c0-1c83",
    "status": "YELLOW",
    "message": "Storage usage over 75%",
    "location": {
     "database": {
      "name": "/my-cluster/my-database"
     }
    },
    "type": "STORAGE",
    "level": 2
   }
  ],
  "location": {
   "id": 117,
   "host": "my-host.net",
   "port": 19001
  }
}
```

Load testing

You can use the `workload` command to run different types of workload against your DB.

General format of the command:

```
ydb [global options...] workload [subcommands...]
```

- `global options` : Global options.
- `subcommands` : The subcommands.

See the description of the command to run the data load:

```
ydb workload --help
```

Available subcommands

The following types of load tests are supported at the moment:

- Stock: An online store warehouse simulator.
- Key-value: Key-Value load test.
- ClickBench: ClickBench analytical benchmark.
- TPC-C: TPC-C benchmark.
- TPC-H: TPC-H benchmark.
- TPC-DS: TPC-DS benchmark.
- Topic: Topic load test.
- Transfer: Transfer load test.
- Query: Query load test.
- Fulltext: Fulltext load test.
- Vector: Vector search load test.

Global parameters for all workloads

Parameter name	Short name	Parameter description
`--seconds <value>`	`-s <value>`	Duration of the test, in seconds. Default: 10.
`--threads <value>`	`-t <value>`	The number of parallel threads creating the load. Default: 10.
`--rate <value>`	-	Total rate for all threads, in transactions per second. Default: 0 (no rate limit).
`--quiet`	-	Outputs only the total result.
`--print-timestamp`	-	Prints the time together with the statistics of each time window.
`--client-timeout`	-	Transport timeout in milliseconds.
`--operation-timeout`	-	Operation timeout in milliseconds.
`--cancel-after`	-	Timeout for canceling an operation in milliseconds.
`--window`	-	Statistics collection window, in seconds. Default: 1.

Stock load

Simulates a warehouse of an online store: creates multi-product orders, gets a list of orders per customer.

Types of load

This load test runs 5 types of load:

- user-hist: Reads the specified number of orders made by the customer with id = 10000. This creates a workload to read the same rows from different threads.
- rand-user-hist: Reads the specified number of orders made by a randomly selected customer. A load that reads data from different threads is created.
- add-rand-order: Generates an order at random. For example, a customer has created an order of 2 products, but hasn't yet paid for it, hence the quantities in stock aren't decreased for the products. The database writes the data about the order and products. The read/write load is created (the INSERT checks for an existing entry before inserting the data).
- put-rand-order: Generates an order at random and processes it. For example, a customer has created and paid an order of 2 products. The data about the order and products is written to the database, product availability is checked and quantities in stock are decreased. A mixed data load is created.
- put-same-order: Creates orders with the same set of products. For example, all customers buy the same set of products (a newly released phone and a charger). This creates a workload of competing updates of the same rows in the table.

Load test initialization

To get started, create tables and populate them with data:

```
ydb workload stock init [init options...]
```

- `init options`: Initialization options.

See the description of the command to init the data load:

```
ydb workload stock init --help
```

Available parameters

Parameter name	Short name	Parameter description
`--products <value>`	`-p <value>`	Number of products. Valid values: between 1 and 500000. Default: 100.
`--quantity <value>`	`-q <value>`	Quantity of each product in stock. Default: 1000.
`--orders <value>`	`-o <value>`	Initial number of orders in the database. Default: 100.
`--min-partitions <value>`	-	Minimum number of shards for tables. Default: 40.
`--auto-partition <value>`	-	Enabling/disabling auto-sharding. Possible values: 0 or 1. Default: 1.

3 tables are created using the following DDL statements:

```
CREATE TABLE `stock`(product Utf8, quantity Int64, PRIMARY KEY(product)) WITH (AUTO_PARTITIONING_BY_LOAD = EN
ABLED, AUTO_PARTITIONING_MIN_PARTITIONS_COUNT = <min-partitions>);
CREATE TABLE `orders`(id Uint64, customer Utf8, created Datetime, processed Datetime, PRIMARY KEY(id), INDEX
ix_cust GLOBAL ON (customer, created)) WITH (READ_REPLICAS_SETTINGS = "per_az:1", AUTO_PARTITIONING_BY_LOAD =
ENABLED, AUTO_PARTITIONING_MIN_PARTITIONS_COUNT = <min-partitions>, UNIFORM_PARTITIONS = <min-partitions>, AU
TO_PARTITIONING_MAX_PARTITIONS_COUNT = 1000);
CREATE TABLE `orderLines`(id_order Uint64, product Utf8, quantity Int64, PRIMARY KEY(id_order, product)) WITH
(AUTO_PARTITIONING_BY_LOAD = ENABLED, AUTO_PARTITIONING_MIN_PARTITIONS_COUNT = <min-partitions>, UNIFORM_PART
ITIONS = <min-partitions>, AUTO_PARTITIONING_MAX_PARTITIONS_COUNT = 1000);
```

Examples of load initialization

Creating a database with 1000 products, 10000 items of each product, and no orders:

```
ydb workload stock init -p 1000 -q 10000 -o 0
```

Creating a database with 10 products, 100 items of each product, 10 orders, and a minimum number of shards equal 100:

```
ydb workload stock init -p 10 -q 100 -o 10 --min-partitions 100
```

Running a load test

To run the load, execute the command:

```
ydb workload stock run [workload type...] [global workload options...] [specific workload options...]
```

During this test, workload statistics for each time window are displayed on the screen.

- `workload type`: The types of workload.
- `global workload options`: Global options for all types of load.
- `specific workload options`: Options of a specific load type.

See the description of the command to run the data load:

```
ydb workload run --help
```

The user-hist workload

This type of load reads the specified number of orders for the customer with id = 10000.

YQL query:

```
DECLARE $cust AS Utf8;
DECLARE $limit AS UInt32;

SELECT id, customer, created FROM orders view ix_cust
    WHERE customer = 'Name10000'
    ORDER BY customer DESC, created DESC
    LIMIT $limit;
```

To run this type of load, execute the command:

```
ydb workload stock run user-hist [global workload options...] [specific workload options...]
```

- `global workload options`: Global options for all types of load.
- `specific workload options`: Options of a specific load type.

Parameters for user-hist

Parameter name	Short name	Parameter description
`--limit <value>`	`-l <value>`	The required number of orders. Default: 10.

The rand-user-hist workload

This type of load reads the specified number of orders from randomly selected customers.

YQL query:

```
DECLARE $cust AS Utf8;
DECLARE $limit AS UInt32;

SELECT id, customer, created FROM orders view ix_cust
    WHERE customer = $cust
    ORDER BY customer DESC, created DESC
    LIMIT $limit;
```

To run this type of load, execute the command:

```
ydb workload stock run rand-user-hist [global workload options...] [specific workload options...]
```

- `global workload options`: Global options for all types of load.
- `specific workload options`: Options of a specific load type.

Parameters for rand-user-hist

Parameter name	Short name	Parameter description
`--limit <value>`	`-l <value>`	The required number of orders. Default: 10.

The add-rand-order workload

This type of load creates a randomly generated order. The order includes several different products, 1 item per product. The number of products in the order is generated randomly based on an exponential distribution.

YQL query:

```
DECLARE $ido AS UInt64;
DECLARE $cust AS Utf8;
DECLARE $lines AS List<Struct<product:Utf8,quantity:Int64>>;
DECLARE $time AS DateTime;

INSERT INTO `orders`(id, customer, created) VALUES
    ($ido, $cust, $time);
UPSERT INTO `orderLines`(id_order, product, quantity)
    SELECT $ido, product, quantity FROM AS_TABLE( $lines );
```

To run this type of load, execute the command:

```
ydb workload stock run add-rand-order [global workload options...] [specific workload options...]
```

- `global workload options`: Global options for all types of load.
- `specific workload options`: Options of a specific load type.

Parameters for add-rand-order

Parameter name	Short name	Parameter description
`--products <value>`	`-p <value>`	Number of products in the test. Default: 100.

The put-rand-order workload

This type of load creates a randomly generated order and processes it. The order includes several different products, 1 item per product. The number of products in the order is generated randomly based on an exponential distribution. Order processing consists in decreasing the number of ordered products in stock.

YQL query:

```
DECLARE $ido AS UInt64;
DECLARE $cust AS Utf8;
DECLARE $lines AS List<Struct<product:Utf8,quantity:Int64>>;
DECLARE $time AS DateTime;

INSERT INTO `orders`(id, customer, created) VALUES
    ($ido, $cust, $time);

UPSERT INTO `orderLines`(id_order, product, quantity)
    SELECT $ido, product, quantity FROM AS_TABLE( $lines );

$prods = SELECT * FROM orderLines AS p WHERE p.id_order = $ido;

$cnt = SELECT COUNT(*) FROM $prods;

$newq =
    SELECT
        p.product AS product,
        COALESCE(s.quantity, 0) - p.quantity AS quantity
    FROM $prods AS p
    LEFT JOIN stock AS s
    ON s.product = p.product;

$check = SELECT COUNT(*) AS cntd FROM $newq as q WHERE q.quantity >= 0;

UPSERT INTO stock
    SELECT product, quantity FROM $newq WHERE $check=$cnt;

$upo = SELECT id, $time AS tm FROM orders WHERE id = $ido AND $check = $cnt;

UPSERT INTO orders SELECT id, tm AS processed FROM $upo;

SELECT * FROM $newq AS q WHERE q.quantity < 0
```

To run this type of load, execute the command:

```
ydb workload stock run put-rand-order [global workload options...] [specific workload options...]
```

- `global workload options` : Global options for all types of load.
- `specific workload options` : Options of a specific load type.

Parameters for put-rand-order

Parameter name	Short name	Parameter description
`--products <value>`	`-p <value>`	Number of products in the test. Default: 100.

The put-same-order workload

This type of load creates an order with the same set of products and processes it. Order processing consists in decreasing the number of ordered products in stock.

YQL query:

```
DECLARE $ido AS UInt64;
DECLARE $cust AS Utf8;
DECLARE $lines AS List<Struct<product:Utf8,quantity:Int64>>;
DECLARE $time AS DateTime;

INSERT INTO `orders`(id, customer, created) VALUES
    ($ido, $cust, $time);

UPSERT INTO `orderLines`(id_order, product, quantity)
    SELECT $ido, product, quantity FROM AS_TABLE( $lines );

$prods = SELECT * FROM orderLines AS p WHERE p.id_order = $ido;

$cnt = SELECT COUNT(*) FROM $prods;

$newq =
    SELECT
        p.product AS product,
        COALESCE(s.quantity, 0) - p.quantity AS quantity
    FROM $prods AS p
```

```
    LEFT JOIN stock AS s
    ON s.product = p.product;

$check = SELECT COUNT(*) AS cntd FROM $newq as q WHERE q.quantity >= 0;

UPSERT INTO stock
    SELECT product, quantity FROM $newq WHERE $check=$cnt;

$upo = SELECT id, $time AS tm FROM orders WHERE id = $ido AND $check = $cnt;

UPSERT INTO orders SELECT id, tm AS processed FROM $upo;

SELECT * FROM $newq AS q WHERE q.quantity < 0
```

To run this type of load, execute the command:

```
ydb workload stock run put-same-order [global workload options...] [specific workload options...]
```

- `global workload options` : Global options for all types of load.
- `specific workload options` : Options of a specific load type.

Parameters for put-same-order

Parameter name	Short name	Parameter description
`--products <value>`	`-p <value>`	Number of products per order. Default: 100.

Examples of running the loads

- Run the `add-rand-order` workload for 5 seconds across 10 threads with 1000 products.

```
ydb workload stock run add-rand-order -s 5 -t 10 -p 1000
```

Possible result:

```
Elapsed Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
1           132 0       0       69      108     132     157
2           157 0       0       63      88      97      104
3           156 0       0       62      84      104     120
4           160 0       0       62      77      90      94
5           174 0       0       61      77      97      100

Txs     Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
779       155.8 0       0       62      89      108     157
```

- Run the `put-same-order` workload for 5 seconds across 5 threads with 2 products per order, printing out only final results.

```
ydb workload stock run put-same-order -s 5 -t 5 -p 1000 --quiet
```

Possible result:

```
Txs     Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
16          3.2 67      3       855     1407    1799    1799
```

- Run the `rand-user-hist` workload for 5 seconds across 100 threads, printing out time for each time window.

```
ydb workload stock run rand-user-hist -s 5 -t 10 --print-timestamp
```

Possible result:

```
Elapsed Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)        Timestamp
1          1046 0       0       7       16      25      50      2022-02-08T17:47:26Z
2          1070 0       0       7       17      22      28      2022-02-08T17:47:27Z
3          1041 0       0       7       17      22      28      2022-02-08T17:47:28Z
4          1045 0       0       7       17      23      31      2022-02-08T17:47:29Z
5           998 0       0       8       18      23      42      2022-02-08T17:47:30Z

Txs     Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
5200       1040 0       0       8       17      23      50
```

Interpretation of results

- `Elapsed` : Time window ID. By default, a time window is 1 second.
- `Txs/sec` : Number of successful load transactions in the time window.
- `Retries` : The number of repeat attempts to execute the transaction by the client in the time window.
- `Errors` : The number of errors that occurred in the time window.
- `p50(ms)` : 50th percentile of request latency, in ms.
- `p95(ms)` : 95th percentile of request latency, in ms.
- `p99(ms)` : 99th percentile of request latency, in ms.
- `pMax(ms)` : 100th percentile of request latency, in ms.

- `Timestamp`: Timestamp of the end of the time window.

ClickBench load

The load is based on data and queries from the https://github.com/ClickHouse/ClickBench repository, and the queries and table layout are adapted to YDB.

The benchmark generates typical workload in the following areas: clickstream and traffic analysis, web analytics, machine-generated data, structured logs, and event data. It covers typical queries in analytics and real-time dashboards.

The dataset for this benchmark was obtained from an actual traffic recording of one of the world's largest web analytics platforms. It has been anonymized while keeping all the essential data distributions. The query set was improvised to reflect realistic workloads, while the queries are not directly from production.

Common command options

All commands support the common option `--path`, which specifies the path to a table in the database:

```
ydb workload clickbench --path clickbench/hits ...
```

Available options

Name	Description	Default value
`--path` or `-p`	Specifies the table path.	`clickbench/hits`

Initializing a load test

Before running the benchmark, create a table:

```
ydb workload clickbench --path clickbench/hits init
```

See the description of the command to init the data load:

```
ydb workload clickbench init --help
```

Available parameters

Name	Description	Default value
`--store <value>`	Table storage type. Possible values: `row`, `column`, `external-s3`.	`column`.
`--external-s3-prefix <value>`	Only relevant for external tables. Root path to the dataset in S3 storage.	
`--external-s3-endpoint <value>` or `-e <value>`	Only relevant for external tables. Link to S3 Bucket with data.	
`--string`	Use `String` type for text fields. `Utf8` is used by default.	
`--datetime`	Use `Date`, `Datetime` and `Timestamp` type for time-related fields.	`Date32`, `Datetime64` and `Timestamp64`
`--partition-size`	Maximum partition size in megabytes (AUTO_PARTITIONING_PARTITION_SIZE_MB) for row tables.	2000
`--clear`	If the table at the specified path has already been created, it will be deleted.	
`--dry-run`	Do not execute initialization queries, but only display their text.	

Loading data into a table

Download the data archive, then load the data into the table:

```
wget https://datasets.clickhouse.com/hits_compatible/hits.csv.gz
ydb workload clickbench --path clickbench/hits import files --input hits.csv.gz
```

For source files, you can use CSV and TSV files, as well as directories containing such files. They can be either compressed or not.

Available parameters

Name	Description	Default value
`--input <path>` or `-i <path>`	Path to the source data files. Both unpacked and packed CSV and TSV files, as well as directories containing such files, are supported. Data can be downloaded from the official ClickBench website: csv.gz, tsv.gz. To speed up the process, these files can be split into smaller parts, allowing parallel downloads.	
`--state <path>`	Path to the download state file. If the download is interrupted, it will resume from the same point when restarted.	
`--clear-state`	Relevant if the `--state` parameter is specified. Clears the state file and restarts the download from the beginning.	

`--dry-run`	Do not execute loading queries, but only display their text.	

Common parameters of the import command

Name	Description	Default value
`--upload-threads <value>` or `-t <value>`	The number of execution threads for data preparation.	The number of available cores on the client.
`--bulk-size <value>`	The size of the chunk for sending data, in rows.	10000
`--max-in-flight <value>`	The maximum number of data chunks that can be processed simultaneously.	128
`--file-output-path <value>` or `-f <path>`	If this option is set, the data will not be loaded into the database, but will be saved to the directory .	

Run a load test

Run the load:

```
ydb workload clickbench --path clickbench/hits run
```

During the test, load statistics are displayed for each request.

See the command description to run the load:

```
ydb workload clickbench run --help
```

Common parameters for all load types

Name	Description	Default value
`--dry-run`	Do not execute initialization queries, but only display their text.	
`--check-canonical` or `-c`	Use special version of queries (they have deterministic answers) and compare results with canonical ones.	
`--output <value>`	The name of the file where the query execution results will be saved.	`results.out`
`--iterations <value>`	The number of times each load query will be executed.	`1`
`--json <name>`	The name of the file where query execution statistics will be saved in `json` format.	Not saved by default
`--ministat <name>`	The name of the file where query execution statistics will be saved in `ministat` format.	Not saved by default
`--csv <name>`	The name of the file to save the CSV version of the result table.	Not saved by default
`--plan <name>`	The name of the file to save the query plan. Files like `<name>.<query number>.explain` and `<name>.<query number>.<iteration number>` will be saved in formats: `ast`, `json`, `svg`, and `table`.	Not saved by default
`--query-prefix <setting>`	Query prefix. Every prefix is a line that will be added to the beginning of each query. For multiple prefix lines use this option several times.	Not specified by default
`--retries`	Max retry count for every request.	`0`
`--include`	Names, numbers or ranges of query numbers to be executed as part of the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	All queries executed
`--exclude`	Names, numbers or ranges of query numbers to be excluded from the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	None excluded by default
`--verbose` or `-v`	Print additional information to the screen during query execution.	
`--global-timeout <value>`	Global timeout for all queries. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--request-timeout <value>`	Timeout for each iteration of each query. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--threads <value>` or `-t <value>`	The number of parallel threads generating the load. Zero means that queries will be executed in the main thread; otherwise, queries will be mixed.	`0`
`--stats <value>`	Extended execution statistics collection mode. Available values: `full`, `profile`.	`full`

ClickBench-specific options

Name	Description	Default value

`--syntax <value>`	Syntax of the queries to use. Available values: `yql`. For more information about working with YQL syntax, see here.	`yql`

Cleanup test data

Run cleanup:

```
ydb workload clickbench --path clickbench/hits clean
```

The command has no parameters.

Key-Value load

A simple load type using a YDB database as a Key-Value storage.

Types of load

This load test runs several types of load:

- upsert: Using the UPSERT operation, inserts rows that are tuples (key1, key2, ... keyK, value1, value2, ... valueN) into the table created previously with the init command, the K and N numbers are specified in the settings.
- insert: The function is the same as the upsert load, only the INSERT operation is used for insertion.
- select: Reads data using the SELECT * WHERE key = $key operation. A query always affects all table columns, but isn't always a point query, and the number of primary key variations can be controlled using parameters.
- read-rows: Reads data using the ReadRows operation, which performs faster key reading than select operation. A query always affects all table columns, but isn't always a point query, and the number of primary key variations can be controlled using parameters.
- mixed: Simultaneously writes and reads data, additionally checking that all written data is successfully read.

Load test initialization

To get started, you must create tables. When creating them, you can specify how many rows to insert during initialization:

```
ydb workload kv init [init options...]
```

- `init options`: Initialization options.

View a description of the command to initialize the table:

```
ydb workload kv init --help
```

Available parameters

Parameter name	Parameter description
`--init-upserts <value>`	Number of insertion operations to be performed during initialization. Default: 1000.
`--min-partitions`	Minimum number of shards for tables. Default: 40.
`--partition-size`	Maximum size of one shard (the `AUTO_PARTITIONING_PARTITION_SIZE_MB` setting). Default: 2000.
`--auto-partition`	Enabling/disabling auto-sharding. Possible values: 0 or 1. Default: 1.
`--max-first-key`	Maximum value of the primary key of the table. Default: .
`--len`	The size of the rows in bytes that are inserted into the table as values. Default: 8.
`--cols`	Number of columns in the table. Default: 2 counting Key.
`--int-cols`	Number of first columns in the table that will have the `Uint64` type; subsequent columns will have the `String` type. Default: 1.
`--key-cols`	Number of first columns in the table included in the key. Default: 1.
`--rows`	Number of affected rows in one query. Default: 1.

The following command is used to create a table:

```
CREATE TABLE `kv_test`(
    c0 Uint64,
    c1 Uint64,
    ...
    cI Uint64,
    cI+1 String,
    ...
    cN String,
    PRIMARY KEY(c0, c1, ... cK)) WITH (
        AUTO_PARTITIONING_BY_LOAD = ENABLED,
        AUTO_PARTITIONING_MIN_PARTITIONS_COUNT = partsNum,
        UNIFORM_PARTITIONS = partsNum,
        AUTO_PARTITIONING_PARTITION_SIZE_MB = partSize,
        AUTO_PARTITIONING_MAX_PARTITIONS_COUNT = 1000
    )
)
```

Examples of load initialization

Example of a command to create a table with 1000 rows:

```
ydb workload kv init --init-upserts 1000
```

Deleting a table

When the work is complete, you can delete the table:

```
ydb workload kv clean
```

The following YQL command is executed:

```
DROP TABLE `kv_test`
```

Examples of using clean

```
ydb workload kv clean
```

Running a load test

To run the load, execute the command:

```
ydb workload kv run [workload type...] [global workload options...] [specific workload options...]
```

During this test, workload statistics for each time window are displayed on the screen.

- `workload type`: The types of workload.
- `global workload options`: The global options for all types of load.
- `specific workload options`: Options of a specific load type.

See the description of the command to run the data load:

```
ydb workload kv run --help
```

Global parameters for all types of load

Parameter name	Short name	Parameter description
`--seconds <value>`	`-s <value>`	Duration of the test, in seconds. Default: 10.
`--threads <value>`	`-t <value>`	The number of parallel threads creating the load. Default: 10.
`--rate <value>`	-	Total rate for all threads, in requests per second. Default: 0 (no rate limit).
`--quiet`	-	Outputs only the total result.
`--print-timestamp`	-	Print the time together with the statistics of each time window.
`--client-timeout`	-	Transport timeout in milliseconds.
`--operation-timeout`	-	Operation timeout in milliseconds.
`--cancel-after`	-	Timeout for canceling an operation in milliseconds.
`--window`	-	Statistics collection window in seconds. Default: 1.
`--max-first-key`	-	Maximum value of the primary key of the table. Default: .
`--cols`	-	Number of columns in the table. Default: 2 counting Key.
`--int-cols`	-	Number of first columns in the table that will have the `Uint64` type; subsequent columns will have the `String` type. Default: 1.
`--key-cols`	-	Number of first columns in the table included in the key. Default: 1.
`--rows`	-	Number of affected rows in one query. Default: 1.

Upsert load

This load type inserts tuples (key, value1, value2, ..., valueN)

To run this type of load, execute the command:

```
ydb workload kv run upsert [global workload options...] [specific workload options...]
```

- `global workload options`: The global options for all types of load.
- `specific workload options`: Options of a specific load type.

For example, for the parameters `--rows 2 --cols 3 --int-cols 2`, the YQL query will look like this:

```
DECLARE $c0_0 AS Uint64;
DECLARE $c0_1 AS Uint64;
DECLARE $c0_2 AS String;
DECLARE $c1_0 AS Uint64;
DECLARE $c1_1 AS Uint64;
DECLARE $c1_2 AS String;
UPSERT INTO `kv_test` (c0, c1, c2) VALUES ($c0_0, $c0_1, $c0_2), ($c1_0, $c1_1, $c1_2)
```

Parameters for upsert

Parameter name	Parameter description
`--len`	The size of the rows in bytes that are inserted into the table as values. Default: 8.

Insert load

This load type inserts tuples (key, value1, value2, ..., valueN)

To run this type of load, execute the command:

```
ydb workload kv run insert [global workload options...] [specific workload options...]
```

- `global workload options` : The global options for all types of load.
- `specific workload options` : Options of a specific load type.

For example, for the parameters `--rows 2 --cols 3 --int-cols 2`, the YQL query will look like this:

```
DECLARE $c0_0 AS Uint64;
DECLARE $c0_1 AS Uint64;
DECLARE $c0_2 AS String;
DECLARE $c1_0 AS Uint64;
DECLARE $c1_1 AS Uint64;
DECLARE $c1_2 AS String;
INSERT INTO `kv_test` (c0, c1, c2) VALUES ($c0_0, $c0_1, $c0_2), ($c1_0, $c1_1, $c1_2)
```

Parameters for insert

Parameter name	Parameter description
`--len`	The size of the rows in bytes that are inserted into the table as values. Default: 8.

Select load

This type of load creates SELECT queries that return rows based on an exact match of the primary key.

To run this type of load, execute the command:

```
ydb workload kv run select [global workload options...]
```

- `global workload options` : The global options for all types of load.

For example, for the parameters `--rows 2 --cols 3 --int-cols 2`, the YQL query will look like this:

```
DECLARE $r0_0 AS Uint64;
DECLARE $r0_1 AS Uint64;
DECLARE $r1_0 AS Uint64;
DECLARE $r1_1 AS Uint64;
SELECT c0, c1, c2 FROM `kv_test` WHERE c0 = $r0_0 AND c1 = $r0_1 OR c0 = $r1_0 AND c1 = $r1_1
```

Read-rows load

This type of load creates ReadRows queries that return rows based on an exact match of the primary key.

To run this type of load, execute the command:

```
ydb workload kv run read-rows [global workload options...]
```

- `global workload options` : The global options for all types of load.

Mixed load

This type of load simultaneously writes and reads tuples (key, value1, value2, ..., valueN), additionally checking that all written data is successfully read.

To run this type of load, execute the command:

```
ydb workload kv run mixed [global workload options...] [specific workload options...]
```

- `global workload options` : The global options for all types of load.
- `specific workload options` - Options of a specific load type.

Parameters for mixed

Parameter name	Parameter description
`--len`	The size of the rows in bytes that are inserted into the table as values. Default: 8.
`--change-partitions-size`	Enabling/disabling random modification of the `AUTO_PARTITIONING_PARTITION_SIZE_MB` setting. Possible values: 0 or 1. Default: 0.
`--do-select`	Enabling/disabling reads using the select query. Possible values: 0 or 1. Default: 1.
`--do-read-rows`	Enabling/disabling reads using the read-rows query. Possible values: 0 or 1. Default: 1.

Topic load

Applies load to your YDB topics, using them as message queues. You can use a variety of input parameters to simulate production load: message number, message size, target write rate, and number of consumers and producers.

As you apply load to your topic, the console displays the results (the number of written messages, message write rate, and others).

To generate load against your topic:

1. Initialize the load.
2. Run one of the available load types:
   - write: Generate messages and write them to the topic asynchronously.
   - read: Read messages from the topic asynchronously.
   - full: Read and write messages asynchronously in parallel.

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Initializing a load test

Before executing the load, you need to initialize it. During initialization, you will create a topic named `workload-topic` with the specified options. To initialize the load, run the following command:

```
ydb [global options...] workload topic init [options...]
```

- `global options`: Global options.
- `options`: Subcommand options.

Subcommand options:

Option name	Option description
`--topic`	Topic name. Default value: `workload-topic`.
`--partitions`, `-p`	Number of topic partitions. Default value: `128`.
`--consumers`, `-c`	Number of topic consumers. Default value: `1`.
`--consumer-prefix`	Consumer name prefix. Default value: `workload-consumer`. For example, if the number of consumers `--consumers` is `2` and the prefix `--consumer-prefix` is `workload-consumer`, then the following consumer names will be used: `workload-consumer-0`, `workload-consumer-1`.

To create a topic with `256` partitions and `2` consumers, run this command:

```
ydb --profile quickstart workload topic init --partitions 256 --consumers 2
```

Write load

This load type generates and writes messages to the topic asynchronously.

General format of the command that generates the write load:

```
ydb [global options...] workload topic run write [options...]
```

- `global options`: Global options.
- `options`: Subcommand options.

View the description of the command that generates the write load:

```
ydb workload topic run write --help
```

Subcommand options:

Option name	Option description
`--seconds`, `-s`	Test duration in seconds. Default value: `60`.
`--window`, `-w`	Statistics window in seconds. Default value: `1`.
`--quiet`, `-q`	Output only the final test result.
`--print-timestamp`	Print the time together with the statistics of each time window.

`--warmup`	Test warm-up period (in seconds). Within the period, no statistics are calculated. It's needed to eliminate the effect of transition processes at startup. Default value: `5`.
`--percentile`	Percentile that is output in statistics. Default value: `50`.
`--topic`	Topic name. Default value: `workload-topic`.
`--threads`, `-t`	Number of producer threads. Each thread will write to all partitions of the specified topic. Default value: `1`.
`--message-size`, `-m`	Message size in bytes. Use the `K`, `M`, or `G` suffix to set the size in KB, MB, or GB, respectively. Default value: `10K`.
`--message-rate`	Total target write rate in messages per second. Can't be used together with the `--byte-rate` option. Default value: `0` (no limit).
`--byte-rate`	Total target write rate in bytes per second. Can't be used together with the `--message-rate` option. Use the `K`, `M`, or `G` suffix to set the rate in KB/s, MB/s, or GB/s, respectively. Default value: `0` (no limit).
`--codec`	Codec used to compress messages on the client before sending them to the server. Compression increases CPU usage on the client when reading and writing messages, but usually enables you to reduce the amounts of data stored and transmitted over the network. When consumers read messages, they decompress them by the codec that was used to write the messages, with no special options needed. Acceptable values: `RAW` - no compression (default), `GZIP`, `ZSTD`.
`--use-tx`	Use transactions. Disabled by default.
`--tx-commit-interval`	Transaction commit interval, in milliseconds. A transaction is committed if the time specified in the `--tx-commit-interval` parameter elapses or if the number of messages specified in the `--tx-commit-messages` parameter is written. Default value: `1000`.
`--tx-commit-messages`	Number of messages required to commit a transaction. A transaction is committed if the time specified in the `--tx-commit-interval` parameter elapses or if the number of messages specified in the `--tx-commit-messages` parameter is written. Default value: `1 000 000`.

To write data to `100` producer threads at the target rate of `80` MB/s for `10` seconds, run this command:

```
ydb --profile quickstart workload topic run write --threads 100 --byte-rate 80M
```

You will see statistics for in-progress time windows and final statistics when the test is complete:

```
Window  Write speed     Write time      Inflight
#       msg/s   MB/s    percentile,ms   percentile,msg
1       20      0       1079            72
2       8025    78      1415            78
3       7987    78      1431            79
4       7888    77      1471            101
5       8126    79      1815            116
6       7018    68      1447            79
7       8938    87      2511            159
8       7055    68      1463            78
9       7062    69      1455            79
10      9912    96      3679            250
Window  Write speed     Write time      Inflight
#       msg/s   MB/s    percentile,ms   percentile,msg
Total   7203    70      3023            250
```

- `Window`: Sequence number of the statistics window.
- `Write speed`: Message write rate in messages per second and MB/s.
- `Write time`: Percentile of the message write time, in milliseconds.
- `Inflight`: Maximum number of messages awaiting commit across all partitions.

If you enable transactions, the command output will also include the `Commit time` column. It displays the percentile of transaction commit time, in milliseconds.

Read load

This type of load reads messages from the topic asynchronously. To make sure that the topic includes messages, run the write load before you start reading.

General format of the command to generate the read load:

```
ydb [global options...] workload topic run read [options...]
```

- `global options`: Global options.
- `options`: Subcommand options.

View the description of the command to generate the read load:

```
ydb workload topic run read --help
```

Subcommand options:

Option name	Option description
`--seconds`, `-s`	Test duration in seconds. Default value: `60`.
`--window`, `-w`	Statistics window in seconds. Default value: `1`.
`--quiet`, `-q`	Output only the final test result.
`--print-timestamp`	Print the time together with the statistics of each time window.
`--warmup`	Test warm-up period (in seconds). Within the period, no statistics are calculated. It's needed to eliminate the effect of transition processes at startup. Default value: `5`.
`--percentile`	Percentile that is output in statistics. Default value: `50`.
`--topic`	Topic name. Default value: `workload-topic`.
`--consumers`, `-c`	Number of consumers. Default value: `1`.
`--consumer-prefix`	Consumer name prefix. Default value: `workload-consumer`. For example, if the number of consumers `--consumers` is `2` and the prefix `--consumer-prefix` is `workload-consumer`, then the following consumer names will be used: `workload-consumer-0`, `workload-consumer-1`.
`--threads`, `-t`	Number of consumer threads. Default value: `1`.

To use `2` consumers to read data from the topic, with `100` threads per consumer, run the following command:

```
ydb --profile quickstart workload topic run read --consumers 2 --threads 100
```

You will see statistics for in-progress time windows and final statistics when the test is complete:

```
Window  Lag             Lag time        Read speed      Full time
#       percentile,msg  percentile,ms   msg/s   MB/s    percentile,ms
1       0               0               0       0       0
2       30176           0               66578   650     0
3       30176           0               68999   674     0
4       30176           0               66907   653     0
5       27835           0               67628   661     0
6       30176           0               67938   664     0
7       30176           0               71628   700     0
8       20338           0               61367   599     0
9       30176           0               61770   603     0
10      30176           0               58291   569     0
Window  Lag             Lag time        Read speed      Full time
#       percentile,msg  percentile,ms   msg/s   MB/s    percentile,ms
Total   30176           0               80267   784     0
```

- `Window`: Sequence number of the statistics window.
- `Lag`: Maximum consumer lag in the statistics window. Messages across all partitions are included.
- `Lag time`: Percentile of the message lag time in milliseconds.
- `Read`: Message read rate for the consumer (in messages per second and MB/s).
- `Full time`: Percentile of the full message processing time (from writing by the producer to reading by the consumer), in milliseconds.

Read and write load

This type of load both reads messages from the topic and writes them to the topic asynchronously. This command is equivalent to running both read and write loads in parallel.

General format of the command to generate the read and write load:

```
ydb [global options...] workload topic run full [options...]
```

- `global options`: Global options.
- `options`: Subcommand options.

View the description of the command to run the read and write load:

```
ydb workload topic run full --help
```

Subcommand options:

Option name	Option description
`--seconds`, `-s`	Test duration in seconds. Default value: `60`.
`--window`, `-w`	Statistics window in seconds. Default value: `1`.
`--quiet`, `-q`	Output only the final test result.
`--print-timestamp`	Print the time together with the statistics of each time window.
`--warmup`	Test warm-up period (in seconds). Within the period, no statistics are calculated. It's needed to eliminate the effect of transition processes at startup. Default value: `5`.
`--percentile`	Percentile that is output in statistics. Default value: `50`.
`--topic`	Topic name. Default value: `workload-topic`.
`--producer-threads`, `-p`	Number of producer threads. Each thread will write to all partitions of the specified topic. Default value: `1`.
`--message-size`, `-m`	Message size in bytes. Use the `K`, `M`, or `G` suffix to set the size in KB, MB, or GB, respectively. Default value: `10K`.
`--message-rate`	Total target write rate in messages per second. Can't be used together with the `--message-rate` option. Default value: `0` (no limit).
`--byte-rate`	Total target write rate in bytes per second. Can't be used together with the `--byte-rate` option. Use the `K`, `M`, or `G` suffix to set the rate in KB/s, MB/s, or GB/s, respectively. Default value: `0` (no limit).
`--codec`	Codec used to compress messages on the client before sending them to the server. Compression increases CPU usage on the client when reading and writing messages, but usually enables you to reduce the amounts of data stored and transmitted over the network. When consumers read messages, they decompress them by the codec that was used to write the messages, with no special options needed. Acceptable values: `RAW` - no compression (default), `GZIP`, `ZSTD`.
`--consumers`, `-c`	Number of consumers. Default value: `1`.
`--consumer-prefix`	Consumer name prefix. Default value: `workload-consumer`. For example, if the number of consumers `--consumers` is `2` and the prefix `--consumer-prefix` is `workload-consumer`, then the following consumer names will be used: `workload-consumer-0`, `workload-consumer-1`.
`--threads`, `-t`	Number of consumer threads. Default value: `1`.
`--use-tx`	Use transactions. Disabled by default.
`--tx-commit-interval`	Transaction commit interval, in milliseconds. A transaction is committed if the time specified in the `--tx-commit-interval` parameter elapses or if the number of messages specified in the `--tx-commit-messages` parameter is written. Default value: `1000`.
`--tx-commit-messages`	Number of messages required to commit a transaction. A transaction is committed if the time specified in the `--tx-commit-interval` parameter elapses or if the number of messages specified in the `--tx-commit-messages` parameter is written. Default value: `1 000 000`.

Example of a command that reads `50` threads by `2` consumers and writes data to `100` producer threads at the target rate of `80` MB/s and duration of `10` seconds:

```
ydb --profile quickstart workload topic run full --producer-threads 100 --consumers 2 --consumer-threads 50 --byte-rate 80M
```

You will see statistics for in-progress time windows and final statistics when the test is complete:

```
Window  Write speed     Write time      Inflight        Lag             Lag time        Read speed      Full time
#       msg/s   MB/s    percentile,ms   percentile,msg  percentile,msg  percentile,ms   msg/s   MB/s    percentile,ms
1       0       0       0               0               0               0               0       0       0
2       1091    10      2143            8               2076            20607           40156   392     30941
3       1552    15      2991            12              7224            21887           41040   401     31886
4       1733    16      3711            15              10036           22783           38488   376     32577
5       1900    18      4319            15              10668           23551           34784   340     33372
6       2793    27      5247            21              9461            24575           33267   325     34893
7       2904    28      6015            22              12150           25727           34423   336     35507
8       2191    21      5087            21              12150           26623           29393   287     36407
9       1952    19      2543            10              7627            27391           33284   325     37814
```

```
10      1992    19      2655            9               10104           28671           29101   284     38797
Window  Write speed     Write time      Inflight        Lag             Lag time        Read speed      Full
time
#       msg/s   MB/s    percentile,ms   percentile,msg  percentile,msg  percentile,ms   msg/s   MB/s    perce
ntile,ms
Total   1814    17      5247            22              12150           28671           44827   438     40252
```

- `Window` : Sequence number of the statistics window.
- `Write speed` : Message write rate in messages per second and MB/s.
- `Write time` : Percentile of the message write time, in milliseconds.
- `Inflight` : Maximum number of messages awaiting commit across all partitions.
- `Lag` : Maximum number of messages awaiting reading, in the statistics window. Messages across all partitions are included.
- `Lag time` : Percentile of the message lag time in milliseconds.
- `Read` : Message read rate for the consumer (in messages per second and MB/s).
- `Full time` : Percentile of the full message processing time, from writing by the producer to reading by the consumer, in milliseconds.

Deleting a topic

When the work is complete, you can delete the test topic: General format of the topic deletion command:

```
ydb [global options...] workload topic clean [options...]
```

- `global options` : Global options.
- `options` : Subcommand options.

Subcommand options:

Option name	Option description
`--topic`	Topic name. Default value: `workload-topic`.

To delete the `workload-topic` test topic, run the following command:

```
ydb --profile quickstart workload topic clean
```

Transfer load

Starts the load in the form of transactions YDB involving topics and tables simultaneously. The data is read from the topic and written to the table. To simulate a real load, you can set various input parameters: the number of messages, the size of messages, the target write speed, the number of consumers and producers, the number of partitions. During operation, the console displays the results: the number of written messages, the speed of writing messages, etc.

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Initializing the test environment

Before starting the load, it is necessary to initialize the test environment. You can use the command `ydb workload transfer topic-to-table init` to do this. It will create a topic and a table with the necessary parameters.

Command syntax:

```
ydb [global options...] workload transfer topic-to-table init [options...]
```

- `global options` — global parameters.
- `options` - parameters of the subcommand.

View the command description:

```
ydb workload transfer topic-to-table init --help
```

Parameters of the subcommand:

Parameter name	Parameter description	Default value
`--topic`	Topic name	`transfer-topic`
`--consumer-prefix`	Prefix of the consumers name	`workload-consumer`
`--table`	Table name	`transfer-table`
`--consumers`	Number of topic consumers	`1`
`--topic-partitions`	Number of topic partitions	`128`
`--table-partitions`	Number of table partitions	`128`

After executing the `init` subcommand, a table, topic and consumers will be created. Reader names are created by the rule `${CONSUMER_PREFIX}-${INDEX}`. The value of `${INDEX}` is an integer from 0 to the value of the parameter `--consumers` minus 1.

For example, the command `ydb --profile quickstart workload transfer topic-to-table init --consumers 2 --topic-partitions 143 --table-partitions 237` will create a topic `transfer-topic` with 2 consumers, 143 partitions, and a table `transfer-table` with 237 partitions. The consumer names are `workload-consumer-0` and `workload-consumer-1`.

Running a load test

The test simulates the load from an application that receives messages from a topic, processes them and writes the processing results to a database table.

During the operation of the program, two types of work streams are simulated:

- Input stream: messages are written to the topic in the non-transaction mode. The user can control the writing speed, the message size, the number of producers.
- Processing flow: messages are read from the topic and written to the table using the YDB transaction.

The following actions are performed in the processing flow within a single transaction:

- messages from the topic are being read until the `--commit-period` period has expired;
- one `UPSERT` command and a `COMMIT` command are executed on the table to commit the transaction after the period expires.

Command syntax:

```
ydb [global options...] workload transfer topic-to-table run [options...]
```

- `global options` — global parameters.
- `options` - parameters of the subcommand.

View the command description:

```
ydb workload transfer topic-to-table run --help
```

Parameters of the subcommand:

Parameter name	Parameter Description	Default value
`--seconds`, `-s`	Duration of the test in seconds	`60`
`--window`, `-w`	Duration of the statistics collection window in seconds	`1`

`--quiet`, `-q`	Output only the final test result	`0`
`--print-timestamp`	Print the time together with the statistics of each time window	`0`
`--percentile`	Percentile in statistics output	`50`
`--warmup`	The warm-up time of the test in seconds. No statistics are calculated during this time	`5`
`--topic`	Topic name	`transfer-topic`
`--consumer-prefix`	Prefix of the consumers name	`workload-consumer`
`--table`	Table name	`transfer-table`
`--producer-threads`, `-p`	Number of producer threads	`1`
`--consumer-threads`, `-t`	Number of consumer threads	`1`
`--consumers`, `-c`	Number of consumers	`1`
`--message-size`, `-m`	Message size in bytes. It is possible to specify in KB, MB, GB by adding suffixes `K`, `M`, `G` respectively	`10240`
`--message-rate`	Target total write speed. In messages per second. Excludes the use of the `--byte-rate` parameter	`0`
`--byte-rate`	Target total write speed. In bytes per second. Excludes the use of the `--message-rate` parameter. It is possible to specify in KB/s, MB/s, GB/s by adding suffixes `K`, `M`, `G` respectively	`0`
`--tx-commit-interval`	The period between transaction `COMMIT` calls. In milliseconds	`1000`
`--tx-commit-messages`	The period between transaction `COMMIT` calls. In number of messages	`1000000`
`--only-topic-in-tx`	Only topic partitions are forced to participate in transactions. Excludes the use of the `--only-table-in-tx` parameter	`0`
`--only-table-in-tx`	Only table shards are forced to participate in transactions. Excludes the use of the `--only-topic-in-tx` parameter	`0`

For example, the command `ydb --profile quickstart workload transfer topic-to-table run` will run a test lasting 60 seconds. The data for the first 5 seconds will not be taken into account in the work statistics. Example of console output:

```
Window  Write speed     Write time      Inflight        Read speed      Topic time      Select time     Upser
t time     Commit time
#       msg/s   MB/s    percentile,ms   percentile,msg  msg/s   MB/s    percentile,ms   percentile,ms   perce
ntile,ms   percentile,ms
1       0       0       0               0               0       0       0               0               0
0
2       0       0       0               0               0       0       0               0               0
0
3       0       0       0               0               0       0       0               0               0
0
4       0       0       0               0               0       0       0               0               0
0
5       0       0       0               0               0       0       0               0               0
0
6       103     1       1023            83              103     1       1025            0               0
0
7       103     1       999             78              103     1       1001            0               0
0
8       103     1       1003            93              103     1       1002            0               0
0
9       103     1       1003            88              103     1       1003            0               0
0
10      103     1       999             79              103     1       999             0               0
0
11      103     1       1119            89              0       0       0               0               0
0
12      103     1       1023            90              206     2       1028            90              223
695
13      103     1       975             84              103     1       976             0               0
0
14      103     1       1003            91              103     1       1006            0               0
0
15      103     1       1003            93              103     1       1005            0               0
0
16      103     1       1103            89              103     1       1100            0               0
0
17      103     1       1063            89              103     1       1061            0               0
0
...
```

- `Window` — the serial number of the time window for collecting statistics.
- `Write speed` — the speed of writing messages by producers. In messages per second and in megabytes per second.
- `Write time` — the specified percentile of the message writing time in ms.

- `Inflight` — the maximum number of messages waiting for confirmation for all batches.
- `Lag` — the specified percentile of maximum number of messages waiting to be read in the statistics collection window. Messages for all batches are taken into account.
- `Lag time` — the specified percentile of message delay time in ms.
- `Read speed` — the speed of reading messages by consumers. In messages per second and in megabytes per second.
- `Select time`, `Upsert time`, `Commit time` — the specified percentile of the execution time of Select, Insert, Commit operations in ms.

Removing the test environment

After the test is completed, you can delete the test environment.

Command syntax:

```
ydb [global options...] workload transfer topic-to-table clean [options...]
```

- `global options` — global parameters.
- `options` - parameters of the subcommand.

View the command description:

```
ydb workload transfer topic-to-table clean --help
```

Parameters of the subcommand:

Parameter Name	Parameter Description	Default value
`--topic`	Topic name	`transfer-topic`
`--table`	Table name	`transfer-table`

For example, the command `ydb --profile quickstart workload transfer topic-to-table clean` will delete the topic `transfer-topic`, its consumers and the table `transfer-table`.

TPC-C Workload

The workload is based on the TPC-C specification, with the queries and table schemas adapted for YDB.

TPC-C is an industry-standard On-Line Transaction Processing (OLTP) benchmark. It simulates a retail company with a configurable number of warehouses, each containing 10 districts and 3,000 customers per district. A corresponding inventory exists for the warehouses. Customers place orders composed of several items. The company tracks payments, deliveries, and order history, and periodically performs inventory checks.

As a result, the benchmark generates a workload of concurrent distributed transactions with varying types and complexities.

In TPC-C, the warehouse is the basic scale unit: the more warehouses you configure, the larger the dataset and the higher the transaction volume. Each warehouse holds roughly 100 MiB of data. The maximum number of warehouses a YDB cluster can sustain depends on hardware capacity, configuration, and inter-node network latency. As a rule of thumb, allocating about 50 warehouses per compute CPU core is a good starting point.

Here is a quick start snippet:

```
ydb workload tpcc --path tpcc/10wh init -w 10
ydb workload tpcc --path tpcc/10wh import -w 10
ydb workload tpcc --path tpcc/10wh run -w 10
```

By default, when executed in an interactive terminal, both the `import` and `run` commands display their execution progress using a Terminal User Interface (TUI).

Common Command Options

All commands support the common `--path` option, which specifies the path to the directory containing the benchmark tables in the database:

```
ydb workload tpcc --path tpcc/10wh ...
```

Available Options

Name	Description	Default value
`--path` or `-p`	Database path where the benchmark tables are located.	`/`

Initializing a Load Test

Before running the benchmark, create the tables:

```
ydb workload tpcc --path tpcc/10wh init --warehouses 10
```

See the command description:

```
ydb workload tpcc init --help
```

Available parameters

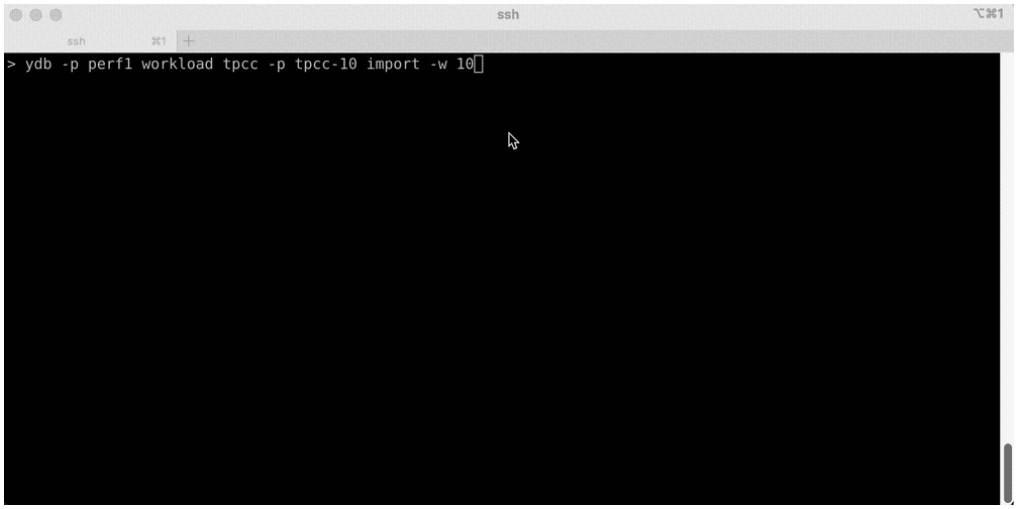
Name	Description	Default value
`--warehouses` or `-w`	A number of TPC-C warehouses.	10

Loading data into a table

The data will be generated and loaded into the tables directly by ydb:

```
ydb workload tpcc --path tpcc/10wh import --warehouses 10
```

Example usage:



See the command description:

```
ydb workload tpcc import --help
```

Available options

Name	Description	Default value
`--warehouses` or `-w`	A number of TPC-C warehouses.	10
`--compact`	Compact tables after importing data.	
`--threads`	A number of threads loading the TPC-C data to the database.	auto
`--no-tui`	Disable TUI, which is enabled by default in interactive mode.	

The optimal number of loading threads depends on your YDB cluster's size and configuration. As a rule of thumb, for clusters with several hundred CPU cores, starting with around 50 loading threads is reasonable. For larger clusters, you can scale this number further.

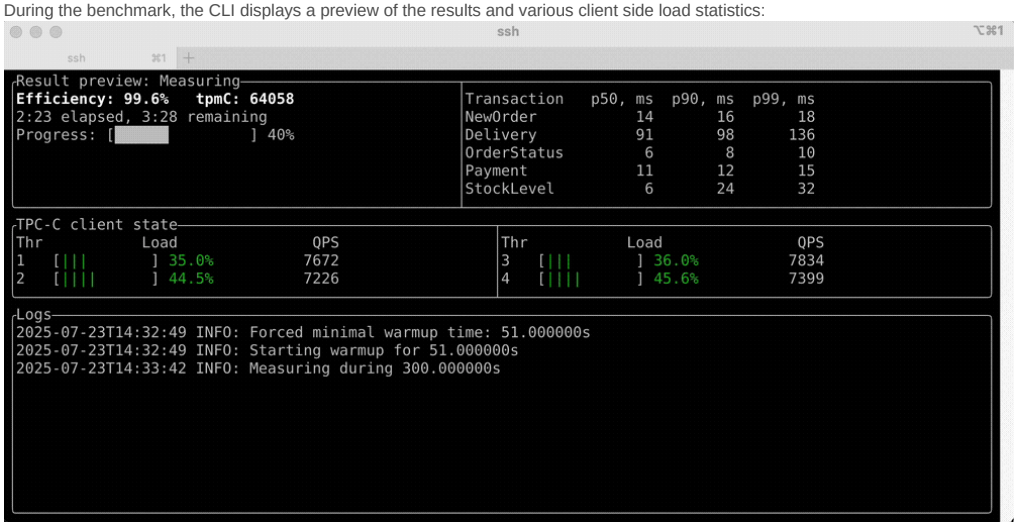
However, on the client side, it is recommended to keep the number of loading threads to no more than 50–75% of the available CPU cores.

Run the load test

Run the load:

```
ydb workload tpcc --path tpcc/10wh run --warehouses 10
```

During the benchmark, the CLI displays a preview of the results and various client side load statistics:



See the command description:

```
ydb workload tpcc run --help
```

Available options

Name	Description	Default value
`--warehouses` or `-w`	A number of TPC-C warehouses.	10
`--warmup`	Warmup time. Example: 10s, 5m, 1h.	auto
`--time` or `-t`	Execution time. Example: 10s, 5m, 1h.	2h
`--max-sessions` or `-m`	A soft limit on the number of DB sessions.	auto
`--threads`	A number of threads executing queries	auto
`--format` or `-f`	Output format: 'Pretty', 'Json'	Pretty
`--no-tui`	Disable TUI, which is enabled by default in interactive mode.	

The optimal number of sessions depends on your YDB cluster's size and configuration. As a rule of thumb, a good starting point is to multiply the total number of CPU cores allocated to YDB compute (dynnodes) by 5–10.

We recommend starting with a multiplier of 5. If YDB is underutilized during the benchmark run, consider increasing the number of sessions.

Results

Benchmark results include tpmC, efficiency, and per-transaction-type latencies. As stated by the official specification:

> "The performance metric reported by TPC-C is a "business throughput" measuring the number of orders processed per minute. Multiple transactions are used to simulate the business activity of processing an order, and each transaction is subject to a response time constraint. The performance metric for this benchmark is expressed in transactions-per-minute-C (tpmC)."

The TPC-C specification limits the number of transactions that can be processed per warehouse. The theoretical maximum is 12.86 tpmC per warehouse. To increase the overall load—and thereby the tpmC—you must scale the number of warehouses.

Efficiency is calculated using the following formula:

Test data cleaning

Run cleaning:

```
ydb workload tpcc --path tpcc/10wh clean
```

The command has no parameters.

Data consistency check

`check` command verifies the consistency of TPC-C data. It can be executed after either the `import` or `run` commands. The `check` command is intended primarily for development and is of limited interest to end users.

Example:

```
ydb workload tpcc --path tpcc/10wh check
```

See the command description:

```
ydb workload tpcc check --help
```

Available options

Name	Description	Default value
`--warehouses` or `-w`	A number of TPC-C warehouses.	10
`--just-imported`	Turns on additional checks. Should be used only when data has been just imported and no runs have been done yet.	

TPC-H workload

The workload is based on the TPC-H specification, with the queries and table schemas adapted for YDB.

The benchmark generates a workload typical for decision support systems.

Common command options

All commands support the common `--path` option, which specifies the path to the directory containing tables in the database:

```
ydb workload tpch --path tpch/s1 ...
```

Available options

Name	Description	Default value
`--path` or `-p`	Path to the directory with tables.	`/`

Initializing a load test

Before running the benchmark, create a table:

```
ydb workload tpch --path tpch/s1 init
```

See the command description:

```
ydb workload tpch init --help
```

Available parameters

Name	Description	Default value
`--store <value>`	Table storage type. Possible values: `row`, `column`, `external-s3`.	`column`
`--external-s3-prefix <value>`	Relevant only for external tables. Root path to the dataset in S3 storage.	
`--external-s3-endpoint <value>` or `-e <value>`	Relevant only for external tables. Link to the S3 bucket with data.	
`--string`	Use the `String` type for text fields.	`Utf8`
`--datetime`	Use for time-related fields of type `Date`, `Datetime`, and `Timestamp`.	`Date32`, `Datetime64`, `Timestamp64`
`--partition-size`	Maximum partition size in megabytes (AUTO_PARTITIONING_PARTITION_SIZE_MB) for row tables.	2000
`--float-mode <value>`	Specifies the data type to use for fractional fields. Possible values are `double` and `decimal`. `double` uses the `Double` type, `decimal` uses `Decimal` with dimensions specified by the test standard.	`double`
`--scale`	Sets the percentage of the benchmark's data size and workload to use, relative to full scale.	1
`--clear`	If the table at the specified path already exists, it will be deleted.	

Loading data into a table

The data will be generated and loaded into a table directly by ydb:

```
ydb workload tpch --path tpch/s1 import generator --scale 1
```

See the command description:

```
ydb workload tpch import --help
```

Available options

Name	Description	Default value
`--scale <value>`	Data scale. Powers of ten are usually used. Also supports fractional scale, which is not described in the TPC-H specification. It can be useful for quickly testing small YDB databases. Examples: `0.1`, `0.3`.	
`--tables <value>`	Comma-separated list of tables to generate. Available tables: `customer`, `nation`, `order_line`, `part_psupp`, `region`, `supplier`.	All tables
`--proccess-count <value>` or `-C <value>`	Data generation can be split into several processes, this parameter specifies the number of processes.	1

`--proccess-index <value>` or `-i <value>`	Data generation can be split into several processes, this parameter specifies the process number.	0
`--state <path>`	Path to the generation state file. If the generation was interrupted for some reason, the download will be continued from the same place when it is started again.	
`--clear-state`	Relevant if the `--state` parameter is specified. Clear the state file and start the download from the beginning.	
`--dry-run`	Do not execute loading queries, but only display their text.	

Common parameters of the import command

Name	Description	Default value
`--upload-threads <value>` or `-t <value>`	The number of execution threads for data preparation.	The number of available cores on the client.
`--bulk-size <value>`	The size of the chunk for sending data, in rows.	10000
`--max-in-flight <value>`	The maximum number of data chunks that can be processed simultaneously.	128
`--file-output-path <value>` or `-f <path>`	If this option is set, the data will not be loaded into the database, but will be saved to the directory .	

Run the load test

Run the load:

```
ydb workload tpch --path tpch/s1 run
```

During the test, load statistics are displayed for each request.

See the command description:

```
ydb workload tpch run --help
```

Common parameters for all load types

Name	Description	Default value
`--dry-run`	Do not execute initialization queries, but only display their text.	
`--check-canonical` or `-c`	Use special version of queries (they have deterministic answers) and compare results with canonical ones.	
`--output <value>`	The name of the file where the query execution results will be saved.	`results.out`
`--iterations <value>`	The number of times each load query will be executed.	`1`
`--json <name>`	The name of the file where query execution statistics will be saved in `json` format.	Not saved by default
`--ministat <name>`	The name of the file where query execution statistics will be saved in `ministat` format.	Not saved by default
`--csv <name>`	The name of the file to save the CSV version of the result table.	Not saved by default
`--plan <name>`	The name of the file to save the query plan. Files like `<name>.<query number>.explain` and `<name>.<query number>.<iteration number>` will be saved in formats: `ast`, `json`, `svg`, and `table`.	Not saved by default
`--query-prefix <setting>`	Query prefix. Every prefix is a line that will be added to the beginning of each query. For multiple prefix lines use this option several times.	Not specified by default
`--retries`	Max retry count for every request.	`0`
`--include`	Names, numbers or ranges of query numbers to be executed as part of the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	All queries executed
`--exclude`	Names, numbers or ranges of query numbers to be excluded from the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	None excluded by default
`--verbose` or `-v`	Print additional information to the screen during query execution.	
`--global-timeout <value>`	Global timeout for all queries. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--request-timeout <value>`	Timeout for each iteration of each query. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--threads <value>` or `-t <value>`	The number of parallel threads generating the load. Zero means that queries will be executed in the main thread; otherwise, queries will be mixed.	`0`

`--stats <value>`	Extended execution statistics collection mode. Available values: `full`, `profile`.	`full`

TPC-H-specific options

Name	Description	Default value
`--syntax <value>`	Syntax of the queries to use. Available values: `yql`. For more information about working with YQL syntax, see here.	`yql`
`--float-mode <value>`	Float mode. Can be `float`, `decimal` or `decimal_ydb`. If the value is `float` - float will be used, `decimal` means that decimal with canonical size specified in the TPC-H specification (`Decimal(12, 2)`) will be used, and `decimal_ydb` means that all float will be converted to `Decimal(22, 9)`. For more information about the Decimal type, see documentation.	`float`
`--scale <value>`	Scale factor. See the TPC-H specification, chapter 3. Used in TPC-H queries. Also supports fractional scale, which is not described in the TPC-H specification. It can be useful for quickly testing small YDB databases. Examples: `0.1`, `0.3`. For scale factors `1`, `10`, `100`, `1000` canonical answers are specified (see the `--check-canonical` option description).	1

Test data cleaning

Run cleaning:

```
ydb workload tpch --path tpch/s1 clean
```

The command has no parameters.

TPC-DS workload

The workload is based on the TPC-DS specification, with the queries and table schemas adapted for YDB.

This benchmark generates a workload typical for decision support systems.

Common command options

All commands support the common option `--path`, which specifies the path to the directory containing benchmark tables in the database:

```
ydb workload tpcds --path tpcds/s1 ...
```

Available options

Name	Description	Default value
`--path` or `-p`	Path to the directory with tables.	`/`

Initializing the load test

Before running the benchmark, create a table:

```
ydb workload tpcds --path tpcds/s1 init
```

See the command description to run the load:

```
ydb workload tpcds init --help
```

Available parameters

Name	Description	Default value
`--store <value>`	Table storage type. Possible values: `row`, `column`, `external-s3`.	`column`
`--external-s3-prefix <value>`	Relevant only for external tables. Root path to the dataset in S3 storage.	
`--external-s3-endpoint <value>` or `-e <value>`	Relevant only for external tables. Link to the S3 bucket with data.	
`--string`	Use the `String` type for text fields.	`Utf8`
`--datetime`	Use for time-related fields of type `Date`, `Datetime`, and `Timestamp`.	`Date32`, `Datetime64`, `Timestamp64`
`--partition-size`	Maximum partition size in megabytes (AUTO_PARTITIONING_PARTITION_SIZE_MB) for row tables.	2000
`--float-mode <value>`	Specifies the data type to use for fractional fields. Possible values are `double` and `decimal`. `double` uses the `Double` type, `decimal` uses `Decimal` with dimensions specified by the test standard.	`double`
`--scale`	Sets the percentage of the benchmark's data size and workload to use, relative to full scale.	1
`--clear`	If the table at the specified path already exists, it will be deleted.	

Loading data into the table

The data will be generated and loaded into the table directly by YDB CLI:

```
ydb workload tpcds --path tpcds/s1 import generator --scale 1
```

See the command description:

```
ydb workload tpcds import --help
```

Available options

Name	Description	Default value
`--scale <value>`	Data scale. Typically, powers of ten are used. Also supports fractional scale, which is not described in the TPC-DS specification. It can be useful for quickly testing small YDB databases. Examples: `0.1`, `0.3`.	
`--tables <value>`	Comma-separated list of tables to generate. Available tables: `customer`, `nation`, `order_line`, `part_psupp`, `region`, `supplier`.	All tables
`--process-count <value>` or `-C <value>`	Specifies the number of processes for parallel data generation.	`1`

`--process-index <value>` or `-i <value>`	Specifies the process number when data generation is split into multiple processes.	`0`
`--state <path>`	Path to the state file for resuming generation. If the generation is interrupted, it will resume from the same point when restarted.	
`--clear-state`	Relevant if the `--state` parameter is specified. Clears the state file and restarts the download from the beginning.	
`--dry-run`	Do not execute loading queries, but only display their text.	

Common parameters of the import command

Name	Description	Default value
`--upload-threads <value>` or `-t <value>`	The number of execution threads for data preparation.	The number of available cores on the client.
`--bulk-size <value>`	The size of the chunk for sending data, in rows.	10000
`--max-in-flight <value>`	The maximum number of data chunks that can be processed simultaneously.	128
`--file-output-path <value>` or `-f <path>`	If this option is set, the data will not be loaded into the database, but will be saved to the directory .	

Run the load test

Run the load:

```
ydb workload tpcds --path tpcds/s1 run
```

During the benchmark, load statistics are displayed for each request.

See the command description:

```
ydb workload tpcds run --help
```

Common parameters for all load types

Name	Description	Default value
`--dry-run`	Do not execute initialization queries, but only display their text.	
`--check-canonical` or `-c`	Use special version of queries (they have deterministic answers) and compare results with canonical ones.	
`--output <value>`	The name of the file where the query execution results will be saved.	`results.out`
`--iterations <value>`	The number of times each load query will be executed.	`1`
`--json <name>`	The name of the file where query execution statistics will be saved in `json` format.	Not saved by default
`--ministat <name>`	The name of the file where query execution statistics will be saved in `ministat` format.	Not saved by default
`--csv <name>`	The name of the file to save the CSV version of the result table.	Not saved by default
`--plan <name>`	The name of the file to save the query plan. Files like `<name>.<query number>.explain` and `<name>.<query number>.<iteration number>` will be saved in formats: `ast`, `json`, `svg`, and `table`.	Not saved by default
`--query-prefix <setting>`	Query prefix. Every prefix is a line that will be added to the beginning of each query. For multiple prefix lines use this option several times.	Not specified by default
`--retries`	Max retry count for every request.	`0`
`--include`	Names, numbers or ranges of query numbers to be executed as part of the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	All queries executed
`--exclude`	Names, numbers or ranges of query numbers to be excluded from the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	None excluded by default
`--verbose` or `-v`	Print additional information to the screen during query execution.	
`--global-timeout <value>`	Global timeout for all queries. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--request-timeout <value>`	Timeout for each iteration of each query. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--threads <value>` or `-t <value>`	The number of parallel threads generating the load. Zero means that queries will be executed in the main thread; otherwise, queries will be mixed.	`0`

`--stats <value>`	Extended execution statistics collection mode. Available values: `full`, `profile`.	`full`

TPC-DS-specific options

Name	Description	Default value
`--syntax <value>`	Syntax of the queries to use. Available values: `yql`. For more information about working with YQL syntax, see here.	`yql`
`--float-mode <value>`	Float mode. Can be `float`, `decimal` or `decimal_ydb`. If the value is `float` - float will be used, `decimal` means that decimal with canonical size specified in the TPC-DS specification (`Decimal(12, 2)`) will be used, and `decimal_ydb` means that all float will be converted to `Decimal(22, 9)`. For more information about the Decimal type, see documentation.	`float`
`--scale <value>`	Scale factor. See the TPC-DS specification, chapter 3. Used in TPC-DS queries. Also supports fractional scale, which is not described in the TPC-DS specification. It can be useful for quickly testing small YDB databases. Examples: `0.1`, `0.3`. For scale factors `1`, `10`, `100`, `1000` canonical answers are specified (see the `--check-canonical` option description).	1

Test data cleanup

Run cleanup:

```
ydb workload tpcds --path tpcds/s1 clean
```

The command has no parameters.

User workload

Allows you to perform load testing according to a specified user scenario.

A test scenario is a directory called `suite`, which contains a set of subdirectories: `init`, `import`, and `run`, corresponding to the `ydb workload query` commands.

- The `init` directory can have an arbitrary structure and contains SQL queries for creating and configuring objects in the database, such as tables or indexes.
- The `import` directory contains data to be loaded into tables. Each table corresponds to a subdirectory in the `import` directory.
- The `run` directory can have an arbitrary structure and contains SQL queries for load testing.

Example of a `suite` directory on GitHub.

The user specifies the path to the `suite` directory using the `--suite-path` option in each command.

General syntax

General syntax for calling user workload commands:

```
ydb [global options...] workload query [--path <path/in/db>] <command> [options...]
```

- `global options` — global parameters.
- `--path` — the path in the database for load testing objects.
- `<command>` — one of the commands: `init`, `import`, `run`, `clean`.
- `options` — parameters for a specific command.

Common command parameters

All commands support the common `--path` parameter, which sets the path in the database for load testing objects. All created objects will be created at this path. The path will also be taken into account during test execution and cleanup.

Available parameters

Name	Description	Default value
`--path` or `-p`	Path in the database for load testing objects.	`/`

Load test initialization

Before starting the test, create tables and other database objects:

```
ydb workload query --path user_path init --suite-path ~/user-suite
```

The `init` command works as follows:

- If a `suite` is specified and it contains an `init` directory, the `init` directory is scanned, including its nested directories, using a depth-first search with lexicographical ordering of files and subdirectories at each level.
- Files with `.sql` and `.yql` extensions are selected.
- The content of each file is read and executed sequentially as an SQL query.
- After that, queries specified directly from the command line using the `--query` parameter are executed.
- If any of the queries result in an error, the command execution stops.

It is assumed that relatively "lightweight" queries, such as creating tables and indexes, are performed at this stage. It is not recommended to include scripts for filling tables with data in the files in the `init` directory. These operations should be moved to the `import` phase.

Macros can be used in queries:

- `{db}` — The absolute path in the database to the testing directory. It is a combination of the `--database` and `--path` option values.

View the command description for initializing tables:

```
ydb workload query init --help
```

Available parameters

Name	Description	Default value
`--suite-path <path>`	Path to the `suite` directory.	
`--query <query>` or `-q <query>`	DDL query to execute. Can be used multiple times.	
`--clear`	If a table being created already exists at the specified path, it will be deleted first.	
`--dry-run`	Do not execute initialization queries, only print them.	

Loading data into a table

Load data into tables:

```
ydb workload query --path user_path import --suite-path ~/user-suite
```

The `import` command works as follows:

- If a `suite` is specified and it contains an `import` directory, it performs a sequential traversal of its subdirectories.
- It is assumed that each subdirectory corresponds to a table in the database.
- Files in supported data formats are used in the subdirectory, namely `csv`, `tsv`, `csv.gz`, `tsv.gz`, and `parquet`.
- Data is loaded from files into the corresponding tables. Loading is multi-threaded, with both the loading of different files and parts of a single file being parallelized. The order of loading is not guaranteed.

The loading process can take a long time and may be interrupted for some reason. To be able to continue loading from the point of interruption, a state mechanism can be applied. Using the `--state` option, you can specify the path to a state file. If the state file exists, it will be used to continue loading. If there is no state file, it will be created during the loading process. It can also be cleared using the `--clear-state` option.

View the command description for loading data:

```
ydb workload query import --help
```

Available parameters

Name	Description	Default value
`--suite-path <path>`	Path to the `suite` directory.	
`--state <path>`	Path to the loading state file. If the loading was interrupted for some reason, it will be continued from the same place upon a new start.	
`--clear-state`	Relevant if the `--state` parameter is specified. Clear the state file and start loading from the beginning.	
`--dry-run`	Do not load data into the database, only prepare the data and output a message about it.	

Common parameters of the import command

Name	Description	Default value
`--upload-threads <value>` or `-t <value>`	The number of execution threads for data preparation.	The number of available cores on the client.
`--bulk-size <value>`	The size of the chunk for sending data, in rows.	10000
`--max-in-flight <value>`	The maximum number of data chunks that can be processed simultaneously.	128
`--file-output-path <value>` or `-f <path>`	If this option is set, the data will not be loaded into the database, but will be saved to the directory .	

Running a load test

Start the load:

```
ydb workload query --path user_path run --suite-path ~/user-suite
```

The `run` command works as follows:

- If the `suite` contains a `run` directory, the command traverses it and all its nested directories using a depth-first search, ordering files and subdirectories alphabetically at each level. If the `run` directory is missing, execution stops.
- It selects files with `.sql` and `.yql` extensions and includes them in the list of SQL queries.
- After that, it adds queries specified directly from the command line using the `--query` parameter to the list.
- Depending on the options, it either executes all queries sequentially, each query a specified number of times, or in parallel in multiple threads with query shuffling.
- Errors in queries will be recorded in the statistics but will not cause the testing to stop.

During the test, load statistics for each query and aggregated statistics for all queries are displayed on the screen.

View the command description for starting the load:

```
ydb workload query run --help
```

Common parameters for all load types

Name	Description	Default value
`--dry-run`	Do not execute initialization queries, but only display their text.	
`--check-canonical` or `-c`	Use special version of queries (they have deterministic answers) and compare results with canonical ones.	
`--output <value>`	The name of the file where the query execution results will be saved.	`results.out`
`--iterations <value>`	The number of times each load query will be executed.	`1`
`--json <name>`	The name of the file where query execution statistics will be saved in `json` format.	Not saved by default
`--ministat <name>`	The name of the file where query execution statistics will be saved in `ministat` format.	Not saved by default

`--csv <name>`	The name of the file to save the CSV version of the result table.	Not saved by default
`--plan <name>`	The name of the file to save the query plan. Files like `<name>.<query number>.explain` and `<name>.<query number>.<iteration number>` will be saved in formats: `ast`, `json`, `svg`, and `table`.	Not saved by default
`--query-prefix <setting>`	Query prefix. Every prefix is a line that will be added to the beginning of each query. For multiple prefix lines use this option several times.	Not specified by default
`--retries`	Max retry count for every request.	`0`
`--include`	Names, numbers or ranges of query numbers to be executed as part of the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	All queries executed
`--exclude`	Names, numbers or ranges of query numbers to be excluded from the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	None excluded by default
`--verbose` or `-v`	Print additional information to the screen during query execution.	
`--global-timeout <value>`	Global timeout for all queries. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--request-timeout <value>`	Timeout for each iteration of each query. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--threads <value>` or `-t <value>`	The number of parallel threads generating the load. Zero means that queries will be executed in the main thread; otherwise, queries will be mixed.	`0`
`--stats <value>`	Extended execution statistics collection mode. Available values: `full`, `profile`.	`full`

Query-specific options

Name	Description	Default value
`--suite-path <path>`	Path to the `suite` directory.	
`--query <query>` or `-q <query>`	Query to execute. Can be used multiple times.	

Canonical results

A canonical (expected) result can be specified for each query from a file. If the query response does not match the canonical one, this will be reflected in the statistics and command output. Differences will also be provided.

The canonical result is specified using a file with the same name and an additional `.result` extension. These are CSV files with headers and some additional syntax:

- If a query has more than one result set, the result file must contain the corresponding number of datasets separated by empty lines.

  For example, the query

  ```
  SELECT COUNT(*) AS count FROM table_1;
  SELECT MIN(col1) AS min, MAX(col1) AS max FROM table_2;
  ```

  Might have the following result file:

  ```
  count
  10

  min,max
  4,5
  ```

- The last line can be specified as `...`, which means that the query result can have more lines than in the canonical result, but the first lines of the result must match the canonical ones.
- By default, floating-point numbers are compared with a relative precision of `1e-3` percent, but in the canonical result, you can specify any absolute or relative precision for each value, for example: `1.5+-0.01`, `2.4e+10+-1%`.

For queries specified with the `--query` option, a canonical result cannot be set.

The canonical result will not be used unless the `--check-canonical` flag is set.

Cleaning up test data

Start the cleanup:

```
ydb workload query --path user_path clean
```

Performs recursive deletion of database objects at the path specified in the `--path` parameter.

The command has no parameters.

Example of running a user test

Download an example from the repository

```
git clone https://github.com/ydb-platform/ydb.git
cd ydb/ydb/tests/functional/tpc/data/e1
```

Start initialization

```
ydb workload query --path test_path init --suite-path .
```

Result:

```
Init tables ...
Init tables ...Ok
```

Start data import

```
ydb workload query --path test_path import --suite-path .
```

Result:

```
Fill table test_table_1...
Fill table test_table_1 OK 0 / 0 (0.353816s)
Fill table test_table_2...

Fill table test_table_2 OK 0 / 0 (0.102385s)
```

Start the load test

```
ydb workload query --path user_path run --suite-path e1
```

Result:

```
first_query_set.1.sql:
    iteration 0:    ok    0.131573s seconds
first_query_set.2.yql:
    iteration 0:    ok    0.089327s seconds
second_query_set.join.sql:
    iteration 0:    ok    0.145536s seconds

Results for 1 iterations
┌──────────────────────────┬──────────┬──────────┬──────────┬──────────┬──────────┬─────────────┬──────────┬──────────
──┬──────────┬──────────┬─────────────┬─────────────┬──────────────┬─────────────┬──────────────┬──────────────┐
| Query #                  | ColdTime | Min      | Max      | Mean     | Median   | UnixBench | Std      | RttMin
| RttMax   | RttAvg   | CompilationMin | CompilationMax | CompilationAvg | GrossTime | SuccessCount | FailsCoun
t | DiffsCount |
├──────────────────────────┼──────────┼──────────┼──────────┼──────────┼──────────┼─────────────┼──────────┼──────────
──┼──────────┼──────────┼─────────────┼─────────────┼──────────────┼─────────────┼──────────────┼──────────────┤
| first_query_set.1.sql    |   0.086  |   0.086  |   0.086  |   0.086  |   0.086  |   0.086   |   0.000  |   0.04
5  |   0.045  |   0.045  |   0.041      |   0.041      |   0.041      |   0.132   | 1            |
|            |
├──────────────────────────┼──────────┼──────────┼──────────┼──────────┼──────────┼─────────────┼──────────┼──────────
──┼──────────┼──────────┼─────────────┼─────────────┼──────────────┼─────────────┼──────────────┼──────────────┤
| first_query_set.2.yql    |   0.081  |   0.081  |   0.081  |   0.081  |   0.080  |   0.081   |   0.001  |   0.00
8  |   0.008  |   0.008  |   0.039      |   0.039      |   0.039      |   0.089   | 1            |
|            |
├──────────────────────────┼──────────┼──────────┼──────────┼──────────┼──────────┼─────────────┼──────────┼──────────
──┼──────────┼──────────┼─────────────┼─────────────┼──────────────┼─────────────┼──────────────┼──────────────┤
| second_query_set.join.sql |   0.131  |   0.131  |   0.131  |   0.131  |   0.131  |   0.131   |   0.000  |   0.01
4  |   0.014  |   0.014  |   0.082      |   0.082      |   0.082      |   0.146   | 1            |
|            |
├──────────────────────────┼──────────┼──────────┼──────────┼──────────┼──────────┼─────────────┼──────────┼──────────
──┼──────────┼──────────┼─────────────┼─────────────┼──────────────┼─────────────┼──────────────┼──────────────┤
| Sum                      |   0.299  |   0.299  |   0.299  |   0.299  |   0.297  |   0.299   |   0.000  |   0.06
8  |   0.068  |   0.068  |   0.161      |   0.161      |   0.162      |   0.367   | 3            |
|            |
├──────────────────────────┼──────────┼──────────┼──────────┼──────────┼──────────┼─────────────┼──────────┼──────────
──┼──────────┼──────────┼─────────────┼─────────────┼──────────────┼─────────────┼──────────────┼──────────────┤
| Avg                      |   0.100  |   0.100  |   0.100  |   0.100  |   0.099  |   0.100   |   0.000  |   0.02
3  |   0.023  |   0.023  |   0.054      |   0.054      |   0.054      |   0.122   | 3            |
|            |
├──────────────────────────┼──────────┼──────────┼──────────┼──────────┼──────────┼─────────────┼──────────┼──────────
──┼──────────┼──────────┼─────────────┼─────────────┼──────────────┼─────────────┼──────────────┼──────────────┤
| GAvg                     |   0.097  |   0.097  |   0.097  |   0.097  |   0.097  |   0.097   |   0.000  |   0.01
8  |   0.018  |   0.001  |   0.051      |   0.051      |   0.001      |   0.000   | 3            |
|            |
```

```
└────────────────────────────┴───────────┴──────────┴──────────┴──────────┴──────────┴────────────┴──────────┴────────
──┴──────────┴──────────┴─────────────────┴─────────────────┴─────────────────┴────────────┴───────────────┴────────
────┴─────────────┘

Results saved to results.out
```

Clean up tables

```
ydb workload query --path user_path clean
```

Fulltext load

Allows you to test YDB fulltext search performance using a document dataset. Supports both real datasets (e.g., MS MARCO) and synthetically generated text based on a Markov chain model.

The Markov model assumes that each next word depends on one or more previous words. The number of previous words on which the next word is based is called the model's order. First, we use an ordinary random generator with a uniform distribution to determine the text length, which we will then generate word by word using the Markov model until we reach the required number of words. Such data construction is necessary so that the text resembles some real human text rather than being just a random jumble of letters. A Markov model can be used both for generating the text of documents stored in a database and for generating select and upsert queries.

Command structure

```
ydb [global options...] workload fulltext [options...] <subcommand>
```

Subcommands:

```
fulltext            YDB fulltext workload
├─ init               Initialize tables for the workload
├─ import             Load data and build a fulltext index
│   ├─ files            Import data from files
│   └─ generator        Generate random text using a Markov chain model
├─ run                Run the workload
│   ├─ select           Search documents using fulltext queries
│   └─ upsert           Insert or update documents in the table
├─ clean              Drop tables created for load testing
└─ model              Build a Markov chain model from a text dataset
```

Common command options

All commands support the following option:

Name	Description	Default value
`--path` or `-p`	Path in the database where workload tables will be created.	`fulltext_workload`

Initializing the workload

Create the tables for the workload:

```
ydb workload fulltext init
```

Available options

Name	Description	Default value
`--min-partitions <value>`	Minimum number of table partitions.	`40`
`--partition-size <value>`	Target partition size, in MB.	`2000`
`--auto-partition <value>`	Enable auto-partitioning by load (`1` — enabled, `0` — disabled).	`1`
`--clear`	Drop and recreate the table if it already exists.	

Loading data

After initialization, load data into the table and build the fulltext index. There are two subcommands: `files` to import from an existing dataset and `generator` to generate synthetic data.

After import is complete, a fulltext index named `index` is automatically built on the `text` column.

Importing from files

Import documents from files (CSV, TSV, or Parquet, optionally gzip-compressed). The dataset must contain `id` and `text` columns. In the internal table, the `id` column is stored as `Uint64` and the `text` column as `String`.

Example:

```
ydb workload fulltext import files
```

Available options

Name	Description	Default value
`--input <path>` or `-i <path>`	Path to the dataset file or directory. Supported formats: CSV/TSV (optionally gzip-compressed), Parquet. Only `id` and `text` columns are imported.	Required

Common parameters of the import command

Name	Description	Default value
`--upload-threads <value>` or `-t <value>`	The number of execution threads for data preparation.	The number of available cores on the client.

`--bulk-size <value>`	The size of the chunk for sending data, in rows.	10000
`--max-in-flight <value>`	The maximum number of data chunks that can be processed simultaneously.	128
`--file-output-path <value>` or `-f <path>`	If this option is set, the data will not be loaded into the database, but will be saved to the directory .	

Generating synthetic data

Generate random text data using a Markov chain model and load it into the table. You must first build the model or download a pre-built one.

- To load an already built model:

```
wget https://storage.yandexcloud.net/ydb-public/markov_dict.tsv.gz
```

- To create your own model from Wikipedia data:

```
from datasets import load_dataset

ds = load_dataset(
   "rumbleFTW/wikipedia-20220301-en-raw",
   split="train[:1000000]",
   streaming=False,
   )
ds.to_csv('wikipedia_sample.csv.gz', compression='gzip', index=False)
```

```
ydb workload fulltext import generator
```

Available options

Name	Description	Default value
`--model <path>` or `-m <path>`	Path to the Markov chain model file (`.tsv.gz`).	Required
`--rows <value>`	Number of rows to generate.	`100000`
`--min-sentence-len <value>`	Minimum number of words in a generated document.	`100`
`--max-sentence-len <value>`	Maximum number of words in a generated document.	`1000`

We use parameters `--min-sentence-len` and `--max-sentence-len` to generate target text length with uniform distribution.

Common parameters of the import command

Name	Description	Default value
`--upload-threads <value>` or `-t <value>`	The number of execution threads for data preparation.	The number of available cores on the client.
`--bulk-size <value>`	The size of the chunk for sending data, in rows.	10000
`--max-in-flight <value>`	The maximum number of data chunks that can be processed simultaneously.	128
`--file-output-path <value>` or `-f <path>`	If this option is set, the data will not be loaded into the database, but will be saved to the directory .	

Running the workload

Run load testing using one of two modes: `select` (fulltext search queries) or `upsert` (inserting new documents).

Search workload

Executes fulltext search queries against the indexed table. Queries can be generated from a Markov chain model or read from a pre-loaded query table.

```
ydb workload fulltext run select --model markov_dict.tsv.gz
```

Available options

Name	Description	Default value
`--model <path>` or `-m <path>`	Path to the Markov chain model file (`.tsv.gz`) for generating queries. Either `--model` or `--query-table` must be specified.	
`--query-table <name>`	Name of the table containing pre-loaded queries. The table must have a `query` column. Either `--model` or `--query-table` must be specified.	
`--index-name <name>`	Name of the fulltext index to use.	`index`
`--min-query-len <value>`	Minimum number of words in a generated query.	`1`
`--max-query-len <value>`	Maximum number of words in a generated query.	`5`

`--top-size <value>`	Number of rows to sample from the table to build the query word set.	`1000`
`--limit <value>`	Limit the number of results returned per query. `0` means no limit.	`0`

Common parameters for all load types

Name	Description	Default value
`--dry-run`	Do not execute initialization queries, but only display their text.	
`--check-canonical` or `-c`	Use special version of queries (they have deterministic answers) and compare results with canonical ones.	
`--output <value>`	The name of the file where the query execution results will be saved.	`results.out`
`--iterations <value>`	The number of times each load query will be executed.	`1`
`--json <name>`	The name of the file where query execution statistics will be saved in `json` format.	Not saved by default
`--ministat <name>`	The name of the file where query execution statistics will be saved in `ministat` format.	Not saved by default
`--csv <name>`	The name of the file to save the CSV version of the result table.	Not saved by default
`--plan <name>`	The name of the file to save the query plan. Files like `<name>.<query number>.explain` and `<name>.<query number>.<iteration number>` will be saved in formats: `ast`, `json`, `svg`, and `table`.	Not saved by default
`--query-prefix <setting>`	Query prefix. Every prefix is a line that will be added to the beginning of each query. For multiple prefix lines use this option several times.	Not specified by default
`--retries`	Max retry count for every request.	`0`
`--include`	Names, numbers or ranges of query numbers to be executed as part of the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	All queries executed
`--exclude`	Names, numbers or ranges of query numbers to be excluded from the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	None excluded by default
`--verbose` or `-v`	Print additional information to the screen during query execution.	
`--global-timeout <value>`	Global timeout for all queries. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--request-timeout <value>`	Timeout for each iteration of each query. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--threads <value>` or `-t <value>`	The number of parallel threads generating the load. Zero means that queries will be executed in the main thread; otherwise, queries will be mixed.	`0`
`--stats <value>`	Extended execution statistics collection mode. Available values: `full`, `profile`.	`full`

Upsert workload

Continuously inserts new documents into the table using a Markov chain model to generate text.

```
ydb workload fulltext run upsert --model markov_dict.tsv.gz
```

Available options

Name	Description	Default value
`--model <path>` or `-m <path>`	Path to the Markov chain model file (`.tsv.gz`).	Required
`--index-name <name>`	Name of the fulltext index to use.	`index`
`--bulk-size <value>`	Number of rows per upsert batch.	`100`
`--min-sentence-len <value>`	Minimum number of words in a generated document.	`100`
`--max-sentence-len <value>`	Maximum number of words in a generated document.	`1000`

Common parameters for all load types

Name	Description	Default value
`--dry-run`	Do not execute initialization queries, but only display their text.	
`--check-canonical` or `-c`	Use special version of queries (they have deterministic answers) and compare results with canonical ones.	
`--output <value>`	The name of the file where the query execution results will be saved.	`results.out`

`--iterations <value>`	The number of times each load query will be executed.	`1`
`--json <name>`	The name of the file where query execution statistics will be saved in `json` format.	Not saved by default
`--ministat <name>`	The name of the file where query execution statistics will be saved in `ministat` format.	Not saved by default
`--csv <name>`	The name of the file to save the CSV version of the result table.	Not saved by default
`--plan <name>`	The name of the file to save the query plan. Files like `<name>.<query number>.explain` and `<name>.<query number>.<iteration number>` will be saved in formats: `ast`, `json`, `svg`, and `table`.	Not saved by default
`--query-prefix <setting>`	Query prefix. Every prefix is a line that will be added to the beginning of each query. For multiple prefix lines use this option several times.	Not specified by default
`--retries`	Max retry count for every request.	`0`
`--include`	Names, numbers or ranges of query numbers to be executed as part of the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	All queries executed
`--exclude`	Names, numbers or ranges of query numbers to be excluded from the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	None excluded by default
`--verbose` or `-v`	Print additional information to the screen during query execution.	
`--global-timeout <value>`	Global timeout for all queries. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--request-timeout <value>`	Timeout for each iteration of each query. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--threads <value>` or `-t <value>`	The number of parallel threads generating the load. Zero means that queries will be executed in the main thread; otherwise, queries will be mixed.	`0`
`--stats <value>`	Extended execution statistics collection mode. Available values: `full`, `profile`.	`full`

Building a Markov chain model

Before using the generator or the `run upsert` / `run select` modes with generated queries, you need to build a Markov chain model from a text dataset. The model captures word transition probabilities and is used to generate realistic text.

```
ydb workload fulltext model --input wikipedia_sample.csv.gz --output markov_dict.tsv.gz --order 3
```

Available options

Name	Description	Default value
`--input <path>` or `-i <path>`	Path to the dataset file or directory. Supports `.csv[.gz]` and `.tsv[.gz]` formats. The file must have a `text` column.	Required
`--output <path>` or `-o <path>`	Output file path for the model dictionary.	`markov_dict.tsv.gz`
`--order <value>` or `-n <value>`	Order of the Markov chain (n-gram context size). Order 1 uses only one word to predict next word, order 2 uses context from two words, etc. Must be between 1 and 5.	`1`

The `--order` parameter specifies the number of previous words used to predict the next word. Typical values range from 1 to 5: lower values produce noisier text, while higher values generate text that is more similar to the source. The size of the Markov model grows exponentially with the `--order` value.

Cleaning up

Drop tables created for load testing:

```
ydb workload fulltext clean
```

Usage examples

Example with a generated dataset

1. Download or train Markov chain model:
   - Download the model from S3:

     ```
     wget https://storage.yandexcloud.net/ydb-public/markov_dict.tsv.gz
     ```

   - Train the model from Wikipedia dataset:

     ```
     from datasets import load_dataset
     ```

```
ds = load_dataset(
   "rumbleFTW/wikipedia-20220301-en-raw",
   split="train[:1000000]",
   streaming=False,
   )
ds.to_csv('wikipedia_sample.csv.gz', compression='gzip', index=False)
```

```
ydb workload fulltext model --input wikipedia_sample.csv.gz --output markov_dict.tsv.gz --order 3
```

2. Initialize the workload table:

```
ydb workload fulltext init
```

3. Generate and load synthetic documents:

```
ydb workload fulltext import generator --model markov_dict.tsv.gz
```

4. Run the search workload:

```
ydb workload fulltext run select --model markov_dict.tsv.gz
```

Output example:

```
Window      Txs Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
1            23 23      0       0       242     407     453     453
2            18 18      0       1       611     807     915     915
3             6 6       0       7       161     991     991     991
4            35 35      0       0       173     803     923     923
5            59 59      0       1       135     647     759     803
6            26 26      0       0       257     335     339     339
7            15 15      0       1       651     963     975     975
8            11 11      0       1       871     943     963     963
9            18 18      0       6       190     967     999     999
10           31 31      0       1       433     755     995     995

Total       Txs Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
10          242 24.2    0       18      259     915     991     999
```

Column description:

- Window – Sequential number of the time window (e.g., each second or fixed interval).
- Txs – Number of transactions successfully completed in this window (or total across all windows in the bottom section).
- Txs/Sec – Transaction rate per second for the given window (or average for total).
- Retries – Number of automatic retries performed due to temporary errors (e.g., conflicts or throttling).
- Errors – Number of unrecoverable errors encountered.
- p50(ms) – Median (50th percentile) transaction latency in milliseconds.
- p95(ms) – 95th percentile latency (milliseconds).
- p99(ms) – 99th percentile latency (milliseconds).
- pMax(ms) – Maximum observed latency within the window (or overall maximum for total).

5. Run the upsert workload:

```
ydb workload fulltext run upsert --model markov_dict.tsv.gz
```

Output example:

```
Window      Txs Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
1           255 255     0       0       31      34      117     123
2           279 279     0       0       32      37      42      45
3           282 282     0       0       31      35      37      40
4           277 277     0       0       32      37      39      42
5           281 281     0       0       32      38      40      42
6           278 278     0       0       32      39      41      43
7           279 279     0       0       32      38      39      43
8           275 275     0       0       33      39      42      46
9           282 282     0       0       32      37      39      40
10          293 293     0       0       31      37      39      41

Total      Txs Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
10        2781 278.1   0       0       32      38      41      123
```

6. Clean up:

```
ydb workload fulltext clean
```

Example with the MS MARCO dataset

1. Download the quality bundle (contains `documents.tsv.gz`, `queries.tsv.gz`, `markov_dict.tsv.gz`, and `query_relevances.tsv.gz`):

```
wget https://storage.yandexcloud.net/ydb-public/quality-bundle.tar
tar -xf quality-bundle.tar
```

2. Initialize the workload table:

```
ydb workload fulltext init
```

3. Import documents from the dataset:

```
ydb workload fulltext import files
```

4. Run the search workload using the pre-built queries:

```
ydb workload fulltext run select --quality
```

Output example:

```
Search quality measurement...
Search quality measurement completed for 100 queries in 1 seconds.
nDCG@10:  0.270
Errors:   0
Window      Txs Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
1            23 23      0       0       242     407     453     453
2            18 18      0       1       611     807     915     915
3             6 6       0       7       161     991     991     991
4            35 35      0       0       173     803     923     923
5            59 59      0       1       135     647     759     803
6            26 26      0       0       257     335     339     339
7            15 15      0       1       651     963     975     975
8            11 11      0       1       871     943     963     963
9            18 18      0       6       190     967     999     999
10           31 31      0       1       433     755     995     995

Total       Txs Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
10          242 24.2    0       18      259     915     991     999
```

5. Run the upsert workload:

```
ydb workload fulltext run upsert
```

Output example:

```
Window      Txs Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
1           255 255     0       0       31      34      117     123
2           279 279     0       0       32      37      42      45
3           282 282     0       0       31      35      37      40
4           277 277     0       0       32      37      39      42
5           281 281     0       0       32      38      40      42
6           278 278     0       0       32      39      41      43
7           279 279     0       0       32      38      39      43
8           275 275     0       0       33      39      42      46
9           282 282     0       0       32      37      39      40
10          293 293     0       0       31      37      39      41

Total       Txs Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
10         2781 278.1   0       0       32      38      41      123
```

6. Clean up:

```
ydb workload fulltext clean
```

Vector load

Allows you to test YDB vector search performance and recall using exact and approximate search. Supports both global and filtered vector indexes.

The workload supports importing vectors from a real dataset (e.g., Wikipedia embeddings) or generating synthetic random vectors. After loading data, you can build a vector index, run search queries to measure recall and performance, and clean up the workload tables.

Command structure

```
ydb [global options...] workload vector [options...] <subcommand>
```

Subcommands:

```
vector                YDB vector workload.
├─ init                 Create and initialize tables for the workload
├─ import               Fill workload tables with data
│   ├─ files              Import vectors from files
│   └─ generator          Generate random vectors
├─ build-index          Create and initialize a vector index for the workload
├─ drop-index           Drop the vector index created for the workload
├─ run                  Run YDB vector workload
│   ├─ select            Search vectors and measure performance/recall
│   └─ upsert            Insert or update vector rows in the table
└─ clean                Drop tables created for load testing
```

Initializing the workload

Create the table for the workload:

```
ydb workload vector init
```

Available options

Name	Description	Default value
`--table <name>`	Name of the main table that stores vectors.	`vector_index_workload`
`--min-partitions <value>`	Minimum number of table partitions.	`40`
`--partition-size <value>`	Target partition size, in MB.	`2000`
`--auto-partition <value>`	Enable auto-partitioning by load (`1` — enabled, `0` — disabled).	`1`
`--prefixed`	Add a `prefix` column to the table for use with a filtered (prefixed) vector index.	
`--clear`	Drop and recreate the table if it already exists.	
`--dry-run`	Print the DDL query instead of executing it.	

The created table has the following schema:

- `id Uint64 NOT NULL` — primary key;
- `embedding String` — serialized embedding vector;
- `prefix Uint64 NOT NULL` (only when `--prefixed` is specified) — filter column for prefixed indexes.

Loading data

After initialization, load data into the table. There are two subcommands: `files` to import vectors from an existing dataset and `generator` to generate synthetic random vectors.

After import is complete, a vector index named `index` is automatically built on the `embedding` column (unless `--index-type None` is specified).

Importing from files

Import vectors from files (CSV, TSV, or Parquet, optionally gzip-compressed). The dataset must contain `id` and `embedding` columns. Any additional columns are ignored.

For CSV/TSV files, embeddings must be encoded as a list of floats, e.g., `"[ 1.0, 2.0, 3.0 ]"`. For Parquet files, embeddings can be a list of `float32` values or already serialized YDB binary embeddings.

Example:

```
ydb workload vector import files
```

Available options

Name	Description	Default value
`--input <path>` or `-i <path>`	Path to the dataset file or directory. Supported formats: CSV/TSV (optionally gzip-compressed), Parquet. Only `id` and `embedding` columns are imported.	Required

`--format <format>`	Source files format. One of `csv`, `tsv`, `parquet`. When set, only files matching the specified format are imported from a directory. When not set, format is auto-detected from file extensions.	
`--embedding-column-name <name>`	Alternative source column name for the embedding field in input files.	`embedding`
`--table <name>`	Name of the table to load data into.	`vector_index_workload`
`--index <name>`	Name of the vector index to build after import.	`index`
`--index-type <type>`	Index type to build after import. Possible values: `None`, `KmeansTree`. Set to `None` to skip index building.	`KmeansTree`
`--vector-type <type>`	Type of vectors. One of `float`, `int8`, `uint8`, `bit`.	`float`
`--vector-dimension <value>`	Vector dimension (size of embedding vectors).	`1024`
`--distance <value>`	Distance or similarity function. One of `inner_product`, `cosine`, `euclidean`, `manhattan`.	`inner_product`
`--kmeans-tree-levels <value>`	Number of levels in the kmeans tree. If not set, auto-detected by server. See kmeans-tree type.	Auto-detected
`--kmeans-tree-clusters <value>`	Number of clusters in kmeans. If not set, auto-detected by server. See kmeans-tree type.	Auto-detected
`--kmeans-tree-covering <value>`	Build a covering index (`1` — enabled, `0` — disabled).	`0`
`--kmeans-tree-prefixed <value>`	Build a prefixed (filtered) index (`1` — enabled, `0` — disabled). The table must have been created with the `--prefixed` option.	`0`

Note

For more details on `--kmeans-tree-*` index building parameters, see kmeans-tree type.

Common parameters of the import command

Name	Description	Default value
`--upload-threads <value>` or `-t <value>`	The number of execution threads for data preparation.	The number of available cores on the client.
`--bulk-size <value>`	The size of the chunk for sending data, in rows.	10000
`--max-in-flight <value>`	The maximum number of data chunks that can be processed simultaneously.	128
`--file-output-path <value>` or `-f <path>`	If this option is set, the data will not be loaded into the database, but will be saved to the directory .	

Generating synthetic data

Generate random vectors and load them into the table. Vector components are sampled from a uniform distribution and serialized into YDB binary embedding format.

```
ydb workload vector import generator
```

Available options

Name	Description	Default value
`--rows <value>`	Number of rows to generate.	`10000`
`--prefix-count <value>`	Number of distinct prefix values for a prefixed index.	`100`
`--seed <value>`	Seed for the random number generator.	`42`
`--table <name>`	Name of the table to load data into.	`vector_index_workload`
`--index <name>`	Name of the vector index to build after import.	`index`
`--index-type <type>`	Index type to build after import. Possible values: `None`, `KmeansTree`.	`KmeansTree`
`--vector-type <type>`	Type of vectors. One of `float`, `int8`, `uint8`, `bit`.	`float`
`--vector-dimension <value>`	Vector dimension.	`1024`
`--distance <value>`	Distance or similarity function.	`inner_product`
`--kmeans-tree-levels <value>`	Number of levels in the kmeans tree. If not set, auto-detected by server.	Auto-detected
`--kmeans-tree-clusters <value>`	Number of clusters in kmeans. If not set, auto-detected by server.	Auto-detected

`--kmeans-tree-covering <value>`	Build a covering index.	`0`
`--kmeans-tree-prefixed <value>`	Build a prefixed (filtered) index.	`0`

Note

For more details on `--kmeans-tree-*` index building parameters, see kmeans-tree type.

Common parameters of the import command

Name	Description	Default value
`--upload-threads <value>` or `-t <value>`	The number of execution threads for data preparation.	The number of available cores on the client.
`--bulk-size <value>`	The size of the chunk for sending data, in rows.	10000
`--max-in-flight <value>`	The maximum number of data chunks that can be processed simultaneously.	128
`--file-output-path <value>` or `-f <path>`	If this option is set, the data will not be loaded into the database, but will be saved to the directory .	

Building a vector index

If the table was loaded with `--index-type None`, or if you want to build an additional index with different parameters, you can build a vector index on an existing table using the `build-index` command.

```
ydb workload vector build-index --distance cosine
```

Available options

Name	Description	Default value
`--table <name>`	Name of the table to build the index on.	`vector_index_workload`
`--index <name>`	Name of the index to create.	`index`
`--vector-type <type>`	Type of vectors. One of `float`, `int8`, `uint8`, `bit`.	`float`
`--vector-dimension <value>`	Vector dimension.	`1024`
`--distance <value>`	Distance or similarity function.	`inner_product`
`--kmeans-tree-levels <value>`	Number of levels in the kmeans tree. If not set, auto-detected by server.	Auto-detected
`--kmeans-tree-clusters <value>`	Number of clusters in kmeans. If not set, auto-detected by server.	Auto-detected
`--dry-run`	Print the DDL query instead of executing it.	

Note

For more details on `--kmeans-tree-*` index building parameters, see kmeans-tree type.

Dropping a vector index

Drop a previously built vector index.

```
ydb workload vector drop-index
```

Available options

Name	Description	Default value
`--table <name>`	Name of the table that holds the index.	`vector_index_workload`
`--index <name>`	Name of the index to drop.	`index`
`--dry-run`	Print the DDL query instead of executing it.	

Running the workload

Run load testing using one of two modes: `select` (vector search queries) or `upsert` (inserting new vector rows).

Search workload

Executes vector search queries against the indexed table. Optionally measures recall by comparing approximate search results with exact (full-scan) results, then runs a performance benchmark.

```
ydb workload vector run select --recall
```

Available options

Name	Description	Default value
`--table <name>`	Name of the main table with vectors.	`vector_index_workload`
`--index <name>`	Name of the vector index to use.	`index`
`--query-table <name>`	Name of the table with predefined search vectors. If not specified, random rows from the main table are used.	
`--targets <value>`	Number of vectors to use as test targets.	`100`
`--limit <value>`	Maximum number of nearest vectors to return per query.	`5`
`--kmeans-tree-clusters <value>`	Maximum number of clusters to inspect during search (KMeansTreeSearchTopSize).	`1`
`--recall`	Measure recall of the approximate search compared to exact search.	
`--recall-threads <value>`	Number of concurrent queries during recall measurement.	`10`
`--non-indexed`	Take vector settings from the index, but search without the index (full scan).	
`--stale-ro`	Read with StaleRO consistency mode.	

Warning

Pay attention to the `--kmeans-tree-clusters` parameter — increasing it significantly improves search recall at the expense of speed. Try values from 1 to the number of clusters specified when creating the index.

Common parameters for all load types

Name	Description	Default value
`--dry-run`	Do not execute initialization queries, but only display their text.	
`--check-canonical` or `-c`	Use special version of queries (they have deterministic answers) and compare results with canonical ones.	
`--output <value>`	The name of the file where the query execution results will be saved.	`results.out`
`--iterations <value>`	The number of times each load query will be executed.	`1`
`--json <name>`	The name of the file where query execution statistics will be saved in `json` format.	Not saved by default
`--ministat <name>`	The name of the file where query execution statistics will be saved in `ministat` format.	Not saved by default
`--csv <name>`	The name of the file to save the CSV version of the result table.	Not saved by default
`--plan <name>`	The name of the file to save the query plan. Files like `<name>.<query number>.explain` and `<name>.<query number>.<iteration number>` will be saved in formats: `ast`, `json`, `svg`, and `table`.	Not saved by default
`--query-prefix <setting>`	Query prefix. Every prefix is a line that will be added to the beginning of each query. For multiple prefix lines use this option several times.	Not specified by default
`--retries`	Max retry count for every request.	`0`
`--include`	Names, numbers or ranges of query numbers to be executed as part of the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	All queries executed
`--exclude`	Names, numbers or ranges of query numbers to be excluded from the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	None excluded by default
`--verbose` or `-v`	Print additional information to the screen during query execution.	
`--global-timeout <value>`	Global timeout for all queries. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--request-timeout <value>`	Timeout for each iteration of each query. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--threads <value>` or `-t <value>`	The number of parallel threads generating the load. Zero means that queries will be executed in the main thread; otherwise, queries will be mixed.	`0`
`--stats <value>`	Extended execution statistics collection mode. Available values: `full`, `profile`.	`full`

Upsert workload

Continuously inserts new vector rows into the table, generating random embeddings on the fly.

```
ydb workload vector run upsert
```

Available options

Name	Description	Default value
`--table <name>`	Name of the table to insert into.	`vector_index_workload`
`--index <name>`	Name of the vector index.	`index`
`--bulk-size <value>`	Number of rows per upsert batch.	`100`
`--prefixed`	Generate upserts with a `prefix` column (for prefixed indexes).	
`--prefix-count <value>`	Number of distinct prefix values. Used only when `--prefixed` is set.	`1000`

Common parameters for all load types

Name	Description	Default value
`--dry-run`	Do not execute initialization queries, but only display their text.	
`--check-canonical` or `-c`	Use special version of queries (they have deterministic answers) and compare results with canonical ones.	
`--output <value>`	The name of the file where the query execution results will be saved.	`results.out`
`--iterations <value>`	The number of times each load query will be executed.	`1`
`--json <name>`	The name of the file where query execution statistics will be saved in `json` format.	Not saved by default
`--ministat <name>`	The name of the file where query execution statistics will be saved in `ministat` format.	Not saved by default
`--csv <name>`	The name of the file to save the CSV version of the result table.	Not saved by default
`--plan <name>`	The name of the file to save the query plan. Files like `<name>.<query number>.explain` and `<name>.<query number>.<iteration number>` will be saved in formats: `ast`, `json`, `svg`, and `table`.	Not saved by default
`--query-prefix <setting>`	Query prefix. Every prefix is a line that will be added to the beginning of each query. For multiple prefix lines use this option several times.	Not specified by default
`--retries`	Max retry count for every request.	`0`
`--include`	Names, numbers or ranges of query numbers to be executed as part of the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	All queries executed
`--exclude`	Names, numbers or ranges of query numbers to be excluded from the load. Specified as a comma-separated list, e.g.: `1,2,4-6`.	None excluded by default
`--verbose` or `-v`	Print additional information to the screen during query execution.	
`--global-timeout <value>`	Global timeout for all queries. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--request-timeout <value>`	Timeout for each iteration of each query. Supports time units (e.g., '5s', '1m'). Plain number interpreted as milliseconds.	Not specified by default. The time is unlimited.
`--threads <value>` or `-t <value>`	The number of parallel threads generating the load. Zero means that queries will be executed in the main thread; otherwise, queries will be mixed.	`0`
`--stats <value>`	Extended execution statistics collection mode. Available values: `full`, `profile`.	`full`

Cleaning up

Drop tables created for load testing:

```
ydb workload vector clean
```

Test algorithm

The `run select` mode performs the following stages:

1. A test set is generated:
   - If a table with a test set is specified via `--query-table`, the first `--targets` rows are selected from it in primary key order.
   - Otherwise, `--targets` random rows are selected from the main table.
2. Recall measurement is performed (if `--recall` is specified):

- Two queries are executed for each item in the test set.
- The first query performs an exact vector search (full scan) based on vector distance, forming the `R_exact` result set.
- The second query performs an approximate vector search using the vector index, forming the `R_approx` result set.
- If a filtered vector index is selected, both queries scan only the entries with matching filter column values.
- Recall of the approximate search is calculated using the formula (where `|A|` is the number of elements in the set A and `A ∩ B` is the intersection of sets A and B).

3. Performance measurement is performed:
   - Indexed search queries are run for a specified duration and with a specified number of parallel threads.
   - Each query is executed for a random entry from the test set.
   - The average number of requests per second (RPS) and response time percentiles are calculated.

Usage examples

Example with generated data

1. Initialize the workload table:

```
ydb workload vector init
```

2. Generate and load synthetic vectors:

```
ydb workload vector import generator --rows 100000 --distance cosine
```

3. Run the search workload:

```
ydb workload vector run select --recall
```

   Output example:

```
Recall: 0.8950
Window      Txs Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
1           100 100     0       0       5       8       12      15
2             98 98     0       0       5       9       13      16
...

Total       Txs Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
10          980 98.0    0       0       5       9       14      18
```

   Column descriptions:

   - `Window` — Sequential number of the time window (e.g., each second or fixed interval).
   - `Txs` — Number of transactions successfully completed in this window (or total across all windows in the bottom section).
   - `Txs/Sec` — Transaction rate per second for the given window (or average for total).
   - `Retries` — Number of automatic retries performed due to temporary errors (e.g., conflicts or throttling).
   - `Errors` — Number of unrecoverable errors encountered.
   - `p50(ms)` — Median (50th percentile) transaction latency in milliseconds.
   - `p95(ms)` — 95th percentile latency (milliseconds).
   - `p99(ms)` — 99th percentile latency (milliseconds).
   - `pMax(ms)` — Maximum observed latency within the window (or overall maximum for total).

4. Run the upsert workload:

```
ydb workload vector run upsert
```

5. Clean up:

```
ydb workload vector clean
```

Example with an external dataset

1. Prepare tables and load data. See the recipe Vector index with external dataset loading.

   Examples of creating vector indexes are available on the Vector Indexes documentation page.

2. Run the search workload:

```
ydb -e grpc://hostname:2135 -d /Root/testdb workload vector run select \
    --table wikipedia --index idx_vector \
    --query-table wikipedia_sample --recall
```

   Output example:

```
Recall: 0.8950
Window      Txs Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
1           100 100     0       0       5       8       12      15
2             98 98     0       0       5       9       13      16
...

Total       Txs Txs/Sec Retries Errors  p50(ms) p95(ms) p99(ms) pMax(ms)
10          980 98.0    0       0       5       9       14      18
```

Remarks

- The search workload generates SQL queries that select nearest rows by vector distance from table `--table` using index `--index`.
- You don't need to specify column names (embedding, filter, primary key columns) or the distance function — they are automatically extracted from the index definition.
- Random selection of test vectors from `--table` works only if its primary key is numeric.
- If `--query-table` is specified, it must have the same vector and filter column names as `--table`.
- Recall measurement (`--recall`) is performed as a separate stage before the main performance test and shows the average overlap ratio (from 0 to 1) of indexed search results with full-scan search results across all selected test vectors.

Preparing a test sample from a large table

> **Warning**
>
> This step is only required if you don't already have a test dataset and if the primary key of the main table isn't numeric, because auto-generation of the test set doesn't work for non-numeric keys.

In this case, you can prepare a test set by selecting random rows from the main table manually. Example query to create the table for test samples:

```
CREATE TABLE vector_index_sample (
    id Uint64 NOT NULL,
    prefix Uint64 NOT NULL,
    embedding String NOT NULL,
    PRIMARY KEY (id)
);
```

A query to fill it with approximately 1000 rows from a large table:

```
INSERT INTO vector_index_sample
SELECT id, prefix, embedding FROM large_table
WHERE RandomNumber(id) < 0xFFFFFFFFFFFFFFFF / <number_of_rows_in_table> * 1000;
```

> **Tip**
>
> You can find the approximate number of rows in the table in table statistics without running `SELECT COUNT(*)`.

Managing YDB configuration

> **Note**
>
> Before YDB CLI 2.20.0, the `ydb admin cluster config` commands had the following format: `ydb admin config`.

This section contains commands for managing the YDB cluster configuration.

- Apply the `dynconfig.yaml` configuration to the cluster:

```
ydb admin cluster config replace -f dynconfig.yaml
```

- Check if it is possible to apply the configuration dynconfig.yaml to the cluster (check all validators, the configuration version in the yaml file must be 1 higher than the cluster configuration version, the cluster name must match):

```
ydb admin cluster config replace -f dynconfig.yaml --dry-run
```

- Apply the `dynconfig.yaml` configuration to the cluster, ignoring version and cluster checks (the version and cluster values will be overwritten with correct values):

```
ydb admin cluster config replace -f dynconfig.yaml --force
```

- Fetch the main cluster configuration:

```
ydb admin cluster config fetch
```

- Generate all possible final configurations for `dynconfig.yaml`:

```
ydb admin cluster config resolve --all -f dynconfig.yaml
```

- Generate the final configuration for `dynconfig.yaml` with the `tenant=/Root/test` and `canary=true` labels:

```
ydb admin cluster config resolve -f dynconfig.yaml --label tenant=/Root/test --label canary=true
```

- Generate the final configuration for `dynconfig.yaml` for labels from node 100:

```
ydb admin cluster config resolve -f dynconfig.yaml --node-id 100
```

- Generate a dynamic configuration file, based on a static configuration on the cluster:

```
ydb admin cluster config genereate
```

- Initialize a directory with the configuration, using the path to the configuration file:

```
ydb admin node config init --config-dir <path_to_directory> --from-config <path_to_configuration_file>
```

- Initialize a directory with the configuration, using the configuration from the cluster:

```
ydb admin node config init --config-dir <path_to_directory> --seed-node <cluster_node_endpoint>
```

Managing temporary configuration

This section contains commands for managing temporary configurations.

- Fetch all temporary configurations from the cluster:

```
ydb admin volatile-config fetch --all --output-directory <dir>
```

- Fetch the temporary configuration with id 1 from the cluster:

```
ydb admin volatile-config fetch --id 1
```

- Apply the `volatile.yaml` temporary configuration to the cluster:

```
ydb admin volatile-config add -f volatile.yaml
```

- Delete temporary configurations with ids 1 and 3 on the cluster:

```
ydb admin volatile-config drop --id 1 --id 3
```

- Delete all temporary configurations on the cluster:

```
ydb admin volatile-config drop --all
```

Parameters

- `-f, --filename <filename.yaml>` — read input from a file, `-` for STDIN. For commands that accept multiple files (e.g., resolve), you can specify it multiple times, the file type will be determined by the metadata field
- `--output-directory <dir>` — dump/resolve files to a directory
- `--strip-metadata` — remove the metadata field from the output
- `--all` — extends the output of commands to the entire configuration (see advanced configuration)
- `--allow-unknown-fields` — allows ignoring unknown fields in the configuration

Scenarios

Update the main cluster configuration

```
# Fetch the cluster configuration
ydb admin cluster config fetch > dynconfig.yaml
# Edit the configuration with your favorite editor
vim dynconfig.yaml
# Apply the configuration dynconfig.yaml to the cluster
ydb admin cluster config replace -f dynconfig.yaml
```

Similarly, in one line:

```
ydb admin cluster config fetch | yq '.config.actor_system_config.scheduler.resolution = 128' | ydb admin clus
ter config replace -f -
```

Command output:

```
OK
```

View the configuration for a specific set of labels

```
ydb admin cluster config resolve --remote --label tenant=/Root/db1 --label canary=true
```

Command output:

```
---
label_sets:
- dynamic:
    type: COMMON
    value: true
config:
  actor_system_config:
    use_auto_config: true
    node_type: COMPUTE
    cpu_count: 4
```

View the configuration for a specific node

```
ydb admin cluster config resolve --remote --node-id <node_id>
```

Command output:

```
---
label_sets:
- dynamic:
    type: COMMON
    value: true
config:
  actor_system_config:
    use_auto_config: true
    node_type: COMPUTE
    cpu_count: 4
```

Save all configurations locally

```
ydb admin cluster config fetch --all --output-directory <configs_dir>
ls <configs_dir>
```

Command output:

```
dynconfig.yaml volatile_1.yaml volatile_3.yaml
```

View all configurations locally

```
ydb admin cluster config fetch --all
```

Command output:

```
---
metadata:
  kind: main
  cluster: unknown
  version: 1
config:
  actor_system_config:
    use_auto_config: true
    node_type: COMPUTE
    cpu_count: 4
allowed_labels: {}
selector_config: []
---
metadata:
  kind: volatile
  cluster: unknown
  version: 1
  id: 1
# some comment example
selectors:
- description: test
  selector:
    tenant: /Root/db1
  config:
    actor_system_config: !inherit
      use_auto_config: true
      cpu_count: 12
```

View the final configuration for a specific node from the locally saved original configuration

```
ydb admin cluster config resolve -k <configs_dir> --node-id <node_id>
```

Command output:

```
---
label_sets:
- dynamic:
    type: COMMON
    value: true
config:
  actor_system_config:
    use_auto_config: true
    node_type: COMPUTE
    cpu_count: 4
```

Configuration Management

The YDB CLI provides commands for managing the dynamic configuration at different levels of the system.

General command syntax:

```
ydb [global options...] admin [scope] config [subcommands...]
```

- `global options` — Global parameters.
- `scope` — Configuration scope (`cluster`, `node`).
- `subcommands` — Subcommands for managing configuration.

View the command description:

```
ydb admin --help
```

Available Configuration Scopes

Cluster Configuration

Managing cluster-level configuration:

```
ydb admin cluster config [subcommands...]
```

Available subcommands:

- fetch - Fetches the current dynamic cluster configuration.
- generate - Generates dynamic configuration based on the static configuration on the cluster.
- replace - Replaces the dynamic configuration.
- version - Show configuration version on nodes (V1/V2).

Node Configuration

Managing node-level configuration:

```
ydb admin node config [subcommands...]
```

Available subcommands:

- init - Initializes the directory for node configuration.

Cluster Configuration Management Commands

Cluster configuration management commands are designed for working with the configuration at the level of the entire YDB cluster. These commands allow cluster administrators to view, modify, and manage settings that apply to all cluster nodes.

> **Alert**
>
> Commands in this section can harm your cluster if used incorrectly. Due to the potentially dangerous nature of these commands, ALL global parameters must be specified explicitly. Profiles are disabled by default and are only used when explicitly specified (--profile). Some commands do not require global options that are otherwise mandatory.

General syntax for calling cluster configuration management commands:

```
ydb [global options] admin cluster config [command options] <subcommand>
```

Where:

- `ydb` – The command to run the YDB CLI from the operating system command line.
- `[global options]` – Global options, common to all YDB CLI commands.
- `admin cluster config` – The command for managing cluster configuration.
- `[command options]` – Command options specific to each command and subcommand.
- `<subcommand>` – The subcommand.

Commands

The following is a list of available subcommands for managing cluster configuration. Any command can be called from the command line with the `--help` option to get help for it.

Command / Subcommand	Brief Description
admin cluster config fetch	Fetch the current dynamic configuration (aliases: `get`, `dump`)
admin cluster config generate	Generate dynamic configuration from the static startup configuration
admin cluster config replace	Replace the dynamic configuration
admin cluster config vesion	Show configuration version on nodes (V1/V2)

admin cluster config generate

With the `admin cluster config generate` command, you can generate a dynamic configuration file based on the static configuration file on the YDB cluster.
The dynamic configuration uses the format of an extended static configuration; the command automates the conversion process.

General command syntax:

```
ydb [global options...] admin cluster config generate
```

- `global options` — Global parameters.

View the description of the dynamic configuration generation command:

```
ydb admin cluster config generate --help
```

Examples

Generate the dynamic configuration based on the static configuration:

```
ydb admin cluster config generate > config.yaml
```

After executing this command, the `config.yaml` file will contain a YAML document in the following format:

```
metadata:
  kind: MainConfig
  cluster: ""
  version: 0
config:
  <static cluster configuration>
```

Using the Generated Dynamic Configuration

After generating the dynamic configuration, you can perform the following steps:

1. Add configuration parameters to the dynamic configuration file.
2. Apply the dynamic configuration to the cluster using the `admin cluster config replace` command.

admin cluster config fetch

With the `admin cluster config fetch` command, you can retrieve the current dynamic configuration of the YDB cluster.

General command syntax:

```
ydb [global options...] admin cluster config fetch
```

- `global options` — Global parameters.

View the description of the dynamic configuration fetch command:

```
ydb admin cluster config fetch --help
```

Examples

Fetch the current dynamic configuration of the cluster:

```
ydb --endpoint grpc://localhost:2135 admin cluster config fetch > config.yaml
```

admin cluster config replace

With the `admin cluster config replace` command, you can upload a dynamic configuration to the YDB cluster.

> **Alert**
>
> Commands in this section can harm your cluster if used incorrectly. Due to the potentially dangerous nature of these commands, ALL global parameters must be specified explicitly. Profiles are disabled by default and are only used when explicitly specified (--profile). Some commands do not require global options that are otherwise mandatory.

General command syntax:

```
ydb [global options...] admin cluster config replace [options...]
```

- `global options` — Global parameters.
- `options` — Subcommand parameters.

View the description of the dynamic configuration replacement command:

```
ydb admin cluster config replace --help
```

Subcommand Parameters

Name	Description
`-f`, `--filename`	Path to the file containing the configuration.
`--allow-unknown-fields`	Allow unknown fields in the configuration. If the flag is not set, unknown fields in the configuration result in an error.
`--ignore-local-validation`	Ignore basic client-side configuration validation. If the flag is not set, YDB CLI performs basic client-side configuration validation.

Examples

Upload the dynamic configuration file to the cluster:

```
ydb admin cluster config replace --filename config.yaml
```

Upload the dynamic configuration file to the cluster, ignoring local applicability checks:

```
ydb admin cluster config replace -f config.yaml --ignore-local-validation
```

Upload the dynamic configuration file to the cluster, ignoring the check for unknown fields:

```
ydb admin cluster config replace -f config.yaml --allow-unknown-fields
```

Node Configuration Management Commands

Node configuration management commands are designed for working with the configuration at the level of individual YDB cluster nodes. These commands allow cluster administrators to initialize, update, and manage the settings of individual nodes.

> **Alert**
>
> Commands in this section can harm your cluster if used incorrectly. Due to the potentially dangerous nature of these commands, ALL global parameters must be specified explicitly. Profiles are disabled by default and are only used when explicitly specified (--profile). Some commands do not require global options that are otherwise mandatory.

General syntax for calling node configuration management commands:

```
ydb [global options] admin node config [command options] <subcommand>
```

- `ydb` – The command to run the YDB CLI from the operating system command line.
- `[global options]` – Global options, common to all YDB CLI commands.
- `admin node config` – The command for managing node configuration.
- `[command options]` – Command options specific to each command and subcommand.
- `<subcommand>` – The subcommand.

Commands

The following is a list of available subcommands for managing node configuration. Any command can be called from the command line with the `--help` option to get help for it.

Command / Subcommand	Brief Description
admin node config init	Initialize the directory for node configuration

admin node config init

When deploying a new YDB cluster or adding nodes to an existing one (scaling out), each node requires a directory to store its configuration. The `admin node config init` command creates and prepares this directory by placing a specified configuration file in it or retrieving the configuration from another cluster node (seed node).

General command syntax:

```
ydb [global options...] admin node config init [options...]
```

- `global options` — Global parameters.
- `options` — Subcommand parameters.

View the description of the node configuration initialization command:

```
ydb admin node config init --help
```

Subcommand Parameters

Name	Description
`-d`, `--config-dir`	**Required**. Path to the directory for storing the configuration file.
`-f`, `--from-config`	Path to the initial configuration file. Required for initial cluster deployment. Can also be used for scaling out if a file with the current cluster configuration has been delivered to the node beforehand.
`-s`, `--seed-node`	Endpoint of the source node (seed node) from which the configuration will be retrieved. Used for scaling out the cluster.

Examples

Initialize the node's configuration directory using the specified configuration file:

```
ydb admin node config init --config-dir /opt/ydb/cfg-dir --from-config config.yaml
```

Initialize the node's configuration by retrieving it from a source node:

```
ydb admin node config init --config-dir /opt/ydb/cfg-dir --seed-node <node.ydb.tech>:2135
```

Usage

After successfully initializing the node's configuration directory, you can start the `ydbd` process on this node by adding the `--config-dir` parameter specifying the path to the directory. From this point on, when the cluster configuration is updated, the system automatically saves the updated config to the specified directory, eliminating the need to manually update the configuration file on the node.

When the node restarts, it automatically loads the current configuration from this directory.

Bridge cluster management commands

Feature of Yandex Enterprise Database

This functionality is available only in the Yandex Enterprise Database. In the open-source version of YDB it is absent.

Commands for managing the cluster in bridge mode let you view pile state, perform planned and emergency PRIMARY change, temporarily take a pile out for maintenance, and return it to the cluster.

Alert

Commands in this section can harm your cluster if used incorrectly. Due to the potentially dangerous nature of these commands, **ALL** global parameters must be specified explicitly. Profiles are disabled by default and are only used when explicitly specified (`--profile <profile-name>`). Some commands do not require global options that are otherwise mandatory.

General syntax for bridge cluster management commands:

```
ydb [global options...] admin cluster bridge [command options...] <subcommand>
```

where:

- `ydb` — command to launch YDB CLI from the operating system command line;
- `[global options]` — global parameters common to all YDB CLI commands;
- `admin cluster bridge` — cluster configuration management command;
- `[command options]` — parameters specific to each command and subcommand;
- `<subcommand>` — subcommand.

Commands

Below is the list of available subcommands for bridge cluster management. You can run any command with the `--help` option for help.

Command / subcommand	Brief description
admin cluster bridge list	List pile state
admin cluster bridge switchover	Planned PRIMARY change
admin cluster bridge failover	Emergency failover
admin cluster bridge takedown	Take pile out of cluster
admin cluster bridge rejoin	Return pile to cluster

admin cluster bridge list

> **Feature of Yandex Enterprise Database**
>
> This functionality is available only in the Yandex Enterprise Database. In the open-source version of YDB it is absent.

Use the `admin cluster bridge list` command to list the state of each pile in bridge mode.

General command syntax:

```
ydb [global options...] admin cluster bridge list [options...]
```

- `global options` — global parameters for the CLI.
- `options` — subcommand parameters.

View command help:

```
ydb admin cluster bridge list --help
```

Subcommand parameters

Name	Description
`--format <pretty, json, csv>`	Output format. Valid values: `pretty`, `json`, `csv`. Default: `pretty`.

Examples

List piles in human-readable format:

```
ydb admin cluster bridge list

pile-a: PRIMARY
pile-b: SYNCHRONIZED
```

Output state in JSON format:

```
ydb admin cluster bridge list --format json

{
  "pile-a": "PRIMARY",
  "pile-b": "SYNCHRONIZED"
}
```

Output state in CSV format:

```
ydb admin cluster bridge list --format csv

pile,state
pile-a,PRIMARY
pile-b,SYNCHRONIZED
```

admin cluster bridge switchover

> **Feature of Yandex Enterprise Database**
>
> This functionality is available only in the Yandex Enterprise Database. In the open-source version of YDB it is absent.

Use the `admin cluster bridge switchover` command to perform a smooth, planned transition of the specified pile to `PRIMARY` via the intermediate `PROMOTED` state. For details, see the scenario description.

> **Alert**
>
> Commands in this section can harm your cluster if used incorrectly. Due to the potentially dangerous nature of these commands, **ALL** global parameters must be specified explicitly. Profiles are disabled by default and are only used when explicitly specified (`--profile <profile-name>`). Some commands do not require global options that are otherwise mandatory.

General command syntax:

```
ydb [global options...] admin cluster bridge switchover [options...]
```

- `global options` — global parameters for the CLI.
- `options` — subcommand parameters.

View command help:

```
ydb admin cluster bridge switchover --help
```

Subcommand parameters

Name	Description
`--new-primary <pile>`	Name of the pile that should become the new PRIMARY.

Requirements

- The target pile must be in the `SYNCHRONIZED` state.

Examples

Transition pile `pile-b` from `SYNCHRONIZED` to `PRIMARY` via the intermediate `PROMOTED` state:

```
ydb admin cluster bridge switchover --new-primary pile-b
```

Verifying the result

After a short time (a few minutes), verify that pile states have changed correctly using the list command:

```
ydb admin cluster bridge list

pile-a: SYNCHRONIZED
pile-b: PRIMARY
```

admin cluster bridge failover

> **Feature of Yandex Enterprise Database**
>
> This functionality is available only in the Yandex Enterprise Database. In the open-source version of YDB it is absent.

Use the `admin cluster bridge failover` command to perform emergency disable of a pile when it is unavailable. You can optionally specify the pile that will become the new `PRIMARY`.

> **Alert**
>
> Commands in this section can harm your cluster if used incorrectly. Due to the potentially dangerous nature of these commands, **ALL** global parameters must be specified explicitly. Profiles are disabled by default and are only used when explicitly specified (`--profile <profile-name>`). Some commands do not require global options that are otherwise mandatory.

General command syntax:

```
ydb [global options...] admin cluster bridge failover [options...]
```

- `global options` — global parameters for the CLI.
- `options` — subcommand parameters.

View command help:

```
ydb admin cluster bridge failover --help
```

Subcommand parameters

Name	Description
`--pile <pile>`	Name of the unavailable pile.
`--new-primary <pile>`	Name of the pile that should become the new `PRIMARY`. Specify if the unavailable pile was `PRIMARY`.

Requirements

- If the current `PRIMARY` is unavailable, you must specify `--new-primary` and choose a pile in the `SYNCHRONIZED` state. If `--new-primary` is missing or the chosen pile is not in `SYNCHRONIZED` state, the command returns an error without making any changes.
- The cluster will not enter an invalid state: if requirements are violated, the command makes no changes and reports an error.
- If the pile has not failed but you need to disable it, use planned disable — the `takedown` command.
- The command requires cluster administrator privileges. If mandatory authentication is enabled, login and password authentication cannot be used while the cluster is in an emergency state — use client certificate authentication instead. For details, see Authentication specifics when the cluster is in an emergency state.

Examples

Perform emergency disable for an unavailable pile named `pile-a`:

```
ydb admin cluster bridge failover --pile pile-a
```

Perform emergency disable for an unavailable `PRIMARY` pile and designate a synchronized pile as the new `PRIMARY`:

```
ydb admin cluster bridge failover --pile pile-a --new-primary pile-b
```

Verifying the result

Use the list command to verify that the unavailable pile has been moved to `DISCONNECTED` and (if `--new-primary` was specified) a new `PRIMARY` has been selected:

```
ydb admin cluster bridge list

pile-a: DISCONNECTED
pile-b: PRIMARY
```

admin cluster bridge takedown

> **Feature of Yandex Enterprise Database**
>
> This functionality is available only in the Yandex Enterprise Database. In the open-source version of YDB it is absent.

Use the `admin cluster bridge takedown` command to perform planned disable of a pile. If you are disabling the current `PRIMARY`, you must specify the new `PRIMARY`.

> **Alert**
>
> Commands in this section can harm your cluster if used incorrectly. Due to the potentially dangerous nature of these commands, **ALL** global parameters must be specified explicitly. Profiles are disabled by default and are only used when explicitly specified (`--profile <profile-name>`). Some commands do not require global options that are otherwise mandatory.

General command syntax:

```
ydb [global options...] admin cluster bridge takedown [options...]
```

- `global options` — global parameters.
- `options` — subcommand parameters.

View command help:

```
ydb admin cluster bridge takedown --help
```

Subcommand parameters

Name	Description
`--pile <pile>`	Name of the pile to take out of the cluster.
`--new-primary <pile>`	Name of the pile that should become the new `PRIMARY` if the current `PRIMARY` is being disabled.

Requirements

- If you are disabling the current `PRIMARY`, you must specify `--new-primary` and choose a pile in the `SYNCHRONIZED` state.

Examples

Take `SYNCHRONIZED` pile `pile-b` out of the cluster:

```
ydb admin cluster bridge takedown --pile pile-b
```

Take `PRIMARY` pile `pile-a` out of the cluster and transition `pile-b` from `SYNCHRONIZED` to `PRIMARY`:

```
ydb admin cluster bridge takedown --pile pile-a --new-primary pile-b
```

Verifying the result

Verify the resulting pile states with the list command:

```
ydb admin cluster bridge list

pile-a: PRIMARY
pile-b: DISCONNECTED
```

If you disabled the current `PRIMARY` with `--new-primary`, verify that the chosen pile has become `PRIMARY`:

```
ydb admin cluster bridge list

pile-a: DISCONNECTED
pile-b: PRIMARY
```

admin cluster bridge rejoin

Feature of Yandex Enterprise Database

This functionality is available only in the Yandex Enterprise Database. In the open-source version of YDB it is absent.

Use the `admin cluster bridge rejoin` command to return the specified pile to the cluster after maintenance or recovery. After the command runs, the pile is expected to transition from `DISCONNECTED` to `NOT_SYNCHRONIZED`, then sync automatically and transition to `SYNCHRONIZED`.

Alert

Commands in this section can harm your cluster if used incorrectly. Due to the potentially dangerous nature of these commands, **ALL** global parameters must be specified explicitly. Profiles are disabled by default and are only used when explicitly specified (`--profile <profile-name>`). Some commands do not require global options that are otherwise mandatory.

General command syntax:

```
ydb [global options...] admin cluster bridge rejoin [options...]
```

- `global options` — global parameters.
- `options` — subcommand parameters.

View command help:

```
ydb admin cluster bridge rejoin --help
```

Subcommand parameters

Name	Description
`--pile <pile>`	Name of the pile to return to the cluster.

Requirements

- The pile must be in the `DISCONNECTED` state before it can be returned.

Examples

Return pile `pile-a` from `DISCONNECTED` state:

```
ydb admin cluster bridge rejoin --pile pile-a
```

Verifying the result

Right after the command runs, the pile is expected to transition to `NOT_SYNCHRONIZED`. Verify with the list command:

```
ydb admin cluster bridge list

pile-a: NOT_SYNCHRONIZED
pile-b: PRIMARY
```

After synchronization completes, the pile transitions to `SYNCHRONIZED`:

```
ydb admin cluster bridge list

pile-a: SYNCHRONIZED
pile-b: PRIMARY
```

Reference for YDB SDK

To work with YDB, OpenSource SDKs are available for the following programming languages:

Language	GitHub Repository	API Reference
Go	ydb-platform/ydb-go-sdk	https://pkg.go.dev/github.com/ydb-platform/ydb-go-sdk/v3
Java	ydb-platform/ydb-java-sdk	N/A
Python	ydb-platform/ydb-python-sdk	https://ydb-platform.github.io/ydb-python-sdk
C#	ydb-platform/ydb-dotnet-sdk	N/A
JavaScript	ydb-platform/ydb-js-sdk	https://ydb.js.org
Rust	ydb-platform/ydb-rs-sdk	N/A
C++	ydb-platform/ydb-cpp-sdk	N/A
PHP	ydb-platform/ydb-php-sdk	N/A

The SDK documentation contains the following sections:

- Installation
- Authentication
- Handling errors
- SDK feature comparison
- Parameterized queries
- Data formats
- Working with topics
- Working with coordination nodes
- Observability in YDB SDK

See also:

- Documentation for application developers
- Test applications
- Code recipes
- Generate code from YQL

Reference for YDB SDK

To work with YDB, OpenSource SDKs are available for the following programming languages:

Language	GitHub Repository	API Reference
Go	ydb-platform/ydb-go-sdk	https://pkg.go.dev/github.com/ydb-platform/ydb-go-sdk/v3
Java	ydb-platform/ydb-java-sdk	N/A
Python	ydb-platform/ydb-python-sdk	https://ydb-platform.github.io/ydb-python-sdk
C#	ydb-platform/ydb-dotnet-sdk	N/A
JavaScript	ydb-platform/ydb-js-sdk	https://ydb.js.org
Rust	ydb-platform/ydb-rs-sdk	N/A
C++	ydb-platform/ydb-cpp-sdk	N/A
PHP	ydb-platform/ydb-php-sdk	N/A

The SDK documentation contains the following sections:

- Installation
- Authentication
- Handling errors
- SDK feature comparison
- Parameterized queries
- Data formats
- Working with topics
- Working with coordination nodes
- Observability in YDB SDK

See also:

- Documentation for application developers
- Test applications
- Code recipes
- Generate code from YQL

Installing the YDB SDK

Follow the instructions below to quickly install the OpenSource SDK. Make sure to preinstall and configure tools for working with the selected programming language and package managers on your workstation.

The build process using the source code is described in the source code repositories on GitHub. Follow the links given on the YDB SDK - Overview page.

Go

Run the command from the command line:

```
go get -u github.com/ydb-platform/ydb-go-sdk/v3
```

To ensure that the installation is successful, make sure that your environment is running Go 1.17 or higher.

Java

Add dependencies to the Maven project as described in the "Install the SDK" step of the `readme.md` file in the source code repository.

Python

Run the command from the command line:

```
python3 -m pip install ydb
```

If the command fails, make sure your environment has Python 3.8 or newer installed with the pip package manager enabled.

C# (.NET)

```
dotnet add package Ydb.Sdk
```

JavaScript

```
npm install @ydbjs/core @ydbjs/query
```

The minimum supported Node.js® version is 20.19.

Rust

Add to Cargo.toml last version of ydb crate.

C++

Debian packages (Ubuntu 24.04)

Pre-built `.deb` packages for Ubuntu 24.04 (Noble), amd64 are attached to each GitHub release. Download and install them locally:

```
# Replace <TAG> with a release tag from GitHub releases
TAG=<TAG>
BASE="https://github.com/ydb-platform/ydb-cpp-sdk/releases/download/${TAG}"

wget "${BASE}/yandex-googleapis-api-common-protos-1.0.0-Linux.deb"
wget "${BASE}/libydb-cpp-dev_${TAG#v}_amd64.deb"
# Optional plugins:
wget "${BASE}/libydb-cpp-iam-dev_${TAG#v}_amd64.deb"
wget "${BASE}/libydb-cpp-otel-metrics-dev_${TAG#v}_amd64.deb"
wget "${BASE}/libydb-cpp-otel-tracing-dev_${TAG#v}_amd64.deb"

sudo apt-get update
sudo apt-get install -y \
    ./yandex-googleapis-api-common-protos-*.deb \
    ./libydb-cpp-dev_*.deb ./libydb-cpp-iam-dev_*.deb \
    ./libydb-cpp-otel-metrics-dev_*.deb ./libydb-cpp-otel-tracing-dev_*.deb
```

Available packages:

- `yandex-googleapis-api-common-protos` — required protobuf dependency;
- `libydb-cpp-dev` — core SDK: static library, public headers, and CMake package files;
- `libydb-cpp-iam-dev` — IAM credentials plugin (optional);
- `libydb-cpp-otel-metrics-dev` — OpenTelemetry metrics plugin (optional);
- `libydb-cpp-otel-tracing-dev` — OpenTelemetry tracing plugin (optional; requires `libydb-cpp-otel-metrics-dev` for OTel headers and libraries).

Note

- Supported platform: Ubuntu 24.04 (Noble), amd64.
- The `.deb` packages do not declare apt dependencies on each other; install `yandex-googleapis-api-common-protos` together with the core SDK package.
- For other platforms, use the build-from-source instructions below.

Using the SDK in CMake

After installing the packages, use the SDK in your CMake project:

```
find_package(ydb-cpp-sdk REQUIRED COMPONENTS Driver Table Topic)
target_link_libraries(myapp PRIVATE YDB-CPP-SDK::Driver YDB-CPP-SDK::Table)
```

Pass `-DCMAKE_PREFIX_PATH=/usr/share/yandex` when configuring your project, since the packages install under the Yandex prefix.

Build from source

Clone the ydb-cpp-sdk repository and run the following commands from the command line:

```
cmake --preset release-${compiler} -D YDB_SDK_INSTALL=On
cmake --build --preset default
cmake --install build --prefix ${ydb_install_dir}
```

- `compiler` — your compiler (`clang` or `gcc`);
- `ydb_install_dir` — the path where you want to install the SDK.

Make sure all dependencies are installed before running the commands. See the README in the source repository for the full list of build dependencies and instructions.

PHP

```
composer require ydb-platform/ydb-php-sdk
```

Authentication in the SDK

As we discussed in the YDB server connection article, the client must add an authentication token to each request. The authentication token is checked by the server. If the authentication is successful, the request is authorized and executed. Otherwise, the `Unauthenticated` error returns.

The YDB SDK uses an object that is responsible for generating these tokens. SDK provides built-in methods for getting such an object:

1. The methods that pass parameters explicitly, with each method implementing a certain authentication mode.
2. The method that determines the authentication mode and relevant parameters based on environmental variables.

Usually, you create a token generation object before you initialize the YDB driver, and you pass the object to the driver constructor as a parameter. The C++ and Go SDKs additionally let you work with multiple databases and token generation objects through a single driver.

If the token generation object is not defined, the driver won't add any authentication data to the requests. This approach enables you to successfully connect to locally deployed YDB clusters without enabling mandatory authentication. If you enable mandatory authentication, database requests without an authentication token will be rejected with an authentication error.

Methods for creating token generation objects

You can click any of the methods below to go to the source code of an example in the repository. You can also learn about the authentication code recipes.

Python

Mode	Method
Anonymous	ydb.AnonymousCredentials()
Access Token	ydb.AccessTokenCredentials(token)
Metadata	ydb.iam.MetadataUrlCredentials()
Service Account Key	ydb.iam.ServiceAccountCredentials.from_file(key_file, iam_endpoint=None, iam_channel_credentials=None)
Static Credentials	ydb.StaticCredentials.from_user_password(user, password)
OAuth 2.0 token exchange	ydb.oauth2_token_exchange.Oauth2TokenExchangeCredentials(), ydb.oauth2_token_exchange.Oauth2TokenExchangeCredentials.from_file(cfg_file, iam_endpoint=None)
Determined by environment variables	`ydb.credentials_from_env_variables()`

Go

Mode	Package	Method
Anonymous	ydb-go-sdk/v3	ydb.WithAnonymousCredentials()
Access Token	ydb-go-sdk/v3	ydb.WithAccessTokenCredentials(token)
Metadata	ydb-go-yc	yc.WithMetadataCredentials(ctx)
Service Account Key	ydb-go-yc	yc.WithServiceAccountKeyFileCredentials(key_file)
Static Credentials	ydb-go-sdk/v3	ydb.WithStaticCredentials(user, password)
OAuth 2.0 token exchange	ydb-go-sdk/v3	ydb.WithOauth2TokenExchangeCredentials(options...), ydb.WithOauth2TokenExchangeCredentialsFile(configFilePath)
Determined by environment variables	ydb-go-sdk-auth-environ	environ.WithEnvironCredentials(ctx)

Java

Mode	Method
Anonymous	tech.ydb.core.auth.NopAuthProvider.INSTANCE
Access Token	new tech.ydb.core.auth.TokenAuthProvider(accessToken);
Metadata	tech.ydb.auth.iam.CloudAuthHelper.getMetadataAuthProvider();
Service Account Key	tech.ydb.auth.iam.CloudAuthHelper.getServiceAccountFileAuthProvider(saKeyFile);
OAuth 2.0 token exchange	tech.ydb.auth.OAuth2TokenExchangeProvider.fromFile(cfgFile);
Determined by environment variables	tech.ydb.auth.iam.CloudAuthHelper.getAuthProviderFromEnviron();

JavaScript

Mode	Method
Anonymous	AnonymousCredentialsProvider()
Access Token	AccessTokenCredentialsProvider({ token })
Metadata	MetadataCredentialsProvider()

Service Account Key	ServiceAccountCredentialsProvider.fromFile(saKeyFile)
Static Credentials	StaticCredentialsProvider({ username, password }, endpoint)
Determined by environment variables	EnvironCredentialsProvider(connectionString)

Rust

Mode	Method
Anonymous	ydb::AnonymousCredentials or ydb::AccessTokenCredentials::from("")
Access Token	ydb::AccessTokenCredentials::from("token")
Metadata	ydb::MetadataUrlCredentials
Service Account Key	ydb::ServiceAccountCredentials
Static Credentials	ydb::StaticCredentialsAuth
Determined by environment variables	ydb::FromEnvCredentials
Execution of an external command	ydb::CommandLineCredentials (for example, for authentication using a Yandex.Cloud IAM token from the developer's computer `ydb::CommandLineCredentials.from_cmd("yc iam create-token")`)

PHP

Mode	Method
Anonymous	AnonymousAuthentication()
Access Token	AccessTokenAuthentication($accessToken)
Oauth Token	OAuthTokenAuthentication($oauthToken)
Metadata	MetadataAuthentication()
Service Account Key	JwtWithJsonAuthentication(key_id, $service_account_id, $privateKeyFile)
Determined by environment variables	EnvironCredentials()
Static Credentials	StaticAuthentication($user, $password)

Procedure for determining the authentication mode and parameters from the environment

The following algorithm that is the same for all SDKs applies:

1. If the value of the `YDB_SERVICE_ACCOUNT_KEY_FILE_CREDENTIALS` environment variable is set, the **System Account Key** authentication mode is used and the key is taken from the file whose name is specified in this variable.
2. Otherwise, if the value of the `YDB_ANONYMOUS_CREDENTIALS` environment variable is set to 1, the anonymous authentication mode is used.
3. Otherwise, if the value of the `YDB_METADATA_CREDENTIALS` environment variable is set to 1, the **Metadata** authentication mode is used.
4. Otherwise, if the value of the `YDB_ACCESS_TOKEN_CREDENTIALS` environment variable is set, the **Access token** authentication mode is used, where the this variable value is passed.
5. Otherwise, if the value of the `YDB_OAUTH2_KEY_FILE` environment variable is set, the **OAuth 2.0 token exchange** authentication mode is used, and the parameters are taken from the JSON file specified in this variable.
6. Otherwise, the **Metadata** authentication mode is used.

If the last step of the algorithm is selecting the **Metadata** mode, you can deploy a working application on VMs and in Yandex.Cloud Cloud Functions without setting any environment variables.

File format for OAuth 2.0 token exchange authentication mode parameters

Description of fields of JSON file with **OAuth 2.0 token exchange** authentication mode parameters. The set of fields depends on the original token type, `JWT` and `FIXED`.

In the table below, `creds_json` means a JSON with parameters for exchanging the original token for an access token.

Fields not described in this table are ignored.

Field	Type	Description	Default value/optionality
`grant-type`	string	Grant type	`urn:ietf:params:oauth:grant-type:token-exchange`
`res`	string | list of strings	Resource	optional
`aud`	string | list of strings	Audience option for token exchange request	optional
`scope`	string | list of strings	Scope	optional
`requested-token-type`	string	Requested token type	`urn:ietf:params:oauth:token-type:access_token`
`subject-credentials`	creds_json	Subject credentials	optional

`actor-credentials`	creds_json	Actor credentials	optional
`token-endpoint`	string	Token endpoint. In the case of YDB CLI, it is overridden by the `--iam-endpoint` option.	optional
Description of fields of `creds_json` (JWT)			
`type`	string	Token source type. Set `JWT`	
`alg`	string	Algorithm for JWT signature. Supported algorithms: ES256, ES384, ES512, HS256, HS384, HS512, PS256, PS384, PS512, RS256, RS384, RS512	
`private-key`	string	(Private) key in PEM format (for algorithms `ES*`, `PS*`, `RS*`) or Base64 format (for algorithms `HS*`) for JWT signature	
`kid`	string	`kid` JWT standard claim (key id)	optional
`iss`	string	`iss` JWT standard claim (issuer)	optional
`sub`	string	`sub` JWT standard claim (subject)	optional
`aud`	string	`aud` JWT standard claim (audience)	optional
`jti`	string	`jti` JWT standard claim (JWT id)	optional
`ttl`	string	JWT token TTL	`1h`
Description of fields of `creds_json` (FIXED)			
`type`	string	Token source type. Set `FIXED`	
`token`	string	Token value	
`token-type`	string	Token type value. It will become `subject_token_type/actor_token_type` parameter in token exchange request.	

Example

An example for JWT token exchange

```
{
  "subject-credentials": {
    "type": "JWT",
    "alg": "RS256",
    "private-key": "-----BEGIN RSA PRIVATE KEY-----\n...-----END RSA PRIVATE KEY-----\n",
    "kid": "my_key_id",
    "sub": "account_id"
  }
}
```

Peculiarities of YDB Python SDK v2 (deprecated version)

Warning

The behavior of the YDB Python SDK v2 (deprecated version) differs from the above-described version.

- The algorithm of the `construct_credentials_from_environ()` function from the YDB Python SDK v2:
  - If the value of the `USE_METADATA_CREDENTIALS` environment variable is set to 1, the **Metadata** authentication mode is used.
  - Otherwise, if the value of the `YDB_TOKEN` environment variable is set, the **Access Token** authentication mode is used, where this variable value is passed.
  - Otherwise, if the value of the `SA_KEY_FILE` environment variable is set, the **System Account Key** authentication mode is used and the key is taken from the file whose name is specified in this variable.
  - Or else, no authentication information is added to requests.
- If no object responsible for generating tokens is passed when initializing the driver, the general procedure for reading environment variables applies.

Parameterized queries

YDB supports and recommends the use of so-called parameterized queries. In such queries, data is transmitted separately from the query body itself, and special parameters in the SQL query are used to indicate where the data goes. Parameterized query plans are cached on the server (by default, up to 1000 entries), which lets you reuse plans for similar queries that differ only in parameter values.

In many YDB SDKs, parameterized query caching is enabled by default. You can disable it if needed by setting the `KeepInCache` parameter to `false`.

Request with data in the request body:

```
SELECT sa.title AS season_title, sr.title AS series_title
FROM seasons AS sa INNER JOIN series AS sr ON sa.series_id = sr.series_id
WHERE sa.series_id = 15 AND sa.season_id = 3
```

The corresponding parameterized query:

```
DECLARE $seriesId AS Uint64;
DECLARE $seasonId AS Uint64;

SELECT sa.title AS season_title, sr.title AS series_title
FROM seasons AS sa INNER JOIN series AS sr ON sa.series_id = sr.series_id
WHERE sa.series_id = $seriesId AND sa.season_id = $seasonId
```

Parameterized queries are written in the form of a template in which certain types of names are replaced with specific parameters each time the query is executed. Tokens starting with the sign `$` such as `$seriesId` and `$seasonId` in the query above are used to denote parameters.

Parameterized queries provide the following advantages:

- For repeated requests, the database server can cache the query plan for parameterized queries. This radically reduces CPU consumption and increases system throughput.
- The use of parameterized queries protects against vulnerabilities such as SQL Injection.
- Less code — only the query arguments change, not the query itself. You do not need to concatenate the query string manually.

See also

- Parameterized queries

Data formats

YDB supports several data representation formats and schema return modes for queries executed through QueryService. The format and schema return mode are set when the query runs and apply to every statement that returns data (for example, a SELECT statement or the `RETURNING` clause). How you configure these settings may differ across SDKs.

Data representation formats

The following sections describe the available data representation formats:

- Protobuf (Value) — the default format; rows are returned one by one with automatic conversion to native types in your programming language.
- Apache Arrow — a columnar format for analytics workloads and high-throughput transfer of large result sets.

Schema return modes

When a query runs as a stream, results arrive in parts. The schema return mode controls which parts of the stream include schema information.

- **Always (default)** — the data schema is included in every part of the result stream. Convenient when each part is processed independently.
- **First only** — the data schema is included only in the first part of the stream for each result set; later parts omit it. Reduces metadata volume when you stream large results.

The exact schema payload depends on the data format; each format's section describes what is returned.

Protobuf (Value) format

This is the default format. Data is returned row by row as Protobuf: each row is a serialized set of named values with their YQL types. The SDK converts values to native types for your language and exposes a type-safe API.

Use this format when you need:

- Transactional (OLTP) workloads with point reads of individual rows.
- Applications that process small fragments of a table at full row width.
- Native integration with your language's types.

YQL type mapping

Mapping from YQL types to native language types is implemented in the SDK and depends on the SDK you use. See your SDK's documentation for the exact mapping.

Returned schema

The schema is a list of columns with their YQL types. That is enough for the SDK to interpret values: given the column type, each value is converted to a native type.

SDK examples

Python

This format is the default when you run queries through QueryService. The example below shows how to set the result format explicitly.

```
pool = ydb.QuerySessionPool(driver)

query = """
    SELECT * FROM example ORDER BY Key LIMIT 100;
"""

result = pool.execute_with_retries(
    query,
    result_set_format=ydb.QueryResultSetFormat.VALUE,
    schema_inclusion_mode=ydb.QuerySchemaInclusionMode.FIRST_ONLY,
)

for result_set in result:
    print(f"Record batch with {len(result_set.rows)} rows and {len(result_set.columns)} columns")
```

Java

This format is the default when you run queries through QueryService.

Apache Arrow format

Results are returned in the columnar Apache Arrow format (IPC standard, version 5.0) without conversion inside the SDK, which makes large result sets efficient to process.

Prefer this format for:

- Analytical (OLAP) workloads with columnar processing — aggregations, filters, scans over multiple columns on large samples.
- Systems that integrate natively with Apache Arrow.
- Scenarios where throughput for large data transfers matters most.

YQL type mapping

YQL data types are mapped to Apache Arrow types as follows.

Numeric types

YQL type	Arrow type	Notes
`Bool`	`uint8`	`true` / `false` encoded as `1` / `0`
`Int8`	`int8`	
`Int16`	`int16`	
`Int32`	`int32`	
`Int64`	`int64`	
`Uint8`	`uint8`	
`Uint16`	`uint16`	
`Uint32`	`uint32`	
`Uint64`	`uint64`	
`Float`	`float`	
`Double`	`double`	
`Decimal(p, s)`	`fixed_size_binary(16)`	120 bits used; ±inf and NaN markers supported
`DyNumber`	`string`	String representation of the number

String types

YQL type	Arrow type	Notes
`String`	`binary`	
`Utf8`	`string`	
`Json`	`string`	
`JsonDocument`	`string`	String form of binary JSON
`Yson`	`binary`	
`Uuid`	`fixed_size_binary(16)`	16 bytes in mixed-endian order

Date and time types

YQL type	Arrow type	Notes
`Date`	`uint16`	Day precision
`Date32`	`int32`	Day precision
`Datetime`	`uint32`	Second precision
`Datetime64`	`int64`	Second precision
`Timestamp`	`uint64`	Microsecond precision
`Timestamp64`	`int64`	Microsecond precision
`Interval`	`int64`	Microseconds
`Interval64`	`int64`	Microseconds
`TzDate`	`struct<datetime: uint16, timezone: string>`	Time zone name as a string
`TzDate32`	`struct<datetime: int32, timezone: string>`	Time zone name as a string
`TzDatetime`	`struct<datetime: uint32, timezone: string>`	Time zone name as a string
`TzDatetime64`	`struct<datetime: int64, timezone: string>`	Time zone name as a string
`TzTimestamp`	`struct<datetime: uint64, timezone: string>`	Time zone name as a string

`TzTimestamp64`	`struct<datetime: int64, timezone: string>`	Time zone name as a string

Note

In Arrow, basic date/time types are represented as unsigned integers, while extended date/time types types use signed integers.

Container types

YQL type	Arrow type	Notes
`List<T>`	`list<item: T>`	
`Tuple<T1, T2, ...>`	`struct<field0: T1, field1: T2, ...>`	
`Struct<name: T, ...>`	`struct<name: T, ...>`	
`Dict<K, V>`	`list<struct<key: K, payload: V>>`	
`Set<T>`	`list<struct<key: T, payload: struct<>>>`	Same as `Dict<T, Void>`
`Variant<T1, ..., Tn>`	`dense_union<field0: T1, ...>`	For n <= 128
`Variant<T1, ..., Tn>`	`dense_union<dense_union<field0: T1, ...>, ...>`	For 128 < n <= 16384
`Variant<name1: T1, ..., nameN: Tn>`	`dense_union<name1: T1, ...>`	For n <= 128
`Variant<name1: T1, ..., nameN: Tn>`	`dense_union<dense_union<name1: T1, ...>, ...>`	For 128 < n <= 16384

Warning

`Variant` over tuple or struct cannot be represented in Apache Arrow if the number of child types exceeds 16384 (128 * 128).

Optional and special types

YQL type	Arrow type	Notes
`Null`	`null`	Singular type
`Void`	`struct<>`	Singular type
`EmptyList`	`struct<>`	Singular type
`EmptyDict`	`struct<>`	Singular type
`Tagged<T>`	`T`	Tag stripped
`Optional<T>`	`struct<opt: T>`	When `T` is `Variant`, `Optional`, `Pg`, or singular
`Optional<T>`	`T`	For all other `T`

Types in the `pg` family

All `pg` types are represented as Arrow `string` values containing the textual form.

Data compression

You can enable compression for Arrow payloads. Supported codecs:

Codec	Description
None (default)	
`ZSTD`	Zstandard. Compression level is configurable
`LZ4_FRAME`	LZ4. Compression level is not configurable

Returned schema

The schema has two parts:

- **Column list with YQL types** — describes the logical types in YQL terms so your application can interpret semantics regardless of Arrow representation details.
- **Serialized Arrow RecordBatch schema** — describes the binary layout in Apache Arrow terms. Required to deserialize RecordBatch on the client.

Both are needed because YQL and Arrow types do not always map one-to-one.

Apache Arrow examples in the SDK

Python

```
import pyarrow

pool = ydb.QuerySessionPool(driver)
```

```
query = """
    SELECT * FROM example ORDER BY Key LIMIT 100;
"""

format_settings = ydb.ArrowFormatSettings(
    compression_codec=ydb.ArrowCompressionCodec(ydb.ArrowCompressionCodecType.ZSTD, 10)
)

result = pool.execute_with_retries(
    query,
    result_set_format=ydb.QueryResultSetFormat.ARROW,
    schema_inclusion_mode=ydb.QuerySchemaInclusionMode.FIRST_ONLY,
    arrow_format_settings=format_settings,
)

for result_set in result:
    schema = pyarrow.ipc.read_schema(pyarrow.py_buffer(result_set.arrow_format_meta.schema))
    batch = pyarrow.ipc.read_record_batch(pyarrow.py_buffer(result_set.data), schema)
    print(f"Record batch with {batch.num_rows} rows and {batch.num_columns} columns")
```

Java

```
String query = "SELECT * FROM example ORDER BY Key LIMIT 100;";

ExecuteQuerySettings settings = ExecuteQuerySettings.newBuilder()
        .useApacheArrowFormat(ApacheArrowFormat.zstd())
        .build();

try (RootAllocator allocator = new RootAllocator()) {
    retryCtx.supplyResult(session -> session
            .createQuery(query, TxMode.SERIALIZABLE_RW, Params.empty(), settings)
            .execute(new ApacheArrowCompressedPartsHandler(allocator) {
                @Override
                public void onNextPart(QueryResultPart part) {
                    ResultSetReader rs = part.getResultSetReader();
                    while (rs.next()) {
                        String key = rs.getColumn("Key").getText();
                        System.out.println("Read row with key " + key);
                    }
                    System.out.printf("Record batch with %d rows and %d columns%n",
                            rs.getRowCount(), rs.getColumnCount());
                }
            })
    ).join().getStatus().expectSuccess("execute query problem");
}
```

Working with topics

This article provides examples of how to use the YDB SDK to work with topics.

Before performing the examples, create a topic and add a consumer.

Topic usage examples

C++

Reader example on GitHub

Go

Examples on GitHub

Java

Examples on GitHub

Python

Examples on GitHub

C#

Examples on GitHub

JavaScript

Examples on GitHub

Rust

Examples on GitHub (`topic-writer`, `topic-reader-retry`, `topic-read-in-transaction-example`).

Initializing a connection

Go

Use a YDB driver instance created with `ydb.Open`. The topic client is available via `db.Topic()`.

```
package main

import (
  "context"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/topic/topicoptions"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)

  // db.Topic() — client for topics
  writer, err := db.Topic().StartWriter("topic-path")
  if err != nil {
    panic(err)
  }

  reader, err := db.Topic().StartReader("consumer-name",
    topicoptions.ReadTopic("topic-path"),
  )
  if err != nil {
    panic(err)
  }
  _ = writer
  _ = reader
}
```

C++

To interact with YDB Topics, create an instance of the YDB driver and topic client.

The YDB driver lets the app and YDB interact at the transport layer. The driver must exist during the YDB access lifecycle and be initialized before creating a client.

Topic client (source code) requires the YDB driver for work. It handles topics and manages read and write sessions.

App code snippet for driver initialization:

```
auto driverConfig = NYdb::TDriverConfig()
    .SetEndpoint(opts.Endpoint)
    .SetDatabase(opts.Database)
    .SetAuthToken(std::getenv("YDB_TOKEN"));
```

```
NYdb::TDriver driver(driverConfig);
```

This example uses authentication token from the `YDB_TOKEN` environment variable. For details see Connecting to a database and Authentication pages.

App code snippet for creating a client:

```
NYdb::NTopic::TTopicClient topicClient(driver);
```

Java

To interact with YDB Topics, create an instance of the YDB transport and topic client.

The YDB transport lets the app and YDB interact at the transport layer. The transport must exist during the YDB access lifecycle and be initialized before creating a client.

App code snippet for transport initialization:

```
try (GrpcTransport transport = GrpcTransport.forConnectionString(connString)
        .withAuthProvider(CloudAuthHelper.getAuthProviderFromEnviron())
        .build()) {
    // Use YDB transport
}
```

In this example `CloudAuthHelper.getAuthProviderFromEnviron()` helper method is used which retrieves auth token from environment variables.
For example, `YDB_ACCESS_TOKEN_CREDENTIALS`.
For details see Connecting to a database and Authentication pages.

Topic client (source code) uses YDB transport and handles all topics topic operations, manages read and write sessions.

App code snippet for creating a client:

```
try (TopicClient topicClient = TopicClient.newClient(transport)
              .setCompressionExecutor(compressionExecutor)
              .build()) {
  // Use topic client
}
```

Both provided examples use (try-with-resources) block.
It allows to automatically close client and transport on leaving this block, considering both classes extends `AutoCloseable`.

C#

To interact with YDB Topics, create an instance of the YDB driver and topic client.

The YDB transport allows the app and YDB to interact at the transport layer. The transport must exist during the YDB access lifecycle and be initialized before creating a client.

App code snippet for transport initialization:

```
var config = new DriverConfig(
    endpoint: "grpc://localhost:2136",
    database: "/local"
);

await using var driver = await Driver.CreateInitialized(
    config: config,
    loggerFactory: loggerFactory
);
```

This example uses anonymous authentication. For details, see Connecting to a database and Authentication.

App code snippet for creating various clients:

```
var topicClient = new TopicClient(driver);

await using var writer = new WriterBuilder<string>(driver, topicName)
{
    ProducerId = "ProducerId_Example"
}.Build();

await using var reader = new ReaderBuilder<string>(driver)
{
    ConsumerName = "Consumer_Example",
    SubscribeSettings = { new SubscribeSettings(topicName) }
}.Build();
```

Python

To work with topics, create a YDB driver instance. The topic client is available via the `topic_client` attribute and is used for management operations on topics and for creating writers and readers.

Native SDK

```
import os
import ydb
```

```
driver_config = ydb.DriverConfig(
    endpoint=os.environ["YDB_ENDPOINT"],
    database=os.environ["YDB_DATABASE"],
)
driver = ydb.Driver(driver_config)
driver.wait(timeout=5)
# driver.topic_client — client for working with topics
writer = driver.topic_client.writer(topic_path)
reader = driver.topic_client.reader(topic=topic_path, consumer=consumer_name)
```

Native SDK (Asyncio)

```
import os
import ydb

driver_config = ydb.DriverConfig(
    endpoint=os.environ["YDB_ENDPOINT"],
    database=os.environ["YDB_DATABASE"],
)
async with ydb.aio.Driver(driver_config) as driver:
    await driver.wait(timeout=5)
    # driver.topic_client — client for working with topics
    writer = driver.topic_client.writer(topic_path)
    reader = driver.topic_client.reader(topic=topic_path, consumer=consumer_name)
```

For more on connecting to the database and authentication.

JavaScript

```
const t = topic(driver);

await using reader = t.createReader({
  topic: "/Root/demo-topic",
  consumer: "demo-consumer",
});

await using writer = t.createWriter({
  topic: "/Root/demo-topic",
  producer: "demo-producer",
});
```

Rust

```
use ydb::{ClientBuilder, YdbResult};

#[tokio::main]
async fn main() -> YdbResult<()> {
    let client = ClientBuilder::new_from_connection_string("grpc://localhost:2136/local")?.client()?;
    client.wait().await?;
    let mut topic_client = client.topic_client();
    // topic_client.create_reader(...), create_writer_with_params(...), ...
    Ok(())
}
```

Managing topics

Creating a topic

C++

For a full list of supported parameters, see the source code.

Example of creating a topic with three partitions and ZSTD codec support:

```
auto settings = NYdb::NTopic::TCreateTopicSettings()
    .PartitioningSettings(3, 3)
    .AppendSupportedCodecs(NYdb::NTopic::ECodec::ZSTD);

auto status = topicClient
    .CreateTopic("my-topic", settings)  // returns TFuture<TStatus>
    .GetValueSync();
```

Go

For a full list of supported parameters, see the SDK documentation.

Example of creating a topic with a list of supported codecs and a minimum number of partitions:

```
err := db.Topic().Create(ctx, "topic-path",
  // optional
  topicoptions.CreateWithSupportedCodecs(topictypes.CodecRaw, topictypes.CodecGzip),

  // optional
  topicoptions.CreateWithMinActivePartitions(3),
)
```

Python

Example of creating a topic with a list of supported codecs and a minimum number of partitions:

Native SDK

```
driver.topic_client.create_topic(topic_path,
    supported_codecs=[ydb.TopicCodec.RAW, ydb.TopicCodec.GZIP], # optional
    min_active_partitions=3,                                    # optional
)
```

Native SDK (Asyncio)

```
await driver.topic_client.create_topic(topic_path,
    supported_codecs=[ydb.TopicCodec.RAW, ydb.TopicCodec.GZIP],  # optional
    min_active_partitions=3,                                     # optional
)
```

Java

For a full list of supported parameters, see the source code.

```
topicClient.createTopic(topicPath, CreateTopicSettings.newBuilder()
                // Optional
                .setSupportedCodecs(SupportedCodecs.newBuilder()
                        .addCodec(Codec.RAW)
                        .addCodec(Codec.GZIP)
                        .build())
                // Optional
                .setPartitioningSettings(PartitioningSettings.newBuilder()
                        .setMinActivePartitions(3)
                        .build())
                .build());
```

C#

Example of creating a topic with a list of supported codecs and a minimum number of partitions:

```
await topicClient.CreateTopic(new CreateTopicSettings
{
    Path = topicName,
    Consumers = { new Consumer("Consumer_Example") },
    SupportedCodecs = { Codec.Raw, Codec.Gzip },
    PartitioningSettings = new PartitioningSettings
    {
        MinActivePartitions = 3
    }
});
```

JavaScript

```
const topicService = driver.createClient(TopicServiceDefinition);
await topicService.createTopic(
  create(CreateTopicRequestSchema, {
    path: "/path-to-my-topic",
    partitioningSettings: {
      minActivePartitions: 1n,
      maxActivePartitions: 100n,
    },
    consumers: [{ name: "my-consumer" }],
  }),
);
```

Rust

```
use ydb::{Codec, CreateTopicOptionsBuilder, YdbResult};

topic_client
    .create_topic(
        "/local/my-topic".into(),
        CreateTopicOptionsBuilder::default()
            .min_active_partitions(3)
            .supported_codecs(vec![Codec::Raw, Codec::Zstd])
            .build()?,
    )
    .await?;
```

Updating a topic

When you update a topic, you must specify the topic path and the parameters to be changed.

C++

For a full list of supported parameters, see the source code.

Example of adding an important consumer and setting two days retention time for the topic:

```
auto alterSettings = NYdb::NTopic::TAlterTopicSettings()
    .BeginAddConsumer("my-consumer")
        .Important(true)
    .EndAddConsumer()
    .SetRetentionPeriod(TDuration::Days(2));

auto status = topicClient
    .AlterTopic("my-topic", alterSettings)  // returns TFuture<TStatus>
    .GetValueSync();
```

Go

For a full list of supported parameters, see the SDK documentation.

Example of adding a consumer to a topic:

```
err := db.Topic().Alter(ctx, "topic-path",
  topicoptions.AlterWithAddConsumers(topictypes.Consumer{
    Name:            "new-consumer",
    SupportedCodecs: []topictypes.Codec{topictypes.CodecRaw, topictypes.CodecGzip}, // optional
  }),
)
```

Python

Example of updating a topic's list of supported codecs and minimum number of partitions:

Native SDK

```
driver.topic_client.alter_topic(topic_path,
    set_supported_codecs=[ydb.TopicCodec.RAW, ydb.TopicCodec.GZIP], # optional
    set_min_active_partitions=3,                                    # optional
)
```

Native SDK (Asyncio)

```
await driver.topic_client.alter_topic(topic_path,
    set_supported_codecs=[ydb.TopicCodec.RAW, ydb.TopicCodec.GZIP],  # optional
    set_min_active_partitions=3,                                     # optional
)
```

Java

For a full list of supported parameters, see the source code.

```
topicClient.alterTopic(topicPath, AlterTopicSettings.newBuilder()
                .addAddConsumer(Consumer.newBuilder()
                        .setName("new-consumer")
                        .setSupportedCodecs(SupportedCodecs.newBuilder()
                                .addCodec(Codec.RAW)
                                .addCodec(Codec.GZIP)
                                .build())
                        .build())
                .build());
```

JavaScript

```
const topicService = driver.createClient(TopicServiceDefinition);
await topicService.alterTopic(
  create(AlterTopicRequestSchema, {
    path: "/path-to-my-topic",
    addConsumers: [{ name: "my-consumer-2" }],
  }),
);
```

Rust

```
use ydb::{AlterTopicOptionsBuilder, YdbResult};

topic_client
    .alter_topic(
        "/local/my-topic".into(),
        AlterTopicOptionsBuilder::default()
            .set_min_active_partitions(Some(5))
            .build()?,
    )
    .await?;
```

Getting topic information

C++

Use `DescribeTopic` method to get information about topic.

For a full list of description fields, see the source code.

Example of using topic description:

```
auto result = topicClient.DescribeTopic("my-topic").GetValueSync();
if (result.IsSuccess()) {
    const auto& description = result.GetTopicDescription();
    std::cout << "Topic description: " << GetProto(description) << std::endl;
}
```

There is another method `DescribeConsumer` to get informtaion about consumer.

Go

```
  descResult, err := db.Topic().Describe(ctx, "topic-path")
if err != nil {
  log.Fatalf("failed drop topic: %v", err)
  return
}
fmt.Printf("describe: %#v\n", descResult)
```

Python

Native SDK

```
info = driver.topic_client.describe_topic(topic_path)
print(info)
```

Native SDK (Asyncio)

```
info = await driver.topic_client.describe_topic(topic_path)
print(info)
```

Java

Use `describeTopic` method to get information about topic.

For a full list of description fields, see the source code.

```
Result<TopicDescription> topicDescriptionResult = topicClient.describeTopic(topicPath)
        .join();
TopicDescription description = topicDescriptionResult.getValue();
```

JavaScript

```
const topicService = driver.createClient(TopicServiceDefinition);
await topicService.describeTopic(
  create(DescribeTopicRequestSchema, {
    path: "/path-to-my-topic",
  }),
);
```

Rust

```
use ydb::{DescribeTopicOptionsBuilder, YdbResult};

let description = topic_client
    .describe_topic(
        "/local/my-topic".into(),
        DescribeTopicOptionsBuilder::default().include_stats(true).build()?,
    )
    .await?;
```

Deleting a topic

To delete a topic, just specify the path to it.

C++

```
auto status = topicClient.DropTopic("my-topic").GetValueSync();
```

Go

```
  err := db.Topic().Drop(ctx, "topic-path")
```

Python

Native SDK

```
driver.topic_client.drop_topic(topic_path)
```

Native SDK (Asyncio)

```
await driver.topic_client.drop_topic(topic_path)
```

Java

```
topicClient.dropTopic(topicPath);
```

C#

```
await topicClient.DropTopic(topicName);
```

JavaScript

```
const topicService = driver.createClient(TopicServiceDefinition);
await topicService.dropTopic(
  create(DropTopicRequestSchema, {
    path: "/path-to-my-topic",
  }),
);
```

Rust

```
topic_client.drop_topic("/local/my-topic".into()).await?;
```

Message writes

Connecting to a topic for message writes

Only connections with matching producer and message group identifiers are currently supported (`producer_id` shoud be equal to `message_group_id`). This restriction will be removed in the future.

C++

The write session object with `IWriteSession` interface is used to connect to a topic for writing.

For a full list of write session settings, see the source code.

Example of creating a write session:

```
std::string producerAndGroupID = "group-id";
auto settings = NYdb::NTopic::TWriteSessionSettings()
    .Path("my-topic")
    .ProducerId(producerAndGroupID)
    .MessageGroupId(producerAndGroupID);

auto session = topicClient.CreateWriteSession(settings);
```

Go

```
producerAndGroupID := "group-id"
writer, err := db.Topic().StartWriter(producerAndGroupID, "topicName",
  topicoptions.WithMessageGroupID(producerAndGroupID),
)
if err != nil {
    return err
}
```

Python

Native SDK

```
writer = driver.topic_client.writer(topic_path)
```

Native SDK (Asyncio)

```
writer = driver.topic_client.writer(topic_path)
```

Java

Synchronous API

Writer settings initialization:

```
String producerAndGroupID = "group-id";
WriterSettings settings = WriterSettings.newBuilder()
      .setTopicPath(topicPath)
      .setProducerId(producerAndGroupID)
      .setMessageGroupId(producerAndGroupID)
      .build();
```

Sync writer creation:

```
SyncWriter writer = topicClient.createSyncWriter(settings);
```

After the writer is created, initialize it. There are two methods:

- `init()`: non-blocking; starts initialization in the background and does not wait for completion.

  ```
  writer.init();
  ```

- `initAndWait()`: blocking; starts initialization and waits for completion. If initialization fails, an exception is thrown.

  ```
  try {
      writer.initAndWait();
      logger.info("Init finished successfully");
  } catch (Exception exception) {
      logger.error("Exception while initializing writer: ", exception);
      return;
  }
  ```

Asynchronous API

Writer settings initialization:

```
String producerAndGroupID = "group-id";
WriterSettings settings = WriterSettings.newBuilder()
      .setTopicPath(topicPath)
      .setProducerId(producerAndGroupID)
      .setMessageGroupId(producerAndGroupID)
      .build();
```

Async writer creation and initialization:

```
AsyncWriter writer = topicClient.createAsyncWriter(settings);

// Init in background
writer.init()
        .thenRun(() -> logger.info("Init finished successfully"))
        .exceptionally(ex -> {
            logger.error("Init failed with ex: ", ex);
            return null;
        });
```

C#

```
await using var writer = new WriterBuilder<string>(driver, topicName)
{
    ProducerId = "ProducerId_Example"
}.Build();
```

JavaScript

```
await using writer = createTopicWriter(driver, {
  topic: topicName,
  producer: producerName,
});
```

Rust

```
use ydb::{TopicWriter, TopicWriterOptionsBuilder, YdbResult};

let writer: TopicWriter = topic_client
    .create_writer_with_params(
        TopicWriterOptionsBuilder::default()
            .topic_path("/local/my-topic".into())
            .producer_id("group-id".into())
            .message_group_id("group-id".into())
            .build()?,
    )
    .await?;
```

Writing messages

C++

`IWriteSession` interface allows asynchronous write.

The user processes three kinds of events in a loop: `TReadyToAcceptEvent`, `TAcksEvent`, and `TSessionClosedEvent`.

For each kind of event user can set a handler in write session settings before session creation. Also, a common handler can be set.

If handler is not set for a particular event, it will be delivered to SDK client via `GetEvent` / `GetEvents` methods. `WaitEvent` method allows user to await for a next event in non-blocking way with `TFuture<void>()` interface.

To write a message, user uses a move-only `TContinuationToken` object, which has been created by the SDK and has been delivered to the user with a `TReadyToAcceptEvent` event. During write user can set an arbitrary sequential number and a message creation timestamp. By default they are generated by the SDK.

`Write` is asynchronous. Data from messages is processed and stored in the internal buffer. Settings `MaxMemoryUsage`, `MaxInflightCount`, `BatchFlushInterval`, and `BatchFlushSizeBytes` control sending in the background. Write session reconnects to the YDB if the connection fails and resends the message if possible, with regard to `RetryPolicy` setting. If an error that cannot be repeated is received, write session stops and sends `TSessionClosedEvent` to the client.

Example of writing using event loop without any handlers set up:

```
// Event loop
while (true) {
    // Get event
    // May block for a while if write session is busy
    std::optional<NYdb::NTopic::TWriteSessionEvent::TEvent> event = session->GetEvent(/*block=*/true);

    if (auto* readyEvent = std::get_if<NYdb::NTopic::TWriteSessionEvent::TReadyToAcceptEvent>(&*event)) {
        session->Write(std::move(event.ContinuationToken), "This is yet another message.");

    } else if (auto* ackEvent = std::get_if<NYdb::NTopic::TWriteSessionEvent::TAcksEvent>(&*event)) {
        std::cout << ackEvent->DebugString() << std::endl;

    } else if (auto* closeSessionEvent = std::get_if<NYdb::NTopic::TSessionClosedEvent>(&*event)) {
        break;
    }
}
```

Go

To send a message, just save Reader in the Data field, from which the data can be read. You can expect the data of each message to be read once (or until the first error). By the time you return the data from Write, it will already have been read and stored in the internal buffer.

By default, SeqNo and the message creation date are set automatically.

By default, Write is performed asynchronously: data from messages is processed and stored in the internal buffer, sending is done in the background. Writer reconnects to the YDB if the connection fails and resends the message if possible. If an error that cannot be repeated is received , Writer stops and subsequent Write calls will end with an error.

```
err := writer.Write(ctx,
  topicwriter.Message{Data: strings.NewReader("1")},
  topicwriter.Message{Data: bytes.NewReader([]byte{1,2,3})},
  topicwriter.Message{Data: strings.NewReader("3")},
)
if err != nil {
  return err
}
```

Python

To deliver messages, you can either simply transmit message content (bytes, str) or set certain properties manually. You can send objects one-by-one or as a list. The `write` method is asynchronous. The method returns immediately once messages are put to the client's internal buffer; this is usually a fast process. If the internal buffer is filled up, you might need to wait until part of the data is sent to the server.

Native SDK

```
# Simple delivery of messages, without explicit metadata.
# Easy to get started, easy to use if everything you need is the message content.
writer = driver.topic_client.writer(topic_path)
writer.write("mess")  # Rows will be transmitted in UTF-8; this is the easiest way to send
                      # text messages.
writer.write(bytes([1, 2, 3]))  # These bytes will be transmitted as they are, this is the easiest way to sen
d
                                # binary data.
writer.write(["mess-1", "mess-2"])  # This line sends multiple messages per call
                                    # to decrease overheads on internal SDK processes.
                                    # This makes sense when the message stream is high.

# This is the full form; it is used when except the message content you need to manually specify its properti
es.
writer = driver.topic_client.writer(topic="topic-path", auto_seqno=False, auto_created_at=False)

writer.write(ydb.TopicWriterMessage("asd", seqno=123, created_at=datetime.datetime.now()))
writer.write(ydb.TopicWriterMessage(bytes([1, 2, 3]), seqno=124, created_at=datetime.datetime.now()))

# In the full form, you can also send multiple messages per function call.
# This approach is useful when the message stream is high, and you want to
# reduce overheads on SDK internal calls.
writer.write([
  ydb.TopicWriterMessage("asd", seqno=123, created_at=datetime.datetime.now()),
  ydb.TopicWriterMessage(bytes([1, 2, 3]), seqno=124, created_at=datetime.datetime.now(),
  ])
```

Native SDK (Asyncio)

```
writer = driver.topic_client.writer(topic_path)
await writer.write("mess")
```

```
await writer.write(bytes([1, 2, 3]))
await writer.write(["mess-1", "mess-2"])
```

Java

Synchronous API

The `send` method blocks until a message is placed in the writer's send queue.
Putting a message in this queue means the writer will do its best to deliver it.
For example, if the write session is dropped, the writer reconnects and tries to resend the message on a new session.
However, enqueueing does not guarantee the message will eventually be written.
Errors can shut down the writer before queued messages are sent.
If you need a per-message guarantee that data was written, use the async writer and inspect the status returned by `send`.

```
writer.send(Message.of("11".getBytes()));

long timeoutSeconds = 5; // How long should we wait for a message to be put into sending buffer
try {
    writer.send(
            Message.newBuilder()
                    .setData("22".getBytes())
                    .setCreateTimestamp(Instant.now().minusSeconds(5))
                    .build(),
            timeoutSeconds,
            TimeUnit.SECONDS
    );
} catch (TimeoutException exception) {
    logger.error("Send queue is full. Couldn't put message into sending queue within {} seconds", timeoutSeco
nds);
} catch (InterruptedException | ExecutionException exception) {
    logger.error("Couldn't put the message into sending queue due to exception: ", exception);
}
```

Asynchronous API

The `send` method is non-blocking for the async client. It enqueues a message for sending.
It returns `CompletableFuture<WriteAck>` so you can verify the message was actually written.
If the queue overflows, `QueueOverflowException` is thrown.
That signals the producer should slow down: skip messages or retry with exponential backoff.
You can also increase the client buffer size (`setMaxSendBufferMemorySize`) to hold more messages before overflow.

```
try {
    // Non-blocking. Throws QueueOverflowException if send queue is full
    writer.send(Message.of("33".getBytes()));
} catch (QueueOverflowException exception) {
    // Send queue is full. Need to retry with backoff or skip
}
```

C#

Asynchronous writing of a message to a topic.

```
var asyncWriteTask = writer.WriteAsync("Hello, Example YDB Topics!"); // Task<WriteResult>
```

JavaScript

```
// Write a message to the internal buffer.
writer.write(Buffer.from("Hello, world!", "utf-8"));

// To send immediately, call flush.
await writer.flush();

// Or close the writer.
await writer.close();
```

Rust

```
use ydb::{TopicWriterMessageBuilder, YdbResult};

writer
    .write(
        TopicWriterMessageBuilder::default()
            .data(b"payload".to_vec())
            .build()?,
    )
    .await?;
writer.stop().await?;
```

Message writes with storage confirmation on the server

C++

`IWriteSession` interface allows getting server acknowledgments for writes.

Status of server-side message write is represented with `TAcksEvent`. One event can contain the statuses of several previously sent messages.Status is one of the following: message write is confirmed (`EES_WRITTEN`), message is discarded as a duplicate of a previously written message (`EES_ALREADY_WRITTEN`) or message is discarded because of failure (`EES_DISCARDED`).

Example of setting TAcksEvent handler for a write session:

```
auto settings = NYdb::NTopic::TWriteSessionSettings()
  // other settings are set here
  .EventHandlers(
    NYdb::NTopic::TWriteSessionSettings::TEventHandlers()
      .AcksHandler(
        [&](NYdb::NTopic::TWriteSessionEvent::TAcksEvent& event) {
          for (const auto& ack : event.Acks) {
            if (ack.State == NYdb::NTopic::TWriteSessionEvent::TWriteAck::EEventState::EES_WRITTEN) {
              ackedSeqNo.insert(ack.SeqNo);
              std::cout << "Acknowledged message with seqNo " << ack.SeqNo << std::endl;
            }
          }
        }
      )
  );

auto session = topicClient.CreateWriteSession(settings);
```

In this write session user does not receive `TAcksEvent` events in the `GetEvent` / `GetEvents` loop. Instead, SDK will call given handler on every acknowledgment coming from server. In the same way user can set up handlers for other types of events.

Go

When connected, you can specify the synchronous message write option: topicoptions.WithSyncWrite(true). Then Write will only return after receiving a confirmation from the server that all messages passed in the call have been saved. If necessary, the SDK will reconnect and retry sending messages as usual. In this mode, the context only controls the response time from the SDK, meaning the SDK will continue trying to send messages even after the context is canceled.

```
producerAndGroupID := "group-id"
writer, _ := db.Topic().StartWriter(producerAndGroupID, "topicName",
  topicoptions.WithMessageGroupID(producerAndGroupID),
  topicoptions.WithSyncWrite(true),
)

err = writer.Write(ctx,
  topicwriter.Message{Data: strings.NewReader("1")},
  topicwriter.Message{Data: bytes.NewReader([]byte{1,2,3})},
  topicwriter.Message{Data: strings.NewReader("3")},
)
if err != nil {
  return err
}
```

Python

There are two ways to get a message write acknowledgement from the server:

- `flush()` - waits until all the messages previously written to the internal buffer are acknowledged.
- `write_with_ack(...)` - sends a message and waits for the acknowledgement of its delivery from the server. This method is slow when you are sending multiple messages in a row.

```
# Put multiple messages to the internal buffer and then wait
# until all of them are delivered to the server.
for mess in messages:
    writer.write(mess)

writer.flush()

# You can send multiple messages and wait for an acknowledgment for the entire group.
writer.write_with_ack(["mess-1", "mess-2"])

# Waiting on sending each message: this method will return the result only after an
# acknowledgment from the server.
# This is the slowest message delivery option; use it when this mode is
# absolutely needed.
writer.write_with_ack("message")
```

Java

The `send` method returns `CompletableFuture<WriteAck>`. Successful completion means the server acknowledged the write. The `WriteAck` structure contains seqNo, offset, and write status:

```
writer.send(Message.of(message))
      .whenComplete((result, ex) -> {
          if (ex != null) {
              logger.error("Exception on writing message message: ", ex);
          } else {
              switch (result.getState()) {
                  case WRITTEN:
                      WriteAck.Details details = result.getDetails();
                      StringBuilder str = new StringBuilder("Message was written successfully");
                      if (details != null) {
```

```
                        str.append(", offset: ").append(details.getOffset());
                    }
                    logger.debug(str.toString());
                    break;
                case ALREADY_WRITTEN:
                    logger.warn("Message has already been written");
                    break;
                default:
                    break;
            }
        }
    });
```

C#

Asynchronous writing of a message to a topic. If the internal buffer overflows, it waits for the buffer to be released before resending.

```
await writer.WriteAsync("Hello, Example YDB Topics!");
```

If the server is unavailable, messages may accumulate while waiting to be sent. In this case, you can pass a cancellation token (`CancellationToken`) to control waiting. However, if the user cancels the recorded message, it will still be canceled.

```
var writeCts = new CancellationTokenSource();
writeCts.CancelAfter(TimeSpan.FromSeconds(3));

await writer.WriteAsync("Hello, Example YDB Topics!", writeCts.Token);
```

JavaScript

All messages are written to an internal buffer. There are three mechanisms they reach the server: two automatic and one manual. The manual path is calling `writer.flush`, which returns the last seqNo persisted on the server. Automatic flushes happen when:

- The internal buffer exceeds `maxBufferBytes` (default 256 MiB).
- The periodic flush interval `flushIntervalMs` ticks (default 10 ms).

```
await using writer = createTopicWriter(driver, {
  topic: topicName,
  producer: producerName,
  // Callback that is called when writer receives an acknowledgment for a message.
  onAck: (seqNo, status) => {
    console.log("ACK", seqNo, status);
  },
})

writer.write(Buffer.from("Hello, world!", "utf-8"));

// Get the latest seqNo confirmed by the server.
await writer.flush();
```

Rust

```
use ydb::{TopicWriterMessageBuilder, YdbResult};

writer
    .write_with_ack(
        TopicWriterMessageBuilder::default()
            .data(b"payload".to_vec())
            .build()?,
    )
    .await?;
```

Selecting a codec for message compression

For more details on using data compression for topics, see here.

C++

The message compression can be set on the write session creation with `Codec` and `CompressionLevel` settings. By default, GZIP codec is chosen.

Example of creating a write session with no data compression:

```
auto settings = NYdb::NTopic::TWriteSessionSettings()
  // other settings are set here
  .Codec(ECodec::RAW);

auto session = topicClient.CreateWriteSession(settings);
```

Write session allows sending a message compressed with other codec. For this use `WriteEncoded` method, specify codec used and original message byte size. The codec must be allowed in topic settings.

Go

By default, the SDK selects the codec automatically based on topic settings. In automatic mode, the SDK first sends one group of messages with each of the allowed codecs, then it sometimes tries to compress messages with all the available codecs, and then selects the codec that yields the smallest message size. If the list of allowed codecs for the topic is empty, the SDK makes automatic selection between Raw and Gzip codecs.

If necessary, a fixed codec can be set in the connection options. It will then be used and no measurements will be taken.

```
producerAndGroupID := "group-id"
writer, _ := db.Topic().StartWriter(producerAndGroupID, "topicName",
  topicoptions.WithMessageGroupID(producerAndGroupID),
  topicoptions.WithCodec(topictypes.CodecGzip),
)
```

Python

By default, the SDK selects the codec automatically based on topic settings. In automatic mode, the SDK first sends one group of messages with each of the allowed codecs, then it sometimes tries to compress messages with all the available codecs, and then selects the codec that yields the smallest message size. If the list of allowed codecs for the topic is empty, the SDK makes automatic selection between Raw and Gzip codecs.

If necessary, a fixed codec can be set in the connection options. It will then be used and no measurements will be taken.

```
writer = driver.topic_client.writer(topic_path,
    codec=ydb.TopicCodec.GZIP,
)
```

Java

```
String producerAndGroupID = "group-id";
WriterSettings settings = WriterSettings.newBuilder()
        .setTopicPath(topicPath)
        .setProducerId(producerAndGroupID)
        .setMessageGroupId(producerAndGroupID)
        .setCodec(Codec.ZSTD)
        .build();
```

JavaScript

```
await using writer = t.createWriter({
  codec: Codec.RAW,
});

await using writer = t.createWriter({
  codec: Codec.GZIP,
});

await using writer = t.createWriter({
  codec: Codec.LZOP,
});

await using writer = t.createWriter({
  codec: 10000, // CUSTOM (allowed range: 10000–19999)
});
```

Rust

Per-writer message compression codec is not configurable in the Rust SDK yet; messages are sent with the `Raw` codec.

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#341

Writing messages in no-deduplication mode

C++\nIf no ProducerId is specified on write session setup, the session runs in no-deduplication mode. The example below demonstrates such a session setup:

```
auto settings = NYdb::NTopic::TWriteSessionSettings()
    .Path(myTopicPath);
auto session = topicClient.CreateWriteSession(settings);
```

If, on other hand, you want to ensure deduplication is enabled, you can specify the ProducerId option or call the `DeduplicationEnabled()` method from WriteSessionSettings. The 'Connecting to a topic for message writes' section has an example of write session that has deduplication enabled.

Go\nIn **ydb-go-sdk, when you create a writer without explicitly passing `topicoptions.WithWriterProducerID`, the SDK still assigns a producer ID (it generates one automatically). A mode equivalent to omitting `ProducerId` in the C++ example above is not available in the current SDK version.**

Java\nThis functionality is not currently supported.

Python

This functionality is not currently supported.

C#

This functionality is not currently supported.

JavaScript

This functionality is not currently supported.

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#341

PHP

This functionality is not currently supported.

Using message metadata feature

You can provide some metadata for any particular message when writing. This metadata can be a list of up to 1000 key-value pairs per message.
All the metadata provided when writing a message is sent to a consumer with the message during reading.

C++

To take advantage of message metadata feature, use the `Write()` method with `TWriteMessage` argument as below:

```
auto settings = NYdb::NTopic::TWriteSessionSettings()
    .Path(myTopicPath)
// set all other settings;
;

auto session = topicClient.CreateWriteSession(settings);

std::optional<NYdb::NTopic::TWriteSessionEvent::TEvent> event = session->GetEvent(/*block=*/true);
NYdb::NTopic::TWriteMessage message("This is yet another message").MessageMeta({
    {"meta-key", "meta-value"},
    {"another-key", "value"}
});

if (auto* readyEvent = std::get_if<NYdb::NTopic::TWriteSessionEvent::TReadyToAcceptEvent>(&*event)) {
    session->Write(std::move(event.ContinuationToken), std::move(message));
};
```

Java

Construct messages with the builder to take advantage of the message metadata feature. You can add `MetadataItem` objects to a message. Each item consists of a key of type `String` and a value of type `byte[]`.

`MetadataItem`s can be set as a `List`:

```
List<MetadataItem> metadataItems = Arrays.asList(
        new MetadataItem("meta-key", "meta-value".getBytes()),
        new MetadataItem("another-key", "value".getBytes())
);
writer.send(
        Message.newBuilder()
                .setMetadataItems(metadataItems)
                .build()
);
```

Or each `MetadataItem` can be added individually:

```
writer.send(
        Message.newBuilder()
                .addMetadataItem(new MetadataItem("meta-key", "meta-value".getBytes()))
                .addMetadataItem(new MetadataItem("another-key", "value".getBytes()))
                .build()
);
```

While reading, metadata can be received from a `Message` with the `getMetadataItems()` method:

```
Message message = reader.receive();
List<MetadataItem> metadata = message.getMetadataItems();
```

Python

To write a message that includes metadata, create the `TopicWriterMessage` object with the `metadata_items` argument as shown below:

```
message = ydb.TopicWriterMessage(data=f"message-data", metadata_items={"meta-key": "meta-value"})
writer.write(message)
```

While reading, retrieve metadata from the `metadata_items` field of the `PublicMessage` object:

```
message = reader.receive_message()
for meta_key, meta_value in message.metadata_items.items():
    print(f"{meta_key}: {meta_value}")
```

C#

```
await writer.WriteAsync(
    new Ydb.Sdk.Services.Topic.Writer.Message<string>("Hello Example YDB Topics!")
```

```
        { Metadata = { new Metadata("meta-key", "meta-value"u8.ToArray()) } }
);
```

Go

Set metadata in the `Metadata` field of `topicwriter.Message`:

```
err := writer.Write(ctx, topicwriter.Message{
  Data: strings.NewReader("message-data"),
  Metadata: map[string][]byte{
    "meta-key":    []byte("meta-value"),
    "another-key": []byte("value"),
  },
})
```

When reading, metadata is available on the message:

```
msg, err := reader.ReadMessage(ctx)
if err != nil {
  return err
}
for k, v := range msg.Metadata {
  fmt.Printf("%s: %s\n", k, string(v))
}
```

JavaScript

```
writer.write(Buffer.from("Hello, world!", "utf-8"), {
  metadataItems: {
    "meta-key": new TextEncoder().encode("meta-value"),
  },
});
```

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#341

Write in a transaction

C++

To write to a topic within a transaction, it is necessary to pass a transaction object reference to the `Write` method of the writing session.

Example on GitHub

```
NYdb::NQuery::TQueryClient queryClient(driver);

NYdb::NStatusHelpers::ThrowOnError(queryClient.RetryQuerySync([](NYdb::NQuery::TSession session) -> NYdb::TSt
atus {
    auto beginTxResult = session.BeginTransaction().GetValueSync();
    if (!beginTxResult.IsSuccess()) {
        return beginTxResult;
    }
    auto tx = beginTxResult.GetTransaction();

    NYdb::NTopic::TWriteMessage writeMessage("message");

    topicSession->Write(std::move(writeMessage), tx);
    return tx.Commit().GetValueSync();
}));
```

Go

To write to a topic within a transaction, create a transactional writer by calling TopicClient.StartTransactionalWriter with the `tx` argument. Once created, you can send messages as usual. There's no need to close the transactional writer manually, as it will be closed automatically when the transaction ends.

Example on GitHub

```
err := db.Query().DoTx(ctx, func(ctx context.Context, tx query.TxActor) error {
  writer, err := db.Topic().StartTransactionalWriter(tx, topicName)
  if err != nil {
    return err
  }

  return writer.Write(ctx, topicwriter.Message{Data: strings.NewReader("asd")})
})
```

Python

To write to a topic within a transaction, create a transactional writer by calling `topic_client.tx_writer` with the `tx` argument. Once created, you can send messages as usual. There's no need to close the transactional writer manually, as it will be closed automatically when the transaction ends.

In the example below, there is no explicit call to `tx.commit()`; it occurs implicitly upon the successful execution of the `callee` lambda.

Example on GitHub

```
with ydb.QuerySessionPool(driver) as session_pool:

    def callee(tx: ydb.QueryTxContext):
        tx_writer: ydb.TopicTxWriter = driver.topic_client.tx_writer(tx, topic)

        for i in range(message_count):
            result_stream = tx.execute(query=f"select {i} as res;")
            for result_set in result_stream:
                message = str(result_set.rows[0]["res"])
                tx_writer.write(ydb.TopicWriterMessage(message))
                print(f"Message {message} was written with tx.")

    session_pool.retry_tx_sync(callee)
```

Python (asyncio)

To write to a topic within a transaction, create a transactional writer by calling `topic_client.tx_writer` with the `tx` argument. Once created, you can send messages as usual. There's no need to close the transactional writer manually, as it will be closed automatically when the transaction ends.

In the example below, there is no explicit call to `tx.commit()`; it occurs implicitly upon the successful execution of the `callee` lambda.

Example on GitHub

```
async with ydb.aio.QuerySessionPool(driver) as session_pool:

    async def callee(tx: ydb.aio.QueryTxContext):
        tx_writer: ydb.TopicTxWriterAsyncIO = driver.topic_client.tx_writer(tx, topic)

        for i in range(message_count):
            async with await tx.execute(query=f"select {i} as res;") as result_stream:
                async for result_set in result_stream:
                    message = str(result_set.rows[0]["res"])
                    await tx_writer.write(ydb.TopicWriterMessage(message))
                    print(f"Message {result_set.rows[0]['res']} was written with tx.")

    await session_pool.retry_tx_async(callee)
```

Java

Synchronous API

Example on GitHub

You can pass a transaction in `SendSettings` for the `send` method.
The message is then written together with that transaction's commit.

```
// creating a session in the table service
Result<Session> sessionResult = tableClient.createSession(Duration.ofSeconds(10)).join();
if (!sessionResult.isSuccess()) {
    logger.error("Couldn't get a session from the pool: {}", sessionResult);
    return; // retry or shutdown
}
Session session = sessionResult.getValue();
// creating a transaction in the table service
// this transaction is not yet active and has no id
TableTransaction transaction = session.createNewTransaction(TxMode.SERIALIZABLE_RW);

// get message text within the transaction
Result<DataQueryResult> dataQueryResult = transaction.executeDataQuery("SELECT \"Hello, world!\";")
        .join();
if (!dataQueryResult.isSuccess()) {
    logger.error("Couldn't execute DataQuery: {}", dataQueryResult);
    return; // retry or shutdown
}
// now the transaction is active and has an id

ResultSetReader rsReader = dataQueryResult.getValue().getResultSet(0);
byte[] message;
if (rsReader.next()) {
    message = rsReader.getColumn(0).getBytes();
} else {
    return; // retry or shutdown
}

writer.send(
        Message.of(message),
        SendSettings.newBuilder()
                .setTransaction(transaction)
                .build()
);

// flush to wait until all messages reach server before commit
writer.flush();
```

```
Status commitStatus = transaction.commit().join();
analyzeCommitStatus(commitStatus);
```

Asynchronous API

Example on GitHub

You can pass a transaction in `SendSettings` for the `send` method.
The message is then written together with that transaction's commit.

```
// creating a session in the table service
Result<Session> sessionResult = tableClient.createSession(Duration.ofSeconds(10)).join();
if (!sessionResult.isSuccess()) {
    logger.error("Couldn't get a session from the pool: {}", sessionResult);
    return; // retry or shutdown
}
Session session = sessionResult.getValue();
// creating a transaction in the table service
// this transaction is not yet active and has no id
TableTransaction transaction = session.createNewTransaction(TxMode.SERIALIZABLE_RW);

// get message text within the transaction
Result<DataQueryResult> dataQueryResult = transaction.executeDataQuery("SELECT \"Hello, world!\";")
        .join();
if (!dataQueryResult.isSuccess()) {
    logger.error("Couldn't execute DataQuery: {}", dataQueryResult);
    return; // retry or shutdown
}
// now the transaction is active and has an id

ResultSetReader rsReader = dataQueryResult.getValue().getResultSet(0);
byte[] message;
if (rsReader.next()) {
    message = rsReader.getColumn(0).getBytes();
} else {
    return; // retry or shutdown
}

try {
    writer.send(Message.newBuilder()
                            .setData(message)
                            .build(),
                    SendSettings.newBuilder()
                            .setTransaction(transaction)
                            .build())
            .whenComplete((result, ex) -> {
                if (ex != null) {
                    logger.error("Exception while sending a message: ", ex);
                } else {
                    switch (result.getState()) {
                        case WRITTEN:
                            WriteAck.Details details = result.getDetails();
                            logger.info("Message was written successfully, offset: " + details.getOffset());
                            break;
                        case ALREADY_WRITTEN:
                            logger.info("Message has already been written");
                            break;
                        default:
                            break;
                    }
                }
            })
            // Waiting for the message to reach the server before committing the transaction
            .join();

    Status commitStatus = transaction.commit().join();
    analyzeCommitStatus(commitStatus);
} catch (QueueOverflowException exception) {
    logger.error("Queue overflow exception while sending a message{}: ", index, exception);
    // Send queue is full. Need to retry with backoff or skip
}
```

Note

Transaction requirements:

- It should be an active transaction (that has an id) from one of YDB services. I.e., Table or Query.
- Only the `SERIALIZABLE_RW` transaction isolation level is supported in the Topic Service.

JavaScript

This section is under development.

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#341

Reading messages

Connecting to a topic for message reads

Reading messages from a topic can be done by specifying a Consumer associated with that topic, as well as without a Consumer. If a Consumer is not specified, the client application must calculate the offset for reading messages on its own. A more detailed example of reading without a Consumer is discussed in the relevant section.

A Consumer can be created on creating or altering a topic.
Topic can have several Consumers and for each of them server stores its own reading progress.

C++

The read session object with `IReadSession` interface is used to connect to one or more topics for reading.

For a full list of read session settings, see `TReadSessionSettings` class in the source code.

To establish a connection to the existing `my-topic` topic using the added `my-consumer` consumer, use the following code:

```
auto settings = NYdb::NTopic::TReadSessionSettings()
    .ConsumerName("my-consumer")
    .AppendTopics("my-topic");

auto session = topicClient.CreateReadSession(settings);
```

Go

To establish a connection to the existing `my-topic` topic using the added `my-consumer` consumer, use the following code:

```
reader, err := db.Topic().StartReader("my-consumer", topicoptions.ReadTopic("my-topic"))
if err != nil {
    return err
}
```

Python

To establish a connection to the existing `my-topic` topic using the added `my-consumer` consumer, use the following code:

```
reader = driver.topic_client.reader(topic="my-topic", consumer="my-consumer")
```

Java

Synchronous API

Reader settings initialization:

```
ReaderSettings settings = ReaderSettings.newBuilder()
        .setConsumerName(consumerName)
        .addTopic(TopicReadSettings.newBuilder()
                .setPath(topicPath)
                .setReadFrom(Instant.now().minus(Duration.ofHours(24))) // Optional
                .setMaxLag(Duration.ofMinutes(30)) // Optional
                .build())
        .build();
```

Sync reader creation:

```
SyncReader reader = topicClient.createSyncReader(settings);
```

After a reader is created, initialize it. The sync reader supports two methods:

- `init()`: non-blocking; starts initialization in the background and does not wait for completion.

  ```
  reader.init();
  ```

- `initAndWait()`: blocking; waits for initialization. On failure, an exception is thrown.

  ```
  try {
      reader.initAndWait();
      logger.info("Init finished successfully");
  } catch (Exception exception) {
      logger.error("Exception while initializing reader: ", exception);
      return;
  }
  ```

Asynchronous API

Reader settings initialization:

```
ReaderSettings settings = ReaderSettings.newBuilder()
        .setConsumerName(consumerName)
        .addTopic(TopicReadSettings.newBuilder()
                .setPath(topicPath)
                .setReadFrom(Instant.now().minus(Duration.ofHours(24))) // Optional
                .setMaxLag(Duration.ofMinutes(30)) // Optional
```

```
                .build())
        .build();
```

For the async reader, provide `ReadEventHandlersSettings` with a `ReadEventHandler` implementation that defines how events are handled while reading.

```
ReadEventHandlersSettings handlerSettings = ReadEventHandlersSettings.newBuilder()
    .setEventHandler(new Handler())
    .build();
```

Optionally, specify an executor for message handling in `ReadEventHandlersSettings`.
You can subclass `AbstractReadEventHandler` and override `onMessages`. Example:

```
private class Handler extends AbstractReadEventHandler {
    @Override
    public void onMessages(DataReceivedEvent event) {
        for (Message message : event.getMessages()) {
            StringBuilder str = new StringBuilder();
            logger.info("Message received. SeqNo={}, offset={}", message.getSeqNo(), message.getOffset());

            process(message);

            message.commit().thenRun(() -> {
                logger.info("Message committed");
            });
        }
    }
}
```

Async reader creation and initialization:

```
AsyncReader reader = topicClient.createAsyncReader(readerSettings, handlerSettings);
// Init in background
reader.init()
        .thenRun(() -> logger.info("Init finished successfully"))
        .exceptionally(ex -> {
            logger.error("Init failed with ex: ", ex);
            return null;
        });
```

C#

```
await using var reader = new ReaderBuilder<string>(driver)
{
    ConsumerName = "Consumer_Example",
    SubscribeSettings = { new SubscribeSettings(topicName) }
}.Build();
```

JavaScript

```
await using reader = createTopicReader(driver, {
  topic: topicName,
  consumer: consumerName,
});
```

Rust

```
use ydb::YdbResult;

let mut reader = topic_client
    .create_reader("my-consumer", "/local/my-topic")
    .await?;
```

Additional options are used to specify multiple topics and other parameters.
To establish a connection to the `my-topic` and `my-specific-topic` topics using the `my-consumer` consumer and also set the time to start reading messages, use the following code:

C++

```
auto settings = NYdb::NTopic::TReadSessionSettings()
    .ConsumerName("my-consumer")
    .AppendTopics("my-topic")
    .AppendTopics(
        NYdb::NTopic::TTopicReadSettings("my-specific-topic")
            .ReadFromTimestamp(someTimestamp)
    );

auto session = topicClient.CreateReadSession(settings);
```

Go

```
reader, err := db.Topic().StartReader("my-consumer", []topicoptions.ReadSelector{
    {
        Path: "my-topic",
    },
    {
        Path:       "my-specific-topic",
        ReadFrom:   time.Date(2022, 7, 1, 10, 15, 0, 0, time.UTC),
    },
    },
)
if err != nil {
    return err
}
```

Python

This feature is under development.

Java

```
ReaderSettings settings = ReaderSettings.newBuilder()
        .setConsumerName(consumerName)
        .addTopic(TopicReadSettings.newBuilder()
                .setPath("my-topic")
                .build())
        .addTopic(TopicReadSettings.newBuilder()
                .setPath("my-specific-topic")
                .setReadFrom(Instant.now().minus(Duration.ofHours(24))) // Optional
                .setMaxLag(Duration.ofMinutes(30)) // Optional
                .build())
        .build();
```

C#

```
await using var reader = new ReaderBuilder<string>(driver)
{
    ConsumerName = "Consumer_Example",
    SubscribeSettings =
    {
        new SubscribeSettings(topicName),
        new SubscribeSettings(topicName + "_another") { ReadFrom = DateTime.Now }
    }
}.Build();
```

JavaScript

```
await using reader = createTopicReader(driver, {
  topic: {
    path: topicPath,
    partitionIds: [1n, 2n, 3n],
  },
  consumer: consumerName,
});

await using reader = createTopicReader(driver, {
  topic: {
    path: topicPath,
    maxLag: "1s", // number, import('ms').StringValue, protobuf Duration
  },
  consumer: consumerName,
});

await using reader = createTopicReader(driver, {
  topic: {
    path: topicPath,
    readFrom: new Date(), // number, Date, protobuf Timestamp
  },
  consumer: consumerName,
});

await using reader = createTopicReader(driver, {
  topic: [
    {
      path: topicPath,
      partitionIds: [1n, 2n, 3n],
    },
    {
      path: topicPath2,
      maxLag: "1s",
    },
    {
      path: topicPath3,
      readFrom: new Date(),
    },
    // ...
  ],
```

```
  consumer: consumerName,
});
```

Rust

```
use std::time::{Duration, SystemTime};

use ydb::{TopicReaderOptionsBuilder, TopicSelector, TopicSelectors, YdbResult};

let mut reader = topic_client
    .create_reader_with_params(
        TopicReaderOptionsBuilder::default()
            .consumer("my-consumer".into())
            .topic(TopicSelectors(vec![
                TopicSelector::new("/local/my-topic"),
                TopicSelector {
                    path: "/local/my-specific-topic".into(),
                    partition_ids: None,
                    read_from: Some(SystemTime::now() - Duration::from_secs(3600)),
                },
            ]))
            .build()?,
    )
    .await?;
```

Reading messages

The server stores the consumer offset. After reading a message, the client should send a commit to the server. The consumer offset changes and only uncommitted messages will be read in case of a new connection.

You can read messages without a commit as well. In this case, all uncommited messages, including those processed, will be read if there is a new connection.

Information about which messages have already been processed can be saved on the client side by sending the starting consumer offset to the server when creating a new connection. This does not change the consumer offset on the server.

Data from topics can be read in the context of transactions. In this case, the reading offset will only advance when the transaction is committed. On reconnect, all uncommitted messages will be read again.

C++

The user processes several kinds of events in a loop: `TDataReceivedEvent`, `TCommitOffsetAcknowledgementEvent`, `TStartPartitionSessionEvent`, `TStopPartitionSessionEvent`, `TPartitionSessionStatusEvent`, `TPartitionSessionClosedEvent` and `TSessionClosedEvent`.

For each kind of event user can set a handler in read session settings before session creation. Also, a common handler can be set.

If handler is not set for a particular event, it will be delivered to SDK client via `GetEvent` / `GetEvents` methods. The `WaitEvent` method allows user to await for a next event in non-blocking way with `TFuture<void>()` interface.

Go

The SDK receives data from the server in batches and buffers it. Depending on the task, the client code can read messages from the buffer one by one or in batches.

Python

The SDK receives data from the server in batches and buffers it. Depending on the task, the client code can read messages from the buffer one by one or in batches.

Java

The SDK receives data from the server in batches and buffers it. Depending on the task, the client code can read messages from the buffer one by one or in batches.

C#

The SDK receives data from the server in batches and buffers it. Depending on the task, the client code can read messages from the buffer one by one or in batches.

JavaScript

This section is under development.

Rust

Full example of reading a topic in a transaction with table writes: `topic-read-in-transaction-example.rs`.

```
let batch = reader.pop_batch_in_tx(&mut tx).await?;
// process batch.messages and commit the transaction
```

Reading without a commit

Reading messages one by one

C++

Reading messages one-by-one is not supported in the C++ SDK. Class `TDataReceivedEvent` represents a batch of read messages.

Go

```
func SimpleReadMessages(ctx context.Context, r *topicreader.Reader) error {
    for {
```

```
        mess, err := r.ReadMessage(ctx)
        if err != nil {
            return err
        }
        processMessage(mess)
    }
}
```

Python

Native SDK

```
while True:
    message = reader.receive_message()
    process(message)
```

Native SDK (Asyncio)

```
while True:
    message = await reader.receive_message()
    process(message)
```

Java

Synchronous API

To read messages one-by-one without commit, do not call `commit` on messages:

```
while(true) {
    Message message = reader.receive();
    process(message);
}
```

Asynchronous API

The async client does not support reading messages one-by-one.

C#

```
try
{
    while (!readerCts.IsCancellationRequested)
    {
        var message = await reader.ReadAsync(readerCts.Token);

        logger.LogInformation("Received message: [{MessageData}]", message.Data);
    }
}
catch (OperationCanceledException)
{
}
```

JavaScript

```
for await (let batch of reader.read()) {
  for await (let msg of batch) {
  }
}
```

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#330

Reading message batches

C++

One simple way to read messages is to use `SimpleDataHandlers` setting when creating a read session. With it you only set a handler for a `TDataReceivedEvent`. SDK will call it for each batch of messages that came from server. By default, SDK does not send back acknowledgments of successful reads.

```
auto settings = NYdb::NTopic::TReadSessionSettings()
    .EventHandlers_.SimpleDataHandlers(
        [](NYdb::NTopic::TReadSessionEvent::TDataReceivedEvent& event) {
            std::cout << "Get data event " << NYdb::NTopic::DebugString(event);
        }
    );

auto session = topicClient.CreateReadSession(settings);

// Wait SessionClosed event.
session->GetEvent(/* block = */true);
```

In this example client creates read session and just awaits session close in the main thread. All other event types are handled by SDK.

Go

```
func SimpleReadBatches(ctx context.Context, r *topicreader.Reader) error {
    for {
        batch, err := r.ReadMessageBatch(ctx)
        if err != nil {
            return err
        }
        processBatch(batch)
    }
}
```

Python

Native SDK

```
while True:
    batch = reader.receive_batch()
    process(batch)
```

Native SDK (Asyncio)

```
while True:
    batch = await reader.receive_batch()
    process(batch)
```

JavaScript

```
for await (let batch of reader.read()) {
}
```

Java

Synchronous API

Batch reading is not supported in the sync reader.

Asynchronous API

To read without commit, do not call `commit`:

```
private class Handler extends AbstractReadEventHandler {
    @Override
    public void onMessages(DataReceivedEvent event) {
        for (Message message : event.getMessages()) {
            process(message);
        }
    }
}
```

C#

```
try
{
    while (!readerCts.IsCancellationRequested)
    {
        var batchMessages = await reader.ReadBatchAsync(readerCts.Token);

        foreach (var message in batchMessages.Batch)
        {
            logger.LogInformation("Received message: [{MessageData}]", message.Data);
        }
    }
}
catch (OperationCanceledException)
{
}
```

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#330

Reading with a commit

Confirmation of message processing (commit) informs the server that the message from the topic has been processed by the recipient and does not need to be sent anymore. When using acknowledged reading, it is necessary to confirm all received messages without skipping any. Message commits on the server occur after confirming a consecutive interval of messages "without gaps," and the confirmations themselves can be sent in any order.

For example, if messages 1, 2, 3 are received from the server, the program processes them in parallel and sends confirmations in the following order: 1, 3, 2. In this case, message 1 will be committed first, and messages 2 and 3 will be committed only after the server

receives confirmation of the processing of message 2.

If a commit fails with an error, the application should log it and continue; it makes no sense to retry the commit. At this point, it is not known if the message was actually confirmed.

Reading messages one by one with commits

C++

Reading messages one-by-one is not supported in the C++ SDK. Class `TDataReceivedEvent` represents a batch of read messages.

Go

```
func SimpleReadMessages(ctx context.Context, r *topicreader.Reader) error {
    for {
      mess, err := r.ReadMessage(ctx)
      if err != nil {
          return err
      }
      processMessage(mess)
      r.Commit(mess.Context(), mess)
    }
}
```

The `Commit` call is fast by default, saving data into an internal buffer and returning control to the caller. The real message to the server is sent in the background. To prevent losing the last commits, call the `Reader.Close()` method before exiting the program.

Python

The `commit` call is fast, saving data into an internal buffer and returning control back to the caller. The real message to the server is sent in the background. To prevent losing the last commits, you should call the `Reader.Close()` method before exiting the program.

Native SDK

```
while True:
    message = reader.receive_message()
    process(message)
    reader.commit(message)
```

Native SDK (Asyncio)

```
while True:
    message = await reader.receive_message()
    process(message)
    reader.commit(message)
```

Java

To commit a message just call `commit` method on it.
This method returns `CompletableFuture<Void>` which successful completion means that the server confirmed commit.
In case of an error on commit do not retry it. Most likely, an error is caused be session shutdown.
The reader (maybe another one) will create a new session for this partition and the message will be read again.

```
message.commit()
       .whenComplete((result, ex) -> {
           if (ex != null) {
               // Read session was probably closed, there is nothing we can do here.
               // Do not retry this commit on the same message.
               logger.error("exception while committing message: ", ex);
           } else {
               logger.info("message committed successfully");
           }
       });
```

C#

```
try
{
    while (!readerCts.IsCancellationRequested)
    {
        var message = await reader.ReadAsync(readerCts.Token);

        logger.LogInformation("Received message: [{MessageData}]", message.Data);

        try
        {
            await message.CommitAsync();
        }
        catch (ReaderException e)
        {
            logger.LogError(e, "Failed to commit a message");
        }
    }
}
catch (OperationCanceledException)
{
}
```

JavaScript

```
for await (let batch of reader.read()) {
  for (let msg of batch) {
    await reader.commit(msg);
  }
}
```

Reading message batches with commits

C++

Same as above example, when using `SimpleDataHandlers` handlers you only set handler for a `TDataReceivedEvent`. SDK will call it for each batch of messages that came from server. By setting `commitDataAfterProcessing = true`, you tell SDK to send back commits after executing a handler for corresponding event.

```
auto settings = NYdb::NTopic::TReadSessionSettings()
    .EventHandlers_.SimpleDataHandlers(
        [](NYdb::NTopic::TReadSessionEvent::TDataReceivedEvent& event) {
            std::cout << "Get data event " << NYdb::NTopic::DebugString(event);
        }
        , /* commitDataAfterProcessing = */true
    );

auto session = topicClient.CreateReadSession(settings);

// Wait SessionClosed event.
session->GetEvent(/* block = */true);
```

Go

```
func SimpleReadMessageBatch(ctx context.Context, r *topicreader.Reader) error {
    for {
      batch, err := r.ReadMessageBatch(ctx)
      if err != nil {
          return err
      }
      processBatch(batch)
      r.Commit(batch.Context(), batch)
    }
}
```

The `Commit` call is fast by default, saving data into an internal buffer and returning control back to the caller. The real message to the server is sent in the background. To prevent losing the last commits, you should call the `Reader.Close()` method before exiting the program.

Python

Native SDK

```
while True:
    batch = reader.receive_batch()
    process(batch)
    reader.commit(batch)
```

Native SDK (Asyncio)

```
while True:
    batch = await reader.receive_batch()
    process(batch)
    reader.commit(batch)
```

The `commit` call is fast, saving data into an internal buffer and returning control back to the caller. The real message to the server is sent in the background. To prevent losing the last commits, you should call the `Reader.Close()` method before exiting the program.

Java

Synchronous API

Not applicable: the sync reader does not support batch reads.

Asynchronous API

In `onMessages`, commit the whole batch in `DataReceivedEvent`:

```
@Override
public void onMessages(DataReceivedEvent event) {
    for (Message message : event.getMessages()) {
        process(message);
    }
    event.commit()
           .whenComplete((result, ex) -> {
               if (ex != null) {
                   // Read session was probably closed, there is nothing we can do here.
                   // Do not retry this commit on the same event.
                   logger.error("exception while committing message batch: ", ex);
```

```
                } else {
                    logger.info("message batch committed successfully");
                }
            });
}
```

C#

```
try
{
    while (!readerCts.IsCancellationRequested)
    {
        var batchMessages = await reader.ReadBatchAsync(readerCts.Token);

        foreach (var message in batchMessages.Batch)
        {
            logger.LogInformation("Received message: [{MessageData}]", message.Data);
        }

        try
        {
            await batchMessages.CommitBatchAsync();
        }
        catch (ReaderException e)
        {
            logger.LogError(e, "Failed to commit a message");
        }
    }
}
catch (OperationCanceledException)
{
}
```

JavaScript

```
for await (let batch of reader.read()) {
  await reader.commit(batch);
}
```

Rust

```
let batch = reader.read_batch().await?;
reader.commit(batch.get_commit_marker())?;
// or with waiting for server acknowledgment:
reader.commit_with_ack(batch.get_commit_marker()).await?;
```

Reading with consumer offset storage on the client side

Instead of committing messages, the client application may track reading progress on its own. In this case, it can provide a handler, which will be called back on each partition read start. This handler may set a starting reading position for this partition.

C++

The starting position of a specific partition read can be set during `TStartPartitionSessionEvent` handling.
For this purpose, `TStartPartitionSessionEvent::Confirm` has a `readOffset` parameter.
Additionally, there is a `commitOffset` parameter that tells the server to consider all messages with lesser offsets committed.

Setting handler example:

```
settings.EventHandlers_.StartPartitionSessionHandler(
    [](NYdb::NTopic::TReadSessionEvent::TStartPartitionSessionEvent& event) {
        auto readFromOffset = GetOffsetToReadFrom(event.GetPartitionId());
        event.Confirm(readFromOffset);
    }
);
```

In the code above, `GetOffsetToReadFrom` is part of the example, not SDK. Use your own method to provide the correct starting offset for a partition with a given partition id.

Also, `TReadSessionSettings` has a `ReadFromTimestamp` setting for reading only messages newer than the given timestamp. This setting is intended to skip some messages, not for precise reading start positioning. Several first-received messages may still have timestamps less than the specified one.

Go

```
func ReadWithExplicitPartitionStartStopHandlerAndOwnReadProgressStorage(ctx context.Context, db ydb.Connectio
n) error {
    readContext, stopReader := context.WithCancel(context.Background())
    defer stopReader()

    readStartPosition := func(
        ctx context.Context,
        req topicoptions.GetPartitionStartOffsetRequest,
    ) (res topicoptions.GetPartitionStartOffsetResponse, err error) {
        offset, err := readLastOffsetFromDB(ctx, req.Topic, req.PartitionID)
        res.StartFrom(offset)
```

```
        // Reader will stop if return err != nil
        return res, err
    }

    r, err := db.Topic().StartReader("my-consumer", topicoptions.ReadTopic("my-topic"),
        topicoptions.WithGetPartitionStartOffset(readStartPosition),
    )
    if err != nil {
        return err
    }

    go func() {
        <-readContext.Done()
        _ = r.Close(ctx)
    }()

    for {
        batch, err := r.ReadMessageBatch(readContext)
        if err != nil {
            return err
        }

        processBatch(batch)
        _ = externalSystemCommit(batch.Context(), batch.Topic(), batch.PartitionID(), batch.EndOffset())
    }
}
```

Python

This feature is under development.

Java

The starting offset for reading in Java can only be set for AsyncReader.
In `StartPartitionSessionEvent`, a `StartPartitionSessionSettings` object with the desired ReadOffset can be passed to the `confirm` method.
The offset that should be considered as committed can be set with the `setCommittedOffset` method.

```
@Override
public void onStartPartitionSession(StartPartitionSessionEvent event) {
    event.confirm(StartPartitionSessionSettings.newBuilder()
            .setReadOffset(lastReadOffset) // Long
            .setCommitOffset(lastCommitOffset) // Long
            .build());
}
```

The `setReadFrom` setting is used for reading only messages with write timestamps no less than the given one.

JavaScript

```
await using reader = createTopicReader(driver, {
  topic: topicName,
  consumer: consumerName,
  onPartitionSessionStart: (evt) => {
    return {
      readOffset: 0n,
      commitOffset: 0n,
    };
  },
});
```

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#330

Reading without a Consumer

Reading progress is usually saved on a server for each Consumer. However, such progress can't be saved if a reader is created without a specified `Consumer`.

Go

Pass an empty string as the consumer name and use the `topicoptions.WithReaderWithoutConsumer(false)` option (this mode is **experimental**; see VERSIONING in the SDK repository). In the read selector, specify the topic path and partition list. Message commits are not available in this mode (`CommitModeNone`); on reconnects you must restore progress on the client side—see client-side offset storage.

```
reader, err := db.Topic().StartReader(
  "",
  topicoptions.ReadSelectors{{
    Path:       "topic-path",
    Partitions: []int64{0, 1, 2},
  }},
  topicoptions.WithReaderWithoutConsumer(false),
)
if err != nil {
```

```
  return err
}
```

Java

To read without a Consumer, the `withoutConsumer()` method should be called explicitly on the `ReaderSettings` builder:

```
ReaderSettings settings = ReaderSettings.newBuilder()
        .withoutConsumer()
        .addTopic(TopicReadSettings.newBuilder()
                .setPath(TOPIC_NAME)
                .build())
        .build();
```

In this case, reading progress on the server will be lost on partition session restart.
To avoid reading from the beginning each time, starting offsets should be set on each partition session start:

```
@Override
public void onStartPartitionSession(StartPartitionSessionEvent event) {
    event.confirm(StartPartitionSessionSettings.newBuilder()
            .setReadOffset(lastReadOffset) // the last offset read by this client, Long
            .build());
}
```

Python

To read without a `Consumer`, create a reader using the `reader` method with specifying these arguments:

- `topic` - `ydb.TopicReaderSelector` object with defined `path` and `partitions` list;
- `consumer` - should be `None`;
- `event_handler` - inheritor of `ydb.TopicReaderEvents.EventHandler` that implements the `on_partition_get_start_offset` function. This function is responsible for returning the initial offset for reading messages when the reader starts and during reconnections. The client application must specify this offset in the parameter `ydb.TopicReaderEvents.OnPartitionGetStartOffsetResponse.start_offset`. The function can also be implemented as asynchronous.

Example:

```
class CustomEventHandler(ydb.TopicReaderEvents.EventHandler):
    def on_partition_get_start_offset(self, event: ydb.TopicReaderEvents.OnPartitionGetStartOffsetRequest):
        return ydb.TopicReaderEvents.OnPartitionGetStartOffsetResponse(
            start_offset=0,
        )

reader = driver.topic_client.reader(
    topic=ydb.TopicReaderSelector(
        path="topic-path",
        partitions=[0, 1, 2],
    ),
    consumer=None,
    event_handler=CustomEventHandler(),
)
```

JavaScript

This section is under development.

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#330

Reading in a transaction

C++

Before reading messages, the client code must pass a transaction object reference to the reading session settings.

Example on GitHub

```
readSession->WaitEvent().Wait(TDuration::Seconds(1));

NYdb::NStatusHelpers::ThrowOnError(queryClient.RetryQuerySync([&readSession](NYdb::NQuery::TSession session)
-> NYdb::TStatus {
    auto beginTxResult = session.BeginTransaction(NYdb::Query::TTxSettings::SerializableRW()).GetValueSync();
    if (!beginTxResult.IsSuccess()) {
        return beginTxResult;
    }
    auto tx = beginTxResult.GetTransaction();

    auto topicSettings = NYdb::NTopic::TReadSessionGetEventSettings()
        .Block(false);
        .Tx(tx);

    auto events = readSession->GetEvents(topicSettings);

    for (auto& event : events) {
        // process the event and write results to a table
```

```
    }

    return tx.Commit().GetValueSync();
}));
```

> **Warning**
>
> When processing `events`, you do not need to confirm processing for `TDataReceivedEvent` events explicitly.

Confirmation of the `TStopPartitionSessionEvent` event processing must be done after calling `Commit`.

```
std::optional<NYdb::NTopic::TStopPartitionSessionEvent> stopPartitionSession;

auto events = readSession->GetEvents(topicSettings);

for (auto& event : events) {
    if (auto* e = std::get_if<NYdb::NTopic::TStopPartitionSessionEvent>(&event)) {
        stopPartitionSessionEvent = std::move(*e);
    } else {
        // process the event and write results to a table
    }
}

auto commitResult = tx.Commit(commitSettings).GetValueSync();
if (!commitResult.IsSuccess()) {
    return commitResult;
}

if (stopPartitionSessionEvent) {
    stopPartitionSessionEvent->Commit();
}
```

Go

To read messages from a topic within a transaction, use the Reader.PopMessagesBatchTx method. It reads a batch of messages and adds their commit to the transaction, so there's no need to commit them separately. The reader can be reused across different transactions. However, it's important to commit transactions in the same order as the messages are read from the reader, as message commits in the topic must be performed strictly in order. The simplest way to ensure this is by using the reader within a loop.

Example on GitHub

```
for {
  err := db.Query().DoTx(ctx, func(ctx context.Context, tx query.TxActor) error {
    batch, err := reader.PopMessagesBatchTx(ctx, tx) // the batch will be committed along with the transactio
n
    if err != nil {
      return err
    }

    return processBatch(ctx, batch)
  })
  if err != nil {
    handleError(err)
  }
}
```

Python

To read messages from a topic within a transaction, use the `reader.receive_batch_with_tx` method. It reads a batch of messages and adds their commit to the transaction, so there's no need to commit them separately. The reader can be reused across different transactions. However, it's essential to commit transactions in the same order as the messages are read from the reader, as message commits in the topic must be performed strictly in order - otherwise transaction will get an error during commit. The simplest way to ensure this is by using the reader within a loop.

Native SDK

Example on GitHub

```
with driver.topic_client.reader(topic, consumer) as reader:
    with ydb.QuerySessionPool(driver) as session_pool:
        for _ in range(message_count):

            def callee(tx: ydb.QueryTxContext):
                batch = reader.receive_batch_with_tx(tx, max_messages=1)
                print(f"Message {batch.messages[0].data.decode()} was read with tx.")

            session_pool.retry_tx_sync(callee)
```

Native SDK (Asyncio)

Example on GitHub

```
async with driver.topic_client.reader(topic, consumer) as reader:
    async with ydb.aio.QuerySessionPool(driver) as session_pool:
        for _ in range(message_count):

            async def callee(tx: ydb.aio.QueryTxContext):
```

```
                batch = await reader.receive_batch_with_tx(tx, max_messages=1)
                print(f"Message {batch.messages[0].data.decode()} was read with tx.")

            await session_pool.retry_tx_async(callee)
```

Java

Synchronous API

Example on GitHub

You can pass a transaction in `ReceiveSettings` for `receive`:

```
Message message = reader.receive(ReceiveSettings.newBuilder()
        .setTransaction(transaction)
        .build());
```

The message is then committed together with that transaction; do not commit it separately.
`receive` links message offsets to the transaction on the server via `sendUpdateOffsetsInTransaction` and returns after the response arrives.

Asynchronous API

Example on GitHub

In `onMessages`, you can attach one or more messages to a transaction by calling `reader.updateOffsetsInTransaction`.
Wait for that call to finish before committing the transaction.
The method takes a partition offset list; you can build it manually or use `getPartitionOffsets()` on `Message` or `DataReceivedEvent`.

```
@Override
public void onMessages(DataReceivedEvent event) {
    for (Message message : event.getMessages()) {
        // creating a session in the table service
        Result<Session> sessionResult = tableClient.createSession(Duration.ofSeconds(10)).join();
        if (!sessionResult.isSuccess()) {
            logger.error("Couldn't get a session from the pool: {}", sessionResult);
            return; // retry or shutdown
        }
        Session session = sessionResult.getValue();
        // creating a transaction in the table service
        // this transaction is not yet active and has no id
        TableTransaction transaction = session.createNewTransaction(TxMode.SERIALIZABLE_RW);

        // do something else in the transaction
        transaction.executeDataQuery("SELECT 1").join();
        // now the transaction is active and has an id
        // analyzeQueryResultIfNeeded();

        Status updateStatus = reader.updateOffsetsInTransaction(transaction,
                        message.getPartitionOffsets(), new UpdateOffsetsInTransactionSettings.Builder().build
())
                // Do not commit a transaction without waiting for updateOffsetsInTransaction result to avoid
a race condition
                .join();
        if (!updateStatus.isSuccess()) {
            logger.error("Couldn't update offsets in a transaction: {}", updateStatus);
            return; // retry or shutdown
        }

        Status commitStatus = transaction.commit().join();
        analyzeCommitStatus(commitStatus);
    }
}
```

> **Note**
>
> Transaction requirements:
>
> - It should be an active transaction (that has an id) from one of YDB services. I.e., Table or Query.
> - Only the `SERIALIZABLE_RW` transaction isolation level is supported in the Topic Service.

JavaScript

This section is under development.

Rust

Full example of reading a topic in a transaction with table writes: `topic-read-in-transaction-example.rs`.

```
let batch = reader.pop_batch_in_tx(&mut tx).await?;
// process batch.messages and commit the transaction
```

Processing a server read interrupt

YDB uses server-based partition balancing between clients. This means that the server can interrupt the reading of messages from random partitions.

In case of a *soft interruption*, the client receives a notification that the server has finished sending messages from the partition and messages will no longer be read. The client can finish processing messages and send a commit to the server.

In case of a *hard interruption*, the client receives a notification that it is no longer possible to work with partitions. The client must stop processing the read messages. Uncommited messages will be transferred to another consumer.

Soft reading interruption

C++

The `TStopPartitionSessionEvent` class is used for soft reading interruption. It helps user to stop message processing gracefully.

Example of event loop fragment:

```
auto event = readSession->GetEvent(/*block=*/true);
if (auto* stopPartitionSessionEvent = std::get_if<NYdb::NTopic::TReadSessionEvent::TStopPartitionSessionEvent
>(&*event)) {
    stopPartitionSessionEvent->Confirm();
} else {
  // other event types
}
```

Go

The client code immediately receives all messages from the buffer (on the SDK side) even if they are not enough to form a batch during batch processing.

```
r, _ := db.Topic().StartReader("my-consumer", nil,
    topicoptions.WithBatchReadMinCount(1000),
)

for {
    batch, _ := r.ReadMessageBatch(ctx) // <- if partition soft stop batch can be less, then 1000
    processBatch(batch)
    _ = r.Commit(batch.Context(), batch)
}
```

Python

No special processing is required.

Native SDK

```
while True:
    batch = reader.receive_batch()
    process(batch)
    reader.commit(batch)
```

Native SDK (Asyncio)

```
while True:
    batch = await reader.receive_batch()
    process(batch)
    reader.commit(batch)
```

Java

Synchronous API

Not applicable: the sync reader does not let you customize handling for this event.
The client automatically acknowledges stop to the server.

Asynchronous API

Override `onStopPartitionSession(StopPartitionSessionEvent event)` on your `ReadEventHandler` (see Connecting for reads).
You must call `event.confirm()` so the server can continue shutdown.

```
@Override
public void onStopPartitionSession(StopPartitionSessionEvent event) {
    logger.info("Partition session {} stopped. Committed offset: {}", event.getPartitionSessionId(),
            event.getCommittedOffset());
    // This event means that no more messages will be received by server
    // Received messages still can be read from ReaderBuffer
    // Messages still can be committed, until confirm() method is called

    // Confirm that session can be closed
    event.confirm();
}
```

JavaScript

This section is under development.

Rust

The Rust SDK handles partition session stop and close events internally; there is no public API to customize soft or hard interruption handling yet.

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#330

Hard reading interruption

C++

The hard interruption of reading messages is implemented using an `TPartitionSessionClosedEvent` event. It can be received either as soft interrupt confirmation response, or in the case of lost connection. The user can find out the reason for session closing using the `GetReason` method.

Example of event loop fragment:

```
auto event = readSession->GetEvent(/*block=*/true);
if (auto* partitionSessionClosedEvent = std::get_if<NYdb::NTopic::TReadSessionEvent::TPartitionSessionClosedE
vent>(&*event)) {
    if (partitionSessionClosedEvent->GetReason() == TPartitionSessionClosedEvent::EReason::ConnectionLost) {
        std::cout << "Connection with partition was lost" << std::endl;
    }
} else {
  // other event types
}
```

Go

When reading is interrupted, the message or message batch context is canceled.

```
ctx := batch.Context() // batch.Context() will cancel if partition revoke by server or connection broke
if len(batch.Messages) == 0 {
    return
}

buf := &bytes.Buffer{}
for _, mess := range batch.Messages {
    buf.Reset()
    _, _ = buf.ReadFrom(mess)
    _, _ = io.Copy(buf, mess)
    writeMessagesToDB(ctx, buf.Bytes())
}
```

Python

In this example, processing of messages within the batch will stop if the partition is reassigned during operation. This kind of optimization requires that you run extra code on the client side. In simple cases when processing of reassigned partitions is not a problem, you may skip this optimization.

Native SDK

```
def process_batch(batch):
    for message in batch.messages:
        if not batch.alive:
            return False
        process(message)
    return True

batch = reader.receive_batch()
if process_batch(batch):
    reader.commit(batch)
```

Native SDK (Asyncio)

```
def process_batch(batch):
    for message in batch.messages:
        if not batch.alive:
            return False
        process(message)
    return True

batch = await reader.receive_batch()
if process_batch(batch):
    reader.commit(batch)
```

Java

Synchronous API

Not applicable: the sync reader does not let you customize handling for this event.

Asynchronous API

```
@Override
public void onPartitionSessionClosed(PartitionSessionClosedEvent event) {
```

```
    logger.info("Partition session {} is closed.", event.getPartitionSession().getPartitionId());
}
```

JavaScript

This section is under development.

Rust

The Rust SDK handles partition session stop and close events internally; there is no public API to customize soft or hard interruption handling yet.

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#330

Topic autoscaling

Go

Autoscaling of a topic can be enabled during its creation using the `topicoptions.CreateWithAutoPartitioningSettings` option:

```
import (
  ...

  "github.com/ydb-platform/ydb-go-sdk/v3/topic/topicoptions"
  "github.com/ydb-platform/ydb-go-sdk/v3/topic/topictypes"
)

err := db.Topic().Create(ctx,
  "topic",
  topicoptions.CreateWithAutoPartitioningSettings(
    topictypes.AutoPartitioningSettings{
      AutoPartitioningStrategy: topictypes.AutoPartitioningStrategyScaleUp,
    },
  ),
)
```

When needed, you can set additional parameters in AutoPartitioningSettings:

```
err := db.Topic().Create(ctx,
  "topic",
  topicoptions.CreateWithAutoPartitioningSettings(
    topictypes.AutoPartitioningSettings{
      AutoPartitioningStrategy: topictypes.AutoPartitioningStrategyScaleUp,
      AutoPartitioningWriteSpeedStrategy: topictypes.AutoPartitioningWriteSpeedStrategy{
        StabilizationWindow:    time.Minute,
        UpUtilizationPercent:   80,
      },
    },
  ),
)
```

Changes to an existing topic can be made using the `topicoptions.AlterWithAutoPartitioningStrategy` option with `.Topic().Alter`:

```
import (
  ...

  "github.com/ydb-platform/ydb-go-sdk/v3/topic/topicoptions"
  "github.com/ydb-platform/ydb-go-sdk/v3/topic/topictypes"
)

err := db.Topic().Alter(
  ctx,
  "topic",
  topicoptions.AlterWithAutoPartitioningStrategy(
    topictypes.AutoPartitioningStrategyScaleUp,
  ),
)

// other options
err := db.Topic().Alter(
  ctx,
  "topic",
  topicoptions.AlterWithAutoPartitioningStrategy(
    topictypes.AutoPartitioningStrategyScaleUp,
  ),
  topicoptions.AlterWithAutoPartitioningWriteSpeedStabilizationWindow(time.Minute),
  topicoptions.AlterWithAutoPartitioningWriteSpeedUpUtilizationPercent(80),
)
```

The SDK supports two topic reading modes with autoscaling enabled: full support mode and compatibility mode. The reading mode is set using the `topicoptions.WithReaderSupportSplitMergePartitions` option when creating the reader. Full support mode is used by default (`true`).

From a practical perspective, these modes do not differ for the end user. However, the full support mode differs from the compatibility mode in terms of who guarantees the order of reading—the client or the server. Compatibility mode is achieved through server-side processing and generally operates slower.

```
import (
  ...

  "github.com/ydb-platform/ydb-go-sdk/v3/topic/topicoptions"
  "github.com/ydb-platform/ydb-go-sdk/v3/topic/topictypes"
)

// full support mode (autoscaling processing in SDK, by default)
reader, err := db.Topic().StartReader(
  "consumer",
  topicoptions.ReadTopic("topic"),
  topicoptions.WithReaderSupportSplitMergePartitions(true),
)

// compatibility mode (autoscaling processing on server)
reader, err := db.Topic().StartReader(
  "consumer",
  topicoptions.ReadTopic("topic"),
  topicoptions.WithReaderSupportSplitMergePartitions(false),
)
```

Python

Autoscaling of a topic can be enabled during its creation using the `auto_partitioning_settings` argument of `create_topic`:

Native SDK

```
driver.topic_client.create_topic(
    topic,
    consumers=[consumer],
    min_active_partitions=10,
    max_active_partitions=100,
    auto_partitioning_settings=ydb.TopicAutoPartitioningSettings(
        strategy=ydb.TopicAutoPartitioningStrategy.SCALE_UP,
        up_utilization_percent=80,
        down_utilization_percent=20,
        stabilization_window=datetime.timedelta(seconds=300),
    ),
)
```

Native SDK (Asyncio)

```
await driver.topic_client.create_topic(
    topic,
    consumers=[consumer],
    min_active_partitions=10,
    max_active_partitions=100,
    auto_partitioning_settings=ydb.TopicAutoPartitioningSettings(
        strategy=ydb.TopicAutoPartitioningStrategy.SCALE_UP,
        up_utilization_percent=80,
        down_utilization_percent=20,
        stabilization_window=datetime.timedelta(seconds=300),
    ),
)
```

Changes to an existing topic can be made using the `alter_auto_partitioning_settings` argument of `alter_topic`:

```
driver.topic_client.alter_topic(
    topic_path,
    alter_auto_partitioning_settings=ydb.TopicAlterAutoPartitioningSettings(
        set_strategy=ydb.TopicAutoPartitioningStrategy.SCALE_UP,
        set_up_utilization_percent=80,
        set_down_utilization_percent=20,
        set_stabilization_window=datetime.timedelta(seconds=300),
    ),
)
```

The SDK supports two topic reading modes with autoscaling enabled: full support mode and compatibility mode. The reading mode can be set in the `auto_partitioning_support` argument when creating the reader. Full support mode is used by default.

```
reader = driver.topic_client.reader(
    topic,
    consumer,
    auto_partitioning_support=True, # Full support is enabled
)

# or

reader = driver.topic_client.reader(
    topic,
    consumer,
    auto_partitioning_support=False, # Compatibility mode is enabled
)
```

From a practical perspective, these modes do not differ for the end user. However, the full support mode differs from the compatibility mode in terms of who guarantees the order of reading—the client or the server. Compatibility mode is achieved through server-side

processing and generally operates slower.

JavaScript

This section is under development.

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#311

Commit outside the reader

Most often, committing is conveniently done within the reader that has read the messages. However, there are scenarios where committing needs to be performed by a separate process. In such cases, a method of committing outside the reader is necessary.

Go

Committing outside the reader is done with `db.Topic().CommitOffset`:

```
// Basic usage — commit an offset without an active read session
err := db.Topic().CommitOffset(
  ctx,
  topicPath,
  partitionID,
  consumer,
  offset,
)
```

If a read session is active when you commit (via `StartReader` or `StartListener`), pass its ID with `WithCommitOffsetReadSessionID` so the server does not interrupt the current read session:

```
import (
  // ...
  "github.com/ydb-platform/ydb-go-sdk/v3/topic/topicoptions"
)

// Read session ID
sessionID := reader.ReadSessionID()
// or: sessionID := listener.ReadSessionID()

err = db.Topic().CommitOffset(
  ctx,
  topicPath,
  partitionID,
  consumer,
  offset,
  topicoptions.WithCommitOffsetReadSessionID(sessionID),
)
```

Python

Committing outside the reader is done with the `topic_client.commit_offset` method:

Native SDK

```
driver.topic_client.commit_offset(
    topic_path,
    consumer_name,
    partition_id,
    offset,
    reader.read_session_id,  # optional: avoids interrupting the active read session
)
```

Native SDK (Asyncio)

```
await driver.topic_client.commit_offset(
    topic_path,
    consumer_name,
    partition_id,
    offset,
    reader.read_session_id,  # optional: avoids interrupting the active read session
)
```

JavaScript

This section is under development.

Java

```
TopicClient client = ...;

String sessionID = reader.getSessionId();
// For AsyncReader, the session ID can be obtained when handling SessionStartedEvent

client.commitOffset(
    topicPath,
```

```
    CommitOffsetSettings.newBuilder()
        .setReadSessionId(sessionID)
        .setPartitionId(partitionID)
        .setConsumer(consumer)
        .setOffset(offset)
        .build()
).join().expectSuccess("Error commit!");
```

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#330

Working with coordination nodes

This article describes how to use the YDB SDK to coordinate the work of multiple client application instances using coordination nodes and their semaphores.

Creating a coordination node

Coordination nodes are created in YDB databases in the same namespace as other schema objects, such as tables and topics.

Go

```
err := db.Coordination().CreateNode(ctx,
    "/path/to/mynode",
)
```

C++

```
TClient client(driver);
auto status = client
    .CreateNode("/path/to/mynode")
    .ExtractValueSync();
Y_ABORT_UNLESS(status.IsSuccess());
```

When creating a node, you can optionally specify `TNodeSettings` with the following settings:

- `ReadConsistencyMode` - defaults to `RELAXED`, allowing the reading of potentially outdated values during leader transitions. You can optionally enable the `STRICT` mode, where all reads are processed through the consensus algorithm, ensuring the most recent value is returned, albeit at a higher cost.
- `AttachConsistencyMode` - defaults to `STRICT`, requiring the consensus algorithm for session recovery. Optionally, the `RELAXED` mode can be enabled for session recovery during failures, bypassing this requirement. This mode may be necessary for a large number of clients, facilitating session recovery without consensus, which maintains overall correctness but may lead to outdated reads during leader transitions and session expiration in problematic scenarios.
- `SelfCheckPeriod` (default 1 second) - the interval at which the service performs self-liveness checks. It is not recommended to change this setting except under special circumstances.
  - A larger value reduces server load but increases the delay in detecting leader changes and informing the service.
  - A smaller value increases server load and improves problem detection speed, but may result in false positives when the service incorrectly identifies issues.
- `SessionGracePeriod` (default 10 seconds) - the duration during which a new leader refrains from closing existing open sessions, prolonging their validity.
  - A smaller value reduces the window during which sessions from non-existent clients, which failed to report their absence during leader changes, hold semaphores and block other clients.
  - A smaller value also increases the likelihood of false positives, where a living leader might terminate itself as a precaution, uncertain if this period has concluded on a potential new leader.
  - This value must be strictly greater than `SelfCheckPeriod`.

Java

```
CoordinationClient client = CoordinationClient.newClient(transport);
```

Create a node by calling `createNode` with the full path to the node in the database. You can take the database path prefix from `client.getDatabase()`.

When needed, configure the node via NodeConfig using the `NodeConfig.create().with…` chain. Available parameters: `SelfCheckPeriod` and `SessionGracePeriod`, read and attach consistency modes (`readConsistencyMode`, `attachConsistencyMode`), and rate limiter counters mode (`rateLimiterCountersMode`). Defaults match the C++ description above. Pass the resulting `NodeConfig` to `CoordinationNodeSettings`.

```
import java.time.Duration;

import tech.ydb.coordination.CoordinationClient;
import tech.ydb.coordination.description.NodeConfig;
import tech.ydb.coordination.settings.CoordinationNodeSettings;

String nodePath = client.getDatabase() + "/path/to/mynode";

NodeConfig config = NodeConfig.create()
    .withDurationsConfig(Duration.ofSeconds(1), Duration.ofSeconds(10))
    .withReadConsistencyMode(NodeConfig.ConsistencyMode.RELAXED)
    .withAttachConsistencyMode(NodeConfig.ConsistencyMode.STRICT);

CoordinationNodeSettings settings = CoordinationNodeSettings.newBuilder()
    .withNodeConfig(config)
    .build();

client.createNode(nodePath, settings).join().expectSuccess("create node failed");
```

You can also use `alterNode` (change configuration), `dropNode` (delete the node), and `describeNode` (read the current configuration).

Python

Native SDK

```
import ydb
```

```
client = driver.coordination_client
client.create_node("/path/to/mynode")
```

Native SDK (Asyncio)

```
import ydb

client = driver.coordination_client
await client.create_node("/path/to/mynode")
```

JavaScript

```
import { CoordinationClient } from "@ydbjs/coordination";

let client = new CoordinationClient(driver);
await client.createNode("/path/to/mynode", {});
```

Working with sessions

Creating a session

To start working with coordination nodes, a client must establish a session within which it will perform all operations with the coordination node.

Go

```
session, err := db.Coordination().CreateSession(ctx,
    "/path/to/mynode", // Coordination Node name in the database
)
```

C++

```
TClient client(driver);
const TSession& session = client
   .StartSession("/path/to/mynode")
   .ExtractValueSync()
   .ExtractResult();
```

When establishing a session, you can optionally pass a `TSessionSettings` structure with the following settings:

- `Description` - a text description of the session, displayed in internal interfaces and can be useful for problem diagnosis.
- `OnStateChanged` - called on significant changes during the session's life, passing the corresponding state:
  - `ATTACHED` - the session is connected and operating in normal mode.
  - `DETACHED` - the session temporarily lost connection with the service but can still be restored.
  - `EXPIRED` - the session lost connection with the service and cannot be restored.
- `OnStopped` - called when the session stops attempting to restore the connection with the service, which can be useful for establishing a new connection.
- `Timeout` - the maximum timeout during which the session can be restored after losing connection with the service.

Java

A session (see CoordinationSession) is created with `createSession`; call `connect()` to establish the bidirectional gRPC stream with the node (asynchronous, returns `CompletableFuture<Status>`). Retry behavior and connect timeout are configured in CoordinationSessionSettings (`withConnectTimeout`, `withRetryPolicy`, `withExecutor`).

```
import tech.ydb.coordination.CoordinationSession;
import tech.ydb.coordination.settings.CoordinationSessionSettings;

CoordinationSession session = client.createSession(
    "/path/to/mynode",
    CoordinationSessionSettings.newBuilder().build()
);

session.connect().join().expectSuccess("connect failed");
```

Typical flow: after a successful `connect()`, run semaphore operations for your scenario (locking, leadership, and so on), then close the session. Using try-with-resources is convenient so that `close()` stops the stream to the node when you are done.

Python

Native SDK

```
import ydb

client = driver.coordination_client
with client.session("/path/to/mynode") as session:
    # work with the session
    pass
```

Native SDK (Asyncio)

```
import ydb

client = driver.coordination_client
async with client.session("/path/to/mynode") as session:
    # work with the session
    pass
```

JavaScript

```
import { CoordinationClient } from "@ydbjs/coordination";

let client = new CoordinationClient(driver);
await using session = await client.createSession("/path/to/mynode", {}, signal);
```

Session control

It's important for the client application to monitor the session state, as it can only rely on the state of captured semaphores while the session is alive. When the session ends by client or server initiative, the client can no longer assume that other clients in the cluster haven't captured its semaphores and changed their state.

Go

In Go SDK, the session context `session.Context()` is used to track such situations, which terminates along with the session. The SDK can handle transport-level errors on its own and re-establish connection with the service, trying to restore the session if still possible. Thus, the client only needs to monitor the session context to react timely to its loss.

C++

In the C++ SDK, an active session continuously maintains and automatically re-establishes the connection with the YDB cluster in the background.

Python

In the Python SDK, the session automatically restores the connection to the YDB cluster after failures. Use a context manager (`with` or `async with`) to ensure the session is closed when leaving the block. When working with semaphores through a context manager (`with session.semaphore(name)` or `async with session.semaphore(name)`), the semaphore is released automatically when leaving the block, and the session is closed when the context exits.

Java

Call `close()` when your scenario is finished; this explicitly releases the connection to the node. Until the session is closed, the SDK retries the connection on network failures according to `CoordinationSessionSettings`. Hold a semaphore only while solving your business task and release it with `SemaphoreLease.release()` when the resource is no longer needed.

JavaScript

In the JavaScript SDK, you can use `session.signal` to monitor session lifetime: it is aborted when the session is closed or expires. The SDK handles transport-level errors and reconnects to the service, attempting to restore the session when possible. This means the client only needs to watch the session signal and avoid any operations once the session is closed or expired.

For long-running applications, the JavaScript SDK provides a recommended pattern to automatically obtain a new session when the previous one is lost: `for await (session of client.openSession()) { session.signal }`

Working with semaphores

Creating a semaphore

When creating a semaphore, you can specify its limit. The limit determines the maximum value to which it can be increased. Calls attempting to increase the semaphore value above this limit will wait until their increase requests can be fulfilled without exceeding the semaphore's limit.

Go

```
err := session.CreateSemaphore(ctx,
    "my-semaphore", // semaphore name
    10              // semaphore limit
)
```

C++

```
session
    .CreateSemaphore(
        "my-semaphore",  // semaphore name
        10               // semaphore limit
    )
    .ExtractValueSync()
    .ExtractResult();
```

You can also pass a string that will be stored with the semaphore and returned when it's captured:

```
session
    .CreateSemaphore(
        "my-semaphore",  // semaphore name
        10,              // semaphore limit
        "my-data"        // semaphore data
```

```
    )
    .ExtractValueSync()
    .ExtractResult();
```

Python

In the Python SDK, a semaphore is created implicitly on the first `acquire()` call in `session.semaphore(name, limit)`. The limit is set when the semaphore object is created.

Native SDK

```
import ydb

client = driver.coordination_client
with client.session("/path/to/mynode") as session:
    # semaphore is created on first acquire() with limit 10
    semaphore = session.semaphore("my-semaphore", 10)
```

Native SDK (Asyncio)

```
import ydb

client = driver.coordination_client
async with client.session("/path/to/mynode") as session:
    # semaphore is created on first acquire() with limit 10
    semaphore = session.semaphore("my-semaphore", 10)
```

JavaScript

```
const sem = session.semaphore("connections");
await sem.create({
  limit: 10,
  data: new Uint8Array(),
});
```

Java

Create a semaphore explicitly with `createSemaphore` on a connected session. You can pass custom binary data stored with the semaphore (`byte[] data`); the overload without `data` is equivalent to passing `null` . If a semaphore with that name already exists, the operation completes with an “already exists” status.

```
session.createSemaphore("my-semaphore", 10, new byte[] {0x00, 0x12})
    .join()
    .expectSuccess("create semaphore failed");
```

Acquiring a semaphore

To acquire a semaphore, the client must call the `AcquireSemaphore` method and wait for a special `Lease` object. This object represents confirmation that the semaphore value was successfully increased and can be considered as such until explicit release of such semaphore or termination of the session in which such confirmation was received.

Go

```
lease, err := session.AcquireSemaphore(ctx,
    "my-semaphore",  // semaphore name
    5,              // value to increase semaphore by
)
```

Similar to the session, the `Lease` object also has a context that terminates at one of these moments.

To cancel waiting for semaphore acquisition, it's sufficient to cancel the passed context `ctx` .

C++

```
session
    .AcquireSemaphore(
        "my-semaphore",                      // semaphore name
        TAcquireSemaphoreSettings().Count(5)  // value to increase semaphore by
    )
    .ExtractValueSync()
    .ExtractResult();
```

When acquiring, you can optionally pass a `TAcquireSemaphoreSettings` structure with the following settings:

- `Count` - value by which the semaphore is increased upon acquisition.
- `Data` - additional data that can be put into the semaphore.
- `OnAccepted` - called when the operation is queued. For example, if the semaphore couldn't be acquired immediately.
  - Won't be called if the semaphore is acquired immediately.
  - It's important to consider that the call can happen in parallel with the `TFuture` result.
- `Timeout` - maximum time during which the operation can stay in the queue on the server.
  - Operation will return `false` if the semaphore couldn't be acquired within `Timeout` after being added to the queue.

    - With `Timeout` set to 0, the operation works like `TryAcquire`, i.e., the semaphore will either be acquired atomically and the operation will return `true`, or the operation will return `false` without using queues.
- `Ephemeral` - if `true`, then the name is an ephemeral semaphore. Such semaphores are automatically created at first `Acquire` and automatically deleted with the last `Release`.
- `Shared()` - alias for setting `Count = 1`, acquiring semaphore in shared mode.
- `Exclusive()` - alias for setting `Count = max`, acquiring semaphore in exclusive mode. (For semaphores created with limit `Max<ui64>()`.)

Python

Native SDK

```
import ydb

client = driver.coordination_client
with client.session("/path/to/mynode") as session:
    semaphore = session.semaphore("my-semaphore", 10)
    with semaphore:
        # semaphore acquired for 1 unit (default)
        pass
    # or manually:
    semaphore = session.semaphore("my-semaphore", 10)
    semaphore.acquire(count=5)
    # work with the resource
    semaphore.release()
```

Native SDK (Asyncio)

```
import ydb

client = driver.coordination_client
async with client.session("/path/to/mynode") as session:
    semaphore = session.semaphore("my-semaphore", 10)
    async with semaphore:
        # semaphore acquired for 1 unit (default)
        pass
    # or manually:
    semaphore = session.semaphore("my-semaphore", 10)
    await semaphore.acquire(count=5)
    # work with the resource
    await semaphore.release()
```

JavaScript

```
{
  await using lease = await sem.acquire({ count: 1, data: new Uint8Array() });
  await doWork(lease.signal);
} // lease.release() called automatically
```

Java

Acquisition is done via `acquireSemaphore` with the semaphore name, token `count`, optional operation data, and queue wait timeout as java.time.Duration. The method returns `CompletableFuture<Result<SemaphoreLease>>` (see Result and SemaphoreLease). If no semaphore exists with the given name, the operation fails with an exception (see the method javadoc).

```
import java.time.Duration;

import tech.ydb.coordination.SemaphoreLease;
import tech.ydb.core.Result;

Result<SemaphoreLease> result = session
    .acquireSemaphore("my-semaphore", 5, Duration.ofSeconds(30))
    .join();

result.getStatus().expectSuccess("cannot acquire semaphore");
SemaphoreLease lease = result.getValue();
```

For **ephemeral** semaphores, use `acquireEphemeralSemaphore` (the `exclusive` flag sets the acquisition mode); such semaphores are created on first acquire and removed after the last release.

The API documentation states that a session can hold **only one** semaphore at a time; repeated calls for the same name **replace** the previous operation (for example, to reduce `count` or change the timeout).

The taken value of an acquired semaphore can be decreased (but not increased) by calling the `AcquireSemaphore` method again with a smaller value.

Updating semaphore data

Using the `UpdateSemaphore` method, you can update (replace) the semaphore data that was attached during its creation.

Go

```
err := session.UpdateSemaphore(
    "my-semaphore",                                 // semaphore name
```

```
    options.WithUpdateData([]byte("updated-data")),   // new semaphore data
)
```

C++

```
session
    .UpdateSemaphore(
        "my-semaphore",  // semaphore name
        "updated-data"   // new semaphore data
    )
    .ExtractValueSync()
    .ExtractResult();
```

Python

Native SDK

```
import ydb

client = driver.coordination_client
with client.session("/path/to/mynode") as session:
    semaphore = session.semaphore("my-semaphore", 10)
    semaphore.update(b"updated-data")
```

Native SDK (Asyncio)

```
import ydb

client = driver.coordination_client
async with client.session("/path/to/mynode") as session:
    semaphore = session.semaphore("my-semaphore", 10)
    await semaphore.update(b"updated-data")
```

Java

```
session.updateSemaphore("my-semaphore", "updated-data".getBytes(java.nio.charset.StandardCharsets.UTF_8))
    .join()
    .expectSuccess("update semaphore failed");
```

JavaScript

```
const sem = session.semaphore("connections");
await sem.update({
  limit: 5,
  data: new Uint8Array(),
});
```

This call doesn't require acquiring the semaphore and doesn't lead to it. If you need only one specific client to update the data, this must be explicitly ensured, for example, by acquiring the semaphore, updating the data, and releasing the semaphore back.

Getting semaphore data

Go

```
description, err := session.DescribeSemaphore(
    "my-semaphore"                                  // semaphore name
    options.WithDescribeOwners(true), // to get list of owners
    options.WithDescribeWaiters(true), // to get list of waiters
)
```

C++

```
session
    .DescribeSemaphore(
        "my-semaphore"  // semaphore name
    )
    .ExtractValueSync()
    .ExtractResult();
```

When getting information about a semaphore, you can optionally pass a `TDescribeSemaphoreSettings` structure with the following settings:

- `OnChanged` - called once after data changes on the server (with a `bool` parameter, if `true` - the call occurred due to some changes, if `false` - it's a false call and you need to repeat `DescribeSemaphore` to restore the subscription).
- `WatchData` - call `OnChanged` when semaphore data changes.
- `WatchOwners` - call `OnChanged` when semaphore owners change.
- `IncludeOwners` - return the list of owners in the results.
- `IncludeWaiters` - return the list of waiters in the results.

The call result is a structure with the following fields:

- `Name` - semaphore name.

- `Data` - semaphore data.
- `Count` - the current value of the semaphore.
- `Limit` - the limit specified when creating the semaphore.
- `Owners` - list of semaphore owners.
- `Waiters` - list of clients waiting in the semaphore queue.
- `Ephemeral` - whether the semaphore is ephemeral.

The `Owners` and `Waiters` fields in the result contain a list of structures with the following fields:

- `OrderId` - sequence number of the acquire operation on the semaphore (can be used for identification, for example if `OrderId` changed, it means the session called `ReleaseSemaphore` and a new `AcquireSemaphore`).
- `SessionId` - identifier of the session that made this `AcquireSemaphore` call.
- `Timeout` - timeout value used in the `AcquireSemaphore` call for queued operations.
- `Count` - value requested in `AcquireSemaphore`.
- `Data` - data specified in `AcquireSemaphore`.

Python

Native SDK

```
import ydb

client = driver.coordination_client
with client.session("/path/to/mynode") as session:
    semaphore = session.semaphore("my-semaphore", 10)
    description = semaphore.describe()
    # description contains: name, data, count, limit, owners, waiters, ephemeral
```

Native SDK (Asyncio)

```
import ydb

client = driver.coordination_client
async with client.session("/path/to/mynode") as session:
    semaphore = session.semaphore("my-semaphore", 10)
    description = await semaphore.describe()
    # description contains: name, data, count, limit, owners, waiters, ephemeral
```

Java

The `describeSemaphore` method takes a semaphore name and a DescribeSemaphoreMode: data only, with owners list, with waiters list, or both.

```
import tech.ydb.coordination.description.SemaphoreDescription;
import tech.ydb.coordination.settings.DescribeSemaphoreMode;

SemaphoreDescription description = session
    .describeSemaphore("my-semaphore", DescribeSemaphoreMode.WITH_OWNERS_AND_WAITERS)
    .join()
    .getValue();
```

Owner and waiter list entries (`getOwnersList`, `getWaitersList`) expose session id, timeout, requested `count`, operation data, and `orderId` (see the nested `SemaphoreDescription.Session` type in the sources).

To subscribe to changes, use `watchSemaphore` with the same describe mode and WatchSemaphoreMode (data, owners, or both). SemaphoreWatcher holds a `SemaphoreDescription` snapshot and `getChangedFuture()` — `CompletableFuture<Result<SemaphoreChangedEvent>>` (see SemaphoreChangedEvent, fields `isDataChanged`, `isOwnersChanged`). The future completes on the next event; call `watchSemaphore` again to continue watching (see tests).

JavaScript

```
const sem = session.semaphore("connections");
await sem.describe({
  owners: true,
  waiters: true,
});
```

Releasing a semaphore

Go

To release a semaphore acquired in a session, call the `Release` method on the `Lease` object.

```
err := lease.Release()
```

C++

```
session
    .ReleaseSemaphore(
        "my-semaphore"  // semaphore name
    )
```

```
    .ExtractValueSync()
    .ExtractResult();
```

Python

In the Python SDK, a semaphore is released with the `release()` method on the semaphore object. When using a context manager (`with` or `async with`), release happens automatically when leaving the block.

Native SDK

```
import ydb

client = driver.coordination_client
with client.session("/path/to/mynode") as session:
    semaphore = session.semaphore("my-semaphore", 10)
    semaphore.acquire(count=5)
    # work with the resource
    semaphore.release()
```

Native SDK (Asyncio)

```
import ydb

client = driver.coordination_client
async with client.session("/path/to/mynode") as session:
    semaphore = session.semaphore("my-semaphore", 10)
    await semaphore.acquire(count=5)
    # work with the resource
    await semaphore.release()
```

Java

Release via SemaphoreLease.release() (asynchronous, `CompletableFuture<Status>`).

```
lease.release().join().expectSuccess("release failed");
```

JavaScript

To release a semaphore acquired within a session, call `release()` on the `Lease` object. If you use `await using`, the lease is released automatically when leaving the scope.

```
await lease.release();
```

Important implementation details

The `AcquireSemaphore` and `ReleaseSemaphore` operations are idempotent. When `AcquireSemaphore` is invoked on a semaphore, subsequent calls to `AcquireSemaphore` only adjust the acquisition parameters. For instance, if `AcquireSemaphore` is called with `count=10`, the operation might be queued. You can call `AcquireSemaphore` again with `count=9` before or after successful acquisition, reducing the number of acquired units. The new operation replaces the old one, which will complete with an `ABORTED` code if it hasn't completed successfully yet. The queue position remains unchanged despite replacing one `AcquireSemaphore` operation with another.

Both `AcquireSemaphore` and `ReleaseSemaphore` operations return a `bool` indicating whether the semaphore state was altered. For example, `AcquireSemaphore` might return `false` if the semaphore couldn't be acquired within the `Timeout` period because it was acquired by another entity. Similarly, `ReleaseSemaphore` might return `false` if the semaphore isn't acquired within the current session.

A queued `AcquireSemaphore` operation can be prematurely terminated by calling `ReleaseSemaphore`. Regardless of how many `AcquireSemaphore` calls are made for a specific semaphore within one session, a single `ReleaseSemaphore` call releases it. Thus, `AcquireSemaphore` and `ReleaseSemaphore` operations cannot function as `Acquire` / `Release` on a recursive mutex.

The `DescribeSemaphore` operation with `WatchData` or `WatchOwners` flags set establishes a subscription for semaphore changes. Any previous subscription to the same semaphore within the session is canceled, triggering `OnChanged(false)`. It is advisable to disregard `OnChanged` from earlier `DescribeSemaphore` calls if a new replacing call is made, for instance, by tracking a current call ID.

The `OnChanged(false)` call might occur not only due to cancellation by a new `DescribeSemaphore` but also for various reasons, such as temporary connection loss between the gRPC client and server, temporary connection loss between the gRPC server and the current service leader, or service leader changes. This happens at the slightest suspicion that a notification might have been lost. To restore the subscription, client code must issue a new `DescribeSemaphore` call, properly managing the situation where the result of the new call might differ (for example, if the notification was indeed lost).

Examples

- Distributed lock
- Leader election
- Service discovery
- Configuration publication

Handling errors

You need to handle errors properly when using the YDB SDK.

Errors can be divided into three categories:

- **Temporary failures** (retryable). Such errors include a short-term loss of network connectivity, temporary unavailability, overload of a YDB subsystem, or a failure of YDB to respond to a query within the set timeout. If one of these errors occurs, retrying the failed query is likely to be successful after some time.
- **Errors that cannot be fixed with a retry** (non-retryable). Such errors are caused by incorrectly written queries, YDB internal errors, or queries that mismatch the data schema. Retrying such queries will not resolve the issue. This situation requires developer attention.
- **Errors that can presumably be fixed with a retry after the client application response** (conditionally retryable). Such errors include no response within the set timeout or an authentication request. Only idempotent operations can be fixed with a retry.

Handling retryable errors

The YDB SDK provides a built-in mechanism for handling temporary failures. By default, the SDK uses the recommended retry policy, which can be changed to meet the requirements of the client application. YDB returns status codes that let you determine whether a retry is appropriate and which interval to select.

You should retry an operation only if an error refers to a temporary failure. Do not retry invalid operations, such as inserting a row with an existing primary key value into a table or inserting data that mismatches the table schema.

It is extremely important to optimize the number of retries and the interval between them. An excessive number of retries and too short an interval between them result in excessive load. An insufficient number of retries prevents the operation from completing.

The built-in retry mechanisms in YDB SDKs use the following backoff strategies depending on the returned status code:

- **Instant retry** – Retries are made immediately.
- **Fast exponential backoff** – The initial interval is several milliseconds. For each subsequent attempt, the interval increases exponentially.
- **Slow exponential backoff** – The initial interval is several seconds. For each subsequent attempt, the interval increases exponentially.

When selecting an interval manually, the following strategies are usually used:

- **Exponential backoff** – For each subsequent attempt, the interval increases exponentially.
- **Intervals in increments** – For each subsequent attempt, the interval increases in certain increments.
- **Constant intervals** – Retries are made at the same intervals.
- **Instant retry** – Retries are made immediately.
- **Random selection** – Retries are made after a randomly selected time interval.

When selecting an interval and the number of retries, consider the YDB status codes.

Do not use endless retries, as this may result in excessive load.

Do not repeat instant retries more than once.

For code samples, see Retrying.

Status codes

When an error occurs, the YDB SDK returns an error object that includes status codes. The returned status code may come from the YDB server, gRPC transport, or the SDK itself.

Status codes within the range of 400000-400999 are YDB server codes that are identical for all YDB SDKs. Refer to Status codes from the YDB server.

Status codes within the range of 401000-401999 are SDK-specific. For more information about SDK-specific codes, refer to the corresponding SDK documentation.

For more information about gRPC status codes, see the gRPC documentation.

Logging errors

When using the SDK, we recommend logging all errors and exceptions:

- Log the number of retries made. An increase in the number of regular retries often indicates issues.
- Log all errors, including their types, termination codes, and causes.
- Log the total operation execution time, including operations that terminate after retries.

See also

- gRPC status codes
- YDB server status codes
- Questions and answers: Errors

Status codes from the YDB server

Code	Status	Retryability	Backoff strategy	Recreate session
400000	SUCCESS	–	–	–
400010	BAD_REQUEST	non-retryable[2]	–	no
400020	UNAUTHORIZED	non-retryable[2]	–	no
400030	INTERNAL_ERROR	non-retryable[2]	–	no
400040	ABORTED	retryable[3]	fast[4]	no
400050	UNAVAILABLE	retryable[3]	fast[4]	no
400060	OVERLOADED	retryable[3]	slow[5]	no
400070	SCHEME_ERROR	non-retryable[2]	–	no
400080	GENERIC_ERROR	non-retryable[2]	–	no
400090	TIMEOUT	non-retryable[2]	–	no
400100	BAD_SESSION	retryable[3]	instant[6]	yes
400120	PRECONDITION_FAILED	non-retryable[2]	–	no
400130	ALREADY_EXISTS	non-retryable[2]	–	no
400140	NOT_FOUND	non-retryable[2]	–	no
400150	SESSION_EXPIRED	retryable[3]	instant[6]	yes
400160	CANCELLED	non-retryable[2]	–	no
400170	UNDETERMINED	conditionally-retryable[7]	fast[4]	no
400180	UNSUPPORTED	non-retryable[2]	–	no
400190	SESSION_BUSY	retryable[3]	fast[4]	yes
400200	EXTERNAL_ERROR	non-retryable[2]	–	no

400000: SUCCESS

The query was processed successfully.

No response is required. Continue application execution.

400010: BAD_REQUEST Non-retryable

Invalid query syntax or missing required fields.

Correct the query.

400020: UNAUTHORIZED Non-retryable

Access to the requested schema object (for example, a table or directory) is denied.

Request access from its owner.

400030: INTERNAL_ERROR Non-retryable

An unknown internal error occurred.

File a GitHub issue or contact YDB technical support.

400040: ABORTED Retryable | Fast Backoff

The operation was aborted. Possible reasons might include lock invalidation with `TRANSACTION_LOCKS_INVALIDATED` in detailed error messages.

Retry the entire transaction.

400050: UNAVAILABLE Retryable | Fast Backoff

A part of the system is not available.

Retry the last action (query).

400060: OVERLOADED Retryable | Slow Backoff

A part of the system is overloaded.

Retry the last action (query) and reduce the query rate.

400070: SCHEME_ERROR Non-retryable

The query does not match the schema.

Correct the query or schema.

400080: GENERIC_ERROR Non-retryable

An unclassified error occurred, possibly related to the query.

See the detailed error message. If necessary, file a GitHub issue or contact YDB technical support.

400090: TIMEOUT Conditionally retryable | Instant

The query timeout expired.

If the query is idempotent, retry it.

400100: BAD_SESSION Retryable | Instant

This session is no longer available.

Create a new session.

400120: PRECONDITION_FAILED Non-retryable

The query cannot be executed in the current state. For example, inserting data into a table with an existing key.

Correct the state or query, then retry.

400130: ALREADY_EXISTS Non-retryable

The database object being created already exists in the YDB cluster.

The response depends on the application logic.

400140: NOT_FOUND Non-retryable

The database object was not found in the YDB database.

The response depends on the application logic.

400150: SESSION_EXPIRED Conditionally retryable | Instant

The session has already expired.

Create a new session.

400160: CANCELLED Non-retryable

The request was canceled on the server. For example, a user canceled a long-running query in the YDB UI, or the query included the cancel_after timeout option.

If the query took too long to complete, try optimizing it. If you used the `cancel_after` timeout option, increase the timeout value.

400170: UNDETERMINED Conditionally retryable | Fast Backoff

An unknown transaction status. The query ended with a failure, making it impossible to determine the transaction status. Queries that terminate with this status are subject to transaction integrity and atomicity guarantees. That is, either all changes are registered, or the entire transaction is canceled.

For idempotent transactions, retry the entire transaction after a short delay. Otherwise, the response depends on the application logic.

400180: UNSUPPORTED Non-retryable

The query is not supported by YDB either because support for such queries is not yet implemented or is not enabled in the YDB configuration.

Correct the query or enable support for such queries in YDB.

400190: SESSION_BUSY Retryable | Fast Backoff

The session is busy.

Create a new session.

400200: EXTERNAL_ERROR Non-retryable

An error occurred in an external system, for example, when processing a federated query or importing data from an external data source.

See the detailed error message. If necessary, file a GitHub issue or contact YDB technical support.

See also

Questions and answers: Errors

---

[2] **Errors that cannot be fixed with a retry** (non-retryable). Such errors are caused by incorrectly written queries, YDB internal errors, or queries that mismatch the data schema. Retrying such queries will not resolve the issue. This situation requires developer attention.

[3] **Temporary failures** (retryable). Such errors include a short-term loss of network connectivity, temporary unavailability, overload of a YDB subsystem, or a failure of YDB to respond to a query within the set timeout. If one of these errors occurs, retrying the failed query is likely to be successful after some time.

[4] **Fast exponential backoff** is one of the backoff strategies used in YDB SDK when retrying queries that return an error.

The initial interval for this strategy is several **milliseconds**. For each subsequent attempt, the interval increases exponentially.

For more information, see Handling retryable errors.

[5] **Slow exponential backoff** is one of the backoff strategies used by YDB SDK when retrying queries that return an error.

The initial interval for this strategy is several **seconds**. For each subsequent attempt, the interval increases exponentially.

For more information, see Handling retryable errors.

[6] **Instant retry** is one of the backoff strategies used in YDB SDK when retrying queries that return an error.

This strategy retries queries immediately after receiving an error.

For more information, see Handling retryable errors.

[7] **Errors that can presumably be fixed with a retry after the client application response** (conditionally retryable). Such errors include no response within the set timeout or an authentication request. Only idempotent operations can be fixed with a retry.

gRPC status codes

YDB provides the gRPC API, which you can use to manage your database resources and data. The following table describes the gRPC status codes:

Code	Status	Retryability	Backoff strategy	Recreate session
0	OK	–	–	–
1	CANCELLED	conditionally-retryable[8]	fast[9]	yes
2	UNKNOWN	non-retryable[10]	–	yes
3	INVALID_ARGUMENT	non-retryable[10]	–	yes
4	DEADLINE_EXCEEDED	conditionally-retryable[8]	fast[9]	yes
5	NOT_FOUND	non-retryable[10]	–	yes
6	ALREADY_EXISTS	non-retryable[10]	–	yes
7	PERMISSION_DENIED	non-retryable[10]	–	yes
8	RESOURCE_EXHAUSTED	retryable[11]	slow[12]	no
9	FAILED_PRECONDITION	non-retryable[10]	–	yes
10	ABORTED	retryable[11]	instant[13]	yes
11	OUT_OF_RANGE	non-retryable[10]	–	no
12	UNIMPLEMENTED	non-retryable[10]	–	yes
13	INTERNAL	conditionally-retryable[8]	fast[9]	yes
14	UNAVAILABLE	conditionally-retryable[8]	fast[9]	yes
15	DATA_LOSS	non-retryable[10]	–	yes
16	UNAUTHENTICATED	non-retryable[10]	–	yes

0: OK

Not an error; returned on success.

1: CANCELLED Conditionally retryable | Fast Backoff

The operation was cancelled, typically by the caller.

2: UNKNOWN Non-retryable

Unknown error. For example, this error may be returned when a `Status` value received from another address space belongs to an error space that is not known in this address space. Errors raised by APIs that do not return enough error information may also be converted to this error.

3: INVALID_ARGUMENT Non-retryable

The client specified an invalid argument. Unlike `FAILED_PRECONDITION`, `INVALID_ARGUMENT` indicates arguments that are problematic regardless of the system state (e.g., a malformed file name).

4: DEADLINE_EXCEEDED Conditionally retryable | Fast Backoff

The query was not processed within the specified client timeout, or a network issue occurred.

Check the specified timeout, network access, endpoint, and other network settings. Reduce the query rate and optimize queries.

5: NOT_FOUND Non-retryable

A requested scheme object (for example, a table or directory) was not found.

6: ALREADY_EXISTS Non-retryable

The scheme object that a client attempted to create (e.g., file or directory) already exists.

7: PERMISSION_DENIED Non-retryable

The caller does not have permission to execute the specified operation.

8: RESOURCE_EXHAUSTED Retryable | Slow Backoff

There are not enough resources available to fulfill the query.

Reduce the query rate and check client balancing.

9: FAILED_PRECONDITION Non-retryable

The query cannot be executed in the current state (for example, inserting data into a table with an existing key).

Fix the state or query, then retry.

10: ABORTED Retryable | Instant

The operation was aborted, typically due to a concurrency issue, such as a transaction abort.

11: OUT_OF_RANGE Non-retryable

The operation was attempted past the valid range. Unlike `INVALID_ARGUMENT`, this error indicates a problem that may be fixed if the system state changes.

12: UNIMPLEMENTED Non-retryable

The operation is not implemented, supported, or enabled in this service.

13: INTERNAL Conditionally retryable | Fast Backoff

Internal errors. This means that some invariants expected by the underlying system have been broken. This error code is reserved for significant problems.

14: UNAVAILABLE Conditionally retryable | Fast Backoff

The service is currently unavailable. This is most likely a transient condition that can be corrected by retrying with a backoff. Note that it is not always safe to retry non-idempotent operations.

15: DATA_LOSS Non-retryable

Unrecoverable data loss or corruption.

16: UNAUTHENTICATED Non-retryable

The request did not have valid authentication credentials.

Retry the request with valid authentication credentials.

---

[8] **Errors that can presumably be fixed with a retry after the client application response** (conditionally retryable). Such errors include no response within the set timeout or an authentication request. Only idempotent operations can be fixed with a retry.

[9] **Fast exponential backoff** is one of the backoff strategies used in YDB SDK when retrying queries that return an error.

The initial interval for this strategy is several **milliseconds**. For each subsequent attempt, the interval increases exponentially.

For more information, see Handling retryable errors.

[10] **Errors that cannot be fixed with a retry** (non-retryable). Such errors are caused by incorrectly written queries, YDB internal errors, or queries that mismatch the data schema. Retrying such queries will not resolve the issue. This situation requires developer attention.

[11] **Temporary failures** (retryable). Such errors include a short-term loss of network connectivity, temporary unavailability, overload of a YDB subsystem, or a failure of YDB to respond to a query within the set timeout. If one of these errors occurs, retrying the failed query is likely to be successful after some time.

[12] **Slow exponential backoff** is one of the backoff strategies used by YDB SDK when retrying queries that return an error.

The initial interval for this strategy is several **seconds**. For each subsequent attempt, the interval increases exponentially.

For more information, see Handling retryable errors.

[13] **Instant retry** is one of the backoff strategies used in YDB SDK when retrying queries that return an error.

This strategy retries queries immediately after receiving an error.

For more information, see Handling retryable errors.

gRPC API overview

YDB provides the gRPC API, which you can use to manage your DB resources and data. API methods and data structures are described using Protocol Buffers (proto 3). For more information, see .proto specifications with comments. Also YDB uses special gRPC metadata headers.

The following services are available:

- Health Check API.

gRPC metadata headers

YDB uses the following gRPC metadata headers:

- gRPC headers which a client sends to YDB:
  - `x-ydb-database` - database
  - `x-ydb-auth-ticket` - auth token from a credentials provider
  - `x-ydb-sdk-build-info` - YDB SDK build info
  - `x-ydb-trace-id` - user-defined request ID. If not defined by client YDB SDK generates automatically using UUID format
  - `x-ydb-application-name` - optional user-defined application name
  - `x-ydb-client-capabilities` - supported client SDK capabilities (`session-balancer` and other)
  - `x-ydb-client-pid` - client application process ID
  - `traceparent` - OpenTelemetry trace ID (specification)
- gRPC headers which YDB sends to client with responses:
  - `x-ydb-server-hints` - notifications from YDB (such as `session-close` and other)
  - `x-ydb-consumed-units` - consumed units on the current request

Health Check API

YDB has a built-in self-diagnostic system, which can be used to get a brief report on the database status and information about existing issues.

To initiate the check, call the `SelfCheck` method from `NYdb::NMonitoring` namespace in the SDK. You must also pass the name of the checked DB as usual.

C++

App code snippet for creating a client:

```
auto client = NYdb::NMonitoring::TMonitoringClient(driver);
```

Calling `SelfCheck` method:

```
auto settings = TSelfCheckSettings();
settings.ReturnVerboseStatus(true);
auto result = client.SelfCheck(settings).GetValueSync();
```

Go

This functionality is not currently supported.

Java

This functionality is not currently supported.

Python

This functionality is not currently supported.

JavaScript

This functionality is not currently supported in the JavaScript SDK. You can create a monitoring client and call health-check APIs yourself:

```
const monitoring = driver.createClient(MonitoringServiceDefinition);
await monitoring.selfCheck();
```

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#494

Call parameters

`SelfCheck` method provides information in the form of a set of issues which could look like this:

```
{
  "id": "RED-27c3-70fb",
  "status": "RED",
  "message": "Database has multiple issues",
  "location": {
    "database": {
      "name": "/slice"
    }
  },
  "reason": [
    "RED-27c3-4e47",
    "RED-27c3-53b5",
    "YELLOW-27c3-5321"
  ],
  "type": "DATABASE",
  "level": 1
}
```

This is a short messages each about a single issue. All parameters will affect the amount of information the service returns for the specified database.

The complete list of extra parameters is presented below:

C++

```
struct TSelfCheckSettings : public TOperationRequestSettings<TSelfCheckSettings>{
    FLUENT_SETTING_OPTIONAL(bool, ReturnVerboseStatus);
    FLUENT_SETTING_OPTIONAL(EStatusFlag, MinimumStatus);
    FLUENT_SETTING_OPTIONAL(ui32, MaximumLevel);
};
```

Go

This functionality is not currently supported.

Java

This functionality is not currently supported.

Python

This functionality is not currently supported.

JavaScript

This section is under development.

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#494

Parameter	Type	Description
`ReturnVerboseStatus`	`bool`	If `ReturnVerboseStatus` is specified, the response will also include a summary of the overall health of the database in the `database_status` field (Example). Default is false.
`MinimumStatus`	[EStatusFlag] (#issue-status)	Each issue has a `status` field. If `minimum_status` is specified, issues with a higher `status` will be discarded. By default, all issues will be listed.
`MaximumLevel`	`int32`	Each issue has a `level` field. If `maximum_level` is specified, issues with deeper levels will be discarded. By default, all issues will be listed.

Response structure

For the full response structure, see the ydb_monitoring.proto file in the YDB Git repository.
Calling the `SelfCheck` method will return the following message:

```
message SelfCheckResult {
    SelfCheck.Result self_check_result = 1;
    repeated IssueLog issue_log = 2;
    repeated DatabaseStatus database_status = 3;
    LocationNode location = 4;
}
```

The shortest `HealthCheck` response looks like this . It is returned if there is nothing wrong with the database.

If any issues are detected, the `issue_log` field will contain descriptions of the issues with the following structure:

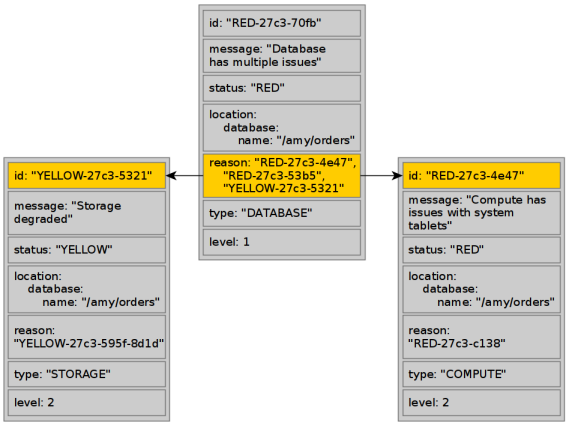
```
message IssueLog {
    string id = 1;
    StatusFlag.Status status = 2;
    string message = 3;
    Location location = 4;
    repeated string reason = 5;
    string type = 6;
    uint32 level = 7;
}
```

Description of fields in the response

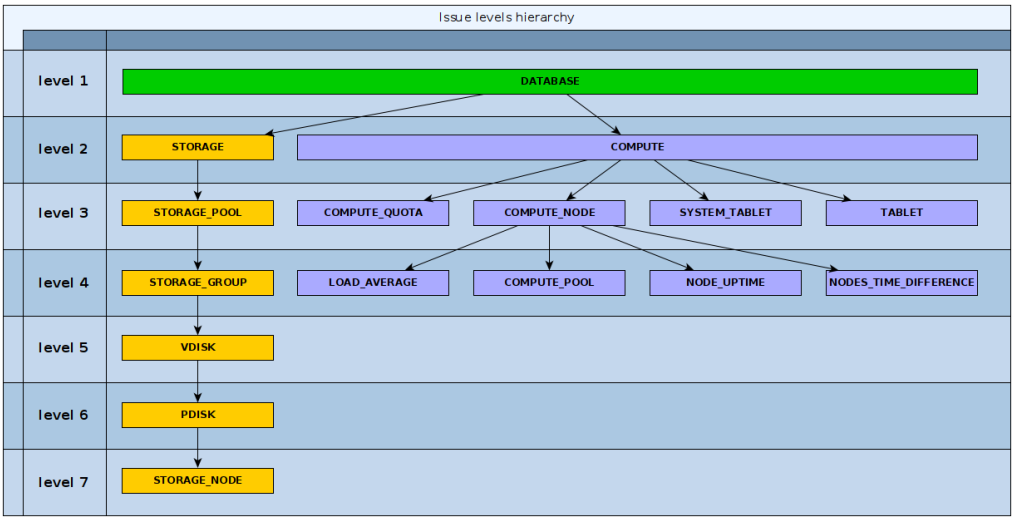
Field	Description
`self_check_result`	enum field which contains the database check result
`issue_log`	A list of issues; each entry describes a problem at a particular level of the system.
`issue_log.id`	A unique issue ID within this response.
`issue_log.status`	enum field which contains the issue status
`issue_log.message`	Text that describes the issue.
`issue_log.location`	Location of the issue. This can be a physical location or an execution context.
`issue_log.reason`	This is a set of elements, each of which describes an issue in the system at a certain level.
`issue_log.type`	Issue category (by subsystem). Each type is at a certain level and interconnected with others through a rigid hierarchy (as shown in the picture above).
`issue_log.level`	Issue nesting depth.
`database_status`	If the settings include `ReturnVerboseStatus` parameter, the `database_status` field will be populated. This field offers a comprehensive summary of the overall health of the database. It is designed to provide a quick overview of the database's condition, helping to assess its health and identify any major issues at a high level. Example. For the full response structure, see the ydb_monitoring.proto file in the YDB Git repository.
`location`	Contains information about the host, where the `HealthCheck` service was called

Issues hierarchy

Issues can be arranged hierarchically using the `id` and `reason` fields, which help visualize how issues in different modules affect the overall system state. All issues are arranged in a hierarchy where higher levels can depend on nested levels:



Each issue has a nesting `level`. The higher the `level`, the deeper the issue is within the hierarchy. Issues with the same `type` always have the same `level`, and they can be represented hierarchically.



Database check result

The most general status of the database `self_check_result` can have the following values:

Value	Description
`GOOD`	No issues were detected.
`DEGRADED`	Degradation of at least one of the database systems was detected, but the database is still functioning (for example, allowable disk loss).
`MAINTENANCE_REQUIRED`	Significant degradation was detected, there is a risk of database unavailability, and human intervention is required.
`EMERGENCY`	A serious problem was detected in the database, with complete or partial unavailability.

Issue status

The status (severity) of the current issue `issue_log.status`:

Value	Description
`GREY`	Unable to determine the status (an issue with the self-diagnostic subsystem).
`GREEN`	No issues detected.
`BLUE`	Temporary minor degradation that does not affect database availability; the system is expected to return to `GREEN`.
`YELLOW`	A minor issue with no risks to availability. It is recommended to continue monitoring the issue.
`ORANGE`	A serious issue, a step away from losing availability. Maintenance may be required.
`RED`	A component is faulty or unavailable.

Possible issues

DATABASE

Database has multiple issues, Database has compute issues, Database has storage issues

Description: These issues depend solely on the underlying `COMPUTE` and `STORAGE` layers. This represents the most general status of a database.

STORAGE

There are no storage pools

Description: Information about storage pools is unavailable. Most likely, the storage pools are not configured.

Storage degraded, Storage has no redundancy, Storage failed

Description: These issues depend solely on the underlying `STORAGE_POOLS` layer.

System tablet BSC didn't provide information

Description: Storage diagnostics will be generated using an alternative method.

Storage usage over 75%, Storage usage over 85%, Storage usage over 90%

Description: Some data needs to be removed, or the database needs to be reconfigured with additional disk space.

STORAGE_POOL

Pool degraded, Pool has no redundancy, Pool failed

Description: These issues depend solely on the underlying `STORAGE_GROUP` layer.

STORAGE_GROUP

Group has no vslots

Description: This situation is not expected; it is an internal issue.

Group layout is incorrect

Description: The storage group was configured incorrectly.

Actions: In the YDB UI, navigate to the database page, select the `Storage` tab, and use the known group `id` to check the configuration of nodes and disks on the nodes.

Group degraded

Description: A number of disks allowed in the group are not available.

Logic of work: `HealthCheck` checks various parameters (fault tolerance mode, number of failed disks, disk status, etc.) and sets the appropriate status for the group accordingly.

Actions: In YDB UI, navigate to the database page, select the `Storage` tab, apply the `Groups` and `Degraded` filters, and use the known group `id` to check the availability of nodes and disks on the nodes.

Group has no redundancy

Description: A storage group has lost its redundancy. Another VDisk failure could result in the loss of the group.

Logic of work: `HealthCheck` monitors various parameters (fault tolerance mode, number of failed disks, disk status, etc.) and sets the appropriate status for the group based on these parameters.

Actions: In YDB UI, navigate to the database page, select the `Storage` tab, apply the `Groups` and `Degraded` filters, and use the known group `id` to check the availability of nodes and disks on those nodes.

Group failed

Description: A storage group has lost its integrity, and data is no longer available. `HealthCheck` evaluates various parameters (fault tolerance mode, number of failed disks, disk status, etc.) and determines the appropriate status, displaying a message accordingly.

Logic of work: `HealthCheck` monitors various parameters (fault tolerance mode, number of failed disks, disk status, etc.) and sets the appropriate status for the group accordingly.

Actions: In YDB UI, navigate to the database page, select the `Storage` tab, apply the `Groups` and `Degraded` filters, and use the known group `id` to check the availability of nodes and disks on those nodes.

VDISK

System tablet BSC did not provide known status

Description: This situation is not expected; it is an internal issue.

VDisk is not available

Description: The disk is not operational.

Actions: In YDB YDB UI, navigate to the database page, select the `Storage` tab, and apply the `Groups` and `Degraded` filters. The group `id` can be found through the related `STORAGE_GROUP` issue. Hover over the relevant VDisk to identify the node with the problem and check the availability of nodes and disks on those nodes.

VDisk is being initialized

Description: The disk is in the process of initialization.

Actions: In YDB UI, navigate to the database page, select the `Storage` tab, and apply the `Groups` and `Degraded` filters. The group `id` can be found through the related `STORAGE_GROUP` issue. Hover over the relevant VDisk to identify the node with the problem and check the availability of nodes and disks on those nodes.

Replication in progress

Description: The disk is accepting queries, but not all data has been replicated.

Actions: In YDB UI, navigate to the database page, select the `Storage` tab, and apply the `Groups` and `Degraded` filters. The group `id` can be found through the related `STORAGE_GROUP` issue. Hover over the relevant VDisk to identify the node with the problem and check the availability of nodes and disks on those nodes.

VDisk have space issue

Description: These issues depend solely on the underlying `PDISK` layer.

PDISK

Unknown PDisk state

Description: `HealthCheck` the system can't parse pdisk state.

PDisk state is

Description: Indicates state of physical disk.

Actions: In YDB UI, navigate to the database page, select the `Storage` tab, set the `Nodes` and `Degraded` filters, and use the known node id and PDisk to check the availability of nodes and disks on the nodes.

Available size is less than 12%, Available size is less than 9%, Available size is less than 6%

Description: Free space on the physical disk is running out.
Actions: In YDB UI, navigate to the database page, select the `Storage` tab, set the `Nodes` and `Out of Space` filters, and use the known node and PDisk identifiers to check the available space.

PDisk is not available

Description: A physical disk is not available.

Actions: In YDB UI, navigate to the database page, select the `Storage` tab, set the `Nodes` and `Degraded` filters, and use the known node and PDisk identifiers to check the availability of nodes and disks on the nodes.

STORAGE_NODE

Storage node is not available

Description: A storage node is not available.

COMPUTE

There are no compute nodes

Description: The database has no nodes available to start the tablets. Unable to determine `COMPUTE_NODE` issues below.

Compute has issues with system tablets

Description: These issues depend solely on the underlying `SYSTEM_TABLET` layer.

Some nodes are restarting too often

Description: These issues depend solely on the underlying `NODE_UPTIME` layer.

Compute is overloaded

Description: These issues depend solely on the underlying `COMPUTE_POOL` layer.

Compute quota usage

Description: These issues depend solely on the underlying `COMPUTE_QUOTA` layer.

Compute has issues with tablets

Description: These issues depend solely on the underlying `TABLET` layer.

COMPUTE_QUOTA

Paths quota usage is over than 90%, Paths quota usage is over than 99%, Paths quota exhausted, Shards quota usage is over than 90%, Shards quota usage is over than 99%, Shards quota exhausted

Description: Quotas are exhausted.

Actions: Check the number of objects (tables, topics) in the database and delete any unnecessary ones.

SYSTEM_TABLET

System tablet is unresponsive, System tablet response time over 1000ms, System tablet response time over 5000ms

Description: The system tablet is either not responding or takes too long to respond.

Actions: In YDB UI, navigate to the `Storage` tab and apply the `Nodes` filter. Check the `Uptime` and the nodes' statuses. If the `Uptime` is short, review the logs to determine the reasons for the node restarts.

TABLET

Tablets are restarting too often

Description: Tablets are restarting too frequently.

Actions: In YDB UI, navigate to the `Nodes` tab. Check the `Uptime` and the nodes' statuses. If the `Uptime` is short, review the logs to determine the reasons for the node restarts.

Tablets/Followers are dead

Description: Tablets are not running (likely cannot be started).

Actions: In YDB UI, navigate to the `Nodes` tab. Check the `Uptime` and the nodes' statuses. If the `Uptime` is short, review the logs to determine the reasons for the node restarts.

LOAD_AVERAGE

LoadAverage above 100%

Description: A physical host is overloaded, meaning the system is operating at or beyond its capacity, potentially due to a high number of processes waiting for I/O operations. For more information on load, see Load (computing).

Logic of work:

- Load Information:
  - Source: `/proc/loadavg`
  - The first number of the three represents the average load over the last 1 minute.
- Logical Cores Information:
  - Primary Source: `/sys/fs/cgroup/cpu.max`
  - Fallback Sources: `/sys/fs/cgroup/cpu/cpu.cfs_quota_us`, `/sys/fs/cgroup/cpu/cpu.cfs_period_us`

The number of cores is calculated by dividing the quota by the period .

Actions: Check the CPU load on the nodes.

COMPUTE_POOL

Pool usage is over than 90%, Pool usage is over than 95%, Pool usage is over than 99%

Description: One of the pools' CPUs is overloaded.

Actions: Add cores to the configuration of the actor system for the corresponding CPU pool.

NODE_UPTIME

The number of node restarts has increased

Description: The number of node restarts has exceeded the threshold. By default, this is set to 10 restarts per hour.

Actions: Check the logs to determine the reasons for the process restarts.

Node is restarting too often

Description: The number of node restarts has exceeded the threshold. By default, this is set to 30 restarts per hour.

Actions: Check the logs to determine the reasons for the process restarts.

NODES_TIME_DIFFERENCE

Node is ... ms behind peer [id], Node is ... ms ahead of peer [id]

Description: Time drift on nodes might lead to potential issues with coordinating distributed transactions. This issue starts to appear when the time difference is 5 ms or more.

Actions: Check for discrepancies in system time between the nodes listed in the alert, and verify the operation of the time synchronization process.

Examples

The shortest `HealthCheck` response looks like this. It is returned if there is nothing wrong with the database:

```
{
  "self_check_result": "GOOD"
}
```

Verbose example

`GOOD` response with `verbose` parameter:

```
{
    "self_check_result": "GOOD",
    "database_status": [
        {
            "name": "/amy/db",
            "overall": "GREEN",
            "storage": {
                "overall": "GREEN",
                "pools": [
                    {
                        "id": "/amy/db:ssdencrypted",
                        "overall": "GREEN",
                        "groups": [
                            {
                                "id": "2181038132",
                                "overall": "GREEN",
                                "vdisks": [
                                    {
                                        "id": "9-1-1010",
                                        "overall": "GREEN",
                                        "pdisk": {
                                            "id": "9-1",
                                            "overall": "GREEN"
                                        }
                                    },
                                    {
                                        "id": "11-1004-1009",
                                        "overall": "GREEN",
                                        "pdisk": {
```

```
                                    "id": "11-1004",
                                    "overall": "GREEN"
                                }
                            },
                            {
                                "id": "10-1003-1011",
                                "overall": "GREEN",
                                "pdisk": {
                                    "id": "10-1003",
                                    "overall": "GREEN"
                                }
                            },
                            {
                                "id": "8-1005-1010",
                                "overall": "GREEN",
                                "pdisk": {
                                    "id": "8-1005",
                                    "overall": "GREEN"
                                }
                            },
                            {
                                "id": "7-1-1008",
                                "overall": "GREEN",
                                "pdisk": {
                                    "id": "7-1",
                                    "overall": "GREEN"
                                }
                            },
                            {
                                "id": "6-1-1007",
                                "overall": "GREEN",
                                "pdisk": {
                                    "id": "6-1",
                                    "overall": "GREEN"
                                }
                            },
                            {
                                "id": "4-1005-1010",
                                "overall": "GREEN",
                                "pdisk": {
                                    "id": "4-1005",
                                    "overall": "GREEN"
                                }
                            },
                            {
                                "id": "2-1003-1013",
                                "overall": "GREEN",
                                "pdisk": {
                                    "id": "2-1003",
                                    "overall": "GREEN"
                                }
                            },
                            {
                                "id": "1-1-1008",
                                "overall": "GREEN",
                                "pdisk": {
                                    "id": "1-1",
                                    "overall": "GREEN"
                                }
                            }
                        ]
                    }
                ]
            }
        ]
    },
    "compute": {
        "overall": "GREEN",
        "nodes": [
            {
                "id": "50073",
                "overall": "GREEN",
                "pools": [
                    {
                        "overall": "GREEN",
                        "name": "System",
                        "usage": 0.000405479
                    },
                    {
                        "overall": "GREEN",
                        "name": "User",
                        "usage": 0.00265229
                    },
                    {
                        "overall": "GREEN",
                        "name": "Batch",
                        "usage": 0.000347933
                    },
                    {
                        "overall": "GREEN",
                        "name": "IO",
```

```
                        "usage": 0.000312022
                    },
                    {
                        "overall": "GREEN",
                        "name": "IC",
                        "usage": 0.000945925
                    }
                ],
                "load": {
                    "overall": "GREEN",
                    "load": 0.2,
                    "cores": 4
                }
            },
            {
                "id": "50074",
                "overall": "GREEN",
                "pools": [
                    {
                        "overall": "GREEN",
                        "name": "System",
                        "usage": 0.000619053
                    },
                    {
                        "overall": "GREEN",
                        "name": "User",
                        "usage": 0.00463859
                    },
                    {
                        "overall": "GREEN",
                        "name": "Batch",
                        "usage": 0.000596071
                    },
                    {
                        "overall": "GREEN",
                        "name": "IO",
                        "usage": 0.0006241
                    },
                    {
                        "overall": "GREEN",
                        "name": "IC",
                        "usage": 0.00218465
                    }
                ],
                "load": {
                    "overall": "GREEN",
                    "load": 0.08,
                    "cores": 4
                }
            },
            {
                "id": "50075",
                "overall": "GREEN",
                "pools": [
                    {
                        "overall": "GREEN",
                        "name": "System",
                        "usage": 0.000579126
                    },
                    {
                        "overall": "GREEN",
                        "name": "User",
                        "usage": 0.00344293
                    },
                    {
                        "overall": "GREEN",
                        "name": "Batch",
                        "usage": 0.000592347
                    },
                    {
                        "overall": "GREEN",
                        "name": "IO",
                        "usage": 0.000525747
                    },
                    {
                        "overall": "GREEN",
                        "name": "IC",
                        "usage": 0.00174265
                    }
                ],
                "load": {
                    "overall": "GREEN",
                    "load": 0.26,
                    "cores": 4
                }
            }
        ],
        "tablets": [
            {
                "overall": "GREEN",
                "type": "SchemeShard",
```

```
                        "state": "GOOD",
                        "count": 1
                    },
                    {
                        "overall": "GREEN",
                        "type": "SysViewProcessor",
                        "state": "GOOD",
                        "count": 1
                    },
                    {
                        "overall": "GREEN",
                        "type": "Coordinator",
                        "state": "GOOD",
                        "count": 3
                    },
                    {
                        "overall": "GREEN",
                        "type": "Mediator",
                        "state": "GOOD",
                        "count": 3
                    },
                    {
                        "overall": "GREEN",
                        "type": "Hive",
                        "state": "GOOD",
                        "count": 1
                    }
                ]
            }
        }
    ]
}
```


Emergency example

Response with `EMERGENCY` status:


```
{
  "self_check_result": "EMERGENCY",
  "issue_log": [
    {
      "id": "RED-27c3-70fb",
      "status": "RED",
      "message": "Database has multiple issues",
      "location": {
        "database": {
          "name": "/slice"
        }
      },
      "reason": [
        "RED-27c3-4e47",
        "RED-27c3-53b5",
        "YELLOW-27c3-5321"
      ],
      "type": "DATABASE",
      "level": 1
    },
    {
      "id": "RED-27c3-4e47",
      "status": "RED",
      "message": "Compute has issues with system tablets",
      "location": {
        "database": {
          "name": "/slice"
        }
      },
      "reason": [
        "RED-27c3-c138-BSController"
      ],
      "type": "COMPUTE",
      "level": 2
    },
    {
      "id": "RED-27c3-c138-BSController",
      "status": "RED",
      "message": "System tablet is unresponsive",
      "location": {
        "compute": {
          "tablet": {
            "type": "BSController",
            "id": [
              "72057594037989391"
            ]
          }
        },
        "database": {
          "name": "/slice"
        }
      },
      "type": "SYSTEM_TABLET",
      "level": 3
```

```
    },
    {
      "id": "RED-27c3-53b5",
      "status": "RED",
      "message": "System tablet BSC didn't provide information",
      "location": {
        "database": {
          "name": "/slice"
        }
      },
      "type": "STORAGE",
      "level": 2
    },
    {
      "id": "YELLOW-27c3-5321",
      "status": "YELLOW",
      "message": "Storage degraded",
      "location": {
        "database": {
          "name": "/slice"
        }
      },
      "reason": [
        "YELLOW-27c3-595f-8d1d"
      ],
      "type": "STORAGE",
      "level": 2
    },
    {
      "id": "YELLOW-27c3-595f-8d1d",
      "status": "YELLOW",
      "message": "Pool degraded",
      "location": {
        "storage": {
          "pool": {
            "name": "static"
          }
        },
        "database": {
          "name": "/slice"
        }
      },
      "reason": [
        "YELLOW-27c3-ef3e-0"
      ],
      "type": "STORAGE_POOL",
      "level": 3
    },
    {
      "id": "RED-84d8-3-3-1",
      "status": "RED",
      "message": "PDisk is not available",
      "location": {
        "storage": {
          "node": {
            "id": 3,
            "host": "man0-0026.ydb-dev.nemax.nebiuscloud.net",
            "port": 19001
          },
          "pool": {
            "group": {
              "vdisk": {
                "pdisk": [
                  {
                    "id": "3-1",
                    "path": "/dev/disk/by-partlabel/NVMEKIKIMR01"
                  }
                ]
              }
            }
          }
        }
      },
      "type": "PDISK",
      "level": 6
    },
    {
      "id": "RED-27c3-4847-3-0-1-0-2-0",
      "status": "RED",
      "message": "VDisk is not available",
      "location": {
        "storage": {
          "node": {
            "id": 3,
            "host": "man0-0026.ydb-dev.nemax.nebiuscloud.net",
            "port": 19001
          },
          "pool": {
            "name": "static",
            "group": {
              "vdisk": {
```

```
                "id": [
                  "0-1-0-2-0"
                ]
              }
            }
          }
        },
        "database": {
          "name": "/slice"
        }
      },
      "reason": [
        "RED-84d8-3-3-1"
      ],
      "type": "VDISK",
      "level": 5
    },
    {
      "id": "YELLOW-27c3-ef3e-0",
      "status": "YELLOW",
      "message": "Group degraded",
      "location": {
        "storage": {
          "pool": {
            "name": "static",
            "group": {
              "id": [
                "0"
              ]
            }
          }
        },
        "database": {
          "name": "/slice"
        }
      },
      "reason": [
        "RED-27c3-4847-3-0-1-0-2-0"
      ],
      "type": "STORAGE_GROUP",
      "level": 4
    }
  ],
  "location": {
    "id": 5,
    "host": "man0-0028.ydb-dev.nemax.nebiuscloud.net",
    "port": 19001
  }
}
```

Observability in YDB SDK

This section describes how to enable YDB SDK diagnostic tools — logging, metrics, and distributed tracing. It is recommended to enable them in advance, before problems occur, so that when investigating an incident you can see the full picture of the system state before, during, and after the failure.

Logging:

- Enabling logging
- Exporting logs to OpenTelemetry

Metrics:

- Metrics with OpenTelemetry
- Enabling metrics in Prometheus

Tracing:

- Tracing with OpenTelemetry
- Enabling tracing in Jaeger

Server-side observability YDB, which is independent of the SDK, is described in the Reference on YDB observability section.

Enabling logging

Below are code examples for enabling logging in different YDB SDKs.

C++

This functionality is not currently supported.

Go

Native SDK

There are several ways to enable logging in an application that uses `ydb-go-sdk`:

Via variable environment `YDB_LOG_SEVERITY_LEVEL`

This environment variable enables the built-in `ydb-go-sdk` logger (synchronous, non-blocking) with output to the standard output stream.
You can set the environment variable as follows:

```
export YDB_LOG_SEVERITY_LEVEL=info
```

(available values `trace`, `debug`, `info`, `warn`, `error`, `fatal`, `quiet`, default `quiet`).

Connect third-party logger `go.uber.org/zap`

```
package main

import (
  "context"
  "os"

  "go.uber.org/zap"

  ydbZap "github.com/ydb-platform/ydb-go-sdk-zap"
  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  var log *zap.Logger // zap-logger with init out of this scope
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydbZap.WithTraces(
      log,
      trace.DetailsAll,
    ),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  ...
}
```

Connect third-party logger `github.com/rs/zerolog`

```
package main

import (
  "context"
  "os"

  "github.com/rs/zerolog"

  ydbZerolog "github.com/ydb-platform/ydb-go-sdk-zerolog"
  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  var log zerolog.Logger // zap-logger with init out of this scope
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydbZerolog.WithTraces(
      &log,
      trace.DetailsAll,
    ),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
```

```
  ...
}
```

Connect own implementation logger `github.com/ydb-platform/ydb-go-sdk/v3/log.Logger`

```
package main

import (
  "context"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/log"
  "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  var logger log.Logger // logger implementation with init out of this scope
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithLogger(
      logger,
      trace.DetailsAll,
    ),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  ...
}
```

Implement own package logging

You can implement your own logging package based on driver events in the `github.com/ydb-platform/ydb-go-sdk/v3/trace` tracing package. The `github.com/ydb-platform/ydb-go-sdk/v3/trace` tracing package contains descriptions of all logged driver events.

Implement obtaining information about server errors `IterateByIssues`

When working with YDB via the Go SDK, you can not only enable logging of requests and responses, but also programmatically obtain detailed information about server errors (issues) — additional messages that YDB returns as part of the response when operations fail. To iterate over the list of issues contained in the server response, use the IterateByIssues method.

database/sql

There are several ways to enable logging in an application that uses `ydb-go-sdk`:

Via variable environment `YDB_LOG_SEVERITY_LEVEL`

This environment variable enables the built-in `ydb-go-sdk` logger (synchronous, non-blocking) with output to the standard output stream.
You can set the environment variable as follows:

```
export YDB_LOG_SEVERITY_LEVEL=info
```

(available values `trace`, `debug`, `info`, `warn`, `error`, `fatal`, `quiet`, default `quiet`).

Connect third-party logger `go.uber.org/zap`

```
package main

import (
  "context"
  "database/sql"
  "os"

  "go.uber.org/zap"

  ydbZap "github.com/ydb-platform/ydb-go-sdk-zap"
  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  var log *zap.Logger // zap-logger with init out of this scope
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydbZap.WithTraces(
```

```
      log,
      trace.DetailsAll,
    ),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  defer connector.Close()

  db := sql.OpenDB(connector)
  defer db.Close()
  ...
}
```

Connect third-party logger `github.com/rs/zerolog`

```
package main

import (
  "context"
  "database/sql"
  "os"

  "github.com/rs/zerolog"

  ydbZerolog "github.com/ydb-platform/ydb-go-sdk-zerolog"
  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  var log zerolog.Logger // zap-logger with init out of this scope
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydbZerolog.WithTraces(
      &log,
      trace.DetailsAll,
    ),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  defer connector.Close()

  db := sql.OpenDB(connector)
  defer db.Close()
  ...
}
```

Connect own implementation logger `github.com/ydb-platform/ydb-go-sdk/v3/log.Logger`

```
package main

import (
  "context"
  "database/sql"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/log"
  "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  var logger log.Logger // logger implementation with init out of this scope
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithLogger(
      logger,
      trace.DetailsAll,
    ),
```

```
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  defer connector.Close()

  db := sql.OpenDB(connector)
  defer db.Close()
  ...
}
```

Implement own package logging

> You can implement your own logging package based on driver events in the `github.com/ydb-platform/ydb-go-sdk/v3/trace` tracing package. The `github.com/ydb-platform/ydb-go-sdk/v3/trace` tracing package contains descriptions of all logged driver events.

Java

Native SDK

The YDB Java SDK uses the slf4j library for logging, which allows using different logging levels (`error`, `warn`, `info`, `debug`, `trace`) for one or more loggers. The following loggers are available in the current implementation:

- The `tech.ydb.core.grpc` logger provides information about the internal implementation of the gRPC protocol
- The `debug` level logs all operations over the gRPC protocol; it is recommended to use only for debugging
- The `info` level is recommended for use by default
- The `tech.ydb.table.impl` logger at the `debug` level allows tracking the internal state of the YDB driver, in particular the session pool operation.
- The `tech.ydb.table.SessionRetryContext` logger at the `debug` level will inform about the number of retries, results of executed queries, execution time of individual retries, and the total execution time of the entire operation
- The `tech.ydb.table.Session` logger at the `debug` level provides information about the query text, response status, and execution time for various session operations

Enabling and configuring Java SDK loggers depends on the `slf4j-api` implementation used.
Here is an example of `log4j2` configuration for the `log4j-slf4j-impl` library

```
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>

  <Loggers>
    <Logger name="io.netty" level="warn" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
    <Logger name="io.grpc.netty" level="warn" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
    <Logger name="tech.ydb.core.grpc" level="info" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
    <Logger name="tech.ydb.table.impl" level="info" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
    <Logger name="tech.ydb.table.SessionRetryContext" level="debug" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
    <Logger name="tech.ydb.table.Session" level="debug" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>

    <Root level="debug" >
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

JDBC

The JDBC driver uses the same logging stack via `slf4j`; configure the `tech.ydb.*` loggers the same way as in the native SDK.

The same debug logs are available in all other frameworks around JDBC (Spring Boot, ORM, connection pools, etc.): they go to YDB through this driver; just connect the same `slf4j` / log4j2 / logback configuration in the application.

Python

The Python SDK uses the standard logging library — `logging`. To enable a specific logging mode:

```
import logging

logging.getLogger('ydb').setLevel(logging.DEBUG)
```

JavaScript

For logging events inside the SDK, the debug library is used.
To enable logs, set the `DEBUG` environment variable with the SDK event filter value — `DEBUG=ydbjs:*`.

Rust

Inside the `ydb` crate, messages go through the standard Rust ecosystem library `tracing` (this is the crate name; it also includes regular text logs at debug/trace level, not just "distributed tracing"). To see output in the console, before creating the client, attach a subscriber, for example `tracing_subscriber::fmt` with the required level (`TRACE` for maximum detail). Example: `basic-logs.rs`.

```
tracing_subscriber::fmt()
    .with_max_level(tracing::Level::TRACE)
    .init();

let client = ydb::ClientBuilder::new_from_connection_string("grpc://localhost:2136?database=local")?
    .client()?;
```

C#

In the YDB C# SDK, logging is connected via the standard `ILoggerFactory` interface from `Microsoft.Extensions.Logging`. You can pass any implementation — a console logger, Serilog, NLog, and others:

```
using Microsoft.Extensions.Logging;
using Ydb.Sdk.Ado;

var loggerFactory = LoggerFactory.Create(builder =>
    builder.AddConsole().SetMinimumLevel(LogLevel.Debug));

var ydbBuilder = new YdbConnectionStringBuilder
{
    Host = "localhost",
    Port = 2136,
    Database = "/local",
    LoggerFactory = loggerFactory
};

await using var dataSource = new YdbDataSource(ydbBuilder);
await using var connection = await dataSource.OpenConnectionAsync();
```

Since `LoggerFactory` accepts the standard `ILoggerFactory`, you can connect Serilog or NLog without additional adapters:

```
// Serilog
var loggerFactory = new SerilogLoggerFactory(Log.Logger);

// NLog
var loggerFactory = LoggerFactory.Create(builder => builder.AddNLog());
```

PHP

In the YDB PHP SDK, for logging you need to use a class that implements `\Psr\Log\LoggerInterface`.
The YDB-PHP-SDK includes built-in loggers in the `YdbPlatform\Ydb\Logger` namespace:

- `NullLogger` - the default one that outputs nothing
- `SimpleStdLogger($level)` - a logger that outputs logs to stderr.

Usage example:

```
$config = [
  'logger' => new \YdbPlatform\Ydb\Logger\SimpleStdLogger(\YdbPlatform\Ydb\Logger\SimpleStdLogger::INFO)
]
$ydb = new \YdbPlatform\Ydb\Ydb($config);
```

Exporting logs to OpenTelemetry

Each YDB SDK writes its own internal logs (driver initialization, session pool, query execution, retries, etc.) through the standard logging facility of its language. Instead of outputting logs only to the console (see Enable logging), they can be redirected to the OpenTelemetry Logs SDK and exported via the standard OTLP protocol to a collector. The collector then forwards the records to the chosen backend for log storage and viewing.

The principle is the same in all SDKs:

1. Create an OpenTelemetry `LoggerProvider` with an OTLP log exporter and the `service.name` resource attribute.
2. Redirect the SDK logger to this provider through an adapter (log appender / bridge) that converts each SDK log record into an OTel log record. For the list of adapters and the status of log support in different languages, see the OpenTelemetry logging documentation.
3. Run the workload — all internal SDK logs are now sent to the collector as OTLP log records.

> **Note**
>
> ### Principles
>
> The method of log collection depends on the language and infrastructure — there is no single format:
>
> - **Programmatic bridges (log appender / bridge).** The application itself sends logs to the collector via OTLP directly from the process (direct-to-Collector workflow). This is the approach shown in the examples below.
> - **Collector agents.** The application writes logs to a file or `stdout`, and a separate agent (e.g., the filelog receiver in the OpenTelemetry Collector) reads, parses, and forwards them to the backend — without changing the application code.

Connecting to the SDK

C++

YDB The YDB C++ SDK writes logs through `TLogBackend` from `util`. Implement a backend that converts each record into an OTel log record, and pass it to the driver via `TDriverConfig::SetLog`. For building the OpenTelemetry C++ SDK, see Getting Started.

```cpp
#include <ydb-cpp-sdk/client/driver/driver.h>
#include <opentelemetry/exporters/otlp/otlp_grpc_log_record_exporter_factory.h>
#include <opentelemetry/exporters/otlp/otlp_grpc_log_record_exporter_options.h>
#include <opentelemetry/logs/provider.h>
#include <opentelemetry/sdk/logs/batch_log_record_processor_factory.h>
#include <opentelemetry/sdk/logs/logger_provider_factory.h>
#include <opentelemetry/sdk/resource/resource.h>
#include <library/cpp/logger/backend.h>
#include <library/cpp/logger/record.h>
#include <memory>

namespace otlp     = opentelemetry::exporter::otlp;
namespace logs_sdk = opentelemetry::sdk::logs;
namespace logs_api = opentelemetry::logs;
namespace resource = opentelemetry::sdk::resource;

namespace {

class TOtelLogBackend final : public TLogBackend {
public:
  explicit TOtelLogBackend(opentelemetry::nostd::shared_ptr<logs_api::Logger> logger)
      : Logger_(std::move(logger)) {}

  void WriteData(const TLogRecord& rec) override {
      Logger_->EmitLogRecord(MapSeverity(rec.Priority),
                             opentelemetry::nostd::string_view(rec.Data, rec.Len));
  }

  void ReopenLog() override {}

private:
  static logs_api::Severity MapSeverity(ELogPriority p) {
      switch (p) {
          case TLOG_EMERG:
          case TLOG_ALERT:
          case TLOG_CRIT:    return logs_api::Severity::kFatal;
          case TLOG_ERR:     return logs_api::Severity::kError;
          case TLOG_WARNING: return logs_api::Severity::kWarn;
          case TLOG_NOTICE:
          case TLOG_INFO:    return logs_api::Severity::kInfo;
          case TLOG_DEBUG:   return logs_api::Severity::kDebug;
          default:           return logs_api::Severity::kTrace;
      }
  }

  opentelemetry::nostd::shared_ptr<logs_api::Logger> Logger_;
};
} // namespace

int main() {
  // 1. Configuring the OTel log provider with an OTLP exporter
  otlp::OtlpGrpcLogRecordExporterOptions exporterOpts;
  exporterOpts.endpoint = "localhost:4317";
  auto exporter  = otlp::OtlpGrpcLogRecordExporterFactory::Create(exporterOpts);
  auto processor = logs_sdk::BatchLogRecordProcessorFactory::Create(std::move(exporter));
```

```
  auto res       = resource::Resource::Create({{"service.name", "ydb-cpp-sdk-otel-logs-sample"}});

  std::shared_ptr<logs_api::LoggerProvider> provider(
      logs_sdk::LoggerProviderFactory::Create(std::move(processor), res));
  logs_api::Provider::SetLoggerProvider(provider);

  auto logger = provider->GetLogger("ydb-cpp-sdk");

  // 2. Passing the bridge to the YDB driver
  auto config = NYdb::TDriverConfig()
      .SetEndpoint("localhost:2136")
      .SetDatabase("/local")
      .SetLog(std::make_unique<TOtelLogBackend>(logger));

  NYdb::TDriver driver(config);
  // ... use driver ...
  driver.Stop(true);
}
```

Go

For the YDB Go SDK, there is a ready-made adapter ydb-go-sdk-otel that converts SDK events into OpenTelemetry signals: traces (`WithTracer`), metrics (`WithMetrics`), and logs (`WithLogger`). The adapter does not configure exporters itself — you create a `LoggerProvider` with an OTLP log exporter, obtain a logger from it, and pass it to the `ydbOtel.WithLogger` option when calling `ydb.Open`. Each internal SDK log record (driver initialization, session pool, query execution, retries, etc.) is then sent to the collector as an OTLP log record.

```
go get github.com/ydb-platform/ydb-go-sdk-otel
go get go.opentelemetry.io/otel/sdk/log
go get go.opentelemetry.io/otel/exporters/otlp/otlplog/otlploggrpc
```

```
package main

import (
    "context"
    "os"

    "github.com/ydb-platform/ydb-go-sdk/v3"
    "github.com/ydb-platform/ydb-go-sdk/v3/trace"

    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/exporters/otlp/otlplog/otlploggrpc"
    sdklog "go.opentelemetry.io/otel/sdk/log"
    "go.opentelemetry.io/otel/sdk/resource"

    ydbOtel "github.com/ydb-platform/ydb-go-sdk-otel"
)

func main() {
    ctx := context.Background()

    // 1. Configuring the OTel log provider with an OTLP exporter.
    exporter, err := otlploggrpc.New(ctx,
        otlploggrpc.WithEndpoint("localhost:4317"),
        otlploggrpc.WithInsecure(),
    )
    if err != nil {
        panic(err)
    }
    res, _ := resource.Merge(resource.Default(), resource.NewSchemaless(
        attribute.String("service.name", "ydb-go-sdk-otel-logs-sample"),
    ))
    lp := sdklog.NewLoggerProvider(
        sdklog.WithResource(res),
        sdklog.WithProcessor(sdklog.NewBatchProcessor(exporter)),
    )
    defer lp.Shutdown(ctx)

    // 2. Opening the YDB driver with the ydb-go-sdk-otel adapter.
    // WithLogger forwards SDK log events to OTel log records.
    logger := lp.Logger("ydb-go-sdk")
    db, err := ydb.Open(ctx,
        os.Getenv("YDB_CONNECTION_STRING"),
        ydbOtel.WithLogger(logger, ydbOtel.WithDetailer(trace.DetailsAll)),
    )
    if err != nil {
        panic(err)
    }
    defer db.Close(ctx)
    // ... use db ...
}
```

Java

YDB Java SDK writes logs via `slf4j`. If logback is used as the `slf4j` implementation, connect the ready-made `opentelemetry-logback-appender-1.0` appender, which forwards each event to the OpenTelemetry Logs SDK:

```
<dependency>
    <groupId>io.opentelemetry</groupId>
```

```
    <artifactId>opentelemetry-sdk</artifactId>
    <version>${otel.version}</version>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-otlp</artifactId>
    <version>${otel.version}</version>
</dependency>
<dependency>
    <groupId>io.opentelemetry.instrumentation</groupId>
    <artifactId>opentelemetry-logback-appender-1.0</artifactId>
    <version>${otel.instrumentation.version}</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>${logback.version}</version>
</dependency>
```

Connect the appender in `logback.xml`:

```
<configuration>
    <appender name="OTEL" class="io.opentelemetry.instrumentation.logback.appender.v1_0.OpenTelemetryAppende
r">
    </appender>

    <root level="INFO">
        <appender-ref ref="OTEL"/>
    </root>
</configuration>
```

Build an OpenTelemetry SDK instance with an OTLP log exporter and set it in the appender by calling `OpenTelemetryAppender.install(...)` before starting the workload:

```
import java.util.concurrent.TimeUnit;

import io.opentelemetry.api.OpenTelemetry;
import io.opentelemetry.api.common.Attributes;
import io.opentelemetry.exporter.otlp.logs.OtlpGrpcLogRecordExporter;
import io.opentelemetry.instrumentation.logback.appender.v1_0.OpenTelemetryAppender;
import io.opentelemetry.sdk.OpenTelemetrySdk;
import io.opentelemetry.sdk.logs.SdkLoggerProvider;
import io.opentelemetry.sdk.logs.export.BatchLogRecordProcessor;
import io.opentelemetry.sdk.resources.Resource;
import tech.ydb.core.grpc.GrpcTransport;
import tech.ydb.query.QueryClient;

String serviceName = "ydb-java-sdk-otel-logs-sample";
Resource resource = Resource.getDefault().merge(
        Resource.create(Attributes.builder().put("service.name", serviceName).build()));

SdkLoggerProvider loggerProvider = SdkLoggerProvider.builder()
        .setResource(resource)
        .addLogRecordProcessor(BatchLogRecordProcessor.builder(
                OtlpGrpcLogRecordExporter.builder().setEndpoint("http://localhost:4317").build()
        ).build())
        .build();

OpenTelemetry openTelemetry = OpenTelemetrySdk.builder()
        .setLoggerProvider(loggerProvider)
        .build();

// Passing the OpenTelemetry SDK to the appender declared in logback.xml.
OpenTelemetryAppender.install(openTelemetry);

try (GrpcTransport transport = GrpcTransport.forConnectionString("grpc://localhost:2136/local").build();
     QueryClient queryClient = QueryClient.newClient(transport).build()) {
    // ... use queryClient ...
} finally {
    loggerProvider.forceFlush().join(10, TimeUnit.SECONDS);
    loggerProvider.shutdown().join(10, TimeUnit.SECONDS);
}
```

Python

YDB Python SDK writes logs via the standard `logging` module (loggers named `ydb.*`). Use the built-in OpenTelemetry `LoggingHandler` to redirect these records to `LoggerProvider` and export via OTLP:

```
pip install opentelemetry-sdk opentelemetry-exporter-otlp-proto-grpc
```

```
import logging

import ydb
from opentelemetry._logs import set_logger_provider
from opentelemetry.exporter.otlp.proto.grpc._log_exporter import OTLPLogExporter
from opentelemetry.sdk._logs import LoggerProvider, LoggingHandler
from opentelemetry.sdk._logs.export import BatchLogRecordProcessor
```

```python
from opentelemetry.sdk.resources import Resource

resource = Resource(attributes={"service.name": "ydb-otel-logs-example"})
logger_provider = LoggerProvider(resource=resource)
logger_provider.add_log_record_processor(
    BatchLogRecordProcessor(OTLPLogExporter(endpoint="http://localhost:4317"))
)
set_logger_provider(logger_provider)

# Bridge stdlib logging -> OpenTelemetry. Binding the handler to the root logger
# intercepts everything that the SDK writes via logging.getLogger("ydb...").
otel_handler = LoggingHandler(level=logging.NOTSET, logger_provider=logger_provider)
logging.basicConfig(level=logging.INFO, handlers=[otel_handler])
logging.getLogger("ydb").setLevel(logging.INFO)

with ydb.Driver(endpoint="grpc://localhost:2136", database="/local") as driver:
    driver.wait(timeout=5)
    with ydb.QuerySessionPool(driver) as pool:
        pool.execute_with_retries("SELECT 1")

logger_provider.shutdown()
```

C#

YDB C# SDK writes logs via the `ILoggerFactory` passed to it. Create a factory based on the OpenTelemetry logging provider with an OTLP exporter and pass it to the data source:

```
dotnet add package OpenTelemetry
dotnet add package OpenTelemetry.Exporter.OpenTelemetryProtocol
```

```csharp
using Microsoft.Extensions.Logging;
using OpenTelemetry.Logs;
using OpenTelemetry.Resources;
using Ydb.Sdk.Ado;

var resourceBuilder = ResourceBuilder.CreateDefault()
    .AddService("ydb-sdk-otel-logs-sample");

using var loggerFactory = LoggerFactory.Create(builder =>
{
    builder.AddOpenTelemetry(options =>
    {
        options.SetResourceBuilder(resourceBuilder);
        options.AddOtlpExporter(o => o.Endpoint = new Uri("http://localhost:4317"));
    });
});

// Passing the factory to the SDK: each internal log (driver initialization, session pool,
// query execution, retries, ...) is exported to the collector as an OTLP log record.
await using var dataSource = new YdbDataSource(
    new YdbConnectionStringBuilder("Host=localhost;Port=2136;Database=/local")
    {
        LoggerFactory = loggerFactory
    });

await using var connection = await dataSource.OpenConnectionAsync();
await new YdbCommand("SELECT 1", connection).ExecuteNonQueryAsync();
```

JavaScript

This functionality is not currently supported.

Rust

This functionality is not currently supported.

Track progress or vote for support in the Rust SDK: ydb-rs-sdk#268

PHP

This functionality is not currently supported.

Metrics with OpenTelemetry

YDB SDK instruments Query Service operations with OpenTelemetry metrics, allowing you to observe the client state — operation duration and count, session pool state — from application code to gRPC calls to YDB. Metrics are exported via the standard OTLP protocol and are compatible with Prometheus, Grafana, VictoriaMetrics, and any other backend that supports OpenTelemetry.

List of metrics

Operation metrics

Name	Type	Unit	Description
`ydb.client.operation.duration`	Histogram	`s`	Duration of a single client operation attempt (`ExecuteQuery`, `Commit`, `Rollback`, `CreateSession`).
`ydb.client.operation.failed`	Counter	`{operation}`	Number of unsuccessful client operations.

Session pool metrics

Name	Type	Unit	Description
`ydb.query.session.create_time`	Histogram	`s`	Duration of creating a new session.
`ydb.query.session.pending_requests`	Counter	`{request}`	Monotonic counter of requests for session acquisition that entered the wait queue since the pool was created.
`ydb.query.session.timeouts`	Counter	`{timeout}`	Monotonic counter of timeouts when waiting for a free session, since the pool was created.
`ydb.query.session.count`	Gauge	`{session}`	Current number of sessions in the pool, broken down by state.
`ydb.query.session.min`	Gauge	`{session}`	Configured minimum session pool size.
`ydb.query.session.max`	Gauge	`{session}`	Configured maximum size of the session pool.

Retry metrics

Name	Type	Unit	Description
`ydb.client.retry.duration`	Histogram	`s`	Total client-visible duration of a logical operation performed through a retry policy, including all attempts and backoff delays.
`ydb.client.retry.attempts`	Histogram	`{attempt}`	Distribution of the number of attempts per logical operation. The value `1` means success on the first attempt.

Attributes

Name	Applies to	Value
`database`	`ydb.client.operation.duration`, `ydb.client.operation.failed`	Database name YDB.
`endpoint`	`ydb.client.operation.duration`, `ydb.client.operation.failed`	Discovery-endpoint in the format `host:port`.
`operation.name`	`ydb.client.operation.duration`, `ydb.client.operation.failed`, `ydb.client.retry.duration`, `ydb.client.retry.attempts`	Client operation name: `ExecuteQuery`, `Commit`, `Rollback`, `CreateSession`.
`status_code`	`ydb.client.operation.failed`	Status code YDB (for example, `BAD_REQUEST`, `SCHEME_ERROR`).
`ydb.query.session.pool.name`	All `ydb.query.session.*` metrics	Session pool name. By default, it is formed as `<endpoint>/<database>` and configured through the API of a specific SDK.
`ydb.query.session.state`	`ydb.query.session.count`	Session state: `idle` or `used`.

Connecting to the SDK

C++

Include the OpenTelemetry metrics header from YDB C++ SDK and register `MetricRegistry` in `TDriverConfig`:

```
#include <ydb-cpp-sdk/client/driver/driver.h>
#include <ydb-cpp-sdk/open_telemetry/metrics.h>

#include <opentelemetry/exporters/otlp/otlp_http_metric_exporter_factory.h>
#include <opentelemetry/exporters/otlp/otlp_http_metric_exporter_options.h>
#include <opentelemetry/sdk/metrics/meter_provider.h>
#include <opentelemetry/sdk/metrics/view/view_registry.h>
#include <opentelemetry/sdk/metrics/export/periodic_exporting_metric_reader_factory.h>
#include <opentelemetry/sdk/resource/resource.h>
#include <opentelemetry/metrics/provider.h>

namespace sdkmetrics = opentelemetry::sdk::metrics;
```

```
namespace otlp      = opentelemetry::exporter::otlp;
namespace resource  = opentelemetry::sdk::resource;
using namespace NYdb;

// 1. Initialize the OTel metrics provider
otlp::OtlpHttpMetricExporterOptions opts;
opts.url = "http://localhost:4318/v1/metrics";

auto exporter = otlp::OtlpHttpMetricExporterFactory::Create(opts);

sdkmetrics::PeriodicExportingMetricReaderOptions readerOpts;
readerOpts.export_interval_millis = std::chrono::milliseconds(5000);
auto reader = sdkmetrics::PeriodicExportingMetricReaderFactory::Create(
    std::move(exporter), readerOpts);

auto res = resource::Resource::Create({{"service.name", "my-service"}});
auto rawProvider = std::make_shared<sdkmetrics::MeterProvider>(
    std::unique_ptr<sdkmetrics::ViewRegistry>(new sdkmetrics::ViewRegistry()), res);
rawProvider->AddMetricReader(std::move(reader));

std::shared_ptr<opentelemetry::metrics::MeterProvider> meterProvider = rawProvider;
opentelemetry::metrics::Provider::SetMeterProvider(meterProvider);

// 2. Wrap in the YDB metrics registry
auto ydbMetricRegistry = NMetrics::CreateOtelMetricRegistry(meterProvider);

// 3. Create a YDB driver with metrics enabled
auto driverConfig = TDriverConfig()
    .SetEndpoint("localhost:2136")
    .SetDatabase("/local")
    .SetMetricRegistry(ydbMetricRegistry);

TDriver driver(driverConfig);
```

Metrics and tracing can be enabled together by registering both `MetricRegistry` and `TraceProvider` in `TDriverConfig`.

Go

Install the OpenTelemetry adapter for the YDB Go SDK:

```
go get github.com/ydb-platform/ydb-go-sdk-otel
```

Configure `MeterProvider`, get `Meter` from it, and pass it to the `ydbOtel.WithMetrics` adapter, which connects to the driver via the `ydb.Open` option. The granularity of collected metrics can be configured via `ydbOtel.WithDetailer`:

```
package main

import (
    "context"
    "os"

    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/exporters/otlp/otlpmetric/otlpmetricgrpc"
    "go.opentelemetry.io/otel/sdk/metric"
    "go.opentelemetry.io/otel/sdk/resource"

    "github.com/ydb-platform/ydb-go-sdk/v3"
    "github.com/ydb-platform/ydb-go-sdk/v3/trace"
    ydbOtel "github.com/ydb-platform/ydb-go-sdk-otel"
)

func main() {
    ctx := context.Background()

    exporter, err := otlpmetricgrpc.New(ctx,
        otlpmetricgrpc.WithEndpoint("localhost:4317"),
        otlpmetricgrpc.WithInsecure(),
    )
    if err != nil {
        panic(err)
    }
    res, _ := resource.Merge(resource.Default(), resource.NewSchemaless(
        attribute.String("service.name", "my-service"),
    ))
    mp := metric.NewMeterProvider(
        metric.WithReader(metric.NewPeriodicReader(exporter)),
        metric.WithResource(res),
    )
    defer mp.Shutdown(ctx)
    otel.SetMeterProvider(mp)

    db, err := ydb.Open(ctx,
        os.Getenv("YDB_CONNECTION_STRING"),
        ydbOtel.WithMetrics(
            mp.Meter("ydb-go-sdk"),
            ydbOtel.WithDetailer(trace.DetailsAll),
        ),
    )
    if err != nil {
        panic(err)
```

```
    }
    defer db.Close(ctx)
}
```

Java

> **Note**
>
> In the Java SDK, only session pool metrics are currently supported.

Pass your `OpenTelemetry` to the SDK via the `OpenTelemetryMeter` adapter and the `QueryClient.Builder#withMeter` method:

```
import io.opentelemetry.api.OpenTelemetry;
import io.opentelemetry.exporter.otlp.metrics.OtlpGrpcMetricExporter;
import io.opentelemetry.sdk.OpenTelemetrySdk;
import io.opentelemetry.sdk.metrics.SdkMeterProvider;
import io.opentelemetry.sdk.metrics.export.PeriodicMetricReader;

import tech.ydb.core.grpc.GrpcTransport;
import tech.ydb.core.metrics.OpenTelemetryMeter;
import tech.ydb.query.QueryClient;

SdkMeterProvider meterProvider = SdkMeterProvider.builder()
    .registerMetricReader(PeriodicMetricReader
        .builder(OtlpGrpcMetricExporter.getDefault())
        .build())
    .build();

OpenTelemetry openTelemetry = OpenTelemetrySdk.builder()
    .setMeterProvider(meterProvider)
    .build();

try (GrpcTransport transport = GrpcTransport
        .forConnectionString(System.getenv("YDB_CONNECTION_STRING"))
        .build();
     QueryClient queryClient = QueryClient.newClient(transport)
        .withMeter(OpenTelemetryMeter.fromOpenTelemetry(openTelemetry))
        .sessionPoolName("my-app")
        .build()) {
    // Use queryClient
}
meterProvider.close();
```

If you have already configured a global `GlobalOpenTelemetry`, you can use `OpenTelemetryMeter.createGlobal()`. `TableClient` also supports `withMeter(...)`.

Python

This functionality is not currently supported.

C#

Add the NuGet package:

```
dotnet add package Ydb.Sdk.OpenTelemetry
```

Register the YDB instrumentation in the OpenTelemetry metrics pipeline:

```
services.AddOpenTelemetry()
    .WithMetrics(builder => builder
        .AddYdb()
        .AddOtlpExporter());
```

Or using a standalone `MeterProvider`:

```
using var meterProvider = Sdk.CreateMeterProviderBuilder()
    .SetResourceBuilder(ResourceBuilder.CreateDefault().AddService("my-service"))
    .AddYdb()
    .AddOtlpExporter()
    .Build();
```

The session pool name is set by the `PoolName=` parameter in the `YdbDataSource` connection string. A full example with load and Grafana is in the SDK repository (Ydb.Sdk.AdoNet.OpenTelemetry/Metrics).

JavaScript

Install the YDB JavaScript SDK telemetry package and OpenTelemetry SDK:

```
npm install @ydbjs/telemetry @opentelemetry/sdk-node @opentelemetry/exporter-metrics-otlp-http
```

Initialize `NodeSDK` before creating `Driver` and register the instrumentation `@ydbjs/telemetry`:

```
import { NodeSDK } from '@opentelemetry/sdk-node'
import { OTLPMetricExporter } from '@opentelemetry/exporter-metrics-otlp-http'
```

```
import { PeriodicExportingMetricReader } from '@opentelemetry/sdk-metrics'
import { Driver } from '@ydbjs/core'
import { query } from '@ydbjs/query'
import { register } from '@ydbjs/telemetry'

const sdk = new NodeSDK({
    serviceName: 'my-service',
    metricReader: new PeriodicExportingMetricReader({
        exporter: new OTLPMetricExporter({
            url: 'http://localhost:4318/v1/metrics',
        }),
    }),
})
sdk.start()

// Must be called BEFORE creating the Driver: the package subscribes to
// diagnostics_channel SDK events and converts them to OTel metrics.
const instrumentation = register()

const driver = new Driver(process.env.YDB_CONNECTION_STRING)
await driver.ready()

const sql = query(driver)
await sql`SELECT 1`

instrumentation.disable()
await driver.close()
await sdk.shutdown()
```

Package `@ydbjs/telemetry` subscribes to events `node:diagnostics_channel` from `@ydbjs/core`, `@ydbjs/query`, `@ydbjs/auth`, and `@ydbjs/retry` and publishes metrics from the list above. Operation duration is exported as `db.client.operation.duration` (OpenTelemetry semantic conventions), not `ydb.client.operation.duration`. The current number of requests waiting for a session is gauge `ydb.query.session.acquire.pending`, not counter `ydb.query.session.pending_requests`. For more details, see the JavaScript SDK repository.

In addition, the JavaScript SDK publishes its own retry metrics with names and semantics different from `ydb.client.retry.*`:

Name	Type	Unit	Description
`ydb.retry.attempts`	Counter	`{attempt}`	Monotonic counter of retry attempts with outcome tag `ydb.retry.outcome` (e.g., `success`, `retried`). Unlike the `ydb.client.retry.attempts` histogram, it does not show the distribution of the number of attempts per query.
`ydb.retry.duration`	Histogram	`s`	Total duration of the retry cycle, including backoff (analogous to `ydb.client.retry.duration`).

Rust

This functionality is not currently supported.

Track progress or vote for support in the Rust SDK: ydb-rs-sdk#268

PHP

This functionality is not currently supported.

Enabling metrics in Prometheus

Below are code examples for enabling metrics in Prometheus in different YDB SDKs.

C++

This feature is not currently supported.

Go

Native SDK

```
package main

import (
    "context"

    "github.com/prometheus/client_golang/prometheus"
    metrics "github.com/ydb-platform/ydb-go-sdk-prometheus/v2"
    "github.com/ydb-platform/ydb-go-sdk/v3"
    "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
    ctx := context.Background()
    registry := prometheus.NewRegistry()
    db, err := ydb.Open(ctx,
        os.Getenv("YDB_CONNECTION_STRING"),
        metrics.WithTraces(
            registry,
            metrics.WithDetails(trace.DetailsAll),
            metrics.WithSeparator("_"),
        ),
    )
    if err != nil {
        panic(err)
    }
    defer db.Close(ctx)
    ...
}
```

database/sql

```
package main

import (
    "context"
    "database/sql"

    "github.com/prometheus/client_golang/prometheus"
    metrics "github.com/ydb-platform/ydb-go-sdk-prometheus/v2"
    "github.com/ydb-platform/ydb-go-sdk/v3"
    "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
    ctx := context.Background()
    registry := prometheus.NewRegistry()
    nativeDriver, err := ydb.Open(ctx,
        os.Getenv("YDB_CONNECTION_STRING"),
        metrics.WithTraces(
            registry,
            metrics.WithDetails(trace.DetailsAll),
            metrics.WithSeparator("_"),
        ),
    )
    if err != nil {
        panic(err)
    }
    defer nativeDriver.Close(ctx)

    connector, err := ydb.Connector(nativeDriver)
    if err != nil {
        panic(err)
    }

    db := sql.OpenDB(connector)
    defer db.Close()
    ...
}
```

Java

This functionality is not currently supported.

Python

This functionality is not currently supported.

C#

This functionality is not currently supported.

JavaScript

This functionality is not currently supported.

Rust

This functionality is not currently supported.

Track progress or vote for support in the Rust SDK: ydb-rs-sdk#267

PHP

This functionality is not currently supported.

Tracing with OpenTelemetry

YDB SDK instruments Query Service operations with OpenTelemetry spans, providing distributed tracing from application code to each gRPC call to YDB. Spans are exported via the standard OTLP protocol and are compatible with Jaeger, Grafana Tempo, Zipkin, and any other backend that supports OpenTelemetry.

Created spans

Currently, spans are supported for Query Service operations. Support for topics and other services is planned in future SDK versions.

The following spans are created:

Span	Type	Description
`ydb.RunWithRetry`	`Internal`	Covers the entire retry cycle for a single operation
`ydb.Try`	`Internal`	One span per attempt, including the first; child RPC spans are attached to it
`ydb.CreateSession`	`Client`	Session creation via gRPC `CreateSession` and `AttachStream`
`ydb.ExecuteQuery`	`Client`	Execution of a single YQL query
`ydb.BeginTransaction`	`Client`	Explicit call to begin a transaction
`ydb.Commit`	`Client`	Transaction commit
`ydb.Rollback`	`Client`	Transaction rollback
`ydb.Driver.Initialize`	`Internal`	Driver initialization: cluster discovery and authentication

A typical span tree for a transactional operation with retry looks as follows:

```
ydb.RunWithRetry  (Internal)
├─ ydb.Try        (Internal)   ← 1st attempt: ERROR
│  ├─ ydb.ExecuteQuery (Client)
│  ├─ ydb.ExecuteQuery (Client)
│  └─ ydb.Commit       (Client) ← ERROR: Transaction Lock Invalidated
└─ ydb.Try        (Internal)   ← 2nd attempt: SUCCESS, ydb.retry.backoff_ms=50
   ├─ ydb.ExecuteQuery (Client)
   ├─ ydb.ExecuteQuery (Client)
   └─ ydb.Commit       (Client)
```

`ydb.RunWithRetry` is the parent span for the entire operation with retries. For each execution attempt, a separate child span `ydb.Try` is created: the first `ydb.Try` corresponds to the first attempt, the second to the first **retry**, and so on. RPC spans of a specific attempt, e.g., `ydb.ExecuteQuery` and `ydb.Commit`, are created inside the corresponding `ydb.Try`.

> **Note**
>
> If an attempt fails, its span `ydb.Try` ends with an error status. On retry, a new `ydb.Try` is created; starting from the second attempt, the attribute `ydb.retry.backoff_ms` is set — the wait time before this attempt in milliseconds. This wait is included in the duration of the next span `ydb.Try`: the span starts before the backoff pause, then after the pause, the RPC calls of this attempt are executed.

Span attributes

SDK uses both standard OpenTelemetry semantic conventions attributes and YDB-specific extensions.

Standard OpenTelemetry attributes

The following attributes belong to the stable OpenTelemetry semantic conventions and can be processed by tracing backends as standard:

Attribute	Where set	Description
`db.system.name`	RPC spans	Always `"ydb"`
`db.namespace`	RPC spans	YDB database path
`server.address`	RPC spans	Primary host from the connection string
`server.port`	RPC spans	Primary port from the connection string
`network.peer.address`	RPC spans	Actual gRPC endpoint used for the call
`network.peer.port`	RPC spans	Actual port of the gRPC endpoint used for the call
`error.type`	Spans that ended with an error	Error type. For example: `"transport_error"`, `"ydb_error"`, or the full exception class name
`db.response.status_code`	RPC spans with `YdbException`	Text name of the YDB status from the error, e.g., `ABORTED`, `UNAVAILABLE`, `OVERLOADED`

YDB-specific attributes

The following attributes are YDB extensions on top of the standard semantic conventions:

Attribute	Where set	Description

`ydb.node.id`	RPC spans	ID of the YDB node that processed the request
`ydb.node.dc`	RPC spans	Datacenter of the YDB node that processed the request
`ydb.retry.backoff_ms`	Spans `ydb.Try`, starting from the second attempt	Wait time before retry in milliseconds

W3C trace context

SDK automatically propagates the W3C `traceparent` header in every outgoing gRPC call. This allows the YDB server to trace internal operations within the same trace — without additional configuration. For more details on server-side tracing, see the section Passing an external trace-id to YDB.

Connecting to the SDK

C++

Include the OpenTelemetry tracing header from YDB C++ SDK and add a dependency on the OTel C++ SDK:

```
#include <ydb-cpp-sdk/client/driver/driver.h>
#include <ydb-cpp-sdk/open_telemetry/trace.h>

#include <opentelemetry/exporters/otlp/otlp_http_exporter_factory.h>
#include <opentelemetry/exporters/otlp/otlp_http_exporter_options.h>
#include <opentelemetry/sdk/trace/tracer_provider.h>
#include <opentelemetry/sdk/trace/simple_processor_factory.h>
#include <opentelemetry/sdk/resource/resource.h>
#include <opentelemetry/trace/provider.h>

namespace sdktrace = opentelemetry::sdk::trace;
namespace otlp     = opentelemetry::exporter::otlp;
namespace resource = opentelemetry::sdk::resource;
using namespace NYdb;

// 1. Initialize the OTel tracing provider
otlp::OtlpHttpExporterOptions opts;
opts.url = "http://localhost:4318/v1/traces";
auto exporter  = otlp::OtlpHttpExporterFactory::Create(opts);
auto processor = sdktrace::SimpleSpanProcessorFactory::Create(std::move(exporter));
auto res       = resource::Resource::Create({{"service.name", "my-service"}});
auto otelProvider = std::make_shared<sdktrace::TracerProvider>(
    std::move(processor), res);
opentelemetry::trace::Provider::SetTracerProvider(otelProvider);

// 2. Wrap in the YDB tracing provider
auto ydbTraceProvider = NTrace::CreateOtelTraceProvider(otelProvider);

// 3. Create a YDB driver with tracing enabled
auto driverConfig = TDriverConfig()
    .SetEndpoint("localhost:2136")
    .SetDatabase("/local")
    .SetTraceProvider(ydbTraceProvider);

TDriver driver(driverConfig);
```

Go

Install the OpenTelemetry adapter for the YDB Go SDK:

```
go get github.com/ydb-platform/ydb-go-sdk-otel
```

Configure `TracerProvider` and pass the adapter to `ydb.Open`:

```
package main

import (
    "context"
    "os"

    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracegrpc"
    "go.opentelemetry.io/otel/sdk/resource"
    sdktrace "go.opentelemetry.io/otel/sdk/trace"

    "github.com/ydb-platform/ydb-go-sdk/v3"
    ydbOtel "github.com/ydb-platform/ydb-go-sdk-otel"
)

func main() {
    ctx := context.Background()

    exporter, err := otlptracegrpc.New(ctx,
        otlptracegrpc.WithEndpoint("localhost:4317"),
        otlptracegrpc.WithInsecure(),
    )
    if err != nil {
        panic(err)
```

```
    }
    res, _ := resource.Merge(resource.Default(), resource.NewSchemaless(
        attribute.String("service.name", "my-service"),
    ))
    tp := sdktrace.NewTracerProvider(
        sdktrace.WithBatcher(exporter),
        sdktrace.WithResource(res),
    )
    defer tp.Shutdown(ctx)
    otel.SetTracerProvider(tp)

    db, err := ydb.Open(ctx,
        os.Getenv("YDB_CONNECTION_STRING"),
        ydbOtel.WithTraces(
            ydbOtel.WithTracer(tp.Tracer("ydb-go-sdk")),
        ),
    )
    if err != nil {
        panic(err)
    }
    defer db.Close(ctx)
}
```

Java

Add YDB SDK and OpenTelemetry dependencies (example for Maven):

```
<dependency>
    <groupId>tech.ydb</groupId>
    <artifactId>ydb-sdk-core</artifactId>
    <version>${ydb.sdk.version}</version>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-sdk</artifactId>
    <version>${otel.version}</version>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-otlp</artifactId>
    <version>${otel.version}</version>
</dependency>
```

Create an OpenTelemetry SDK instance and pass it to the transport via `OpenTelemetryTracer`:

```
import io.opentelemetry.api.OpenTelemetry;
import io.opentelemetry.exporter.otlp.trace.OtlpGrpcSpanExporter;
import io.opentelemetry.sdk.OpenTelemetrySdk;
import io.opentelemetry.sdk.resources.Resource;
import io.opentelemetry.sdk.trace.SdkTracerProvider;
import io.opentelemetry.sdk.trace.export.BatchSpanProcessor;
import io.opentelemetry.semconv.resource.attributes.ResourceAttributes;
import tech.ydb.core.auth.CloudAuthHelper;
import tech.ydb.core.grpc.GrpcTransport;
import tech.ydb.core.opentelemetry.OpenTelemetryTracer;
import tech.ydb.query.QueryClient;

Resource resource = Resource.getDefault().toBuilder()
    .put(ResourceAttributes.SERVICE_NAME, "my-service")
    .build();

SdkTracerProvider tracerProvider = SdkTracerProvider.builder()
    .setResource(resource)
    .addSpanProcessor(BatchSpanProcessor.builder(
        OtlpGrpcSpanExporter.builder()
            .setEndpoint("http://localhost:4317")
            .build()
    ).build())
    .build();

OpenTelemetry openTelemetry = OpenTelemetrySdk.builder()
    .setTracerProvider(tracerProvider)
    .build();

try (GrpcTransport transport = GrpcTransport.forConnectionString(connectionString)
        .withAuthProvider(CloudAuthHelper.getAuthProviderFromEnviron())
        .withTracer(OpenTelemetryTracer.fromOpenTelemetry(openTelemetry))
        .build();
     QueryClient queryClient = QueryClient.newClient(transport).build()) {
    // Use queryClient here
}
```

When using the JDBC driver, simply add the `enableOpenTelemetryTracer=true` parameter to the connection string — the driver will automatically pick up the global OTel provider:

```
jdbc:ydb://<host>:<port>/<database>?enableOpenTelemetryTracer=true
```

Python

Install additional dependencies `opentelemetry` and the OTLP exporter:

```
pip install ydb[opentelemetry]
pip install opentelemetry-exporter-otlp-proto-grpc
```

Call `enable_tracing()` after configuring the global `TracerProvider`:

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExporter
from opentelemetry.sdk.resources import Resource

import ydb
from ydb.opentelemetry import enable_tracing

resource = Resource(attributes={"service.name": "my-service"})
provider = TracerProvider(resource=resource)
provider.add_span_processor(
    BatchSpanProcessor(OTLPSpanExporter(endpoint="http://localhost:4317"))
)
trace.set_tracer_provider(provider)

enable_tracing()

with ydb.Driver(endpoint="grpc://localhost:2136", database="/local") as driver:
    driver.wait(timeout=5)
    with ydb.QuerySessionPool(driver) as pool:
        pool.execute_with_retries("SELECT 1")

provider.shutdown()
```

C#

Add the NuGet package:

```
dotnet add package Ydb.Sdk.OpenTelemetry
```

Register the YDB instrumentation when configuring OpenTelemetry in your service:

```
services.AddOpenTelemetry()
    .WithTracing(builder => builder
        .AddYdb()
        .AddOtlpExporter());
```

JavaScript

Install `@ydbjs/telemetry` together with the OpenTelemetry Node SDK and OTLP exporter:

```
npm install @ydbjs/telemetry @opentelemetry/sdk-node @opentelemetry/exporter-trace-otlp-http
```

Initialize `NodeSDK` before creating the driver and call `register()` from `@ydbjs/telemetry`:

```
import { NodeSDK } from '@opentelemetry/sdk-node'
import { OTLPTraceExporter } from '@opentelemetry/exporter-trace-otlp-http'
import { Driver } from '@ydbjs/core'
import { query } from '@ydbjs/query'
import { register } from '@ydbjs/telemetry'

const sdk = new NodeSDK({
    serviceName: 'my-service',
    traceExporter: new OTLPTraceExporter({ url: 'http://localhost:4318/v1/traces' }),
})
sdk.start()

// Must be called BEFORE creating Driver — W3C propagation middleware
// trace context is set once during driver construction.
const instrumentation = register()

using driver = new Driver(process.env.YDB_CONNECTION_STRING)
await driver.ready()
await using sql = query(driver)
// ...

instrumentation.disable()
await sdk.shutdown()
```

Alternatively, via `--import` for auto-loading before application start:

```
node --import @opentelemetry/sdk-node/register --import @ydbjs/telemetry/register your-app.js
```

Rust

This functionality is not currently supported.

Track progress or vote for support in the Rust SDK: ydb-rs-sdk#268

PHP

This functionality is not currently supported.

Enabling tracing in Jaeger

Below are code examples for enabling tracing in Jaeger in different YDB SDKs.

C++

This feature is not currently supported.

Go

Native SDK

```
package main

import (
    "context"
    "time"

    "github.com/opentracing/opentracing-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"

    "github.com/ydb-platform/ydb-go-sdk/v3"
    "github.com/ydb-platform/ydb-go-sdk/v3/trace"

    tracing "github.com/ydb-platform/ydb-go-sdk-opentracing"
)

const (
    tracerURL   = "localhost:5775"
    serviceName = "ydb-go-sdk"
)

func main() {
    tracer, closer, err := jaegerConfig.Configuration{
        ServiceName: serviceName,
        Sampler: &jaegerConfig.SamplerConfig{
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{
            LogSpans:            true,
            BufferFlushInterval: 1 * time.Second,
            LocalAgentHostPort:  tracerURL,
        },
    }.NewTracer()
    if err != nil {
        panic(err)
    }

    defer closer.Close()

    // set global tracer of this application
    opentracing.SetGlobalTracer(tracer)

    span, ctx := opentracing.StartSpanFromContext(context.Background(), "client")
    defer span.Finish()

    db, err := ydb.Open(ctx,
        os.Getenv("YDB_CONNECTION_STRING"),
        tracing.WithTraces(tracing.WithDetails(trace.DetailsAll)),
    )
    if err != nil {
        panic(err)
    }
    defer db.Close(ctx)
    ...
}
```

database/sql

```
package main

import (
    "context"
    "database/sql"
    "time"

    "github.com/opentracing/opentracing-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"

    "github.com/ydb-platform/ydb-go-sdk/v3"
    "github.com/ydb-platform/ydb-go-sdk/v3/trace"

    tracing "github.com/ydb-platform/ydb-go-sdk-opentracing"
)

const (
    tracerURL   = "localhost:5775"
    serviceName = "ydb-go-sdk"
)
```

```
func main() {
    tracer, closer, err := jaegerConfig.Configuration{
        ServiceName: serviceName,
            Sampler: &jaegerConfig.SamplerConfig{
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{
            LogSpans:            true,
            BufferFlushInterval: 1 * time.Second,
            LocalAgentHostPort:  tracerURL,
        },
    }.NewTracer()
    if err != nil {
        panic(err)
    }

    defer closer.Close()

    // set global tracer of this application
    opentracing.SetGlobalTracer(tracer)

    span, ctx := opentracing.StartSpanFromContext(context.Background(), "client")
    defer span.Finish()

    nativeDriver, err := ydb.Open(ctx,
        os.Getenv("YDB_CONNECTION_STRING"),
        tracing.WithTraces(tracing.WithDetails(trace.DetailsAll)),
    )
    if err != nil {
        panic(err)
    }
    defer nativeDriver.Close(ctx)

    connector, err := ydb.Connector(nativeDriver)
    if err != nil {
        panic(err)
    }

    db := sql.OpenDB(connector)
    defer db.Close()
    ...
}
```

Java

This functionality is not currently supported.

Python

This functionality is not currently supported.

C#

This functionality is not currently supported.

JavaScript

This functionality is not currently supported.

Rust

This functionality is not currently supported.

For distributed tracing, use `tracing` and export via OpenTelemetry (#268).

PHP

This functionality is not currently supported.

Comparison of SDK features

This section allows you to compare the capabilities of the YDB SDK, which are implemented for different programming languages.

Common features

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
support connection string 'grpc[s]://endpoint:port/database'	+	+	+	+	+	–	+	–
SSL/TLS support (system certificates)	+	+	+	+	+	+	+	+
SSL/TLS support (custom certificates)	+	+	+	+	+	+	+	+
Configure/enable GRPC KeepAlive (keeping the connection alive in the background)	+	+	+	+	–	–	+	+
Regular SLO testing on the latest code version	+	+	+	+	–	+	+	–
Issue templates on GitHub	–	+	+	–	+	+	+	+

Client-side balancing

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
Load balancer initialization through Discovery/ListEndpoints	+	+	+	+	+	+	+	+
Disable client-side load balancing (all requests to the initial Endpoint)	+	+	+	–	+	+	+	+
Background Discovery/ListEndpoints (by default, once a minute)	+	+	+	+	+	+	+	+
Support for multiple IP addresses in the initial Endpoint DNS record, Endpoint DNS record, some of which may not be available (DNS load balancing)	+	+	+	–	+	–	+	+
Connection pessimization on transport errors	+	+	+	+	+	+	+	–
Forced Discovery/ListEndpoints if more than half of the nodes are pessimized	+	+	+	+	–	+	+	–
Automatic detection of the nearest DC/availability zone by TCP pings	–	+	+	+	–	+	+	–
Automatic detection of the nearest DC/availability zone by Discovery/ListEndpoints response	+	+	+	+	–	–	–	–
Uniform random selection of nodes (default)	+	+	+	+	+	+	+	+
Load balancing across all nodes of all DCs (default)	+	+	+	+	+	+	+	+
Load balancing across all nodes of a particular DC/availability zone (for example, “a”, “vla”)	+	+	+	+	–	–	–	–
Load balancing across all nodes of all local DCs	+	+	+	+	–	–	–	–
Route all requests into the initial endpoint instead of client-side load balancing (for serverless usage)	+	+	+	+	+	+	+	+

Credentials providers

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
Anonymous (default)	+	+	+	+	+	+	+	+
Static (user - password)	+	+	+	+	+	+	+	+
Token: IAM, Access token	+	+	+	+	+	+	+	+
Service account (Yandex.Cloud specific)	+	+	+	+	+	+	+	+
Metadata (Yandex.Cloud specific)	+	+	+	+	+	+	+	+

Support for YDB data types

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP

Int/Uint(8,16,32,64)	+	+	+	+	+	+	+	+
Float, Double	+	+	+	+	+	+	+	+
Bool	+	+	+	+	+	+	+	+
String, Bytes	+	+	+	+	+	+	+	+
Utf8, Text	+	+	+	+	+	+	+	+
NULL, Optional, Void	+	+	+	+	+	–	+	–
Struct	+	+	+	+	+	+	+	+
List	+	+	+	+	+	–	+	+
Set	–	+	+	–	–	–	–	–
Tuple	+	+	+	+	+	+	+	+
Variant<Struct>,Variant<Tuple>	+	+	+	+	+	–	–	–
Date, DateTime, Timestamp, Interval	+	+	+	+	+	+	+	–
TzDate, TzDateTime, TzTimestamp	+	+	+	+	+	–	–	–
DyNumber	+	+	+	+	+	–	–	–
Decimal (120 bits)	+	+	+	+	+	+	–	+
Json, JsonDocument, Yson	+	+	+	+	+	+	+	–
Wide date types (Date32, Datetime64, Timestamp64, Interval64)	+	+	+	+	–	+	+	–

YDB gRPC services clients

Discovery

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
ListEndpoints	+	–	+	–	+	–	–	+
WhoAmI	+	–	+	–	+	–	–	+

Query

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
Session pool	+	+	+	+	+	+	+	–
Limit the number of concurrent sessions on the client	+	+	+	+	–	+	+	–
Minimum number of sessions in the pool	+	–	+	+	–	+	+	–
Warm up the pool to the specified number of sessions when the pool is created	–	–	+	–	–	–	+	–
Background session attach stream used for clear pool from bad sessions	+	+	+	+	–	+	+	–
Retryer on the session pool (a repeat object is a session)	+	+	+	+	–	+	+	–
Retryer on the session pool (a repeat object is a transaction)	–	+	+	–	–	–	+	–
Support for server-side load balancing of sessions (a CreateSession request must contain the 'session-balancer' value in the 'x-ydb-client-capabilities' metadata header)	+	–	+	+	–	+	+	–
Transactions with tables and topics	+	+	+	–	+	–	+	–
* CreateSession	+	+	+	+	+	+	+	–
* DeleteSession	–	+	+	+	+	+	+	–
* AttachSession	+	+	+	+	+	+	+	–
* CommitTransaction	+	+	+	+	+	+	+	–
* BeginTransaction	+	+	+	+	+	–	+	–
* RollbackTransaction	+	+	+	+	+	+	+	–

* ExecuteQuery	+	+	+	+	+	+	+	–
* ExecuteScript	+	–	+	+	–	–	+	–
* FetchScriptResults	+	–	+	+	–	–	+	–
Execute queries without explicit session creation	+	–	+	–	+	+	+	–

Scheme

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
MakeDirectory	+	+	+	+	+	–	+	+
RemoveDirectory	+	+	+	+	+	–	+	+
ListDirectory	+	+	+	+	+	+	+	+
ModifyPermissions	+	+	+	–	+	–	–	+
DescribePath	+	+	+	+	+	–	+	+

Table

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
Session pool	+	+	+	+	+	+	+	+
Limit the number of concurrent sessions on the client	+	+	+	+	+	+	+	–
Minimum number of sessions in the pool	+	+	+	+	–	+	+	–
Warm up the pool to the specified number of sessions when the pool is created	–	+	+	–	+	–	+	–
Background KeepAlive for idle sessions in the pool	+	+	+	+	+	+	+	–
Background closing of idle sessions in the pool (redundant sessions)	+	+	+	+	–	+	+	–
Automatic dumping of a session from the pool in case of BAD_SESSION/SESSION_BUSY errors	+	+	+	+	+	+	+	+
Retryer on the session pool (a repeat object is a session)	+	+	+	+	+	+	+	+
Retryer on the session pool (a repeat object is a transaction within a session)	–	–	+	–	–	+	+	–
Graceful session shutdown support ("session-close" in "x-ydb-server-hints" metadata means to "forget" a session and not use it again)	+	+	+	+	–	+	+	–
Support for server-side load balancing of sessions (a CreateSession request must contain the "session-balancer" value in the "x-ydb-client-capabilities" metadata header)	+	+	+	+	–	+	+	–
Transactions with tables and topics	+	+	–	–	+	+	+	–
CreateSession	+	+	+	+	+	+	+	+
DeleteSession	+	+	+	+	+	+	+	+
KeepAlive	+	+	+	+	+	+	+	+
CreateTable	+	+	+	+	+	+	+	+
DropTable	+	+	+	+	+	+	+	+
AlterTable	+	+	+	+	+	–	+	+
CopyTable	+	+	+	+	–	+	+	+
CopyTables	+	+	+	+	–	+	+	+
DescribeTable	+	+	+	+	+	–	+	+
ExplainDataQuery	+	+	+	+	–	–	+	+
PrepareDataQuery	+	+	+	+	+	+	+	+

ExecuteDataQuery	+	+	+	+	+	+	+	+
* By default, server cache for all parameter requests (KeepInCache)	–	+	+	+	+	+	+	+
* A separate option to enable/disable server cache for a specific request	+	+	+	+	+	+	+	+
* Truncated result as an error (by default)	–	–	+	–	+	+	+	+
* Truncated result as an error (as an opt-in opt-out option)	–	–	+	–	+	–	+	–
ExecuteSchemeQuery	+	+	+	+	–	+	+	+
BeginTransaction	+	+	+	+	+	+	+	+
CommitTransaction	+	+	+	+	+	+	+	+
RollbackTransaction	+	+	+	+	+	+	+	+
DescribeTableOptions	+	+	+	+	–	–	+	+
StreamExecuteScanQuery	+	+	+	+	+	+	+	+
StreamReadTable	+	+	+	+	+	+	+	+
BulkUpsert	+	+	+	+	+	+	+	+
Manage table attributes	+	–	+	–	–	–	+	–

Operation

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
Consumed Units from metadata of a response to a grpc-request request (for the user to obtain this)	+	+	+	+	+	–	–	–
Obtaining OperationId of the operation for a long-polling status of operation execution	+	+	+	+	–	+	+	–

ScriptingYQL

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
ExecuteYql	+	+	+	–	–	–	–	+
ExplainYql	+	+	+	–	–	–	–	+
StreamExecuteYql	+	–	+	–	–	–	–	–

Coordination

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
CreateNode	+	+	+	+	+	–	+	–
AlterNode	+	+	+	+	+	–	+	–
DropNode	+	+	+	+	+	–	+	–
DescribeNode	+	+	+	+	+	–	+	–
Session (leader election, distributed lock)	+	+	+	+	+	–	+	–

Topic

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
CreateTopic	+	+	+	+	+	+	+	–
DescribeTopic	+	+	+	+	+	+	+	–
AlterTopic	+	+	+	+	+	+	+	–
DropTopic	+	+	+	+	+	+	+	–
StreamWrite	+	+	+	+	+	+	+	–
StreamRead	+	+	+	+	+	–	+	–
CommitOffset	+	+	+	+	–	–	–	–

Ratelimiter

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
CreateResource	+	–	+	–	–	–	–	–
AlterResource	+	–	+	–	–	–	–	–
DropResource	+	–	+	–	–	–	–	–
ListResources	+	–	+	–	–	–	–	–
DescribeResource	+	–	+	–	–	–	–	–
AcquireResource	+	–	+	–	+	–	–	–

Observability

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
SDK event logging	–	–	+	+	+	+	+	+
SDK metrics to Solomon / Monitoring	+	–	+	–	+	–	–	–
SDK metrics to Prometheus	–	–	+	–	+	–	+	–
SDK event tracing to OpenTelemetry	+	–	+	–	+	+	+	–

Examples

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
Auth with token	–	+	+	+	+	+	+	+
Auth with anonymous credentials	–	+	+	+	+	+	–	+
Auth with environ	–	–	+	+	+	–	+	+
Auth with metadata service	–	+	+	+	+	+	+	+
Auth with service account keyfile credentials	–	+	+	+	+	+	+	+
Auth with static credentials (username + password)	–	+	+	+	+	+	+	–
Basic (series)	+	+	+	+	+	+	+	+
Bulk Upsert	+	+	+	+	+	+	+	+
Containers (Struct, Variant, List, Tuple)	+	+	+	–	–	+	–	–
Pagination	+	+	+	+	–	–	–	–
Partition policies	+	–	+	–	–	–	–	–
Read table	–	–	+	–	+	–	+	+
Secondary index Workaround	+	–	+	+	–	–	–	–
Secondary index builtin	+	–	+	–	–	–	–	–
TTL	+	–	+	–	–	–	–	–
TTL Readtable	+	–	+	–	–	–	–	–
URL Shortener (serverless yandex function)	–	–	+	–	+	–	–	–
Topic reader	+	+	+	+	+	+	+	–
Topic writer	+	+	+	+	+	+	+	–

Languages and APIs

- ADO.NET - .NET Access to YDB
- JDBC driver for YDB
- YDB Model Context Protocol Server
- Generate code from YQL (sqlc-ydb)

ADO.NET - .NET Access to YDB

`Ydb.Sdk` is an ADO.NET Data Provider for YDB. It allows programs written in C#, Visual Basic, and F# to access the YDB database server. It is implemented in 100% C# code, is free, and is open source.

Documentation

- Getting Started with ADO.NET
- Installation ADO.NET
- Basic Usage with ADO.NET
- ADO.NET Connection Parameters
- ADO.NET Supported Types and Their Mappings
- Connection ADO.NET to Yandex Cloud
- Using Dapper

Getting Started with ADO.NET

The best way to use `Ydb.Sdk` is to install its NuGet package.

`Ydb.Sdk.Ado` aims to be fully ADO.NET-compatible; its API should feel almost identical to other .NET database drivers.

Here's a basic code snippet to get you started:

```
await using var ydbDataSource = new YdbDataSource("Host=localhost;Port=2136;Database=/local;MaxSessionPool=5
0");
await using var ydbConnection = await ydbDataSource.OpenConnectionAsync();

var ydbCommand = ydbConnection.CreateCommand();
ydbCommand.CommandText = """
                         SELECT series_id, season_id, episode_id, air_date, title
                         FROM episodes
                         WHERE series_id = @series_id AND season_id > @season_id
                         ORDER BY series_id, season_id, episode_id
                         LIMIT @limit_size;
                         """;
ydbCommand.Parameters.Add(new YdbParameter("series_id", DbType.UInt64, 1U));
ydbCommand.Parameters.Add(new YdbParameter("season_id", DbType.UInt64, 1U));
ydbCommand.Parameters.Add(new YdbParameter("limit_size", DbType.UInt64, 3U));

var ydbDataReader = await ydbCommand.ExecuteReaderAsync();

_logger.LogInformation("Selected rows:");
while (await ydbDataReader.ReadAsync())
{
    _logger.LogInformation(
        "series_id: {series_id}, season_id: {season_id}, episode_id: {episode_id}, air_date: {air_date}, titl
e: {title}",
        ydbDataReader.GetUint64(0), ydbDataReader.GetUint64(1), ydbDataReader.GetUint64(2),
        ydbDataReader.GetDateTime(3), ydbDataReader.GetString(4));
}
```

You can find more info about the ADO.NET API in the MSDN docs or in many tutorials on the Internet.

Installation ADO.NET

Official releases of `Ydb.Sdk` are always available on nuget.org. This is the recommended way to use `Ydb.Sdk`.

```
dotnet add package Ydb.Sdk
```

Basic Usage with ADO.NET

This article covers core ADO.NET usage scenarios for YDB, including database connections, query execution, and result processing. See the main documentation for additional details.

Data Source

The entry point for any database operation is DbDataSource.

You can create a YdbDataSource in following ways.

1. **Without parameters**:

   The following code creates a data source with default settings:

   ```
   await using var ydbDataSource = new YdbDataSource();
   ```

   In this case, the connection URL is `grpc://localhost:2136/local` with anonymous authentication.

2. **Using a connection string**:

   Create a data source with an ADO.NET connection string

   ```
   await using var ydbDataSource = new YdbDataSource(
       "Host=database-sample-grpc;Port=2135;Database=/root/database-sample");
   ```

   The data source will use URL: `grpc://database-sample-grpc:2135/root/database-sample.` The supported settings are described on the connection parameters page.

3. **Using a YdbConnectionStringBuilder**:

   ```
   var ydbConnectionBuilder = new YdbConnectionStringBuilder
   {
       Host = "localhost",
       Port = 2136,
       Database = "/local",
       UseTls = false
   };

   await using var ydbDataSource = new YdbDataSource(ydbConnectionBuilder);
   ```

   YdbConnectionStringBuilder also supports additional options beyond the connection string, such as logging and advanced authentication.

Connections

A connection to YDB is established via `YdbConnection`. You obtain connections from `YdbDataSource` using the following methods:

1. **YdbDataSource.OpenConnectionAsync**:

   Opens a connection to YDB using the parameters set on YdbDataSource (see the YDB Data Source section).

   ```
   await using var ydbConnection = await ydbDataSource.OpenConnectionAsync();
   ```

   Recommended for long-running read queries.

2. **YdbDataSource.OpenRetryableConnectionAsync**:

   Opens a connection with automatic operation retries that follow the YDB Retry Policy (see the retries section).

   ```
   await using var ydbConnection = await ydbDataSource.OpenRetryableConnectionAsync();
   ```

   Mode specifics:

   - Interactive transactions are not supported.
   - Commands (YdbCommand) created from this connection automatically retry single operations on transient errors.
   - Attempting to use a transaction will throw an exception (see the transactions section).

   > **Warning**
   >
   > Be careful with “long” reads in this mode: they may lead to Out Of Memory (OOM), because the entire result set is read into memory to obtain final statuses from the server.

3. **Not recommended**: CreateConnection/constructor:

   Using `YdbDataSource.CreateConnection` and the `YdbConnection` constructor (legacy ADO.NET API) is not recommended. If you still need it, open the connection manually:

   - With a connection string:

   ```
   await using var ydbConnection = new YdbConnection(
       "Host=database-sample-grpc;Port=2135;Database=/root/database-sample");
   await ydbConnection.OpenAsync();
   ```

   - With `YdbConnectionStringBuilder`:

```
var ydbConnectionBuilder = new YdbConnectionStringBuilder
{
    Host = "server",
    Port = 2135,
    Database = "/ru-prestable/my-table",
    UseTls = true
};
await using var ydbConnection = new YdbConnection(ydbConnectionBuilder);
await ydbConnection.OpenAsync();
```

- With `YdbDataSource.CreateConnection`:

```
await using var ydbConnection = ydbDataSource.CreateConnection();
await ydbConnection.OpenAsync();
```

Pooling

Opening a new connection to YDB is an expensive operation, so the provider uses a connection pool. When a connection object is disposed or closed, it is not actually closed — instead, it's returned to the pool managed by Ydb.Sdk.Ado. On subsequent requests, a pooled connection is reused. This makes open/close operations fast: open and close connections as needed, and avoid keeping them open unnecessarily for a long time.

> **Note**
>
> Pooling is in effect for connections opened via YdbDataSource (e.g., OpenConnectionAsync/OpenRetryableConnectionAsync), as well as for connections created manually with the YdbConnection constructor followed by OpenAsync().

> **Note**
>
> How it works under the hood: for application code, a "connection" is logical. Under the hood, operations are RPC calls over a small pool of gRPC/HTTP/2 channels. The provider also manages a table session pool. These details are transparent to the user and are controlled by pooling parameters (see Pooling settings: connection-parameters.md#pooling).

To clear the pool and close network channels to YDB nodes:

- `YdbDataSource.DisposeAsync()`: Disposes the data source. Closes all associated pools and network channels tied to the `ConnectionString`.
- `YdbConnection.ClearPool`: Immediately closes all idle connections in the pool associated with the specified connection's `ConnectionString`. Active connections are closed when returned to the pool.

```
await YdbConnection.ClearPool(ydbConnection);
```

- `YdbConnection.ClearAllPools()`: Immediately closes all idle connections in all pools. Active connections are closed when returned to the pool.

```
YdbConnection.ClearAllPools();
```

Basic SQL Execution

Once you have a `YdbConnection`, an `YdbCommand` can be used to execute SQL against it:

```
await using var ydbCommand = new YdbCommand("SELECT some_field FROM some_table", ydbConnection);
await using var ydbDataReader = await ydbCommand.ExecuteReaderAsync();

while (await ydbDataReader.ReadAsync())
{
    Console.WriteLine(ydbDataReader.GetString(0));
}
```

Other Execution Methods

Above, SQL is executed via ExecuteReaderAsync. There are various ways to execute a command, depending on the results you expect from it:

1. ExecuteNonQueryAsync: executes SQL that doesn't return any results, typically `INSERT`, `UPDATE`, or `DELETE` statements.

   > **Warning**
   >
   > YDB does not return the number of rows affected.

2. ExecuteScalarAsync: executes SQL that returns a single scalar value.
3. ExecuteReaderAsync: executes SQL that returns a full result set. Returns a `YdbDataReader`, which can be used to access the result set (as in the example above).

For example, to execute a simple SQL `INSERT` that does not return anything, you can use `ExecuteNonQueryAsync` as follows:

```
await using var ydbCommand = new YdbCommand("INSERT INTO some_table (some_field) VALUES ('Hello YDB!'u)", ydb
Connection);
```

```
await ydbCommand.ExecuteNonQueryAsync();
```

Parameters

When sending data values to the database, always consider using parameters rather than including the values in the SQL, as shown in the following example:

```
await using var ydbConnection = await ydbDataSource.OpenConnectionAsync();

var ydbCommand = ydbConnection.CreateCommand();
ydbCommand.CommandText = """
                         SELECT series_id, season_id, episode_id, air_date, title
                         FROM episodes WHERE series_id = $series_id AND season_id > $season_id
                         ORDER BY series_id, season_id, episode_id
                         LIMIT $limit_size;
                         """;
ydbCommand.Parameters.Add(new YdbParameter("$series_id", DbType.UInt64, 1U));
ydbCommand.Parameters.Add(new YdbParameter("$season_id", DbType.UInt64, 1U));
ydbCommand.Parameters.Add(new YdbParameter("$limit_size", DbType.UInt64, 3U));

var ydbDataReader = await ydbCommand.ExecuteReaderAsync();
```

SQL query parameters can be set using the `YdbParameter` class.

In this example, the parameters `$series_id`, `$season_id`, and `$limit_size` are declared within the SQL query and then added to the command using `YdbParameter` objects.

Alternative Parameter Style with `@` Prefix

Parameters can also be specified using the `@` prefix. In this case, there is no need to declare variables within the query itself. The query will look like this:

```
ydbCommand.CommandText = """
                         SELECT series_id, season_id, episode_id, air_date, title
                         FROM episodes
                         WHERE series_id = @series_id AND season_id > @season_id
                         ORDER BY series_id, season_id, episode_id
                         LIMIT @limit_size;
                         """;
ydbCommand.Parameters.Add(new YdbParameter("series_id", DbType.UInt64, 1U));
ydbCommand.Parameters.Add(new YdbParameter("season_id", DbType.UInt64, 1U));
ydbCommand.Parameters.Add(new YdbParameter("limit_size", DbType.UInt64, 3U));
```

With ADO.NET, the query will be prepared for you so that the variables match YQL. The type of each parameter will be determined by YdbDbType, in its absence — by DbType, otherwise it is derived from .NET is the type of value.

Parameter Types

YDB has a strongly-typed type system: columns and parameters have a type, and types are usually not implicitly converted to other types. This means you have to think about which type you will be sending: trying to insert a string into an integer column (or vice versa) will fail.

For more information on supported types and their mappings, see this page.

Transactions

Interactive transactions and retries are a key part of working with YDB.

YdbDataSource provides helper methods that make it easier to run code in a transaction with automatic retries.

```
await ydbDataSource.ExecuteInTransactionAsync(async ydbConnection =>
{
    var count = (int)(await new YdbCommand(ydbConnection)
        { CommandText = $"SELECT count FROM {tableName} WHERE id = 1" }
        .ExecuteScalarAsync())!;

    await new YdbCommand(ydbConnection)
    {
        CommandText = $"UPDATE {tableName} SET count = @count + 1 WHERE id = 1",
        Parameters = { new YdbParameter { Value = count, ParameterName = "count" } }
    }.ExecuteNonQueryAsync();
},
new YdbRetryPolicyConfig { MaxAttempts = 5 });
```

Retries are performed according to YDB policies (see the retries section).

Transactions with YdbConnection

You can create a transaction in the standard ADO.NET way:

```
await using var connection = await dataSource.OpenConnectionAsync();
await using var transaction = await connection.BeginTransactionAsync();
// ... commands within the transaction ...
await transaction.CommitAsync();
```

Warning

See more about Transaction Lock Invalidation)

Note

This is acceptable and recommended for long reads. Use snapshot read-only transactions for consistent snapshots; they do not take write locks and minimize conflicts.

There are two signatures of this method with a single isolation level parameter:

- `BeginTransaction(TxMode txMode)`

  The `Ydb.Sdk.Services.Query.TxMode` is a YDB specific isolation level, you can read more about it here.
- `BeginTransaction(IsolationLevel isolationLevel)`

  The `System.Data.IsolationLevel` parameter from the standard ADO.NET. The following isolation levels are supported: `Serializable` and `Unspecified`. Both are equivalent to the `TxMode.SerializableRW`.

Calling `BeginTransaction()` without parameters opens a transaction with level the `TxMode.SerializableRW`.

YDB does not support nested or concurrent transactions. At any given moment, only one transaction per connection can be in progress, and starting a new transaction while another is already running throws an exception. Therefore, there is no need to pass the `YdbTransaction` object returned by `BeginTransaction()` to commands you execute. When a transaction is started, all subsequent commands are automatically included until a commit or rollback is made. To ensure maximum portability, however, it is best to set the transaction scope for your commands explicitly.

Retries

Retries are an important part of YDB's design. The ADO.NET provider offers a flexible retry policy tailored to YDB specifics.

Recommendations for choosing an approach:

- Single write operations (without interactive transactions): use `YdbDataSource.OpenRetryableConnectionAsync`.
- Transactional scenarios: use the `YdbDataSource.ExecuteInTransactionAsync` family.
- Executing code with automatic retries outside a transaction: use the `YdbDataSource.ExecuteAsync` family.
- Long read operations: use a regular `YdbConnection`. For consistent snapshots, use snapshot read-only transactions. Avoid retry connections for "long" reads to prevent excessive result buffering.

Passing policy settings

You can pass `YdbRetryPolicyConfig` into `OpenRetryableConnectionAsync`, `ExecuteInTransactionAsync`, and `ExecuteAsync`.

- Connection with retries:

```
await using var conn = await ydbDataSource.OpenRetryableConnectionAsync(
    new YdbRetryPolicyConfig { MaxAttempts = 5 });
```

- Transaction with retries:

```
await ydbDataSource.ExecuteInTransactionAsync(
    async conn => { /* your code */ },
    new YdbRetryPolicyConfig { MaxAttempts = 5 });
```

- Executing a code block with retries:

```
await ydbDataSource.ExecuteAsync(
    async conn => { /* your code */ },
    new YdbRetryPolicyConfig { MaxAttempts = 5 });
```

Параметр	Описание	Значение по умолчанию
MaxAttempts	Total number of attempts, including the initial one. A value of 1 disables retries entirely.	10
EnableRetryIdempotence	Enables retries for statuses with unknown execution outcome on the server. Use only for idempotent operations — otherwise the operation may be applied twice.	false
FastBackoffBaseMs	Base delay (ms) for fast retries: errors that typically resolve quickly (e.g., temporary unavailability, TLI — Transaction Locks Invalidated). Exponential backoff with jitter.	5
FastCapBackoffMs	Maximum delay (ms) for fast retries. Exponential backoff with jitter will not exceed this cap.	500
SlowBackoffBaseMs	Base delay (ms) for slow retries: overload, resource exhaustion, etc. Exponential backoff with jitter.	50
SlowCapBackoffMs	Maximum delay (ms) for slow retries. Exponential backoff with jitter will not exceed this cap.	5000

Custom retry policy

For edge cases, you can implement your own policy by implementing `Ydb.Sdk.Ado.Retry.IRetryPolicy`. The policy receives a `YdbException` and the current attempt number and must return whether to retry and with what delay.

> **Warning**
>
> If you choose this approach, be certain you understand what you're doing: you are opting out of well-tuned default settings.

Error Handling

> **Note**
>
> This section is relevant if you do not use the provider's built-in retries (see the retries section).

All exceptions related to database operations are subclasses of `YdbException`.

To safely handle errors that might occur during command execution, you can use a `try-catch` block. Here is an example:

```
try
{
    await command.ExecuteNonQueryAsync();
}
catch (YdbException e)
{
    Console.WriteLine($"Error executing command: {e}");
}
```

Properties of `YdbException`

The `YdbException` exception has the following properties, which can help you handle errors properly:

- `IsTransient` returns `true` if the error is temporary and can be resolved by retrying. For example, this might occur in cases of a transaction lock violation when the transaction fails to complete its commit.
- `IsTransientWhenIdempotent` returns `true` if the error is temporary and can be resolved by retrying the operation, provided that the database operation is idempotent.
- `StatusCode` contains the database error code, which is helpful for logging and detailed analysis of the issue.

> **Warning**
>
> Please note that ADO.NET does not automatically retry failed operations, and you must implement retry logic in your code.

Examples

Examples are provided on GitHub at link.

ADO.NET Connection Parameters

To connect to a database, the application provides a connection string that specifies parameters such as the host, user, password, and so on. Connection strings have the form `keyword1=value; keyword2=value;`. For more information, see the official doc page on connection strings.

All available connection parameters are defined as properties in the `YdbConnectionStringBuilder`.

Below are the connection string parameters that `Ydb.Sdk.Ado` supports.

Basic Connection

Parameter	Description	Default value
Host	Host of the YDB server.	localhost
Port	Port of the YDB server.	2136
Database	Database path.	/local
User	User name.	Not defined
Password	User password.	Not defined

Security and Encryption

Parameter	Description	Default value
UseTls	Determines whether to establish a secure connection using TLS (`grpcs`); when false, TLS is not used (`grpc`).	false
RootCertificate	Path to a trusted root/intermediate server certificate file (PEM). When set, `UseTls` is automatically set to true.	Not defined

Pooling

Parameter	Description	Default value
MinPoolSize	Minimum session pool size.	0
MaxPoolSize	Maximum session pool size.	100
SessionIdleTimeout	Time to wait (seconds) before closing idle sessions in the pool when the total number of sessions exceeds `MinPoolSize`.	300

Keepalive

Parameter	Description	Default value
ConnectTimeout	Time to wait (in seconds) when establishing a connection to the server. Must be greater than or equal to 0. Set 0 for infinite wait.	10
CreateSessionTimeout	Time to wait (in seconds) when creating a new session. Must be greater than or equal to 0. Set 0 for infinite wait.	5
KeepAlivePingDelay	Idle period (in seconds) after which a keepalive ping is sent when there is no traffic. This property is used together with the `KeepAlivePingTimeout` parameter to check whether the connection is broken. The value must be greater than or equal to 1 second. Set 0 to disable the keep alive connectivity check.	10
KeepAlivePingTimeout	Time to wait (in seconds) after sending a keepalive ping; if no messages are received during this time, the connection is closed. The time to wait must be greater than or equal to 1 second. Set 0 to disable the timeout for maintaining the connection in idle mode.	10

Performance

Параметр	Описание	Значение по умолчанию
EnableMultipleHttp2Connections	Determines whether to automatically scale HTTP/2 connections within a single gRPC channel to a single cluster node. This is rarely required, but can improve performance in scenarios with high load on a single node. When false, a single HTTP/2 connection per node is used.	false
MaxSendMessageSize	Maximum size of outgoing messages in bytes. Note: a 64 MB limit applies on the server; larger messages are rejected with the `ResourceExhausted` error.	67108864 (64 MB)
MaxReceiveMessageSize	Maximum size of incoming messages, bytes.	67108864 (64 MB)
DisableServerBalancer	Determines whether to disable session balancing on the server side; when false, balancing is enabled.	false

Parameters useful for Serverless applications

Параметр	Описание	Значение по умолчанию
DisableDiscovery	Determines whether to disable automatic node discovery and use a direct gRPC connection to the address from the connection string; when false, node discovery (discovery) is performed for client-side load balancing.	false
EnableImplicitSession	Determines whether implicit session management is enabled: the server creates and deletes sessions for each operation; on the client, a session pool is not used. **In this mode, interactive client transactions are not supported.**	false

Connection Builder Parameters

There are also additional parameters that do not participate in forming the connection string. These can only be specified using `YdbConnectionStringBuilder`:

Parameter	Description	Default value
`LoggerFactory`	This parameter accepts an instance that implements the ILoggerFactory interface. The `ILoggerFactory` is a standard interface for logging factories in .NET. It is possible to use popular logging frameworks such as NLog, serilog, log4net	`NullLoggerFactory.Instance`
`CredentialsProvider`	An authentication provider that implements the `Ydb.Sdk.Auth.ICredentialsProvider`. Standard ways for authentication: 1) `Ydb.Sdk.Auth.TokenProvider`. Token authentication for OAuth-like tokens. 2) For Yandex Cloud specific authentication methods, consider using **ydb-dotnet-yc**	`Anonymous`
`ServerCertificates`	Specifies custom server certificates used for TLS/SSL validation. This is useful when working with cloud providers (e.g., Yandex Cloud) that use custom root or intermediate certificates not trusted by default	Not defined

ADO.NET Supported Types and Their Mappings

The following lists the built-in mappings for reading and writing CLR types to YDB types.

Type Mapping Table for Reading

The following shows the mappings used when reading values.

- These are the return types when using `YdbCommand.ExecuteScalarAsync()`, `YdbDataReader.GetValue()`, and similar methods.
- You can read as other types by calling `YdbDataReader.GetFieldValue<T>()` and also `YdbDataReader.GetInt32()` / `YdbDataReader.GetInt64()` to get the raw value.

YDB тип	.NET тип	Non-default .NET типы
Bool	bool	
Int8	sbyte	
Int16	short	
Int32	int	
Int64	long	
Uint8	byte	
Uint16	ushort	
Uint32	uint	
Uint64	ulong	
Float	float	
Double	double	
Decimal (precision, scale)	decimal (см. раздел Decimal)	
Bytes (синоним `String`)	byte[]	
Text (синоним `Utf8`)	string	
Json	string	
JsonDocument	string	
Yson	byte[]	
Uuid	Guid	
Date	DateTime	DateOnly
Date32	DateTime	DateOnly, int (`GetInt32()` — raw value)
Datetime	DateTime	DateOnly
Datetime64	DateTime	DateOnly, long (`GetInt64()` — raw value)
Timestamp	DateTime	DateOnly
Timestamp64	DateTime	DateOnly, long (`GetInt64()` — raw value)
Interval	TimeSpan	long (`GetInt64()` — raw value)
Interval64	TimeSpan	long (`GetInt64()` — raw value)

Decimal

`Decimal (Precision, Scale)` is a parameterized data type in YDB that allows you to explicitly specify:

- `Precision` — the total number of significant digits;
- `Scale` — the number of digits after the decimal point.

For more details, see the documentation.

By default, the Decimal(22, 9) type is used. If you need to set different values for `Precision` and `Scale`, you can do this in your code.

The example below demonstrates how to store the value 1.5 in the database with the Decimal type and parameters Precision = 5 and Scale = 3.

```
await new YdbCommand(ydbConnection)
{
    CommandText = $"INSERT INTO {tableName}(Id, Decimal) VALUES (1, @Decimal);",
    Parameters = new YdbParameter { Name = "Decimal", Value = 1.5m, Precision = 5, Scale = 3 }
}.ExecuteNonQueryAsync();
```

Type Mapping Table for Writing

There are three rules that determine the YDB type sent for a parameter:

1. If the parameter's `YdbDbType` is set, it is used.
2. If the parameter's `DbType` is set, it is used.
3. If none of the above is set, the backend type will be inferred from the CLR value type.

YDB type	.NET type	Non-default .NET types	YdbDbType	DbType
Bool	bool		Bool	Boolean
Int8	sbyte		Int8	SByte
Int16	short	sbyte, byte	Int16	Int16
Int32	int	sbyte, byte, short, ushort	Int32	Int32
Int64	long	sbyte, byte, short, ushort, int, uint	Int64	Int64
Uint8	byte		Uint8	Byte
Uint16	ushort	byte	Uint16	UInt16
Uint32	uint	byte, ushort	Uint32	UInt32
Uint64	ulong	byte, ushort, uint	Uint64	UInt64
Float	float		Float	Single
Double	double	float	Double	Double
Decimal (precision, scale)	decimal		Decimal	Decimal, Currency
Bytes (synonym of `String`)	byte[]		Bytes	Binary
Text (synonym of `Utf8`)	string		Text	String, AnsiString, AnsiStringFixedLength, StringFixedLength
Json	string		Json	
JsonDocument	string		JsonDocument	
Yson	byte[]		Yson	
Uuid	Guid		Uuid	Guid
Date	DateOnly	DateTime	Date	Date
Date32		DateTime, DateOnly, int (raw value)	Date32	
Datetime		DateTime, DateOnly	Datetime	Datetime
Datetime64		DateTime, long (raw value)	Datetime64	
Timestamp	DateTime		Timestamp	DateTime2
Timestamp64		DateTime, long (raw value)	Timestamp64	
Interval	TimeSpan	long (raw value)	Interval	
Interval64		TimeSpan, long (raw value)	Interval64	
List	T[], List	T[], List	List | YdbDbType	

Note

When using List, specify the type as a bitwise OR of YdbDbType.List and the element type. For example, to specify the YdbDbType for List, use `YdbDbType.List | YdbDbType.Int32`.

Connection ADO.NET to Yandex Cloud

Installation

To use Yandex Cloud authentication in your .NET application, install the `Ydb.Sdk.Yc.Auth` NuGet package:

```
dotnet add package Ydb.Sdk.Yc.Auth
```

This package provides the necessary tools for authenticating with Yandex Cloud services.

Authentication

Supported Yandex.Cloud authentication methods:

- `Ydb.Sdk.Yc.ServiceAccountProvider`. Service account authentication, sample usage:

  ```
  var saProvider = new ServiceAccountProvider(
      saFilePath: file, // Path to file with service account JSON info
      loggerFactory: loggerFactory
  );
  ```

- `Ydb.Sdk.Yc.MetadataProvider`. Metadata service authentication, works inside Yandex Cloud VMs and Cloud Functions. Sample usage:

  ```
  var metadataProvider = new MetadataProvider(loggerFactory: loggerFactory);
  ```

Certificates

The library includes default Yandex Cloud server certificates, which are required for connectivity with dedicated YDB databases:

```
var certs = Ydb.Sdk.Yc.YcCerts.GetYcServerCertificates();
```

How to Connect with ADO.NET

To establish a secure connection to YDB using ADO.NET, configure `YdbConnectionStringBuilder` with the required authentication and TLS settings. Below is a detailed example:

```
var builder = new YdbConnectionStringBuilder
{
    // More settings ...
    UseTls = true,
    Port = 2135,
    CredentialsProvider = saProvider, // For service account
    ServerCertificates = YcCerts.GetYcServerCertificates() // custom certificates Yandex Cloud
};
```

Example

ADO.NET connect to Yandex Cloud

JDBC driver for YDB

The JDBC driver for YDB provides database connectivity for applications written in Java or other programming languages running on JVM, like Kotlin, Scala, etc.

Download

Download the JDBC driver for YDB from the latest releases page on GitHub.

Documentation

- Quick start with JDBC driver
- Using the JDBC driver with Maven
- Authentication modes
- JDBC driver properties
- Building the JDBC driver for YDB

Quick start with JDBC driver

1. Download the JDBC driver for YDB.
2. Copy the `.jar` file to the directory specified in the `CLASSPATH` environment variable or load the `.jar` file in your IDE.
3. Connect to YDB. JDBC URL examples:
   - Local Docker container with anonymous authentication and without TLS:
     `jdbc:ydb:grpc://localhost:2136/local`
   - Remote self-hosted cluster:
     `jdbc:ydb:grpcs://<host>:2135/Root/<testdb>?secureConnectionCertificate=~/<myca>.cer`
   - A cloud database instance with a token:
     `jdbc:ydb:grpcs://<host>:2135/<path/to/database>?tokenFile=~/my_token`
   - A cloud database instance with a service account:
     `jdbc:ydb:grpcs://<host>:2135/<path/to/database>?saKeyFile=~/sa_key.json`
4. Execute queries, for example, YdbDriverExampleTest.java.

Using the JDBC driver with Maven

The recommended way to use the YDB JDBC driver in a project is to include it as a Maven dependency. Specify the YDB JDBC driver in the `dependencies` section of `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>tech.ydb.jdbc</groupId>
        <artifactId>ydb-jdbc-driver</artifactId>
        <version><!-- actual version --></version>
    </dependency>
</dependencies>
```

Authentication modes

The JDBC Driver for YDB supports the following authentication modes:

- **Anonymous** is used when a username and password are not specified and no other authentication properties are configured. No authentication credentials are provided.
- **Static credentials** are used when a username and password are specified.
- **Access token** is used when the `token` or `tokenFile` property is configured. This authentication method requires a YDB authentication token, which can be obtained by executing the following YDB CLI command: `ydb auth get-token`.
- **Metadata** is used when the `useMetadata` property is set to `true`. This method extracts the authentication data from the metadata of a virtual machine, serverless container, or serverless function running in a cloud environment.
- **Service Account Key** is used when the `saFile` property is configured. This method extracts the service account key from a file and uses it for authentication.

JDBC driver properties

The JDBC driver for YDB supports the following configuration properties, which can be specified in the JDBC URL or passed via additional properties:

- `saKeyFile` — a path to the service account key file for authentication.
- `iamEndpoint` — custom IAM endpoint for authentication using a service account key.
- `token` — token value for authentication.
- `tokenFile` — a path to the file with token value for authentication.
- `useMetadata` — indicates whether to use metadata authentication. Valid values are:
  - `true` — use metadata authentication.
  - `false` — do not use metadata authentication.

  Default value: `false`.
- `metadataURL` — custom metadata endpoint.
- `localDatacenter` — the name of the data center local to the application being connected.
- `secureConnectionCertificate` — a path to the custom CA certificate file for TLS connections.

> **Note**
>
> The values of the `saKeyFile`, `tokenFile`, or `secureConnectionCertificate` properties can be either absolute paths from the file system root or relative paths from the user's home directory. Examples:
>
> - `saKeyFile=~/mysakey1.json`
> - `tokenFile=/opt/secret/token-file`
> - `secureConnectionCertificate=/etc/ssl/cacert.cer`

JDBC URL examples

- Local Docker container with anonymous authentication and without TLS:
  `jdbc:ydb:grpc://localhost:2136/local`
- Remote self-hosted cluster:
  `jdbc:ydb:grpcs://<host>:2135/Root/<testdb>?secureConnectionCertificate=~/<myca>.cer`
- A cloud database instance with a token:
  `jdbc:ydb:grpcs://<host>:2135/<path/to/database>?tokenFile=~/my_token`
- A cloud database instance with a service account:
  `jdbc:ydb:grpcs://<host>:2135/<path/to/database>?saKeyFile=~/sa_key.json`

Building the JDBC driver for YDB

To execute all tests in the project, run the `mvn test` command.

By default, all tests are run using a local YDB instance in Docker (if the host has Docker or Docker Machine installed).

To disable these tests, run: `mvn test -DYDB_DISABLE_INTEGRATION_TESTS=true`

YDB Model Context Protocol Server

YDB Model Context Protocol (MCP) server allows you to work with YDB databases from any Large Language Model (LLM) that supports MCP using any of the MCP clients. This integration enables AI-powered database operations and natural language interactions with your YDB instances.

Getting Started

Prerequisites

1. Install an MCP client that supports MCP tools (most do). The configuration examples below use a common format supported by several popular MCP clients (Claude Desktop, Cursor, etc.), but you may need to adjust the format to meet your client's requirements.
2. The YDB MCP server is a Python application that is typically co-hosted with the MCP client. There are several options for installing and running the YDB MCP server explained below, but all of them require a pre-installed Python 3.10+ environment.

Anonymous Authentication

uvx

uvx allows you to run Python applications without explicitly installing them.

Configure YDB MCP in your MCP client settings:

```
{
  "mcpServers": {
    "ydb": {
      "command": "uvx",
      "args": [
        "ydb-mcp",
        "--ydb-endpoint", "grpc://localhost:2136/local"
      ]
    }
  }
}
```

pipx

pipx allows you to run applications from PyPI without explicit installation (pipx itself must be installed first).

Configure YDB MCP in your MCP client settings:

```
{
  "mcpServers": {
    "ydb": {
      "command": "pipx",
      "args": [
        "run", "ydb-mcp",
        "--ydb-endpoint", "grpc://localhost:2136/local"
      ]
    }
  }
}
```

pip

Optionally, create and activate a Python virtual environment. Install YDB MCP using pip:

```
pip install ydb-mcp
```

Configure YDB MCP in your MCP client settings:

```
{
  "mcpServers": {
    "ydb": {
      "command": "python3",
      "args": [
        "-m", "ydb_mcp",
        "--ydb-endpoint", "grpc://localhost:2136/local"
      ]
    }
  }
}
```

Login-Password Authentication

uvx

Configure login/password authentication with `uvx`:

```
{
  "mcpServers": {
    "ydb": {
      "command": "uvx",
      "args": [
        "ydb-mcp",
        "--ydb-endpoint", "grpc://localhost:2136/local",
        "--ydb-auth-mode", "login-password",
```

```
        "--ydb-login", "<your-username>",
        "--ydb-password", "<your-password>"
      ]
    }
  }
}
```

pipx

Configure login/password authentication with `pipx`:

```
{
  "mcpServers": {
    "ydb": {
      "command": "pipx",
      "args": [
        "run", "ydb-mcp",
        "--ydb-endpoint", "grpc://localhost:2136/local",
        "--ydb-auth-mode", "login-password",
        "--ydb-login", "<your-username>",
        "--ydb-password", "<your-password>"
      ]
    }
  }
}
```

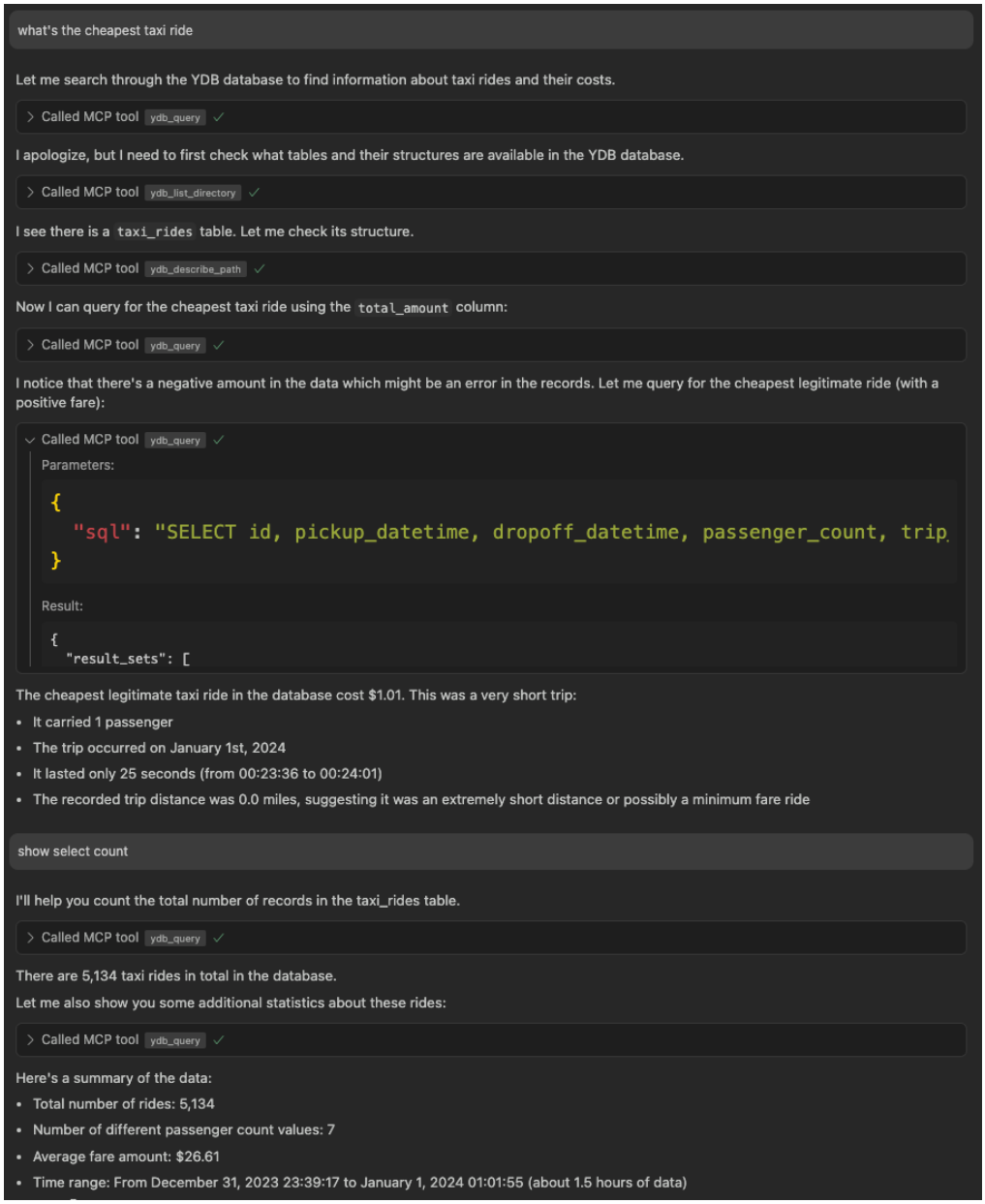
pip

Configure login/password authentication with `pip`-installed YDB MCP:

```
{
  "mcpServers": {
    "ydb": {
      "command": "python3",
      "args": [
        "-m", "ydb_mcp",
        "--ydb-endpoint", "grpc://localhost:2136/local",
        "--ydb-auth-mode", "login-password",
        "--ydb-login", "<your-username>",
        "--ydb-password", "<your-password>"
      ]
    }
  }
}
```

Run Queries

Ask your LLM questions regarding the data stored in YDB using the MCP client configured above. The language model will see the tools available to it via MCP and will use them to execute YQL queries and other YDB API calls. An example of how it might look:



Available Tools

YDB MCP provides the following tools for interacting with YDB databases:

- `ydb_query`: Run a SQL query against a YDB database
  - Parameters:
    - `sql`: SQL query string to execute
- `ydb_query_with_params`: Run a parameterized SQL query with JSON parameters
  - Parameters:
    - `sql`: SQL query string with parameters
    - `params`: JSON string containing parameter values
- `ydb_list_directory`: List directory contents in YDB
  - Parameters:
    - `path`: YDB directory path to list
- `ydb_describe_path`: Get detailed information about a scheme object (table, directory, etc) located at the specified YDB path
  - Parameters:
    - `path`: YDB path to describe
- `ydb_status`: Get the current status of the YDB connection

Command-line arguments and environment variables

The following table describes the command-line arguments and environment variables for the YDB MCP server:

Arguments	Environment variable	Default	Description
`--ydb-endpoint`	`YDB_ENDPOINT`	—	YDB endpoint consisting of protocol, hostname, port, and database name
`--ydb-login`	`YDB_LOGIN`	—	YDB login
`--ydb-password`	`YDB_PASSWORD`	—	YDB password

`--ydb-auth-mode`	`YDB_AUTH_MODE`	`anonymous`	YDB authentication mode. Valid values: `anonymous`, `login-password`
`--log-level`	—	`INFO`	Logging level. Valid values: `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`

Note

Command-line arguments override the corresponding environment variables.

Learn More

For more information visit the YDB MCP GitHub repository.

YDB AI skills (ydb-ai-skills)

ydb-ai-skills is a set of skills for AI coding agents that assist when working with YDB: writing YQL queries, designing the data schema, and reviewing application code for correct use of the YDB SDK. Skills plug into your AI agent and activate automatically based on the request context.

> **Warning**
>
> The project is under active development. The set of skills and their content are expanding, so the composition and behavior may change. See the project repository for up-to-date information.

What's inside

Skill	Purpose
ydb-core	Baseline skill: YDB overview, connection, authentication, schema basics, CLI. Installed automatically.
ydb-table	Auditing and reviewing application code that uses the YDB SDK, schema design, writing YQL.

Supported agents

Skills are distributed in a universal format and installed into the corresponding agent's directory.

Agent	Project directory	Global directory
Claude Code	`.claude/skills/`	`~/.claude/skills/`
Cursor	`.cursor/skills/`	—
Windsurf	`.windsurf/skills/`	—
GitHub Copilot	`.github/skills/`	`~/.copilot/skills/`
Codex CLI	`.agents/skills/`	`~/.codex/skills/`
Roo Code	`.roo/skills/`	—
Gemini CLI	`.gemini/skills/`	`~/.gemini/skills/`
Amp	`.agents/skills/`	`~/.config/agents/skills/`
Kiro	`.kiro/skills/`	—
Trae	`.trae/skills/`	—
Generic	`.agents/skills/`	`~/.agents/skills/`

Installation

```
git clone https://github.com/ydb-platform/ydb-ai-skills.git
cd ydb-ai-skills

# Auto-detect agents in the current project
./install.sh --detect

# Install for a specific agent
./install.sh --agent=claude

# Install only ydb-table (ydb-core is added as the baseline skill)
./install.sh --agent=claude --skills=ydb-table

# Install only ydb-table without the baseline skill
./install.sh --agent=claude --skills=ydb-table --no-core

# Dry run — show what would be done without making changes
./install.sh --agent=claude --dry-run
```

Usage

No separate action is required to start: the agent connects the appropriate skill itself when the task concerns YDB. If the agent does not support auto-activation, specify the skill name (`ydb-core` or `ydb-table`) in the request explicitly.

Code review

The `ydb-table` skill checks application code for common mistakes when working with the YDB SDK and points to the violated rules. To run a review, point the agent at the file or the range of changes.

Local run with Ollama

You can use the skills with an agent running on a local model via Ollama, without calling the cloud API. The `ollama launch` command is available starting from Ollama v0.15.

```
# 1. Pull a local model (tool calling support and a large context are required)
ollama pull gpt-oss:20b

# 2. Run Claude Code on that model — the skills activate as usual
ollama launch claude --model gpt-oss:20b
```

For agentic workflows, a context length of at least 64,000 tokens is recommended (configured in Ollama).

See also

- ydb-ai-skills repository — source code, documentation on authoring and testing skills.
- YQL - Overview
- Reference for YDB SDK
- YDB CLI

Kafka API

YDB supports working with topics using the Kafka protocol version 3.4.0. It allows to integrate YDB with applications originally developed to work with Apache Kafka.

The Kafka API documentation contains the following sections:

- Authentication
- Usage examples
- Kafka Connect
- Constraints

Kafka API

YDB supports working with topics using the Kafka protocol version 3.4.0. It allows to integrate YDB with applications originally developed to work with Apache Kafka.

The Kafka API documentation contains the following sections:

- Authentication
- Usage examples
- Kafka Connect
- Constraints

Kafka API authentication

Enabling authentication

When you run a single-node local YDB cluster, anonymous authentication is used by default. It doesn't require a username and password.

To require authentication see Authentication.

Authentication is always enabled when using the Kafka API in Yandex Cloud.

How does authentication work in the Kafka API?

The Kafka API uses the `SASL_PLAINTEXT/PLAIN`, `SASL_SSL/PLAIN` and mTLS authentication mechanisms.

Auhentication using PLAIN or SCRAM-SHA-256

The following variables are required for authentication:

- `<user-name>` — the username. For information about user management, refer to the Authorization section.
- `<password>` — the user's password. For information about user management, refer to the Authorization section.
- `<database>` — the database path.

These parameters form the following variables, which you can use in the `sasl.jaas.config` Kafka client property:

- `<sasl.username> = <user-name>@<database>`
- `<sasl.password> = <password>`

> **Note**
>
> The `<sasl.username>` and `<sasl.password>` parameters are formed differently. See examples for details.

For authentication examples, see Kafka API usage examples.

Authentication using mTLS

To enable mTLS authentication, the following steps are required.

Server and client certificates creation

For each step below the examples of commands are given. Substitute *** with your values.

1. Create Certificate Authority (CA)

```
openssl genrsa -out ca-key.pem 4096
```

```
openssl req -new -x509 -days 3650 -key ca-key.pem -out ca-cert.pem -subj "/C=***/ST=***/L=***/O=***/CN=MyKafk
aRootCA"
```

2. Create server certificate

```
openssl genrsa -out server-key.pem 4096
```

In the next command put your host name instead of `serverhost.com`.

```
openssl req -new -key server-key.pem -out server-cert.csr -subj "/C=***/ST=***/L=***/O=***/CN=serverhost.com"
```

```
cat > server-ext.cnf << EOF
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment
subjectAltName = DNS:serverhost.com
EOF
```

```
openssl x509 -req -in server-cert.csr -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out server-cert.pem
-days 365 -extfile server-ext.cnf
```

3. Create client certificate

```
openssl genrsa -out client-key.pem 4096
```

Substitute `clienthost.com` with hostname of your client.

```
openssl req -new -key client-key.pem -out client-cert.csr -subj "/C=***/ST=***/L=***/O=***/CN=clienthost.com"
```

```
cat > client-ext.cnf << EOF
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment
```

```
subjectAltName = DNS:clienthost.com
EOF
```

```
openssl x509 -req -in client-cert.csr -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out client-cert.pem
-days 365 -extfile client-ext.cnf
```

4. Add created certificates to keystore and truststore

For server:

```
openssl pkcs12 -export -in server-cert.pem -inkey server-key.pem -out server.p12 -name kafka-server -CAfile c
a-cert.pem -caname root -password pass:changeit

keytool -importkeystore -deststorepass changeit -destkeystore server.keystore.jks -srckeystore server.p12 -sr
cstoretype PKCS12 -srcstorepass changeit -alias kafka-server

keytool -import -trustcacerts -alias ca -file ca-cert.pem -keystore server.truststore.jks -storepass changeit
-noprompt
```

For client:

```
openssl pkcs12 -export -in client-cert.pem -inkey client-key.pem -out client.p12 -name kafka-client -CAfile c
a-cert.pem -caname root -password pass:changeit

keytool -importkeystore -deststorepass changeit -destkeystore client.keystore.jks -srckeystore client.p12 -sr
cstoretype PKCS12 -srcstorepass changeit -alias kafka-client

keytool -import -trustcacerts -alias ca -file ca-cert.pem -keystore client.truststore.jks -storepass changeit
-noprompt
```

After fulfilling these steps you should obtain keystore and truststore, as well as files with certificates and keys.

Client configuration

Java SDK example

```
props.put("security.protocol", "SSL");
props.put("ssl.truststore.password", "changeit");
props.put("ssl.truststore.location", "/full/path/to/client.truststore.jks");
props.put("ssl.keystore.location", "/full/path/to/client.keystore.jks");
props.put("ssl.keystore.password", "changeit");
props.put("ssl.key.password", "changeit");
props.put("ssl.endpoint.identification.algorithm", "");
```

Kafka cli example

```
security.protocol=SSL
ssl.truststore.password=changeit
ssl.truststore.location=/full/path/to/client.truststore.jks
ssl.keystore.location=/full/path/to/client.keystore.jks
ssl.keystore.password=changeit
ssl.key.password=changeit
ssl.endpoint.identification.algorithm=
```

YDB configuration

It is necessary to specify the required fields in the kafka_proxy_config.

```
kafka_proxy_config:
  enable_kafka_proxy: true
  listening_port: your_port

  mtls_enable: true
  key: "server-key.pem" # укажите правильные пути до файлов
  cert: "server-cert.pem"
  ca: "ca-cert.pem"
  enable_self_signed_certs: true # разрешаете ли вы самоподписанные сертификаты
```

In the client_certificate_authorization, specify the following authentication rules:

```
client_certificate_authorization:
  client_certificate_definitions:
    - require_same_issuer: true
      subject_terms:
        - short_name: CN
          suffixes:
            - '.myhost.net' # нужно заменить на нужный суффикс
      member_groups:
        - user@cert # заменить на нужную member группу
  request_client_certificate: true
```

Additionally, for proper operation, you need to use the same certificate as in the gRPC settings, so you need to specify the path to this server certificate in the grpc configuration.

Currently, it is not possible to configure Kafka and gRPC with different server certificates or to specify a server certificate only in the kafka_proxy_config settings when using mTLS.

```
grpc_config:
  cert: "/path/to/server-cert.pem"
```

Kafka API usage examples

This example shows a code snippet for reading data from a topic via Kafka API without a consumer group (Manual Partition Assignment).
You don't need to create a consumer for this reading mode.

Before proceeding with the examples:

1. Create a topic.
2. Add a consumer.
3. If authentication is enabled, create a user.

How to try the Kafka API

In Docker

Run Docker following the quickstart guide, and the Kafka API will be available on port 9092.

Kafka API usage examples

Reading

YDB Topics Kafka API lacks support for the check.crcs option. Therefore, the following parameter must always be specified in the reader configuration: `check.crcs=false`.

Below are examples of reading using the Kafka protocol for various applications, programming languages, and frameworks without authentication.
For examples of how to set up authentication, see Authentication examples.

Built-in Kafka CLI tools

> **Note**
>
> If you get the `java.lang.UnsupportedOperationException: getSubject is supported only if a security manager is allowed` error when using Kafka CLI tools with Java 23, perform one of the following steps:
>
> - Run the command using a different version of Java (how to change the Java version on macOS).
> - Run the command with the Java flag `-Djava.security.manager=allow`. For example: `KAFKA_OPTS=-Djava.security.manager=allow kafka-topics --bootstrap-servers localhost:9092 --list`.

```
kafka-console-consumer --bootstrap-server localhost:9092 \
    --topic my-topic  \
    --group my-group \
    --from-beginning \
    --consumer-property check.crcs=false \
    --consumer-property partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRobinAssignor
```

kcat

```
kcat -C \
  -b <ydb-endpoint> \
  -X check.crcs=false \
  -X partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRobinAssignor \
  -G <consumer-name> <topic-name>
```

Java

```
String HOST = "<ydb-endpoint>";
String TOPIC = "<topic-name>";
String CONSUMER = "<consumer-name>";

Properties props = new Properties();

props.put("bootstrap.servers", HOST);

props.put("key.deserializer", StringDeserializer.class.getName());
props.put("value.deserializer", StringDeserializer.class.getName());

props.put("check.crcs", false);
props.put("partition.assignment.strategy", RoundRobinAssignor.class.getName());

props.put("group.id", CONSUMER);
Consumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList(new String[] {TOPIC}));

while (true) {
  ConsumerRecords<String, String> records = consumer.poll(10000); // timeout 10 sec
  for (ConsumerRecord<String, String> record : records) {
      System.out.println(record.key() + ":" + record.value());
  }
}
```

Spark

```
public class ExampleReadApp {
  public static void main(String[] args) {
```

```
    var conf = new SparkConf().setAppName("my-app").setMaster("local");
    var context = new SparkContext(conf);

    context.setCheckpointDir("checkpoints");
    SparkSession spark = SparkSession.builder()
            .sparkContext(context)
            .config(conf)
            .appName("Simple Application")
            .getOrCreate();

    Dataset<Row> df = spark
            .read()
            .format("kafka")
            .option("kafka.bootstrap.servers", "localhost:9092")
            .option("subscribe", "flink-demo-input-topic")
            .option("kafka.group.id", "spark-example-app")
            .option("startingOffsets", "earliest")
            .option("kafka." + ConsumerConfig.CHECK_CRCS_CONFIG, "false")
            .load();

    df.foreach((ForeachFunction<Row>) row -> {
        System.out.println(row);
    });
  }
}
```

In the example above, Apache Spark 2.12:3.5.3, with a dependency on `org.apache.spark:spark-streaming-kafka-0-10_2.12:3.5.3`, was used.

Flink

```
public class YdbKafkaApiReadExample {

    public static void main(String[] args) throws Exception {
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment()
                .enableCheckpointing(5000, CheckpointingMode.AT_LEAST_ONCE);

        Configuration config = new Configuration();
        config.set(CheckpointingOptions.CHECKPOINT_STORAGE, "filesystem");
        config.set(CheckpointingOptions.CHECKPOINTS_DIRECTORY, "file:///path/to/your/checkpoints");
        env.configure(config);

        KafkaSource<String> kafkaSource = KafkaSource.<String>builder()
                .setBootstrapServers("localhost:9092")
                .setProperty(ConsumerConfig.CHECK_CRCS_CONFIG, "false")
                .setGroupId("flink-demo-consumer")
                .setTopics("my-topic")
                .setStartingOffsets(OffsetsInitializer.earliest())
                .setBounded(OffsetsInitializer.latest())
                .setValueOnlyDeserializer(new SimpleStringSchema())
                        .build();

        env.fromSource(kafkaSource, WatermarkStrategy.noWatermarks(), "kafka-source").print();

        env.execute("YDB Kafka API example read app");
    }
}
```

In the example above, Apache Flink 1.20 is used with the Flink DataStream connector for Kafka.

Writing

Built-in Kafka CLI tools

> **Note**
>
> If you get the `java.lang.UnsupportedOperationException: getSubject is supported only if a security manager is allowed` error when using Kafka CLI tools with Java 23, perform one of the following steps:
>
> - Run the command using a different version of Java (how to change the Java version on macOS).
> - Run the command with the Java flag `-Djava.security.manager=allow`. For example: `KAFKA_OPTS=-Djava.security.manager=allow kafka-topics --bootstrap-servers localhost:9092 --list`.

```
kafka-console-producer --broker-list localhost:9092 --topic my-topic
```

kcat

```
echo "test message" | kcat -P \
    -b <ydb-endpoint> \
    -t <topic-name> \
    -k key
```

Java

```
String HOST = "<ydb-endpoint>";
String TOPIC = "<topic-name>";

Properties props = new Properties();
props.put("bootstrap.servers", HOST);
props.put("acks", "all");

props.put("key.serializer", StringSerializer.class.getName());
props.put("key.deserializer", StringDeserializer.class.getName());
props.put("value.serializer", StringSerializer.class.getName());
props.put("value.deserializer", StringDeserializer.class.getName());

props.put("compression.type", "none");

Producer<String, String> producer = new KafkaProducer<>(props);
producer.send(new ProducerRecord<String, String>(TOPIC, "msg-key", "msg-body"));
producer.flush();
producer.close();
```

Spark

```
public class ExampleWriteApp {
public static void main(String[] args) {
    var conf = new SparkConf().setAppName("my-app").setMaster("local");
    var context = new SparkContext(conf);
    context.setCheckpointDir("path/to/dir/with/checkpoints");
    SparkSession spark = SparkSession.builder()
        .sparkContext(context)
          .config(conf)
          .appName("Simple Application")
          .getOrCreate();

    spark
          .createDataset(List.of("spark-1", "spark-2", "spark-3", "spark-4"), Encoders.STRING())
          .write()
          .format("kafka")
          .option("kafka.bootstrap.servers", "localhost:9092")
          .option("topic", "flink-demo-output-topic")
          .option("kafka.group.id", "spark-example-app")
          .option("startingOffsets", "earliest")
          .save();
  }
}
```

In the example above, Apache Spark 2.12:3.5.3, with a dependency on `org.apache.spark:spark-streaming-kafka-0-10_2.12:3.5.3`, was used.

Flink

```
public class YdbKafkaApiProduceExample {
  private static final String TOPIC = "my-topic";

  public static void main(String[] args) throws Exception {
      final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

      Sink<String> kafkaSink = KafkaSink.<String>builder()
              .setBootstrapServers("localhost:9092") // assuming ydb is running locally with kafka proxy on 9
092 port
              .setRecordSerializer(KafkaRecordSerializationSchema.builder()
                      .setTopic(TOPIC)
                      .setValueSerializationSchema(new SimpleStringSchema())
                      .setKeySerializationSchema(new SimpleStringSchema())
                      .build())
              .setRecordSerializer((el, ctx, ts) -> new ProducerRecord<>(TOPIC, el.getBytes()))
              .setDeliveryGuarantee(DeliveryGuarantee.AT_LEAST_ONCE)
                      .build();

      env.setParallelism(1)
              .fromSequence(0, 10)
              .map(i -> i + "")
              .sinkTo(kafkaSink);

      // Execute program, beginning computation.
      env.execute("ydb_kafka_api_write_example");
  }
}
```

In the example above, Apache Flink 1.20 is used with the Flink DataStream connector for Kafka.

Logstash

```
  output {
    kafka {
      codec => json
      topic_id => "<topic-name>"
      bootstrap_servers => "<ydb-endpoint>"
```

```
    compression_type => none
  }
}
```

Fluent Bit

```
[OUTPUT]
  name                          kafka
  match                         *
  Brokers                       <ydb-endpoint>
  Topics                        <topic-name>
  rdkafka.client.id             Fluent-bit
  rdkafka.request.required.acks 1
  rdkafka.log_level             7
  rdkafka.sasl.mechanism        PLAIN
```

Authentication examples

For more details on authentication, see the Authentication section. Below are examples of authentication in a cloud database and a local database.

> **Note**
>
> Currently, the only available authentication mechanism with Kafka API in YDB Topics is `SASL_PLAIN`.

Authentication examples in on-prem YDB

To use authentication in a multinode self-deployed database:

1. Create a user. How to do this in YQL. How to execute YQL from CLI.
2. Connect to the Kafka API as shown in the examples below. In all examples, it is assumed that:
   - YDB is running locally with the environment variable `YDB_KAFKA_PROXY_PORT=9092`, meaning that the Kafka API is available at `localhost:9092`. For example, you can run YDB in Docker as described here.
   - is the username you specified when creating the user.
   - is the user's password you specified when creating the user.

Examples are shown for reading, but the same configuration parameters work for writing to a topic as well.

Built-in Kafka CLI tools

> **Note**
>
> If you get the `java.lang.UnsupportedOperationException: getSubject is supported only if a security manager is allowed` error when using Kafka CLI tools with Java 23, perform one of the following steps:
>
> - Run the command using a different version of Java (how to change the Java version on macOS).
> - Run the command with the Java flag `-Djava.security.manager=allow`. For example: `KAFKA_OPTS=-Djava.security.manager=allow kafka-topics --bootstrap-servers localhost:9092 --list`.

```
kafka-console-consumer --bootstrap-server localhost:9092 \
--topic <topic-name>  \
--group <consumer-name> \
--from-beginning \
--consumer-property check.crcs=false \
--consumer-property partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRobinAssignor \
--consumer-property security.protocol=SASL_PLAINTEXT \
--consumer-property sasl.mechanism=PLAIN \
--consumer-property "sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required userna
me=\"<username>\" password=\"<password>\";"
```

kcat

```
kcat -C \
  -b localhost:9092 \
  -X security.protocol=SASL_PLAINTEXT \
  -X sasl.mechanism=PLAIN \
  -X sasl.username="<username>" \
  -X sasl.password="<password>" \
  -X check.crcs=false \
  -X partition.assignment.strategy=roundrobin \
  -G <consumer-name> <topic-name>
```

Java

```
String TOPIC = "<topic-name>";
String CONSUMER = "<consumer-name>";

Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");

props.put("key.deserializer", StringDeserializer.class.getName());
props.put("value.deserializer", StringDeserializer.class.getName());
```

```
props.put("check.crcs", false);
props.put("partition.assignment.strategy", RoundRobinAssignor.class.getName());

props.put("security.protocol", "SASL_PLAINTEXT");
props.put("sasl.mechanism", "PLAIN");
props.put("sasl.jaas.config", "sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule requi
red username=\"<username>\" password=\"<password>\";");

props.put("group.id", CONSUMER);
Consumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList(new String[] {TOPIC}));

while (true) {
  ConsumerRecords<String, String> records = consumer.poll(10000); // timeout 10 sec
  for (ConsumerRecord<String, String> record : records) {
      System.out.println(record.key() + ":" + record.value());
  }
}
```

Using Kafka Connect

This section provides configuration examples for popular Kafka Connect connectors to use with YDB.

Kafka Connect is a tool for streaming data between Apache Kafka® and other data stores. YDB supports its topics over the Kafka protocol, so you can use Kafka Connect connectors to work with YDB.

For more information about Kafka Connect and its configuration, see the Apache Kafka® documentation.

Configuring Kafka Connect. A step-by-step guide

This section provides a step-by-step guide for configuring a Kafka Connect connector to copy data from a YDB topic to a file.

The following placeholders are used in this guide:

- `<topic-name>` — the topic name. You can specify either the full name (including the database path) or just the topic name.
- `<sasl.username>` — the SASL username. For more details, see the Authentication section.
- `<sasl.password>` — the SASL password. For more details, see the Authentication section.

1. Create a consumer named `connect-<connector-name>`. The connector name is specified in the `name` field of its configuration file.
2. Download and unpack the Apache Kafka® archive:

   ```
   wget https://downloads.apache.org/kafka/3.6.1/kafka_2.13-3.6.1.tgz && tar -xvf kafka_2.13-3.6.1.tgz --st
   rip 1 --directory /opt/kafka/
   ```

   This example uses Apache Kafka® version `3.6.1`.
3. Create a directory for the worker process configuration:

   ```
   sudo mkdir --parents /etc/kafka-connect-worker
   ```

4. Create the worker process configuration file `/etc/kafka-connect-worker/worker.properties`:

   ```
   # Main properties
   bootstrap.servers=<ydb-endpoint>

   # AdminAPI properties
   sasl.mechanism=PLAIN
   security.protocol=SASL_SSL
   sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="<sasl.userna
   me>" password="<sasl.password>";

   # Producer properties
   producer.sasl.mechanism=PLAIN
   producer.security.protocol=SASL_SSL
   producer.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="<sa
   sl.username>" password="<sasl.password>";

   # Consumer properties
   consumer.sasl.mechanism=PLAIN
   consumer.security.protocol=SASL_SSL
   consumer.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="<sa
   sl.username>" password="<sasl.password>";

   consumer.check.crcs=false

   # Converter properties
   key.converter=org.apache.kafka.connect.storage.StringConverter
   value.converter=org.apache.kafka.connect.storage.StringConverter
   key.converter.schemas.enable=false
   value.converter.schemas.enable=false

   # Worker properties
   plugin.path=/etc/kafka-connect-worker/plugins
   ```

5. Create the configuration file `/etc/kafka-connect-worker/file-sink.properties` for the FileSink connector to copy data from a YDB topic to a file:

   ```
   name=local-file-sink
   connector.class=FileStreamSink
   tasks.max=1
   file=/etc/kafka-connect-worker/file_to_write.json
   topics=<topic-name>
   ```

   Where:

   - `file` - the name of the file where the connector will write data.
   - `topics` - the name of the topic from which the connector will read data.
6. Run Kafka Connect in standalone mode:

   ```
   cd ~/opt/kafka/bin/ && \
   sudo ./connect-standalone.sh \
           /etc/kafka-connect-worker/worker.properties \
           /etc/kafka-connect-worker/file-sink.properties
   ```

Connector configuration examples

This section provides sample Kafka Connect connector configuration files for working with YDB over the Kafka protocol.

From a file to a YDB topic

Example configuration file `/etc/kafka-connect-worker/file-sink.properties` for the FileSource connector to stream data from a file to a topic:

```
name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=/etc/kafka-connect-worker/file_to_read.json
topic=<topic-name>
```

From YDB to PostgreSQL

Example configuration file, `/etc/kafka-connect-worker/jdbc-sink.properties`, for the JDBC Sink connector to stream data from a topic to a PostgreSQL table. This example uses the Kafka Connect JDBC Connector.

```
name=postgresql-sink
connector.class=io.confluent.connect.jdbc.JdbcSinkConnector

connection.url=jdbc:postgresql://<postgresql-host>:<postgresql-port>/<db>
connection.user=<pg-user>
connection.password=<pg-user-pass>

topics=<topic-name>
batch.size=2000
auto.commit.interval.ms=1000

transforms=wrap
transforms.wrap.type=org.apache.kafka.connect.transforms.HoistField$Value
transforms.wrap.field=data

auto.create=true
insert.mode=insert
pk.mode=none
auto.evolve=true
```

From PostgreSQL to YDB

Example configuration file `/etc/kafka-connect-worker/jdbc-source.properties` for the JDBC Source connector to stream data from a PostgreSQL table to a topic. This example uses the Kafka Connect JDBC Connector.

```
name=postgresql-source
connector.class=io.confluent.connect.jdbc.JdbcSourceConnector

connection.url=jdbc:postgresql://<postgresql-host>:<postgresql-port>/<db>
connection.user=<pg-user>
connection.password=<pg-user-pass>

mode=bulk
query=SELECT * FROM "<topic-name>";
topic.prefix=<topic-name>
poll.interval.ms=1000
validate.non.null=false
```

From YDB to S3

Example configuration file `/etc/kafka-connect-worker/s3-sink.properties` for the S3 Sink connector to stream data from a topic to S3. This example uses Aiven's S3 Sink Connector for Apache Kafka.

```
name=s3-sink
connector.class=io.aiven.kafka.connect.s3.AivenKafkaConnectS3SinkConnector
topics=<topic-name>
aws.access.key.id=<s3-access-key>
aws.secret.access.key=<s3-secret>
aws.s3.bucket.name=<bucket-name>
aws.s3.endpoint=<s3-endpoint>
format.output.type=json
file.compression.type=none
```

Kafka API constraints

YDB supports Apache Kafka protocol version 3.4.0 with the following constraints:

1. Message compression is not supported.
2. The topic deletion operation is not supported. To delete a topic, use YQL or YDB CLI.
3. CRC checks are not supported.
4. Support for ACL is not provided. Use YQL to manage access to topics.
5. If auto-partitioning is enabled on a topic, you cannot write to or read from such a topic using the Kafka API.

Amazon SQS API

YDB supports working with topics via the Amazon SQS protocol.

> **Note**
>
> Working with topics via the Amazon SQS protocol is supported only for Amazon JSON protocol.

Work with a single topic can be done simultaneously using multiple protocols. For example, writing can be done using the Topic API, and reading using the Amazon SQS API, and vice versa.

When creating a topic using the `CreateQueue` Amazon SQS API command, the topic is created with auto-partitioning enabled: with one partition and the automatic ability to increase up to 10 active partitions. Auto-partitioning parameters can be changed via YQL or YDB CLI.

Reading via Amazon SQS API

For reading via Amazon SQS API, a shared (common) reader is used, which must be created on the topic before reading via the Amazon SQS protocol. If the topic is created using the `CreateQueue` Amazon SQS API command, a shared (common) reader named `ydb-sqs-consumer` is automatically created.

Messages written via the Topic API may be compressed using gzip, lzop, or zstd algorithms. When reading via the Amazon SQS protocol, the server does not decompress them but transmits them as base64: the reader must perform base64 decoding and then decompression.

Messages written via the Topic API without compression may contain binary data. When reading via the Amazon SQS protocol, the server also encodes them in base64 — the reader must perform base64 decoding.

To allow the reader to determine the compression algorithm, an attribute `BodyEncoding` is provided with each message. If the `BodyEncoding` attribute is missing, decompressing the message is not required. Possible attribute values: `gzip`, `lzop`, `zstd`, or `base64`.

Example of a message whose content is compressed:

```
{
    "Messages": [
        {
            "MessageId": "D523647F-5E89-560B-B1D0-152201831603",
            "ReceiptHandle": "CAAQAA==",
            "MD5OfBody": "d620a162c499920254054f78eac4feed",
            "Body": "KLUv/QBYaQAAeWRiIHdyaXRlZCAyCg==",
            "Attributes": {
                "SentTimestamp": "1780660726888",
                "BodyEncoding": "zstd"
            }
        }
    ]
}
```

Writing via Amazon SQS API

When writing to a topic via the Amazon SQS protocol, messages are evenly distributed across partitions. This guarantees that all messages with the same `MessageGroupId` end up in the same partition.

For writing via the Amazon SQS protocol, message deduplication is supported by `DeduplicationMessageId`, and if `DeduplicationMessageId` is not provided, by message content. Deduplication by content can be enabled using the `CreateQueue` and `SetQueueAttributes` commands by specifying the `ContentBasedDeduplication` parameter.

Deduplication by content is implemented with a 5-minute window: a message with a duplicate `DeduplicationMessageId` can be written again after 5 minutes or more.

There is a limit on the number of messages that can be written to a topic partition with deduplication by content enabled: 500 messages per second. If you need to write more messages to the topic, you should increase the number of partitions. The limit for a topic is calculated as 500 messages/sec/partition × number of partitions. For example, to write 10,000 messages per second, create a topic with 20 partitions.

Documentation sections

- Authentication
- Examples of working with topics via Amazon SQS API
- Creating a queue using YQL
- Shared topic consumer: architecture and limitations

Amazon SQS API

YDB supports working with topics via the Amazon SQS protocol.

> **Note**
>
> Working with topics via the Amazon SQS protocol is supported only for Amazon JSON protocol.

Work with a single topic can be done simultaneously using multiple protocols. For example, writing can be done using the Topic API, and reading using the Amazon SQS API, and vice versa.

When creating a topic using the `CreateQueue` Amazon SQS API command, the topic is created with auto-partitioning enabled: with one partition and the automatic ability to increase up to 10 active partitions. Auto-partitioning parameters can be changed via YQL or YDB CLI.

Reading via Amazon SQS API

For reading via Amazon SQS API, a shared (common) reader is used, which must be created on the topic before reading via the Amazon SQS protocol. If the topic is created using the `CreateQueue` Amazon SQS API command, a shared (common) reader named `ydb-sqs-consumer` is automatically created.

Messages written via the Topic API may be compressed using gzip, lzop, or zstd algorithms. When reading via the Amazon SQS protocol, the server does not decompress them but transmits them as base64: the reader must perform base64 decoding and then decompression.

Messages written via the Topic API without compression may contain binary data. When reading via the Amazon SQS protocol, the server also encodes them in base64 — the reader must perform base64 decoding.

To allow the reader to determine the compression algorithm, an attribute `BodyEncoding` is provided with each message. If the `BodyEncoding` attribute is missing, decompressing the message is not required. Possible attribute values: `gzip`, `lzop`, `zstd`, or `base64`.

Example of a message whose content is compressed:

```
{
    "Messages": [
        {
            "MessageId": "D523647F-5E89-560B-B1D0-152201831603",
            "ReceiptHandle": "CAAQAA==",
            "MD5OfBody": "d620a162c499920254054f78eac4feed",
            "Body": "KLUv/QBYaQAAeWRiIHdyaXRlZCAyCg==",
            "Attributes": {
                "SentTimestamp": "1780660726888",
                "BodyEncoding": "zstd"
            }
        }
    ]
}
```

Writing via Amazon SQS API

When writing to a topic via the Amazon SQS protocol, messages are evenly distributed across partitions. This guarantees that all messages with the same `MessageGroupId` end up in the same partition.

For writing via the Amazon SQS protocol, message deduplication is supported by `DeduplicationMessageId`, and if `DeduplicationMessageId` is not provided, by message content. Deduplication by content can be enabled using the `CreateQueue` and `SetQueueAttributes` commands by specifying the `ContentBasedDeduplication` parameter.

Deduplication by content is implemented with a 5-minute window: a message with a duplicate `DeduplicationMessageId` can be written again after 5 minutes or more.

There is a limit on the number of messages that can be written to a topic partition with deduplication by content enabled: 500 messages per second. If you need to write more messages to the topic, you should increase the number of partitions. The limit for a topic is calculated as 500 messages/sec/partition × number of partitions. For example, to write 10,000 messages per second, create a topic with 20 partitions.

Documentation sections

- Authentication
- Examples of working with topics via Amazon SQS API
- Creating a queue using YQL
- Shared topic consumer: architecture and limitations

Authentication

In the Open Source version of YDB, working with topics via the Amazon SQS protocol is supported only without authentication.

Cluster setup

To disable authentication, in the `security_config` section of the cluster configuration file, set the `enforce_user_token_requirement` parameter to `false`:

```
security_config:
  enforce_user_token_requirement: false
```

Examples of working with topics via Amazon SQS API

This article provides examples of working with topics using the Amazon SQS API via AWS CLI.

> **Note**
>
> Working with topics via the Amazon SQS protocol is supported only for Amazon JSON protocol.

Before running the examples, configure access to the Amazon SQS API (in the Open Source version, disable the authentication requirement — see the section on authentication).

If you use AWS CLI, make sure it does not require configured AWS credentials: either add `--no-sign-request` to the commands, or set any values for `AWS_ACCESS_KEY_ID` / `AWS_SECRET_ACCESS_KEY`.

Forming the endpoint for connection

The endpoint for accessing the Amazon SQS API is formed as follows:

```
https://{db-balancer}:{port}/{database}
```

Where:

- `db-balancer` — DNS name of the HTTPS balancer whose backend includes the database compute nodes (or the address of the node/service where the HTTP Proxy is running).
- `port` — port on which the HTTP Proxy is available.
- `database` — full path of the database where the topics are located.

This endpoint is specified in AWS CLI via the `--endpoint` parameter.

> **Note**
>
> The examples use the endpoint `https://my_db.balancer.example.com:8443/Root/my_db`. It includes:
>
> - `my_db.balancer.example.com` — DNS name of the balancer through which the Amazon SQS protocol is available.
> - `8443` — network port.
> - `/Root/my_db` — database name.

Creating a topic

To create a topic, run the command:

```
ENDPOINT="https://my_db.balancer.example.com:8443/Root/my_db"

aws --endpoint "$ENDPOINT" \
  sqs create-queue --queue-name "my_topic"
```

After executing the command, a topic with the specified name and a shared (common) reader named `ydb-sqs-consumer` will be created. You can verify the topic's existence using the scheme describe command of the YDB CLI:

```
ydb -e grpcs://my_db.balancer.example.com:2135 -d /Root/my_db scheme describe my_topic
```

To create a FIFO queue, use the `FifoQueue=true` attribute. For FIFO queues, it is recommended to end the name with `.fifo` to comply with the Amazon SQS naming convention:

```
ENDPOINT="https://my_db.balancer.example.com:8443/Root/my_db"

aws --endpoint "$ENDPOINT" \
  sqs create-queue \
  --queue-name "my_topic.fifo" \
  --attributes FifoQueue=true
```

After executing the command, a topic named `my_topic.fifo` and a shared (common) reader named `ydb-sqs-consumer` will be created, with message ordering preservation enabled.

Getting a list of topics

To get a list of topics that can be worked with via the Amazon SQS protocol, run the command:

```
ENDPOINT="https://my_db.balancer.example.com:8443/Root/my_db"

aws --endpoint "$ENDPOINT" sqs list-queues
```

Getting the QueueUrl of a topic

To get the `QueueUrl` of a topic, run the command:

```
ENDPOINT="https://my_db.balancer.example.com:8443/Root/my_db"

aws --endpoint "$ENDPOINT" sqs get-queue-url \
  --queue-name "my_topic@my_consumer"
```

Where:

- `{queue_name}` — value of the `--queue-name` parameter, the Amazon SQS queue name in the format `{topic_name}@{consumer_name}`. If the shared (common) reader has the name `ydb-sqs-consumer`, it is sufficient to specify only `{topic_name}`.
- `{topic_name}` — topic name with the path from the root of the database where it was created, for example `production/order` (topic `order` in directory `production`).
- `{consumer_name}` — name of the shared (common) reader on the topic, for example `ydb-sqs-consumer`.

In the example above, `my_topic` is `{topic_name}`, and `my_consumer` is `{consumer_name}`.

Writing to a topic and reading from a topic

For read and write operations, AWS CLI uses the `--queue-url` parameter. It can be obtained via `get-queue-url`.

Below is an example:

- getting `QueueUrl`.
- writing a message.
- reading a message.

```
ENDPOINT="https://my_db.balancer.example.com:8443/Root/my_db"

# get QueueUrl
QUEUE_URL="$(aws --endpoint "$ENDPOINT" sqs get-queue-url --queue-name "my_topic" --query 'QueueUrl' --output
text)"

# write a message to a topic
aws --endpoint "$ENDPOINT" sqs send-message \
  --queue-url "$QUEUE_URL" \
  --message-body "hello from aws cli"

# read a message (long polling)
aws --endpoint "$ENDPOINT" sqs receive-message \
  --queue-url "$QUEUE_URL" \
  --wait-time-seconds 20 \
  --max-number-of-messages 1
```

YDB Cluster Configuration

The cluster configuration is specified in the YAML file passed in the `--yaml-config` parameter when the cluster nodes are run. This article provides an overview of the main configuration sections and links to detailed documentation for each section.

Each configuration section serves a specific purpose in defining how the YDB cluster operates, from hardware resource allocation to security settings and feature flags. The configuration is organized into logical groups that correspond to different aspects of cluster management and operation.

Configuration Sections

The following top-level configuration sections are available, listed in alphabetical order:

Section	Required	Description
actor_system_config	Yes	CPU resource allocation across actor system pools
auth_config	No	Authentication and authorization settings
aws_client_config	No	AWS client default settings
blob_storage_config	No	Static cluster group configuration for system tablets
bridge_config	No	Cluster piles for bridge mode
client_certificate_authorization	No	Client certificate authentication
cms_config	No	Cluster Management System configuration
domains_config	No	Cluster domain configuration including Blob Storage and State Storage
feature_flags	No	Feature flags to enable or disable specific YDB features
healthcheck_config	No	Health check service thresholds and timeout settings
hive_config	No	Hive component configuration for tablet management
host_configs	No	Typical host configurations for cluster nodes
hosts	Yes	Static cluster nodes configuration
kafka_proxy_config	No	Kafka Proxy configuration
log_config	No	Logging configuration and parameters
memory_controller_config	No	Memory allocation and limits for database components
node_broker_config	No	Stable node names configuration
query_service_config	No	Federated query connector configuration
resource_broker_config	No	Resource broker for controlling CPU and memory consumption
security_config	No	Security configuration settings
table_service_config	No	Query processing configuration
tli_config	No	Transaction lock invalidation (TLI) diagnostics parameters
tls	No	TLS configuration for secure connections

Practical Guidelines

While this documentation section focuses on complete reference documentation for available settings, practical recommendations on what to tune and when can be found in the following places:

- As part of the initial YDB cluster deployment:
  - Ansible
  - Kubernetes
  - Manual
- As part of troubleshooting
- As part of security hardening

Sample Cluster Configurations

You can find model cluster configurations for deployment in the repository. Check them out before deploying a cluster.

actor_system_config

The CPU resources are mainly used by the actor system. Depending on the type, all actors run in one of the pools (the `name` parameter). Configuration involves allocating a node's CPU cores across the actor system pools. When allocating them, please keep in mind that PDisks and the gRPC API run outside the actor system and require separate resources.

You can set up your actor system either automatically or manually. In the `actor_system_config` section, specify:

- Node type and the number of CPU cores allocated to the ydbd process by automatic configuration.
- Number of CPU cores for each YDB cluster subsystem in the case of manual configuration.

Automatic configuration adapts to the current system workload. It is recommended in most cases.

You might opt for manual configuration when a certain pool in your actor system is overwhelmed and undermines the overall database performance. You can track the workload on your pools on the YDB UI monitoring page.

Automatic Configuring

Regardless of the `node_type` parameter value, if one or more pools are starving (experiencing a shortage of compute resources), core redistribution follows the priority order below (highest to lowest):

- `IC`
- `System`
- `User`
- `Batch`

For example, if `IC` and `User` are starving while `System` has idle cores, `IC` receives them.

Example of the `actor_system_config` section for automatic configuration of the actor system:

```
actor_system_config:
  use_auto_config: true
  node_type: STORAGE
  cpu_count: 10
```

Parameter	Description
`use_auto_config`	Enabling automatic configuration of the actor system.
`node_type`	Node type. Determines the expected workload and vCPU ratio between the pools. Possible values: • `STORAGE`: The node interacts with network block store volumes and is responsible for managing the Distributed Storage. • `COMPUTE`: The node processes the workload generated by users. • `HYBRID`: The node is used for hybrid load or the usage of `System`, `User`, and `IC` for the node under load is about the same.
`cpu_count`	Number of vCPUs allocated to the node.

Manual Configuring

Example of the `actor_system_config` section for manual configuration of the actor system:

```
actor_system_config:
  executor:
  - name: System
    spin_threshold: 0
    threads: 2
    type: BASIC
  - name: User
    spin_threshold: 0
    threads: 3
    type: BASIC
  - name: Batch
    spin_threshold: 0
    threads: 2
    type: BASIC
  - name: IO
    threads: 1
    time_per_mailbox_micro_secs: 100
    type: IO
  - name: IC
    spin_threshold: 10
    threads: 1
    time_per_mailbox_micro_secs: 100
    type: BASIC
  scheduler:
    progress_threshold: 10000
    resolution: 256
    spin_threshold: 0
```

Parameter	Description
`executor`	Pool configuration. You should only change the number of CPU cores (the `threads` parameter) in the pool configs.

`name`	Pool name that indicates its purpose. Possible values: • `System` : A pool that is designed for running quick internal operations in YDB (it serves system tablets, state storage, distributed storage I/O, and erasure coding). • `User` : A pool that serves the user load (user tablets, queries run in the Query Processor). • `Batch` : A pool that serves tasks with no strict limit on the execution time, background operations like garbage collection and heavy queries run in the Query Processor. • `IO` : A pool responsible for performing any tasks with blocking operations (such as authentication or writing logs to a file). • `IC` : Interconnect, it serves the load related to internode communication (system calls to wait for sending and send data across the network, data serialization, as well as message splits and merges).
`spin_threshold`	The number of CPU cycles before going to sleep if there are no messages. In sleep mode, there is less power consumption, but it may increase request latency under low loads.
`threads`	The number of CPU cores allocated per pool. Make sure the total number of cores assigned to the System, User, Batch, and IC pools does not exceed the number of available system cores.
`max_threads`	Maximum vCPU that can be allocated to the pool from idle cores of other pools. When you set this parameter, the system enables the mechanism of expanding the pool at full utilization, provided that idle vCPUs are available. The system checks the current utilization and reallocates vCPUs once per second.
`max_avg_ping_deviation`	Additional condition to expand the pool's vCPU. When more than 90% of vCPUs allocated to the pool are utilized, you need to worsen SelfPing by more than `max_avg_ping_deviation` microseconds from 10 milliseconds expected.
`time_per_mailbox_micro_secs`	The number of messages per actor to be handled before switching to a different actor.
`type`	Pool type. Possible values: • `IO` should be set for IO pools. • `BASIC` should be set for any other pool.
`scheduler`	Scheduler configuration. The actor system scheduler is responsible for the delivery of deferred messages exchanged by actors. We do not recommend changing the default scheduler parameters.
`progress_threshold`	The actor system supports requesting message sending scheduled for a later point in time. The system might fail to send all scheduled messages at some point. In this case, it starts sending them in "virtual time" by handling message sending in each loop over a period that doesn't exceed the `progress_threshold` value in microseconds and shifting the virtual time by the `progress_threshold` value until it reaches real time.
`resolution`	When making a schedule for sending messages, discrete time slots are used. The slot duration is set by the `resolution` parameter in microseconds.

auth_config

YDB supports various user authentication methods. The configuration for authentication providers is specified in the `auth_config` section.

Configuring Local YDB User Authentication

For more information about the authentication of local YDB users, see Authenticating by username and password. To configure authentication by username and password, define the following parameters in the `auth_config` section:

Parameter	Description
use_login_provider	Indicates whether to allow the authentication of local users with an authentication token that is obtained after entering a username and password. Default value: `true`
enable_login_authentication	Indicates whether to allow adding local users to YDB databases and generating authentication tokens after a local user enters a username and password. Default value: `true`
domain_login_only	Determines the scope of local user access rights in a YDB cluster. Valid values: • `true` — local users exist in a YDB cluster and can be granted rights to access multiple databases. • `false` — local users can exist either at the cluster or database level. The scope of access rights for local users created at the database level is limited to the database, in which they are created. Default value: `true`
login_token_expire_time	Specifies the expiration time of the authentication token created when a local user logs in to YDB. Default value: `12h`

Configuring User Lockout

You can configure YDB to lock a user account out after a specified number of failed attempts to enter the correct password. To configure user lockout, define the `account_lockout` subsection inside the `auth_config` section.

Example of the `account_lockout` section:

```
auth_config:
  #...
  account_lockout:
    attempt_threshold: 4
    attempt_reset_duration: "1h"
  #...
```

Parameter	Description
attempt_threshold	Specifies the number of failed attempts to enter the correct password for a user account, after which the account is blocked for a period specified by the `attempt_reset_duration` parameter. If `attempt_threshold = 0`, the number of attempts to enter the correct password is unlimited. After successful authentication (correct username and password), the counter for failed attempts is reset to 0. Default value: `4`
attempt_reset_duration	Specifies the period that a locked-out account remains locked before automatically becoming unlocked. This period starts after the last failed attempt. During this period, the user will not be able to authenticate in the system even if the correct username and password are entered. If this parameter is set to zero ("0s" - a notation equivalent of 0 seconds), user accounts will be locked indefinitely. In this case you can unlock the account using the ALTER USER ... LOGIN command. The minimum lockout duration is 1 second. Supported time units: • Seconds: `30s` • Minutes: `20m` • Hours: `5h` • Days: `3d` It is not allowed to combine time units in one entry. For example, the entry `1d12h` is incorrect. It should be replaced with an equivalent, such as `36h`. Default value: `1h`

Configuring Password Complexity Requirements

YDB allows local users to authenticate using a login and password. For more information, see authentication by login and password. To enhance security in YDB, configure complexity requirements for the passwords of local users in the `password_complexity` subsection inside the `auth_config` section.

Example of the `password_complexity` section:

```
auth_config:
  #...
  password_complexity:
    min_length: 8
    min_lower_case_count: 1
    min_upper_case_count: 1
    min_numbers_count: 1
    min_special_chars_count: 1
    special_chars: "!@#$%^&*()_+{}|<>?="
    can_contain_username: false
  #...
```

Parameter	Description
min_length	Specifies the minimum password length. Default value: `0` (no requirements)
min_lower_case_count	Specifies the minimum number of lowercase letters that a password must contain. Default value: `0` (no requirements)
min_upper_case_count	Specifies the minimum number of uppercase letters that a password must contain. Default value: `0` (no requirements)
min_numbers_count	Specifies the minimum number of digits that a password must contain. Default value: `0` (no requirements)
min_special_chars_count	Specifies the minimum number of special characters from the `special_chars` list that a password must contain. Default value: `0` (no requirements)
special_chars	Specifies a list of special characters that are allowed in a password. Valid values: `!@#$%^&*()_+{}\|<>?=` Default value: empty (any of the `!@#$%^&*()_+{}|<>?=` characters are allowed)
can_contain_username	Indicates whether passwords can include a username. Default value: `false`

Note

Any changes to the password policy do not affect existing user passwords, so it is not necessary to change current passwords; they will be accepted as they are.

Configuring LDAP Authentication

One of the user authentication methods in YDB is using an LDAP directory. For more details, see Interacting with the LDAP directory. To configure LDAP authentication, define the `ldap_authentication` section inside the `auth_config` section.

Example of the `ldap_authentication` section:

```
auth_config:
  #...
  ldap_authentication:
    hosts:
      - "ldap-hostname-01.example.net"
      - "ldap-hostname-02.example.net"
      - "ldap-hostname-03.example.net"
    port: 389
    base_dn: "dc=mycompany,dc=net"
    bind_dn: "cn=serviceAccaunt,dc=mycompany,dc=net"
    bind_password: "serviceAccauntPassword"
    search_filter: "uid=$username"
    scheme: "ldap"
    requested_group_attribute: "memberOf"
    extended_settings:
      enable_nested_groups_search: true
    use_tls:
      enable: true
      ca_cert_file: "/path/to/ca.pem"
      cert_require: DEMAND
  ldap_authentication_domain: "ldap"
  refresh_time: "1h"
  #...
```

Parameter	Description
`hosts`	Specifies a list of hostnames where the LDAP server is running.
`port`	Specifies the port used to connect to the LDAP server.

`base_dn`	Specifies the root of the subtree in the LDAP directory from which the user entry search begins.
`bind_dn`	Specifies the Distinguished Name (DN) of the service account used to search for the user entry.
`bind_password`	Specifies the password for the service account used to search for the user entry.
`search_filter`	Specifies a filter for searching the user entry in the LDAP directory. The filter string can include the sequence *$username*, which is replaced with the username requested for authentication in the database.
`use_tls`	Configuration settings for the TLS connection between YDB and the LDAP server.
`enable`	Indicates whether a TLS connection using the `StartTls` request will be attempted. When set to `true`, the `ldaps` connection scheme should be disabled by setting `ldap_authentication.scheme` to `ldap`.
`ca_cert_file`	Specifies the path to the certification authority's certificate file.
`cert_require`	Specifies the certificate requirement level for the LDAP server. Possible values: • `NEVER` - YDB does not request a certificate or accepts any presented certificate. • `ALLOW` - YDB requests a certificate from the LDAP server but will establish the TLS session even if the certificate is not trusted. • `TRY` - YDB requires a certificate from the LDAP server and terminates the connection if it is not trusted. • `DEMAND` / `HARD` - These are equivalent to `TRY` and are the default setting, with the value set to `DEMAND`.
`ldap_authentication_domain`	Specifies an identifier appended to the username to distinguish LDAP directory users from those authenticated using other providers. Default value: `ldap`
`scheme`	Specifies the connection scheme to the LDAP server. Possible values: • `ldap` - Connects without encryption, sending passwords in plain text. • `ldaps` - Connects using TLS encryption from the first request. To use `ldaps`, disable the `StartTls` request by setting `ldap_authentication.use_tls.enable` to `false`, and provide certificate details in `ldap_authentication.use_tls.ca_cert_file` and set the certificate requirement level in `ldap_authentication.use_tls.cert_require`. • Any other value defaults to `ldap`. Default value: `ldap`
`requested_group_attribute`	Specifies the attribute used for reverse group membership. The default is `memberOf`.
`extended_settings.enable_nested_groups_search`	Indicates whether to perform a request to retrieve the full hierarchy of groups to which the user's direct groups belong. Possible values: • `true` — YDB requests information about all groups to which the user's direct groups belong. It might take a long time to traverse the entire hierarchy of nested parent groups. • `false` — YDB requests a flat list of groups, to which the user belongs. This request does not traverse possible nested parent groups. Default value: `false`
`host`	Specifies the hostname of the LDAP server. This parameter is deprecated and should be replaced with the `hosts` parameter.

Configuring Third-Party IAM Authentication

YDB supports Yandex Identity and Access Management (IAM) used in Yandex Cloud for user authentication. To configure IAM authentication, define the following parameters:

Parameter	Description
use_access_service	Indicates whether to allow authentication in Yandex Cloud using IAM AccessService. Default value: `false`
access_service_endpoint	Specifies an IAM AccessService address, to which YDB sends requests. Default value: `as.private-api.cloud.yandex.net:4286`
use_access_service_tls	Indicates whether to use TLS connections between YDB and AccessService. Default value: `true`

access_service_domain	Specifies an identifier appended to the username in SID to distinguish Yandex Cloud IAM users from those authenticated using other providers. Default value: `as` ("access service")
path_to_root_ca	Specifies the path to the certification authority's certificate file that is used to interact with AccessService. Default value: `/etc/ssl/certs/YandexInternalRootCA.pem`
access_service_grpc_keep_alive_time_ms	Specifies the period of time, in milliseconds, after which a keepalive ping is sent on the transport to IAM AccessService. Default value: `10000`
access_service_grpc_keep_alive_timeout_ms	Specifies the amount of time, in milliseconds, that YDB waits for the acknowledgement of the keepalive ping from IAM AccessService. If YDB does not receive an acknowledgment within this time, it will close the connection. Default value: `1000`
use_access_service_api_key	Indicates whether to use IAM API keys. The API key is a secret key created in Yandex Cloud IAM for simplified authorization of service accounts with the Yandex Cloud API. Use API keys if requesting an IAM token automatically is not an option. Default value: `false`

Configuring Caching for Authentication Results

During the authentication process, a user session receives an authentication token, which is transmitted along with each request to the cluster YDB. Since YDB is a distributed system, user requests will eventually be processed on one or more YDB nodes. After receiving a request from the user, a YDB node verifies the authentication token. If successful, the node generates a **user token**, which is valid only inside the current node and is used to authorize the actions requested by the user. Subsequent requests with the same authentication token to the same node do not require verification of the authentication token.

To configure the life cycle and other important aspects of managing user tokens, define the following parameters:

refresh_period	Specifies how often a YDB node scans cached user tokens to find the ones that need to be refreshed because the `refresh_time`, `life_time` or `expire_time` interval elapses. The lower this parameter value, the higher the CPU load. Default value: `1s`
refresh_time	Specifies the time interval since the last user token update after which a YDB node updates the user token again. The actual update will occur within the range from `refresh_time/2` to `refresh_time`. Default value: `1h`
life_time	Specifies the time interval for keeping a user token in YDB node cache since its last use. If a YDB node does not receive queries from a user within the specified time interval, the node deletes the user token from its cache. Default value: `1h`
expire_time	Specifies the time period, after which a user token is deleted from YDB node cache. Deletion occurs regardless of the `life_time` interval. **Warning** If a third-party system has successfully authenticated in the YDB node and regularly (more often than the `life_time` interval) sends requests to the same node, YDB will detect the possible deletion or change in the user account privileges only after the `expire_time` interval elapses. The shorter this time period, the more often YDB nodes re-authenticate users and refresh their privileges. However, excessive user re-authentication slows down YDB, especially so for external users. Setting this parameter to seconds negates the effect of caching user tokens. Default value: `24h`
min_error_refresh_time	Specifies minimum period of time that must elapse since a failed attempt (temporary failure) to refresh a user token before retrying the attempt. Together with the `max_error_refresh_time`, determines the possible interval for a delay before retrying a failed attempt to refresh a user token. Each subsequent delay is increased till it reaches the `max_error_refresh_time` value. Retries continue until a user token is refreshed or the `expire_time` period elapses. **Warning** Setting this parameter to `0` is not recommended, because instant retries results in excessive load. Default value: `1s`
max_error_refresh_time	Specifies the maximum time interval that can elapse since a failed attempt (temporary failure) to refresh a user token before retrying the attempt. Together with the `min_error_refresh_time`, determines the possible interval for a delay before retrying a failed attempt to refresh a user token. Each subsequent delay is increased till it reaches the `max_error_refresh_time` value. Retries continue until a user token is refreshed or the `expire_time` period elapses. Default value: `1m`

Node registration token configuration

YDB allows you to configure the authentication method for database nodes when they register with a cluster. This is done via the `node_registration_token` parameter in the `auth_config` section.

Parameter	Description
node_registration_token	Specifies the authentication method for database nodes registering with a cluster. Possible values: • Empty string (`""`) — nodes authenticate using TLS certificates when registering with the cluster. In this mode, nodes must use certificates for authentication during the registration process. For details on configuring node authentication with certificates, see Database node authentication and authorization. • `"root@builtin"` — nodes authenticate using a special debug token when registering with the cluster. This authentication method is deprecated and will be removed in future releases. To ensure cluster security, always prefer TLS certificate-based authentication by setting this parameter to an empty string.

Example of an `auth_config` section with TLS certificate authentication enabled:

```
auth_config:
  ...
  node_registration_token: ""
  ...
```

aws_client_config

The `aws_client_config` section defines defaults for the AWS SDK client that YDB uses when accessing S3 and other object storage systems with a compatible API. It configures timeouts, TLS settings, and logging for all such outbound connections made by the cluster node process; a specific storage configuration can override these values when needed.

Syntax

```
aws_client_config:
  log_config:
    log_level: 0
    filename_prefix: ""
  verify_ssl: true
  request_timeout_ms: 0
  http_request_timeout_ms: 0
  connection_timeout_ms: 10000
  enable_tcp_keep_alive: true
  tcp_keep_alive_interval_ms: 30000
  max_connections_count: 32
  executor_threads_count: 32
  ca_path: "/etc/ssl/certs"
```

Parameters

Parameter	Default	Description
`log_config.log_level`	`0` (Off)	Numeric AWS SDK log verbosity level. See Log levels for level names and numeric values.
`log_config.filename_prefix`	—	Prefix for AWS SDK log files. When set, files are created as `<prefix>YYYY-MM-DD-HH.log`; otherwise logs go to stdout.
`verify_ssl`	`true`	Whether to validate TLS certificates when connecting to AWS endpoints and compatible APIs. Set to `false` only in test environments.
`request_timeout_ms`	`0` (disabled)	Request timeout at the AWS SDK level for the entire request, including DNS lookup, connection setup, TLS handshake, and data transfer. The value is specified in milliseconds. `0` means no SDK-side timeout limit.
`http_request_timeout_ms`	`0` (disabled)	HTTP-layer timeout in milliseconds applied by the HTTP client (Curl). `0` means no timeout.
`connection_timeout_ms`	`10000`	TCP connect timeout in milliseconds.
`enable_tcp_keep_alive`	`true`	Enables TCP keep-alive on client sockets.
`tcp_keep_alive_interval_ms`	`30000` (AWS SDK default)	Interval between TCP keep-alive probes in milliseconds. When not set in the YDB config, the AWS SDK default of 30000 ms is used. In Curl this value is rounded to whole seconds and must be at least 15000.
`max_connections_count`	`32`	Maximum number of concurrent TCP connections that the HTTP client inside AWS SDK may open.
`executor_threads_count`	`32`	Size of the thread pool allocated per unique S3 endpoint. Controls parallelism for asynchronous requests to that endpoint.
`ca_path`	`"/etc/ssl/certs"`	Path to a directory with CA certificates used to verify TLS for outbound connections.
`ca_file`	—	Path to a single CA certificate bundle file. Used when the trust store is not in the default location.

Log levels

Level	Value	Description
`Off`	`0`	Disable AWS SDK logging.
`Fatal`	`1`	Fatal errors only.
`Error`	`2`	Non-fatal errors.
`Warn`	`3`	Warnings that may need attention.
`Info`	`4`	High-level informational messages.
`Debug`	`5`	Debug information.
`Trace`	`6`	Most verbose diagnostic output.

See also

- log_config for logging of the YDB server process. The `log_config` block nested under `aws_client_config` applies only to AWS SDK and is unrelated to the server logging configuration.
- Data export and import for S3 import and export from the CLI. Storage access is configured there, not in `aws_client_config`.

Related scenarios ("S3 in YDB")

Other object storage tasks. Settings from these materials may override the values configured in the `aws_client_config` section.

- CREATE EXTERNAL DATA SOURCE for an external data source with the S3 protocol in YQL.
- Deploying YDB with Federated Query functionality for federated queries and external data sources.
- Import and export data using federated queries for import and export via federated queries to object storage.

blob_storage_config

The `blob_storage_config` section specifies a static cluster group's configuration. A static group is necessary for the operation of the basic cluster tablets, including `Hive`, `SchemeShard`, and `BlobstorageController`. As a rule, these tablets do not store a lot of data, so we don't recommend creating more than one static group.

For a static group, specify the disks and nodes that the static group will be placed on. For example, a configuration for the `erasure: none` model can be as follows:

```
blob_storage_config:
  service_set:
    groups:
    - erasure_species: none
      rings:
      - fail_domains:
        - vdisk_locations:
          - node_id: 1
            path: /dev/disk/by-partlabel/ydb_disk_ssd_02
            pdisk_category: SSD
....
```

For a configuration located in 3 availability zones, specify 3 rings. For a configuration within a single availability zone, specify exactly one ring.

bridge_config

This section describes cluster piles for bridge mode. List the pile names used to bind hosts and other entities. In bridge mode, each host must also specify the corresponding pile name in the `hosts` section (`bridge_pile_name` field); see Bridge mode specifics.

Syntax

```
bridge_config:
  piles:
  - name: <pile_name_1>
  - name: <pile_name_2>
  ...
  - name: <pile_name_n>
```

client_certificate_authorization

The `client_certificate_authorization` section configures authentication of database nodes within the YDB cluster using client certificates. This ensures that service connections between cluster nodes are assigned the correct security identifiers, or SIDs. The process applies to connections that use the gRPC protocol for registering nodes in the cluster and accessing configuration information.

Node authentication settings are configured within the static configuration of the cluster.

The `client_certificate_authorization` section specifies the authentication settings for database node connections by defining the requirements for the content of the "Subject" and "Subject Alternative Name" fields in node certificates, as well as the list of SID values assigned to the connections.

The "Subject" field of the node certificate may contain multiple components (such as `O` – organization, `OU` – organizational unit, `C` – country, `CN` – common name), and checks can be configured against one or more of these components.

The "Subject Alternative Name" field of the node certificate is a list of the node's network names or IP addresses. Checks can be configured to match the names specified in the certificate against the expected values.

Syntax

```
client_certificate_authorization:
  request_client_certificate: Bool
  default_group: <default SID>
  client_certificate_definitions:
    - member_groups: <SID array>
      require_same_issuer: Bool
      subject_dns:
        suffixes: <array of allowed suffixes>
        values: <array of allowed values>
      subject_terms:
      - short_name: <Subject Name component>
        suffixes: <array of allowed suffixes>
        values: <array of allowed values>
    - member_groups: <SID array>
    ...
```

Key	Description
`request_client_certificate`	Request a valid client certificate for node connections. Allowed values: • `false` — A certificate is not required (used by default if the parameter is omitted). • `true` — A certificate is required for all node connections.
`default_group`	SID assigned to all connections providing a trusted client certificate when no explicit settings are provided in the `client_certificate_definitions` section.
`client_certificate_definitions`	Section defining the requirements for database node certificates.
`member_groups`	SIDs assigned to connections that conform to the requirements of the current configuration block.
`require_same_issuer`	Require that the value of the "Issuer" field (typically containing the Certification Authority name) is the same for both client (database node) and server (storage node) certificates. Allowed values: • `true` — The values must be the same (used by default if the parameter is omitted). • `false` — The values can be different (allowing client and server certificates to be issued by different Certification Authorities).
`subject_dns`	Allowed values for the "Subject Alternative Name" field, specified as either full values (using the `values` sub-key) or suffixes (using the `suffixes` sub-key). The check is successful if the actual value matches any full name or any suffix specified.
`subject_terms`	Requirements for the "Subject" field value. Contains the component name (in the `short_name` sub-key) and a list of full values (using the `values` sub-key) or suffixes (using the `suffixes` sub-key). The check is successful if the actual value of each component matches either an allowed full value or an allowed suffix.

Examples

The following configuration fragment enables node authentication and requires the "Subject" field to include the component `O=YDB`. Upon successful authentication, the connection is assigned the `registerNode@cert` SID.

```
client_certificate_authorization:
  request_client_certificate: true
  client_certificate_definitions:
    - member_groups: ["registerNode@cert"]
      subject_terms:
      - short_name: "O"
        values: ["YDB"]
```

The next configuration fragment enables node authentication, and requires "Subject" field to include both `OU=cluster1` and `O=YDB` components. In addition "Subject Alternative Name" field should contain the network name ending with the `.cluster1.ydb.company.net` suffix. Upon successful authentication, the connection will be assigned the `registerNode@cert` SID.

```
client_certificate_authorization:
  request_client_certificate: true
  client_certificate_definitions:
```

```
    - member_groups: ["registerNode@cert"]
      subject_dns:
        suffixes: [".cluster1.ydb.company.net"]
      subject_terms:
      - short_name: "OU"
        values: ["cluster1"]
      - short_name: "O"
        values: ["YDB"]
```

cms_config

Cluster Management System (CMS) is a YDB component that enables safe cluster maintenance, for example, updating its version or replacing failed disks without loss of availability. CMS behavior is configured in the `cms_config` section of the YDB configuration.

Syntax

```
cms_config:
  tenant_limits:
    disabled_nodes_limit: 2
    disabled_nodes_ratio_limit: 5
  cluster_limits:
    disabled_nodes_limit: 3
    disabled_nodes_ratio_limit: 5
  disable_maintenance: true
```

Parameters

Parameter	Default value	Description
`tenant_limits.disabled_nodes_limit`	—	Maximum number of database nodes that can be simultaneously unavailable or blocked.
`tenant_limits.disabled_nodes_ratio_limit`	`13`	Maximum percentage of database nodes that can be simultaneously unavailable or blocked.
`cluster_limits.disabled_nodes_limit`	—	Maximum number of cluster nodes that can be simultaneously unavailable or blocked.
`cluster_limits.disabled_nodes_ratio_limit`	`13`	Maximum percentage of cluster nodes that can be simultaneously unavailable.
`disable_maintenance`	`false`	Flag that suspends new cluster maintenance operations.

domains_config

This section contains the configuration of the YDB cluster root domain, including the Blob Storage (binary object storage) and State Storage configurations.

> **Note**
>
> Not required for clusters with automatic configuration of State Storage and static groups.

Syntax

```
domains_config:
  domain:
  - name: <root domain name>
    storage_pool_types: <Blob Storage configuration>
  state_storage: <State Storage configuration>
  security_config: <authentication configuration>
```

> **Note**
>
> Formally, the `domain` field can contain many elements as it is a list. However, only the first element is meaningful; the rest will be ignored (there can only be one "domain" in a cluster).

Blob Storage Configuration

> **Note**
>
> Not required for clusters with automatic configuration of State Storage and static groups.

This section defines one or more types of storage pools available in the cluster for database data with the following configuration options:

- Storage pool name
- Device properties (for example, disk type)
- Data encryption (on/off)
- Fault tolerance mode

The following fault tolerance modes are available:

Mode	Description
`none`	There is no redundancy. Applies for testing.
`block-4-2`	Redundancy factor of 1.5, applies to single data center clusters.
`mirror-3-dc`	Redundancy factor of 3, applies to multi-data center clusters.

Syntax

```
  storage_pool_types:
  - kind: <storage pool name>
    pool_config:
      box_id: 1
      encryption_mode: <optional, specify 1 to encrypt data on the disk>
      erasure_species: <fault tolerance mode name - none, block-4-2, or mirror-3-dc>
      kind: <storage pool name - specify the same value as above>
      pdisk_filter:
      - property:
        - type: <device type to be compared with the one specified in host_configs.drive.type>
      vdisk_kind: Default
  - kind: <storage pool name>
  ...
```

Each database in the cluster is assigned at least one of the available storage pools selected in the database creation operation. The names of storage pools among those assigned can be used in the `DATA` attribute when defining column groups in YQL operators `CREATE TABLE` / `ALTER TABLE`.

State Storage Configuration

> **Note**
>
> Not required for clusters with automatic configuration of State Storage and static groups.

State Storage is an independent in-memory storage for variable data that supports internal YDB processes. It stores data replicas on multiple assigned nodes.

State Storage usually does not need scaling for better performance, so the number of nodes in it must be kept as small as possible taking into account the required level of fault tolerance.

State Storage availability is key for a YDB cluster because it affects all databases, regardless of which storage pools they use. To ensure fault tolerance of State Storage, its nodes must be selected to guarantee a working majority in case of expected failures.

The following guidelines can be used to select State Storage nodes:

Cluster type	Min number of nodes	Selection guidelines
Without fault tolerance	1	Select one random node.
Within a single availability zone	5	Select five nodes in different block-4-2 storage pool failure domains to ensure that a majority of 3 working nodes (out of 5) remain when two domains fail.
Geo-distributed	9	Select three nodes in different failure domains within each of the three mirror-3-dc storage pool availability zones to ensure that a majority of 5 working nodes (out of 9) remain when the availability zone + failure domain fail.

When deploying State Storage on clusters that use multiple storage pools with a possible combination of fault tolerance modes, consider increasing the number of nodes and spreading them across different storage pools because unavailability of State Storage results in unavailability of the entire cluster.

Syntax

```
state_storage:
- ring:
    node: <StateStorage node array>
    nto_select: <number of data replicas in StateStorage>
  ssid: 1
```

Each State Storage client (for example, DataShard tablet) uses `nto_select` nodes to write copies of its data to State Storage. If State Storage consists of more than `nto_select` nodes, different nodes can be used for different clients, so you must ensure that any subset of `nto_select` nodes within State Storage meets the fault tolerance criteria.

Odd numbers must be used for `nto_select` because using even numbers does not improve fault tolerance in comparison to the nearest smaller odd number.

Complete Configuration Examples

Single Data Center with `block-4-2` Erasure

```
domains_config:
  domain:
  - name: Root
    storage_pool_types:
    - kind: ssd
      pool_config:
        box_id: 1
        erasure_species: block-4-2
        kind: ssd
        pdisk_filter:
        - property:
          - type: SSD
        vdisk_kind: Default
  state_storage:
  - ring:
      node: [1, 2, 3, 4, 5, 6, 7, 8]
      nto_select: 5
    ssid: 1
```

Multi Data Center with `mirror-3-dc` Erasure

```
domains_config:
  domain:
  - name: global
    storage_pool_types:
    - kind: ssd
      pool_config:
        box_id: 1
        erasure_species: mirror-3-dc
        kind: ssd
        pdisk_filter:
        - property:
          - type: SSD
        vdisk_kind: Default
  state_storage:
  - ring:
      node: [1, 2, 3, 4, 5, 6, 7, 8, 9]
      nto_select: 9
    ssid: 1
```

No Fault Tolerance (`none`) - For Testing

```
domains_config:
  domain:
  - name: Root
    storage_pool_types:
    - kind: ssd
      pool_config:
```

```
        box_id: 1
        erasure_species: none
        kind: ssd
        pdisk_filter:
        - property:
          - type: SSD
        vdisk_kind: Default
  state_storage:
  - ring:
      node: [1]
      nto_select: 1
    ssid: 1
```

Multiple Storage Pool Types

```
domains_config:
  domain:
  - name: Root
    storage_pool_types:
    - kind: ssd
      pool_config:
        box_id: '1'
        erasure_species: block-4-2
        kind: ssd
        pdisk_filter:
        - property:
          - {type: SSD}
        vdisk_kind: Default
    - kind: rot
      pool_config:
        box_id: '1'
        erasure_species: block-4-2
        kind: rot
        pdisk_filter:
        - property:
          - {type: ROT}
        vdisk_kind: Default
    - kind: rotencrypted
      pool_config:
        box_id: '1'
        encryption_mode: 1
        erasure_species: block-4-2
        kind: rotencrypted
        pdisk_filter:
        - property:
          - {type: ROT}
        vdisk_kind: Default
    - kind: ssdencrypted
      pool_config:
        box_id: '1'
        encryption_mode: 1
        erasure_species: block-4-2
        kind: ssdencrypted
        pdisk_filter:
        - property:
          - {type: SSD}
        vdisk_kind: Default
  state_storage:
  - ring:
      node: [1, 16, 31, 46, 61, 76, 91, 106]
      nto_select: 5
    ssid: 1
```

feature_flags

The `feature_flags` section enables or disables certain YDB features using boolean flags. To enable a feature, set the corresponding feature flag to `true` in the cluster configuration. For example, to enable support for auto-partitioning of topics in CDC, add the following lines to the configuration:

```
feature_flags:
  enable_topic_autopartitioning_for_cdc: true
```

Warning

Change the `feature_flags` settings only if you are sure of the consequences. Incorrect configuration may disrupt the cluster.

Feature flags

Flag	Function
`enable_json_index`	JSON indexes to speed up search in JSON fields
`enable_json_index_auto_select`	Automatic selection of JSON indexes when executing queries
`enable_fulltext_index`	Full-text index for full-text search
`enable_local_bloom_filter_index`	Local Bloom index of type `bloom_filter`
`enable_local_bloom_ngram_filter_index`	Local Bloom index of type `bloom_ngram_filter`
`enable_topic_autopartitioning_for_cdc`	Auto-partitioning of topics in CDC for row tables
`enable_access_to_index_impl_tables`	Ability to specify the number of replicas for a secondary index
`enable_changefeeds_export`, `enable_changefeeds_import`	Support for change feeds (changefeed) in backup and restore operations
`enable_view_export`	Support for views (`VIEW`) in backup and restore operations
`enable_export_auto_dropping`	Auto-deletion of temporary directories and tables when exporting to S3
`enable_followers_stats`	System views with information about history of overloaded partitions
`enable_strict_acl_check`	Prohibition on granting permissions to non-existent users and on deleting users who have been granted permissions
`enable_strict_user_management`	Strict rules for administering local users (i.e., only the cluster or database administrator can administer local users)
`enable_database_admin`	Adding the database administrator role
`enable_kafka_native_balancing`	Client-side partition balancing when reading via Kafka protocol
`enable_topic_compactification_by_key`	Enabling topic compaction in YDB Topics Kafka API
`enable_kafka_transactions`	Enabling transactions in YDB Topics Kafka API
`enable_external_data_sources`	Enabling external data sources
`enable_grpc_audit`	Enabling audit of gRPC connection state changes
`enable_fs_backups`	Enabling backup and restore operations to a network file system

query_service_config

The `query_service_config` section describes the parameters for YDB to work with external data sources using federated queries.

If access to the required source requires deploying a connector, it must also be configured according to the instructions.

Parameter description

Parameter	Default value	Description
`generic.connector.endpoint.host`	`localhost`	Connector host name.
`generic.connector.endpoint.port`	`2130`	Connector TCP port.
`generic.connector.use_ssl`	`false`	Whether to use connection encryption. When the connector and YDB dynamic node are deployed on the same server, encrypted connection between them is not required, but it can be enabled if needed.
`generic.connector.ssl_ca_crt`	empty string	Path to the CA certificate used for encryption.
`generic.default_settings.name.UsePredicatePushdown`	`false`	Enables predicate pushdown to external data sources: some parts of SQL queries (for example, filters) will be passed for execution to the external source. This can significantly reduce the volume of data transferred over the network from the data source to federated YDB, save its computational resources, and significantly reduce federated query processing time.
`available_external_data_sources`	empty list	List of allowed external data source types. Applied when `all_external_data_sources_are_available: false`. Possible values: • `ObjectStorage`; • `ClickHouse`; • `PostgreSQL`; • `MySQL`; • `Greenplum`; • `MsSQLServer`; • `Ydb`.
`all_external_data_sources_are_available`	`false`	Enable all external data source types. When enabled, the `available_external_data_sources` setting is ignored.

Examples

Enabling ClickHouse and MySQL external sources

```
query_service_config:
  generic:
    connector:
      endpoint:
        host: localhost                  # host name where the connector is deployed
        port: 2130                       # connector port number
      use_ssl: false                     # flag to enable connection encryption
      ssl_ca_crt: "/opt/ydb/certs/ca.crt" # path to CA certificate
    default_settings:
    - name: UsePredicatePushdown
      value: "true"
  all_external_data_sources_are_available: false
  available_external_data_sources:
  - ClickHouse
  - MySQL
```

Enabling all external data source types

```
query_service_config:
  generic:
    connector:
      endpoint:
        host: localhost                  # host name where the connector is deployed
        port: 2130                       # connector port number
      use_ssl: false                     # flag to enable connection encryption
      ssl_ca_crt: "/opt/ydb/certs/ca.crt" # path to CA certificate
    default_settings:
    - name: UsePredicatePushdown
      value: "true"
  all_external_data_sources_are_available: true
```

See also

- Deploying YDB with Federated Query functionality

healthcheck_config

The `healthcheck_config` section configures thresholds and timeout settings used by the YDB health check service. These parameters help configure detection of potential issues, such as excessive restarts or time drift between dynamic nodes.

Syntax

```
healthcheck_config:
  thresholds:
    node_restarts_yellow: 10
    node_restarts_orange: 30
    nodes_time_difference_yellow: 5000
    nodes_time_difference_orange: 25000
    tablets_restarts_orange: 30
  timeout: 20000
```

Parameters

Parameter	Default	Description
`thresholds.node_restarts_yellow`	`10`	Number of node restarts to trigger a `YELLOW` warning
`thresholds.node_restarts_orange`	`30`	Number of node restarts to trigger an `ORANGE` alert
`thresholds.nodes_time_difference_yellow`	`5000`	Max allowed time difference (in us) between dynamic nodes for `YELLOW` issue
`thresholds.nodes_time_difference_orange`	`25000`	Max allowed time difference (in us) between dynamic nodes for `ORANGE` issue
`thresholds.tablets_restarts_orange`	`30`	Number of tablet restarts to trigger an `ORANGE` alert
`timeout`	`20000`	Maximum health check response time (in ms)

host_configs

A YDB cluster consists of multiple nodes, and one or more typical server configurations are usually used for their deployment. To avoid repeating their description for each node, there is a `host_configs` section in the configuration file that lists the used configurations and assigned IDs.

Syntax

```
host_configs:
- host_config_id: 1
  drive:
  - path: <path_to_device>
    type: <type>
  - path: ...
- host_config_id: 2
  ...
```

The `host_config_id` attribute specifies a numeric configuration ID. The `drive` attribute contains a collection of descriptions of connected drives. Each description consists of two attributes:

- `path` : Path to the mounted block device, for example, `/dev/disk/by-partlabel/ydb_disk_ssd_01`
- `type` : Type of the device's physical media: `ssd` , `nvme` , or `rot` (rotational - HDD)

Examples

One configuration with ID 1 and one SSD disk accessible via `/dev/disk/by-partlabel/ydb_disk_ssd_01` :

```
host_configs:
- host_config_id: 1
  drive:
  - path: /dev/disk/by-partlabel/ydb_disk_ssd_01
    type: SSD
```

Two configurations with IDs 1 (two SSD disks) and 2 (three SSD disks):

```
host_configs:
- host_config_id: 1
  drive:
  - path: /dev/disk/by-partlabel/ydb_disk_ssd_01
    type: SSD
  - path: /dev/disk/by-partlabel/ydb_disk_ssd_02
    type: SSD
- host_config_id: 2
  drive:
  - path: /dev/disk/by-partlabel/ydb_disk_ssd_01
    type: SSD
  - path: /dev/disk/by-partlabel/ydb_disk_ssd_02
    type: SSD
  - path: /dev/disk/by-partlabel/ydb_disk_ssd_03
    type: SSD
```

Kubernetes Features

The YDB Kubernetes operator mounts NBS disks for Storage nodes at the path `/dev/kikimr_ssd_00` . To use them, the following `host_configs` configuration must be specified:

```
host_configs:
- host_config_id: 1
  drive:
  - path: /dev/kikimr_ssd_00
    type: SSD
```

The example configuration files provided with the YDB Kubernetes operator contain this section, and it does not need to be changed.

hosts

This group lists the static cluster nodes on which the Storage processes run and specifies their main characteristics:

- Numeric node ID
- DNS host name and port that can be used to connect to a node on the IP network
- ID of the standard host configuration
- Placement in a specific availability zone, rack
- Server inventory number (optional)

Syntax

```
hosts:
- host: <DNS host name>
  host_config_id: <numeric ID of the standard host configuration>
  port: <port> # 19001 by default
  location:
    unit: <string with the server serial number>
    data_center: <string with the availability zone ID>
    rack: <string with the rack ID>
- host: <DNS host name>
  ...
```

Examples

```
hosts:
- host: hostname1
  host_config_id: 1
  node_id: 1
  port: 19001
  location:
    unit: '1'
    data_center: '1'
    rack: '1'
- host: hostname2
  host_config_id: 1
  node_id: 2
  port: 19001
  location:
    unit: '1'
    data_center: '1'
    rack: '1'
```

Bridge mode specifics

In bridge mode, each host must be bound to one of the piles declared in `bridge_config`. Add `bridge_pile_name` under `location` with the pile name. Example:

```
hosts:
- host: hostname1
  host_config_id: 1
  location:
    unit: '1'
    data_center: '1'
    rack: '1'
    bridge_pile_name: 'pile_1'
```

Kubernetes-Specific Details

When deploying YDB with a Kubernetes operator, the entire `hosts` section is generated automatically, replacing any user-specified content in the configuration passed to the operator. All Storage nodes use `host_config_id` = `1`, for which the correct configuration must be specified.

log_config

The `log_config` section controls how YDB server processes and manages its logs. It allows you to customize logging levels for different components, as well as global log formats and output methods.

> **Note**
>
> This document describes application and system logging configuration. For security and audit logging, see Audit log.

Overview

Logging is a critical part of the YDB observability system. The `log_config` section lets you configure various aspects of logging, including:

- Default logging level
- Component-specific logging levels
- Log output format
- Integration with system logs or external logging services

Log Output Methods

- **To stderr**: by default, YDB sends all logs to stderr.
- **To file**: logs can be written to a file using the `backend_file_name` parameter.
- **To syslog**: when the `sys_log: true` parameter is enabled, logs are redirected to the syslog and stop being output to stderr. Logs are sent using the `/dev/log` socket.
- **To Unified Agent**: when configuring the `uaclient_config` section, logs are sent to Unified Agent and stop being output to stderr.

When both `sys_log` and `uaclient_config` are enabled simultaneously, logs will be sent to both syslog and Unified Agent. If you need to continue outputting logs to stderr while using other methods, activate `sys_log_to_stderr: true`.

Configuration Options

Parameter	Type	Default	Description
`default_level`	uint32	5 (`NOTICE`)	Default logging level for all components.
`default_sampling_level`	uint32	7 (`DEBUG`)	Default sampling level for all components.
`default_sampling_rate`	uint32	0	Default sampling rate for all components. If set to N (where N > 0), approximately 1 out of every N log messages with priority between `default_level` and `default_sampling_level` will be logged. For example, to log every 10th message, set to 10. A value of 0 means that no messages in this range will be logged (they are all dropped).
`sys_log`	bool	false	Enable system logging via syslog.
`sys_log_to_stderr`	bool	false	Copy logs to stderr in addition to system log.
`format`	string	"full"	Log output format. Possible values: "full", "short", "json".
`cluster_name`	string	—	Cluster name to include in log records. The `cluster_name` field is added to logs only when using the `json` format or when sending to Unified Agent. In the `full` or `short` formats, this field is not displayed.
`allow_drop_entries`	bool	true	Allow dropping log entries if the logging system is overloaded. When enabled, log entries are buffered in memory and written to the output when either 10 messages accumulate or the time specified by `time_threshold_ms` elapses. If the buffer becomes full, lower-priority messages may be dropped to make room for higher-priority ones.
`use_local_timestamps`	bool	false	Use local time zone for log timestamps (UTC is used by default).
`backend_file_name`	string	—	File name for log output. If specified, logs are written to this file.
`sys_log_service`	string	—	Service name for syslog. Corresponds to the tag field in the old syslog RFC 3164 or the app-name field in the modern RFC 5424 protocol.
`time_threshold_ms`	uint64	1000	If `allow_drop_entries = true`, specifies how often YDB writes buffered log messages to the output, in milliseconds.
`ignore_unknown_components`	bool	true	Ignore logging requests from unknown components.
`entry`	array	[]	Configuration of logging level and/or sampling for specific YDB components, see Entry Objects below.
`uaclient_config`	object	—	Configuration for the Unified Agent client, see UAClientConfig Object below.
`enable_structured_log_in_json`	bool	false	For `format: "json"`: if `false`, `message` includes structured values as `key=value` (for example, `Text (a=1 b=2)`); if `true`, `message` contains only the text, and structured values are emitted as separate JSON string fields (for example, `{"message":"Text","a":"1","b":"2"}`). Reserved field names are prefixed with `_`.

Entry Objects

The `entry` field contains an array of objects with the following structure:

Parameter	Type	Description
`component`	string	Component name. See the full list of available components on GitHub.
`level`	uint32	Log level for this component.
`sampling_level`	uint32	Sampling level for this component. Works similarly to `default_sampling_level`.
`sampling_rate`	uint32	Sampling rate for this component. Works similarly to `default_sampling_rate`.

UAClientConfig Object

The `uaclient_config` field configures integration with Unified Agent:

Parameter	Type	Default	Description
`uri`	string	—	grpc URI of the Unified Agent server.
`shared_secret_key`	string	—	Path to the file with the secret key for client connection authentication.
`max_inflight_bytes`	uint64	100000000	Maximum number of bytes in transit when sending data.
`grpc_reconnect_delay_ms`	uint64	—	Delay between reconnection attempts in milliseconds.
`grpc_send_delay_ms`	uint64	—	Delay between send attempts in milliseconds.
`grpc_max_message_size`	uint64	—	Maximum gRPC message size.
`client_log_file`	string	—	Log file for the UA client itself.
`client_log_priority`	uint32	—	Logging level for the UA client itself.
`log_name`	string	—	Log name that is passed in session metadata.

Log Levels

YDB uses the following log levels, listed from the highest to the lowest severity:

Level	Numeric value	Description
`EMERG`	0	System outage (for example, cluster failure) is possible.
`ALERT`	1	System degradation is possible, system components may fail.
`CRIT`	2	A critical state.
`ERROR`	3	A non-critical error.
`WARN`	4	A warning, it should be responded to and fixed unless it's temporary.
`NOTICE`	5	An event essential for the system or the user has occurred.
`INFO`	6	Debugging information for collecting statistics.
`DEBUG`	7	Debugging information for developers.
`TRACE`	8	Detailed debugging information.

Examples

Basic Configuration

```
log_config:
  default_level: 5  # NOTICE
  format: "full"
```

This configuration outputs logs to stderr with logging level `NOTICE` and above.

File Output Configuration

```
log_config:
  default_level: 5  # NOTICE
  format: "full"
  backend_file_name: "/var/log/ydb/ydb.log"
```

This configuration sends logs to a file while maintaining the default logging level of `NOTICE`.

Syslog Output Configuration

```
log_config:
  default_level: 5  # NOTICE
  sys_log: true
```

```
  sys_log_service: "ydb"
  format: "full"
```

This configuration sends logs to syslog with the service name "ydb".

Setting Per-Component Log Levels

```
log_config:
  default_level: 5  # NOTICE
  entry:
    - component: "SCHEMESHARD"
      level: 7  # DEBUG
    - component: "TABLET_MAIN"
      level: 6  # INFO
  backend_file_name: "/var/log/ydb/ydb.log"
```

Enabling transaction lock invalidation diagnostics (TLI)

To diagnose `transaction locks invalidated` errors, enable logging for the `TLI` component:

```
log_config:
  entry:
    - component: "TLI"
      level: 6  # INFO
```

Once enabled, the server writes logs on every lock conflict, including identifiers and query texts for both victims and breakers. For details on the log entry format, event correlation, and the `find_tli_chain` utility, see TLI logging.

Sampling Configuration

```
log_config:
  default_level: 5  # NOTICE
  default_sampling_level: 7  # DEBUG
  default_sampling_rate: 10  # Log every 10th message between NOTICE and DEBUG
  entry:
    - component: "BLOBSTORAGE"
      sampling_level: 8  # TRACE
      sampling_rate: 100  # Log every 100th message between NOTICE and TRACE
```

This configuration sets up sampling for logs. With default settings, every 10th message with priority between `NOTICE` and `DEBUG` will be logged. For the `BLOBSTORAGE` component, every 100th message with priority between `NOTICE` and `TRACE` will be logged.

JSON Format Configuration

```
log_config:
  default_level: 5  # NOTICE
  format: "json"
  cluster_name: "production-cluster"
  uaclient_config:
    uri: "[fd53::1]:16400"
    grpc_max_message_size: 4194304
    log_name: "ydb_logs"
```

This configuration outputs logs in JSON format and sends them to Unified Agent.

Notes

- Log levels are specified in the configuration as numeric values, not strings. Use the table above to map between numeric values and their meanings.
- If the `backend_file_name` parameter is specified, logs are written to this file. If the `sys_log` parameter is true, logs are sent to the system logger.
- The `format` parameter determines how log entries are formatted. The "full" format includes all available information, "short" provides a more compact format, and "json" outputs logs in JSON format, which is convenient for parsing by logging services.
- The internal log buffer has the following size limits:
  - Default total size: 10MB (10 * 1024 * 1024 bytes)
  - Default grain size: 64KB (1024 * 64 bytes)
  - Maximum message size: 1KB (1024 bytes)

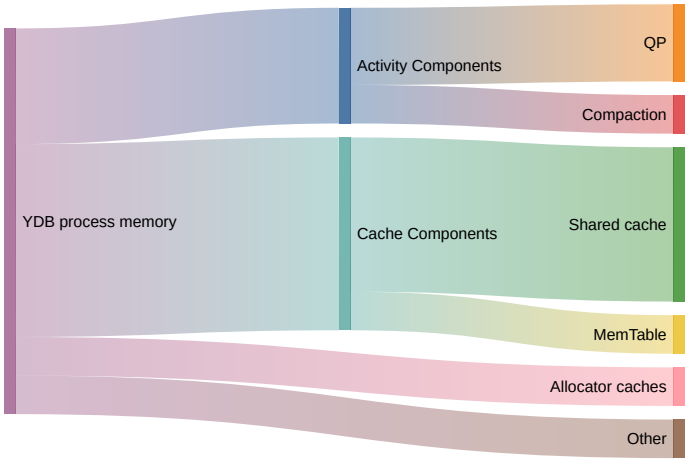
See Also

- Reference on YDB observability
- Metrics reference
- Tracing in YDB
- Audit log

memory_controller_config

Inside nodes, YDB runs many different components that use memory. Most of them require a fixed amount of memory, but some can flexibly vary the amount of memory they use, thereby improving the performance of the entire system.

Overview of memory consumption inside a YDB node by component



If YDB components allocate more memory than is physically available, the operating system will likely terminate the entire YDB process, which is highly undesirable. The goal of the memory controller is to allow YDB to avoid out-of-memory situations while efficiently using the available memory.

Examples of components managed by the memory controller:

- Shared cache: stores recently accessed data pages read from distributed storage to reduce disk I/O operations and speed up data retrieval.
- MemTable: contains data that has not yet been written to SST.
- Query Processor: stores intermediate query processing results.
- Compaction: the process of ordering and cleaning data that runs automatically (in the background) to optimize storage volume.
- Allocator caches: store memory blocks that have been freed but not yet returned to the operating system.

Memory limits can be configured to control overall memory usage, ensuring efficient database operation within available resources.

Hard memory limit

The hard memory limit defines the total amount of memory available to the YDB process.

By default, the hard memory limit for the YDB process equals the memory limit specified in its cgroups.

In environments without a cgroups memory limit, the default hard memory limit equals the total amount of available host memory. This configuration allows the database to use all available resources but may lead to resource contention with other processes on the same host. Although the memory controller tries to account for this external consumption, such usage is not recommended.

The hard memory limit can also be set in the configuration. Note that the database process may still exceed this limit. Therefore, it is strongly recommended to use cgroups memory limits in production environments for strict memory control.

Most other memory limits can be configured either in absolute bytes or as percentages relative to the hard memory limit. Using percentages is convenient for managing clusters with nodes of different capacities. If both absolute byte limits and percentage limits are specified, the memory controller uses a combination of both (maximum for lower limits and minimum for upper limits).

Example of a `memory_controller_config` section with a specified hard memory limit:

```
memory_controller_config:
  hard_limit_bytes: 16106127360
```

Soft memory limit

The soft memory limit defines a dangerous threshold that the YDB process should not exceed under normal circumstances.

If the soft limit is exceeded, YDB gradually reduces the size of the shared cache to zero. In this case, you should add more database nodes to the cluster or reduce memory limits for individual components as soon as possible.

Target memory usage

Target memory usage defines a memory usage threshold for the YDB process that is considered optimal.

Flexible cache sizes are calculated according to their limits to keep the process memory consumption near this value.

For example, in a database that uses little memory for query execution, caches use memory near this threshold, and the remaining memory stays free. If query execution starts consuming more memory, caches begin to reduce their sizes to the minimum threshold.

Memory limits for individual components

Inside YDB, there are two different types of components.

The first type of components, or cache components, function as caches, for example, storing recently used data. Each cache component has minimum and maximum threshold values for the memory limit, allowing it to dynamically adjust its capacity based on the current memory consumption of the YDB process.

The second type of components, or activity components, allocate memory for specific tasks, such as executing queries or the compaction process. Each activity component has a fixed memory limit. There is also an additional shared memory limit for such components, from which they attempt to obtain the necessary memory.

Many other auxiliary components and processes run in parallel with the YDB process, consuming memory. Currently, these components do not have any memory limits.

Memory limits for cache components

Cache components include:

- Shared cache.
- MemTable.

The limits of each cache component are dynamically recalculated every second, so that each component consumes memory proportionally to its limit values, and the total memory consumption remains around the target memory usage.

The minimum threshold of the memory limit for cache components is not reserved, meaning that memory remains available until it is actually used. However, once this memory is filled, components typically retain data, operating within their current memory limit. Thus, the sum of the minimum memory limits of cache components is expected to be less than the target memory usage.

If necessary, both minimum and maximum threshold values should be overridden; otherwise, if a threshold value is missing, it will take the default value.

Example of the `memory_controller_config` section with specified shared cache limits:

```
memory_controller_config:
  shared_cache_min_percent: 10
  shared_cache_max_percent: 30
```

Memory limits for activity components

Activity components include:

- Query Processor.
- Compaction.

The memory limit for each activity component specifies the maximum amount of memory it can attempt to use. However, to prevent the YDB process from exceeding the soft memory limit, the total consumption of activity components is limited by an additional limit called the activity memory limit. If the total memory usage of active components exceeds this limit, any additional memory requests will be denied. When query execution approaches the memory limits, YDB activates spilling to temporarily save intermediate data to disk, preventing memory limit violations.

Thus, although the total individual limits of activity components may collectively exceed the activity memory limit, the individual limit of each component must be less than this overall limit. Additionally, the sum of the minimum memory limits for cache components plus the activity memory limit must be less than the soft memory limit.

There are other activity components that currently do not have any individual memory limits.

Example of the `memory_controller_config` section with the specified limit for QP:

```
memory_controller_config:
  query_execution_limit_percent: 25
```

Query Processor memory limit

The `query_execution_limit_percent` (default 20% of the hard memory limit) and `query_execution_limit_bytes` parameters set the size of the RAM pool that Resource Manager allocates to queries on a node:

If both parameters are set, the effective limit is `min(query_execution_limit_percent × hard_limit_bytes / 100, query_execution_limit_bytes)`. If only one parameter is set, its value is used.

The computed limit is passed to Resource Manager via Resource Broker.

The `spilling_percent` threshold in `table_service_config` is calculated relative to this pool.

Configuration parameters

Each configuration parameter applies in the context of a single database node.

As mentioned earlier, the sum of the minimum memory limits for cache components plus the activity memory limit is expected to be less than the soft memory limit.

This constraint can be expressed in a simplified form:

Or in a detailed form:

Parameters	Default value	Description
`hard_limit_bytes`	CGroup memory limit / Host memory	Hard limit for memory usage.
`soft_limit_percent` / `soft_limit_bytes`	75%	Soft limit for memory usage.
`target_utilization_percent` / `target_utilization_bytes`	50%	Target memory usage.
`activities_limit_percent` / `activities_limit_bytes`	30%	Memory limit for activities.
`shared_cache_min_percent` / `shared_cache_min_bytes`	20%	Minimum threshold for the shared cache memory limit.
`shared_cache_max_percent` / `shared_cache_max_bytes`	50%	Maximum threshold for the shared cache memory limit.
`mem_table_min_percent` / `mem_table_min_bytes`	1%	Minimum threshold for the MemTable memory limit.

`mem_table_max_percent` / `mem_table_max_bytes`	3%	Maximum threshold for the MemTable memory limit.
`query_execution_limit_percent` / `query_execution_limit_bytes`	20%	Memory limit for QP.
`compaction_limit_percent` / `compaction_limit_bytes`	10%	Memory limit for compaction.

node_broker_config

The `node_broker_config` section configures stable node names for dynamic nodes in YDB clusters. Node names are assigned through the Node Broker, which is a system tablet that registers dynamic nodes in the cluster.

Node Broker assigns names to dynamic nodes when they register in the cluster. By default, a node name consists of the hostname and the port on which the node is running.

In a dynamic environment where hostnames often change, such as in Kubernetes, using hostname and port leads to an uncontrollable increase in the number of unique node names. This is true even for a database with a handful of dynamic nodes. Such behavior may be undesirable for a time series monitoring system as the number of metrics grows uncontrollably. To solve this problem, the system administrator can set up *stable* node names.

A stable name identifies a node within the tenant. It consists of a prefix and a node's sequential number within its tenant. If a dynamic node has been shut down, after a timeout, its stable name can be taken by a new dynamic node serving the same tenant.

To enable stable node names, you need to add the following to the cluster configuration:

```
feature_flags:
  enable_stable_node_names: true
```

By default, the prefix is `slot-`. To override the prefix, add the following to the cluster configuration:

```
node_broker_config:
  stable_node_name_prefix: <new prefix>
```

resource_broker_config

The resource broker is an actor service that controls resource consumption on YDB nodes, such as:

- `CPU` — number of threads
- `Memory` — RAM

Different types of activities (background operations, TTL data deletion, etc.) run in different resource broker *queues*. Each queue has a limited number of resources:

Queue name	CPU	Memory	Description
`queue_ttl`	2	—	TTL data deletion operations.
`queue_backup`	2	—	Backup operations.
`queue_restore`	10	—	Restore from backup operations.
`queue_build_index`	10	—	Online secondary index creation operations.
`queue_cdc_initial_scan`	4	—	Initial table scan operations.

Note

It is recommended to **extend** the resource broker configuration using tags `!inherit` and `!append`.

Example of extending the resource broker configuration with a custom limit for the `queue_ttl` queue:

```
resource_broker_config: !inherit
  queues: !append
  - name: queue_ttl
    limit:
      cpu: 4
```

security_config

The `security_config` section defines authentication modes, the initial configuration of local users and groups, and their access rights.

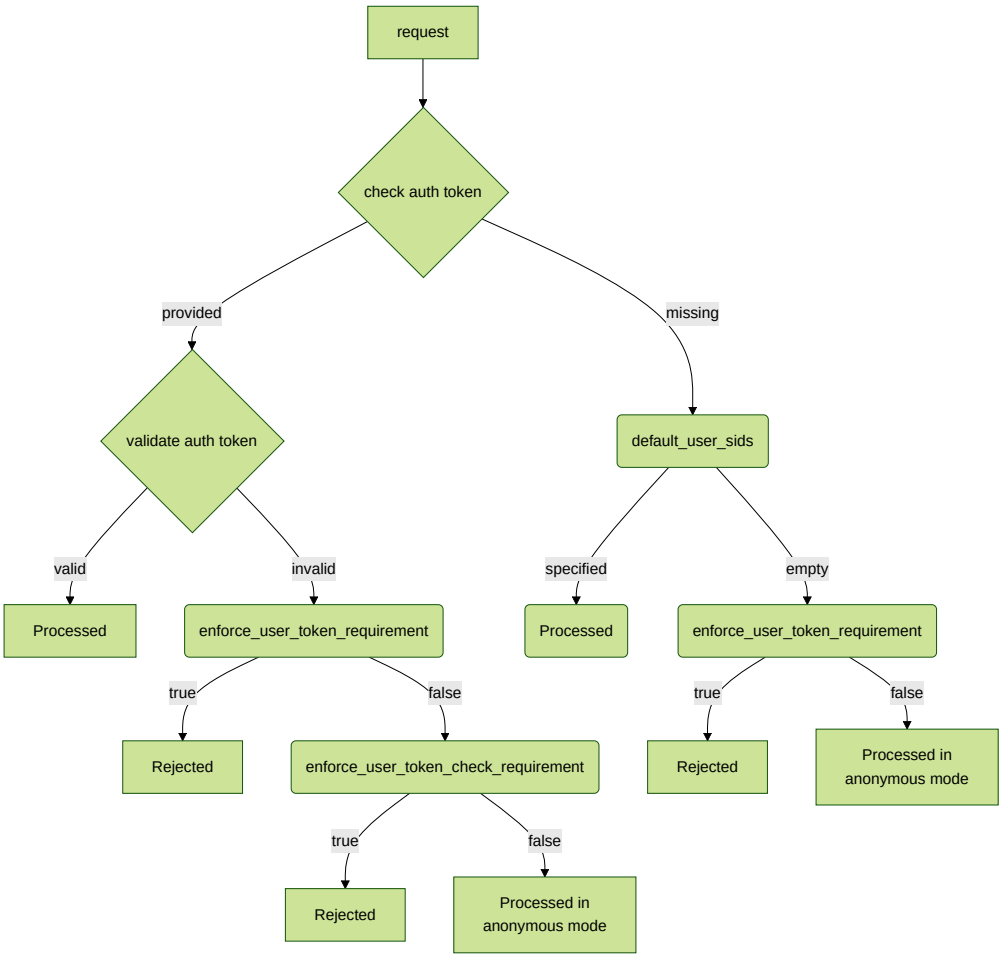
```
security_config:
  # authentication mode configuration
  enforce_user_token_requirement: false
  enforce_user_token_check_requirement: false
  default_user_sids: <authentication token for anonymous requests>
  all_authenticated_users: <group name for all authenticated users>
  all_users_group: <group name for all users>

  # initial security configuration
  default_users: <default user list>
  default_groups: <default group list>
  default_access: <default access rights on the cluster scheme root>

  # access list configuration
  database_allowed_sids: <list of SIDs that are allowed to view database state>
  viewer_allowed_sids: <list of SIDs that are allowed to view the cluster state>
  monitoring_allowed_sids: <list of SIDs that are allowed to monitor and change the cluster state>
  administration_allowed_sids: <list of SIDs that are allowed cluster administration>
  bootstrap_allowed_sids: <list of SIDs allowed to bootstrap the cluster>
  register_dynamic_node_allowed_sids: <list of SIDs that are allowed to register database nodes in the cluste
r>

  # built-in security configuration
  disable_builtin_security: false
  disable_builtin_groups: false
  disable_builtin_access: false
```

Configuring Authentication Mode

Parameter	Description
`enforce_user_token_requirement`	Selects user authentication mode. • `enforce_user_token_requirement: true` — User authentication is mandatory. Requests to YDB must include an auth token. Requests to YDB undergo authentication and authorization. • `enforce_user_token_requirement: false` — User authentication is optional. Requests to YDB are not required to include an auth token. Requests without an auth token are processed in anonymous mode without authorization. Requests with an auth token undergo authentication and authorization. However, requests are still processed in anonymous mode if an authentication error occurs. When `enforce_user_token_check_requirement: true`, requests with authentication errors are blocked. If the `default_user_sids` parameter is defined and not empty (see the description below), its value is used instead of the missing auth token. In this case, authentication and authorization are performed for the access subject defined in `default_user_sids`. Default value: `false`.
`enforce_user_token_check_requirement`	Forbids ignoring authentication errors in the `enforce_user_token_requirement: false` mode. Default value: `false`.
`default_user_sids`	Specifies a list of SIDs for authenticating incoming requests without an auth token. `default_user_sids` acts as an auth token for anonymous requests. The first element in the list must be a user SID. The following elements must be the SIDs of groups to which the user belongs. If the `default_user_sids` list is not empty, mandatory authentication mode (`enforce_user_token_requirement: true`) can be used for anonymous requests. This mode can be used in some YDB testing scenarios or for educational purposes in local YDB installations. Default value: empty.
`all_authenticated_users`	Specifies the name of the virtual group that includes all authenticated users. This virtual group is created automatically by YDB. You cannot delete this virtual group, list its members, or modify them. You can use this group to grant access rights on scheme objects. **Tip** You can get information about access rights on scheme objects in the system views. For more information see, {#T}. Default value: `all-users@well-known`.

`all_users_group`	Specifies the name of the group that includes all local users. If `all_users_group` is not empty, all local users will be added to the group with this name upon creation. The group specified in this parameter must exist when new users are added. The `all_users_group` parameter is used during the initialization of built-in security. Default value: empty.

The following diagram displays the relationship between authentication mode parameters described above:



Bootstrapping Security

The `default_users`, `default_groups`, and `default_access` parameters affect the initial YDB cluster configuration that occurs when YDB starts for the first time. During subsequent runs, the initial configuration is not repeated, and these parameters are ignored.

See Initial cluster security configuration and the related `domains_config` parameters.

Parameter	Description

`default_users`	The list of users to be created when the YDB cluster starts for the first time.
	
	The list consists of login-password pairs. The first user in the list is a superuser.
	
	Warning
	Passwords are specified in plain text, so it is unsafe to use them for an extended period. You must change these passwords in YDB after the first start. For example, use the `ALTER USER` statement.
	
	Example:
	
	`default_users:`
	`- name: root`
	`  password: <...>`
	`- name: user1`
	`  password: <...>`
	
	Errors in the `default_users` list, such as duplicate logins, are logged but do not affect YDB cluster startup.

`default_groups`	The list of groups to be created when the YDB cluster starts for the first time.
	
	The list includes groups and their members.
	
	Warning
	These groups are created for the entire YDB cluster.
	
	Example:
	
	``
	default_groups:
	- name: ADMINS
	  members: root
	- name: USERS
	  members:
	  - ADMINS
	  - root
	  - user1
	``
	
	The order of groups in this list matters: groups are created in the order in which they appear in the `default_groups` parameter. Group members must exist before the group is created. Nonexistent users will not be added to the group.
	
	Failures to add users to groups are logged but do not affect the YDB cluster startup.

`default_access`	The list of access rights to be granted on the cluster scheme root. Access rights are specified using the short access control notation. Example: ``` default_access: - +(CDB|DDB|GAR):ADMINS - +(ConnDB):USERS ``` Errors in access right entries are logged but do not affect YDB cluster startup. Access rights with errors will not be granted.

Configuring Administrative and Other Privileges

Access level lists are configured in the `security_config` section. For detailed information about access level lists, their hierarchy, and how they work, see Access level lists in the Authorization documentation.

Parameter	Description
`database_allowed_sids`	The list of SIDs with the database access level. This level is less privileged than viewer. Subjects can only operate within a database scope — backend calls without a specified database are forbidden. Cluster-level requests (e.g., listing cluster nodes) return 403. Database-scoped requests (e.g., database pages in YDB UI) work normally. Intended for applications or users restricted to a single database.
`viewer_allowed_sids`	The list of SIDs with the viewer access level. This level allows viewing the cluster state, which is not publicly accessible (including most pages in the YDB UI). No changes are allowed.
`monitoring_allowed_sids`	The list of SIDs with the monitoring access level. This level grants additional privileges to monitor and modify the cluster state. For example, it allows performing a backup, restoring a database, or executing YQL statements in the YDB UI.
`administration_allowed_sids`	The list of SIDs with the administration access level. This level grants privileges to administer the YDB cluster and its databases. Also used for config changes, scheme operations requiring admin privileges, and other administrative checks. An empty list means any user is treated as an administrator.
`bootstrap_allowed_sids`	The list of SIDs that are allowed to perform cluster bootstrap. Used to initially bootstrap the cluster from an uninitialized state (when the auth subsystem is not yet functional). Bootstrap is allowed if the subject is in `bootstrap_allowed_sids` or `administration_allowed_sids`. `bootstrap_allowed_sids` allows specific identities to bootstrap the cluster without granting full administration privileges.
`register_dynamic_node_allowed_sids`	The list of SIDs that are allowed to register database nodes. For technical reasons, this list must include `root@builtin`.

Warning

The access level lists are empty by default.

An **empty** list means "allow any user" — any SID is acceptable (or no SID at all if anonymous access is allowed). Empty `administration_allowed_sids` grants any user administrator privileges; empty `viewer_allowed_sids` (with other UI lists empty) allows any user viewer-level access to YDB UI. If all four hierarchical lists are empty, any user has full administrative access.

For a secure YDB deployment, plan the access model beforehand and define the group lists before starting the cluster for the first time.

Access level lists can contain user or group SIDs. A subject matches a list if it contains their user SID or any of their group SIDs (including nested groups) — a single match is sufficient.

It is recommended to add user groups and separate service accounts to the `*_allowed_sids` access level lists. This way, granting access levels to individual users does not require changing the YDB cluster configuration.

Where the lists are used:

- `database_allowed_sids`, `viewer_allowed_sids`, `monitoring_allowed_sids`, `administration_allowed_sids`
  To access YDB UI and viewer HTTP endpoints.
- `administration_allowed_sids`
  For all administrative checks.
- `bootstrap_allowed_sids`
  Only for cluster bootstrap operation.
- `register_dynamic_node_allowed_sids`
  For registering nodes in a cluster (discovery service and cms subsystem).

Note

Access level lists represent layers of additional privileges:

- An access subject that is not included in any access level list can view only publicly available information about the cluster (for example, a list of databases on the cluster or a list of cluster nodes).
- The `database_allowed_sids` list defines the "database" role: these users are less privileged than viewers and cannot issue any backend call without specifying a database; cluster-wide requests are not allowed.
- The `viewer_allowed_sids`, `monitoring_allowed_sids`, and `administration_allowed_sids` lists form a hierarchy: administration implies monitoring and viewer; monitoring implies viewer. A subject only needs to be in the one list that matches their role — inclusion in a higher list automatically grants all lower privileges.

For example:

- A user of a database (e.g. an application that must stay database-scoped) should be added only to `database_allowed_sids`.
- A viewer should be added only to `viewer_allowed_sids`.
- A monitoring user should be added only to `monitoring_allowed_sids` (they automatically get viewer privileges).
- An administrator should be added only to `administration_allowed_sids` (they automatically get monitoring and viewer privileges).

Built-in Security Configuration

The `disable_builtin_security`, `disable_builtin_groups`, and `disable_builtin_access` flags affect the built-in security configuration that occurs when YDB starts for the first time.

Parameter	Description
`disable_builtin_security`	Disable the built-in security configuration. Built-in security configuration automatically creates a `root` superuser, a set of built-in user groups, and grants access rights to these groups at the root of the cluster. This flag is not saved in the cluster configuration. Default value: `false`.
`disable_builtin_groups`	Do not create built-in user groups even if the default user groups are not specified in the `security_config.default_groups` parameter. Default value: `false`
`disable_builtin_access`	Do not add access rights at the root of the cluster scheme for the built-in user groups even if the default access rights are not specified in the `security_config.default_access` parameter. Default value: `false`

self_management_config

The `self_management_config` section configures distributed configuration V2 and automatic management of cluster components.

Syntax

```
self_management_config:
  enabled: true
  automatic_static_group_management: true
  static_group_self_heal_allowed_nodes:
  - 1
  - 2
  - 3
```

Parameters

Parameter	Default value	Description
`enabled`	`false`	Enables the distributed configuration mechanism in configuration V2.
`automatic_static_group_management`	`false`	Allows distributed configuration to change static group VDisk placement automatically.
`static_group_self_heal_allowed_nodes`	`[]`	Restricts the set of nodes to which SelfHeal can move a static group VDisk. Specify a list of node IDs. An empty list means that no restrictions apply.

system_tablet_backup_config

The system tablet backup mechanism provides incremental copying of cluster metadata — such as Hive, BSController, and SchemeShard — to the local file system of cluster hosts. The backup behavior is configured in the `system_tablet_backup_config` section of the YDB configuration.

Syntax

```
system_tablet_backup_config:
    filesystem:
        path: "/path/to/backup/directory"
    exclude_tablet_ids:
        - 72057594037968897
    max_backups_limit: 3
```

Parameters

Parameter	Default value	Description
`filesystem.path`	—	Absolute path to a directory on the host's local file system for storing backups. The directory must be writable by the YDB process on each host where system tablets run. Only local file systems are supported; network file systems such as NFS are not supported.
`exclude_tablet_ids`	—	List of system tablet identifiers to exclude from backup, for example, to reduce disk load or save space.
`max_backups_limit`	`3`	Maximum number of backups of a single tablet stored on the host. After a new copy is successfully created, the oldest one is automatically deleted when the limit is exceeded.

tls

The `tls` section configures TLS settings for data-in-transit encryption in YDB. Each network protocol can have different TLS settings to secure communication between cluster components and clients.

Interconnect

The YDB actor system interconnect is a specialized protocol for communication between YDB nodes.

Example of enabling TLS for the interconnect:

```
interconnect_config:
   start_tcp: true
   encryption_mode: REQUIRED # or OPTIONAL
   path_to_certificate_file: "/opt/ydb/certs/node.crt"
   path_to_private_key_file: "/opt/ydb/certs/node.key"
   path_to_ca_file: "/opt/ydb/certs/ca.crt"
```

YDB as a server

gRPC

The main YDB API is based on gRPC. It is used for external communication with client applications that work natively with YDB via the SDK or CLI.

Example of enabling TLS for gRPC API:

```
grpc_config:
   cert: "/opt/ydb/certs/node.crt"
   key: "/opt/ydb/certs/node.key"
   ca: "/opt/ydb/certs/ca.crt"
```

Kafka Wire Protocol

YDB exposes a separate network port for the Kafka wire protocol. This protocol is used for external communication with client applications initially designed to work with Apache Kafka.

Example of enabling TLS for the Kafka wire protocol with a file containing both the certificate and the private key:

```
kafka_proxy_config:
    ssl_certificate: "/opt/ydb/certs/node.crt"
```

Example of enabling TLS for the Kafka wire protocol with the certificate and private key in separate files:

```
kafka_proxy_config:
    cert: "/opt/ydb/certs/node.crt"
    key: "/opt/ydb/certs/node.key"
```

HTTP

YDB exposes a separate HTTP network port for running the YDB UI, exposing metrics, and other miscellaneous endpoints.

Example of enabling TLS on the HTTP port, making it HTTPS:

```
monitoring_config:
    monitoring_certificate_file: "/opt/ydb/certs/node.crt"
```

YDB as a client

LDAP

YDB supports LDAP for user authentication. The LDAP protocol has two options for enabling TLS.

Example of enabling TLS for LDAP via the `StartTls` protocol extension:

```
auth_config:
  ldap_authentication:
    use_tls:
      enable: true
      ca_cert_file: "/path/to/ca.pem"
      cert_require: DEMAND
  scheme: "ldap"
```

Example of enabling TLS for LDAP via `ldaps`:

```
auth_config:
  ldap_authentication:
    use_tls:
      enable: false
      ca_cert_file: "/path/to/ca.pem"
      cert_require: DEMAND
  scheme: "ldaps"
```

Federated Queries

Federated queries allow YDB to query various external data sources. Whether these queries occur over TLS-encrypted connections is controlled by the `USE_TLS` setting of `CREATE EXTERNAL DATA SOURCE` queries. No changes to the server-side configuration are required.

Tracing

YDB can send tracing data to an external collector via gRPC.

Example of enabling TLS for tracing data by specifying `grpcs://` protocol:

```
tracing_config:
  backend:
    opentelemetry:
      collector_url: grpcs://example.com:4317
      service_name: ydb
```

Asynchronous Replication

Asynchronous replication synchronizes data between two YDB databases, where one serves as a client to the other. Whether this communication uses TLS-encrypted connections is controlled by the `CONNECTION_STRING` setting of CREATE ASYNC REPLICATION queries. Use the `grpcs://` protocol for TLS connections. No changes to the server-side configuration are required.

When using a custom Certificate Authority (CA), pass its certificate in the `CA_CERT` parameter when creating an instance of asynchronous replication.

table_service_config

The `table_service_config` section contains configuration parameters for the table service, including spilling settings.

resource_manager

The `resource_manager` subsection specifies resource management parameters for the table service.

Vector index level table cache

The vector index level table (`indexImplLevelTable`) `vector_kmeans_tree` stores centroids that YDB accesses at each step of descending the cluster tree during vector search. To avoid re-reading these centroids from the distributed storage on every query, YDB can cache them in memory on each database node.

The cache is created separately on each node, uses LRU eviction, and gradually grows its size up to the specified limit.

```
table_service_config:
  resource_manager:
    kqp_level_cache_max_size_bytes: 0
    kqp_level_cache_increase_batch_size_bytes: 33554432
```

resource_manager.kqp_level_cache_max_size_bytes

Type: `uint64`
Default: `0` (cache disabled)
Description: Maximum size of the vector index level table cache in bytes per database node. Setting a non-zero value enables the cache.

resource_manager.kqp_level_cache_increase_batch_size_bytes

Type: `uint64`
Default: `33554432` (32 MiB)
Description: Step size for increasing the level table cache. The cache grows in chunks of this size up to the value of `kqp_level_cache_max_size_bytes` and decreases by the same chunks when the limit is lowered.

Query execution memory limits

The `resource_manager` subsection sets the spilling threshold relative to the query memory pool on the node. The size of this pool is controlled by the `query_execution_limit_percent` / `query_execution_limit_bytes` parameters in `memory_controller_config`.

```
table_service_config:
  resource_manager:
    spilling_percent: 80
```

resource_manager.spilling_percent

Type: `double`
Default: `80`
Description: The fill threshold of the query memory pool at which YDB starts considering spilling the preferred memory management method. The basis for calculation is the Query Processor pool, whose size is set by `query_execution_limit_percent` / `query_execution_limit_bytes`.

When the total memory consumption by queries on a node exceeds `spilling_percent` percent of the available pool, computational operations that support spilling (Grace Hash Join, aggregations, etc.) receive a signal to offload intermediate data to disk instead of further increasing RAM consumption.

For example, with the default value of `80`, spilling is activated when the query pool is about 80% full.

The threshold applies:

- To the total query memory pool on the node (size — see `query_execution_limit_percent` / `query_execution_limit_bytes`).
- To the resource pool if the query is executed in a workload pool with the `total_memory_limit_percent_per_node` limit.

> **Note**
>
> `spilling_percent` does not set a limit on the size of spilling files on disk. Disk quotas are handled by `local_file_config.max_total_size` in the `spilling_service_config` section.

Interaction with other parameters

Parameter	Level	Role
`query_execution_limit_percent` / `query_execution_limit_bytes`	Node (QP)	Query memory pool size
`spilling_percent`	Query / pool	Pool fill threshold after which spilling is preferred
`activities_limit_percent`	Node	Total memory limit for all activity components (QP, compaction, etc.)

`spilling_percent` determines when the query pool on a node is so full that further growth in RAM should give way to spilling. `activities_limit_percent` limits the memory of activities overall and indirectly affects the available RAM, but does not replace the pool relative to which `spilling_percent` is calculated.

Recommendations

- Decrease `spilling_percent` (for example, to `70`) if you need to offload heavy queries to disk earlier and reduce the risk of exhausting the memory pool.
- Increase `spilling_percent` (for example, to `90`) if disk spilling too often degrades performance and the node has enough RAM.
- Coordinate `spilling_percent` with `query_execution_limit_percent` / `query_execution_limit_bytes`: when increasing the query pool limit, you can raise the spilling threshold if the node has enough RAM.

spilling_service_config

Spilling is a memory management mechanism in YDB that temporarily saves data to disk when RAM is insufficient.

Enabling

Spilling is enabled by default. The following parameter controls enabling and disabling the spilling service.

local_file_config.enable

Location: `table_service_config.spilling_service_config.local_file_config.enable`
Type: `boolean`
Default: `true`
Description: Enables or disables the spilling service. When disabled (`false`), spilling does not function, which may lead to errors when processing large amounts of data.

Possible errors

- `Spilling Service not started` / `Service not started` — attempt to use spilling when the Spilling Service is disabled. See Spilling Service Not Started

```
table_service_config:
  spilling_service_config:
    local_file_config:
      enable: true
```

Main configuration parameters

```
table_service_config:
  spilling_service_config:
    local_file_config:
      root: ""
      max_total_size: 21474836480
```

Directory configuration

local_file_config.root

Type: `string` \n**Default:** `""` (temporary directory) \n**Description:** File directory for storing spilling files.

For each `ydbd` process, a separate directory with a unique name is created. Spilling directories have the following name format:

`node_<node_id>_<spilling_service_id>`

Where:

- `node_id`: node identifier
- `spilling_service_id`: unique instance identifier created once during Spilling Service initialization when the ydbd process starts

Spilling files are stored inside each such directory.

Example of a full path to a spilling directory:

```
/tmp/spilling-tmp-<username>/node_1_32860791-037c-42b4-b201-82a0a337ac80
```

Where:

- `/tmp`: value of the `root` parameter
- `<username>`: username under which the `ydbd` process runs

Important notes:

- When the process starts, all existing spilling directories in the specified directory are automatically deleted. Spilling directories have a special name format that includes an instance identifier generated once when the ydbd process starts. When a new process starts, all directories in the spilling directory that match the name format but have a different `spilling_service_id` from the current one are deleted.
- The directory must have sufficient read and write permissions for the user under which ydbd runs.

> **Note**
>
> Spilling is performed only on database nodes.

Possible errors

- `Permission denied`: insufficient directory access permissions. See Permission Denied

local_file_config.max_total_size

Type: `uint64` \n**Default:** `21474836480` (20 GiB) \n**Description:** Maximum total size of all spilling files on each node. If the limit is exceeded, spilling operations fail with an error. The total spilling limit across the entire cluster equals the sum of `max_total_size` values from all nodes.

Recommendations

- Set the value based on available disk space

Possible errors

- `Total size limit exceeded: X/YMb` : maximum total size of spilling files exceeded. See Total Size Limit Exceeded

Memory management

Relationship with memory_controller_config

Spilling activation is closely related to the memory controller settings. Detailed configuration of `memory_controller_config` is described in a separate article.

The key parameter for spilling is **`activities_limit_percent`**, which determines the amount of memory allocated for query processing activities. This parameter affects the memory available for user queries and, consequently, the frequency of spilling activation.

Spilling threshold in resource_manager

The actual threshold at which computational operations switch to spilling is controlled by the `resource_manager.spilling_percent` parameter. It determines at what fill level of the Query Processor memory pool the intermediate query data starts being offloaded to disk. The pool size is set in `memory_controller_config` . For more details, see the Query execution memory limits section.

Impact on spilling

- When `activities_limit_percent` increases, more memory is available for queries → spilling is activated less often
- When `activities_limit_percent` decreases, less memory is available for queries → spilling is activated more often
- When `spilling_percent` decreases, spilling is triggered at a lower fill level of the query pool
- When `spilling_percent` increases, tasks take longer to increase RAM consumption before switching to spilling

Warning

It is important to note that spilling itself also requires memory. If `activities_limit_percent` is set too high, memory may still run out despite spilling, because the spilling mechanism itself consumes memory resources.

File system requirements

File descriptors

Note

For information on configuring file descriptor limits during initial deployment, see the File descriptor limits section.

Configuration examples

High-load system

For maximum performance in high-load systems, it is recommended to increase the spilling size:

```
table_service_config:
  spilling_service_config:
    local_file_config:
      root: ""
      max_total_size: 107374182400   # 100 GiB
```

Limited resources

For systems with limited resources, it is recommended to use conservative settings:

```
table_service_config:
  spilling_service_config:
    local_file_config:
      root: ""
      max_total_size: 5368709120     # 5 GiB
```

Full example

```
table_service_config:
  spilling_service_config:
    local_file_config:
      enable: true
      root: "/var/spilling"
      max_total_size: 53687091200    # 50 GiB
```

See also

- Spilling concept
- Spilling Service architecture

- Spilling troubleshooting
- Memory controller configuration
- Monitoring YDB
- Performance diagnostics

tli_config

The `tli_config` section contains configuration parameters for transaction lock invalidation (TLI) diagnostics.

TLI is a situation where one transaction (the breaker) breaks the optimistic locks of another transaction (the victim), forcing the victim to roll back and retry. For more information about TLI diagnostics, see Transaction lock invalidation.

Configuration parameters

Parameter	Type	Default	Description
`ignored_table_regexes`	repeated string	`[]`	List of table path regular expressions excluded from TLI diagnostics

ignored_table_regexes

Allows excluding specific tables from TLI logging and statistics. If all tables involved in a lock conflict match at least one of the specified regular expressions, no TLI log entry is generated for that conflict.

Changes take effect for new sessions without restarting nodes.

Typical use cases:

- reducing log volume for system or auxiliary tables where conflicts are expected;
- excluding high-conflict queue tables that do not require diagnostics.

Regular expressions are applied to the full table path, e.g. `/Root/mydb/mytable`. The syntax follows ECMAScript regex.

Configuration example

```
tli_config:
  ignored_table_regexes:
    - "/Root/.*/queue_.*"
    - "/Root/system/.*"
```

In this example, the following are excluded from TLI diagnostics:

- tables whose name starts with `queue_` in any database;
- all tables in the `/Root/system/` directory.

See also

- Transaction lock invalidation
- log_config
- Logging in YDB

hive_config

Hive is a YDB component responsible for launching tablets. In various situations and under different load patterns, you might need to configure its behavior. Hive behavior is configured in the `hive_config` section of the YDB cluster configuration. Some configuration options are also available for editing through the Hive web-viewer interface. Settings configured through the interface take priority over those specified in the configuration. Below are all available options, with the corresponding option name in the interface indicated if the option can be edited through the interface.

Tablet Boot Options

These options allow you to control the speed at which tablets are booted and how nodes are selected for them.

Configuration Parameter Name	Hive Web-viewer Parameter Name	Format	Description	Default Value
`max_tablets_scheduled`	MaxTabletsScheduled	Integer	Maximum number of tablets simultaneously in the startup process on a single node.	100
`max_boot_batch_size`	MaxBootBatchSize	Integer	Maximum number of tablets from the Hive boot queue processed at once.	1000
`node_select_strategy`	NodeSelectStrategy	Enumeration	Node selection strategy for tablet startup. Possible values: • `HIVE_NODE_SELECT_STRATEGY_WEIGHTED_RANDOM` — weighted random selection based on consumption; • `HIVE_NODE_SELECT_STRATEGY_EXACT_MIN` — select node with minimum consumption; • `HIVE_NODE_SELECT_STRATEGY_RANDOM_MIN_7P` — select random node among 7% of nodes with lowest consumption; • `HIVE_NODE_SELECT_STRATEGY_RANDOM` — select random node.	`HIVE_NODE_SELECT_STRATEGY_RANDOM_MI`
`boot_strategy`	—	Enumeration	Controls behavior when starting large numbers of tablets. Possible values: • `HIVE_BOOT_STRATEGY_BALANCED` — when the `max_tablets_scheduled` limit is reached on one node, stops starting new tablets on all nodes. • `HIVE_BOOT_STRATEGY_FAST` — when the `max_tablets_scheduled` limit is reached on one node, continues starting tablets on other nodes. If one node starts tablets slightly slower than others when starting a large number of tablets, then using `HIVE_BOOT_STRATEGY_FAST` will result in fewer tablets being started on that node than on the others. Using `HIVE_BOOT_STRATEGY_BALANCED` in the same situation will distribute tablets evenly across nodes, but their startup will take longer.	`HIVE_BOOT_STRATEGY_BALANCED`
`default_tablet_limit`	—	Nested section	Limits on starting tablets of various types on a single node. Specified as a list format where each element has `type` and `max_count` fields.	Empty section

`default_tablet_preference`	—	Nested section	Priorities for selecting data centers for starting tablets of various types. For each tablet type, you can specify multiple data center groups. Data centers within the same group will have equal priority, with earlier groups having priority over subsequent ones. Example format:
			`default_tablet_preference:`
			`- type: Coordinator`
			`data_centers_preference:`
			`- data_centers_group:`
			`- "dc-1"`
			`- "dc-2"`
			`- data_centers_group:`
			`- "dc-3"`
				Empty section

`system_category_id`	—	Integer	When specifying any number other than 0, all coordinators and mediators are launched in the same data center whenever possible.	1

Example

> **Note**
>
> In the `default_tablet_limit` and `default_tablet_preference` subsections, you need to specify tablet types. Exact tablet type names are specified in the glossary.

```
hive_config:
  max_tablets_scheduled: 10
  node_select_strategy: HIVE_NODE_SELECT_STRATEGY_RANDOM
  boot_strategy: HIVE_BOOT_STRATEGY_FAST
```

```
default_tablet_limit:
  - type: PersQueue
    max_count: 15
  - type: DataShard
    max_count: 100
default_tablet_preference:
  - type: Coordinator
    data_centers_preference:
      - data_centers_group:
        - "dc-1"
        - "dc-2"
      - data_centers_group:
        - "dc-3"
```

Auto-Balancing Options

These options control the auto-balancing process: in which situations it starts, how many tablets it moves at what intervals, how it selects nodes and tablets. Some options are presented in two variations: for "emergency balancing," i.e., balancing when one or more nodes are overloaded, and for all other types of balancing.

Configuration Parameter Name	Hive Web-viewer Parameter Name	Format	Description	Default V
`min_scatter_to_balance`	MinScatterToBalance	Real number	Threshold for Scatter metric for CPU, Memory, Network resources. Has lower priority than the parameters below.	0.5
`min_cpuscatter_to_balance`	MinCPUScatterToBalance	Real number	Threshold for Scatter metric for CPU resource.	0.5
`min_memory_scatter_to_balance`	MinMemoryScatterToBalance	Real number	Threshold for Scatter metric for Memory resource.	0.5
`min_network_scatter_to_balance`	MinNetworkScatterToBalance	Real number	Threshold for Scatter metric for Network resource.	0.5
`min_counter_scatter_to_balance`	MinCounterScatterToBalance	Real number	Threshold for Scatter metric for virtual Counter resource.	0.02
`min_node_usage_to_balance`	MinNodeUsageToBalance	Real number	Resource consumption on a node below this value is equated to this value. Used to avoid balancing tablets between lightly loaded nodes.	0.1
`max_node_usage_to_kick`	MaxNodeUsageToKick	Real number	Resource consumption threshold on a node for triggering emergency auto-balancing.	0.9
`node_usage_range_to_kick`	NodeUsageRangeToKick	Real number	Minimum difference in resource consumption level between nodes, below which auto-balancing is considered inappropriate.	0.2
`resource_change_reaction_period`	ResourceChangeReactionPeriod	Integer seconds	Frequency of updating aggregated resource consumption statistics.	10
`tablet_kick_cooldown_period`	TabletKickCooldownPeriod	Integer seconds	Minimum time period between movements of a single tablet.	600
`spread_neighbours`	SpreadNeighbours	true/false	Start tablets of the same schema object (table, topic) on different nodes when possible.	true
`node_balance_strategy`	NodeBalanceStrategy	Enumeration	Strategy for selecting the node from which tablets are moved during auto-balancing. Possible values: • `HIVE_NODE_BALANCE_STRATEGY_WEIGHTED_RANDOM` — weighted random selection based on consumption; • `HIVE_NODE_BALANCE_STRATEGY_HEAVIEST` — select node with maximum consumption; • `HIVE_NODE_BALANCE_STRATEGY_RANDOM` — select random node.	`HIVE_NO`
`tablet_balance_strategy`	TabletBalanceStrategy	Enumeration	Strategy for selecting tablet to move during auto-balancing. Possible values: • `HIVE_TABLET_BALANCE_STRATEGY_WEIGHTED_RANDOM` — weighted random selection based on consumption; • `HIVE_TABLET_BALANCE_STRATEGY_HEAVIEST` — select tablet with maximum consumption; • `HIVE_TABLET_BALANCE_STRATEGY_RANDOM` — select random tablet.	`HIVE_TA`
`min_period_between_balance`	MinPeriodBetweenBalance	Real number seconds	Minimum time period between two auto-balancing iterations. Does not apply to emergency balancing.	0.2
`balancer_inflight`	BalancerInflight	Integer	Number of tablets simultaneously restarting during auto-balancing process. Does not apply to emergency balancing.	1
`max_movements_on_auto_balancer`	MaxMovementsOnAutoBalancer	Integer	Number of tablet movements per auto-balancing iteration. Does not apply to emergency balancing.	1

`continue_auto_balancer`	ContinueAutoBalancer	true/false	When enabled, the next balancing iteration starts without waiting for the end of `resource_change_reaction_period`.	true
`min_period_between_emergency_balance`	MinPeriodBetweenEmergencyBalance	Real number seconds	Similar to `min_period_between_balance`, but for emergency balancing.	0.1
`emergency_balancer_inflight`	EmergencyBalancerInfligh	Integer	Similar to `balancer_inflight`, but for emergency balancing.	1
`max_movements_on_emergency_balancer`	MaxMovementsOnEmergencyBalancer	Integer	Similar to `max_movements_on_auto_balancer`, but for emergency balancing.	2
`continue_emergency_balancer`	ContinueEmergencyBalancer	true/false	Similar to `continue_auto_balancer`, but for emergency balancing.	true
`check_move_expediency`	CheckMoveExpediency	true/false	Check the expediency of tablet movements. If auto-balancing leads to increased Hive CPU resource consumption, you can disable this option.	true
`object_imbalance_to_balance`	ObjectImbalanceToBalance	Real number	Threshold for single object tablet imbalance metric.	0.02
`less_system_tablets_moves`	LessSystemTabletMoves	true/false	Minimize movement of system tablets during auto-balancing.	true
`balancer_ignore_tablet_types`	BalancerIgnoreTabletTypes	List of tablet types. When set through Hive UI — separated by semicolon.	Tablet types that are not subject to auto-balancing.	Empty lis

Examples

With this configuration file, you can completely disable all types of tablet auto-balancing between nodes.

```
hive_config:
  min_cpuscatter_to_balance: 1.0
  min_memory_scatter_to_balance: 1.0
  min_network_scatter_to_balance: 1.0
  min_counter_scatter_to_balance: 1.0
  max_node_usage_to_kick: 3.0
  object_imbalance_to_balance: 1.0
```

With this configuration file, you can disable all types of auto-balancing between nodes for tablets participating in transaction distribution, i.e., coordinators and mediators. Exact tablet type names are specified in the glossary.

```
hive_config:
  balancer_ignore_tablet_types:
    - Coordinator
    - Mediator
```

When using Hive UI for the same effect, you need to enter `Coordinator;Mediator` in the input field for the BalancerIgnoreTabletTypes setting.

Computational Resource Consumption Metrics Collection Options

Hive collects computational resource consumption metrics from each node — CPU time, memory, network — both overall per node and broken down by tablets. These settings allow you to control the collection of these metrics, their normalization and aggregation.

Configuration Parameter Name	Hive Web-viewer Parameter Name	Format	Description	Default Value
`max_resource_cpu`	MaxResourceCPU	Integer microseconds	Maximum CPU consumption per node per second. Default value, used only if the node does not provide a value when registering with Hive.	10000000
`max_resource_memory`	MaxResourceMemory	Integer bytes	Maximum memory consumption per node. Default value, used only if the node does not provide a value when registering with Hive.	512000000000

`max_resource_network`	MaxResourceNetwork	Integer bytes/second	Maximum bandwidth consumption per node. Default value, used only if the node does not provide a value when registering with Hive.	1000000000
`max_resource_counter`	MaxResourceCounter	Integer	Maximum consumption of virtual Counter resource per node.	100000000
`metrics_window_size`	MetricsWindowSize	Integer milliseconds	Size of the window over which tablet resource consumption metrics are aggregated.	60000
`resource_overcommitment`	ResourceOvercommitment	Real number	Overcommitment factor for node resources.	3.0
`pools_to_monitor_for_usage`	—	Pool names separated by comma	Actor system pools whose consumption is taken into account when calculating node resource consumption.	System,User,IC

Storage Channel Distribution Options

Listed here are options related to distributing tablet channels across storage groups: taking into account various metrics, selecting groups, and the channel auto-balancing process across groups.

> **Note**
>
> This table contains advanced settings that in most cases do not require modification.

Configuration Parameter Name	Hive Web-viewer Parameter Name	Format	Description	Default Value
`default_unit_iops`	DefaultUnitIOPS	Integer	Default value for IOPS of one channel.	1
`default_unit_throughput`	DefaultUnitThroughput	Integer bytes/second	Default value for throughput consumption by one channel.	1000
`default_unit_size`	DefaultUnitSize	Integer bytes	Default value for disk space consumption by one channel.	100000000
`storage_overcommit`	StorageOvercommit	Real number	Overcommitment factor for storage group resources.	1.0
`storage_balance_strategy`	StorageBalanceStrategy	Enumeration	Selection of parameter used for distributing tablet channels across storage groups. Possible values: • `HIVE_STORAGE_BALANCE_STRATEGY_IOPS` — only IOPS is considered; • `HIVE_STORAGE_BALANCE_STRATEGY_THROUGHPUT` — only throughput consumption is considered; • `HIVE_STORAGE_BALANCE_STRATEGY_SIZE` — only occupied space volume is considered; • `HIVE_STORAGE_BALANCE_STRATEGY_AUTO` — the parameter with maximum consumption among the above is considered;	`HIVE_STORAGE_BALANC`
`storage_safe_mode`	StorageSafeMode	true/false	Check for exceeding maximum resource consumption of storage groups.	true

`storage_select_strategy`	StorageSelectStrategy	Enumeration	Strategy for selecting storage group for tablet channel. Possible values: • `HIVE_STORAGE_SELECT_STRATEGY_WEIGHTED_RANDOM` — weighted random selection based on consumption; • `HIVE_STORAGE_SELECT_STRATEGY_EXACT_MIN` — select group with minimum consumption; • `HIVE_STORAGE_SELECT_STRATEGY_RANDOM_MIN_7P` — select random group among 7% of groups with lowest consumption; • `HIVE_STORAGE_SELECT_STRATEGY_RANDOM` — select random group; • `HIVE_STORAGE_SELECT_STRATEGY_ROUND_ROBIN` — select group within storage pool using Round-robin principle.	`HIVE_STORAGE_SELECT`
`min_period_between_reassign`	MinPeriodBetweenReassign	Integer seconds	Minimum time period between storage group reassignments for channels of one tablet.	300
`storage_pool_fresh_period`	StoragePoolFreshPeriod	Integer milliseconds	Frequency of updating storage pool information.	60000
`space_usage_penalty_threshold`	SpaceUsagePenaltyThreshold	Real number	Minimum ratio of free space in target group to free space in source group, at which the target group will be penalized by applying a multiplicative penalty to the weight when moving a channel.	1.1
`space_usage_penalty`	SpaceUsagePenalty	Real number	Penalty factor for the penalization described above.	0.2
`channel_balance_strategy`	ChannelBalanceStrategy	Enumeration	Strategy for selecting channel for reassignment during channel balancing. Possible values: • `HIVE_CHANNEL_BALANCE_STRATEGY_WEIGHTED_RANDOM` — weighted random selection based on consumption; • `HIVE_CHANNEL_BALANCE_STRATEGY_HEAVIEST` — select channel with maximum consumption; • `HIVE_CHANNEL_BALANCE_STRATEGY_RANDOM` — select random channel.	`HIVE_CHANNEL_BALANC`
`max_channel_history_size`	MaxChannelHistorySize	Integer	Maximum size of channel history.	200
`storage_info_refresh_frequency`	StorageInfoRefreshFrequency	Integer milliseconds	Frequency of updating storage pool information.	600000
`min_storage_scatter_to_balance`	MinStorageScatterToBalance	Real number	Threshold for Scatter metric for storage groups.	999
`min_group_usage_to_balance`	MinGroupUsageToBalance	Real number	Storage group resource consumption threshold below which balancing is not started.	0.1
`storage_balancer_inflight`	StorageBalancerInflight	Integer	Number of tablets simultaneously restarting during channel balancing.	1

Restart Tracking Options

Hive tracks how often various nodes and tablets restart to identify problematic ones. Using these options, you can configure which tablets or nodes will be considered problematic and how this will affect them. Based on this statistics, nodes and tablets are included in the HealthCheck API report.

Tablet Restart Tracking Options

Configuration Parameter Name	Hive Web-viewer Parameter Name	Format	Description	Default Value
`tablet_restart_watch_period`	—	Integer seconds	Size of window over which statistics on tablet restart count are collected. **This period is used only for statistics passed to HealthCheck.**	3600
`tablet_restarts_period`	—	Integer milliseconds	Size of window over which tablet restart count is calculated for penalizing problematic tablet startup.	1000
`tablet_restarts_max_count`	—	Integer	Number of restarts in the `tablet_restarts_period` window, exceeding which triggers penalization.	2
`postopone_start_period`	—	Integer milliseconds	Frequency of startup attempts for problematic tablets.	1000

Node Restart Tracking Options

Configuration Parameter Name	Hive Web-viewer Parameter Name	Format	Description	Default Value
`node_restart_watch_period`	—	Integer seconds	Size of window over which statistics on node restart count are collected.	3600

`node_restarts_for_penalty`	NodeRestartsForPenalty	Integer	Number of restarts in the `node_restart_watch_period` window after which nodes receive priority reduction.	3

Miscellaneous

Listed here are additional Hive settings.

> **Note**
>
> This table contains advanced settings that in most cases do not require modification.

Configuration Parameter Name	Hive Web-viewer Parameter Name	Format	Description	Default Value
`drain_inflight`	DrainInflight	Integer	Number of tablets simultaneously restarting during graceful movement of all tablets from one node (drain).	10
`request_sequence_size`	—	Integer	Number of tablet identifiers that database Hive requests from root Hive at once.	1000
`min_request_sequence_size`	—	Integer	Minimum number of tablet identifiers that root Hive allocates for database Hive at once.	1000
`max_request_sequence_size`	—	Integer	Maximum number of tablet identifiers allocated for database Hive at once.	1000000
`node_delete_period`	—	Integer seconds	Inactivity period after which a node is deleted from the Hive database.	3600
`warm_up_enabled`	WarmUpEnabled	true/false	When this option is enabled, database Hive waits for all nodes to connect before starting tablets during startup. When disabled, all tablets can be started on the first connected node.	true
`warm_up_boot_waiting_period`	MaxWarmUpBootWaitingPeriod	Integer milliseconds	Waiting time for all known nodes to start during database startup.	30000
`max_warm_up_period`	MaxWarmUpPeriod	Integer seconds	Maximum waiting time for node startup during database startup.	600
`enable_destroy_operations`	—	true/false	Whether destructive manual operations are allowed.	false

`max_pings_in_flight`	—	Integer	Maximum number of connections being established with nodes in parallel.	1000
`cut_history_deny_list`	—	List of tablet types separated by comma	List of tablet types for which history cleanup operation is ignored.	ColumnShard,KeyValue,PersQueue,BlobDepo
`cut_history_allow_list`	—	List of tablet types separated by comma	List of tablet types for which history cleanup operation is allowed.	DataShard
`scale_recommendation_refresh_frequency`	ScaleRecommendationRefreshFrequency	Integer milliseconds	How often the recommendation for the number of compute nodes is updated.	60000
`scale_out_window_size`	ScaleOutWindowSize	Integer	Number of buckets based on which the decision to recommend increasing the number of compute nodes is made.	15
`scale_in_window_size`	ScaleInWindowSize	Integer	Number of buckets based on which the decision to recommend decreasing the number of compute nodes is made.	5
`target_tracking_cpumargin`	TargetTrackingCPUMargin	Real number	Allowable deviation from target CPU utilization value during autoscaling.	0.1
`dry_run_target_tracking_cpu`	DryRunTargetTrackingCPU	Real number	Target CPU utilization value for testing how autoscaling would work.	0

kafka_proxy_config

The `kafka_proxy_config` section of the YDB configuration file enables and configures Kafka Proxy, which provides access to work with YDB Topics via Kafka API.

Description of parameters

Parameter	Type	Default value	Description
`enable_kafka_proxy`	bool	`false`	Enables or disables Kafka Proxy.
`listening_port`	int32	`9092`	The port on which the Kafka API will be available.
`transaction_timeout_ms`	uint32	`300000` (5 minutes)	The maximum timeout for Kafka transactions, after which the transaction will be cancelled.
`auto_create_topics_enable`	bool	`false`	Enables automatic creation of topics when they are accessed. Analogous to the same option in Apache Kafka.
`auto_create_consumers_enable`	bool	`true`	Enables automatic registration of consumers when they are accessed.
`topic_creation_default_partitions`	uint32	`1`	The number of partitions that will be created if the number of partitions is not specified when adding a topic via the Kafka protocol. Analogous to num.partitions option in Apache Kafka.
`ssl_cerificate`	string	—	The path to the SSL certificate file, which includes both the certificate file and the key file. When this parameter is specified, Kafka Proxy automatically starts processing requests using the specified SSL certificate.
`cert`	string	—	The path to the SSL certificate file. When this parameter is specified, Kafka Proxy automatically starts processing requests using the specified SSL certificate.
`key`	string	—	The path to the SSL key file.

Example of a completed config

```yaml
kafka_proxy_config:
  enable_kafka_proxy: true
  listening_port: 9092
  transaction_timeout_ms: 300000 # 5 minutes
  auto_create_topics_enable: true
  auto_create_consumers_enable: true
  topic_creation_default_partitions: 1
  cert: /path/to/cert.pem
  key: /path/to/key.pem
```

Reference on YDB observability

This section of YDB documentation covers various observability-related topics that do **not** depend on a chosen deployment method.

- Metrics reference
- Grafana dashboards for YDB
- Tracing in YDB
- Passing external trace-id to YDB

Metrics reference

This document provides a reference for the main YDB system metrics used for monitoring cluster state, diagnosing performance, and analyzing load.

How to view current metric values in the built-in web interface and configure their collection in Prometheus and Grafana is described in the Setting up YDB cluster monitoring section.

Resource usage metrics

Metric name Type, units	Description Labels
`resources.storage.used_bytes` `IGAUGE`, bytes	Size of user and service data stored in the distributed network storage. `resources.storage.used_bytes` = `resources.storage.table.used_bytes` + `resources.storage.topic.used_bytes`.
`resources.storage.table.used_bytes` `IGAUGE`, bytes	Size of user and service data stored by tables in the distributed network storage. Service data includes data of primary, secondary indexes, vector indexes, full-text indexes, local Bloom indexes, and local min_max index.
`resources.storage.topic.used_bytes` `IGAUGE`, bytes	Size of the distributed network storage used by topics. Equals the sum of `topic.storage_bytes` values of all topics.
`resources.storage.limit_bytes` `IGAUGE`, bytes	Limit on the size of user and system data that the database can store in the distributed network storage.

Common gRPC API metrics

Metric name Type, units	Description Labels
`api.grpc.request.bytes` `RATE`, bytes	Size of requests received by the database in a certain time period. Labels: - *api_service* – name of the gRPC API service, for example `table` or `data_streams`. - *method* – name of the gRPC API service method, for example `ExecuteDataQuery` (for the `table` service), or `PutRecord`, `GetRecords` (for the `data_streams` service).
`api.grpc.request.dropped_count` `RATE`, count	Number of requests whose processing was terminated at the transport (gRPC) level due to an error. Labels: - *api_service* – name of the gRPC API service, for example `table`. - *method* – name of the gRPC API service method, for example `ExecuteDataQuery`.
`api.grpc.request.inflight_count` `IGAUGE`, count	Number of requests being processed simultaneously by the database in a certain time period. Labels: - *api_service* – name of the gRPC API service, for example `table`. - *method* – name of the gRPC API service method, for example `ExecuteDataQuery`.
`api.grpc.request.inflight_bytes` `IGAUGE`, bytes	Size of requests being processed simultaneously by the database in a certain time period. Labels: - *api_service* – name of the gRPC API service, for example `table`. - *method* – name of the gRPC API service method, for example `ExecuteDataQuery`.
`api.grpc.response.bytes` `RATE`, bytes	Size of responses sent by the database in a certain time period. Labels: - *api_service* – name of the gRPC API service, for example `table`. - *method* – name of the gRPC API service method, for example `ExecuteDataQuery`.
`api.grpc.response.count` `RATE`, count	Number of responses sent by the database in a certain time period. Labels: - *api_service* – name of the gRPC API service, for example `table`. - *method* – name of the gRPC API service method, for example `ExecuteDataQuery`. - *status* – request execution status; for more details on statuses, see the Error handling section.
`api.grpc.response.dropped_count` `RATE`, count	Number of responses whose sending was terminated at the transport (gRPC) level due to an error. Labels: - *api_service* – name of the gRPC API service, for example `table`. - *method* – name of the gRPC API service method, for example `ExecuteDataQuery`.
`api.grpc.response.issues` `RATE`, count	Number of errors of a certain type that occurred during query execution in a certain time period. Labels: - *issue_type* – error type, the only value is `optimistic_locks_invalidation`; for more details on lock invalidation, see the Transactions and queries to YDB section.

gRPC API metrics for topics

Metric name Type, units	Description Labels
`grpc.topic.stream_read.commits` `RATE`, units	Number of commits of the `Ydb::TopicService::StreamRead` method. Labels: - *topic* – topic name. - *consumer* – consumer name.

`grpc.topic.stream_read.bytes` `RATE`, units	Number of bytes read by the `Ydb::TopicService::StreamRead` method. Labels: - *topic* – topic name. - *consumer* – consumer name.
`grpc.topic.stream_read.messages` `RATE`, units	Number of messages read by the `Ydb::TopicService::StreamRead` method. Labels: - *topic* – topic name. - *consumer* – consumer name.
`grpc.topic.stream_read.partition_session.errors` `RATE`, units	Number of errors when working with a partition. Labels: - *topic* – topic name. - *consumer* – reader name.
`grpc.topic.stream_read.partition_session.started` `RATE`, count	Number of sessions started per unit of time. Labels: - *topic* – topic name. - *consumer* – reader name.
`grpc.topic.stream_read.partition_session.stopped` `RATE`, count	Number of sessions stopped per unit of time. Labels: - *topic* – topic name. - *consumer* – reader name.
`grpc.topic.stream_read.partition_session.starting_count` `RATE`, count	Number of sessions being started (that is, the client received a command to start a session, but the client has not started the session yet). Labels: - *topic* – topic name. - *consumer* – reader name.
`grpc.topic.stream_read.partition_session.stopping_count` `RATE`, count	Number of sessions being stopped. Labels: - *topic* – topic name. - *consumer* – reader name.
`grpc.topic.stream_read.partition_session.count` `RATE`, count	Number of partition_session. Labels: - *topic* – topic name. - *consumer* – reader name.
`grpc.topic.stream_write.bytes` `RATE`, bytes	Number of bytes written by the `Ydb::TopicService::StreamWrite` method. Labels: - *topic* – topic name.
`grpc.topic.stream_write.uncommitted_bytes` `RATE`, bytes	Number of bytes written by the `Ydb::TopicService::StreamWrite` method within uncommitted transactions. Labels: - *topic* – topic name.
`grpc.topic.stream_write.errors` `RATE`, count	Number of errors when calling the `Ydb::TopicService::StreamWrite` method. Labels: - *topic* – topic name.
`grpc.topic.stream_write.messages` `RATE`, count	Number of messages written by the `Ydb::TopicService::StreamWrite` method. Labels: - *topic* – topic name.
`grpc.topic.stream_write.uncommitted_messages` `RATE`, count	Number of messages written by the `Ydb::TopicService::StreamWrite` method within uncommitted transactions. Labels: - *topic* – topic name.
`grpc.topic.stream_write.partition_throttled_milliseconds` `HIST_RATE`, count	Histogram counter. Intervals are set in milliseconds. Shows the number of messages that waited on the quota. Labels: - *topic* – topic name.
`grpc.topic.stream_write.sessions_active_count` `GAUGE`, count	Number of open write sessions. Labels: - *topic* – topic name.
`grpc.topic.stream_write.sessions_created` `RATE`, count	Number of created write sessions. Labels: - *topic* – topic name.

HTTP API metrics

Metric name Type, units	Description Labels

`api.http.data_streams.request.count` `RATE`, pieces	Number of requests over HTTP. Labels: - *method* – HTTP API service method name, e.g., `PutRecord`, `GetRecords`. - *topic* – topic name.
`api.http.data_streams.request.bytes` `RATE`, bytes	Total size of requests over HTTP. Labels: - *method* – HTTP API service method name, in this case only `PutRecord`. - *topic* – topic name.
`api.http.data_streams.response.count` `RATE`, pieces	Number of responses over HTTP. Labels: - *method* – HTTP API service method name, e.g., `PutRecord`, `GetRecords`. - *topic* – topic name. - *code* – HTTP response code.
`api.http.data_streams.response.bytes` `RATE`, bytes	Total size of responses over HTTP. Labels: - *method* – HTTP API service method name, in this case only `GetRecords`. - *topic* – topic name.
`api.http.data_streams.response.duration_milliseconds` `HIST_RATE`, pieces	Histogram counter. Intervals are specified in milliseconds. Shows the number of responses whose execution time falls within a certain interval. Labels: - *method* – HTTP API service method name. - *topic* – topic name.
`api.http.data_streams.get_records.messages` `RATE`, pieces	Number of messages read by the `GetRecords` method. Labels: - *topic* – topic name.
`api.http.data_streams.put_record.messages` `RATE`, pieces	Number of messages sent by the `PutRecord` method. Labels: - *topic* – topic name.
`api.http.data_streams.put_records.failed_messages` `RATE`, pieces	Number of messages sent by the `PutRecords` method that were not written. Labels: - *topic* – topic name.
`api.http.data_streams.put_records.successful_messages` `RATE`, pieces	Number of messages sent by the `PutRecords` method that were successfully written. Labels: - *topic* – topic name.
`api.http.data_streams.put_records.total_messages` `RATE`, pieces	Number of messages sent by the `PutRecords` method. Labels: - *topic* – topic name.

Kafka API metrics

Metric name Type, units	Description Labels
`api.kafka.request.count` `RATE`, pieces	Number of requests over the Kafka protocol per unit time. Labels: - *method* – Kafka API service method name, e.g., `PRODUCE`, `SASL_HANDSHAKE`.
`api.kafka.request.bytes` `RATE`, bytes	Total size of requests over the Kafka protocol per unit time. Labels: - *method* – Kafka API service method name, e.g., `PRODUCE`, `SASL_HANDSHAKE`.
`api.kafka.response.count` `RATE`, count	Number of responses via the Kafka protocol per unit time. Labels: - *method* – name of the Kafka API service method, e.g., `PRODUCE`, `SASL_HANDSHAKE`. - *error_code* – Kafka response code.
`api.kafka.response.bytes` `RATE`, bytes	Total size of responses via the Kafka protocol per unit time. Labels: - *method* – name of the Kafka API service method, e.g., `PRODUCE`, `SASL_HANDSHAKE`.
`api.kafka.response.duration_milliseconds` `HIST_RATE`, units	Histogram counter. Defines a set of intervals in milliseconds and for each of them shows the number of requests with execution time falling within that interval. Labels: - *method* – Kafka API service method name.
`api.kafka.produce.failed_messages` `RATE`, units	Number of messages per unit time sent by the `PRODUCE` method that were not written. Labels: - *topic* – topic name.
`api.kafka.produce.successful_messages` `RATE`, units	Number of messages per unit time sent by the `PRODUCE` method that were successfully written. Labels: - *topic* – topic name.

`api.kafka.produce.total_messages` `RATE`, units	Number of messages per unit time sent by the `PRODUCE` method. Labels: - *topic* – topic name.

Session metrics

Metric name Type, units	Description Labels
`table.session.active_count` `IGAUGE`, units	Number of sessions currently open by clients.
`table.session.closed_by_idle_count` `RATE`, units	Number of sessions closed by the database server in a certain period of time due to exceeding the time allocated for an inactive session's lifetime.

Transaction processing metrics

Transaction execution duration can be analyzed using a histogram counter. Intervals are set in milliseconds. The chart shows the number of transactions whose duration falls within a certain time interval.

Metric name Type, units	Description Labels
`table.transaction.total_duration_milliseconds` `HIST_RATE`, count	Number of transactions with a specific execution duration on the server and client. Execution duration is the time from the moment of explicit or implicit transaction opening to the moment of committing changes or rolling back. Includes transaction processing time on the server and time on the client between sending different queries within the same transaction. Labels: - *tx_kind* – transaction type, possible values `read_only`, `read_write`, `write_only`, `pure`.
`table.transaction.server_duration_milliseconds` `HIST_RATE`, count	Number of transactions with a specific execution duration on the server. Execution duration is the time of query execution within a transaction on the server. Does not include wait time on the client between sending individual queries within the same transaction. Labels: - *tx_kind* – transaction type, possible values `read_only`, `read_write`, `write_only`, `pure`.
`table.transaction.client_duration_milliseconds` `HIST_RATE`, count	Number of transactions with a specific execution duration on the client. Execution duration is the wait time on the client between sending individual queries within the same transaction. Does not include query execution time on the server. Labels: - *tx_kind* – transaction type, possible values `read_only`, `read_write`, `write_only`, `pure`.

Query processing metrics

Metric name Type, units	Description Labels
`table.query.request.bytes` `RATE`, bytes	Size of YQL query text and parameter values for queries that arrived at the database in a certain period of time.
`table.query.request.parameters_bytes` `RATE`, bytes	Size of parameters for queries that arrived at the database in a certain period of time.
`table.query.response.bytes` `RATE`, bytes	Size of responses sent by the database in a certain period of time.
`table.query.compilation.latency_milliseconds` `HIST_RATE`, units	Histogram counter. Intervals are set in milliseconds. Shows the number of successfully completed compilation requests whose duration falls within a certain time interval.
`table.query.compilation.active_count` `IGAUGE`, units	Number of compilations currently in progress.
`table.query.compilation.count` `RATE`, units	Number of compilations that completed successfully in a certain period of time.
`table.query.compilation.errors` `RATE`, units	Number of compilations that failed in a certain period of time.
`table.query.compilation.cache_hits` `RATE`, units	Number of queries in a certain period of time that did not require compiling the query because a previously created plan was in the prepared query cache.
`table.query.compilation.cache_misses` `RATE`, count	The number of queries in a given period of time that required compiling a query.
`table.query.execution.latency_milliseconds` `HIST_RATE`, count	Histogram counter. Intervals are specified in milliseconds. Shows the number of queries whose execution time falls within a certain interval.

Row table partition metrics

Metric name Type, units of measurement	Description Labels

`table.datashard.row_count` `GAUGE`, count	Number of rows in all row tables of the database.
`table.datashard.size_bytes` `GAUGE`, bytes	Size of data in all row tables of the database.
`table.datashard.used_core_percents` `HIST_GAUGE`, %	Histogram counter. Intervals are specified in percentages. Shows the number of partitions of row tables that use computing resources in a proportion that falls within a certain interval.
`table.datashard.read.rows` `RATE`, count	Number of rows read by all partitions of all row tables in the database during a given period of time.
`table.datashard.read.bytes` `RATE`, bytes	Size of data read by all partitions of all row tables in the database during a given period of time.
`table.datashard.write.rows` `RATE`, count	Number of rows written by all partitions of all row tables in the database during a given period of time.
`table.datashard.write.bytes` `RATE`, bytes	Size of data written by all partitions of all row tables in the database during a given period of time.
`table.datashard.scan.rows` `RATE`, count	Number of rows read via gRPC API calls `StreamExecuteScanQuery` or `StreamReadTable` by all partitions of all row tables in the database over a certain time period.
`table.datashard.scan.bytes` `RATE`, bytes	Size of data read via gRPC API call `StreamExecuteScanQuery` or `StreamReadTable` by all partitions of all row tables in the database over a certain time period.
`table.datashard.bulk_upsert.rows` `RATE`, count	Number of rows added via gRPC API call `BulkUpsert` to all partitions of all row tables in the database over a certain time period.
`table.datashard.bulk_upsert.bytes` `RATE`, bytes	Size of data added via gRPC API call `BulkUpsert` to all partitions of all row tables in the database over a certain time period.
`table.datashard.erase.rows` `RATE`, count	Number of rows deleted from all row tables in the database during a given period of time.
`table.datashard.erase.bytes` `RATE`, bytes	Size of data deleted from all row tables in the database during a given period of time.
`table.datashard.cache_hit.bytes` `RATE`, bytes	Total volume of row table data successfully retrieved from memory (cache). A larger volume of data retrieved from the cache indicates efficient cache usage without accessing the distributed storage.
`table.datashard.cache_miss.bytes` `RATE`, bytes	Total volume of row table data that was requested but not found in memory (cache) and was read from distributed storage. Indicates potential areas for cache optimization.

Column table partition metrics

Metric name Type, units of measurement	Description Labels
`table.columnshard.write.rows` `RATE`, count	Number of rows written by all partitions of all columnar tables in the database during a given period of time.
`table.columnshard.write.bytes` `RATE`, bytes	Size of data written by all partitions of all columnar tables in the database during a given period of time.
`table.columnshard.scan.rows` `RATE`, count	Number of rows read via gRPC API calls `StreamExecuteScanQuery` or `StreamReadTable` by all partitions of all columnar tables in the database over a certain time period.
`table.columnshard.scan.bytes` `RATE`, bytes	Size of data read via gRPC API call `StreamExecuteScanQuery` or `StreamReadTable` by all partitions of all columnar tables in the database over a certain time period.
`table.columnshard.bulk_upsert.rows` `RATE`, count	Number of rows added via gRPC API call `BulkUpsert` to all partitions of all columnar tables in the database over a certain time period.
`table.columnshard.bulk_upsert.bytes` `RATE`, bytes	Size of data added via gRPC API call `BulkUpsert` to all partitions of all columnar tables in the database over a certain time period.

Resource usage metrics (Dedicated mode only)

Metric name Type units of measurement	Description Labels
`resources.cpu.used_core_percents` `RATE`, %	CPU usage. A value of `100` means that one of the cores is 100% utilized. The value can be greater than `100` for configurations with more than 1 core. Labels: - *pool* – compute pool, possible values `user`, `system`, `batch`, `io`, `ic`.
`resources.cpu.limit_core_percents` `IGAUGE`, %	CPU available to the database as a percentage. For example, for a database of three nodes with 4 cores in `pool=user` on each node, the value of this sensor will be `1200`. Labels: - *pool* – compute pool, possible values `user`, `system`, `batch`, `io`, `ic`.
`resources.memory.used_bytes` `IGAUGE`, bytes	RAM used by database nodes.

`resources.memory.limit_bytes` `IGAUGE`, bytes	RAM available to database nodes.

Query processing metrics (Dedicated mode only)

Metric name Type units of measurement	Description Labels
`table.query.compilation.cache_evictions` `RATE`, count	Number of queries evicted from the prepared query cache during a given period of time.
`table.query.compilation.cache_size_bytes` `IGAUGE`, bytes	Size of the prepared query cache.
`table.query.compilation.cached_query_count` `IGAUGE`, count	Number of queries in the prepared query cache.

Topic metrics

Metric name Type, units of measurement	Description Labels
`topic.producers_count` `GAUGE`, count	Number of unique sources of the topic. Labels: - *topic* – topic name.
`topic.storage_bytes` `GAUGE`, bytes	Topic size in bytes. Labels: - *topic* – topic name.
`topic.read.bytes` `RATE`, bytes	Number of bytes read from the topic. Labels: - *topic* – topic name. - *consumer* – reader name.
`topic.read.messages` `RATE`, count	Number of messages read from the topic. Labels: - *topic* – topic name. - *consumer* – reader name.
`topic.read.lag_messages` `RATE`, count	Total number of messages not yet read by the given reader across the topic. Labels: - *topic* – topic name. - *consumer* – reader name.
`topic.read.lag_milliseconds` `HIST_RATE`, count	Histogram counter. Intervals are specified in milliseconds. Shows the number of messages for which the difference between the read time and the message creation time falls within a given interval. Labels: - *topic* – topic name. - *consumer* – reader name.
`topic.write.bytes` `RATE`, bytes	Size of written data. Labels: - *topic* – topic name.
`topic.write.uncommited_bytes` `RATE`, bytes	Size of data written as part of not yet completed transactions. Labels: - *topic* – topic name.
`topic.write.uncompressed_bytes` `RATE`, bytes	Size of decompressed written data. Labels: - *topic* – topic name.
`topic.write.messages` `RATE`, count	Number of written messages. Labels: - *topic* – topic name.
`topic.write.uncommitted_messages` `RATE`, count	Number of messages written as part of not yet completed transactions. Labels: - *topic* – topic name.
`topic.write.message_size_bytes` `HIST_RATE`, count	Histogram counter. Intervals are specified in bytes. Shows the number of messages whose size matches the interval boundaries. Labels: - *topic* – topic name.
`topic.write.lag_milliseconds` `HIST_RATE`, count	Histogram counter. Intervals are specified in milliseconds. Shows the number of messages for which the difference between the write time and the message creation time falls within a given interval. Labels: - *topic* – topic name.

Aggregated topic partition metrics

The following table lists aggregated partition metrics for a topic. Maximum and minimum values are calculated across all partitions of the specified topic.

Metric name Type, unit of measurement	Description Labels
`topic.partition.init_duration_milliseconds_max` `GAUGE`, milliseconds	Maximum partition initialization delay. Labels: - *topic* – topic name.
`topic.partition.producers_count_max` `GAUGE`, count	Maximum number of sources in a partition. Labels: - *topic* – topic name.
`topic.partition.storage_bytes_max` `GAUGE`, bytes	Maximum partition size in bytes. Labels: - *topic* – topic name.
`topic.partition.uptime_milliseconds_min` `GAUGE`, count	Minimum partition uptime after restart. Normally, during a rolling restart, `topic.partition.uptime_milliseconds_min` is close to 0; after the rolling restart finishes, the value of `topic.partition.uptime_milliseconds_min` should increase to infinity. Labels: - *topic* – topic name.
`topic.partition.total_count` `GAUGE`, count	Total number of partitions in the topic. Labels: - *topic* – topic name.
`topic.partition.alive_count` `GAUGE`, count	Number of partitions reporting their metrics. Labels: - *topic* – topic name.
`topic.partition.committed_end_to_end_lag_milliseconds_max` `GAUGE`, milliseconds	Maximum (across all partitions) difference between the current time and the creation time of the last committed message. Labels: - *topic* – topic name. - *consumer* – consumer name.
`topic.partition.committed_lag_messages_max` `GAUGE`, units	Maximum (across all partitions) difference between the last partition offset and the committed partition offset. Labels: - *topic* – topic name. - *consumer* – consumer name.
`topic.partition.committed_read_lag_milliseconds_max` `GAUGE`, milliseconds	Maximum (across all partitions) difference between the current time and the write time of the last committed message. Labels: - *topic* – topic name. - *consumer* – consumer name.
`topic.partition.end_to_end_lag_milliseconds_max` `GAUGE`, milliseconds	Difference between the current time and the minimum creation time among all messages read in the last minute across all partitions. Labels: - *topic* – topic name. - *consumer* – consumer name.
`topic.partition.lag_messages_max` `GAUGE`, units	Maximum difference (across all partitions) between the last offset in the partition and the last read offset. Labels: - *topic* – topic name. - *consumer* – consumer name.
`topic.partition.read.idle_milliseconds_max` `GAUGE`, milliseconds	Maximum idle time (how long the partition has not been read from) across all partitions. Labels: - *topic* – topic name. - *consumer* – consumer name.
`topic.partition.read.lag_milliseconds_max` `GAUGE`, milliseconds	Difference between the current time and the minimum write time among all messages read in the last minute across all partitions. Labels: - *topic* – topic name. - *consumer* – consumer name.
`topic.partition.write.lag_milliseconds_max` `GAUGE`, milliseconds	Maximum difference between the write time and the creation time among all messages written in the last minute. Labels: - *topic* – topic name.
`topic.partition.write.speed_limit_bytes_per_second` `GAUGE`, bytes per second	Write quota in bytes per second per partition. Labels: - *topic* – topic name.
`topic.partition.write.throttled_nanoseconds_max` `GAUGE`, nanoseconds	Maximum write throttling time (waiting on quota) across all partitions. In the limit, if `topic.partition.write.throttled_nanoseconds_max` = 10^9, it means waiting on quota for the entire second. Labels: - *topic* – topic name.

`topic.partition.write.bytes_per_day_max` `GAUGE`, bytes	Maximum number of bytes written in the last 24 hours across all partitions. Labels: - *topic* – topic name.
`topic.partition.write.bytes_per_hour_max` `GAUGE`, bytes	Maximum number of bytes written in the last hour across all partitions. Labels: - *topic* – topic name.
`topic.partition.write.bytes_per_minute_max` `GAUGE`, bytes	Maximum number of bytes written in the last minute across all partitions. Labels: - *topic* – topic name.
`topic.partition.write.idle_milliseconds_max` `GAUGE`, milliseconds	Maximum partition write idle time. Labels: - *topic* – topic name.

Resource pool metrics

Metric name Type, units	Description Labels
`kqp.workload_manager.CpuQuotaManager.AverageLoadPercentage` `RATE`, units	Average database load, `DATABASE_LOAD_CPU_THRESHOLD` operates based on this metric.
`kqp.workload_manager.InFlightLimit` `GAUGE`, units	Limit on the number of concurrently running queries.
`kqp.workload_manager.GlobalInFly` `GAUGE`, units	Current number of concurrently running queries. Displayed only for pools with `CONCURRENT_QUERY_LIMIT` or `DATABASE_LOAD_CPU_THRESHOLD` enabled.
`kqp.workload_manager.QueueSizeLimit` `GAUGE`, units	Size of the queue of queries waiting to be executed.
`kqp.workload_manager.GlobalDelayedRequests` `GAUGE`, units	Number of queries waiting in the execution queue. Displayed only for pools with `CONCURRENT_QUERY_LIMIT` or `DATABASE_LOAD_CPU_THRESHOLD` enabled.

See also

- Grafana dashboards for monitoring YDB metrics

Grafana dashboards for YDB

This page describes Grafana dashboards for YDB.

Instructions on how to install and configure dashboards are provided in the YDB cluster monitoring setup section.

DB status

General database dashboard.

Download the **DB status** dashboard template: dbstatus.json.

DB overview

General database dashboard by categories:

- Health
- API
- API details
- CPU
- CPU pools
- Memory
- Storage
- DataShard
- DataShard details
- Latency

Download the **DB overview** dashboard template: dboverview.json.

YDB Essential Metrics

Dashboard for monitoring key database metrics.

Health section

This section contains charts showing the status of cluster and database components.

Name	Description
Nodes count	Number of running YDB nodes, pcs.
Nodes uptime	Uptime of each node since startup; helps detect restarts and unstable nodes, in seconds.
VDisks count	Number of available VDisks in the cluster, pcs.

Saturation section

This section contains charts reflecting database resource utilization.

Name	Description
CPU by thread pool (dynnodes)	CPU consumption by dynamic nodes per execution pool, in CPU cores.
CPU utilization (dynnodes)	CPU utilization by dynamic nodes per execution pool, in %.
Elapsed Time vs CPU Time	Ratio of real operation execution time (`ElapsedMicrosec`) to CPU time (`CpuMicrosec`) by node. A sustained excess above 100% means sessions are spending time waiting rather than actively working: typically, this is I/O wait or CPU overcommit on the hypervisor side.
RSS size by node	Amount of RAM (Resident set size) consumed by each dynamic node, showing cgroup memory limits, in bytes.
Storage usage	Logical database size and its configured limit, in bytes.
Overloaded shard count	Number of overloaded DataShards by CPU load ranges — from 50% to 100%, pcs.

Traffic section

This section contains charts characterizing database load.

Name	Description
Queries per second by latency buckets	Number of queries per second broken down by latency ranges (from 1 ms to +∞). Each range is highlighted with a separate color — from green for fast queries to purple for slow ones. Allows you to estimate latency distribution and overall RPS, in qps.
Transactions per second by latency buckets	Number of transactions per second broken down by latency ranges (from 1 ms to +∞). Each range is highlighted with a separate color — from green for fast transactions to purple for slow ones. Allows you to estimate latency distribution and overall TPS, in tps.
Rows read, uploaded, updated, deleted	Number of table row operations per second: read, create, update, and delete, in ops/s.
Session count by dynnode	Number of active sessions on each dynamic node, pcs.

Latency section

This section contains charts showing query and transaction execution times.

Name	Description
Query latency percentiles (ms)	Database query execution time at percentiles p50, p90, p95, p99, in milliseconds.
Transaction latency percentiles (ms)	Database transaction execution time at percentiles p50, p90, p95, p99, in milliseconds.

Errors section

This section contains charts showing the number of errors occurring.

Name	Description
YQL Issues per second	Number of YQL query execution errors by error type, in errors/s.
gRPC response errors per second	Number of gRPC responses with errors broken down by status, in errors/s.

Download the **YDB Essential Metrics** dashboard template: ydb-essentials.json.

Actors

CPU consumption in the actor system.

Name	Description
CPU by execution pool (us)	CPU consumption in various execution pools on all nodes, microseconds per second (one million corresponds to one core consumption).
Actor count	Number of actors (by actor type).
CPU	CPU consumption in various execution pools (by actor type).
Events	Event processing metrics in the actor system.

Download the **Actors** dashboard template: actors.json.

CPU

CPU consumption in execution pools.

Name	Description
CPU by execution pool	CPU consumption in various execution pools on all nodes, microseconds per second (one million corresponds to consumption of one core)
Actor count	Number of actors (by actor type)
CPU	CPU consumption in various execution pools
Events	Event processing metrics in various execution pools

Download the **CPU** dashboard template: cpu.json.

gRPC

gRPC layer metrics.

Name	Description
Requests	Number of requests received by the database per second (by gRPC method type)
Request bytes	Size of requests received by the database, bytes per second (by gRPC method type)
Response bytes	Size of responses sent by the database, bytes per second (by gRPC method type)
Dropped requests	Number of requests per second whose processing was terminated at the transport layer due to an error (by gRPC method type)
Dropped responses	Number of responses per second whose sending was terminated at the transport layer due to an error (by gRPC method type)
Requests in flight	Number of requests being processed simultaneously by the database (by gRPC method type)
Request bytes in flight	Size of requests being processed simultaneously by the database (by gRPC method type)

Download the **gRPC** dashboard template: grpc.json.

Query engine

Information about the query execution engine.

Name	Description
Requests	Number of incoming requests per second (by query type)
Request bytes	Size of incoming requests, bytes per second (`query, parameters, total`)
Responses	Number of responses per second (by response type)
Response bytes	Response sizes, bytes per second (`total, query result`)

Sessions	Information about established sessions
Latencies	Histograms of query execution times for various query types

Download the **Query engine** dashboard template: queryengine.json.

TxProxy

Information about transactions from the `DataShard transaction proxy` level.

Name	Description
Transactions	Metrics of datashard transactions
Latencies	Histograms of execution times of various stages of datashard transactions

Download the **TxProxy** dashboard template: txproxy.json.

DataShard

Metrics of the `DataShard` tablet.

Name	Description
Operations	Statistics of operations with the datashard for different operation types
Transactions	Information about transactions of the datashard tablet (by transaction types)
Latencies	Histograms of execution times of various stages of user transactions
Tablet latencies	Histograms of execution times of tablet transactions
Compactions	Information about performed LSM compaction operations
ReadSets	Information about transferred ReadSets during execution of a user transaction
Other	Other metrics

Download the **DataShard** dashboard template: datashard.json.

Database Hive

Metrics of the Hive tablet of the selected database.

The dashboard contains the following filters:

- `database` — used to select the database whose metrics should be displayed;
- `ds` — used to select the Prometheus source whose data should be displayed on the dashboard;
- `Tx type` — determines the transaction type for which graphs will be displayed on the "`{Tx type}` `average time`" panel.

Name	Description
CPU usage by HIVE_ACTOR, HIVE_BALANCER_ACTOR	CPU time consumed by `HIVE_ACTOR` and `HIVE_BALANCER_ACTOR` — the two most important Hive actors.
Self-ping time	Response time of the Hive tablet to its own requests. High values indicate heavy load (and slow responsiveness) of Hive.
Local transaction times	CPU time consumed for executing various types of local transactions in Hive. Displays the load structure on Hive.
Tablet count	Total number of tablets in the database.
Event queue size	Size of the incoming event queue. Consistently high values indicate that Hive is not keeping up with processing events at the required speed.
{Tx type} average time	Average execution time of one local transaction of the type selected in the `Tx type` filter.
Versions	Versions of YDB running on cluster nodes.
Hive node	Node on which Hive is running.

Download the **Database Hive** dashboard template: database-hive-detailed.json.

Topic

The dashboard displays graphs for metrics of a single topic. The topic name is set in the `topic` filter at the top of the dashboard. Below are the panels and metric descriptions.

Name	Description
Total incoming records (bytes) per second	Number of bytes per second written to the topic using the `Ydb::TopicService::StreamWrite` method
Total incoming records (count) per second	Number of messages per second written using the `Ydb::TopicService::StreamWrite` method
Write latency	Write latency: time from message creation to its write to the topic. Percentage of messages for which write latency falls within intervals <100 ms, <200 ms, etc.

Partition throttling	Write throttling duration - waiting for available write quota. Percentage of messages for which write throttling duration falls within intervals <1 ms, <5 ms, etc.
Partition quota usage	Utilization of topic partition write quotas, %
Write sessions active	Number of open write sessions to the topic
Write sessions created	Number of write sessions created per second to the topic

Download the **Topic** dashboard template: topic.json.

Topic — Consumer

The dashboard displays graphs for the metrics of a single topic and its associated consumer. The topic is selected in the `topic` filter, and the consumer in the `consumer` filter. The panels and metric descriptions are listed below.

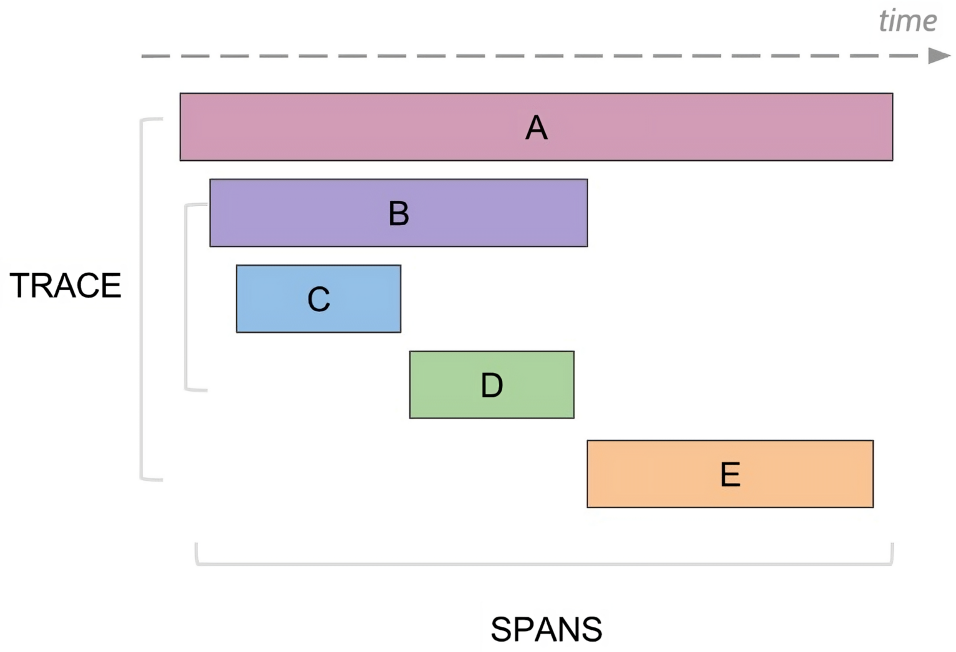
Name	Description
Total incoming records (bytes) per second	Number of bytes per second written to the topic using the `Ydb::TopicService::StreamWrite` method
Total outgoing records (bytes) per second	Number of bytes per second read from the topic by the consumer using the `Ydb::TopicService::StreamRead` method
Total incoming records (count) per second	Number of messages per second written to the topic using the `Ydb::TopicService::StreamWrite` method
Total outgoing records (count) per second	Number of messages per second read from the topic by the consumer using the `Ydb::TopicService::StreamRead` method
End-to-end latency	End-to-end latency: time from message creation to its read. Percentage of messages for which end-to-end latency falls within intervals <100 ms, <200 ms, etc.
Read latency max	Maximum (across all partitions) difference between the current time and the write time of the last message in the topic, ms
Unread messages max	Maximum (across all partitions) difference between the last offset in the partition and the last read offset, in messages
Read idle time max	Maximum idle time (how long the consumer has not read from the partition) across all topic partitions, ms
Uncommitted messages max	Maximum (across all partitions) difference between the last offset in the partition and the last committed offset, in messages
Committed read lag max	Maximum (across all partitions) difference between the current time and the write time of the last committed message in the topic, ms
Partition sessions started	Number of topic read sessions started by the consumer per second

Download the **Topic — Consumer** dashboard template: topic-consumer.json.

Tracing in YDB

> **Note**
>
> The OpenTelemetry website describes the concept of tracing in detail in the Observability Primer article.

Tracing is a tool that allows you to view the detailed path of a request through a distributed system. A set of spans describes the path of a single request (trace). A span is a time segment usually associated with the execution time of a specific operation (e.g., writing information to disk or executing a transaction). Spans form a tree, often with the subtree of a span as its detail, but this is not always the case.



To aggregate disparate spans into traces, they are sent to a *collector*. This service aggregates and stores received spans for subsequent trace analysis. YDB does not include this service; the administrator must set it up independently. Typically, Jaeger is used as a collector.

Minimal configuration

To enable tracing in YDB, add the following section to the configuration:

```
tracing_config:
  backend:
    opentelemetry:
      collector_url: grpc://example.com:4317
      service_name: ydb
  external_throttling:
    - max_traces_per_minute: 10
```

Here, the `collector_url` field sets the URL of an OTLP-compatible span collector. More details on the backend section can be found in the relevant section.

With this configuration, no requests are sampled, and no more than ten requests per minute with an external trace-id are traced by each cluster node.

Section descriptions

Backend

Example section

```
tracing_config:
  # ...
  backend:
    opentelemetry:
      collector_url: grpc://example.com:4317
      service_name: ydb
```

Description

This section describes the span collector. Currently, the only option is `opentelemetry`. Spans are pushed from the cluster node to the collector, requiring the collector to be OTLP compatible.

In the `opentelemetry` section:

- `collector_url` — the URL of the span collector. The scheme can be either `grpc://` for an insecure connection or `grpcs://` for a TLS connection.
- `service_name` — the name under which all spans will be marked.

Both parameters are mandatory.

Uploader

Example section

```
tracing_config:
  # ...
  uploader:
    max_exported_spans_per_second: 30
    max_spans_in_batch: 100
    max_bytes_in_batch: 10485760 # 10 MiB
    max_export_requests_inflight: 3
    max_batch_accumulation_milliseconds: 5000
    span_export_timeout_seconds: 120
```

Description

The uploader is a cluster node component responsible for sending spans to the collector. To avoid overloading the span collector, the uploader will not send more than `max_exported_spans_per_second` spans per second on average.

For optimization, the uploader sends spans in batches. Each batch contains no more than `max_spans_in_batch` spans with a total serialized size of no more than `max_bytes_in_batch` bytes. Each batch accumulates for no more than `max_batch_accumulation_milliseconds` milliseconds. Batches can be sent in parallel, with the maximum number of simultaneously sent batches controlled by the `max_export_requests_inflight` parameter. If more than `span_export_timeout_seconds` seconds have passed since the uploader received the span, the uploader may delete it to send fresher spans.

Default values:

- `max_exported_spans_per_second = inf` (no limits)
- `max_spans_in_batch = 150`
- `max_bytes_in_batch = 20000000`
- `max_batch_accumulation_milliseconds = 1000`
- `span_export_timeout_seconds = inf` (no spans are deleted by the uploader)
- `max_export_requests_inflight = 1`

The `uploader` section may be completely absent, in which case each parameter will use its default value.

> **Note**
>
> The uploader is a node-local component. Therefore, the described limits apply to each node separately, not to the entire cluster.

External throttling

Example section

```
tracing_config:
  # ...
  external_throttling:
    - scope:
        database: /Root/db1
      max_traces_per_minute: 60
      max_traces_burst: 3
```

Description

YDB supports the transmission of external trace-ids to build a coherent request trace. The method for transmitting an external trace-id is described on the Passing external trace-id to YDB page. To avoid overloading the collector, YDB has a mechanism to limit the number of externally traced requests. The limits are described in this section and are a sequence of rules. Each rule contains:

- `scope` – a set of selectors for filtering the request.
- `max_traces_per_minute` – the maximum average number of requests per minute traced by this rule. A positive integer is expected.
- `max_traces_burst` – the maximum burst of externally traced requests. A non-negative integer is expected.

The only mandatory parameter is `max_traces_per_minute`.

A detailed description of these options is provided in the Rule semantics section.

The `external_throttling` section is not mandatory; if it is absent, all trace-ids in requests are **ignored** (no external traces are continued).

This section can be modified without restarting the node using the dynamic configuration mechanism.

Sampling

Example section

```
tracing_config:
  # ...
  sampling:
    - fraction: 0.01
      level: 10
      max_traces_per_minute: 5
      max_traces_burst: 2
    - scope:
```

```
      request_types:
        - KeyValue.ExecuteTransaction
        - KeyValue.Read
    fraction: 0.1
    level: 15
    max_traces_per_minute: 5
    max_traces_burst: 2
```

Description

For diagnosing system issues, looking at a sample request trace can be useful regardless of whether users trace their requests or not. For this purpose, YDB has a request sampling mechanism. For a sampled request, a random trace-id is generated. This section controls request sampling in a format similar to `external_throttling`. Each rule has two additional fields:

- `fraction` – the fraction of requests sampled by this rule. A floating-point number between 0 and 1 is expected.
- `level` — the detail level of the trace. An integer from 0 to 15 is expected. This parameter is described in more detail in the Detail levels section.

Both fields are mandatory.

The `sampling` section is not mandatory; no requests will be sampled if it is absent.

This section can be modified without restarting the node using the dynamic configuration mechanism.

Rule semantics

Selectors

Each rule includes an optional `scope` field with a set of selectors that determine which requests the rule applies to. Currently, the supported selectors are:

- `request_types`

  Accepts a list of request types. A request matches this selector if its type is in the list.

Possible values

- KeyValue.CreateVolume
- KeyValue.DropVolume
- KeyValue.AlterVolume
- KeyValue.DescribeVolume
- KeyValue.ListLocalPartitions
- KeyValue.AcquireLock
- KeyValue.ExecuteTransaction
- KeyValue.Read
- KeyValue.ReadRange
- KeyValue.ListRange
- KeyValue.GetStorageChannelStatus
- Table.CreateSession
- Table.KeepAlive
- Table.AlterTable
- Table.CreateTable
- Table.DropTable
- Table.DescribeTable
- Table.CopyTable
- Table.CopyTables
- Table.RenameTables
- Table.ExplainDataQuery
- Table.ExecuteSchemeQuery
- Table.BeginTransaction
- Table.DescribeTableOptions
- Table.DeleteSession
- Table.CommitTransaction
- Table.RollbackTransaction
- Table.PrepareDataQuery
- Table.ExecuteDataQuery
- Table.BulkUpsert
- Table.StreamExecuteScanQuery
- Table.StreamReadTable
- Table.ReadRows
- Query.ExecuteQuery
- Query.ExecuteScript
- Query.FetchScriptResults
- Query.CreateSession
- Query.DeleteSession
- Query.AttachSession
- Query.BeginTransaction
- Query.CommitTransaction
- Query.RollbackTransaction
- Discovery.WhoAmI
- Discovery.NodeRegistration
- Discovery.ListEndpoints

Note

Tracing is supported not only for the request types listed above. This list includes request types that are supported by the `request_types` selector.

Warning

Note that the QueryService API is experimental and may change in the future.

- `database`

  Filters requests to the specified database.

A request matches a rule if it matches all selectors. `scope` can be absent, which is equivalent to an empty set of selectors, and all requests will fall under this rule.

Rate limiting

The `max_traces_per_minute` and `max_traces_burst` parameters limit the number of requests. In the case of sampling, they limit the number of requests sampled by this rule. In the case of external throttling, they limit the number of external traces that enter the system.

A variation of the leaky bucket is used for rate limiting with a bucket size equal to `max_traces_burst + 1`. For example, if `max_traces_per_minute = 60` and `max_traces_burst = 0`, then with a flow of 10,000 requests per minute, one request will be traced every second. If `max_traces_burst = 20`, then with the same request flow, the first 21 requests will be traced, and then one request per second will be traced.

Warning

The limits on the number of traced requests are local to the cluster node. For example, if each cluster node has a rule specifying `max_traces_per_minute = 1`, then no more than one request per minute will be traced **from each cluster node** by this rule.

Detail levels

As with logs, diagnosing most system issues does not require the most detailed trace. Therefore, in YDB, each span has its own level described by an integer from 0 to 15 inclusive. Each rule in the `sampling` section must include the detail level of the generated trace (`level`); spans with a level less than or equal to `level` will be included in it.

The YDB architecture section describes the system's division into 5 layers:

Layer	Components
1	gRPC Proxies
2	Query Processor
3	Distributes Transactions
4	Tablet, System tablet
5	Distributed Storage

Each layer has seven detail levels:

Level	Value
`Off`	No tracing
`TopLevel`	Lowest detail, no more than two spans per request to the component
`Basic`	Spans of main component operations
`Detailed`	Highest detail applicable for diagnosing problems in production
`Diagnostic`	Detailed debugging information for developers
`Trace`	Very detailed debugging information

The table below shows the distribution of system layer detail levels by trace detail levels:

Trace detail level	gRPC Proxies	Query Processor	Distributed Transactions	Tablets	Distributed Storage
0	`TopLevel`	`Off`	`Off`	`Off`	`Off`
1	`TopLevel`	**`TopLevel`**	`Off`	`Off`	`Off`
2	`TopLevel`	`TopLevel`	**`TopLevel`**	`Off`	`Off`
3	`TopLevel`	`TopLevel`	`TopLevel`	**`TopLevel`**	`Off`
4	`TopLevel`	`TopLevel`	`TopLevel`	`TopLevel`	**`TopLevel`**
5	**`Basic`**	`TopLevel`	`TopLevel`	`TopLevel`	`TopLevel`

6	`Basic`	**`Basic`**	`TopLevel`	`TopLevel`	`TopLevel`
7	`Basic`	`Basic`	**`Basic`**	`TopLevel`	`TopLevel`
8	`Basic`	`Basic`	`Basic`	**`Basic`**	`TopLevel`
9	`Basic`	`Basic`	`Basic`	`Basic`	**`Basic`**
10	**`Detailed`**	**`Detailed`**	`Basic`	`Basic`	`Basic`
11	`Detailed`	`Detailed`	**`Detailed`**	`Basic`	`Basic`
12	`Detailed`	`Detailed`	`Detailed`	**`Detailed`**	`Basic`
13	`Detailed`	`Detailed`	`Detailed`	`Detailed`	**`Detailed`**
14	**`Diagnostic`**	**`Diagnostic`**	**`Diagnostic`**	**`Diagnostic`**	**`Diagnostic`**
15	**`Trace`**	**`Trace`**	**`Trace`**	**`Trace`**	**`Trace`**

Rules

External throttling

The semantics of each rule are as follows: it allocates a quota for the number of requests in this category. For example, if the `external_throttling` section looks like this:

```
tracing_config:
  external_throttling:
    - max_traces_per_minute: 60
    - scope:
        request_types:
          - KeyValue.ReadRange
      max_traces_per_minute: 20
```

With a sufficient flow of requests with an external trace-id, at least 60 requests per minute and at least 20 `KeyValue.ReadRange` type requests per minute will be traced. A total of up to 80 requests per minute will be traced.

The algorithm is as follows: for a request with an external trace-id, the rules that apply to this request are determined. The request consumes the quota of all rules that still have it. The request is not traced only if none of the rules have any quota left.

Sampling

The semantics of the rule for sampling are similar: with a sufficiently low flow of requests in this category, at least a `fraction` of the requests with at least `level` detail will be sampled.

The algorithm is similar: the set of rules that apply to this request is determined for a request without an external trace-id (either due to its initial absence or due to a previous decision not to trace this request). The request consumes the quota of all rules that still have it and that have randomly "decided" to sample it. It is not sampled if no rule decides to sample the request (all rules that "decided" to sample the request have no quota left). Otherwise, the detail level is determined as the maximum among the rules into whose quota the request fell.

For example, with the following `sampling` configuration:

```
tracing_config:
  sampling:
    - scope:
        database: /Root/db1
      fraction: 0.5
      level: 5
      max_traces_per_minute: 100
    - scope:
        database: /Root/db1
      fraction: 0.01
      level: 15
      max_traces_per_minute: 5
```

With a sufficiently low flow of requests to the `/Root/db1` database, the following will be sampled:

- 1% of requests with a detail level of 15
- 49.5% of requests with a detail level of 5

With a sufficiently high flow of requests to the `/Root/db1` database, the following will be sampled:

- 5 requests per minute with a detail level of 15
- between 95 and 100 requests per minute with a detail level of 5

Passing external trace-id to YDB

gRPC API

YDB supports passing external trace-id to build a complete operation trace. The trace-id is passed according to the trace context specification — a string of the form `00-0af7651916cd43dd8448eb211c80319c-b7ad6b7169203331-01` must be passed through the `traceparent` header of the gRPC request. It consists of 4 parts separated by the `-` character:

1. Version – at the time of writing, the specification defines only version 00.
2. Trace id – the identifier of the trace that the request is part of.
3. Parent id – the identifier of the parent span.
4. Flags – a set of recommendations for handling the data passed in the header.

Only version 00 is supported; flags are ignored. If the header does not conform to the specification, it is ignored. All traces obtained this way have detail level 13 (all components are traced at level `Detailed`).

> **Warning**
>
> If the `external_throttling` section is present and the request flow exceeds the set limits, not all requests may be traced. If the `external_throttling` section is absent, the `traceparent` header is **ignored** and no external traces are continued.

SDK support

Some SDKs support trace-id passing; you can find their list and usage examples on the Tracing with OpenTelemetry page.

YDB DSTool overview

With the YDB DSTool utility, you can manage your YDB cluster's disk subsystem. To install and configure the utility, follow the instructions.

YDB DSTool includes the following commands:

Command	Description
device list	List storage devices.
pdisk add-by-serial	Add a PDisk to a set by serial number.
pdisk remove-by-serial	Remove a PDisk from the set by serial number.
pdisk set	Set PDisk parameters.
pdisk list	List PDisks.
pdisk populate	Place the same VDisks on another PDisk for device testing.
vdisk evict	Move VDisks to different PDisks.
vdisk remove-donor	Remove a donor VDisk.
vdisk wipe	Wipe VDisks.
vdisk list	List VDisks.
group add	Add storage groups to a pool.
group check	Check storage groups.
group show blob-info	Display blob information.
group show usage-by-tablets	Display information about tablet usage by groups.
group state	Show or change a storage group's state.
group take-snapshot	Take a snapshot of storage group metadata.
group list	List storage groups.
pool list	List pools.
box list	List sets of PDisks.
node list	List nodes.
cluster balance	Move VDisks from overloaded PDisks.
cluster get	Show cluster parameters.
cluster set	Set cluster parameters.
cluster workload run	Run a workload to test the failure model.
cluster list	Display cluster information.

YDB DSTool overview

With the YDB DSTool utility, you can manage your YDB cluster's disk subsystem. To install and configure the utility, follow the instructions.

YDB DSTool includes the following commands:

Command	Description
device list	List storage devices.
pdisk add-by-serial	Add a PDisk to a set by serial number.
pdisk remove-by-serial	Remove a PDisk from the set by serial number.
pdisk set	Set PDisk parameters.
pdisk list	List PDisks.
pdisk populate	Place the same VDisks on another PDisk for device testing.
vdisk evict	Move VDisks to different PDisks.
vdisk remove-donor	Remove a donor VDisk.
vdisk wipe	Wipe VDisks.
vdisk list	List VDisks.
group add	Add storage groups to a pool.
group check	Check storage groups.
group show blob-info	Display blob information.
group show usage-by-tablets	Display information about tablet usage by groups.
group state	Show or change a storage group's state.
group take-snapshot	Take a snapshot of storage group metadata.
group list	List storage groups.
pool list	List pools.
box list	List sets of PDisks.
node list	List nodes.
cluster balance	Move VDisks from overloaded PDisks.
cluster get	Show cluster parameters.
cluster set	Set cluster parameters.
cluster workload run	Run a workload to test the failure model.
cluster list	Display cluster information.

Installing the YDB DSTool

Linux

To install the YDB DSTool, follow these steps:

1. Run the command:

   ```
   curl -sSL 'https://install.ydb.tech/dstool' | bash
   ```

   The script will install the YDB DSTool. If the script is run from a `bash` or `zsh` shell, it will also add the `ydb-dstool` executable to the `PATH` environment variable. Otherwise, you can run it from the `~/ydb/bin` folder or add it to `PATH` manually.
2. To update the environment variables, restart the command shell.
3. Test it by running the command that shows cluster information:

   ```
   ydb-dstool -e <bs_endpoint> cluster list
   ```

   - `bs_endpoint` : URI of the cluster's HTTP endpoint, the same endpoint that serves the YDB UI. Example: `http://localhost:8765`.

   Result:

   ```
   ┌───────┬───────┬───────┬────────┬────────┬───────┬────────┐
   | Hosts | Nodes | Pools | Groups | VDisks | Boxes | PDisks |
   ├───────┼───────┼───────┼────────┼────────┼───────┼────────┤
   | 8     | 16    | 1     | 5      | 40     | 1     | 32     |
   └───────┴───────┴───────┴────────┴────────┴───────┴────────┘
   ```

macOS

To install the YDB DSTool, follow these steps:

1. Run the command:

   ```
   curl -sSL 'https://install.ydb.tech/dstool' | bash
   ```

   The script will install the YDB DSTool. If the script is run from a `bash` or `zsh` shell, it will also add the `ydb-dstool` executable to the `PATH` environment variable. Otherwise, you can run it from the `~/ydb/bin` folder or add it to `PATH` manually.
2. To update the environment variables, restart the command shell.
3. Test it by running the command that shows cluster information:

   ```
   ydb-dstool -e <bs_endpoint> cluster list
   ```

   - `bs_endpoint` : URI of the cluster's HTTP endpoint, the same endpoint that serves the YDB UI. Example: `http://localhost:8765`.

   Result:

   ```
   ┌───────┬───────┬───────┬────────┬────────┬───────┬────────┐
   | Hosts | Nodes | Pools | Groups | VDisks | Boxes | PDisks |
   ├───────┼───────┼───────┼────────┼────────┼───────┼────────┤
   | 8     | 16    | 1     | 5      | 40     | 1     | 32     |
   └───────┴───────┴───────┴────────┴────────┴───────┴────────┘
   ```

Windows

To install the YDB DSTool, follow these steps:

1. Run the command:
   - in **PowerShell**:

     ```
     iex (New-Object System.Net.WebClient).DownloadString('https://install.ydb.tech/dstool-windows')
     ```

   - in **CMD**:

     ```
     @"%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe" -Command "iex ((New-Object System.Ne
     t.WebClient).DownloadString('https://install.ydb.tech/dstool-windows'))"
     ```

2. Specify whether to add `ydb-dstool` to the `PATH` environment variable:

   ```
   Add ydb-dstool installation dir to your PATH? [Y/n]
   ```

3. To update the environment variables, restart the command shell.

Note

The YDB DSTool uses Unicode characters in the output of some commands. If these characters aren't displayed correctly in the Windows console, switch the encoding to UTF-8:

```
chcp 65001
```

4. Test it by running the command that shows cluster information:

```
ydb-dstool -e <bs_endpoint> cluster list
```

   - `bs_endpoint` : URI of the cluster's HTTP endpoint, the same endpoint that serves the YDB UI. Example: `http://localhost:8765`.

   Result:

```
┌─────────┬─────────┬─────────┬─────────┬─────────┬─────────┬─────────┐
| Hosts | Nodes | Pools | Groups | VDisks | Boxes | PDisks |
├─────────┼─────────┼─────────┼─────────┼─────────┼─────────┼─────────┤
| 8     | 16    | 1     | 5      | 40     | 1     | 32     |
└─────────┴─────────┴─────────┴─────────┴─────────┴─────────┴─────────┘
```

Global options

All the YDB DSTool utility subcommands share the same global options.

Option	Description
`-?`, `-h`, `--help`	Print the built-in help.
`-v`, `--verbose`	Print detailed output while executing the command.
`-q`, `--quiet`	Suppress non-critical messages when executing the command.
`-n`, `--dry-run`	Dry-run the command.
`-e`, `--endpoint`	Endpoint to connect to the YDB cluster, in the format: `[PROTOCOL://]HOST[:PORT]`. Default values: PROTOCOL — `http`, PORT — `8765`.
`--grpc-port`	gRPC port used to invoke procedures.
`--mon-port`	Port to view HTTP monitoring data in JSON format.
`--mon-protocol`	If you fail to specify the cluster connection protocol explicitly in the endpoint, the protocol is taken from here.
`--token-file`	Path to the file with Access Token.
`--ca-file`	Path to a root certificate PEM file used for TLS connections.
`--http`	Use HTTP instead of gRPC to connect to the Blob Storage.
`--http-timeout`	Timeout for I/O operations on the socket when running HTTP(S) queries.
`--insecure`	Allow insecure data delivery over HTTPS. Neither the SSL certificate nor host name are checked in this mode.

device list

Use the `device list` subcommand to get a list of storage devices available on the YDB cluster.

General format of the command:

```
ydb-dstool [global options ...] device list [list options ...]
```

- `global options`: Global options.
- `list options`: Subcommand options.

View a description of the command to get a list of devices:

```
ydb-dstool device list --help
```

Subcommand options

Option	Description
`-H`, `--human-readable`	Output data in human-readable format.
`--sort-by`	Sort column. Use one of the values: `SerialNumber`, `FQDN`, `Path`, `Type`, `StorageStatus`, or `NodeId:PDiskId`.
`--reverse`	Use a reverse sort order.
`--format`	Output format. Use one of the values: `pretty`, `json`, `tsv`, or `csv`.
`--no-header`	Do not output the row with column names.
`--columns`	List of columns to be output. Use one or more of the values: `SerialNumber`, `FQDN`, `Path`, `Type`, `StorageStatus`, or `NodeId:PDiskId`.
`-A`, `--all-columns`	Output all columns.

Examples

The following command will output a list of devices available in the cluster:

```
ydb-dstool -e node-5.example.com device list
```

Result:

```
┌──────────────────┬──────────────────┬──────────────────────────────┬──────┬──────────────────────────────
─┬──────────────────┐
│ SerialNumber       │ FQDN               │ Path                           │ Type │ StorageStatus
│ NodeId:PDiskId │
├──────────────────┼──────────────────┼──────────────────────────────┼──────┼──────────────────────────────
─┼──────────────────┤
│ PHLN123301H41P2BGN │ node-1.example.com │ /dev/disk/by-partlabel/nvme_04 │ NVME │ FREE
│ NULL           │
│ PHLN123301A62P2BGN │ node-6.example.com │ /dev/disk/by-partlabel/nvme_03 │ NVME │ PDISK_ADDED_BY_DEFINE_BOX
│ [6:1001]       │
...
```

pdisk populate

Use the `pdisk populate` subcommand to move a selected set of VDisks to one PDisk. The command can save the active VDisks of a source PDisk to a snapshot file and then use that file to populate a destination PDisk.

> **Warning**
>
> Use `pdisk populate` for controlled device testing: it lets you place exactly the same set of VDisks on a new device and compare how the old and new devices handle that workload.

The Blob Storage Controller validates all selected VDisks and schedules their reassignment in a single atomic configuration transaction. If any VDisk cannot be reassigned, the configuration is not changed. The data migration itself continues asynchronously after the transaction is applied.

General format of the command:

```
ydb-dstool [global options ...] pdisk populate [populate options ...]
```

- `global options` : Global options.
- `populate options`: Subcommand options.

View a description of the command:

```
ydb-dstool pdisk populate --help
```

Subcommand options

Option	Description
`--snapshot-from-pdisk <NodeId:PDiskId>`	Snapshot mode. Collect active VDisks from the specified PDisk. Donor VDisks are skipped.
`-d`, `--destination-pdisk <NodeId:PDiskId>`	Populate mode. Move the VDisks listed in `--snapshot-file` to the specified PDisk.
`--snapshot-file <PATH>`	In snapshot mode, write the VDisk list as JSON to the specified file. In populate mode, read the VDisk list from this file. This option is required in populate mode.
`--suppress-donor-mode`	Do not leave the previous VDisk locations in donor mode after reassignment. This option is only available in populate mode.
`--format <FORMAT>`	Output format: `pretty` (default) or `json`.

Exactly one of `--snapshot-from-pdisk` and `--destination-pdisk` must be specified.

Example

First, save the list of active VDisks on PDisk `[1:1000]` to a snapshot file:

```
ydb-dstool -e node-1.example.com pdisk populate \
  --snapshot-from-pdisk '[1:1000]' \
  --snapshot-file /tmp/pdisk-1-1000.json
```

The snapshot file has the following format:

```
{
  "pdisk_id": "[1:1000]",
  "vdisk_ids": [
    "[80000001:_:0:0:0]",
    "[80000002:_:0:1:0]"
  ]
}
```

Review the snapshot and then move the listed VDisks to PDisk `[2:1000]`:

```
ydb-dstool -e node-1.example.com pdisk populate \
  --destination-pdisk '[2:1000]' \
  --snapshot-file /tmp/pdisk-1-1000.json
```

The command skips VDisks with `GroupId=0`, because they cannot be migrated. If the snapshot contains only such VDisks, the command returns an error without changing the configuration.

ydbops utility overview

> **Note**
>
> The `ydbops` utility is under active development. Although backward-incompatible changes are unlikely, they may still occur.

`ydbops` utility automates some operational tasks on YDB clusters. It supports clusters deployed using Ansible, Kubernetes, or manually.

See also

- To install the utility, follow the instructions.
- See configuration reference for available configuration options.
- The source code of `ydbops` can be found on GitHub.

Currently supported scenarios

See the list of currently supported scenarios here.

Scenarios in development

- Requesting permission to take out a set of YDB nodes for maintenance without breaking YDB fault model invariants.

ydbops utility overview

> **Note**
>
> The `ydbops` utility is under active development. Although backward-incompatible changes are unlikely, they may still occur.

`ydbops` utility automates some operational tasks on YDB clusters. It supports clusters deployed using Ansible, Kubernetes, or manually.

See also

- To install the utility, follow the instructions.
- See configuration reference for available configuration options.
- The source code of `ydbops` can be found on GitHub.

Currently supported scenarios

See the list of currently supported scenarios here.

Scenarios in development

- Requesting permission to take out a set of YDB nodes for maintenance without breaking YDB fault model invariants.

Installing ydbops

> **Note**
>
> The `ydbops` utility is under active development. Although backward-incompatible changes are unlikely, they may still occur.

Download the binary from the releases page

You can download binary releases from Download YDB Ops.

Building from source

1. Clone the `ydbops` repository from GitHub:

   ```
   git clone https://github.com/ydb-platform/ydbops.git
   ```

2. There are two ways to build `ydbops`:
   2.1. Directly with Go
   2.2. Inside a Docker container

The second approach requires a prepared Docker environment and uses the official Docker image for Golang v1.22, guaranteeing a successful build. The Docker container operates using the `Dockerfile` from the repository. The build process in Docker also performs additional tasks: running linter checks and substituting the version for the `ydbops` assembly to register it in the executable file.

Building directly with Go

Prerequsites

Install Go. The recommended version is 1.22.

Compiling

Invoke `go build` in the repository root folder:

```
go build
```

The `ydbops` executable will be available in the repository root folder.

Installing

You can copy the executable file manually or use `make`:

```
make install INSTALL_DIR=install_folder BUILD_DIR=.
```

Inside a Docker container

Prerequsites

- make
- Install docker engine

Compiling

Invoke this command in the repository root folder:

```
make build-in-docker
```

The `ydbops` executables will be available in the `bin` folder. Binary files are generated for Linux and macOS (arm64, amd64).

Binary name	Platform
ydbops	Linux (amd64)
ydbops_darwin_amd64	macOS (amd64)
ydbops_darwin_arm64	macOS (arm64)

Installing

To install the binary file, execute the command `make`.

Optional parameters:

- `INSTALL_DIR`: The folder where the executable file will be installed. Default value: `~/ydb/bin`.
- `BUILD_DIR`: The folder that contains the generated executable file. Use this parameter if you created the executable file manually. For example, use `BUILD_DIR=.` if the executable file is in the current working directory.

```
make install [INSTALL_DIR=<path_to_install_folder>] [BUILD_DIR=<path_to_build_folder>]
```

Sample command to install into `install_folder` from the current folder:

```
make install INSTALL_DIR=install_folder BUILD_DIR=.
```

Configuring ydbops

> **Note**
>
> The `ydbops` utility is under active development. Although backward-incompatible changes are unlikely, they may still occur.

`ydbops` can be run by specifying all the necessary command line arguments on the command invocation. However, it has two features that allow to avoid repeating the commonly used arguments:

- Config file
- Environment variables

Config file

The configuration file for `ydbops` is a YAML-formatted file containing multiple profiles. Profiles for `ydbops` work in the same way as profiles in YDB CLI do.

Default configuration file location follows the same convention as YDB CLI does, it is located in the same folder in `ydbops` subdirectory. For comparison:

- default configuration file for YDB CLI: `$HOME/.ydb/config/config.yaml`
- default configuration file for `ydbops` CLI, `ydbops` subdirectory: `$HOME/ydb/ydbops/config/config.yaml`

Certain command line options can be written in the configuration file instead of being specified directly in the `ydbops` invocation.

Examples

Calling the `ydbops restart` command without a profile:

```
ydbops restart \
 -e grpc://<hostname>:2135 \
 --kubeconfig ~/.kube/config \
 --k8s-namespace <k8s-namespace> \
 --user admin \
 --password-file ~/<password-file> \
 --tenant --tenant-list=my-tenant
```

Calling the same `ydbops restart` command with profile options enabled makes the command much shorter:

```
ydbops restart \
 --config-file ./config.yaml \
 --tenant --tenant-list=<tenant-name>
```

For the invocation above, the following `config.yaml` is assumed to be present:

```
current-profile: my-profile
profiles:
  my-profile:
    endpoint: grpcs://<hostname>:2135
    ca-file: ~/<ca-certificate-file>
    user: <username>
    password-file: ~/<password-file>
    k8s-namespace: <k8s-namespace>
    kubeconfig: ~/.kube/config
```

Profile management commands

Currently, `ydbops` does not support the creation, modification, and activation of profiles via the CLI commands the way that YDB CLI does.

The configuration file needs to be created and edited manually.

Configuration file reference

Here is an example of a configuration file with all possible options that can be specified and example values (most likely, they will not all be needed at the same time):

```
# a special key `current-profile` can be specified to
# be used as the default active profile in the CLI invocation
current-profile: my-profile

# profile definitions are added as subkeys to the `profiles` key
profiles:
  my-profile:
    endpoint: grpcs://your.ydb.cluster.fqdn:2135

    # CA file location if using grpcs to the endpoint
    ca-file: /path/to/custom/ca/file

    # a username and password file if static credentials are used:
    user: your-ydb-user-name
    password-file: /path/to/password-file

    # when using access token
    token-file: /path/to/ydb/token
```

```
# if working with YDB clusters in Kubernetes, kubeconfig path can be specified:
kubeconfig: /path/to/kube/config
k8s-namespace: <k8s-namespace>
```

Environment variables

Alternatively, there is an option to specify several environment variables instead of passing command-line arguments or using config files.

For an explanation of which options take precedence, please invoke `ydbops --help`.

- `YDB_TOKEN` can be passed instead of the `--token-file` flag or `token-file` profile option.
- `YDB_PASSWORD` can be passed instead of the `--password-file` flag or `password-file` profile option.
- `YDB_USER` can be passed instead of the `--user` flag or `user` profile option.

Performing YDB cluster restart using ydbops

> **Note**
>
> The `ydbops` utility is under active development. Although backward-incompatible changes are unlikely, they may still occur.

`ydbops` can be used to perform the rolling restart operation: restarting all or some of YDB cluster nodes while maintaining cluster availability. Why this is not trivial and requires a special utility is explained in the article about maintenance without downtime.

The subcommand responsible for this operation is `ydbops restart`.

General algorithm

There are multiple options for `ydbops restart` that act as filters. Filters are implicitly connected with a logical "and", meaning that if you supply multiple filters, only the nodes that satisfy all of them at once will be targeted. Therefore, specifying no filters targets all nodes for restart.

There are two special filters that are an exception to this rule, `--storage` and `--tenant`:

- Specifying only `--storage` will filter storage nodes only.
- Specifying only `--tenant` will filter tenant nodes only.
- However, specifying both selects all nodes and is equivalent to not specifying these two filters.

The algorithm will always work in two phases:

- First, it will determine whether any storage nodes fit the restart filters and restart these nodes only.
- After all storage nodes have been restarted or it has been determined that no storage nodes are selected, the process repeats for tenant nodes.

Examples

The following examples assume you have specified all the required connection options (such as endpoint or credentials).

Restarting all nodes in the cluster

The command will restart all the nodes in the cluster: all storage nodes first, followed by all tenant nodes.

```
ydbops restart
```

Restarting only storage or tenant nodes

It is possible to restart storage nodes only:

```
ydbops restart --storage
```

Or tenant nodes only:

```
ydbops restart --tenant
```

Additionally, only specific tenants may be restarted by specifying `--tenant-list`:

```
ydbops restart --tenant-list=</domain/database_name_1>,</domain/database_name_2>,...
```

Restarting only specific nodes

It is possible to restart nodes only on specific hosts by supplying FQDNs with the `--hosts` option:

```
ydbops restart --hosts=<node1.some.zone>,<node2.some.zone>
```

Or by supplying node ids directly:

```
ydbops restart --hosts=1,2,3
```

Restarting based on node uptime

It is possible to restart only the nodes that have specific uptime by using the `--started` option:

The option argument needs to be enclosed in quotes, otherwise shell might interpret `>` or `<` signs as stream redirections. See the example:

```
ydbops restart --started '>2024-03-13T17:00:00Z'
```

It might be convenient to restart only the nodes that have been up for over a few minutes, as the others have just restarted recently.

Restarting based on YDB version

It is possible to restart only the nodes whose version is equal to (`==`), not equal to (`!=`), greater than (`>`), or less than (`<`) the desired version.

The option argument needs to be enclosed in quotes; otherwise, the shell might interpret the `>` or `<` signs as stream redirections. See the examples:

```
ydbops restart --version '>24.1.2'
ydbops restart --version '<24.1.2'
ydbops restart --version '!=24.1.2'
ydbops restart --version '==24.1.2'
```

Hotfix versions (for example, `ydb-stable-24-1-14-hotfix-9`) have the same major, minor, and patch numbers as the corresponding non-hotfix version `24.1.14`.

For example, version `ydb-stable-24-1-14-hotfix-9` in comparison operations will be identical to version `ydb-stable-24-1-14`.

YDBOps Reference Sheet

ydbops options

Database Connection Options

DB connection options are described in Connecting to and authenticating with a database.

Service Options

- `--verbose`: increases output verbosity.
- `--profile-file`: use profiles from the specified file. By default, profiles from the `$HOME/ydb/ydbops/config/config.yaml file are used`.
- `--profile <string>`: override currently set profile name from `--profile-file`.
- `--grpc-timeout-seconds <int>`: wait this much time in seconds before timing out any GRPC requests (default 60).
- `--grpc-skip-verify`: do not verify server hostname when using gRPCs.
- `--ca-file <filepath>`: path to root ca file, appends to system pool.

Restart Options

- `--storage`: only include storage nodes. If no `--storage` or `--tenant` is specified, both `--storage` and `--tenant` become active, as it is assumed that the intention is to restart the whole cluster.
- `--tenant`: only include tenant nodes. Additionally, you can specify:
  - `--tenant-list=<tenant-name-1>,<tenant-name-2>`
- `--availability-mode <strong|weak|force>`: see the article about maintenance without downtime. Defaults to `strong`.
- `--restart-duration <int>`: multiplied by `--restart-retry-number`, this gives the total duration in seconds for the maintenance operation. In other words, it is a promise to CMS that a single node restart will finish within given duration. Defaults to 60 (which makes the default CMS request duration 180 seconds in combination with the default value of `--restart-retry-number`)
- `--restart-retry-number <int>`: if restarting a specific node failed, repeat the restart operation this much times. Defaults to 3.
- `--cms-query-interval <int>`: how often to query for updates from CMS while waiting for new nodes. Defaults to 10 seconds.
- `--nodes-inflight <int>`: the maximum number of nodes that are concurrently being restarted.
- `--delay-between-restarts <duration>`: delay before initiating the next node restart.

Filtering Options

Filtering options allow you to narrow down the list of nodes to restart.

- `--hosts <list>`: restart the following hosts. Hosts can be specified in multiple ways:
  - Using node ids: `--hosts=1,2,3`
  - Using host fqdns: `--hosts=<node1.some.zone>,<node2.some.zone>`
- `--exclude-hosts <list>`: do not restart the following hosts even if explicitly included. Syntax is the same as with the `--hosts` option.
- `--started '<sign><timestamp>'`: restart only the nodes that satisfy the particular uptime. Specify the sign (`<` or `>`) and a timestamp in ISO format to filter only the nodes that started before or after a particular timestamp. Be careful to enclose the timestamp with a sign in quotes, otherwise shell might interpret the sign as stream redirection.
  - example: `ydbops restart --started '>2024-03-13T17:00:00Z'`
- `--version '<sign><major>.<minor>.<patch>'`: restart only the nodes that satisfy the particular version filter. Specify the sign (`<`, `>`, `!=` or `==`), then specify the version by supplying three numbers: major, minor, patch versions. Be careful to enclose the timestamp with a sign in quotes, otherwise shell might interpret the sign as stream redirection.
  - example: `ydbops restart --version '>24.3.1'`
  - the command works with ydb processes that have their version in the following format: `ydb-stable-<major>-<minor>-<patch>.*`. Hotfix versions (e.g. `ydb-stable-24-1-14-hotfix-9`) have the same major, minor, patch numbers as their non-hotfix `24.1.14`. For example, `ydb-stable-24-1-14-hotfix-9` is treated in the same way as `ydb-stable-24-1-14`.

Kubernetes Options

If `--kubeconfig` is specified, it is assumed that the cluster to be restarted runs under Kubernetes, and node restart will be achieved by deleting pods as opposed to other ways of restart (e.g. systemd units).

- `--kubeconfig`: location to kubeconfig file which will be used for communicating with Kubernetes API (e.g. restarting nodes by deleting pods). There is no default value, since specifying this option also indicates Kubernetes mode.
- `--k8s-namespace`: namespace where pods with storage or tenant processes are located. The usecase with nodes located in multiple namespaces is currently unsupported.

YDB Docker container reference

This section provides detailed information about working with YDB when delivered as a Docker container.

The YDB Docker container reference documentation includes the following sections:

- Docker image `ydbplatform/local-ydb` tags naming
- Prerequisites for working with YDB in Docker
- Running YDB in Docker
- Configuring the YDB Docker container
- Custom initialization scripts
- Docker stop

Docker image `ydbplatform/local-ydb` tags naming

For the ydbplatform/local-ydb Docker image, the following naming rules apply for tags:

Tag Name	Description
`latest`	Corresponds to the most recent *stable* version of YDB tested on production clusters. Rebuilt for each new YDB release.
`edge`	A candidate for the next *stable* version, currently undergoing testing. Includes new features but may not be stable and thus unsuitable for production environments.
`trunk`, `main`, `nightly`	The latest version of YDB from the main development branch. Includes all recent changes and is rebuilt nightly. Similarly to `edge`, it is not suitable for production environments.
`XX.Y`	Corresponds to the latest minor version of YDB in a major release `XX.Y`, including all patches.
`XX.Y.ZZ`	Corresponds to the YDB release version `XX.Y.ZZ`.
`XX.Y-slim`, `XX.Y.ZZ-slim`	Compressed binaries of YDB (`ydbd` and `ydb`) with smaller image size but a slower startup. Uses UPX.

Prerequisites for working with YDB in Docker

Before using the YDB in Docker, install and configure the Docker environment. Refer to the official documentation for your operating system:

- Linux
- macOS
- Windows

Alternative installation methods, such as colima, are also supported.

The installation of YDB by downloading the container image happens during the first launch.

Running YDB in Docker

Before start

Create a folder for testing YDB and use it as the current working directory:

```
mkdir ~/ydbd && cd ~/ydbd
mkdir ydb_data
mkdir ydb_certs
```

Launching a container with YDB in Docker

Example of the YDB startup command in Docker with detailed comments:

```
docker_args=(
    -d                              # run container in background and print container ID
    --rm                            # automatically remove the container
    --name ydb-local                # assign a name to the container
    --hostname localhost            # hostname
    --platform linux/amd64          # platform
    -p 2135:2135                    # publish a container grpcs port to the host
    -p 2136:2136                    # publish a container grpc port to the host
    -p 8765:8765                    # publish a container http port to the host
    -p 9092:9092                    # publish a container port to the host that provides Kafka compatibility
    -v $(pwd)/ydb_certs:/ydb_certs  # mount directory with TLS certificates
    -v $(pwd)/ydb_data:/ydb_data    # mount working directory
    -e GRPC_TLS_PORT=2135           # grpcs port, needs to match what's published above
    -e GRPC_PORT=2136               # grpc port, needs to match what's published above
    -e MON_PORT=8765                # http port, needs to match what's published above
    -e YDB_KAFKA_PROXY_PORT=9092    # port, needs to match what's published above
    ydbplatform/local-ydb:latest
)

docker run "${docker_args[@]}"
```

> **Note**
>
> If you are using a Mac with an Apple Silicon processor, emulate the x86_64 CPU instruction set with Rosetta:
>
> - colima with the `colima start --arch aarch64 --vm-type=vz --vz-rosetta` options.
> - Docker Desktop with installed and enabled Rosetta 2.
>
> If Rosetta 2 is not enabled, add the `-e YDB_USE_IN_MEMORY_PDISKS=true` parameter to the command for running the Docker container. For more information, see Configuring the YDB Docker container.

For more information about environment variables available when running a Docker container with YDB, see Configuring the YDB Docker container.

With the parameters specified in the example above and running Docker locally, YDB UI YDB will be available at http://localhost:8765.

For more information about stopping and deleting a Docker container with YDB, see Docker stop.

Overriding the configuration file

By default, when starting a Docker container for YDB, a built-in configuration file is used, which provides standard operating parameters. To use your own config, mount it at `/ydb_data/cluster/kikimr_configs/config.yaml`:

```
docker run "${docker_args[@]}" \
  -v $(pwd)/ydb_config/my-ydb-config.yaml:/ydb_data/cluster/kikimr_configs/config.yaml
```

For users who are not experienced with Docker, it's important to understand how to properly mount a configuration file into the container. Below is a step-by-step example:

1. Run the container without specifying a configuration file and without mounting the data directory, so that it generates a default configuration:

   ```
   docker run -d \
     --rm \
     --name ydb-local \
     --hostname localhost \
     --platform linux/amd64 \
     -v $(pwd)/ydb_certs:/ydb_certs \
     -e GRPC_TLS_PORT=2135 \
     -e GRPC_PORT=2136 \
     -e MON_PORT=8765 \
     ydbplatform/local-ydb:latest
   ```

2. Create a directory for your configuration files and copy the generated configuration file from the container directly to it:

   ```
   mkdir ydb_config
   docker cp ydb-local:/ydb_data/cluster/kikimr_configs/config.yaml ydb_config/my-ydb-config.yaml
   ```

3. Stop the container if it's still running, and remove the created data directory:

```
docker stop ydb-local
rm -rf ydb_data
```

4. Edit the copied configuration file `ydb_config/my-ydb-config.yaml` as needed.
5. When running the container, mount the edited configuration file at `/ydb_data/cluster/kikimr_configs/config.yaml`:

```
docker_args=(
    -d
    --rm
    --name ydb-local
    --hostname localhost
    --platform linux/amd64
    -p 2135:2135
    -p 2136:2136
    -p 8765:8765
    -v $(pwd)/ydb_data:/ydb_data
    -v $(pwd)/ydb_certs:/ydb_certs
    -v $(pwd)/ydb_config/my-ydb-config.yaml:/ydb_data/cluster/kikimr_configs/config.yaml
    -e GRPC_TLS_PORT=2135
    -e GRPC_PORT=2136
    -e MON_PORT=8765
    ydbplatform/local-ydb:latest
)

docker run "${docker_args[@]}"
```

In this example, the local file `ydb_config/my-ydb-config.yaml` replaces the default configuration inside the container.

Configuring the YDB Docker container

YDB is configured via environment variables when running in Docker. When starting the YDB Docker container with the `docker run` command, you can specify additional environment variables using the `-e` option to change the container's behavior. For more than one environment variable, specify this option multiple times. Below is the full list of supported environment variables.

Environment variables

Name	Type	Default	Description
`YDB_GRPC_ENABLE_TLS`	`0`, `1`, `false`, or `true`	`true`	Enable the `grpcs://` protocol (gRPC over TLS).
`YDB_GRPC_TLS_DATA_PATH`	`string`	`/ydb_certs`	Directory with TLS certificates for the `grpcs://` connection.
`MON_PORT`	`int`	`8765`	HTTP port of YDB UI
`GRPC_PORT`	`int`	`2136`	gRPC port
`IC_PORT`	`int`	`19001`	Interconnect port
`GRPC_TLS_PORT`	`int`	`2135`	gRPC over TLS port
`YDB_KAFKA_PROXY_PORT`	`int`		Port for connecting using the Kafka API. An empty value disables Kafka API compatibility.
`YDB_ERASURE`	`string`	`none`	Erasure to use, see YDB Cluster Topology
`FQ_CONNECTOR_ENDPOINT`	`string`	`None`	Connection string for the connector to external sources `fq-connector-go`, see Federated query
`YDB_USE_IN_MEMORY_PDISKS`	`0` or `1`	`0`	Makes all data volatile and stored only in RAM. Currently, saving data by disabling this option is supported only on x86_64 processors or virtual machines emulating them.
`YDB_PDISK_SIZE`	`string`	`64GB`	The size of the file for storing data in the `/ydb_data` directory used by the running container. It can be specified as a number in bytes or with a suffix: `KB`, `MB`, or `GB` (for example, `64GB`).
`YDB_DEFAULT_LOG_LEVEL`	`string`	`NOTICE`	Sets the logging level by default. Available values: `CRIT`, `ERROR`, `WARN`, `NOTICE`, `INFO`.
`YDB_ADDITIONAL_LOG_CONFIGS`	`string`	`None`	Sets additional logging levels for specified ydb components in format: `component`:`level`. If you need to enter multiple levels, then enter them separated by commas.
`YDB_FEATURE_FLAGS`	`string`	`None`	Comma-separated list of experimental features YDB
`YDB_ENABLE_COLUMN_TABLES`	`0` or `1`	`0`	Enables column-oriented tables
`YDB_PREINITSCRIPTS_DIR`	`string`	`/preinit.d`	Path to the pre-init scripts directory
`YDB_INITSCRIPTS_DIR`	`string`	`/init.d`	Path to the init scripts directory

Custom initialization scripts

The YDB Docker container supports custom initialization scripts that allow you to automate database setup tasks.

Script directories

There are two directories for placing custom scripts:

Directory	Description	Usage
`/preinit.d`	Scripts in this directory are executed on every container start, **before** the YDB server starts.	Executed on every container start before the server starts. Useful for setting environment variables, configuring logging, or other preparatory tasks that might need to be done each time the container starts.
`/init.d`	Scripts in this directory are executed only **once** after a successful YDB server start. A marker file is created to prevent re-execution on subsequent container restarts.	Executed only once after a successful server start. For example, creating database structure (tables, indexes) and inserting initial data.

Understanding preinit vs init

- **preinit.d**: Executed on every container start, before the YDB server starts. Useful for setting environment variables, configuring logging, or other preparatory tasks that might need to be done each time the container starts.
- **init.d**: Executed only once after a successful YDB server start. Example: creating database structure (tables, indexes) and inserting initial data.

Supported file types

Pre-init scripts (`/preinit.d`)

Extension	Description
`.sh`	Shell scripts. If the script is executable, it is run directly. Otherwise, it is sourced, allowing it to modify environment variables for subsequent scripts.

Init scripts (`/init.d`)

Extension	Description
`.sh`	Shell scripts. If the script is executable, it is run directly. Otherwise, it is sourced.
`.sql`	SQL files. The contents are executed using the YDB command-line interface.
`.sql.gz`	Gzip-compressed SQL files. The contents are decompressed and executed.

Execution order

Scripts are executed in alphabetical order within each directory. Use numeric prefixes to control the execution sequence:

```
/init.d/
├── 01-create-tables.sh
├── 02-create-indexes.sql
└── 03-insert-data.sql.gz
```

Environment variables

You can customize the script directories using environment variables:

Variable	Default	Description
`YDB_PREINITSCRIPTS_DIR`	`/preinit.d`	Path to the pre-init scripts directory
`YDB_INITSCRIPTS_DIR`	`/init.d`	Path to the init scripts directory

Error handling

If any script fails (exits with a non-zero status), the container stops execution and exits with an error. This ensures that initialization errors are immediately visible and prevents the container from running with an incomplete setup.

Examples

Using shell scripts

Create a file `/init.d/01-setup.sh` with the following contents:

```
#!/bin/bash
# /init.d/01-setup.sh
echo "Setting up database..."
/ydb -e grpc://localhost:2136 -d /local --no-discovery sql -s "CREATE TABLE test (id Uint64, PRIMARY KEY (i
d));"
```

Mount the script when starting the container:

```
docker run -d \
    --name ydb-local \
```

```
    -v $(pwd)/init.d:/init.d \
    ydbplatform/local-ydb:latest
```

When the container starts, the `test` table will be automatically created using the YDB CLI. The script runs only once after a successful server start.

Using SQL files

Create a file `/init.d/01-create-tables.sql` with the following contents:

```
-- /init.d/01-create-tables.sql
CREATE TABLE users (
    id Uint64,
    name Utf8,
    email Utf8,
    PRIMARY KEY (id)
);

CREATE TABLE orders (
    id Uint64,
    user_id Uint64,
    amount Double,
    PRIMARY KEY (id)
);
```

Mount the init directory:

```
docker run -d \
    --name ydb-local \
    -v $(pwd)/init.d:/init.d \
    ydbplatform/local-ydb:latest
```

When the container starts, two tables — `users` and `orders` — will be automatically created using the SQL script. The script runs only once after a successful server start.

Using pre-init scripts

Create a file `/preinit.d/01-set-env.sh` with the following contents:

```
# /preinit.d/01-set-env.sh
# This script will be sourced, so exported variables will be available
export YDB_DEFAULT_LOG_LEVEL=INFO
```

Mount the pre-init directory:

```
docker run -d \
    --name ydb-local \
    -v $(pwd)/preinit.d:/preinit.d \
    ydbplatform/local-ydb:latest
```

When the container starts, the `YDB_DEFAULT_LOG_LEVEL` environment variable will be exported with the value `INFO`. The script runs on every container start.

Restoring from a backup using init scripts

Create a file `/init.d/01-restore-backup.sh` with the following contents:

```
# /init.d/01-restore-backup.sh
#!/bin/bash
if [ -d "/backup" ] && [ -n "$(ls -A /backup)" ]; then
    /ydb -e grpc://localhost:2136 -d /local --no-discovery tools restore -p . -i /backup
fi
```

Mount the init directory, as well as the volume with your backup:

```
docker run -d \
    --name ydb-local \
    -v $(pwd)/init.d:/init.d \
    -v $(pwd)/backup:/backup \
    ydbplatform/local-ydb:latest
```

When the container starts, restoration from the `/backup` backup will occur in the `/local` database. The script runs only once after a successful server start.

Docker stop

To stop YDB in Docker, run the following command:

```
docker stop ydb-local
```

If the `--rm` flag was specified at startup, the container will be deleted after stopping.

Kill Docker container with YDB

To delete a Docker container with YDB, run the following command:

```
docker kill ydb-local
```

If you want to clean up the file system, delete your work directory using the `rm -rf ~/ydbd` command. This will permanently remove all data inside the local YDB cluster.

Recipes for working with YDB

This section of YDB documentation contains ready-to-use recipes for various aspects of interacting with YDB. They are grouped into the following categories:

- Code recipes using YDB SDK and frameworks
- YDB CLI recipes
- Topic-to-Table Data Transfer Recipes
- Recipes for streaming queries
- YQL recipes

Backup Collection Recipes

Step-by-step guides for common backup collection scenarios.

Getting Started

- Creating Your First Backup Collection - Create a collection, take backups, and monitor operations

Environment Configuration

- Setting Up Backups for Multiple Environments - Configure backups for development and production
- Backup Strategy for Microservices - Organize backups by service boundaries

External Storage

- Exporting Backups to External Storage - Export to S3 or filesystem for disaster recovery
- Importing and Restoring Backups - Restore from external storage

Maintenance

- Backup Maintenance and Cleanup - Manage backup lifecycle and storage
- Validating and Testing Backups - Verify backup integrity

See also

- Backup collections
- Backup and recovery guide
- CREATE BACKUP COLLECTION
- BACKUP
- RESTORE
- DROP BACKUP COLLECTION

Creating Your First Backup Collection

This guide walks you through creating a backup collection, taking your first backups, and monitoring backup operations.

Creating a backup collection

A backup collection is a schema object stored in the database schema. You create and manage collections using SQL statements, and browse them using schema navigation commands (like `ydb scheme ls`) since they appear as directories in the database structure.

Create a collection that includes the tables you want to back up together:

```
-- Create a collection for related tables
CREATE BACKUP COLLECTION production_backups
    ( TABLE orders
    , TABLE products
    , TABLE customers
    )
WITH ( STORAGE = 'cluster', INCREMENTAL_BACKUP_ENABLED = 'true' );
```

Taking backups

After creating the collection, take an initial full backup, then use incremental backups for subsequent operations:

```
-- Take initial full backup
BACKUP production_backups;

-- Make changes to your data...
-- INSERT, UPDATE, or DELETE operations on backed-up tables

-- Later, take incremental backups to capture the changes
BACKUP production_backups INCREMENTAL;
```

Note

Backup operations run asynchronously and are not idempotent — each `BACKUP` command creates a new backup. Before retrying after a timeout, check operation status with `ydb operation list incbackup`.

Monitoring backup operations

Track backup progress and browse your backup structure:

```
# Check backup operation status
ydb operation list incbackup

# Get details for specific operation
ydb operation get <operation-id>

# Browse backup collections
ydb scheme ls .backups/collections/

# List backups in a collection
ydb scheme ls .backups/collections/production_backups/
```

Next steps

- Setting Up Backups for Multiple Environments
- Exporting Backups to External Storage
- Validating and Testing Backups

Setting Up Backups for Multiple Environments

Configure different backup strategies for development and production environments.

Development environment

Use simpler configurations with fewer tables for testing:

```
-- Create collection with fewer tables for testing
CREATE BACKUP COLLECTION dev_test_backups
    ( TABLE users
    , TABLE test_data
    )
WITH ( STORAGE = 'cluster', INCREMENTAL_BACKUP_ENABLED = 'true' );

-- Take daily full backups in development
BACKUP dev_test_backups;
```

Production environment

Create comprehensive collections with more frequent incremental backups:

```
-- Create comprehensive collection for production
CREATE BACKUP COLLECTION prod_daily_backups
    ( TABLE orders
    , TABLE products
    , TABLE customers
    , TABLE inventory
    , TABLE transactions
    )
WITH ( STORAGE = 'cluster', INCREMENTAL_BACKUP_ENABLED = 'true' );

-- Weekly full backup
BACKUP prod_daily_backups;

-- Daily incremental backups
BACKUP prod_daily_backups INCREMENTAL;
```

Recommended backup schedules

Environment	Full Backup	Incremental Backup
Development	Daily	None
Staging	Weekly	Daily
Production	Weekly	Daily or hourly

Next steps

- Backup Strategy for Microservices
- Exporting Backups to External Storage

Backup Strategy for Microservices

Organize backup collections by service boundaries for independent backup and recovery operations.

Service-specific collections

Create separate backup collections for each microservice:

```
-- User service backup collection
CREATE BACKUP COLLECTION user_service_backups
    ( TABLE profiles
    , TABLE preferences
    , TABLE sessions
    )
WITH ( STORAGE = 'cluster', INCREMENTAL_BACKUP_ENABLED = 'true' );

-- Order service backup collection
CREATE BACKUP COLLECTION order_service_backups
    ( TABLE orders
    , TABLE order_items
    , TABLE payments
    )
WITH ( STORAGE = 'cluster', INCREMENTAL_BACKUP_ENABLED = 'true' );

-- Inventory service backup collection
CREATE BACKUP COLLECTION inventory_service_backups
    ( TABLE products
    , TABLE stock_levels
    , TABLE warehouses
    )
WITH ( STORAGE = 'cluster', INCREMENTAL_BACKUP_ENABLED = 'true' );
```

Taking backups independently

Back up each service on its own schedule:

```
-- Take backups for each service independently
BACKUP user_service_backups;
BACKUP order_service_backups;
BACKUP inventory_service_backups;

-- Later, take incremental backups
BACKUP user_service_backups INCREMENTAL;
BACKUP order_service_backups INCREMENTAL;
BACKUP inventory_service_backups INCREMENTAL;
```

Benefits of service-specific collections

- **Independent recovery**: Restore individual services without affecting others
- **Flexible scheduling**: Different backup frequencies per service based on criticality
- **Reduced blast radius**: Issues with one collection don't affect others
- **Team ownership**: Each team can manage their own backup strategy

Next steps

- Exporting Backups to External Storage
- Importing and Restoring Backups

Exporting Backups to External Storage

Export backup collections to S3-compatible storage or filesystem for disaster recovery.

Export to S3

For large backups or disaster recovery, export directly to S3-compatible storage:

```
# Export a backup collection to S3
ydb export s3 \
  --s3-endpoint storage.yandexcloud.net \
  --bucket my-backup-bucket \
  --item src=.backups/collections/production_backups,dst=backups/production_backups

# Export specific backup to S3
ydb export s3 \
  --s3-endpoint storage.yandexcloud.net \
  --bucket my-backup-bucket \
  --item src=.backups/collections/production_backups/20250821141425Z_full,dst=backups/20250821141425Z_full
```

Export to filesystem

For smaller backups or local testing, export to local filesystem:

```
# Export a backup collection
ydb tools dump -p .backups/collections/production_backups -o /backup/exports/production_backups_export

# Export specific backup from a collection
ydb tools dump -p .backups/collections/production_backups/20250601120000Z_full -o /backup/exports/backup_2025
0601
```

Export best practices

- **Export each backup separately**: Export full backups and incrementals individually to maintain chain integrity
- **Preserve chain order**: When exporting a chain, export the full backup first, then incrementals in order
- **Verify exports**: Check that exported data is complete before deleting cluster-stored backups
- **Schedule regular exports**: Automate exports to external storage for disaster recovery

Next steps

- Importing and Restoring Backups
- Backup Maintenance and Cleanup

Importing and Restoring Backups

Import backups from external storage and restore data to your database.

Importing from filesystem

Import previously exported backups:

```
# Import backup to target database
ydb tools restore -p .backups/collections/production_backups_restored -i /backup/exports/production_backups_e
xport

# Import specific backup into a collection
ydb tools restore -p .backups/collections/emergency_restore -i /backup/exports/backup_20250601
```

Importing from S3

```
ydb import s3 \
  --s3-endpoint storage.yandexcloud.net \
  --bucket my-backup-bucket \
  --item src=backups/production_backups,dst=.backups/collections/production_backups
```

Restoring data

After importing backups to the cluster, restore the data:

```
RESTORE production_backups;
```

Disaster recovery workflow

1. **Import full backup**: Import the base full backup first
2. **Import incrementals**: Import each incremental backup in order
3. **Execute RESTORE**: Run the RESTORE command to apply the backup chain

```
# Step 1: Import full backup
ydb tools restore -p .backups/collections/recovery/20250821141425Z_full -i /backup/full_20250821

# Step 2: Import incrementals in order
ydb tools restore -p .backups/collections/recovery/20250822070000Z_incremental -i /backup/inc_20250822

# Step 3: Restore
ydb yql -s "RESTORE recovery;"
```

Next steps

- Backup Maintenance and Cleanup
- Validating and Testing Backups

Backup Maintenance and Cleanup

Manage backup lifecycle and clean up old backups to control storage usage.

Identifying backup chains

Backup directories are named with timestamps and suffixes (`_full` or `_incremental`). Incremental backups belong to the most recent full backup that precedes them chronologically.

```
# List backups sorted by time
ydb scheme ls .backups/collections/production_backups/ | sort
```

Example output:

```
20250601120000Z_full        # Chain 1: full backup
20250601130000Z_incremental # Chain 1: incremental (belongs to 20250601 full)
20250602070000Z_incremental # Chain 1: incremental (belongs to 20250601 full)
20250608120000Z_full        # Chain 2: new full backup starts new chain
20250608130000Z_incremental # Chain 2: incremental (belongs to 20250608 full)
```

When you create a new full backup, it starts a new chain. All incrementals created after that full backup (until the next full) belong to that chain.

Manual cleanup

Remove old backup chains when they are no longer needed:

```
# Remove old backup directories
ydb scheme rmdir -r .backups/collections/production_backups/20250601120000Z_full/

# Always remove complete chains, never partial chains
# Example: Remove old full backup and all its incrementals
ydb scheme rmdir -r .backups/collections/production_backups/20250601120000Z_full/
ydb scheme rmdir -r .backups/collections/production_backups/20250601130000Z_incremental/
ydb scheme rmdir -r .backups/collections/production_backups/20250602070000Z_incremental/
```

> **Warning**
>
> Never delete individual backups from the middle of a chain. Deleting a full backup makes all its incremental backups unrestorable. Always delete complete chains together.

Dropping a collection

Remove an entire backup collection when it's no longer needed:

```
-- Drop a collection when no longer needed (removes collection and all backups)
DROP BACKUP COLLECTION old_collection_name;
```

Safe cleanup workflow

1. Create a new full backup
2. Verify the new backup is complete
3. Export old backup chains to external storage if needed
4. Delete old backup chains (full backup + all its incrementals together)

Storage management tips

- **Monitor storage growth**: Track backup storage usage regularly
- **Set retention policies**: Define how long to keep backup chains
- **Start new chains periodically**: Create new full backups weekly or bi-weekly to limit chain length
- **Export before deleting**: Always export to external storage before removing cluster backups

Next steps

- Validating and Testing Backups
- Exporting Backups to External Storage

Validating and Testing Backups

Verify backup integrity and test restoration procedures.

Verify backup completion

Check that backup operations completed successfully:

```
# List backup operations and check status
ydb operation list incbackup
```

Look for these status values in the output:

Status	Meaning
`SUCCESS`	Operation completed successfully
`PROGRESS`	Operation is still running
`CANCELLED`	Operation was cancelled (currently not applicable — cancel is not yet supported)
`ERROR`	Operation failed

```
# Get detailed information about a specific operation
ydb operation get <operation-id>

# Verify backup directory exists
ydb scheme ls .backups/collections/production_backups/ | tail -1
```

If an operation shows `ERROR` status, check the operation details for the error message and verify that all tables in the collection are accessible.

Verify external exports

Regularly verify that backups can be exported to external storage:

```
# Export to S3 and verify
ydb export s3 \
  --s3-endpoint storage.yandexcloud.net \
  --bucket my-backup-bucket \
  --item src=.backups/collections/production_backups/20250821141425Z_full,dst=backups/test_export

# Check export completed successfully
ydb operation list export/s3

# Verify exported data exists in S3 (using your S3 client)
# aws s3 ls s3://my-backup-bucket/backups/test_export/
```

Test backup restoration

Regularly test that backups can be restored:

```
# Export a recent backup for testing
ydb tools dump -p .backups/collections/production_backups/20250601120000Z_full -o /tmp/test_restore

# Test restore to temporary location (in test environment)
ydb tools restore -p /Root/test_restore_verification -i /tmp/test_restore
```

Validation checklist

- [ ] Backup operations complete without errors
- [ ] Backup directories are created with expected timestamps
- [ ] Exported backups can be imported successfully
- [ ] Restored data matches expected content
- [ ] Incremental chain integrity is maintained

Recommended testing schedule

Test Type	Frequency	Description
Operation status	Daily	Check that scheduled backups complete
Export verification	Weekly	Verify exports to external storage
Full restore test	Monthly	Complete restore to test environment
Disaster recovery drill	Quarterly	Full DR simulation

Setting up monitoring and alerts

YDB does not provide built-in backup monitoring or alerting. Implement external monitoring to detect backup failures:

Example monitoring script

```
#!/bin/bash
# Check for failed backup operations
FAILED=$(ydb operation list incbackup --format json | jq '[.operations[] | select(.status == "ERROR")] | length')

if [ "$FAILED" -gt 0 ]; then
  echo "ALERT: $FAILED backup operation(s) failed" | mail -s "YDB Backup Alert" admin@example.com
fi

# Check that a backup was created in the last 24 hours
LATEST=$(ydb scheme ls .backups/collections/production_backups/ | sort | tail -1)
# Add your own logic to parse timestamp and compare with current time
```

Monitoring recommendations

- Schedule monitoring scripts via cron to run after expected backup times
- Check both operation status (`ERROR`) and that new backups were created
- Integrate with your existing alerting system (email, Slack, PagerDuty, etc.)
- Monitor backup storage growth to avoid running out of space

Next steps

- Creating Your First Backup Collection
- Backup Maintenance and Cleanup

Recipes for system tablet backup

Step-by-step guides on enabling system tablet backup and restoring them.

Enabling and restoration

- Enabling and configuring backup — configuring `system_tablet_backup_config` and preparing storage
- Restoring system tablets — switching to Recovery mode and restoring from a local backup

See also

- System tablet backup — concepts and operating principles
- Backup and restoration — overview of backup methods
- system_tablet_backup_config — configuration parameters

Enabling and configuring backup of system tablets

Note

For conceptual information about the mechanism, see the System tablet backup section.

Warning

Only local file systems are supported as backup storage. Network file systems, such as NFS, are not supported and may cause backup corruption or tablet failures.

By default, backup of system tablets is **disabled** because it requires storage configuration. YDB nodes typically do not use a local file system, so enabling it requires an informed decision by the administrator.

To enable it, add the `system_tablet_backup_config` section to the cluster configuration:

```
system_tablet_backup_config:
    filesystem:
        path: "/path/to/backup/directory"
```

The `path` parameter is an absolute path to a directory on the host's local file system for storing backups.

Warning

The backup directory must be writable by the YDB process on each host where system tablets run. Ensure that each host's disk has enough free space to store several backups of all system tablets. For large production environments consisting of dozens of storage nodes, at least 5 GB is recommended.

After changing the configuration, restart the cluster nodes that may run system tablets. Backup will start automatically.

Warning

By default, system tablets can run on any static nodes of the cluster, including those whose disks are part of a static group. Backups are stored locally on the hosts where system tablets run. This means that if static group hosts are lost, the backups that may be needed to restore the static group can also be lost.

It is recommended to configure the `bootstrap_config` section in the cluster configuration so that system tablets run on nodes that are not part of the static group. In this case, backups will be stored separately from the static group data.

Recovering system tablets

Note

For conceptual information about the mechanism, see the System tablet backup section.

Warning

Recovering system tablets is a critical operation that may result in data loss. Perform it only if you have a clear understanding of the problem and after consulting with the operations team. Before starting, make sure you have reviewed all the steps.

Step 1. Put the tablet into Recovery mode

The tablet to be recovered must be put into Recovery mode. In this mode, the tablet starts and is accessible via the YDB UI, but **does not work normally** and **does not read data from the distributed storage**, allowing recovery operations to be performed. Other tablets will continue to operate normally, allowing the cluster to keep functioning, but some control-plane operations may be unavailable.

Warning

If recovery is performed after a complete loss of the static group, all system tablets must be put into Recovery mode simultaneously with recreating the static group on new hosts. If you recreate the static group without putting the tablets into Recovery mode, they will automatically start on top of an empty static group, leading to incorrect cluster operation.

1. Determine the ID of the system tablet to be recovered. The tablet ID can be found in the Tablets section of the YDB UI.
2. Determine the list of nodes where the system tablet to be recovered can run. This list is located in the `bootstrap_config` section of the corresponding tablet in the cluster configuration. If the `bootstrap_config` section is missing from the configuration, use the list of all static nodes of the cluster specified in the `hosts` section of the cluster configuration.
3. Modify the configuration by adding `boot_mode: RECOVERY` to the `bootstrap_config` section of the tablet being recovered.
   - When using configuration V1, you need to modify the static configuration on all nodes where the tablet being recovered can run.
   - When using configuration V2, follow the instructions.
   - Example for tablet `Hive` with ID `72057594037968897`:

```
bootstrap_config:
    tablet:
    - type: FLAT_HIVE
      node:
      - 1
      - 2
      - 3
      info:
          tablet_id: '72057594037968897'
          channels:
          - channel: 0
          history:
          - from_generation: 0
              group_id: 0
          channel_erasure_name: mirror-3-dc
          - channel: 1
          history:
          - from_generation: 0
              group_id: 0
          channel_erasure_name: mirror-3-dc
          - channel: 2
          history:
          - from_generation: 0
              group_id: 0
          channel_erasure_name: mirror-3-dc
      boot_mode: RECOVERY
```

4. Restart all nodes where the tablet being recovered can run. If any node is unavailable and cannot be restarted, isolate it from the cluster over the network — for example, using a firewall.

   Warning

   Make sure that all nodes where the tablet being recovered can run are restarted with the updated configuration or isolated from the cluster over the network before starting recovery. During recovery, old data is erased and replaced with data from the backup. If at that moment the tablet starts in normal mode on a node with the old configuration, it may start working with partially recovered data.

5. Make sure that:
   - There are no issues with the tablet in HealthCheck.
   - The tablet is not restarting.
   - The recovery form is available in the tablet's App in the YDB UI.

Step 2. Find the backup files

1. Determine which hosts to search. First, check the hosts where the tablet was running before the failure. These hosts can be identified using logs or a monitoring system.

If you cannot determine specific hosts, check all hosts where the tablet could have been running. This list is located in the `bootstrap_config` section of the corresponding tablet in the cluster configuration. If the `bootstrap_config` section is missing from the configuration, use the list of all static nodes of the cluster specified in the `hosts` section of the cluster configuration.

2. Find the directory with backups. On each candidate host, check for the presence of backups. The path to backups is determined by the `path` parameter in the `system_tablet_backup_config` configuration:

```
ls /path/to/backup/directory/<tablet_type>/<tablet_id>/
```

Example for tablet `Hive` with ID `72057594037968897`:

```
ls /tablet/hive/72057594037968897/
```

```
backup_20251007T181003_g213_s1001
backup_20251007T191002_g214_s1040
backup_20251007T193502_g214_s1222
```

3. Select the most recent backup. The name of each backup contains key information: `backup_<timestamp>_g<generation>_s<step>`, where:
   - `timestamp` — backup creation time.
   - — tablet `generation`, increases with each tablet restart.
   - `step` — tablet step within a generation, increases with each change in tablet state.

   Select the backup with the **maximum generation**, and if generations are equal, with the **maximum step**. If backups are found on multiple hosts, compare them and select the most recent one.
4. Make sure the backup is fully written. The selected backup must contain the `snapshot` directory, **not** `snapshot.tmp`. The presence of `snapshot.tmp` means that the snapshot write was not completed and the backup is not suitable for recovery. In this case, select the previous most recent backup.

```
ls /tablet/hive/72057594037968897/backup_20251007T193502_g214_s1222/snapshot/
```

```
manifest.json
schema.json
Tablet.json
TabletFollowerGroup.json
...
```

Step 3. Transfer the backup files

1. Determine which host the tablet is running on in Recovery mode. To do this, open the YDB UI and find the node where the tablet is running.
2. If the backup files are on a different host, copy them to the host with the tablet in Recovery mode using `scp`, `rsync`, or any other available tool:

```
scp -r /tablet/hive/72057594037968897/backup_20251007T193502_g214_s1222 \
    user@target-host:~/backup_20251007T193502_g214_s1222
```

3. Make sure the files are readable by the YDB process on the target host.

Step 4. Perform the recovery

1. Open the App of the tablet being restored in the YDB UI.
2. In the recovery form, specify the full path to the directory with the backup files, for example:

```
/tablet/hive/72057594037968897/backup_20251007T193502_g214_s1222
```

3. If necessary, set the flags:
   - **Dry Run** — performs a trial recovery without making changes to the storage. Allows you to verify the correctness of the backup.
   - **Skip Checksum Validation** — skips the checksum verification of the backup files.

> **Warning**
>
> Always perform a Dry Run before the first recovery to verify the correctness of the backup.

> **Warning**
>
> Skip checksum verification only if you have manually edited the backup files. In other cases, it is recommended to keep verification enabled to ensure the integrity of the restored data.

4. Click the **Start Restore** button. Before starting the recovery, the system will ask for confirmation, as the operation overwrites the existing tablet data. Confirm the action to start. After the recovery starts, the form becomes unavailable — a restart is not possible until the tablet is restarted.

> **Note**
>
> You can also start the recovery using `curl` by sending a POST request to the tablet's App page with the `restoreBackup` parameter:
>
> ```
> curl -X POST "http://<host>:<mon_port>/tablets/app?TabletID=72057594037968897&restoreBackup=/tablet/hive/72057594037968897/backup_20251007T193502_g214_s1222"
> ```
>
> Where `<host>` is the cluster node address, `<mon_port>` is the monitoring port of this node.

5. Monitor the recovery progress. The page **does not update automatically** — to get the current status, refresh the page manually. The recovery duration depends on the backup size.

   > **Note**
   >
   > The recovery is performed server-side and does not depend on the browser. You can close the page or browser — this **will not interrupt** the recovery process. When you reopen the page, the current status will be displayed.

   The current operation status is displayed below the form. Possible statuses:

   - `Restoring from '<path>'` — recovery is in progress, data from the backup is being read and written to the storage. Additionally, a **progress bar** with the operation completion percentage is displayed.
   - `Restore from '<path>' completed successfully` — recovery completed successfully. You can proceed to the next step.
   - `Restore from '<path>' completed, but changelog is not fully restored` — the main tablet data has been restored, but the tail of the change log in the backup is damaged, and some recent changes are lost. Examine which data was lost and, if necessary, restore it manually, or proceed to the next step.
   - `Restore from '<path>' failed: <error description>` — recovery completed with an error. Examine the error description and retry if necessary.

   To forcibly interrupt the recovery, **restart the tablet**.
6. Wait for the recovery to complete successfully. If the recovery form becomes available again (the **Start Restore** button is active, the status is reset), this means the tablet was restarted and the recovery was interrupted. In this case, start the recovery again from step 2.

Step 5. Return the tablet to normal operation mode

After successful recovery:

1. Determine the list of nodes on which the system tablet being recovered can run. This list is located in the `bootstrap_config` section of the corresponding tablet in the cluster configuration. If the `bootstrap_config` section is absent from the configuration, use the list of all static nodes of the cluster specified in the `hosts` section of the cluster configuration.
2. Modify the configuration by removing `boot_mode: RECOVERY` from the `bootstrap_config` section of the tablet being recovered.
   - When using configuration V1, you need to change the static configuration on all nodes on which the tablet being recovered can run.
   - When using configuration V2, follow the instructions.
3. Restart all nodes on which the tablet being recovered can run. If any nodes were isolated from the cluster over the network in previous steps, remove the network isolation.
4. Make sure that:
   - There are no issues with the tablet in HealthCheck.
   - The tablet does not restart.
   - The recovery form is absent in the tablet's App in YDB UI.

Code recipes using YDB SDK and frameworks

This section contains code recipes in different programming languages for solving various common practical tasks using the YDB SDK.

Contents:

- Driver initialization
- Authentication
  - Using a token
  - Anonymous
  - Service account file
  - Metadata service
  - Using environment variables
  - Using login and password
- Load balancing
  - Uniform random selection
  - Prefer nearest datacenter
  - Prefer availability zone
- Retry query execution
- Set session pool size
- Data insert
- Batch data insert
- Setting transaction execution mode
- Configuring table TTL (Time to Live)
- Vector search
- Coordination
  - Distributed lock
  - Service discovery
  - Configuration publishing
  - Leader election
- Problem diagnostics
  - Enable logging
  - Connect metrics to Prometheus
  - Tracing with OpenTelemetry

See also:

- YDB for Application Developers / Software Engineers
- Example applications working with YDB
- Reference for YDB SDK

Initialize the driver

To connect to YDB, you must specify the required parameters (see Connecting to a YDB server) and optional parameters that control driver behavior.

Below are examples of connecting to YDB (creating a driver) in different YDB SDKs.

Go

Native SDK

```
package main

import (
  "context"

  "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()

  db, err := ydb.Open(ctx, "grpc://localhost:2136/local")
  if err != nil {
      panic(err)
  }
  defer db.Close()

  // ...
}
```

database/sql

Using a connector (recommended)

```
package main

import (
  "context"
  "database/sql"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()

  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  defer connector.Close()

  db := sql.OpenDB(connector)
  defer db.Close()

  // ...
}
```

Using a connection string

The `database/sql` driver is registered when you import the driver package with a blank import:

```
package main

import (
  "database/sql"

  _ "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  db, err := sql.Open("ydb", "grpc://localhost:2136/local")
  if err != nil {
    panic(err)
  }
  defer db.Close()
```

```
  // ...
}
```

Java

Native SDK

```
public void work() {
    try (GrpcTransport transport = GrpcTransport.forConnectionString("grpc://localhost:2136/local")
            .build()) {
        // Work with transport
        doWork(transport);
    }
}
```

JDBC

```
public void work() throws SQLException {
    // tech.ydb.jdbc.YdbDriver must be on the classpath for DriverManager auto-loading
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local")) {
        // Work with connection
        doWork(connection);
    }
}
```

For Spring Boot, set the URL and driver class in `application.properties` or `application.yml` (`spring.datasource.url`, `spring.datasource.driver-class-name`).

Spring Boot, ORMs, and other JDBC stacks (Hibernate, JOOQ, MyBatis, and so on) talk to YDB like any JDBC source: add the YDB JDBC dependency and configure the URL — you do not need to configure native `GrpcTransport` separately.

Python

Native SDK

```
import ydb

with ydb.Driver(connection_string="grpc://localhost:2136?database=/local") as driver:
  driver.wait(timeout=5)
  ...
```

Native SDK (Asyncio)

```
import ydb
import asyncio

async def ydb_init():
  async with ydb.aio.Driver(endpoint="grpc://localhost:2136", database="/local") as driver:
    await driver.wait()
    ...

asyncio.run(ydb_init())
```

C# (.NET)

```
using Ydb.Sdk;

var config = new DriverConfig(
    endpoint: "grpc://localhost:2136",
    database: "/local"
);

await using var driver = await Driver.CreateInitialized(config);
```

JavaScript

```
import { Driver } from '@ydbjs/core'

const driver = new Driver('grpc://localhost:2136/local')
await driver.ready()
```

PHP

```
<?php

use YdbPlatform\Ydb\Ydb;

$config = [
    // Database path
    'database'    => '/ru-central1/b1glxxxxxxxxxxxxxxx/etn0xxxxxxxxxxxxxxx',

    // Database endpoint
```

```
    'endpoint'    => 'ydb.serverless.yandexcloud.net:2135',

    // Auto discovery (dedicated server only)
    'discovery'   => false,

    // IAM config
    'iam_config'  => [
        // 'root_cert_file' => './CA.pem', // Root CA file (uncomment for dedicated server)
    ],

    'credentials' => new \YdbPlatform\Ydb\Auth\Implement\AccessTokenAuthentication('AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA') // use from reference/ydb-sdk/auth
];

$ydb = new Ydb($config);
```

Rust

```
use ydb::{AccessTokenCredentials, ClientBuilder, YdbResult};

#[tokio::main]
async fn main() -> YdbResult<()> {
    let client = ClientBuilder::new_from_connection_string("grpc://localhost:2136/local")?
        .with_credentials(AccessTokenCredentials::from("..."))
        .client()?;
    client.wait().await?;
    Ok(())
}
```

Authentication

YDB supports multiple authentication methods when connecting to the server side. Each of them is usually specific to a particular pair of environments, that is, depends on where you run your client application (in the trusted YDB zone or outside it) and the YDB server part (in a Docker container, Yandex.Cloud, data cloud, or an independent cluster).

This section contains code recipes with authentication settings in different YDB SDKs. For a general description of the SDK authentication principles, see the Authentication in an SDK.

Table of contents:

- Using a token
- Anonymous
- Service account file
- Metadata service
- Using environment variables
- Username and password based

Authentication using a token

Below are examples of authentication with a token in different YDB SDKs.

Go

Native SDK

```
package main

import (
  "context"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  ...
}
```

database/sql

If you use a connector to create a connection to YDB

```
package main

import (
  "context"
  "database/sql"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)
  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  db := sql.OpenDB(connector)
  defer db.Close()
  ...
}
```

If you use a connection string

```
package main

import (
  "context"
  "database/sql"
  "os"

  _ "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  db, err := sql.Open("ydb", "grpcs://localhost:2135/local?token="+os.Getenv("YDB_TOKEN"))
  if err != nil {
    panic(err)
  }
  defer db.Close()
  ...
}
```

Java

Native SDK

```
public void work(String accessToken) {
    AuthProvider authProvider = new TokenAuthProvider(accessToken);

    try (GrpcTransport transport = GrpcTransport.forConnectionString("grpcs://localhost:2135/local")
            .withAuthProvider(authProvider)
            .build();
         QueryClient queryClient = QueryClient.newClient(transport).build()) {

        doWork(queryClient);
    }
}
```

JDBC

```
public void work() throws SQLException {
    // Connection with an explicit authentication token value
    Properties props1 = new Properties();
    props1.setProperty("token", "AQAD-XXXXXXXXXXXXXXXXXXX");
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local", props1))
{
        doWork(connection);
    }

    // Connection with the token read from the specified file
    Properties props2 = new Properties();
    props2.setProperty("tokenFile", "~/.ydb_token");
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local", props2))
{
        doWork(connection);
    }
}
```

In Spring Boot, ORMs, and other JDBC wrappers, use the same JDBC URL and authentication parameters as above (for example `spring.datasource.url` with query parameters or `spring.datasource.*` for the token and token file).

JavaScript

```
import { Driver } from "@ydbjs/core";
import { AccessTokenCredentialsProvider } from "@ydbjs/auth/access-token";

const driver = new Driver("grpc://localhost:2136/local", {
  credentialsProvider: new AccessTokenCredentialsProvider({
    token: "accessToken",
  }),
});

await driver.ready();
```

Python

Native SDK

```
import os
import ydb

with ydb.Driver(
    connection_string=os.environ["YDB_CONNECTION_STRING"],
    credentials=ydb.credentials.AccessTokenCredentials(os.environ["YDB_TOKEN"]),
) as driver:
    driver.wait(timeout=5)
    ...
```

Native SDK (Asyncio)

```
import os
import ydb
import asyncio

async def ydb_init():
    async with ydb.aio.Driver(
        endpoint=os.environ["YDB_ENDPOINT"],
        database=os.environ["YDB_DATABASE"],
        credentials=ydb.credentials.AccessTokenCredentials(os.environ["YDB_TOKEN"]),
    ) as driver:
        await driver.wait()
        ...

asyncio.run(ydb_init())
```

SQLAlchemy

```
import os
import sqlalchemy as sa

engine = sa.create_engine(
    "yql+ydb://localhost:2136/local",
    connect_args={
        "credentials": {"token": os.environ["YDB_TOKEN"]}
    }
)
with engine.connect() as connection:
    result = connection.execute(sa.text("SELECT 1"))
```

C# (.NET)

```
using Ydb.Sdk;
using Ydb.Sdk.Auth;

const string endpoint = "grpc://localhost:2136";
const string database = "/local";
const string token = "MY_VERY_SECURE_TOKEN";

var config = new DriverConfig(
    endpoint: endpoint,
    database: database,
    credentials: new TokenProvider(token)
);

await using var driver = await Driver.CreateInitialized(config);
```

PHP

```
<?php

use YdbPlatform\Ydb\Ydb;
use YdbPlatform\Ydb\Auth\Implement\AccessTokenAuthentication;

$config = [

    // Database path
    'database'    => '/local',

    // Database endpoint
    'endpoint'    => 'localhost:2136',

    // Auto discovery (dedicated server only)
    'discovery'   => false,

    // IAM config
    'iam_config'  => [
        'insecure' => true,
        // 'root_cert_file' => './CA.pem', // Root CA file (uncomment for dedicated server)
    ],

    'credentials' => new AccessTokenAuthentication('AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA')
];

$ydb = new Ydb($config);
```

Anonymous authentication

Below are examples of anonymous authentication in different YDB SDKs.

Go

Native SDK

Anonymous authentication is the default.
You can enable it explicitly as follows:

```
package main

import (
  "context"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAnonymousCredentials(),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  ...
}
```

database/sql

Anonymous authentication is the default.
You can enable it explicitly as follows:

```
package main

import (
  "context"
  "database/sql"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAnonymousCredentials(),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)
  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  db := sql.OpenDB(connector)
  defer db.Close()
  ...
}
```

Java

Native SDK

```
public void work(String connectionString) {
    AuthProvider authProvider = NopAuthProvider.INSTANCE;

    try (GrpcTransport transport = GrpcTransport.forConnectionString(connectionString)
            .withAuthProvider(authProvider)
            .build();
         QueryClient queryClient = QueryClient.newClient(transport).build()) {

        doWork(queryClient);
    }
}
```

JDBC

```
public void work() throws SQLException {
    // Connection with no extra options — anonymous authentication
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local")) {
        doWork(connection);
    }
}
```

In Spring Boot, ORMs, and other JDBC wrappers, use the same JDBC URL as above (for example `spring.datasource.url`).

JavaScript

```
import { Driver } from "@ydbjs/core";
import { AnonymousCredentialsProvider } from "@ydbjs/auth/anonymous";

const driver = new Driver("grpc://localhost:2136/local", {
  credentialsProvider: new AnonymousCredentialsProvider(),
});

await driver.ready();
```

Python

Native SDK

```
import os
import ydb

with ydb.Driver(
    connection_string=os.environ["YDB_CONNECTION_STRING"],
    credentials=ydb.credentials.AnonymousCredentials(),
) as driver:
    driver.wait(timeout=5)
    ...
```

Native SDK (Asyncio)

```
import os
import ydb
import asyncio

async def ydb_init():
    async with ydb.aio.Driver(
        endpoint=os.environ["YDB_ENDPOINT"],
        database=os.environ["YDB_DATABASE"],
        credentials=ydb.credentials.AnonymousCredentials(),
    ) as driver:
        await driver.wait()
        ...

asyncio.run(ydb_init())
```

SQLAlchemy

```
import sqlalchemy as sa

engine = sa.create_engine("yql+ydb://localhost:2136/local")
with engine.connect() as connection:
    result = connection.execute(sa.text("SELECT 1"))
```

C# (.NET)

```
using Ydb.Sdk;
using Ydb.Sdk.Auth;

const string endpoint = "grpc://localhost:2136";
const string database = "/local";

var config = new DriverConfig(
    endpoint: endpoint,
    database: database,
    credentials: new AnonymousProvider()
);

await using var driver = await Driver.CreateInitialized(config);
```

PHP

```
<?php

use YdbPlatform\Ydb\Ydb;
use YdbPlatform\Ydb\Auth\Implement\AnonymousAuthentication;
```

```php
$config = [

    // Database path
    'database'    => '/local',

    // Database endpoint
    'endpoint'    => 'localhost:2136',

    // Auto discovery (dedicated server only)
    'discovery'   => false,

    // IAM config
    'iam_config'  => [
        'insecure' => true,
        // 'root_cert_file' => './CA.pem', // Root CA file (uncomment for dedicated server)
    ],

    'credentials' => new AnonymousAuthentication()
];

$ydb = new Ydb($config);
```

Authentication using a service account file

Below are examples of authentication with a service account file in different YDB SDKs.

Go

Native SDK

```
package main

import (
  "context"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  yc "github.com/ydb-platform/ydb-go-yc"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    yc.WithServiceAccountKeyFileCredentials(
      os.Getenv("YDB_SERVICE_ACCOUNT_KEY_FILE_CREDENTIALS"),
    ),
    yc.WithInternalCA(), // append Yandex Cloud certificates
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  ...
}
```

database/sql

```
package main

import (
  "context"
  "database/sql"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  yc "github.com/ydb-platform/ydb-go-yc"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    yc.WithServiceAccountKeyFileCredentials(
      os.Getenv("YDB_SERVICE_ACCOUNT_KEY_FILE_CREDENTIALS"),
    ),
    yc.WithInternalCA(), // append Yandex Cloud certificates
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)
  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  db := sql.OpenDB(connector)
  defer db.Close()
  ...
}
```

Java

Native SDK

```
public void work(String connectionString, String saKeyPath) {
    AuthProvider authProvider = CloudAuthHelper.getServiceAccountFileAuthProvider(saKeyPath);

    try (GrpcTransport transport = GrpcTransport.forConnectionString(connectionString)
            .withAuthProvider(authProvider)
            .build();
         QueryClient queryClient = QueryClient.newClient(transport).build()) {

        doWork(queryClient);
    }
}
```

JDBC

```
public void work() throws SQLException {
    Properties props = new Properties();
    props.setProperty("saKeyFile", "~/keys/sa_key.json");
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local", props))
{
        doWork(connection);
    }

    // You can also set saKeyFile in the JDBC URL
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local?saKeyFile=
~/keys/sa_key.json")) {
        doWork(connection);
    }
}
```

In Spring Boot, ORMs, and other JDBC wrappers, use the same JDBC URL and `saKeyFile` (in the URL or in `DataSource` properties) as above.

JavaScript

Loading service account data from a file:

```
import { Driver } from "@ydbjs/core";
import { ServiceAccountCredentialsProvider } from "@ydbjs/auth-yandex-cloud";

const driver = new Driver("grpc://localhost:2136/local", {
  credentialsProvider: ServiceAccountCredentialsProvider.fromFile("./authorized_key.json"),
});

await driver.ready();
```

Loading service account data from a third-party source (for example, a secrets store):

```
import { Driver } from "@ydbjs/core";
import { ServiceAccountCredentialsProvider } from "@ydbjs/auth-yandex-cloud";

const driver = new Driver("grpc://localhost:2136/local", {
  credentialsProvider: new ServiceAccountCredentialsProvider({
    id: "serviceAccountId",
    keyId: "accessKeyId",
    privateKey: "-----BEGIN PRIVATE KEY-----\n...",
  }),
});

await driver.ready();
```

Python

Native SDK

```
import os
import ydb
import ydb.iam

with ydb.Driver(
    connection_string=os.environ["YDB_CONNECTION_STRING"],
    # service account key should be in the local file,
    # and SA_KEY_FILE environment variable should point to it
    credentials=ydb.iam.ServiceAccountCredentials.from_file(os.environ["SA_KEY_FILE"]),
) as driver:
    driver.wait(timeout=5)
    ...
```

Native SDK (Asyncio)

```
import os
import asyncio
import ydb
import ydb.iam

async def ydb_init():
    async with ydb.aio.Driver(
        endpoint=os.environ["YDB_ENDPOINT"],
        database=os.environ["YDB_DATABASE"],
        # service account key should be in the local file,
        # and SA_KEY_FILE environment variable should point to it
        credentials=ydb.iam.ServiceAccountCredentials.from_file(os.environ["SA_KEY_FILE"]),
    ) as driver:
        await driver.wait()
        ...

asyncio.run(ydb_init())
```

SQLAlchemy

```
import os
import sqlalchemy as sa
import ydb.iam

engine = sa.create_engine(
    "yql+ydb://localhost:2136/local",
    connect_args={
        "credentials": ydb.iam.ServiceAccountCredentials.from_file(
            os.environ["YDB_SERVICE_ACCOUNT_KEY_FILE_CREDENTIALS"]
        )
    }
)
with engine.connect() as connection:
    result = connection.execute(sa.text("SELECT 1"))
```

C# (.NET)

```
using Ydb.Sdk;
using Ydb.Sdk.Yc;

const string endpoint = "grpc://localhost:2136";
const string database = "/local";

var saProvider = new ServiceAccountProvider(
    saFilePath: "path/to/sa_file.json" // Path to file with service account JSON info);
);
await saProvider.Initialize();

var config = new DriverConfig(
    endpoint: endpoint,
    database: database,
    credentials: saProvider
);

await using var driver = await Driver.CreateInitialized(config);
```

PHP

```
<?php

use YdbPlatform\Ydb\Ydb;
use YdbPlatform\Ydb\Auth\JwtWithJsonAuthentication;

$config = [
    'database'    => '/ru-central1/b1glxxxxxxxxxxxxxxx/etn0xxxxxxxxxxxxxxx',
    'endpoint'    => 'ydb.serverless.yandexcloud.net:2135',
    'discovery'   => false,
    'iam_config'  => [
        'temp_dir'       => './tmp', // Temp directory
        // 'root_cert_file' => './CA.pem', // Root CA file (uncomment for dedicated server)ы
    ],

    'credentials' => new JwtWithJsonAuthentication('./jwtjson.json')
];

$ydb = new Ydb($config);
```

or

```
<?php

use YdbPlatform\Ydb\Ydb;
use YdbPlatform\Ydb\Auth\JwtWithPrivateKeyAuthentication;

$config = [
    'database'    => '/ru-central1/b1glxxxxxxxxxxxxxxx/etn0xxxxxxxxxxxxxxx',
    'endpoint'    => 'ydb.serverless.yandexcloud.net:2135',
    'discovery'   => false,
    'iam_config'  => [
        'temp_dir'           => './tmp', // Temp directory
        // 'root_cert_file' => './CA.pem', // Root CA file (uncomment for dedicated server)

    ],

    'credentials' => new JwtWithPrivateKeyAuthentication(
        "ajexxxxxxxxx","ajeyyyyyyyyy",'./private.key')

];

$ydb = new Ydb($config);
```

Authentication using the metadata service

Below are examples of authentication with the metadata service in different YDB SDKs.

Go

Native SDK

```
package main

import (
  "context"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  yc "github.com/ydb-platform/ydb-go-yc-metadata"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    yc.WithCredentials(),
    yc.WithInternalCA(), // append Yandex Cloud certificates
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  ...
}
```

database/sql

```
package main

import (
  "context"
  "database/sql"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  yc "github.com/ydb-platform/ydb-go-yc-metadata"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    yc.WithCredentials(),
    yc.WithInternalCA(), // append Yandex Cloud certificates
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)
  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  db := sql.OpenDB(connector)
  defer db.Close()
  ...
}
```

Java

Native SDK

```
public void work(String connectionString) {
    AuthProvider authProvider = CloudAuthHelper.getMetadataAuthProvider();

    try (GrpcTransport transport = GrpcTransport.forConnectionString(connectionString)
            .withAuthProvider(authProvider)
            .build();
         QueryClient queryClient = QueryClient.newClient(transport).build()) {

        doWork(queryClient);
    }
}
```

JDBC

```
public void work() throws SQLException {
    Properties props = new Properties();
    props.setProperty("useMetadata", "true");
```

```
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local", props))
{
        doWork(connection);
    }

    // You can also set useMetadata in the JDBC URL
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local?useMetadat
a=true")) {
        doWork(connection);
    }
}
```

In Spring Boot, ORMs, and other JDBC wrappers, pass the same JDBC URLs and parameters (`useMetadata` in the URL or in the data source properties) as in the example above.

JavaScript

```
import { Driver } from "@ydbjs/core";
import { MetadataCredentialsProvider } from "@ydbjs/auth/metadata";

const driver = new Driver("grpc://localhost:2136/local", {
  credentialsProvider: new MetadataCredentialsProvider(),
});

await driver.ready();
```

Python

Native SDK

```
import os
import ydb
import ydb.iam

with ydb.Driver(
    connection_string=os.environ["YDB_CONNECTION_STRING"],
    credentials=ydb.iam.MetadataUrlCredentials(),
) as driver:
    driver.wait(timeout=5)
    ...
```

Native SDK (Asyncio)

```
import os
import ydb
import ydb.iam
import asyncio

async def ydb_init():
    async with ydb.aio.Driver(
        endpoint=os.environ["YDB_ENDPOINT"],
        database=os.environ["YDB_DATABASE"],
        credentials=ydb.iam.MetadataUrlCredentials(),
    ) as driver:
        await driver.wait()
        ...

asyncio.run(ydb_init())
```

SQLAlchemy

```
import sqlalchemy as sa
import ydb.iam

engine = sa.create_engine(
    "yql+ydb://localhost:2136/local",
    connect_args={
        "credentials": ydb.iam.MetadataUrlCredentials()
    }
)
with engine.connect() as connection:
    result = connection.execute(sa.text("SELECT 1"))
```

C# (.NET)

```
using Ydb.Sdk;
using Ydb.Sdk.Yc;

var metadataProvider = new MetadataProvider();

// Await initial IAM token.
await metadataProvider.Initialize();

var config = new DriverConfig(
    endpoint: endpoint, // Database endpoint, "grpcs://host:port"
    database: database, // Full database path
```

```
    credentials: metadataProvider
);

await using var driver = await Driver.CreateInitialized(config);
```

PHP

```php
<?php

use YdbPlatform\Ydb\Ydb;
use YdbPlatform\Ydb\Auth\Implement\MetadataAuthentication;

$config = [

    // Database path
    'database'    => '/local',

    // Database endpoint
    'endpoint'    => 'localhost:2136',

    // Auto discovery (dedicated server only)
    'discovery'   => false,

    // IAM config
    'iam_config'  => [
        'insecure' => true,
        // 'root_cert_file' => './CA.pem', // Root CA file (uncomment for dedicated server)
    ],

    'credentials' => new MetadataAuthentication()
];

$ydb = new Ydb($config);
```

Authentication using environment variables

When using this method, the authentication mode and its parameters are defined by the environment that an application is run in, as described here.

By setting one of the following environment variables, you can control the authentication method:

- `YDB_SERVICE_ACCOUNT_KEY_FILE_CREDENTIALS=<path/to/sa_key_file>` — use a service account key file in Yandex Cloud.
- `YDB_ANONYMOUS_CREDENTIALS="1"` — use anonymous authentication. Useful for testing against a Docker container with YDB.
- `YDB_METADATA_CREDENTIALS="1"` — use the metadata service inside Yandex Cloud (Yandex Cloud Function or VM).
- `YDB_ACCESS_TOKEN_CREDENTIALS=<access_token>` — use token-based authentication.

Below are examples of authentication using environment variables in different YDB SDKs.

Go

Native SDK

```
package main

import (
  "context"
  "os"

  environ "github.com/ydb-platform/ydb-go-sdk-auth-environ"
  "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    environ.WithEnvironCredentials(ctx),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  ...
}
```

database/sql

```
package main

import (
  "context"
  "database/sql"
  "os"

  environ "github.com/ydb-platform/ydb-go-sdk-auth-environ"
  "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    environ.WithEnvironCredentials(ctx),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)
  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  db := sql.OpenDB(connector)
  defer db.Close()
  ...
}
```

Java

Native SDK

```
public void work(String connectionString) {
    AuthProvider authProvider = new EnvironAuthProvider();

    try (GrpcTransport transport = GrpcTransport.forConnectionString(connectionString)
            .withAuthProvider(authProvider)
            .build();
         QueryClient queryClient = QueryClient.newClient(transport).build()) {

        doWork(queryClient);
```

```
    }
}
```

JDBC

```
public void work() throws SQLException {
    // No explicit credentials: the driver reads YDB_* environment variables in the order
    // described in [Authentication](../../reference/ydb-sdk/auth.md#env)
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local", new Prop
erties())) {
        doWork(connection);
    }
}
```

In Spring Boot, ORMs, and other JDBC wrappers, use the same JDBC URL; credentials from the environment are picked up the same way as in the example above (for example via `spring.datasource.url`).

JavaScript

```
  import { Driver, getCredentialsFromEnv } from 'ydb-sdk';

  export async function connect(endpoint: string, database: string) {
      const authService = getCredentialsFromEnv();
      const driver = new Driver({endpoint, database, authService});
      const timeout = 10000;
      if (!await driver.ready(timeout)) {
          console.log(`Driver has not become ready in ${timeout}ms!`);
          process.exit(1);
      }
      console.log('Driver connected')
      return driver
  }
```

Python

Native SDK

```
import os
import ydb

with ydb.Driver(
    connection_string=os.environ["YDB_CONNECTION_STRING"],
    credentials=ydb.credentials_from_env_variables(),
) as driver:
    driver.wait(timeout=5)
    ...
```

Native SDK (Asyncio)

```
import os
import ydb
import asyncio

async def ydb_init():
    async with ydb.aio.Driver(
        endpoint=os.environ["YDB_ENDPOINT"],
        database=os.environ["YDB_DATABASE"],
        credentials=ydb.credentials_from_env_variables(),
    ) as driver:
        await driver.wait()
        ...

asyncio.run(ydb_init())
```

SQLAlchemy

```
import os
import sqlalchemy as sa
import ydb

engine = sa.create_engine(
    "yql+ydb://localhost:2136/local",
    connect_args={
        "credentials": ydb.credentials_from_env_variables()
    }
)
with engine.connect() as connection:
    result = connection.execute(sa.text("SELECT 1"))
```

PHP

```
<?php

use YdbPlatform\Ydb\Ydb;
use YdbPlatform\Ydb\Auth\EnvironCredentials;
```

```php
$config = [

    // Database path
    'database'    => '/local',

    // Database endpoint
    'endpoint'    => 'localhost:2136',

    // Auto discovery (dedicated server only)
    'discovery'   => false,

    // IAM config
    'iam_config'  => [
        'insecure' => true,
        // 'root_cert_file' => './CA.pem', // Root CA file (uncomment for dedicated server)
    ],

    'credentials' => new EnvironCredentials()
];

$ydb = new Ydb($config);
```

Username and password based authentication

Below are examples of authentication with a username and password in different YDB SDKs.

C++

```
auto driverConfig = NYdb::TDriverConfig()
  .SetEndpoint(endpoint)
  .SetDatabase(database)
  .SetCredentialsProviderFactory(NYdb::CreateLoginCredentialsProviderFactory({
      .User = "user",
      .Password = "password",
  }));

NYdb::TDriver driver(driverConfig);
```

Go

Native SDK

You can pass the username and password in the connection string. For example:

```
"grpcs://login:password@localhost:2135/local"
```

You can also pass them explicitly using the `ydb.WithStaticCredentials` option:

```
package main

import (
  "context"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
      os.Getenv("YDB_CONNECTION_STRING"),
      ydb.WithStaticCredentials("user", "password"),
  )
  if err != nil {
      panic(err)
  }
  defer db.Close(ctx)
  ...
}
```

database/sql

You can pass the username and password in the connection string. For example:

```
package main

import (
  "context"

  _ "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  db, err := sql.Open("ydb", "grpcs://login:password@localohost:2135/local")
  if err != nil {
      panic(err)
  }
  defer db.Close()
  ...
}
```

You can also pass them explicitly when initializing the driver via a connector using the `ydb.WithStaticCredentials` option:

```
package main

import (
  "context"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
      os.Getenv("YDB_CONNECTION_STRING"),
      ydb.WithStaticCredentials("user", "password"),
```

```
  )
  if err != nil {
      panic(err)
  }
  defer nativeDriver.Close(ctx)
  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  db := sql.OpenDB(connector)
  defer db.Close()
  ...
}
```

Java

Native SDK

```
public void work(String connectionString, String username, String password) {
    StaticCredentials authProvider = new StaticCredentials(username, password);

    try (GrpcTransport transport = GrpcTransport.forConnectionString(connectionString)
            .withAuthProvider(authProvider)
            .build();
         QueryClient queryClient = QueryClient.newClient(transport).build()) {

        doWork(queryClient);
    }
}
```

JDBC

```
public void work(String username, String password) throws SQLException {
    Properties props = new Properties();
    props.setProperty("username", username);
    props.setProperty("password", password);
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local", props))
{
        doWork(connection);
    }

    // Username and password can be passed directly
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local", usernam
e, password)) {
        doWork(connection);
    }
}
```

In Spring Boot, ORMs, and other JDBC wrappers, use the same JDBC URL, username, and password as above (for example `spring.datasource.url`, `spring.datasource.username`, `spring.datasource.password`, or the pool's equivalent settings).

JavaScript

```
import { Driver } from "@ydbjs/core";
import { StaticCredentialsProvider } from "@ydbjs/auth/static";

const driver = new Driver("grpc://localhost:2136/local", {
  credentialsProvider: new StaticCredentialsProvider(
    { username: user, password: password },
    "grpc://localhost:2136",
  ),
});

await driver.ready();
```

Python

Native SDK

```
import os
import ydb

config = ydb.DriverConfig(
    endpoint=os.environ["YDB_ENDPOINT"],
    database=os.environ["YDB_DATABASE"],
)

credentials = ydb.StaticCredentials(
    driver_config=config,
    user=os.environ["YDB_USER"],
    password=os.environ["YDB_PASSWORD"]
)

with ydb.Driver(driver_config=config, credentials=credentials) as driver:
    driver.wait(timeout=5)
    ...
```

Native SDK (Asyncio)

```
import os
import ydb
import asyncio

config = ydb.DriverConfig(
    endpoint=os.environ["YDB_ENDPOINT"],
    database=os.environ["YDB_DATABASE"],
)

credentials = ydb.StaticCredentials(
    driver_config=config,
    user=os.environ["YDB_USER"],
    password=os.environ["YDB_PASSWORD"],
)

async def ydb_init():
    async with ydb.aio.Driver(driver_config=config, credentials=credentials) as driver:
        await driver.wait()
        ...

asyncio.run(ydb_init())
```

SQLAlchemy

```
import os
import sqlalchemy as sa

engine = sa.create_engine(
    "yql+ydb://localhost:2136/local",
    connect_args={
        "credentials": {
            "username": os.environ["YDB_USER"],
            "password": os.environ["YDB_PASSWORD"]
        }
    }
)
with engine.connect() as connection:
    result = connection.execute(sa.text("SELECT 1"))
```

C# (.NET)

```
using Ydb.Sdk;
using Ydb.Sdk.Auth;

const string endpoint = "grpc://localhost:2136";
const string database = "/local";

var config = new DriverConfig(
    endpoint: endpoint, // Database endpoint, "grpcs://host:port"
    database: database, // Full database path
    credentials: new StaticCredentialsProvider(user, password)
);

await using var driver = await Driver.CreateInitialized(config);
```

PHP

```
<?php

use YdbPlatform\Ydb\Ydb;
use YdbPlatform\Ydb\Auth\Implement\StaticAuthentication;

$config = [

    // Database path
    'database'    => '/local',

    // Database endpoint
    'endpoint'    => 'localhost:2136',

    // Auto discovery (dedicated server only)
    'discovery'   => false,

    // IAM config
    'iam_config'  => [
        'insecure' => true,
        // 'root_cert_file' => './CA.pem', // Root CA file (uncomment for dedicated server)
    ],

    'credentials' => new StaticAuthentication($user, $password)
];

$ydb = new Ydb($config);
```

Balancing

YDB uses client load balancing because it is more efficient when a lot of traffic from multiple client applications comes to a database. In most cases, it just works in the YDB SDK. However, sometimes specific settings for client load balancing are required, for example, to reduce server hops and request time or to distribute the load across availability zones.

Note that custom balancing is limited when it comes to YDB sessions. Custom balancing in the YDB SDKs is performed only when creating a new YDB session on a specific node. Once the session is created, all queries in this session are passed to the node where the session was created. Queries in the same YDB session are not balanced between different YDB nodes.

This section contains code recipes with client load balancing settings in different YDB SDKs.

Table of contents:

- Random choice
- Prefer the nearest data center
- Prefer the specific availability zone
- Prefer a pile with a specific state

Random choice

The YDB SDK uses the `random_choice` (uniform random) balancing algorithm by default, except the C++ SDK, which defaults to "prefer the nearest data center".

Below are examples of explicitly setting the "random choice" balancing algorithm in different YDB SDKs.

Go

Native SDK

```
package main

import (
  "context"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/balancers"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithBalancer(
      balancers.RandomChoice(),
    ),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  // ...
}
```

database/sql

Client-side balancing in the YDB `database/sql` driver happens only when opening a new connection (in `database/sql` terms), which maps to a YDB session on a specific node. After the session is created, all queries on that session go to that node. Queries on the same YDB session are not balanced across nodes.

Example for "random choice" balancing:

```
package main

import (
  "context"
  "database/sql"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/balancers"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithBalancer(
      balancers.RandomChoice(),
    ),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }

  db := sql.OpenDB(connector)
  defer db.Close()
  // ...
}
```

C++

```
#include <ydb-cpp-sdk/client/driver/driver.h>

int main() {
  auto connectionString = std::string(std::getenv("YDB_CONNECTION_STRING"));

  auto driverConfig = NYdb::TDriverConfig(connectionString)
    .SetBalancingPolicy(NYdb::TBalancingPolicy::UseAllNodes());
```

```
  NYdb::TDriver driver(driverConfig);
  // ...
  driver.Stop(true);
  return 0;
}
```

Python

Native SDK

```
import os
import ydb

driver_config = ydb.DriverConfig(
    endpoint=os.environ["YDB_ENDPOINT"],
    database=os.environ["YDB_DATABASE"],
    credentials=ydb.credentials_from_env_variables(),
    use_all_nodes=True,  # uniform random choice
)

with ydb.Driver(driver_config) as driver:
    driver.wait(timeout=5)
    # ...
```

Native SDK (Asyncio)

```
import os
import ydb
import asyncio

async def ydb_init():
    driver_config = ydb.DriverConfig(
        endpoint=os.environ["YDB_ENDPOINT"],
        database=os.environ["YDB_DATABASE"],
        credentials=ydb.credentials_from_env_variables(),
        use_all_nodes=True,  # uniform random choice
    )
    async with ydb.aio.Driver(driver_config) as driver:
        await driver.wait()
        # ...

asyncio.run(ydb_init())
```

SQLAlchemy

```
import os
import sqlalchemy as sa

engine = sa.create_engine(
    os.environ["YDB_SQLALCHEMY_URL"],
    connect_args={
        "driver_config_kwargs": {
            "use_all_nodes": True,  # uniform random choice
        }
    },
)
```

JavaScript

This section is under development.

Java

Native SDK

"Random choice" in the Java SDK corresponds to the `USE_ALL_NODES` policy in `BalancingSettings` (this is also the default if you do not override settings).

```
import tech.ydb.core.grpc.BalancingSettings;
import tech.ydb.core.grpc.GrpcTransport;

try (GrpcTransport transport = GrpcTransport.forConnectionString("grpc://localhost:2136/local")
        .withBalancingSettings(BalancingSettings.fromPolicy(BalancingSettings.Policy.USE_ALL_NODES))
        .build()) {
    // ...
}
```

JDBC

Balancing when selecting a new session is handled by the native transport inside the driver; if needed, use the same parameters as in the native SDK via JDBC connection settings.

In Spring Boot, ORMs, and other JDBC wrappers, use the same JDBC URL and balancing parameters as with the driver directly (for example `spring.datasource.url` with query parameters or `DataSource` properties).

Rust

The RandomChoice policy (random endpoint selection among discovery nodes) is enabled **by default** — no extra configuration is required.

Prefer the nearest data center

Below are examples of setting the "prefer the nearest data center" balancing algorithm in different YDB SDKs.

Go

Native SDK

```
package main

import (
  "context"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/balancers"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithBalancer(
      balancers.PreferLocalDC(
        balancers.RandomChoice(),
      ),
    ),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  // ...
}
```

database/sql

Client-side balancing in the YDB `database/sql` driver happens only when opening a new connection (in `database/sql` terms), which corresponds to a YDB session on a specific node. After the session is created, all queries on that session go to that node. Queries on the same YDB session are not balanced across nodes.

Example for "prefer the nearest data center" balancing:

```
package main

import (
  "context"
  "database/sql"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/balancers"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithBalancer(
      balancers.PreferLocalDC(
        balancers.RandomChoice(),
      ),
    ),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }

  db := sql.OpenDB(connector)
  defer db.Close()
  // ...
}
```

C++

The YDB C++ SDK uses the `prefer_local_dc` (prefer nearest data center) algorithm by default.

```
#include <ydb-cpp-sdk/client/driver/driver.h>

int main() {
  auto connectionString = std::string(std::getenv("YDB_CONNECTION_STRING"));
```

```
  auto driverConfig = NYdb::TDriverConfig(connectionString)
    .SetBalancingPolicy(NYdb::TBalancingPolicy::UsePreferableLocation());

  NYdb::TDriver driver(driverConfig);
  // ...
  driver.Stop(true);
  return 0;
}
```

Python

Native SDK

```
import os
import ydb

driver_config = ydb.DriverConfig(
    endpoint=os.environ["YDB_ENDPOINT"],
    database=os.environ["YDB_DATABASE"],
    credentials=ydb.credentials_from_env_variables(),
    use_all_nodes=False,  # prefer the nearest data center
)

with ydb.Driver(driver_config) as driver:
    driver.wait(timeout=5)
    # ...
```

Native SDK (Asyncio)

```
import os
import ydb
import asyncio

async def ydb_init():
    driver_config = ydb.DriverConfig(
        endpoint=os.environ["YDB_ENDPOINT"],
        database=os.environ["YDB_DATABASE"],
        credentials=ydb.credentials_from_env_variables(),
        use_all_nodes=False,  # prefer the nearest data center
    )
    async with ydb.aio.Driver(driver_config) as driver:
        await driver.wait()
        # ...

asyncio.run(ydb_init())
```

SQLAlchemy

```
import os
import sqlalchemy as sa

engine = sa.create_engine(
    os.environ["YDB_SQLALCHEMY_URL"],
    connect_args={
        "driver_config_kwargs": {
            "use_all_nodes": False,  # prefer the nearest data center
        }
    },
)
```

JavaScript

This section is under development.

Java

Native SDK

```
import tech.ydb.core.grpc.BalancingSettings;
import tech.ydb.core.grpc.GrpcTransport;

try (GrpcTransport transport = GrpcTransport.forConnectionString("grpc://localhost:2136/local")
        .withBalancingSettings(BalancingSettings.detectLocalDs())
        .build()) {
    // ...
}
```

JDBC

See JDBC driver properties; configure balancing via the native transport if needed.

In Spring Boot, ORMs, and other JDBC wrappers, use the same JDBC URL and balancing settings as with the driver directly (for example `spring.datasource.url` or `DataSource` properties).

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#239

Prefer a specific availability zone

Below are examples of setting the "prefer availability zone" balancing algorithm in different YDB SDKs.

Go

Native SDK

```
package main

import (
  "context"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/balancers"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithBalancer(
      balancers.PreferLocations(
        balancers.RandomChoice(),
        "a",
        "b",
      ),
    ),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  // ...
}
```

database/sql

Client-side balancing in the YDB `database/sql` driver happens only when opening a new connection (in `database/sql` terms), which maps to a YDB session on a specific node. After the session is created, all queries on that session go to that node. Queries on the same YDB session are not balanced across nodes.

Example for "prefer availability zone" balancing:

```
package main

import (
  "context"
  "database/sql"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/balancers"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithBalancer(
      balancers.PreferLocations(
        balancers.RandomChoice(),
        "a",
        "b",
      ),
    ),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }

  db := sql.OpenDB(connector)
  defer db.Close()
  // ...
}
```

C++

The C++ SDK lets you pick only one availability zone as preferred.

```cpp
#include <ydb-cpp-sdk/client/driver/driver.h>

int main() {
  auto connectionString = std::string(std::getenv("YDB_CONNECTION_STRING"));

  auto driverConfig = NYdb::TDriverConfig(connectionString)
    .SetBalancingPolicy(NYdb::TBalancingPolicy::UsePreferableLocation("datacenter1"));

  NYdb::TDriver driver(driverConfig);
  // ...
  driver.Stop(true);
  return 0;
}
```

Python

This functionality is not currently supported.

JavaScript

This section is under development.

Java

In the **Java SDK**, availability zone preference is set on the gRPC transport.

Native SDK

```java
import tech.ydb.core.grpc.BalancingSettings;
import tech.ydb.core.grpc.GrpcTransport;

try (GrpcTransport transport = GrpcTransport.forConnectionString("grpc://localhost:2136/local")
        .withBalancingSettings(BalancingSettings.fromLocation("a")) // preferred availability zone
        .build()) {
    // ...
}
```

JDBC

See supported availability-zone parameters in JDBC driver properties, or configure balancing through the native API when embedding the driver.

In Spring Boot, ORMs, and other JDBC wrappers, pass the same JDBC URL and zone parameters as for a direct connection (for example in `spring.datasource.url` or pool properties).

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#238

Prefer a pile with a specific state

Below is an example of setting the balancing algorithm option that prefers a pile in a given state in the YDB SDK.

If no state is specified when setting the option, the SDK prefers the PRIMARY pile.

This option only makes sense if the cluster is operating in bridge mode. Otherwise, the SDK uses random choice balancing.

Go

This functionality is not currently supported.

C++

```
#include <ydb-cpp-sdk/client/driver/driver.h>

int main() {
  auto connectionString = std::string(std::getenv("YDB_CONNECTION_STRING"));

  auto driverConfig = NYdb::TDriverConfig(connectionString)
    .SetBalancingPolicy(NYdb::TBalancingPolicy::UsePreferablePileState(NYdb::EPileState::PRIMARY));

  NYdb::TDriver driver(driverConfig);
  // ...
  driver.Stop(true);
  return 0;
}
```

Python

This functionality is not currently supported.

JavaScript

This section is under development.

Java

This functionality is not currently supported.

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#491

Retrying

YDB is a distributed database management system with automatic load scaling.
Routine maintenance can be carried out on the server side, with server racks or entire data centers temporarily shut down.
This may result in errors when working with YDB.
Depending on the error type, you should respond differently.
YDB SDKs provide built-in tools for retries to ensure high availability,
with error types accounted for and defined handling for them.

Below are code examples showing the YDB SDK built-in tools for retries:

C++

In the YDB C++ SDK, retries with correct error handling is implemented by several programming interfaces:

Synchronous retry attempts

The `RetryQuerySync` method is used to execute queries with automatic retries.
The method accepts a lambda function that receives a session object and returns the query result.
YDB C++ SDK automatically analyzes errors and performs retries according to their type.

Example code using `RetryQuerySync`:

```
#include <ydb-cpp-sdk/client/query/client.h>

void ExecuteQueryWithRetry(NYdb::NQuery::TQueryClient client) {
    auto result = client.RetryQuerySync([](NYdb::NQuery::TSession session) -> NYdb::TStatus {
        auto query = R"(
            SELECT series_id, title
            FROM series
            WHERE series_id = 1;
        )";

        auto result = session.ExecuteQuery(
            query,
            NYdb::NQuery::TTxControl::BeginTx(NYdb::NQuery::TTxSettings::SerializableRW()).CommitTx()
        ).GetValueSync();

        if (!result.IsSuccess()) {
            return result;
        }

        // Process query results
        auto resultSet = result.GetResultSet(0);
        NYdb::TResultSetParser parser(resultSet);
        while (parser.TryNextRow()) {
            std::cout << "Series"
                << ", Id: " << parser.ColumnParser("series_id").GetOptionalUint64().value()
                << ", Title: " << parser.ColumnParser("title").GetOptionalUtf8().value()
                << std::endl;
        }

        return result;
    });

    if (!result.IsSuccess()) {
        // Handle error after all retry attempts
        std::cerr << "Query failed: " << result.GetIssues().ToString() << std::endl;
    }
}
```

Asynchronous retry attempts

The `RetryQuery` method is used for asynchronous query execution with automatic retries.
The method returns `NThreading::TFuture`, which allows for asynchronous operations.

Example code using `RetryQuery`:

```
#include <ydb-cpp-sdk/client/query/client.h>

void ExecuteQueryWithRetryAsync(NYdb::NQuery::TQueryClient client) {
    auto future = client.RetryQuery([](NYdb::NQuery::TSession session) -> NYdb::TAsyncStatus {
        auto query = R"(
            SELECT series_id, title, release_date
            FROM series
            WHERE series_id = 1;
        )";

        return session.ExecuteQuery(
            query,
            NYdb::NQuery::TTxControl::BeginTx(NYdb::NQuery::TTxSettings::SerializableRW()).CommitTx()
        ).Apply([](const NYdb::NQuery::TAsyncExecuteQueryResult& asyncResult) -> NYdb::TStatus {
            auto result = asyncResult.GetValue();
            if (!result.IsSuccess()) {
                return result;
            }

            // Process query results
            auto resultSet = result.GetResultSet(0);
            NYdb::TResultSetParser parser(resultSet);
```

```
            while (parser.TryNextRow()) {
                std::cout << "Series"
                    << ", Id: " << parser.ColumnParser("series_id").GetOptionalUint64().value()
                    << ", Title: " << parser.ColumnParser("title").GetOptionalUtf8().value()
                    << std::endl;
            }

            return result;
        });
    });

    // Wait for completion
    auto status = future.GetValueSync();
    if (!status.IsSuccess()) {
        std::cerr << "Query failed: " << status.GetIssues().ToString() << std::endl;
    }
}
```

Retry attempts for streaming queries

For executing streaming queries with automatic retries, use `StreamExecuteQuery`.
Streaming queries allow processing large volumes of data by receiving results in parts.

Example code using `RetryQuerySync` with `StreamExecuteQuery`:

```
#include <ydb-cpp-sdk/client/query/client.h>

void StreamQueryWithRetry(NYdb::NQuery::TQueryClient client) {
    auto result = client.RetryQuerySync([](NYdb::NQuery::TSession session) -> NYdb::TStatus {
        auto query = R"(
            SELECT series_id, title, release_date
            FROM series
            WHERE series_id > 0;
        )";

        auto resultStreamQuery = session.StreamExecuteQuery(
            query,
            NYdb::NQuery::TTxControl::NoTx()
        ).GetValueSync();

        if (!resultStreamQuery.IsSuccess()) {
            return resultStreamQuery;
        }

        // Process results in parts
        bool eos = false;
        while (!eos) {
            auto streamPart = resultStreamQuery.ReadNext().ExtractValueSync();

            if (!streamPart.IsSuccess()) {
                eos = true;
                if (!streamPart.EOS()) {
                    return streamPart;
                }
                continue;
            }

            if (streamPart.HasResultSet()) {
                auto rs = streamPart.ExtractResultSet();
                NYdb::TResultSetParser parser(rs);
                while (parser.TryNextRow()) {
                    std::cout << "Series"
                        << ", Id: " << parser.ColumnParser("series_id").GetOptionalUint64().value()
                        << ", Title: " << parser.ColumnParser("title").GetOptionalUtf8().value()
                        << std::endl;
                }
            }
        }

        return resultStreamQuery;
    });

    if (!result.IsSuccess()) {
        std::cerr << "Stream query failed: " << result.GetIssues().ToString() << std::endl;
    }
}
```

Configuring retry parameters

Users can configure the behavior of the retry mechanism using the `TRetryOperationSettings` class:

- `MaxRetries(uint32_t)` - maximum number of retry attempts (default is 10)
- `Idempotent(bool)` - indicates whether the operation is idempotent. Idempotent operations are retried for a broader range of errors
- `RetryNotFound(bool)` - whether to retry operations that returned a `NOT_FOUND` status (default is true)
- `MaxTimeout(TDuration)` - maximum time for all retry attempts
- `FastBackoffSettings(TBackoffSettings)` - settings for fast retries

- `SlowBackoffSettings(TBackoffSettings)` - settings for slow retries

Example of using retry settings:

```
#include <ydb-cpp-sdk/client/query/client.h>
#include <ydb-cpp-sdk/client/retry/retry.h>

void ExecuteWithCustomRetry(NYdb::NQuery::TQueryClient client) {
    auto retrySettings = NYdb::NRetry::TRetryOperationSettings()
        .Idempotent(true)
        .MaxRetries(20)
        .MaxTimeout(TDuration::Seconds(30));

    auto result = client.RetryQuerySync([](NYdb::NQuery::TSession session) -> NYdb::TStatus {
        auto query = R"(
            UPSERT INTO series (series_id, title)
            VALUES (10, "New Series");
        )";

        auto result = session.ExecuteQuery(
            query,
            NYdb::NQuery::TTxControl::BeginTx(NYdb::NQuery::TTxSettings::SerializableRW()).CommitTx()
        ).GetValueSync();

        if (!result.IsSuccess()) {
            return result;
        }

        // Process query result
        std::cout << "Query executed successfully" << std::endl;
        return result;
    }, retrySettings);

    if (!result.IsSuccess()) {
        std::cerr << "Operation failed: " << result.GetIssues().ToString() << std::endl;
    }
}
```

Go

Native SDK

In the YDB Go SDK, correct error handling is implemented by several programming interfaces:

General-purpose repeat function

The basic logic of error handling is implemented by the helper `retry.Retry` function.
The details of repeat query execution are mostly hidden.

The user can affect the logic of the `retry.Retry` function in two ways:

- Via the context (where you can set the deadline and cancel)
- Via the operation's idempotency flag `retry.WithIdempotent()`. By default, the operation is considered non-idempotent.

The user passes a custom function to `retry.Retry` that returns an error by its signature.
If the custom function returns `nil`, then repeat queries stop.
If the custom function returns an error, the YDB Go SDK tries to identify this error and executes retries depending on it.

Example of the code that uses the `retry.Retry` function:

```
package main

import (
  "context"
  "time"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/retry"
)

func main() {
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  var cancel context.CancelFunc
  // fix deadline for retries
  ctx, cancel := context.WithTimeout(ctx, time.Second)
  err = retry.Retry(
    ctx,
    func(ctx context.Context) error {
      whoAmI, err := db.Discovery().WhoAmI(ctx)
      if err != nil {
        return err
      }
      fmt.Println(whoAmI)
      return nil
```

```
    },
    retry.WithIdempotent(true),
  )
  if err != nil {
    panic(err)
  }
}
```

Repeat attempts in case of failed YDB session objects

For repeat error handling at the level of a YDB table service session, you can use the `db.Table().Do(ctx, op)` function, which provides a prepared session for query execution.
`db.Table().Do(ctx, op)` uses the `retry` package and tracks the lifetime of the YDB sessions.
Based on its signature, the user's operation `op` should return an error or `nil` so that the driver can "decide" what to do based on the error type: repeat the operation or not, with delay or without, and in this session or a new one.
The user can affect the logic of repeat queries using the context and the idempotence flag, while the YDB Go SDK interprets errors returned by `op`.

Example of the code that uses the `db.Table().Do(ctx, op)` function:

```
err := db.Table().Do(ctx, func(ctx context.Context, s table.Session) (err error) {
  desc, err = s.DescribeTableOptions(ctx)
  return
}, table.WithIdempotent())
if err != nil {
  return err
}
```

Repeat attempts in case of failed YDB interactive transaction objects

For repeat error handling at the level of a YDB table service interactive transaction, you can use the `db.Table().DoTx(ctx, txOp)` function, which provides a YDB prepared session transaction for query execution.
`db.Table().DoTx(ctx, txOp)` uses the `retry` package and tracks the lifetime of the YDB sessions.
Based on its signature, the user's operation `txOp` should return an error or `nil` so that the driver can "decide" what to do based on the error type: repeat the operation or not, with delay or without, and in this transaction or a new one.
The user can affect the logic of repeat queries using the context and the idempotence flag, while the YDB Go SDK interprets errors returned by `op`.

Example of the code that uses the `db.Table().DoTx(ctx, op)` function:

```
err := db.Table().DoTx(ctx, func(ctx context.Context, tx table.TransactionActor) error {
  _, err := tx.Execute(ctx,
    "DECLARE $id AS Int64; INSERT INTO test (id, val) VALUES($id, 'asd')",
    table.NewQueryParameters(table.ValueParam("$id", types.Int64Value(100500))),
  )
  return err
}, table.WithIdempotent())
if err != nil {
  return err
}
```

Queries to other YDB services

(`db.Scripting()`, `db.Scheme()`, `db.Coordination()`, `db.Ratelimiter()`, `db.Discovery()`) also use the `retry.Retry` function inside to execute repeat queries and don't require external auxiliary functions for repeats.

database/sql

The standard `database/sql` package uses the internal logic of repeats based on the errors a specific driver implementation returns. For example, the `database/sql` code frequently shows the three-attempt repeats policy:

- Two attempts at a present connection or new one (if the `database/sql` connection pool is empty).
- One attempt at a new connection.

This repeat policy is mostly enough to survive temporary unavailability of YDB nodes or issues with a YDB session.

The YDB Go SDK provides special functions to ensure execution of a user's operation:

Repeat attempts in case of failed `*sql.Conn` connection objects:

For repeat error handling at `*sql.Conn` connection objects, you can use the auxiliary `retry.Do(ctx, db, op)` function, which provides a prepared `*sql.Conn` session for query execution.
You need to pass the context, database object, and the user's operation for execution to the `retry.Do` function.
The user's code can affect the logic of repeat queries using the context and the idempotence flag, while the YDB Go SDK, in turn, interprets errors returned by `op`.

The user's `op` operation must return an error or `nil`:

- If the custom function returns `nil`, then repeat queries stop.
- If the custom function returns an error, the YDB Go SDK tries to identify this error and performs retries depending on it.

Example of the code that uses the `retry.Do` function:

```
import (
  "context"
```

```
  "database/sql"
  "fmt"
  "log"

  "github.com/ydb-platform/ydb-go-sdk/v3/retry"
)

func main() {
  ...
  err = retry.Do(ctx, db, func(ctx context.Context, cc *sql.Conn) (err error) {
    row = cc.QueryRowContext(ctx, `
        PRAGMA TablePathPrefix("/local");
        DECLARE $seriesID AS Uint64;
        DECLARE $seasonID AS Uint64;
        DECLARE $episodeID AS Uint64;
        SELECT views FROM episodes WHERE series_id = $seriesID AND season_id = $seasonID AND episode_id =
$episodeID;
      `,
      sql.Named("seriesID", uint64(1)),
      sql.Named("seasonID", uint64(1)),
      sql.Named("episodeID", uint64(1)),
    )
    var views sql.NullFloat64
    if err = row.Scan(&views); err != nil {
      return fmt.Errorf("cannot scan views: %w", err)
    }
    if views.Valid {
      return fmt.Errorf("unexpected valid views: %v", views.Float64)
    }
    log.Printf("views = %v", views)
    return row.Err()
  }, retry.WithDoRetryOptions(retry.WithIdempotent(true)))
  if err != nil {
    log.Printf("retry.Do failed: %v\n", err)
  }
}
```

Repeat attempts in case of failed `*sql.Tx` interactive transaction objects:

For repeat error handling at `*sql.Tx` interactive transaction objects, you can use the auxiliary `retry.DoTx(ctx, db, op)` function, which provides a prepared `*sql.Tx` transaction for query execution.
You need to pass the context, database object, and the user's operation for execution to the `retry.DoTx` function.
The function is passed a prepared `*sql.Tx` transaction, where queries to YDB should be executed.
The user's code can affect the logic of repeat queries using the context and the operation idempotence flag, while the YDB Go SDK, in turn, interprets errors returned by `op`.

The user's `op` operation must return an error or `nil`:

- If the custom function returns `nil`, then repeat queries stop.
- If the custom function returns an error, the YDB Go SDK tries to identify this error and performs retries depending on it.

By default, `retry.DoTx` uses the read-write isolation mode of the `sql.LevelDefault` transaction and you can change it using the `retry.WithTxOptions` parameter.

Example of the code that uses the `retry.Do` function:

```
import (
  "context"
  "database/sql"
  "fmt"
  "log"

  "github.com/ydb-platform/ydb-go-sdk/v3/retry"
)

func main() {
  ...
  err = retry.DoTx(ctx, db, func(ctx context.Context, tx *sql.Tx) error {
    row := tx.QueryRowContext(ctx,`
        PRAGMA TablePathPrefix("/local");
        DECLARE $seriesID AS Uint64;
        DECLARE $seasonID AS Uint64;
        DECLARE $episodeID AS Uint64;
        SELECT views FROM episodes WHERE series_id = $seriesID AND season_id = $seasonID AND episode_id =
$episodeID;
      `,
      sql.Named("seriesID", uint64(1)),
      sql.Named("seasonID", uint64(1)),
      sql.Named("episodeID", uint64(1)),
    )
    var views sql.NullFloat64
    if err = row.Scan(&views); err != nil {
      return fmt.Errorf("cannot select current views: %w", err)
    }
    if !views.Valid {
      return fmt.Errorf("unexpected invalid views: %v", views)
    }
    t.Logf("views = %v", views)
    if views.Float64 != 1 {
```

```
      return fmt.Errorf("unexpected views value: %v", views)
    }
    return nil
  }, retry.WithDoTxRetryOptions(retry.WithIdempotent(true)), retry.WithTxOptions(&sql.TxOptions{
    Isolation: sql.LevelSnapshot,
    ReadOnly:  true,
  }))
  if err != nil {
    log.Printf("do tx failed: %v\n", err)
  }
}
```

Java

Native SDK

In the YDB Java SDK, repeat queries are implemented by the `SessionRetryContext` helper class. This class is constructed with the `SessionRetryContext.create` method to which you pass the `SessionSupplier` interface implementation (usually an instance of the `TableClient` class or the `QueryClient` class).

Additionally, the user can specify some other options:

- `maxRetries(int maxRetries)`: The maximum number of operation retries, not counting the first execution. Default value: `10`
- `retryNotFound(boolean retryNotFound)`: The option to retry operations that returned the `NOT_FOUND` status. Enabled by default.
- `idempotent(boolean idempotent)`: Indicates idempotence of operations. Idempotent operations will be retried for a broader range of errors. Disabled by default.

The `SessionRetryContext` class provides two methods to run operations with retries.

- `CompletableFuture<Status> supplyStatus`: Executing the operation that returns the status. As an argument, it accepts the lambda `Function<Session, CompletableFuture<Status>> fn`
- `CompletableFuture<Result<T>> supplyResult`: Executing the operation that returns data. As an argument, it accepts the lambda `Function<Session, CompletableFuture<Result<T>>> fn`

When using the `SessionRetryContext` class, make sure that the operation will be retried in the following cases:

- The lambda function returned a retryable error code
- The lambda function invoked an `UnexpectedResultException` with a retryable error code

Sample code using SessionRetryContext.supplyStatus:

```
private void createTable(TableClient tableClient, String database, String tableName) {
    SessionRetryContext retryCtx = SessionRetryContext.create(tableClient).build();
    TableDescription pets = TableDescription.newBuilder()
            .addNullableColumn("species", PrimitiveType.Text)
            .addNullableColumn("name", PrimitiveType.Text)
            .addNullableColumn("color", PrimitiveType.Text)
            .addNullableColumn("price", PrimitiveType.Float)
            .setPrimaryKeys("species", "name")
            .build();

    String tablePath = database + "/" + tableName;
    retryCtx.supplyStatus(session -> session.createTable(tablePath, pets))
            .join().expectSuccess();
}
```

Sample code using SessionRetryContext.supplyResult:

```
private void selectData(TableClient tableClient, String tableName) {
    SessionRetryContext retryCtx = SessionRetryContext.create(tableClient).build();
    String selectQuery
            = "DECLARE $species AS Text;"
            + "DECLARE $name AS Text;"
            + "SELECT * FROM " + tableName + " "
            + "WHERE species = $species AND name = $name;";

    Params params = Params.of(
            "$species", PrimitiveValue.newText("cat"),
            "$name", PrimitiveValue.newText("Tom")
    );

    DataQueryResult data = retryCtx
            .supplyResult(session -> session.executeDataQuery(selectQuery, TxControl.onlineRo(), params))
            .join().getValue();

    ResultSetReader rsReader = data.getResultSet(0);
    logger.info("Result of select query:");
    while (rsReader.next()) {
        logger.info("  species: {}, name: {}, color: {}, price: {}",
                rsReader.getColumn("species").getText(),
                rsReader.getColumn("name").getText(),
                rsReader.getColumn("color").getText(),
                rsReader.getColumn("price").getFloat()
        );
```

```
    }
}
```

JDBC

Retries at the `SessionRetryContext` level apply to the native API (`TableClient` / `QueryClient`). When using JDBC, implement retries at the application level or use native transport and clients as in Driver initialization.

Python

Native SDK

In the YDB Python SDK, retries are implemented in `QuerySessionPool` using the `RetrySettings` class. `RetrySettings` supports:

- `max_retries` — maximum number of retries (default 10)
- `idempotent` — whether the operation is idempotent; idempotent operations are retried for a broader set of errors (default False)
- `backoff_ceiling` , `backoff_slot_duration` — parameters for exponential backoff
- `fast_backoff_settings` , `slow_backoff_settings` — fast and slow retry tuning

For queries with retries, `QuerySessionPool` provides `retry_operation_sync` and `execute_with_retries` . Use `execute_with_retries` for one-off queries in implicit transaction mode. For explicit transactions or multiple operations in one transaction, use `retry_operation_sync` .

Example using `execute_with_retries` :

```
import ydb

def execute_query(pool: ydb.QuerySessionPool):
    result_sets = pool.execute_with_retries(
        "SELECT series_id, title FROM series WHERE series_id = 1;",
        retry_settings=ydb.RetrySettings(idempotent=True),
    )
    # ...
```

Example using `retry_operation_sync` :

```
import ydb

def execute_query(pool: ydb.QuerySessionPool):
    def callee(session: ydb.QuerySession):
          with session.transaction().execute(
              "SELECT 1",
              commit_tx=True,
          ) as result_sets:
              pass

    result = pool.retry_operation_sync(
        callee,
        retry_settings=ydb.RetrySettings(max_retries=20, idempotent=True),
    )
    # ...
```

Native SDK (Asyncio)

Example using `execute_with_retries` :

```
import ydb

async def execute_query(pool: ydb.aio.QuerySessionPool):
    result_sets = await pool.execute_with_retries(
        "SELECT series_id, title FROM series WHERE series_id = 1;",
        retry_settings=ydb.RetrySettings(idempotent=True),
    )
    # ...
```

Example using `retry_operation_sync` :

```
import ydb

async def execute_query(pool: ydb.aio.QuerySessionPool):
    async def callee(session):
        async with session.transaction(tx_mode=ydb.QuerySerializableReadWrite()) as tx:
            async with await tx.execute("SELECT 1", commit_tx=True) as result_sets:
                pass

    await pool.retry_operation_async(
        callee,
        retry_settings=ydb.RetrySettings(max_retries=20, idempotent=True),
    )
    # ...
```

SQLAlchemy

When using YDB through SQLAlchemy, retries happen internally and are not configured from the outside.

Rust

`QueryClient` automatically retries methods that execute a single transactional SQL statement (`query_row`, `exec`, and similar). For multiple operations in one transaction, use `retry_transaction`.

```rust
use ydb::{AccessTokenCredentials, ClientBuilder, YdbResult};

#[tokio::main]
async fn main() -> YdbResult<()> {
    let client = ClientBuilder::new_from_connection_string(
        "grpc://localhost:2136/local",
    )?
    .with_credentials(AccessTokenCredentials::from("..."))
    .client()?;

    client.wait().await?;

    let mut qc = client.query_client().clone_with_idempotent_operations(true);

    // query_row / exec: internal retries
    let mut row = qc
        .query_row("SELECT series_id, title FROM series WHERE series_id = 1")
        .await?;

    // multiple operations in one retried transaction
    let title: String = qc
        .retry_transaction(async |tx| {
            let mut row = tx
                .query_row("SELECT series_id, title FROM series WHERE series_id = 1")
                .await?;
            Ok(row.remove_field_by_name("title")?.try_into()?)
        })
        .await?;

    Ok(())
}
```

JavaScript

Retries and reconnections are built into the SDK; you do not need to configure anything separately.

The retrier is available in the separate `@ydbjs/retry` package.

```javascript
import { retry } from '@ydbjs/retry'

let attempts = 0
const result = retry({ retry: isError, budget: 3 }, async () => {
  if (attempts >= 2) {
    return 'success'
  }

  attempts++
  throw new Error('test error')
})
```

Setting the session pool size

YDB creates an actor for each session. Consequently, the session pool size on the client affects resource consumption (memory, CPU) on the YDB server.

For example, if 1000 clients of the same database open 1000 sessions each, one million actors are created on the server, consuming a large amount of memory and CPU. Without a client-side session limit, this can slow the cluster down and put it in a degraded state.

By default, native YDB SDK drivers use a limit of 50 sessions. When using third-party libraries such as Go `database/sql`, no limit is set.

Set the client session limit to the minimum needed for normal application operation. Remember that a session is single-threaded on both server and client. If the application must run 1000 concurrent in-flight requests to YDB, set the limit to about 1000 sessions.

Distinguish estimated RPS (requests per second) from in-flight requests. RPS is the total number of requests completed in one second. For example, with RPS = 10000 and average latency 100 ms, a limit of 1000 sessions is enough: each session can run about 10 sequential requests per second on average.

Below are examples of setting the session pool limit in different YDB SDKs.

Go

Native SDK

```
package main

import (
  "context"

  "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithSessionPoolSizeLimit(500),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  ...
}
```

database/sql

The `database/sql` library has its own connection pool. Each `database/sql` connection maps to one YDB session. Tune the pool with `sql.DB.SetMaxOpenConns` and `sql.DB.SetMaxIdleConns`. See the `database/sql` documentation.

Example using a `database/sql` connection pool:

```
package main

import (
  "context"
  "database/sql"

  _ "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  db, err := sql.Open("ydb", os.Getenv("YDB_CONNECTION_STRING"))
  if err != nil {
    panic(err)
  }
  defer db.Close()
  db.SetMaxOpenConns(100)
  db.SetMaxIdleConns(100)
  db.SetConnMaxIdleTime(time.Second) // workaround for background keep-aliving of YDB sessions
  ...
}
```

Java

Native SDK

```
this.queryClient = QueryClient.newClient(transport)
        // 10 — minimum active sessions kept in the pool during cleanup
        // 500 — maximum pool size
        .sessionPoolMinSize(10)
        .sessionPoolMaxSize(500)
        .build();
```

JDBC

JDBC applications typically use external connection pools such as HikariCP or C3p0. By default, the YDB JDBC driver observes how many connections the external pool opens and adjusts the session pool accordingly. Configure session limits by configuring HikariCP or C3p0 correctly.

Example HikariCP settings in Spring:

```
spring.datasource.url=jdbc:ydb:grpc://localhost:2136/local
spring.datasource.driver-class-name=tech.ydb.jdbc.YdbDriver
spring.datasource.hikari.maximum-pool-size=100 # max JDBC connections
```

Python

Native SDK

```
import os
import ydb

with ydb.Driver(
    connection_string=os.environ["YDB_CONNECTION_STRING"],
    credentials=ydb.credentials_from_env_variables(),
) as driver:
    driver.wait(timeout=5)
    with ydb.QuerySessionPool(driver, size=500) as pool:
        # ...
```

Native SDK (Asyncio)

```
import os
import ydb
import asyncio

async def ydb_init():
    async with ydb.aio.Driver(
        connection_string=os.environ["YDB_CONNECTION_STRING"],
        credentials=ydb.credentials_from_env_variables(),
    ) as driver:
        await driver.wait()
        async with ydb.aio.QuerySessionPool(driver, size=500) as pool:
            # ...

asyncio.run(ydb_init())
```

SQLAlchemy

Setting the pool size is not currently supported.

JavaScript

This section is under development.

Rust

For `QueryClient`, set the session pool size via `QuerySessionPoolSettings::with_limit` and `with_implicit_session_pool` (or `with_session_pool` for explicit sessions):

```
use ydb::{ClientBuilder, QuerySessionPoolSettings, YdbResult};

#[tokio::main]
async fn main() -> YdbResult<()> {
    let client = ClientBuilder::new_from_connection_string(
        std::env::var("YDB_CONNECTION_STRING")?,
    )?
    .client()?;
    client.wait().await?;

    let mut qc = client
        .query_client()
        .with_implicit_session_pool(
            QuerySessionPoolSettings::new().with_limit(500),
        );

    let mut row = qc.query_row("SELECT 1 AS one").await?;
    Ok(())
}
```

Upsert data

Below are code examples of using the built-in upsert tools of the YDB SDK:

C++

Native SDK

```
#include <ydb-cpp-sdk/client/query/client.h>

void UpsertSeries(NYdb::NQuery::TQueryClient& client) {
  NYdb::NStatusHelpers::ThrowOnError(client.RetryQuerySync(
      [](NYdb::NQuery::TSession session) {
          constexpr auto query = R"(
              DECLARE $seriesData AS List<Struct<
                  series_id: Uint64,
                  title: Utf8,
                  series_info: Utf8,
                  comment: Optional<Utf8>
              >>;

              UPSERT INTO series
              (
                  series_id,
                  title,
                  series_info,
                  comment
              )
              SELECT
                  series_id,
                  title,
                  series_info,
                  comment
              FROM AS_TABLE($seriesData);
          )";

          auto params = NYdb::TParamsBuilder()
              .AddParam("$seriesData")
                  .BeginList()
                  .AddListItem()
                      .BeginStruct()
                      .AddMember("series_id").Uint64(1)
                      .AddMember("title").Utf8("IT Crowd")
                      .AddMember("series_info").Utf8(
                          "The IT Crowd is a British sitcom produced by Channel 4, written by Graham Linehan,
produced by "
                          "Ash Atalla and starring Chris O'Dowd, Richard Ayoade, Katherine Parkinson, and Mat
t Berry.")
                      .AddMember("comment").OptionalUtf8(std::nullopt)
                      .EndStruct()
                  .AddListItem()
                      .BeginStruct()
                      .AddMember("series_id").Uint64(2)
                      .AddMember("title").Utf8("Silicon Valley")
                      .AddMember("series_info").Utf8(
                          "Silicon Valley is an American comedy television series created by Mike Judge, John
Altschuler and "
                          "Dave Krinsky. The series focuses on five young men who founded a startup company i
n Silicon Valley.")
                      .AddMember("comment").OptionalUtf8("lorem ipsum")
                      .EndStruct()
                  .EndList()
                  .Build()
              .Build();

          return session.ExecuteQuery(
              query,
              NYdb::NQuery::TTxControl::BeginTx(NYdb::NQuery::TTxSettings::SerializableRW()).CommitTx(),
              params).GetValueSync();
      },
      NYdb::NQuery::TRetryOperationSettings()
          .Idempotent(true)
  ));
}
```

userver

```
#include <userver/ydb/io/supported_types.hpp>
#include <userver/ydb/table.hpp>

struct SeriesData final {
    std::uint64_t series_id;
    ydb::Utf8 title;
    ydb::Utf8 series_info;
    std::optional<ydb::Utf8> comment;
};

void UpsertSeries(ydb::TableClient& client) {
    auto builder = client.GetBuilder();
    builder.Add(
```

```
        "$seriesData",
        std::vector<SeriesData>{
            {
                .series_id = 1,
                .title = ydb::Utf8{"IT Crowd"},
                .series_info = ydb::Utf8{
                    "The IT Crowd is a British sitcom produced by Channel 4, written by Graham Linehan, produ
ced by "
                    "Ash Atalla and starring Chris O'Dowd, Richard Ayoade, Katherine Parkinson, and Matt Berr
y."
                },
                .comment = std::nullopt,
            },
            {
                .series_id = 2,
                .title = ydb::Utf8{"Silicon Valley"},
                .series_info = ydb::Utf8{
                    "Silicon Valley is an American comedy television series created by Mike Judge, John Altsc
huler and "
                    "Dave Krinsky. The series focuses on five young men who founded a startup company in Sili
con Valley."
                },
                .comment = ydb::Utf8{"lorem ipsum"},
            },
        }
    );

    client.ExecuteQuery(
        ydb::OperationSettings{
            .tx_mode = ydb::TransactionMode::kSerializableRW,
            .is_idempotent = true,
        },
        ydb::Query{R"(
            DECLARE $seriesData AS List<Struct<
                series_id: Uint64,
                title: Utf8,
                series_info: Utf8,
                comment: Optional<Utf8>
            >>;

            UPSERT INTO series
            (
                series_id,
                title,
                series_info,
                comment
            )
            SELECT
                series_id,
                title,
                series_info,
                comment
            FROM AS_TABLE($seriesData);
        )"},
        std::move(builder)
    );
}
```

Go

Native SDK

```
package main

import (
  "context"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/table"
  "github.com/ydb-platform/ydb-go-sdk/v3/table/types"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  // execute upsert with native ydb data
  err = db.Table().DoTx( // Do retry operation on errors with best effort
    ctx, // context manages exiting from Do
    func(ctx context.Context, tx table.TransactionActor) (err error) { // retry operation
      res, err := tx.Execute(ctx, `
          PRAGMA TablePathPrefix("/path/to/table");
          DECLARE $seriesID AS Uint64;
```

```
          DECLARE $seasonID AS Uint64;
          DECLARE $episodeID AS Uint64;
          DECLARE $views AS Uint64;
          UPSERT INTO episodes ( series_id, season_id, episode_id, views )
          VALUES ( $seriesID, $seasonID, $episodeID, $views );
        `,
        table.NewQueryParameters(
          table.ValueParam("$seriesID", types.Uint64Value(1)),
          table.ValueParam("$seasonID", types.Uint64Value(1)),
          table.ValueParam("$episodeID", types.Uint64Value(1)),
          table.ValueParam("$views", types.Uint64Value(1)), // increment views
        ),
      )
      if err != nil {
        return err
      }
      if err = res.Err(); err != nil {
        return err
      }
      return res.Close()
    }, table.WithIdempotent(),
  )
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
}
```

database/sql

```
package main

import (
  "context"
  "database/sql"
  "os"

  _ "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/retry"
  "github.com/ydb-platform/ydb-go-sdk/v3/types"
)

func main() {
  db, err := sql.Open("ydb", os.Getenv("YDB_CONNECTION_STRING"))
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  // execute upsert with native ydb data
  err = retry.DoTx(ctx, db, func(ctx context.Context, tx *sql.Tx) error {
    if _, err = tx.ExecContext(ctx,`
        PRAGMA TablePathPrefix("/local");
        REPLACE INTO series
        SELECT
          series_id,
          title,
          series_info,
          comment
        FROM AS_TABLE($seriesData);
      `,
      sql.Named("seriesData", types.ListValue(
        types.StructValue(
          types.StructFieldValue("series_id", types.Uint64Value(1)),
          types.StructFieldValue("title", types.TextValue("IT Crowd")),
          types.StructFieldValue("series_info", types.TextValue(
            "The IT Crowd is a British sitcom produced by Channel 4, written by Graham Linehan, produced by
"+
            "Ash Atalla and starring Chris O'Dowd, Richard Ayoade, Katherine Parkinson, and Matt Berry.",
          )),
          types.StructFieldValue("comment", types.NullValue(types.TypeText)),
        ),
        types.StructValue(
          types.StructFieldValue("series_id", types.Uint64Value(2)),
          types.StructFieldValue("title", types.TextValue("Silicon Valley")),
          types.StructFieldValue("series_info", types.TextValue(
            "Silicon Valley is an American comedy television series created by Mike Judge, John Altschuler an
d "+
            "Dave Krinsky. The series focuses on five young men who founded a startup company in Silicon Vall
ey.",
          )),
          types.StructFieldValue("comment", types.TextValue("lorem ipsum")),
        ),
      )),
    ); err != nil {
      return err
    }
    return nil
  }, retry.WithDoTxRetryOptions(retry.WithIdempotent(true)))
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
```

```
  }
}
```

Java

Native SDK

Use `SessionRetryContext` and `TableSession.executeDataQuery` with the `$seriesData` parameter of type `List<Struct<...>>`. Values for `AS_TABLE($seriesData)` are collected in the same way as row structures in the batch upsert example.

```
SessionRetryContext retryCtx = SessionRetryContext.create(tableClient).build();

String yql = """
        PRAGMA TablePathPrefix("/local");
        DECLARE $seriesData AS List<Struct<
            series_id: Uint64,
            title: Utf8,
            series_info: Utf8,
            comment: Optional<Utf8>
        >>;
        UPSERT INTO series
        SELECT series_id, title, series_info, comment FROM AS_TABLE($seriesData);
        """;

Params params = Params.of("$seriesData", seriesDataListValue);

retryCtx.supplyResult(session -> session.executeDataQuery(yql, TxControl.serializableRw(), params))
        .join();
```

JDBC

```
try (Connection conn = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local");
     PreparedStatement ps = conn.prepareStatement(
             """
             REPLACE INTO series (series_id, title, series_info, comment)
             SELECT series_id, title, series_info, comment FROM AS_TABLE($seriesData);
             """
     )) {
    // The $seriesData parameter is set according to the query type (see the JDBC driver documentation)
}
```

In Spring Boot, Hibernate, JOOQ, and other frameworks on top of JDBC, the driver also tries to optimize **large** sequences of inserts and updates: when necessary, **UPSERT**, `INSERT`, `UPDATE`, `DELETE` are automatically **grouped into batches** on the driver side (including large batches from ORM).

Python

Native SDK

To upsert data, use `QuerySessionPool` and the `execute_with_retries` method with a parameterized YQL query. The query operates on the container type `List<Struct<..>>`, which allows passing multiple rows in a single call.

```
import os
import ydb

with ydb.Driver(
    connection_string=os.environ["YDB_CONNECTION_STRING"],
    credentials=ydb.credentials_from_env_variables(),
) as driver:
    driver.wait(timeout=5)
    pool = ydb.QuerySessionPool(driver)

    series_struct_type = ydb.StructType()
    series_struct_type.add_member("series_id", ydb.PrimitiveType.Uint64)
    series_struct_type.add_member("title", ydb.PrimitiveType.Utf8)
    series_struct_type.add_member("series_info", ydb.PrimitiveType.Utf8)
    series_struct_type.add_member("comment", ydb.OptionalType(ydb.PrimitiveType.Utf8))

    series_data = [
        {
            "series_id": 1,
            "title": "IT Crowd",
            "series_info": "The IT Crowd is a British sitcom...",
            "comment": None,
        },
        {
            "series_id": 2,
            "title": "Silicon Valley",
            "series_info": "Silicon Valley is an American comedy...",
            "comment": "lorem ipsum",
        },
    ]

    pool.execute_with_retries(
        """
        DECLARE $seriesData AS List<Struct<
            series_id: Uint64,
```

```
            title: Utf8,
            series_info: Utf8,
            comment: Optional<Utf8>
        >>;

        UPSERT INTO series
        (
            series_id,
            title,
            series_info,
            comment
        )
        SELECT
            series_id,
            title,
            series_info,
            comment
        FROM AS_TABLE($seriesData);
        """,
        {"$seriesData": (series_data, ydb.ListType(series_struct_type))},
        retry_settings=ydb.RetrySettings(idempotent=True),
    )
```

Native SDK (Asyncio)

```
import os
import ydb
import asyncio

async def ydb_init():
    async with ydb.aio.Driver(
        connection_string=os.environ["YDB_CONNECTION_STRING"],
        credentials=ydb.credentials_from_env_variables(),
    ) as driver:
        await driver.wait()
        pool = ydb.aio.QuerySessionPool(driver)

        series_struct_type = ydb.StructType()
        series_struct_type.add_member("series_id", ydb.PrimitiveType.Uint64)
        series_struct_type.add_member("title", ydb.PrimitiveType.Utf8)
        series_struct_type.add_member("series_info", ydb.PrimitiveType.Utf8)
        series_struct_type.add_member("comment", ydb.OptionalType(ydb.PrimitiveType.Utf8))

        series_data = [
            {"series_id": 1, "title": "IT Crowd", "series_info": "The IT Crowd is a British sitcom...", "comm
ent": None},
            {"series_id": 2, "title": "Silicon Valley", "series_info": "Silicon Valley is an American comed
y...", "comment": "lorem ipsum"},
        ]

        await pool.execute_with_retries(
            """
            DECLARE $seriesData AS List<Struct<
                series_id: Uint64,
                title: Utf8,
                series_info: Utf8,
                comment: Optional<Utf8>
            >>;

            UPSERT INTO series (series_id, title, series_info, comment)
            SELECT series_id, title, series_info, comment FROM AS_TABLE($seriesData);
            """,
            {"$seriesData": (series_data, ydb.ListType(series_struct_type))},
            retry_settings=ydb.RetrySettings(idempotent=True),
        )

asyncio.run(ydb_init())
```

SQLAlchemy

When using YDB via SQLAlchemy, the `ydb_sqlalchemy.upsert` function is used to insert data, which generates an `UPSERT INTO` query based on the table and the passed values. You can insert either a single row or multiple rows in one call:

```
import os
import sqlalchemy as sa
from sqlalchemy import Column, Integer, MetaData, String, Table
import ydb_sqlalchemy as ydb_sa

engine = sa.create_engine(os.environ["YDB_SQLALCHEMY_URL"])

series = Table(
    "series",
    MetaData(),
    Column("series_id", Integer, primary_key=True),
    Column("title", String),
    Column("series_info", String),
    Column("comment", String, nullable=True),
)

with engine.connect() as connection:
```

```
    stmt = ydb_sa.upsert(series).values(
        [
            {
                "series_id": 1,
                "title": "IT Crowd",
                "series_info": "The IT Crowd is a British sitcom...",
                "comment": None,
            },
            {
                "series_id": 2,
                "title": "Silicon Valley",
                "series_info": "Silicon Valley is an American comedy...",
                "comment": "lorem ipsum",
            },
        ]
    )
    connection.execute(stmt)
    connection.commit()
```

C#

```
using Ydb.Sdk.Ado;
using Ydb.Sdk.Ado.YdbType;

await using var dataSource = new YdbDataSource("Host=localhost;Port=2136;Database=/local");
await using var connection = await dataSource.OpenRetryableConnectionAsync();

var seriesData = new List<YdbStruct>
{
    new()
    {
        { "series_id", 1UL, YdbDbType.Uint64 },
        { "title", "IT Crowd", YdbDbType.Text },
        { "series_info", "The IT Crowd is a British sitcom produced by Channel 4.", YdbDbType.Text },
        { "comment", null, YdbDbType.Text },
    },
    new()
    {
        { "series_id", 2UL, YdbDbType.Uint64 },
        { "title", "Silicon Valley", YdbDbType.Text },
        { "series_info", "Silicon Valley is an American comedy television series.", YdbDbType.Text },
        { "comment", "lorem ipsum", YdbDbType.Text },
    },
};

var command = new YdbCommand("UPSERT INTO series SELECT * FROM AS_TABLE($series_data)", connection);
command.Parameters.Add(new YdbParameter("$series_data", seriesData));

await command.ExecuteNonQueryAsync();
```

JavaScript

```
import { Driver } from '@ydbjs/core'
import { query } from '@ydbjs/query'

const driver = new Driver('grpc://localhost:2136/local')
await driver.ready()

const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' },
]

const sql = query(driver)
await sql`UPSERT INTO users SELECT * FROM AS_TABLE(${users})`
```

Rust

```
use ydb::{
    ydb_params, ydb_struct, AccessTokenCredentials, ClientBuilder, Value, YdbResult,
};

fn series_row(
    series_id: u64,
    title: &str,
    series_info: &str,
    comment: Option<&str>,
) -> YdbResult<Value> {
    let comment_val = match comment {
        None => Value::optional_from(Value::Text(String::new()), None)?,
        Some(s) => Value::optional_from(
            Value::Text(String::new()),
            Some(Value::Text(s.into())),
        )?,
    };
    Ok(ydb_struct!(
        "series_id" => series_id,
        "title" => title,
```

```
        "series_info" => series_info,
        "comment" => comment_val,
    ))
}

#[tokio::main]
async fn main() -> YdbResult<()> {
    let client = ClientBuilder::new_from_connection_string(
        "grpc://localhost:2136?database=local",
    )?
    .with_credentials(AccessTokenCredentials::from("..."))
    .client()?;

    client.wait().await?;

    let example = series_row(0, "", "", None)?;
    let series_data = Value::list_from(
        example,
        vec![
            series_row(
                1,
                "IT Crowd",
                "The IT Crowd is a British sitcom...",
                None,
            )?,
            series_row(
                2,
                "Silicon Valley",
                "Silicon Valley is an American comedy...",
                Some("lorem ipsum"),
            )?,
        ],
    )?;

    client
        .query_client()
        .exec(
            r#"
            PRAGMA TablePathPrefix("/local");
            UPSERT INTO series
            (
                series_id,
                title,
                series_info,
                comment
            )
            SELECT
                series_id,
                title,
                series_info,
                comment
            FROM AS_TABLE($seriesData);
            "#,
        )
        .params(ydb_params!("$seriesData" => series_data))
        .await?;

    Ok(())
}
```

PHP

```
<?php

use YdbPlatform\Ydb\Session;
use YdbPlatform\Ydb\Ydb;

$ydb = new Ydb($config);

$yql = <<<'EOS'
PRAGMA TablePathPrefix("/local");
DECLARE $seriesData AS List<Struct<
    series_id: Uint64,
    title: Utf8,
    series_info: Utf8,
    comment: Optional<Utf8>
>>;

UPSERT INTO series
(
    series_id,
    title,
    series_info,
    comment
)
SELECT
    series_id,
    title,
    series_info,
    comment
FROM AS_TABLE($seriesData);
```

```
EOS;

$seriesData = [
    [
        'series_id' => 1,
        'title' => 'IT Crowd',
        'series_info' => 'The IT Crowd is a British sitcom...',
        'comment' => null,
    ],
    [
        'series_id' => 2,
        'title' => 'Silicon Valley',
        'series_info' => 'Silicon Valley is an American comedy...',
        'comment' => 'lorem ipsum',
    ],
];

$ydb->table()->retryTransaction(
    function (Session $session) use ($yql, $seriesData) {
        return $session->prepare($yql)->execute([
            'seriesData' => $seriesData,
        ]);
    },
    true
);
```

Bulk upsert of data

YDB supports bulk insert of many rows without atomicity guarantees. The write is split into several independent transactions, each touching a single partition, with parallel execution. This makes the approach more efficient than plain YQL. On success, the `BulkUpsert` method guarantees that all data passed in the request is inserted.

> **Warning**
>
> When you load data to column-oriented tables using `BulkUpsert`, you must provide values for **all** columns, even `NULL` values.

Below are examples of using the YDB SDK built-in tools for bulk insert:

Go

Native SDK

Bulk upsert with native YDB data

```
package main

import (
  "context"
  "fmt"
  "os"
  "time"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/table"
  "github.com/ydb-platform/ydb-go-sdk/v3/table/types"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  type logMessage struct {
    App       string
    Host      string
    Timestamp time.Time
    HTTPCode  uint32
    Message   string
  }
  // prepare native go data
  const batchSize = 10000
  logs := make([]logMessage, 0, batchSize)
  for i := 0; i < batchSize; i++ {
    message := logMessage{
      App:       fmt.Sprintf("App_%d", i/256),
      Host:      fmt.Sprintf("192.168.0.%d", i%256),
      Timestamp: time.Now().Add(time.Millisecond * time.Duration(i%1000)),
      HTTPCode:  200,
    }
    if i%2 == 0 {
      message.Message = "GET / HTTP/1.1"
    } else {
      message.Message = "GET /images/logo.png HTTP/1.1"
    }
    logs = append(logs, message)
  }
  // execute bulk upsert with native ydb data
  err = db.Table().Do( // Do retry operation on errors with best effort
    ctx, // context manage exiting from Do
    func(ctx context.Context, s table.Session) (err error) { // retry operation
      rows := make([]types.Value, 0, len(logs))
      for _, msg := range logs {
        rows = append(rows, types.StructValue(
          types.StructFieldValue("App", types.UTF8Value(msg.App)),
          types.StructFieldValue("Host", types.UTF8Value(msg.Host)),
          types.StructFieldValue("Timestamp", types.TimestampValueFromTime(msg.Timestamp)),
          types.StructFieldValue("HTTPCode", types.Uint32Value(msg.HTTPCode)),
          types.StructFieldValue("Message", types.UTF8Value(msg.Message)),
        ))
      }
      return s.BulkUpsert(ctx, "/local/bulk_upsert_example", types.ListValue(rows...))
    },
  )
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
}
```

Bulk upsert `CSV` data

```
package main

import (
  "context"
  "fmt"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/table"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()

  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)

  csv := `skip row

id,val
42,"text42"
43,"text43"
44,hello
`

  err = db.Table().BulkUpsert(ctx, "/local/bulk_upsert_example", table.BulkUpsertDataCsv(
    []byte(csv),
    table.WithCsvHeader(),
    table.WithCsvSkipRows(2),
    table.WithCsvNullValue([]byte("hello")), // "hello" would be interpreted as NULL
  ))
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
}
```

Bulk upsert `Apache Arrow` data

In the following example, the arrow package is used to prepare the data.

```
package main

import (
  "bytes"
  "context"
  "fmt"
  "os"

  "github.com/apache/arrow-go/v18/arrow"
  "github.com/apache/arrow-go/v18/arrow/array"
  "github.com/apache/arrow-go/v18/arrow/ipc"
  "github.com/apache/arrow-go/v18/arrow/memory"
  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/table"
)

func main() {
  ctx := context.Background()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx) // cleanup resources

  mem := memory.NewGoAllocator()

  schema := arrow.NewSchema([]arrow.Field{
    {Name: "id", Type: arrow.PrimitiveTypes.Int64},
    {Name: "val", Type: arrow.BinaryTypes.String},
  }, nil)

  b := array.NewRecordBuilder(mem, schema)
  defer b.Release()

  b.Field(0).(*array.Int64Builder).AppendValues(
    []int64{123, 234}, nil)
```

```
  b.Field(1).(*array.StringBuilder).AppendValues(
    []string{"data1", "data2"}, nil)

  rec := b.NewRecordBatch()
  defer rec.Release()

  schemaPayload := ipc.GetSchemaPayload(rec.Schema(), mem)
  defer schemaPayload.Release()

  dataPayload, err := ipc.GetRecordBatchPayload(rec)
  if err != nil {
    panic(err)
  }
  defer dataPayload.Release()

  var schemaBuf bytes.Buffer
  _, err = schemaPayload.WritePayload(&schemaBuf)
  if err != nil {
    panic(err)
  }

  var dataBuf bytes.Buffer
  _, err = dataPayload.WritePayload(&dataBuf)
  if err != nil {
    panic(err)
  }

  err = db.Table().BulkUpsert(ctx, "/local/bulk_upsert_example", table.BulkUpsertDataArrow(
    dataBuf.Bytes(),
    table.WithArrowSchema(schemaBuf.Bytes()), // schema is required
  ))
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
}
```

database/sql

The YDB `database/sql` driver does not support non-transactional bulk insert.
For bulk insert, use transactional insert.

Java

Native SDK

```
private static final String TABLE_NAME = "bulk_upsert";
private static final int BATCH_SIZE = 1000;

public static void main(String[] args) {
    String connectionString = args[0];

    try (GrpcTransport transport = GrpcTransport.forConnectionString(connectionString)
            .withAuthProvider(NopAuthProvider.INSTANCE) // use anonymous credentials
            .build()) {

        // For bulk upsert, the full table path needs to be specified
        String tablePath = transport.getDatabase() + "/" + TABLE_NAME;
        try (TableClient tableClient = TableClient.newClient(transport).build()) {
            SessionRetryContext retryCtx = SessionRetryContext.create(tableClient).build();
            execute(retryCtx, tablePath);
        }
    }
}

public static void execute(SessionRetryContext retryCtx, String tablePath) {
    // table description
    StructType structType = StructType.of(
        "app", PrimitiveType.Text,
        "timestamp", PrimitiveType.Timestamp,
        "host", PrimitiveType.Text,
        "http_code", PrimitiveType.Uint32,
        "message", PrimitiveType.Text
    );

    // generate batch of records
    List<Value<?>> list = new ArrayList<>(50);
    for (int i = 0; i < BATCH_SIZE; i += 1) {
        // add a new row as a struct value
        list.add(structType.newValue(
            "app", PrimitiveValue.newText("App_" + String.valueOf(i / 256)),
            "timestamp", PrimitiveValue.newTimestamp(Instant.now().plusSeconds(i)),
            "host", PrimitiveValue.newText("192.168.0." + i % 256),
            "http_code", PrimitiveValue.newUint32(i % 113 == 0 ? 404 : 200),
            "message", PrimitiveValue.newText(i % 3 == 0 ? "GET / HTTP/1.1" : "GET /images/logo.png HTTP/1.
1")
        ));
    }

    // Create list of structs
    ListValue rows = ListType.of(structType).newValue(list);
```

```
    // Do retry operation on errors with best effort
    retryCtx.supplyStatus(
        session -> session.executeBulkUpsert(tablePath, rows, new BulkUpsertSettings())
    ).join().expectSuccess("bulk upsert problem");
}
```

JDBC

```
private static final int BATCH_SIZE = 1000;

public static void main(String[] args) {
    String connectionUrl = args[0];

    try (Connection conn = DriverManager.getConnection(connectionUrl)) {
        try (PreparedStatement ps = conn.prepareStatement(
                "BULK UPSERT INTO bulk_upsert (app, timestamp, host, http_code, message) VALUES (?, ?, ?, ?,
?);"
        )) {
            for (int i = 0; i < BATCH_SIZE; i += 1) {
                ps.setString(1, "App_" + String.valueOf(i / 256));
                ps.setTimestamp(2, Timestamp.from(Instant.now().plusSeconds(i)));
                ps.setString(3, "192.168.0." + i % 256);
                ps.setLong(4, i % 113 == 0 ? 404 : 200);
                ps.setString(5, i % 3 == 0 ? "GET / HTTP/1.1" : "GET /images/logo.png HTTP/1.1");
                ps.addBatch();
            }

            ps.executeBatch();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

In Spring Boot, Hibernate, JOOQ, and other ORM stacks on JDBC you can run native YQL (including from repositories and `@Query`). The driver tries to optimize large inserts; `UPDATE`, `INSERT`, `DELETE`, `UPSERT` through JDBC are batched on the driver side when appropriate.

Python

Native SDK

```
import posixpath
import ydb

def bulk_upsert(driver: ydb.Driver, path: str):
    column_types = (
        ydb.BulkUpsertColumns()
        .add_column("id", ydb.PrimitiveType.Uint64)
        .add_column("val", ydb.OptionalType(ydb.PrimitiveType.Utf8))
    )
    rows = [
        {"id": 1, "val": "1"},
        {"id": 2, "val": "2"},
        {"id": 3, "val": "3"},
    ]
    driver.table_client.bulk_upsert(posixpath.join(path, "tablename"), rows, column_types)
```

Native SDK (Asyncio)

```
import os
import posixpath
import ydb
import asyncio

async def bulk_upsert(driver: ydb.aio.Driver, path: str):
    column_types = (
        ydb.BulkUpsertColumns()
        .add_column("id", ydb.PrimitiveType.Uint64)
        .add_column("val", ydb.OptionalType(ydb.PrimitiveType.Utf8))
    )
    rows = [
        {"id": 1, "val": "1"},
        {"id": 2, "val": "2"},
        {"id": 3, "val": "3"},
    ]
    await driver.table_client.bulk_upsert(
        posixpath.join(path, "tablename"), rows, column_types
    )

async def main():
    async with ydb.aio.Driver(
        connection_string=os.environ["YDB_CONNECTION_STRING"],
        credentials=ydb.credentials_from_env_variables(),
    ) as driver:
        await driver.wait()
        await bulk_upsert(driver, "/local")
```

```
asyncio.run(main())
```

SQLAlchemy

```
import os
import sqlalchemy as sa
import ydb

engine = sa.create_engine(os.environ["YDB_SQLALCHEMY_URL"])
with engine.connect() as connection:
    dbapi_conn = connection.connection

    column_types = (
          ydb.BulkUpsertColumns()
          .add_column("id", ydb.PrimitiveType.Uint64)
          .add_column("val", ydb.OptionalType(ydb.PrimitiveType.Utf8))
      )
    rows = [
        {"id": 1, "val": "1"},
        {"id": 2, "val": "2"},
        {"id": 3, "val": "3"},
    ]

    dbapi_conn.bulk_upsert("tablename", rows, column_types)
```

C#

This functionality is not currently supported.

JavaScript

This functionality is not currently supported.

Rust

```
use ydb::{ydb_struct, AccessTokenCredentials, ClientBuilder, Value, YdbResult};

#[tokio::main]
async fn main() -> YdbResult<()> {
    let client = ClientBuilder::new_from_connection_string(
        "grpc://localhost:2136?database=local",
    )?
    .with_credentials(AccessTokenCredentials::from("..."))
    .client()?;

    client.wait().await?;

    let rows: Vec<Value> = vec![
        ydb_struct!(
            "id" => 1_u64,
            "val" => Value::Text("1".into()),
        ),
        ydb_struct!(
            "id" => 2_u64,
            "val" => Value::Text("2".into()),
        ),
        ydb_struct!(
            "id" => 3_u64,
            "val" => Value::Text("3".into()),
        ),
    ];

    client
        .table_client()
        .retry_execute_bulk_upsert("/local/tablename".into(), rows)
        .await?;

    Ok(())
}
```

PHP

```
<?php

use YdbPlatform\Ydb\Ydb;

$ydb = new Ydb($config);

$rows = [
    ['id' => 1, 'val' => '1'],
    ['id' => 2, 'val' => '2'],
    ['id' => 3, 'val' => '3'],
];

$ydb->table()->bulkUpsert('tablename', $rows, [
    'id'  => 'UINT64',
    'val' => 'UTF8',
]);
```

Setting up the transaction execution mode

To run your queries, first you need to specify the transaction execution mode in the YDB SDK.

Below are code examples showing the YDB SDK built-in tools to create an object for the *transaction execution mode*.

ImplicitTx

ImplicitTx mode allows executing a single query without explicit transaction control. The query is executed in its own implicit transaction that automatically commits if successful.

Go

Native SDK

```
package main

import (
  "context"
  "fmt"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/query"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  row, err := db.Query().QueryRow(ctx, "SELECT 1",
    query.WithTxControl(query.ImplicitTxControl()),
  )
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
  // work with row
  _ = row
}
```

database/sql

```
package main

import (
  "context"
  "database/sql"
  "fmt"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  defer connector.Close()

  db := sql.OpenDB(connector)
  defer db.Close()

  // ImplicitTx — query without an explicit transaction (auto-commit)
  row := db.QueryRowContext(ctx, "SELECT 1")
  var result int
  if err := row.Scan(&result); err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
}
```

Java

Native SDK

```
import tech.ydb.query.QueryClient;
import tech.ydb.query.tools.QueryReader;
import tech.ydb.query.tools.SessionRetryContext;

// ...
try (QueryClient queryClient = QueryClient.newClient(transport).build()) {
    SessionRetryContext retryCtx = SessionRetryContext.create(queryClient).build();
    QueryReader reader = retryCtx.supplyResult(
        session -> QueryReader.readFrom(session.createQuery("SELECT 1", TxMode.NONE))
    );
    // work with reader
}
```

JDBC

On the JDBC side, implicit mode corresponds to auto-commit (`Connection.setAutoCommit(true)` by default): each standalone statement runs as its own transaction and commits automatically. With `setAutoCommit(false)`, boundaries are defined by explicit `commit` / `rollback`.

In the **current** YDB JDBC driver, for **standalone read** calls, **snapshot** mode is used when **`setReadOnly(true)`** is set on the `Connection`. **Write** queries use **Serializable Read/Write** (the connection must not be read-only).

In **Spring**, the **`@Transactional(readOnly = true)`** attribute triggers `Connection.setReadOnly(true)` when the transaction starts, so snapshot is selected automatically for read-only flows. Hibernate, JOOQ, and other JDBC wrappers use the same connection — set read-only the same way (or via framework transaction settings if they propagate it to `Connection`).

Python

Native SDK

```
import ydb

def execute_query(pool: ydb.QuerySessionPool):
    pool.execute_with_retries("SELECT 1")
```

Native SDK (Asyncio)

```
import ydb

async def execute_query(pool: ydb.aio.QuerySessionPool):
    await pool.execute_with_retries("SELECT 1")
```

SQLAlchemy

```
import sqlalchemy as sa
from ydb_sqlalchemy import IsolationLevel

engine = sa.create_engine("yql+ydb://localhost:2136/local")
with engine.connect().execution_options(isolation_level=IsolationLevel.AUTOCOMMIT) as connection:
    result = connection.execute(sa.text("SELECT 1"))
```

C++

```
auto result = session.ExecuteQuery(
    "SELECT 1",
    NYdb::NQuery::TTxControl::NoTx()
).GetValueSync();
```

C# (.NET)

ADO.NET

```
using Ydb.Sdk.Ado;

await using var connection = await dataSource.OpenRetryableConnectionAsync();

// Execute without explicit transaction (auto-commit)
await using var command = new YdbCommand(connection) { CommandText = "SELECT 1" };
await command.ExecuteNonQueryAsync();
```

Entity Framework

```
using Microsoft.EntityFrameworkCore;

await using var context = await dbContextFactory.CreateDbContextAsync();
```

```
// Entity Framework auto-commit mode (no explicit transaction)
var result = await context.SomeEntities.FirstOrDefaultAsync();
```

linq2db

```
using LinqToDB;
using LinqToDB.Data;

using var db = new DataConnection(
    new DataOptions().UseConnectionString(
        "YDB",
        "Host=localhost;Port=2136;Database=/local;UseTls=false"
    )
);

// linq2db auto-commit mode (no explicit transaction)
var result = db.GetTable<Employee>().FirstOrDefault(e => e.Id == 1);
```

```
using Ydb.Sdk.Services.Query;

// ImplicitTx - single query without explicit transaction
var response = await queryClient.Exec("SELECT 1");
```

JavaScript

```
import { sql } from '@ydbjs/query';

// ...

// ImplicitTx - single query without explicit transaction
const result = await sql`SELECT 1`;
```

Rust

```
let mut qc = client.query_client();
// ImplicitTx is the default for QueryClient methods that execute a single SQL statement:
// the server picks isolation from SQL (SELECT → snapshot RO, DML → serializable RW).
let mut row = qc.query_row("SELECT 1 AS one").await?;
```

PHP

```
<?php

use YdbPlatform\Ydb\Ydb;

$config = [
    // YDB config
];

$ydb = new Ydb($config);
$result = $ydb->table()->query('SELECT 1;');
```

Serializable

Go

Native SDK

```
package main

import (
  "context"
  "fmt"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/query"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  row, err := db.Query().QueryRow(ctx, "SELECT 1",
    query.WithTxControl(query.SerializableReadWriteTxControl(query.CommitTx())),
```

```
  )
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
  // work with row
  _ = row
}
```

database/sql

```
package main

import (
  "context"
  "database/sql"
  "fmt"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/retry"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  defer connector.Close()

  db := sql.OpenDB(connector)
  defer db.Close()

  err = retry.DoTx(ctx, db,
    func(ctx context.Context, tx *sql.Tx) error {
      row := tx.QueryRowContext(ctx, "SELECT 1")
      var result int
      return row.Scan(&result)
    },
    retry.WithIdempotent(true),
    // Serializable Read-Write mode is used by default for transactions.
    // Alternatively, set it explicitly as shown below.
    retry.WithTxOptions(&sql.TxOptions{
      Isolation: sql.LevelSerializable,
      ReadOnly:  false,
    }),
  )
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
}
```

Java

Native SDK

```
import tech.ydb.query.QueryClient;
import tech.ydb.query.TxMode;
import tech.ydb.query.tools.QueryReader;
import tech.ydb.query.tools.SessionRetryContext;

// ...
try (QueryClient queryClient = QueryClient.newClient(transport).build()) {
    SessionRetryContext retryCtx = SessionRetryContext.create(queryClient).build();
    QueryReader reader = retryCtx.supplyResult(
        session -> QueryReader.readFrom(session.createQuery("SELECT 1", TxMode.SERIALIZABLE_RW))
    );
    // work with reader
}
```

JDBC

For **standalone** JDBC calls, the **current** driver implementation uses **Serializable Read/Write** — including from Spring Boot, ORMs, and other JDBC wrappers. **Snapshot** for reads is enabled via `setReadOnly(true)` (see ImplicitTx). Auto-commit semantics and explicit transactions are described there as well.

Python

Native SDK

```python
import ydb

def execute_query(pool: ydb.QuerySessionPool):
    def callee(session: ydb.QuerySession):
        with session.transaction(ydb.QuerySerializableReadWrite()).execute(
            "SELECT 1",
            commit_tx=True,
        ) as result_sets:
            pass

    pool.retry_operation_sync(callee)
```

Native SDK (Asyncio)

```python
import ydb

async def execute_query(pool: ydb.aio.QuerySessionPool):
    async def callee(session):
        async with session.transaction(tx_mode=ydb.QuerySerializableReadWrite()) as tx:
            async with await tx.execute("SELECT 1", commit_tx=True) as result_sets:
                pass

    await pool.retry_operation_async(callee)
```

SQLAlchemy

```python
import sqlalchemy as sa
from ydb_sqlalchemy import IsolationLevel

engine = sa.create_engine("yql+ydb://localhost:2136/local")
with engine.connect().execution_options(isolation_level=IsolationLevel.SERIALIZABLE) as connection:
    result = connection.execute(sa.text("SELECT 1"))
```

C++

```cpp
auto settings = NYdb::NQuery::TTxSettings::SerializableRW();
auto result = session.ExecuteQuery(
    "SELECT 1",
    NYdb::NQuery::TTxControl::BeginTx(settings).CommitTx()
).GetValueSync();
```

C# (.NET)

ADO.NET

```csharp
using Ydb.Sdk.Ado;

// Serializable Read-Write mode is used by default
await _ydbDataSource.ExecuteInTransactionAsync(async ydbConnection =>
    {
        var ydbCommand = ydbConnection.CreateCommand();
        ydbCommand.CommandText = """
                                 UPSERT INTO episodes (series_id, season_id, episode_id, title, air_date)
                                 VALUES (2, 5, 13, "Test Episode", Date("2018-08-27"))
                                 """;
        await ydbCommand.ExecuteNonQueryAsync();
        ydbCommand.CommandText = """
                                 INSERT INTO episodes(series_id, season_id, episode_id, title, air_date)
                                 VALUES
                                     (2, 5, 21, "Test 21", Date("2018-08-27")),
                                     (2, 5, 22, "Test 22", Date("2018-08-27"))
                                 """;
        await ydbCommand.ExecuteNonQueryAsync();
    }
);
```

Entity Framework

```csharp
var strategy = db.Database.CreateExecutionStrategy();

// Serializable Read-Write mode is used by default
strategy.ExecuteInTransaction(
    db,
    ctx =>
    {
        ctx.Users.AddRange(
            new User { Name = "Alex", Email = "alex@example.com" },
            new User { Name = "Kirill", Email = "kirill@example.com" }
        );
```

```
        ctx.SaveChanges();

        var users = ctx.Users.OrderBy(u => u.Id).ToList();
        Console.WriteLine("Users in database:");
        foreach (var user in users)
            Console.WriteLine($"- {user.Id}: {user.Name} ({user.Email})");
    },
    ctx => ctx.Users.Any(u => u.Email == "alex@example.com")
        && ctx.Users.Any(u => u.Email == "kirill@example.com")
);
```

linq2db

```
using LinqToDB;
using LinqToDB.Data;

// linq2db uses Serializable isolation by default
using var db = new DataConnection(
    new DataOptions().UseConnectionString(
        "YDB",
        "Host=localhost;Port=2136;Database=/local;UseTls=false"
    )
);

// Serializable Read-Write mode is used by default
await using var tr = await db.BeginTransactionAsync();

await db.InsertAsync(new Episode
{
    SeriesId = 2, SeasonId = 5, EpisodeId = 13, Title = "Test Episode", AirDate = new DateTime(2018, 08,
27)
});
await db.InsertAsync(new Episode
    { SeriesId = 2, SeasonId = 5, EpisodeId = 21, Title = "Test 21", AirDate = new DateTime(2018, 08, 27)
});
await db.InsertAsync(new Episode
    { SeriesId = 2, SeasonId = 5, EpisodeId = 22, Title = "Test 22", AirDate = new DateTime(2018, 08, 27)
});

await tr.CommitAsync();
```

```
using Ydb.Sdk.Services.Query;

// Serializable Read-Write mode is used by default
var response = await queryClient.Exec("SELECT 1");
```

JavaScript

```
import { sql } from '@ydbjs/query';

// ...

// Serializable Read-Write mode is used by default
await sql.begin({ idempotent: true }, async (tx) => {
    return await tx`SELECT 1`;
});

// Or explicitly specify transaction mode
await sql.begin({ isolation: 'serializableReadWrite', idempotent: true }, async (tx) => {
    return await tx`SELECT 1`;
});
```

Rust

```
use ydb::{QueryTransactionOptions, QueryTxMode};

let qc = client
    .query_client()
    .clone_with_transaction_options(
        QueryTransactionOptions::new().with_mode(QueryTxMode::SerializableReadWrite),
    );
qc.retry_transaction(async |tx| {
    tx.query_row("SELECT 1 AS one").await?;
    Ok(())
})
.await?;
```

PHP

```
<?php

use YdbPlatform\Ydb\Ydb;

$config = [
```

```
    // YDB config
];

$ydb = new Ydb($config);
$result = $ydb->table()->retryTransaction(
  function (Session $session) {
      return $session->query('SELECT 1 AS value;');
  },
  true,
  null,
  ['tx_mode' => 'serializable_read_write']
);
```

Online Read-Only

Go

Native SDK

```
package main

import (
  "context"
  "fmt"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/query"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  row, err := db.Query().QueryRow(ctx, "SELECT 1",
    query.WithTxControl(
      query.OnlineReadOnlyTxControl(query.WithInconsistentReads()),
    ),
  )
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
  // work with row
  _ = row
}
```

database/sql

```
package main

import (
  "context"
  "database/sql"
  "fmt"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/retry"
  "github.com/ydb-platform/ydb-go-sdk/v3/table"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  defer connector.Close()

  db := sql.OpenDB(connector)
  defer db.Close()

  err = retry.Do(
```

```
    ydb.WithTxControl(ctx, table.OnlineReadOnlyTxControl(table.WithInconsistentReads())),
    db,
    func(ctx context.Context, conn *sql.Conn) error {
      row := conn.QueryRowContext(ctx, "SELECT 1")
      var result int
      return row.Scan(&result)
    },
    retry.WithIdempotent(true),
  )
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
}
```

Java

Native SDK

```
import tech.ydb.query.QueryClient;
import tech.ydb.query.TxMode;
import tech.ydb.query.tools.QueryReader;
import tech.ydb.query.tools.SessionRetryContext;

// ...
try (QueryClient queryClient = QueryClient.newClient(transport).build()) {
    SessionRetryContext retryCtx = SessionRetryContext.create(queryClient).build();
    QueryReader reader = retryCtx.supplyResult(
        session -> QueryReader.readFrom(session.createQuery("SELECT 1", TxMode.ONLINE_RO))
    );
    // work with reader
}
```

JDBC

Online Read-Only from the Query API is **not** configured separately for **standalone** JDBC calls. For **snapshot** reads, call `Connection.setReadOnly(true)` (in Spring — `@Transactional(readOnly = true)`). For writes, see ImplicitTx.

Python

Native SDK

```
import ydb

def execute_query(pool: ydb.QuerySessionPool):
    def callee(session: ydb.QuerySession):
        with session.transaction(ydb.QueryOnlineReadOnly()).execute(
            "SELECT 1",
            commit_tx=True,
        ) as result_sets:
            pass

    pool.retry_operation_sync(callee)
```

Native SDK (Asyncio)

```
import ydb

async def execute_query(pool: ydb.aio.QuerySessionPool):
    async def callee(session):
        async with session.transaction(tx_mode=ydb.QueryOnlineReadOnly()) as tx:
            async with await tx.execute("SELECT 1", commit_tx=True) as result_sets:
                pass

    await pool.retry_operation_async(callee)
```

SQLAlchemy

```
import sqlalchemy as sa
from ydb_sqlalchemy import IsolationLevel

engine = sa.create_engine("yql+ydb://localhost:2136/local")
with engine.connect().execution_options(isolation_level=IsolationLevel.ONLINE_READONLY) as connection:
    result = connection.execute(sa.text("SELECT 1"))
```

C++

```
auto settings = NYdb::NQuery::TTxSettings::OnlineRO();
auto result = session.ExecuteQuery(
    "SELECT 1",
    NYdb::NQuery::TTxControl::BeginTx(settings).CommitTx()
).GetValueSync();
```

C# (.NET)

ADO.NET

```
using Ydb.Sdk.Ado;

await using var connection = await dataSource.OpenConnectionAsync();
await using var transaction = await connection.BeginTransactionAsync(TransactionMode.OnlineRo);
await using var command = new YdbCommand(connection) { CommandText = "SELECT 1" };
await using var reader = await command.ExecuteReaderAsync();
await transaction.CommitAsync();
```

Entity Framework

Entity Framework does not support OnlineRo mode directly.
Use ydb-dotnet-sdk or ADO.NET for this isolation level.

linq2db

linq2db does not support OnlineRo mode directly.
Use ydb-dotnet-sdk or ADO.NET for this isolation level.

```
using Ydb.Sdk.Ado;
using Ydb.Sdk.Services.Query;

var response = await queryClient.ReadAllRows("SELECT 1", txMode: TransactionMode.OnlineRo);
```

JavaScript

```
import { sql } from '@ydbjs/query';

// ...

await sql.begin({ isolation: 'onlineReadOnly', idempotent: true }, async (tx) => {
    return await tx`SELECT 1`;
});
```

Rust

```
use ydb::QueryTxMode;

let mut qc = client.query_client();

// Online RO — consistent reads (allow_inconsistent_reads = false)
let mut row = qc
    .query_row("SELECT 1 AS one")
    .with_tx_mode(QueryTxMode::OnlineReadOnly)
    .await?;

// Online inconsistent RO — higher throughput (allow_inconsistent_reads = true)
let mut row = qc
    .query_row("SELECT 1 AS one")
    .with_tx_mode(QueryTxMode::OnlineReadOnlyInconsistent)
    .await?;
```

PHP

```
<?php

use YdbPlatform\Ydb\Ydb;

$config = [
    // YDB config
];

$ydb = new Ydb($config);
$result = $ydb->table()->retrySession(function (Session $session) {
  $query = $session->newQuery('SELECT 1 AS value;');
  $query->beginTx('online_read_only');
  return $query->execute();
}, true);
```

Stale Read-Only

Go

Native SDK

```
package main

import (
  "context"
  "fmt"
  "os"
```

```
  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/query"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  row, err := db.Query().QueryRow(ctx, "SELECT 1",
    query.WithTxControl(query.StaleReadOnlyTxControl()),
  )
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
  // work with row
  _ = row
}
```

database/sql

```
package main

import (
  "context"
  "database/sql"
  "fmt"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/retry"
  "github.com/ydb-platform/ydb-go-sdk/v3/table"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  defer connector.Close()

  db := sql.OpenDB(connector)
  defer db.Close()

  err = retry.Do(
    ydb.WithTxControl(ctx, table.StaleReadOnlyTxControl()),
    db,
    func(ctx context.Context, conn *sql.Conn) error {
      row := conn.QueryRowContext(ctx, "SELECT 1")
      var result int
      return row.Scan(&result)
    },
    retry.WithIdempotent(true),
  )
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
}
```

Java

Native SDK

```
import tech.ydb.query.QueryClient;
import tech.ydb.query.TxMode;
import tech.ydb.query.tools.QueryReader;
import tech.ydb.query.tools.SessionRetryContext;

// ...
try (QueryClient queryClient = QueryClient.newClient(transport).build()) {
    SessionRetryContext retryCtx = SessionRetryContext.create(queryClient).build();
    QueryReader reader = retryCtx.supplyResult(
        session -> QueryReader.readFrom(session.createQuery("SELECT 1", TxMode.STALE_RO))
```

```
    );
    // work with reader
}
```

JDBC

Stale Read-Only from the Query API is **not** configured separately for **standalone** JDBC calls. For **snapshot** reads, call `Connection.setReadOnly(true)` (in Spring — `@Transactional(readOnly = true)`). For writes, see ImplicitTx.

Python

Native SDK

```
import ydb

def execute_query(pool: ydb.QuerySessionPool):
    def callee(session: ydb.QuerySession):
        with session.transaction(ydb.QueryStaleReadOnly()).execute(
            "SELECT 1",
            commit_tx=True,
        ) as result_sets:
            pass

    pool.retry_operation_sync(callee)
```

Native SDK (Asyncio)

```
import ydb

async def execute_query(pool: ydb.aio.QuerySessionPool):
    async def callee(session):
        async with session.transaction(tx_mode=ydb.QueryStaleReadOnly()) as tx:
            async with await tx.execute("SELECT 1", commit_tx=True) as result_sets:
                pass

    await pool.retry_operation_async(callee)
```

SQLAlchemy

```
import sqlalchemy as sa
from ydb_sqlalchemy import IsolationLevel

engine = sa.create_engine("yql+ydb://localhost:2136/local")
with engine.connect().execution_options(isolation_level=IsolationLevel.STALE_READONLY) as connection:
    result = connection.execute(sa.text("SELECT 1"))
```

C++

```
auto settings = NYdb::NQuery::TTxSettings::StaleRO();
auto result = session.ExecuteQuery(
    "SELECT 1",
    NYdb::NQuery::TTxControl::BeginTx(settings).CommitTx()
).GetValueSync();
```

C# (.NET)

ADO.NET

```
using Ydb.Sdk.Ado;
using Ydb.Sdk.Services.Query;

await using var connection = await dataSource.OpenConnectionAsync();
await using var transaction = await connection.BeginTransactionAsync(TransactionMode.StaleRo);
await using var command = new YdbCommand(connection) { CommandText = "SELECT 1", Transaction = transactio
n };
await using var reader = await command.ExecuteReaderAsync();
await transaction.CommitAsync();
```

Entity Framework

> Entity Framework does not support StaleRo mode directly.
> Use ydb-dotnet-sdk or ADO.NET for this isolation level.

linq2db

> linq2db does not support StaleRo mode directly.
> Use ydb-dotnet-sdk or ADO.NET for this isolation level.

```
using Ydb.Sdk.Ado;
using Ydb.Sdk.Services.Query;
```

```
var response = await queryClient.ReadAllRows("SELECT 1", txMode: TransactionMode.StaleRo);
```

JavaScript

```
import { sql } from '@ydbjs/query';

// ...

await sql.begin({ isolation: 'staleReadOnly', idempotent: true }, async (tx) => {
    return await tx`SELECT 1`;
});
```

Rust

```
use ydb::QueryTxMode;

let mut qc = client.query_client();
// Stale Read-Only is supported only for QueryClient methods that execute a single SQL statement (not for ret
ry_transaction).
let mut row = qc
    .query_row("SELECT 1 AS one")
    .with_tx_mode(QueryTxMode::StaleReadOnly)
    .await?;
```

PHP

```
<?php

use YdbPlatform\Ydb\Ydb;

$config = [
    // YDB config
];

$ydb = new Ydb($config);
$result = $ydb->table()->retrySession(function (Session $session) {
  $query = $session->newQuery('SELECT 1 AS value;');
  $query->beginTx('stale_read_only');
  return $query->execute();
}, true);
```

Snapshot Read-Only

Go

Native SDK

```
package main

import (
  "context"
  "fmt"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/query"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  row, err := db.Query().QueryRow(ctx, "SELECT 1",
    query.WithTxControl(query.SnapshotReadOnlyTxControl()),
  )
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
  // work with row
  _ = row
}
```

database/sql

```
package main

import (
  "context"
  "database/sql"
```

```
  "fmt"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/retry"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  defer connector.Close()

  db := sql.OpenDB(connector)
  defer db.Close()

  // Snapshot Read-Only — consistent reads at a point in time
  err = retry.DoTx(ctx, db, func(ctx context.Context, tx *sql.Tx) error {
    row := tx.QueryRowContext(ctx, "SELECT 1")
    var result int
    return row.Scan(&result)
  }, retry.WithIdempotent(true), retry.WithTxOptions(&sql.TxOptions{
    Isolation: sql.LevelSnapshot,
    ReadOnly:  true,
  }))
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
}
```

Java

Native SDK

```
import tech.ydb.query.QueryClient;
import tech.ydb.query.TxMode;
import tech.ydb.query.tools.QueryReader;
import tech.ydb.query.tools.SessionRetryContext;

// ...
try (QueryClient queryClient = QueryClient.newClient(transport).build()) {
    SessionRetryContext retryCtx = SessionRetryContext.create(queryClient).build();
    QueryReader reader = retryCtx.supplyResult(
        session -> QueryReader.readFrom(session.createQuery("SELECT 1", TxMode.SNAPSHOT_RO))
    );
    // work with reader
}
```

JDBC

For **standalone read-only** JDBC calls, **snapshot** mode is enabled when `Connection.setReadOnly(true)` is set on the connection (in Spring, `@Transactional(readOnly = true)` propagates this to the driver when the transaction starts). See ImplicitTx for details.

Python

Native SDK

```
import ydb

def execute_query(pool: ydb.QuerySessionPool):
    def callee(session: ydb.QuerySession):
        with session.transaction(ydb.QuerySnapshotReadOnly()).execute(
            "SELECT 1",
            commit_tx=True,
        ) as result_sets:
            pass

    pool.retry_operation_sync(callee)
```

Native SDK (Asyncio)

```
import ydb

async def execute_query(pool: ydb.aio.QuerySessionPool):
    async def callee(session):
        async with session.transaction(tx_mode=ydb.QuerySnapshotReadOnly()) as tx:
            async with await tx.execute("SELECT 1", commit_tx=True) as result_sets:
```

```
                pass

    await pool.retry_operation_async(callee)
```

SQLAlchemy

```
import sqlalchemy as sa
from ydb_sqlalchemy import IsolationLevel

engine = sa.create_engine("yql+ydb://localhost:2136/local")
with engine.connect().execution_options(isolation_level=IsolationLevel.SNAPSHOT_READONLY) as connection:
    result = connection.execute(sa.text("SELECT 1"))
```

C++

```
auto settings = NYdb::NQuery::TTxSettings::SnapshotRO();
auto result = session.ExecuteQuery(
    "SELECT 1",
    NYdb::NQuery::TTxControl::BeginTx(settings).CommitTx()
).GetValueSync();
```

C# (.NET)

ADO.NET

```
using Ydb.Sdk.Ado;

await using var connection = await dataSource.OpenConnectionAsync();
await using var transaction = await connection.BeginTransactionAsync(TransactionMode.SnapshotRo);
await using var command = new YdbCommand(connection) { CommandText = "SELECT 1" };
await using var reader = await command.ExecuteReaderAsync();
await transaction.CommitAsync();
```

Entity Framework

Entity Framework does not support Snapshot Read-Only mode directly.
Use ydb-dotnet-sdk or ADO.NET for this isolation level.

linq2db

linq2db does not support Snapshot Read-Only mode directly.
Use ydb-dotnet-sdk or ADO.NET for this isolation level.

```
using Ydb.Sdk.Ado;
using Ydb.Sdk.Services.Query;

var response = await queryClient.ReadAllRows("SELECT 1", TransactionMode.SnapshotRo);
```

JavaScript

```
import { sql } from '@ydbjs/query';

// ...

await sql.begin({ isolation: 'snapshotReadOnly', idempotent: true }, async (tx) => {
    return await tx`SELECT 1`;
});
```

Rust

```
use ydb::{QueryTransactionOptions, QueryTxMode};

let mut qc = client.query_client();
qc.query_row("SELECT 1 AS one")
    .with_tx_mode(QueryTxMode::SnapshotReadOnly)
    .await?;

let qc = qc.clone_with_transaction_options(
    QueryTransactionOptions::new().with_mode(QueryTxMode::SnapshotReadOnly),
);
qc.retry_transaction(async |tx| {
    tx.query_row("SELECT 1 AS one").await?;
    Ok(())
})
.await?;
```

PHP

```
<?php

use YdbPlatform\Ydb\Ydb;
```

```php
$config = [
    // YDB config
];

$ydb = new Ydb($config);
$result = $ydb->table()->retryTransaction(
  function (Session $session) {
      return $session->query('SELECT 1 AS value;');
  },
  true,
  null,
  ['tx_mode' => 'snapshot_read_only']
);
```

Snapshot Read-Write

Go

Native SDK

```go
package main

import (
  "context"
  "fmt"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/query"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  row, err := db.Query().QueryRow(ctx, "SELECT 1",
    query.WithTxControl(query.SnapshotReadWriteTxControl(query.CommitTx())),
  )
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
  // work with row
  _ = row
}
```

database/sql

```go
package main

import (
  "context"
  "database/sql"
  "fmt"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/retry"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAccessTokenCredentials(os.Getenv("YDB_TOKEN")),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  defer connector.Close()

  db := sql.OpenDB(connector)
  defer db.Close()

  // Snapshot Read-Write — consistent reads at a point in time with writes allowed
  err = retry.DoTx(ctx, db, func(ctx context.Context, tx *sql.Tx) error {
```

```
    row := tx.QueryRowContext(ctx, "SELECT 1")
    var result int
    return row.Scan(&result)
  }, retry.WithIdempotent(true), retry.WithTxOptions(&sql.TxOptions{
    Isolation: sql.LevelSnapshot,
    ReadOnly:  false,
  }))
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
}
```

Java

Native SDK

```
import tech.ydb.query.QueryClient;
import tech.ydb.query.TxMode;
import tech.ydb.query.tools.QueryReader;
import tech.ydb.query.tools.SessionRetryContext;

// ...
try (QueryClient queryClient = QueryClient.newClient(transport).build()) {
    SessionRetryContext retryCtx = SessionRetryContext.create(queryClient).build();
    QueryReader reader = retryCtx.supplyResult(
        session -> QueryReader.readFrom(session.createQuery("SELECT 1", TxMode.SNAPSHOT_RW))
    );
    // work with reader
}
```

JDBC

Snapshot Read-Write from the Query API is **not** set separately for **standalone** JDBC calls; for **writes**, the **current** driver implementation uses **Serializable Read/Write** (see ImplicitTx and Serializable).

Python

Native SDK

```
import ydb

def execute_query(pool: ydb.QuerySessionPool):
    def callee(session: ydb.QuerySession):
        with session.transaction(ydb.QuerySnapshotReadWrite()).execute(
            "SELECT 1",
            commit_tx=True,
        ) as result_sets:
            pass

    pool.retry_operation_sync(callee)
```

Native SDK (Asyncio)

```
import ydb

async def execute_query(pool: ydb.aio.QuerySessionPool):
    async def callee(session):
        async with session.transaction(tx_mode=ydb.QuerySnapshotReadWrite()) as tx:
            async with await tx.execute("SELECT 1", commit_tx=True) as result_sets:
                pass

    await pool.retry_operation_async(callee)
```

SQLAlchemy

```
import sqlalchemy as sa
from ydb_sqlalchemy import IsolationLevel

engine = sa.create_engine("yql+ydb://localhost:2136/local")
with engine.connect().execution_options(isolation_level=IsolationLevel.SNAPSHOT_READWRITE) as connection:
    result = connection.execute(sa.text("SELECT 1"))
```

C++

```
auto settings = NYdb::NQuery::TTxSettings::SnapshotRW();
auto result = session.ExecuteQuery(
    "SELECT 1",
    NYdb::NQuery::TTxControl::BeginTx(settings).CommitTx()
).GetValueSync();
```

C# (.NET)

ADO.NET

```
await _ydbDataSource.ExecuteInTransactionAsync(async ydbConnection =>
    {
```

```
        var ydbCommand = ydbConnection.CreateCommand();
        ydbCommand.CommandText = """
                                 UPSERT INTO episodes (series_id, season_id, episode_id, title, air_date)
                                 VALUES (2, 5, 13, "Test Episode", Date("2018-08-27"))
                                 """;
        await ydbCommand.ExecuteNonQueryAsync();
        ydbCommand.CommandText = """
                                 INSERT INTO episodes(series_id, season_id, episode_id, title, air_date)
                                 VALUES
                                     (2, 5, 21, "Test 21", Date("2018-08-27")),
                                     (2, 5, 22, "Test 22", Date("2018-08-27"))
                                 """;
        await ydbCommand.ExecuteNonQueryAsync();
    }, TransactionMode.SnapshotRw
);
```

EF

```
var strategy = db.Database.CreateExecutionStrategy();

strategy.Execute(() =>
    {
        using var ctx = new AppDbContext(options);
        using var tr = ctx.Database.BeginTransaction(IsolationLevel.Snapshot);

        ctx.Users.AddRange(
            new User { Name = "Alex", Email = "alex@example.com" },
            new User { Name = "Kirill", Email = "kirill@example.com" }
        );

        ctx.SaveChanges();

        var users = ctx.Users.OrderBy(u => u.Id).ToList();
        Console.WriteLine("Users in database:");
        foreach (var user in users)
            Console.WriteLine($"- {user.Id}: {user.Name} ({user.Email})");
     }
);
```

linq2db

```
await using var db = new MyYdb(BuildOptions());
await using var tr = await db.BeginTransactionAsync(IsolationLevel.Snapshot);

await db.InsertAsync(new Episode
{
    SeriesId = 2, SeasonId = 5, EpisodeId = 13, Title = "Test Episode", AirDate = new DateTime(2018, 08,
27)
});
await db.InsertAsync(new Episode
    { SeriesId = 2, SeasonId = 5, EpisodeId = 21, Title = "Test 21", AirDate = new DateTime(2018, 08, 27)
});
await db.InsertAsync(new Episode
    { SeriesId = 2, SeasonId = 5, EpisodeId = 22, Title = "Test 22", AirDate = new DateTime(2018, 08, 27)
});

await tr.CommitAsync();
```

```
using Ydb.Sdk.Ado;
using Ydb.Sdk.Services.Query;

var response = await queryClient.ReadAllRows("SELECT 1", TransactionMode.SnapshotRw);
```

JavaScript

```
import { sql } from '@ydbjs/query';

// ...

await sql.begin({ isolation: 'snapshotReadWrite', idempotent: true }, async (tx) => {
    return await tx`SELECT 1`;
});
```

Rust

```
use ydb::{QueryTransactionOptions, QueryTxMode};

let mut qc = client.query_client();
qc.query_row("SELECT 1 AS one")
    .with_tx_mode(QueryTxMode::SnapshotReadWrite)
    .await?;
```

```rust
let qc = qc.clone_with_transaction_options(
    QueryTransactionOptions::new().with_mode(QueryTxMode::SnapshotReadWrite),
);
qc.retry_transaction(async |tx| {
    tx.query_row("SELECT 1 AS one").await?;
    Ok(())
})
.await?;
```

PHP

Snapshot Read-Write mode is not supported in the PHP SDK.

Configuring Time to Live (TTL)

This section contains recipes for configuration of table's TTL with YDB SDK.

Enabling TTL for an existing table

In the example below, the items of the `mytable` table will be deleted an hour after the time set in the `created_at` column:

C++

```
session.AlterTable(
  "mytable",
  TAlterTableSettings()
    .BeginAlterTtlSettings()
      .Set("created_at", TDuration::Hours(1))
    .EndAlterTtlSettings()
);
```

Go

```
err := session.AlterTable(ctx, "mytable",
  options.WithSetTimeToLiveSettings(
    options.NewTTLSettings().ColumnDateType("created_at").ExpireAfter(time.Hour),
  ),
)
```

Python

```
session.alter_table('mytable', set_ttl_settings=ydb.TtlSettings().with_date_type_column('created_at', 3600))
```

JavaScript

This section is under development.

Java

```
AlterTableSettings settings = new AlterTableSettings()
        .setTableTtl(TableTtl.dateTimeColumn("created_at", 3600));

session.alterTable("mytable", settings).join().expectSuccess();
```

The example below shows how to use the `modified_at` column with a numeric type (`Uint32`) as a TTL column. The column value is interpreted as the number of seconds since the Unix epoch:

C++

```
session.AlterTable(
  "mytable",
  TAlterTableSettings()
    .BeginAlterTtlSettings()
      .Set("modified_at", TTtlSettings::EUnit::Seconds, TDuration::Hours(1))
    .EndAlterTtlSettings()
);
```

Go

```
err := session.AlterTable(ctx, "mytable",
  options.WithSetTimeToLiveSettings(
    options.NewTTLSettings().ColumnSeconds("modified_at").ExpireAfter(time.Hour),
  ),
)
```

Python

```
session.alter_table('mytable', set_ttl_settings=ydb.TtlSettings().with_value_since_unix_epoch('modified_at',
UNIT_SECONDS, 3600))
```

JavaScript

This section is under development.

Java

```
AlterTableSettings settings = new AlterTableSettings()
        .setTableTtl(TableTtl.valueSinceUnixEpoch(
                "modified_at",
                TableTtl.TtlUnit.SECONDS,
                3600
        ));

session.alterTable("mytable", settings).join().expectSuccess();
```

Enabling data eviction to S3-compatible external storage

> **Warning**
>
> Supported only for column-oriented tables. Support for row-oriented tables is currently under development.

To enable data eviction, an external data source object that describes a connection to the external storage is needed. Refer to YQL recipe for examples of creating an external data source.

In the following example, rows of the table `mytable` will be moved to the bucket described in the external data source `/Root/s3_cold_data` one hour after the time recorded in the column `created_at` and will be deleted after 24 hours:

C++

```
session.AlterTable(
    "mytable",
    TAlterTableSettings()
        .BeginAlterTtlSettings()
            .Set("created_at", {
                    TTtlTierSettings(TDuration::Hours(1), TTtlEvictToExternalStorageAction("/Root/s3_cold_dat
a")),
                    TTtlTierSettings(TDuration::Hours(24), TTtlDeleteAction("/Root/s3_cold_data"))
                })
        .EndAlterTtlSettings()
);
```

Java

This functionality is not currently supported.

Go

This functionality is not currently supported.

Python

This functionality is not currently supported.

JavaScript

This section is under development.

Enabling TTL for a newly created table

For a newly created table, you can pass TTL settings along with the table description:

C++

```
session.CreateTable(
  "mytable",
  TTableBuilder()
    .AddNullableColumn("id", EPrimitiveType::Uint64)
    .AddNullableColumn("expire_at", EPrimitiveType::Timestamp)
    .SetPrimaryKeyColumn("id")
    .SetTtlSettings("expire_at")
    .Build()
);
```

Go

```
err := session.CreateTable(ctx, "mytable",
  options.WithColumn("id", types.Optional(types.TypeUint64)),
  options.WithColumn("expire_at", types.Optional(types.TypeTimestamp)),
  options.WithTimeToLiveSettings(
    options.NewTTLSettings().ColumnDateType("expire_at"),
  ),
)
```

Python

```
session.create_table(
  'mytable',
  ydb.TableDescription()
    .with_column(ydb.Column('id', ydb.OptionalType(ydb.DataType.Uint64)))
    .with_column(ydb.Column('expire_at', ydb.OptionalType(ydb.DataType.Timestamp)))
    .with_primary_key('id')
    .with_ttl(ydb.TtlSettings().with_date_type_column('expire_at'))
)
```

JavaScript

This section is under development.

Java

```
TableDescription description = TableDescription.newBuilder()
        .addNullableColumn("id", PrimitiveType.Uint64)
        .addNullableColumn("expire_at", PrimitiveType.Timestamp)
```

```
        .setPrimaryKey("id")
        .setTtlSettings(TableTtl.dateTimeColumn("expire_at", 0))
        .build();

session.createTable("mytable", description).join().expectSuccess();
```

Disabling TTL

C++

```
session.AlterTable(
  "mytable",
  TAlterTableSettings()
    .BeginAlterTtlSettings()
      .Drop()
    .EndAlterTtlSettings()
);
```

Go

```
err := session.AlterTable(ctx, "mytable",
  options.WithDropTimeToLive(),
)
```

Python

```
session.alter_table('mytable', drop_ttl_settings=True)
```

JavaScript

This section is under development.

Java

```
AlterTableSettings settings = new AlterTableSettings()
        .setTableTtl(TableTtl.notSet());

session.alterTable("mytable", settings).join().expectSuccess();
```

Getting TTL settings

The current TTL settings can be obtained from the table description:

C++

```
auto desc = session.DescribeTable("mytable").GetValueSync().GetTableDescription();
auto ttl = desc.GetTtlSettings();
```

Go

```
desc, err := session.DescribeTable(ctx, "mytable")
if err != nil {
  // process error
}
ttl := desc.TimeToLiveSettings
```

Python

```
desc = session.describe_table('mytable')
ttl = desc.ttl_settings
```

JavaScript

This section is under development.

Java

```
TableTtl ttl = session.describeTable("mytable").join().getValue().getTableDescription().getTableTtl();
```

Vector search

This section contains code recipes in different programming languages for vector search tasks using the YDB SDK.

The following operations are covered in detail:

- Vector search
  - Connecting to YDB
  - Creating a table
  - Inserting vectors
  - Adding an index
  - Search by vector
  - Full example

This recipe creates a text store with the following structure:

Field	Description
`id`	Text identifier
`document`	Text
`embedding`	Vector representation of the text

The recipe assumes that `embedding` is already available.

Connecting to YDB

This section describes the minimum steps required to run queries in YDB.
For more information on connecting to YDB, see Initialize the driver.

C++

```
#include <ydb-cpp-sdk/client/query/client.h>

int main() {
    auto driverConfig = NYdb::CreateFromEnvironment(endpoint + "/?database=" + database);
    NYdb::TDriver driver(driverConfig);
    NYdb::NQuery::TQueryClient client(driver);

    // ...
}
```

Go

```
package main

import (
    "context"
    "os"

    "github.com/ydb-platform/ydb-go-sdk/v3"
)

func main() {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()
    db, err := ydb.Open(ctx,
        os.Getenv("YDB_CONNECTION_STRING"),
    )
    if err != nil {
        panic(err)
    }
    defer db.Close(ctx)
}
```

Java

For queries, use `QueryClient` and `SessionRetryContext` (see driver initialization). Below is minimal setup and a client for the YQL Query Service:

```
import tech.ydb.core.grpc.GrpcTransport;
import tech.ydb.query.QueryClient;
import tech.ydb.query.tools.SessionRetryContext;

String connectionString = System.getenv().getOrDefault("YDB_CONNECTION_STRING", "grpc://localhost:2136/loca
l");

try (GrpcTransport transport = GrpcTransport.forConnectionString(connectionString).build();
    QueryClient queryClient = QueryClient.newClient(transport).build()) {

    SessionRetryContext retryCtx = SessionRetryContext.create(queryClient).build();
    // retryCtx.supplyResult(session -> QueryReader.readFrom(session.createQuery(...)))
}
```

Python

Native SDK

To run queries, create a `ydb.QuerySessionPool`.

```
import ydb

driver = ydb.Driver(
    endpoint=ydb_endpoint,
    database=ydb_database,
    credentials=ydb_credentials,
)
driver.wait(5, fail_fast=True)
pool = ydb.QuerySessionPool(driver)
```

Native SDK (Asyncio)

To run queries, create a `ydb.aio.QuerySessionPool`:

```
import asyncio
import ydb

async def main():
    async with ydb.aio.Driver(
        endpoint=ydb_endpoint,
        database=ydb_database,
        credentials=ydb_credentials,
    ) as driver:
        await driver.wait(5, fail_fast=True)
        pool = ydb.aio.QuerySessionPool(driver)
        # ... use pool ...

asyncio.run(main())
```

C#

Vector search examples for the .NET SDK are not available yet.

This functionality is not currently supported.

Track progress or vote for support: ydb-dotnet-sdk#657

JavaScript

```
import { Driver } from '@ydbjs/core'
import { query, unsafe, identifier } from '@ydbjs/query'

const driver = new Driver('grpc://localhost:2136/local')
await driver.ready()
const sql = query(driver)
```

Rust

```
use ydb::{ClientBuilder, YdbResult};

let client = ClientBuilder::new_from_connection_string(connection_string)?.client()?;
client.wait().await?;
let mut qc = client.query_client().clone_with_idempotent_operations(true);
```

PHP

Vector search examples for the PHP SDK are not available yet.

This functionality is not currently supported.

Track progress or vote for support: ydb-php-sdk#268

Creating a table

First, create a table to store documents and their vector representations.

Table structure:

Column name	Data type	Description
`id`	`Utf8`	Document identifier
`document`	`Utf8`	Document text
`embedding`	`String`	Vector representation of the document

> **Warning**
>
> The `String` type is used to store vectors. For details, see the exact vector search.

C++

```
void CreateVectorTable(NYdb::NQuery::TQueryClient& client, const std::string& tableName)
{
    std::string query = std::format(R"(
        CREATE TABLE IF NOT EXISTS `{}` (
            id Utf8,
            document Utf8,
            embedding String,
            PRIMARY KEY (id)
        ))", tableName);

    NYdb::NStatusHelpers::ThrowOnError(client.RetryQuerySync([&](NYdb::NQuery::TSession session) {
        return session.ExecuteQuery(query, NYdb::NQuery::TTxControl::NoTx()).ExtractValueSync();
    }));

    std::cout << "Vector table created: " << tableName << std::endl;
}
```

Go

```
func createVectorTable(ctx context.Context, db *ydb.Driver, tableName string) error {
    query := fmt.Sprintf(`
        CREATE TABLE IF NOT EXISTS %s (
            id Utf8,
            document Utf8,
            embedding String,
            PRIMARY KEY (id)
        );`, "`"+tableName+"`")

    return db.Query().Exec(ctx, query)
}
```

Java

```
import tech.ydb.common.transaction.TxMode;
import tech.ydb.query.tools.QueryReader;
import tech.ydb.query.tools.SessionRetryContext;
import tech.ydb.table.query.Params;

void createVectorTable(SessionRetryContext retryCtx, String tableName) {
    String query = String.format("""
            CREATE TABLE IF NOT EXISTS `%s` (
                id Utf8,
                document Utf8,
                embedding String,
                PRIMARY KEY (id)
            );""", tableName);

    retryCtx.supplyResult(session -> QueryReader.readFrom(
            session.createQuery(query, TxMode.NONE, Params.empty())
    )).join().getValue();

    System.out.println("Vector table created: " + tableName);
}
```

Python

Native SDK

```
import ydb

def create_vector_table(pool: ydb.QuerySessionPool, table_name: str) -> None:
    query = f"""
    CREATE TABLE IF NOT EXISTS `{table_name}` (
        id Utf8,
        document Utf8,
        embedding String,
        PRIMARY KEY (id)
    );"""

    pool.execute_with_retries(query)

    print(f"Vector table {table_name} created")
```

Native SDK (Asyncio)

```
import ydb

async def create_vector_table(pool: ydb.aio.QuerySessionPool, table_name: str) -> None:
    query = f"""
    CREATE TABLE IF NOT EXISTS `{table_name}` (
        id Utf8,
        document Utf8,
        embedding String,
        PRIMARY KEY (id)
    );"""
```

```
    await pool.execute_with_retries(query)

    print(f"Vector table {table_name} created")
```

C#

Vector search examples for the .NET SDK are not available yet.

This functionality is not currently supported.

Track progress or vote for support: ydb-dotnet-sdk#657

JavaScript

```
await sql`CREATE TABLE IF NOT EXISTS `table_name` (
    id Utf8,
    document Utf8,
    embedding String,
    PRIMARY KEY (id)
);`
```

Rust

```
qc.exec(format!(
    "CREATE TABLE IF NOT EXISTS `{table_name}` (
        id Utf8,
        document Utf8,
        embedding String,
        PRIMARY KEY (id)
    );", tableName,
))
.await?;
```

PHP

Vector search examples for the PHP SDK are not available yet.

This functionality is not currently supported.

Track progress or vote for support: ydb-php-sdk#268

Inserting vectors

To insert vectors, prepare and run a parameterized YQL query. Parameterization unifies inserts for different data.

The query uses the container type `List<Struct<...>>` (a list of structs), so you can pass any number of objects in one round trip.

In YDB tables, vectors are stored as serialized byte sequences. **Prefer performing this conversion on the client.** Alternatively, delegate conversion to the server using Knn UDF. Examples for both approaches follow.

C++

Recommended approach

```
std::string ConvertVectorToBytes(const std::vector<float>& vector)
{
    std::string result;
    for (const auto& value : vector) {
        const char* bytes = reinterpret_cast<const char*>(&value);
        result += std::string(bytes, sizeof(float));
    }
    return result + "\x01";
}

void InsertItemsAsBytes(
    NYdb::NQuery::TQueryClient& client,
    const std::string& tableName,
    const std::vector<TItem>& items)
{
    std::string query = std::format(R"(
        DECLARE $items AS List<Struct<
            id: Utf8,
            document: Utf8,
            embedding: String
        >>;
        UPSERT INTO `{0}`
        (
            id,
            document,
            embedding
        )
        SELECT
            id,
            document,
            embedding,
        FROM AS_TABLE($items);
    )", tableName);

    NYdb::TParamsBuilder paramsBuilder;
    auto& valueBuilder = paramsBuilder.AddParam("$items");
```

```
    valueBuilder.BeginList();
    for (const auto& item : items) {
        valueBuilder.AddListItem();
        valueBuilder.BeginStruct();
        valueBuilder.AddMember("id").Utf8(item.Id);
        valueBuilder.AddMember("document").Utf8(item.Document);
        valueBuilder.AddMember("embedding").String(ConvertVectorToBytes(item.Embedding));
        valueBuilder.EndStruct();
    }
    valueBuilder.EndList();
    valueBuilder.Build();

    NYdb::NStatusHelpers::ThrowOnError(client.RetryQuerySync([params = paramsBuilder.Build(), &query](NYdb::N
Query::TSession session) {
        return session.ExecuteQuery(query, NYdb::NQuery::TTxControl::BeginTx(NYdb::NQuery::TTxSettings::Seria
lizableRW()).CommitTx(), params).ExtractValueSync();
    }));

    std::cout << items.size() << " items inserted" << std::endl;
}
```

Note

The `ConvertVectorToBytes` function assumes a little-endian byte order on the client (e.g. x86_64). If a different byte order is used, adapt the `ConvertVectorToBytes` function accordingly.

Alternative approach

Warning

Passing vectors as `List<Float>` and converting on the YQL side with `Knn::ToBinaryStringFloat` has lower performance than client-side byte array encoding.

```
void InsertItemsAsFloatList(
    NYdb::NQuery::TQueryClient& client,
    const std::string& tableName,
    const std::vector<TItem>& items)
{
    std::string query = std::format(R"(
        DECLARE $items AS List<Struct<
            id: Utf8,
            document: Utf8,
            embedding: List<Float>
        >>;

        UPSERT INTO `{}`
        (
            id,
            document,
            embedding
        )
        SELECT
            id,
            document,
            Untag(Knn::ToBinaryStringFloat(embedding), "FloatVector"),
        FROM AS_TABLE($items);
    )", tableName);

    NYdb::TParamsBuilder paramsBuilder;
    auto& valueBuilder = paramsBuilder.AddParam("$items");
    valueBuilder.BeginList();
    for (const auto& item : items) {
        valueBuilder.AddListItem();
        valueBuilder.BeginStruct();
        valueBuilder.AddMember("id").Utf8(item.Id);
        valueBuilder.AddMember("document").Utf8(item.Document);
        valueBuilder.AddMember("embedding").BeginList();
        for (const auto& value : item.Embedding) {
            valueBuilder.AddListItem().Float(value);
        }
        valueBuilder.EndList();
        valueBuilder.EndStruct();
    }
    valueBuilder.EndList();
    valueBuilder.Build();

    NYdb::NStatusHelpers::ThrowOnError(client.RetryQuerySync([params = paramsBuilder.Build(), &query](NYdb::N
Query::TSession session) {
        return session.ExecuteQuery(query, NYdb::NQuery::TTxControl::BeginTx(NYdb::NQuery::TTxSettings::Seria
lizableRW()).CommitTx(), params).ExtractValueSync();
    }));

    std::cout << items.size() << " items inserted" << std::endl;
}
```

Go

The function converts a float32 vector to the binary layout and runs a parameterized query:

```
import (
    "encoding/binary"
    "math"
)

func convertVectorToBytes(vector []float32) []byte {
    buf := make([]byte, len(vector)*4+1)
    for i, v := range vector {
    binary.LittleEndian.PutUint32(buf[i*4:], math.Float32bits(v))
    }
    buf[len(buf)-1] = 0x01
    return buf
}

func insertItems(ctx context.Context, db *ydb.Driver, tableName string, items []Item) error {
    query := fmt.Sprintf(`
    DECLARE $items AS List<Struct<
        id: Utf8,
        document: Utf8,
        embedding: String
    >>;

    UPSERT INTO %s
    (id, document, embedding)
    SELECT id, document, embedding
    FROM AS_TABLE($items);
    `, "`"+tableName+"`")

    rows := make([]types.Value, 0, len(items))
    for _, item := range items {
    rows = append(rows, types.StructValue(
        types.StructFieldValue("id", types.UTF8Value(item.ID)),
        types.StructFieldValue("document", types.UTF8Value(item.Document)),
        types.StructFieldValue("embedding", types.BytesValue(convertVectorToBytes(item.Embedding))),
    ))
    }

    return db.Query().Exec(ctx, query,
    query.WithParameters(
        ydb.ParamsBuilder().Param("$items").BeginList().AddItems(rows...).EndList().Build(),
    ),
    )
}
```

Java

Recommended approach

```
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.util.ArrayList;
import java.util.List;

import tech.ydb.common.transaction.TxMode;
import tech.ydb.query.tools.QueryReader;
import tech.ydb.query.tools.SessionRetryContext;
import tech.ydb.table.query.Params;
import tech.ydb.table.values.ListType;
import tech.ydb.table.values.ListValue;
import tech.ydb.table.values.PrimitiveType;
import tech.ydb.table.values.PrimitiveValue;
import tech.ydb.table.values.StructType;
import tech.ydb.table.values.Value;

byte[] convertVectorToBytes(float[] vector) {
    ByteBuffer bb = ByteBuffer.allocate(vector.length * Float.BYTES + 1).order(ByteOrder.LITTLE_ENDIAN);
    for (float v : vector) {
        bb.putFloat(v);
    }
    bb.put((byte) 0x01);
    return bb.array();
}

void insertItemsAsBytes(SessionRetryContext retryCtx, String tableName, List<Item> items) {
    String query = String.format("""
            DECLARE $items AS List<Struct<
                id: Utf8,
                document: Utf8,
                embedding: String
            >>;

            UPSERT INTO `%s`
            (
                id,
                document,
                embedding
```

```
            )
            SELECT
                id,
                document,
                embedding,
            FROM AS_TABLE($items);""", tableName);

    StructType rowType = StructType.of(
            "id", PrimitiveType.Text,
            "document", PrimitiveType.Text,
            "embedding", PrimitiveType.Bytes
    );

    List<Value<?>> rows = new ArrayList<>(items.size());
    for (Item item : items) {
        rows.add(rowType.newValue(
                "id", PrimitiveValue.newText(item.id()),
                "document", PrimitiveValue.newText(item.document()),
                "embedding", PrimitiveValue.newBytes(convertVectorToBytes(item.embedding()))
        ));
    }

    ListValue itemsParam = ListType.of(rowType).newValue(rows);
    Params params = Params.of("$items", itemsParam);

    retryCtx.supplyResult(session -> QueryReader.readFrom(
            session.createQuery(query, TxMode.SERIALIZABLE_RW, params)
    )).join().getValue();

    System.out.println(items.size() + " items inserted");
}

// record Item(String id, String document, float[] embedding) {}
```

Note

The `ConvertVectorToBytes` function assumes a little-endian byte order on the client (e.g. x86_64). If a different byte order is used, adapt the `ConvertVectorToBytes` function accordingly.

Alternative approach

Warning

Passing vectors as `List<Float>` and converting on the YQL side with `Knn::ToBinaryStringFloat` has lower performance than client-side byte array encoding.

A variant that passes vector components as `List<Float>` and converts on the YQL side with `Knn::ToBinaryStringFloat` — same pattern as the alternative Python and C++ examples below:

```
import java.util.ArrayList;
import java.util.List;

import tech.ydb.common.transaction.TxMode;
import tech.ydb.query.tools.QueryReader;
import tech.ydb.query.tools.SessionRetryContext;
import tech.ydb.table.query.Params;
import tech.ydb.table.values.ListType;
import tech.ydb.table.values.ListValue;
import tech.ydb.table.values.PrimitiveType;
import tech.ydb.table.values.PrimitiveValue;
import tech.ydb.table.values.StructType;
import tech.ydb.table.values.Value;

void insertItemsAsFloatList(SessionRetryContext retryCtx, String tableName, List<Item> items) {
    String query = String.format("""
            DECLARE $items AS List<Struct<
                id: Utf8,
                document: Utf8,
                embedding: List<Float>
            >>;

            UPSERT INTO `%s`
            (
                id,
                document,
                embedding
            )
            SELECT
                id,
                document,
                Untag(Knn::ToBinaryStringFloat(embedding), "FloatVector"),
            FROM AS_TABLE($items);""", tableName);

    StructType rowType = StructType.of(
            "id", PrimitiveType.Text,
            "document", PrimitiveType.Text,
            "embedding", ListType.of(PrimitiveType.Float)
```

```
    );

    List<Value<?>> rows = new ArrayList<>(items.size());
    for (Item item : items) {
        Value<?>[] emb = new Value<?>[item.embedding().length];
        for (int i = 0; i < item.embedding().length; i++) {
            emb[i] = PrimitiveValue.newFloat(item.embedding()[i]);
        }
        ListValue embList = ListType.of(PrimitiveType.Float).newValueOwn(emb);
        rows.add(rowType.newValue(
                "id", PrimitiveValue.newText(item.id()),
                "document", PrimitiveValue.newText(item.document()),
                "embedding", embList
        ));
    }

    ListValue itemsParam = ListType.of(rowType).newValue(rows);
    Params params = Params.of("$items", itemsParam);

    retryCtx.supplyResult(session -> QueryReader.readFrom(
            session.createQuery(query, TxMode.SERIALIZABLE_RW, params)
    )).join().getValue();

    System.out.println(items.size() + " items inserted");
}

// record Item(String id, String document, float[] embedding) {}
```

Python

Recommended approach

Use `ydb.convert_floats_to_embedding_bytes` from Python SDK 3.33.1 or later to serialize vectors.

The method accepts an array of dictionaries `items`, where each dictionary contains `id` (identifier), `document` (text), and `embedding` (vector representation of the text, pre-serialized to a byte sequence).

The example below creates `items_struct_type = ydb.StructType()` with field types. To pass a list of such structs, wrap it in `ydb.ListType`: `ydb.ListType(items_struct_type)`.

Native SDK

```
import ydb

def insert_items_vector_as_bytes(
    pool: ydb.QuerySessionPool,
    table_name: str,
    items: list[dict],
) -> None:
    query = f"""
    DECLARE $items AS List<Struct<
        id: Utf8,
        document: Utf8,
        embedding: String
    >>;

    UPSERT INTO `{table_name}`
    (
        id,
        document,
        embedding
    )
    SELECT
        id,
        document,
        embedding,
    FROM AS_TABLE($items);
    """

    items_struct_type = ydb.StructType()
    items_struct_type.add_member("id", ydb.PrimitiveType.Utf8)
    items_struct_type.add_member("document", ydb.PrimitiveType.Utf8)
    items_struct_type.add_member("embedding", ydb.PrimitiveType.String)

    for item in items:
        item["embedding"] = ydb.convert_floats_to_embedding_bytes(item["embedding"])

    pool.execute_with_retries(
        query, {"$items": (items, ydb.ListType(items_struct_type))}
    )

    print(f"{len(items)} items inserted")
```

Native SDK (Asyncio)

```
import ydb

async def insert_items_vector_as_bytes(
    pool: ydb.aio.QuerySessionPool,
    table_name: str,
```

```
    items: list[dict],
) -> None:
    query = f"""
    DECLARE $items AS List<Struct<
        id: Utf8,
        document: Utf8,
        embedding: String
    >>;

    UPSERT INTO `{table_name}`
    (
        id,
        document,
        embedding
    )
    SELECT
        id,
        document,
        embedding,
    FROM AS_TABLE($items);
    """

    items_struct_type = ydb.StructType()
    items_struct_type.add_member("id", ydb.PrimitiveType.Utf8)
    items_struct_type.add_member("document", ydb.PrimitiveType.Utf8)
    items_struct_type.add_member("embedding", ydb.PrimitiveType.String)

    for item in items:
        item["embedding"] = ydb.convert_floats_to_embedding_bytes(item["embedding"])

    await pool.execute_with_retries(
        query, {"$items": (items, ydb.ListType(items_struct_type))}
    )

    print(f"{len(items)} items inserted")
```

Alternative approach

> **Warning**
>
> Passing vectors as `List<Float>` and converting on the YQL side with `Knn::ToBinaryStringFloat` has lower performance than client-side byte array encoding.

The method accepts an array of dictionaries `items`, where each dictionary contains `id` (identifier), `document` (text), and `embedding` (vector representation of the text).

The example below creates `items_struct_type = ydb.StructType()` with field types. To pass a list of such structs, wrap it in `ydb.ListType`: `ydb.ListType(items_struct_type)`.

Native SDK

```
import ydb

def insert_items_vector_as_float_list(
    pool: ydb.QuerySessionPool,
    table_name: str,
    items: list[dict],
) -> None:
    query = f"""
    DECLARE $items AS List<Struct<
        id: Utf8,
        document: Utf8,
        embedding: List<Float>
    >>;

    UPSERT INTO `{table_name}`
    (
        id,
        document,
        embedding
    )
    SELECT
        id,
        document,
        Untag(Knn::ToBinaryStringFloat(embedding), "FloatVector"),
    FROM AS_TABLE($items);
    """

    items_struct_type = ydb.StructType()
    items_struct_type.add_member("id", ydb.PrimitiveType.Utf8)
    items_struct_type.add_member("document", ydb.PrimitiveType.Utf8)
    items_struct_type.add_member("embedding", ydb.ListType(ydb.PrimitiveType.Float))

    pool.execute_with_retries(
        query, {"$items": (items, ydb.ListType(items_struct_type))}
    )

    print(f"{len(items)} items inserted")
```

Native SDK (Asyncio)

```
import ydb

async def insert_items_vector_as_float_list(
    pool: ydb.aio.QuerySessionPool,
    table_name: str,
    items: list[dict],
) -> None:
    query = f"""
    DECLARE $items AS List<Struct<
        id: Utf8,
        document: Utf8,
        embedding: List<Float>
    >>;

    UPSERT INTO `{table_name}`
    (
        id,
        document,
        embedding
    )
    SELECT
        id,
        document,
        Untag(Knn::ToBinaryStringFloat(embedding), "FloatVector"),
    FROM AS_TABLE($items);
    """

    items_struct_type = ydb.StructType()
    items_struct_type.add_member("id", ydb.PrimitiveType.Utf8)
    items_struct_type.add_member("document", ydb.PrimitiveType.Utf8)
    items_struct_type.add_member("embedding", ydb.ListType(ydb.PrimitiveType.Float))

    await pool.execute_with_retries(
        query, {"$items": (items, ydb.ListType(items_struct_type))}
    )

    print(f"{len(items)} items inserted")
```

C#

Vector search examples for the .NET SDK are not available yet.

This functionality is not currently supported.

Track progress or vote for support: ydb-dotnet-sdk#657

JavaScript

Recommended approach

```
function convertVectorToBytes(vector) {
    const bytes = new Uint8Array(vector.length * 4 + 1);
    const view = new DataView(bytes.buffer);

    for (let i = 0; i < vector.length; i++) {
        view.setFloat32(i * 4, vector[i], true);
    }

    bytes[bytes.length - 1] = 0x01;
    return bytes;
}

const items = [
    {
    id: "first_doc",
    document: "My Document",
    embedding: convertVectorToBytes(new Float32Array([1.5, 2.5, 3.5]))
    }
]

await sql`
    UPSERT INTO `table_name` (id, document, embedding)
    SELECT id, document, embedding,
    FROM AS_TABLE($items);`
```

Alternative approach

> **Warning**
>
> Passing vectors as `List<Float>` and converting on the YQL side with `Knn::ToBinaryStringFloat` has lower performance than client-side byte array encoding.

```
const items = [
    {
```

```
    id: "first_doc",
    document: "My Document",
    embedding: new Float32Array([1.5, 2.5, 3.5])
    }
]

await sql`
    UPSERT INTO `table_name` (id, document, embedding)
    SELECT id, document, Untag(Knn::ToBinaryStringFloat(embedding), "FloatVector"),
    FROM AS_TABLE($items);`
```

Rust

```
use ydb::{Bytes, Value, ydb_struct};

fn convert_vector_to_bytes(vector: &[f32]) -> Bytes {
    let mut buf = Vec::with_capacity(vector.len() * 4 + 1);
    for v in vector {
        buf.extend_from_slice(&v.to_le_bytes());
    }
    buf.push(0x01);
    Bytes::from(buf)
}

let example = ydb_struct!(
    "id" => "",
    "document" => "",
    "embedding" => Bytes::default(),
);
let rows: Vec<Value> = items
    .iter()
    .map(|item| ydb_struct!(
        "id" => item.id,
        "document" => item.document,
        "embedding" => convert_vector_to_bytes(&item.embedding),
    ))
    .collect();
let list = Value::list_from(example, rows)?;

qc.exec(insert_query).param("$items", list).await?;
```

PHP

Vector search examples for the PHP SDK are not available yet.

This functionality is not currently supported.

Track progress or vote for support: ydb-php-sdk#268

Adding an index

A vector index supports efficient approximate nearest-neighbor search. See vector indexes.

Adding an index takes two steps:

1. Create a temporary index.
2. Save the temporary index as permanent.

This works both for the first build and for rebuilds when an index already exists.

Available strategies:

- `similarity=cosine`;
- `similarity=inner_product`;
- `distance=cosine`;
- `distance=euclidean`;
- `distance=manhattan`.

Each strategy defines the function used for subsequent search. For details on these functions, see distance and similarity functions.

Parameters for the `vector_kmeans_tree` index type are described in the vector index.

C++

```
void AddIndex(
    NYdb::TDriver& driver,
    NYdb::NQuery::TQueryClient& client,
    const std::string& database,
    const std::string& tableName,
    const std::string& indexName,
    const std::string& strategy)
{
    std::string query = std::format(R"(
        ALTER TABLE `{0}`
        ADD INDEX {1}__temp
        GLOBAL USING vector_kmeans_tree
        ON (embedding)
        WITH (
            {2}
        );
```

```
    )", tableName, indexName, strategy);

    NYdb::NStatusHelpers::ThrowOnError(client.RetryQuerySync([&](NYdb::NQuery::TSession session) {
        return session.ExecuteQuery(query, NYdb::NQuery::TTxControl::NoTx()).ExtractValueSync();
    }));

    NYdb::NTable::TTableClient tableClient(driver);
    NYdb::NStatusHelpers::ThrowOnError(tableClient.RetryOperationSync([&](NYdb::NTable::TSession session) {
        return session.AlterTable(database + "/" + tableName, NYdb::NTable::TAlterTableSettings()
            .AppendRenameIndexes(NYdb::NTable::TRenameIndex{
                .SourceName_ = indexName + "__temp",
                .DestinationName_ = indexName,
                .ReplaceDestination_ = true
            })
        ).ExtractValueSync();
    }));

    std::cout << "Table index `" << indexName << "` for table `" << tableName << "` added" << std::endl;
}
```

Go

```
func addVectorIndex(
    ctx context.Context,
    db *ydb.Driver,
    tableName, indexName, strategy string
) error {
    tempIndexName := indexName + "__temp"
    query := fmt.Sprintf(`
        ALTER TABLE %s
        ADD INDEX %s
        GLOBAL USING vector_kmeans_tree
        ON (embedding)
        WITH (
        %s
        );
    `, "`"+tableName+"`", tempIndexName, strategy)

    if err := db.Query().Exec(ctx, query); err != nil {
        return err
    }

    return db.Table().Do(ctx, func(ctx context.Context, s table.Session) error {
        return s.AlterTable(ctx, path.Join(db.Name(), tableName),
        options.WithRenameIndex(tempIndexName, indexName, true),
        )
    })
}
```

Java

```
import tech.ydb.core.grpc.GrpcTransport;
import tech.ydb.common.transaction.TxMode;
import tech.ydb.query.tools.QueryReader;
import tech.ydb.query.tools.SessionRetryContext;
import tech.ydb.table.query.Params;
import tech.ydb.table.settings.AlterTableSettings;

void addVectorIndex(
        GrpcTransport transport,
        SessionRetryContext queryRetry,
        SessionRetryContext tableRetry,
        String tableName,
        String indexName,
        String strategy) {

    String tempIndexName = indexName + "__temp";
    String query = String.format("""
            ALTER TABLE `%s`
            ADD INDEX %s
            GLOBAL USING vector_kmeans_tree
            ON (embedding)
            WITH (
                %s
            );
            """, tableName, tempIndexName, strategy);

    queryRetry.supplyResult(session -> QueryReader.readFrom(
            session.createQuery(query, TxMode.NONE, Params.empty())
    )).join().getValue();

    String tablePath = transport.getDatabase() + "/" + tableName;
    AlterTableSettings settings = new AlterTableSettings()
            .addRenameIndex(tempIndexName, indexName, true);

    tableRetry.supplyStatus(session -> session.alterTable(tablePath, settings))
            .join()
            .expectSuccess("alter table rename index");

    System.out.println("Table index `" + indexName + "` for table `" + tableName + "` added");
```

```
}

// SessionRetryContext tableRetry = SessionRetryContext.create(TableClient.newClient(transport).build()).buil
d();
```

Python

Native SDK

```
import ydb

def add_vector_index(
    pool: ydb.QuerySessionPool,
    driver: ydb.Driver,
    table_name: str,
    index_name: str,
    strategy: str,
):
    temp_index_name = f"{index_name}__temp"
    query = f"""
    ALTER TABLE `{table_name}`
    ADD INDEX {temp_index_name}
    GLOBAL USING vector_kmeans_tree
    ON (embedding)
    WITH (
        {strategy}
    );
    """

    pool.execute_with_retries(query)
    driver.table_client.alter_table(
        f"{driver._driver_config.database}/{table_name}",
        rename_indexes=[
            ydb.RenameIndexItem(
                source_name=temp_index_name,
                destination_name=f"{index_name}",
                replace_destination=True,
            ),
        ],
    )
    print(f"Table index {index_name} created.")
```

Native SDK (Asyncio)

```
import ydb

async def add_vector_index(
    pool: ydb.aio.QuerySessionPool,
    driver: ydb.aio.Driver,
    table_name: str,
    index_name: str,
    strategy: str
):
    temp_index_name = f"{index_name}__temp"
    query = f"""
    ALTER TABLE `{table_name}`
    ADD INDEX {temp_index_name}
    GLOBAL USING vector_kmeans_tree
    ON (embedding)
    WITH (
        {strategy}
    );
    """

    await pool.execute_with_retries(query)
    await driver.table_client.alter_table(
        f"{driver._driver_config.database}/{table_name}",
        rename_indexes=[
            ydb.RenameIndexItem(
                source_name=temp_index_name,
                destination_name=f"{index_name}",
                replace_destination=True,
            ),
        ],
    )

    print(f"Table index {index_name} created.")
```

C#

Vector search examples for the .NET SDK are not available yet.

This functionality is not currently supported.

Track progress or vote for support: ydb-dotnet-sdk#657

JavaScript

Vector index creation and a full end-to-end program for the JavaScript SDK are not available yet.

This functionality is not currently supported.

Track progress or vote for support: ydb-js-sdk#627

Rust

```
let temp_index_name = format!("{index_name}__temp");
qc.exec(format!(
    "ALTER TABLE `{table_name}`
    ADD INDEX {temp_index_name}
    GLOBAL USING vector_kmeans_tree
    ON (embedding)
    WITH ({strategy}, vector_type=\"Float\", vector_dimension={dimension},
            levels={levels}, clusters={clusters});"
))
.await?;
qc.exec(format!(
    "ALTER TABLE `{table_name}` RENAME INDEX `{temp_index_name}` TO `{index_name}`;"
))
.await?;
```

PHP

Vector search examples for the PHP SDK are not available yet.

This functionality is not currently supported.

Track progress or vote for support: ydb-php-sdk#268

Search by vector

Vector search uses a special YQL query where you define a similarity or distance function.
Supported values:

- `CosineSimilarity`;
- `InnerProductSimilarity`;
- `CosineDistance`;
- `ManhattanDistance`;
- `EuclideanDistance`.

For details, see distance and similarity functions.

You can specify the index name. If set, the query will include the `VIEW index_name` expression so that the vector index is used for search.

The method returns a list of dictionaries with the fields `id`, `document`, and `score` (a number indicating similarity or distance to the query vector).

C++

Recommended approach

```
std::vector<TResultItem> SearchItemsAsBytes(
    NYdb::NQuery::TQueryClient& client,
    const std::string& tableName,
    const std::vector<float>& embedding,
    const std::string& strategy,
    std::uint64_t limit,
    std::uint64_t topClusters = 10,
    const std::optional<std::string>& indexName = std::nullopt)
{
    std::string viewIndex = indexName ? "VIEW " + *indexName : "";
    std::string sortOrder = strategy.ends_with("Similarity") ? "DESC" : "ASC";

    std::string query = std::format(R"(
        PRAGMA ydb.KMeansTreeSearchTopSize = "{5}";
        DECLARE $embedding as String;
        SELECT
            id,
            document,
            Knn::{2}(embedding, $embedding) as score
        FROM {0} {1}
        ORDER BY score {3}
        LIMIT {4};
    )", tableName, viewIndex, strategy, sortOrder, limit, topClusters);

    auto params = NYdb::TParamsBuilder()
        .AddParam("$embedding")
            .String(ConvertVectorToBytes(embedding))
            .Build()
        .Build();

    std::vector<TResultItem> result;

    NYdb::NStatusHelpers::ThrowOnError(client.RetryQuerySync([params, &query, &result](NYdb::NQuery::TSession
session) {
        auto execResult = session.ExecuteQuery(query, NYdb::NQuery::TTxControl::BeginTx(NYdb::NQuery::TTxSett
ings::SerializableRW()).CommitTx(), params).ExtractValueSync();
        if (execResult.IsSuccess()) {
            auto parser = execResult.GetResultSetParser(0);
            while (parser.TryNextRow()) {
                result.push_back({
```

```
                    .Id = *parser.ColumnParser(0).GetOptionalUtf8(),
                    .Document = *parser.ColumnParser(1).GetOptionalUtf8(),
                    .Score = *parser.ColumnParser(2).GetOptionalFloat()
                });
            }
        }
        return execResult;
    }));

    return result;
}
```

Alternative approach

> **Warning**
>
> Passing vectors as `List<Float>` and converting on the YQL side with `Knn::ToBinaryStringFloat` has lower performance than client-side byte array encoding.

```
std::vector<TResultItem> SearchItemsAsFloatList(
    NYdb::NQuery::TQueryClient& client,
    const std::string& tableName,
    const std::vector<float>& embedding,
    const std::string& strategy,
    std::uint64_t limit,
    std::uint64_t topClusters = 10,
    const std::optional<std::string>& indexName = std::nullopt)
{
    std::string viewIndex = indexName ? "VIEW " + *indexName : "";
    std::string sortOrder = strategy.ends_with("Similarity") ? "DESC" : "ASC";

    std::string query = std::format(R"(
        PRAGMA ydb.KMeansTreeSearchTopSize = "{5}";
        DECLARE $embedding as List<Float>;

        $TargetEmbedding = Knn::ToBinaryStringFloat($embedding);

        SELECT
            id,
            document,
            Knn::{2}(embedding, $TargetEmbedding) as score
        FROM {0} {1}
        ORDER BY score
        {3}
        LIMIT {4};
    )", tableName, viewIndex, strategy, sortOrder, limit, topClusters);

    NYdb::TParamsBuilder paramsBuilder;
    auto& valueBuilder = paramsBuilder.AddParam("$embedding");
    valueBuilder.BeginList();
    for (auto value : embedding) {
        valueBuilder.AddListItem().Float(value);
    }
    valueBuilder.EndList().Build();

    std::vector<TResultItem> result;

    NYdb::NStatusHelpers::ThrowOnError(client.RetryQuerySync([params = paramsBuilder.Build(), &query, &resul
t](NYdb::NQuery::TSession session) {
        auto execResult = session.ExecuteQuery(query, NYdb::NQuery::TTxControl::BeginTx(NYdb::NQuery::TTxSett
ings::SerializableRW()).CommitTx(), params).ExtractValueSync();
        if (execResult.IsSuccess()) {
            auto parser = execResult.GetResultSetParser(0);
            while (parser.TryNextRow()) {
                result.push_back({
                    .Id = *parser.ColumnParser(0).GetOptionalUtf8(),
                    .Document = *parser.ColumnParser(1).GetOptionalUtf8(),
                    .Score = *parser.ColumnParser(2).GetOptionalFloat()
                });
            }
        }
        return execResult;
    }));

    return result;
}
```

Go

```
type ResultItem struct {
    ID       string
    Document string
    Score    float32
}

func searchItems(
    ctx context.Context,
```

```
    db *ydb.Driver,
    tableName string,
    embedding []float32,
    strategy string,
    limit int,
    indexName string,
) ([]ResultItem, error) {
    viewIndex := ""
    if indexName != "" {
        viewIndex = "VIEW " + indexName
    }
    sortOrder := "DESC"
    if !strings.HasSuffix(strategy, "Similarity") {
        sortOrder = "ASC"
    }
    q := fmt.Sprintf(`
        DECLARE $embedding AS String;
        SELECT id, document, Knn::%s(embedding, $embedding) AS score
        FROM %s %s
        ORDER BY score %s
        LIMIT %d;
        `, strategy, tableName, viewIndex, sortOrder, limit)

    row, err := db.Query().Query(ctx, q,
    query.WithParameters(
        ydb.ParamsBuilder().Param("$embedding").Bytes(convertVectorToBytes(embedding)).Build(),
    ),
    )
    if err != nil {
        return nil, err
    }
    defer row.Close(ctx)

    var items []ResultItem
    for rs, err := row.NextResultSet(ctx); err == nil; rs, err = row.NextResultSet(ctx) {
        for r, err := rs.NextRow(ctx); err == nil; r, err = rs.NextRow(ctx) {
            var item ResultItem
            if err := r.ScanNamed(
                query.Named("id", &item.ID),
                query.Named("document", &item.Document),
                query.Named("score", &item.Score),
            ); err != nil {
                return nil, err
            }
            items = append(items, item)
        }
    }
    return items, nil
}
```

Java

Recommended approach

```
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import tech.ydb.common.transaction.TxMode;
import tech.ydb.query.tools.QueryReader;
import tech.ydb.query.tools.SessionRetryContext;
import tech.ydb.table.query.Params;
import tech.ydb.table.result.ResultSetReader;
import tech.ydb.table.values.PrimitiveValue;

byte[] convertVectorToBytes(float[] vector) {
    ByteBuffer bb = ByteBuffer.allocate(vector.length * Float.BYTES + 1).order(ByteOrder.LITTLE_ENDIAN);
    for (float v : vector) {
        bb.putFloat(v);
    }
    bb.put((byte) 0x01);
    return bb.array();
}

List<ResultItem> searchItemsAsBytes(
        SessionRetryContext retryCtx,
        String tableName,
        float[] embedding,
        String strategy,
        long limit,
        Optional<String> indexName) {

    String viewIndex = indexName.map(n -> "VIEW " + n).orElse("");
    String sortOrder = strategy.endsWith("Similarity") ? "DESC" : "ASC";

    String query = String.format("""
            DECLARE $embedding as String;
            SELECT
                id,
```

```
                document,
                Knn::%s(embedding, $embedding) as score
            FROM %s %s
            ORDER BY score %s
            LIMIT %d;
            """, strategy, tableName, viewIndex, sortOrder, limit);

    Params params = Params.of("$embedding", PrimitiveValue.newBytes(convertVectorToBytes(embedding)));

    QueryReader reader = retryCtx.supplyResult(session -> QueryReader.readFrom(
            session.createQuery(query, TxMode.SERIALIZABLE_RW, params)
    )).join().getValue();

    List<ResultItem> result = new ArrayList<>();
    ResultSetReader rs = reader.getResultSet(0);
    while (rs.next()) {
        result.add(new ResultItem(
                rs.getColumn("id").getText(),
                rs.getColumn("document").getText(),
                rs.getColumn("score").getFloat()
        ));
    }
    return result;
}

// record ResultItem(String id, String document, float score) {}
```

Alternative approach

> **Warning**
>
> Passing vectors as `List<Float>` and converting on the YQL side with `Knn::ToBinaryStringFloat` has lower performance than client-side byte array encoding.

The same search with the query vector passed as `List<Float>`:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import tech.ydb.common.transaction.TxMode;
import tech.ydb.query.tools.QueryReader;
import tech.ydb.query.tools.SessionRetryContext;
import tech.ydb.table.query.Params;
import tech.ydb.table.result.ResultSetReader;
import tech.ydb.table.values.ListType;
import tech.ydb.table.values.ListValue;
import tech.ydb.table.values.PrimitiveType;
import tech.ydb.table.values.PrimitiveValue;
import tech.ydb.table.values.Value;

List<ResultItem> searchItemsAsFloatList(
        SessionRetryContext retryCtx,
        String tableName,
        float[] embedding,
        String strategy,
        long limit,
        Optional<String> indexName) {

    String viewIndex = indexName.map(n -> "VIEW " + n).orElse("");
    String sortOrder = strategy.endsWith("Similarity") ? "DESC" : "ASC";

    String query = String.format("""
            DECLARE $embedding as List<Float>;

            $target_embedding = Knn::ToBinaryStringFloat($embedding);

            SELECT
                id,
                document,
                Knn::%s(embedding, $target_embedding) as score
            FROM %s %s
            ORDER BY score
            %s
            LIMIT %d;
            """, strategy, tableName, viewIndex, sortOrder, limit);

    Value<?>[] floats = new Value<?>[embedding.length];
    for (int i = 0; i < embedding.length; i++) {
        floats[i] = PrimitiveValue.newFloat(embedding[i]);
    }
    ListValue emb = ListType.of(PrimitiveType.Float).newValueOwn(floats);
    Params params = Params.of("$embedding", emb);

    QueryReader reader = retryCtx.supplyResult(session -> QueryReader.readFrom(
            session.createQuery(query, TxMode.SERIALIZABLE_RW, params)
    )).join().getValue();

    List<ResultItem> result = new ArrayList<>();
```

```
    ResultSetReader rs = reader.getResultSet(0);
    while (rs.next()) {
        result.add(new ResultItem(
                rs.getColumn("id").getText(),
                rs.getColumn("document").getText(),
                rs.getColumn("score").getFloat()
        ));
    }
    return result;
}

// record ResultItem(String id, String document, float score) {}
```

Python

Recommended approach

Native SDK

```
import ydb

def search_items_vector_as_bytes(
    pool: ydb.QuerySessionPool,
    table_name: str,
    embedding: list[float],
    strategy: str = "CosineSimilarity",
    limit: int = 1,
    index_name: str | None = None,
    top_clusters: int = 10,
) -> list[dict]:
    view_index = f"VIEW {index_name}" if index_name else ""

    sort_order = "DESC" if strategy.endswith("Similarity") else "ASC"

    query = f"""
    PRAGMA ydb.KMeansTreeSearchTopSize = "{top_clusters}";
    DECLARE $embedding as String;

    SELECT
        id,
        document,
        Knn::{strategy}(embedding, $embedding) as score
    FROM {table_name} {view_index}
    ORDER BY score {sort_order}
    LIMIT {limit};
    """

    result = pool.execute_with_retries(
        query,
        {
            "$embedding": (
                ydb.convert_floats_to_embedding_bytes(embedding),
                ydb.PrimitiveType.String,
            ),
        },
    )

    items = []

    for result_set in result:
        for row in result_set.rows:
            items.append(
                {
                    "id": row["id"],
                    "document": row["document"],
                    "score": row["score"],
                }
            )

    return items
```

Native SDK (Asyncio)

```
import ydb

async def search_items_vector_as_bytes(
    pool: ydb.aio.QuerySessionPool,
    table_name: str,
    embedding: list[float],
    strategy: str = "CosineSimilarity",
    limit: int = 1,
    index_name: str | None = None,
) -> list[dict]:
    view_index = f"VIEW {index_name}" if index_name else ""

    sort_order = "DESC" if strategy.endswith("Similarity") else "ASC"

    query = f"""
    DECLARE $embedding as String;
```

```
    SELECT
        id,
        document,
        Knn::{strategy}(embedding, $embedding) as score
    FROM {table_name} {view_index}
    ORDER BY score {sort_order}
    LIMIT {limit};
    """

    result = await pool.execute_with_retries(
        query,
        {
            "$embedding": (
                ydb.convert_floats_to_embedding_bytes(embedding),
                ydb.PrimitiveType.String,
            ),
        },
    )

    items = []

    for result_set in result:
        for row in result_set.rows:
            items.append(
                {
                    "id": row["id"],
                    "document": row["document"],
                    "score": row["score"],
                }
            )

    return items
```

Alternative approach

> **Warning**
>
> Passing vectors as `List<Float>` and converting on the YQL side with `Knn::ToBinaryStringFloat` has lower performance than client-side byte array encoding.

Native SDK

```
import ydb

def search_items_vector_as_float_list(
    pool: ydb.QuerySessionPool,
    table_name: str,
    embedding: list[float],
    strategy: str = "CosineSimilarity",
    limit: int = 1,
    index_name: str | None = None,
    top_clusters: int = 10,
) -> list[dict]:
    view_index = f"VIEW {index_name}" if index_name else ""

    sort_order = "DESC" if strategy.endswith("Similarity") else "ASC"

    query = f"""
    PRAGMA ydb.KMeansTreeSearchTopSize = "{top_clusters}";
    DECLARE $embedding as List<Float>;

    $target_embedding = Knn::ToBinaryStringFloat($embedding);

    SELECT
        id,
        document,
        Knn::{strategy}(embedding, $target_embedding) as score
    FROM {table_name} {view_index}
    ORDER BY score
    {sort_order}
    LIMIT {limit};
    """

    result = pool.execute_with_retries(
        query,
        {
            "$embedding": (embedding, ydb.ListType(ydb.PrimitiveType.Float)),
        },
    )

    items = []

    for result_set in result:
        for row in result_set.rows:
            items.append(
                {
                    "id": row["id"],
                    "document": row["document"],
                    "score": row["score"],
```

```
                }
            )

    return items
```

Native SDK (Asyncio)

```
import ydb

async def search_items_vector_as_float_list(
    pool: ydb.aio.QuerySessionPool,
    table_name: str,
    embedding: list[float],
    strategy: str = "CosineSimilarity",
    limit: int = 1,
    index_name: str | None = None,
) -> list[dict]:
    view_index = f"VIEW {index_name}" if index_name else ""

    sort_order = "DESC" if strategy.endswith("Similarity") else "ASC"

    query = f"""
    DECLARE $embedding as List<Float>;

    $target_embedding = Knn::ToBinaryStringFloat($embedding);

    SELECT
        id,
        document,
        Knn::{strategy}(embedding, $target_embedding) as score
    FROM {table_name} {view_index}
    ORDER BY score
    {sort_order}
    LIMIT {limit};
    """

    result = await pool.execute_with_retries(
        query,
        {
            "$embedding": (embedding, ydb.ListType(ydb.PrimitiveType.Float)),
        },
    )

    items = []

    for result_set in result:
        for row in result_set.rows:
            items.append(
                {
                    "id": row["id"],
                    "document": row["document"],
                    "score": row["score"],
                }
            )

    return items
```

C#

Vector search examples for the .NET SDK are not available yet.

This functionality is not currently supported.

Track progress or vote for support: ydb-dotnet-sdk#657

JavaScript

Recommended approach

```
const limit;
const embedding = convertVectorToBytes(new Float32Array([1.5, 2.5, 3.5]))

await sql`SELECT
    id,
    document,
    Knn::CosineSimilarity(embedding, ${embedding}) as score
FROM `table_name`
ORDER BY score DESC
LIMIT ${unsafe(limit)};
```

Alternative approach

> **Warning**
>
> Passing vectors as `List<Float>` and converting on the YQL side with `Knn::ToBinaryStringFloat` has lower performance than client-side byte array encoding.

```
const limit;
const embedding = new Float32Array([1.5, 2.5, 3.5])

await sql`SELECT
        id,
        document,
        Knn::CosineSimilarity(embedding, Knn::ToBinaryStringFloat(${embedding})) as score
    FROM `table_name`
    ORDER BY score DESC
    LIMIT ${unsafe(limit)};
```

Rust

```
let view_index = index_name.map(|n| format!("VIEW {n}")).unwrap_or_default();
let sort_order = if strategy.ends_with("Similarity") { "DESC" } else { "ASC" };

let mut stream = qc
    .query(search_query)
    .param("$embedding", convert_vector_to_bytes(&embedding))
    .await?;
while let Some(result_set) = stream.next_result_set().await? {
    for mut row in result_set {
        // id, document, score
    }
}
stream.close().await?;
```

PHP

Vector search examples for the PHP SDK are not available yet.

This functionality is not currently supported.

Track progress or vote for support: ydb-php-sdk#268

Full example

The following example combines all the steps above:

1. Dropping the existing table
2. Creating a new table
3. Inserting items
4. Searching for nearest vectors without an index
5. Adding a vector index
6. Searching for nearest vectors with the index

C++

```
void PrintResults(const std::vector<TResultItem>& items)
{
    if (items.empty()) {
        std::cout << "No items found" << std::endl;
        return;
    }

    for (const auto& item : items) {
        std::cout << "[score=" << item.Score << "] " << item.Id << ": " << item.Document << std::endl;
    }
}

void VectorExample(
    const std::string& endpoint,
    const std::string& database,
    const std::string& tableName,
    const std::string& indexName)
{
    auto driverConfig = NYdb::CreateFromEnvironment(endpoint + "/?database=" + database);
    NYdb::TDriver driver(driverConfig);
    NYdb::NQuery::TQueryClient client(driver);

    try {
        DropVectorTable(client, tableName);
        CreateVectorTable(client, tableName);
        std::vector<TItem> items = {
            {.Id = "1", .Document = "document 1", .Embedding = {0.98, 0.1, 0.01}},
            {.Id = "2", .Document = "document 2", .Embedding = {1.0, 0.05, 0.05}},
            {.Id = "3", .Document = "document 3", .Embedding = {0.9, 0.1, 0.1}},
            {.Id = "4", .Document = "document 4", .Embedding = {0.03, 0.0, 0.99}},
            {.Id = "5", .Document = "document 5", .Embedding = {0.0, 0.0, 0.99}},
            {.Id = "6", .Document = "document 6", .Embedding = {0.0, 0.02, 1.0}},
            {.Id = "7", .Document = "document 7", .Embedding = {0.0, 1.05, 0.05}},
            {.Id = "8", .Document = "document 8", .Embedding = {0.02, 0.98, 0.1}},
            {.Id = "9", .Document = "document 9", .Embedding = {0.0, 1.0, 0.05}},
        };
        InsertItemsAsBytes(client, tableName, items);
        PrintResults(SearchItemsAsBytes(client, tableName, {1.0, 0.0, 0.0}, "CosineSimilarity", 3));
        AddIndex(driver, client, database, tableName, indexName, "similarity=cosine", 3, 1, 3);
```

```
        PrintResults(SearchItemsAsBytes(client, tableName, {1.0, 0.0, 0.0}, "CosineSimilarity", 3, 10, indexN
ame));
    } catch (const std::exception& e) {
        std::cerr << "Execution failed: " << e.what() << std::endl;
    }

    driver.Stop(true);
}
```

Full source code is available at this link.

Go

Vector search is fully supported in the Go SDK. A complete example combining all operations above (table creation, data insert, index creation, search) is built from the code fragments on this page. See a working example in the ydb-go-sdk.

Java

The example combines steps from the sections above: `QueryClient` + `SessionRetryContext` for YQL and `TableClient` + `SessionRetryContext` for `ALTER TABLE` with index rename. Methods `createVectorTable`, `insertItemsAsBytes`, `searchItemsAsBytes`, `addVectorIndex`, and types `Item` / `ResultItem` are as in the corresponding fragments on this page.

```
import java.util.List;
import java.util.Optional;

import tech.ydb.common.transaction.TxMode;
import tech.ydb.core.grpc.GrpcTransport;
import tech.ydb.query.QueryClient;
import tech.ydb.query.tools.QueryReader;
import tech.ydb.query.tools.SessionRetryContext;
import tech.ydb.table.TableClient;
import tech.ydb.table.query.Params;

public class VectorSearchJavaExample {

    record Item(String id, String document, float[] embedding) {}
    record ResultItem(String id, String document, float score) {}

    public static void main(String[] args) {
        String connectionString = System.getenv().getOrDefault("YDB_CONNECTION_STRING", "grpc://localhost:213
6/local");
        String tableName = "ydb_vector_search";
        String indexName = "ydb_vector_index";

        try (GrpcTransport transport = GrpcTransport.forConnectionString(connectionString).build();
            QueryClient queryClient = QueryClient.newClient(transport).build();
            TableClient tableClient = TableClient.newClient(transport).build()) {

            SessionRetryContext queryRetry = SessionRetryContext.create(queryClient).build();
            SessionRetryContext tableRetry = SessionRetryContext.create(tableClient).build();

            dropVectorTableIfExists(queryRetry, tableName);
            createVectorTable(queryRetry, tableName);

            List<Item> items = List.of(
                    new Item("1", "vector 1", new float[]{0.98f, 0.1f, 0.01f}),
                    new Item("2", "vector 2", new float[]{1.0f, 0.05f, 0.05f}),
                    new Item("3", "vector 3", new float[]{0.9f, 0.1f, 0.1f}),
                    new Item("4", "vector 4", new float[]{0.03f, 0.0f, 0.99f}),
                    new Item("5", "vector 5", new float[]{0.0f, 0.0f, 0.99f}),
                    new Item("6", "vector 6", new float[]{0.0f, 0.02f, 1.0f}),
                    new Item("7", "vector 7", new float[]{0.0f, 1.05f, 0.05f}),
                    new Item("8", "vector 8", new float[]{0.02f, 0.98f, 0.1f}),
                    new Item("9", "vector 9", new float[]{0.0f, 1.0f, 0.05f})
            );

            insertItemsAsBytes(queryRetry, tableName, items);
            printResults(searchItemsAsBytes(queryRetry, tableName, new float[]{1, 0, 0},
                    "CosineSimilarity", 3, Optional.empty()));

            addVectorIndex(transport, queryRetry, tableRetry, tableName, indexName,
                    "similarity=cosine", 3, 1, 3);

            printResults(searchItemsAsBytes(queryRetry, tableName, new float[]{1, 0, 0},
                    "CosineSimilarity", 3, Optional.of(indexName)));
        }
    }

    static void dropVectorTableIfExists(SessionRetryContext queryRetry, String tableName) {
        String ddl = String.format("DROP TABLE IF EXISTS `%s`", tableName);
        queryRetry.supplyResult(s -> QueryReader.readFrom(
                s.createQuery(ddl, TxMode.NONE, Params.empty())
        )).join().getValue();
        System.out.println("Vector table dropped");
    }

    static void printResults(List<ResultItem> items) {
        if (items.isEmpty()) {
            System.out.println("No items found");
            return;
        }
        for (ResultItem item : items) {
            System.out.printf("[score=%f] %s: %s%n", item.score(), item.id(), item.document());
```

```
        }
    }
}
```

The output matches the Python example above.

Python

Usage example

Native SDK

```python
import os
import ydb

def print_results(items):
    if len(items) == 0:
        print("No items found")
        return

    for item in items:
        print(f"[score={item['score']}] {item['id']}: {item['document']}")

def drop_vector_table_if_exists(pool: ydb.QuerySessionPool, table_name: str) -> None:
    pool.execute_with_retries(f"DROP TABLE IF EXISTS `{table_name}`")

    print("Vector table dropped")

def main(
    ydb_endpoint: str,
    ydb_database: str,
    ydb_credentials: ydb.AbstractCredentials,
    table_name: str,
    index_name: str,
):
    driver = ydb.Driver(
        endpoint=ydb_endpoint,
        database=ydb_database,
        credentials=ydb_credentials,
    )
    driver.wait(5, fail_fast=True)
    pool = ydb.QuerySessionPool(driver)

    drop_vector_table_if_exists(pool, table_name)

    create_vector_table(pool, table_name)

    items = [
        {"id": "1", "document": "vector 1", "embedding": [0.98, 0.1, 0.01]},
        {"id": "2", "document": "vector 2", "embedding": [1.0, 0.05, 0.05]},
        {"id": "3", "document": "vector 3", "embedding": [0.9, 0.1, 0.1]},
        {"id": "4", "document": "vector 4", "embedding": [0.03, 0.0, 0.99]},
        {"id": "5", "document": "vector 5", "embedding": [0.0, 0.0, 0.99]},
        {"id": "6", "document": "vector 6", "embedding": [0.0, 0.02, 1.0]},
        {"id": "7", "document": "vector 7", "embedding": [0.0, 1.05, 0.05]},
        {"id": "8", "document": "vector 8", "embedding": [0.02, 0.98, 0.1]},
        {"id": "9", "document": "vector 9", "embedding": [0.0, 1.0, 0.05]},
    ]

    insert_items_vector_as_bytes(pool, table_name, items)

    items = search_items_vector_as_bytes(
        pool,
        table_name,
        embedding=[1, 0, 0],
        strategy="CosineSimilarity",
        limit=3,
        top_clusters=10,
    )
    print_results(items)

    add_vector_index(
        pool,
        driver,
        table_name,
        index_name=index_name,
        strategy="similarity=cosine",
        dimension=3,
        levels=1,
        clusters=3,
    )

    items = search_items_vector_as_bytes(
        pool,
        table_name,
        embedding=[1, 0, 0],
        index_name=index_name,
        strategy="CosineSimilarity",
        limit=3,
        top_clusters=10,
    )
    print_results(items)
```

```
    pool.stop()
    driver.stop()

if __name__ == "__main__":
    main(
        ydb_endpoint=os.environ.get("YDB_ENDPOINT", "grpc://localhost:2136"),
        ydb_database=os.environ.get("YDB_DATABASE", "/local"),
        ydb_credentials=ydb.credentials_from_env_variables(),
        table_name="ydb_vector_search",
        index_name="ydb_vector_index",
    )
```

Native SDK (Asyncio)

```
import os
import ydb
import asyncio

def print_results(items):
    if len(items) == 0:
        print("No items found")
        return

    for item in items:
        print(f"[score={item['score']}] {item['id']}: {item['document']}")

async def drop_vector_table_if_exists(pool: ydb.aio.QuerySessionPool, table_name: str) -> None:
    await pool.execute_with_retries(f"DROP TABLE IF EXISTS `{table_name}`")

    print("Vector table dropped")

async def main(
    ydb_endpoint: str,
    ydb_database: str,
    ydb_credentials: ydb.AbstractCredentials,
    table_name: str,
    index_name: str,
):
    async with ydb.aio.Driver(
        endpoint=ydb_endpoint,
        database=ydb_database,
        credentials=ydb_credentials,
    ) as driver:
        await driver.wait(5, fail_fast=True)
        pool = ydb.aio.QuerySessionPool(driver)

        await drop_vector_table_if_exists(pool, table_name)

        await create_vector_table(pool, table_name)

        items = [
            {"id": "1", "document": "vector 1", "embedding": [0.98, 0.1, 0.01]},
            {"id": "2", "document": "vector 2", "embedding": [1.0, 0.05, 0.05]},
            {"id": "3", "document": "vector 3", "embedding": [0.9, 0.1, 0.1]},
            {"id": "4", "document": "vector 4", "embedding": [0.03, 0.0, 0.99]},
            {"id": "5", "document": "vector 5", "embedding": [0.0, 0.0, 0.99]},
            {"id": "6", "document": "vector 6", "embedding": [0.0, 0.02, 1.0]},
            {"id": "7", "document": "vector 7", "embedding": [0.0, 1.05, 0.05]},
            {"id": "8", "document": "vector 8", "embedding": [0.02, 0.98, 0.1]},
            {"id": "9", "document": "vector 9", "embedding": [0.0, 1.0, 0.05]},
        ]

        await insert_items_vector_as_bytes(pool, table_name, items)

        items = await search_items_vector_as_bytes(
            pool,
            table_name,
            embedding=[1, 0, 0],
            strategy="CosineSimilarity",
            limit=3,
        )
        print_results(items)

        await add_vector_index(
            pool,
            driver,
            table_name,
            index_name=index_name,
            strategy="similarity=cosine",
            dimension=3,
            levels=1,
            clusters=3,
        )

        items = await search_items_vector_as_bytes(
            pool,
            table_name,
            embedding=[1, 0, 0],
            index_name=index_name,
            strategy="CosineSimilarity",
```

```
            limit=3,
        )
        print_results(items)

        await pool.stop()

if __name__ == "__main__":
    asyncio.run(main(
        ydb_endpoint=os.environ.get("YDB_ENDPOINT", "grpc://localhost:2136"),
        ydb_database=os.environ.get("YDB_DATABASE", "/local"),
        ydb_credentials=ydb.credentials_from_env_variables(),
        table_name="ydb_vector_search",
        index_name="ydb_vector_index",
    ))
```

Program output:

```
Vector table dropped
Vector table created
9 items inserted
[score=0.997509241104126] 2: vector 2
[score=0.9947828650474548] 1: vector 1
[score=0.9878783822059631] 3: vector 3
Table index ydb_vector_index created.
[score=0.997509241104126] 2: vector 2
[score=0.9947828650474548] 1: vector 1
[score=0.9878783822059631] 3: vector 3
```

The table was created, 9 documents were added, and vector similarity search ran successfully both before and after adding the vector index.

Full source code is available at this link.

C#

Vector search examples for the .NET SDK are not available yet.

This functionality is not currently supported.

Track progress or vote for support: ydb-dotnet-sdk#657

JavaScript

Vector index creation and a full end-to-end program for the JavaScript SDK are not available yet.

This functionality is not currently supported.

Track progress or vote for support: ydb-js-sdk#627

Rust

Full example on GitHub: vector-search.rs.

```
// drop → create table → insert → search → add index → search with VIEW
drop_vector_table_if_exists(&mut qc, table_name).await?;
create_vector_table(&mut qc, table_name).await?;
insert_items_as_bytes(&mut qc, table_name, &items).await?;
let hits = search_items_as_bytes(&mut qc, table_name, &[1.0, 0.0, 0.0],
    "CosineSimilarity", 3, None).await?;
add_vector_index(&mut qc, table_name, index_name, "similarity=cosine", 3, 1, 3).await?;
let hits = search_items_as_bytes(&mut qc, table_name, &[1.0, 0.0, 0.0],
    "CosineSimilarity", 3, Some(index_name)).await?;
```

PHP

Vector search examples for the PHP SDK are not available yet.

This functionality is not currently supported.

Track progress or vote for support: ydb-php-sdk#268

Distributed lock

Consider a scenario where it is necessary to ensure that only one instance of a client application accesses a shared resource at any given time. To achieve this, the semaphore mechanism in YDB coordination nodes can be utilized.

Semaphore lease mechanism

In contrast to local multithreaded programming, clients in distributed systems do not directly acquire locks or semaphores. Instead, they lease them for a specified duration, which can be periodically renewed. Due to reliance on physical time which can vary between machines, clients and the server might encounter situations where multiple clients believe they have acquired the same semaphore simultaneously, even if the server's perspective differs. To reduce the likelihood of such occurrences, it is crucial to configure automatic time synchronization beforehand, both on servers hosting client applications and on the YDB side, ideally using a unified time source.

Therefore, while distributed locking through such mechanisms cannot guarantee the complete absence of simultaneous resource access, it can significantly lower the probability of such events. This approach serves as an optimization to prevent unnecessary competition among clients for a shared resource. Absolute guarantees against concurrent resource requests could be implemented on the resource side.

Code example

Go

```
for {
  if session, err := db.Coordination().CreateSession(ctx, path); err != nil {
    return fmt.Errorf("cannot create session: %v", err);
  }

  if lease, err := session.AcquireSemaphore(ctx,
    semaphore,
    coordination.Exclusive,
    options.WithEphemeral(true),
  ); err != nil {
    // the session is likely lost, try to create a new one and get the lock in it
    session.Close(ctx);
    continue;
  }

  // lock acquired, start processing
  select {
     case <-lease.Context().Done():
  }

  // lock released, end processing
}
```

Python

Native SDK

```
import ydb

def coordination_service_workflow(driver: ydb.Driver, node_path: str, semaphore_name: str):
    client = driver.coordination_client

    client.create_node(node_path)

    with client.session(node_path) as session:
        with session.semaphore(semaphore_name) as semaphore:
            print("Some exclusive work")
```

Native SDK (Asyncio)

```
import os
import ydb

async def coordination_service_workflow(driver: ydb.aio.Driver, node_path: str, semaphore_name: str):
    client = driver.coordination_client
    await client.create_node(node_path)
    async with client.session(node_path) as session:
        async with session.semaphore(semaphore_name) as semaphore:
            print("Some exclusive work")
```

JavaScript

```
import { Driver } from '@ydbjs/core'
import { CoordinationClient } from '@ydbjs/coordination'

const driver = new Driver('grpc://localhost:2136/local')
const client = new CoordinationClient(driver)

await using session = await client.createSession('/local/my-app')
await using lock = await session.mutex('job-lock').lock()
await doWork(lock.signal)

// For long lived applications
```

```
for await (let session of client.openSession('/local/my-app')) {
  let mutex = session.mutex('job-lock')

  try {
    // Blocks until the lock is acquired.
    await using lock = await mutex.lock()

    await doWork(lock.signal)
  } catch {
    if (session.signal.aborted) continue // session expired, retry
    throw error
  }

  break
}
```

Java

```
CoordinationClient client = CoordinationClient.newClient(transport);
client.createNode(nodePath).join().expectSuccess();

try (CoordinationSession session = client.createSession(nodePath)) {
    session.connect().join().expectSuccess();
    SemaphoreLease lease = session.acquireEphemeralSemaphore(semaphoreName, true, Duration.ofMinutes(5))
            .join().getValue();
    try {
        // exclusive access to the resource
    } finally {
        lease.release().join();
    }
}
```

Leader election

Consider a scenario where multiple application instances need to elect a leader among themselves and be aware of the current leader at any given time.

This scenario can be implemented using semaphores in YDB coordination nodes as follows:

1. A semaphore is created (for example, named `my-service-leader`) with `Limit=1`.
2. All application instances call `AcquireSemaphore` with `Count=1`, specifying their endpoint in the `Data` field.
3. Only one application instance's call will complete quickly, while others will be queued. The application instance whose call completes successfully becomes the current leader.
4. All application instances call `DescribeSemaphore` with `WatchOwners=true` and `IncludeOwners=true`. The result's `Owners` field will contain at most one element, from which the current leader's endpoint can be determined via its `Data` field.
5. When the leader changes, `OnChanged` is called. In this case, application instances make a similar `DescribeSemaphore` call to learn about the new leader.

Service discovery

Consider a scenario where application instances are dynamically started and publish their endpoints, while other clients need to receive this list and respond to its changes.

This scenario can be implemented using semaphores in YDB coordination nodes as follows:

1. Create a semaphore (for example, named `my-service-endpoints`) with `Limit=Max<ui64>()`.
2. All application instances call `AcquireSemaphore` with `Count=1`, specifying their endpoint in the `Data` field.
3. Since the semaphore limit is very high, all `AcquireSemaphore` calls should complete quickly.
4. At this point, publication is complete, and application instances only need to respond to session stops by republishing themselves through a new session.
5. Clients call `DescribeSemaphore` with `IncludeOwners=true` and optionally with `WatchOwners=true`. In the result, the `Owners` field's `Data` will contain the endpoints of registered application instances.
6. When the list of endpoints changes, `OnChanged` is called. In this case, clients make a similar `DescribeSemaphore` call and receive the updated list.

Configuration publication

Let's consider a scenario where we need to publish a small configuration for multiple application instances that should promptly react to its changes.

This scenario can be implemented using semaphores in YDB coordination nodes as follows:

1. A semaphore is created (for example, named `my-service-config`).
2. The updated configuration is published through `UpdateSemaphore`.
3. Application instances call `DescribeSemaphore` with `WatchData=true`. In the result, the `Data` field will contain the current version of the configuration.
4. When the configuration changes, `OnChanged` is called. In this case, application instances make a similar `DescribeSemaphore` call and receive the updated configuration.

Diagnosing problems

When diagnosing problems related to YDB, diagnostic tools help: logging, metrics, and distributed tracing. It is recommended to enable them in advance, before problems occur, so that when investigating an incident, you can see the full picture of the system state before, during, and after the failure.

This section contains code recipes for enabling diagnostic tools in different YDB SDKs.

Contents:

- Enable logging
- Connect metrics to Prometheus
- Tracing with OpenTelemetry
- Connect tracing to Jaeger

Enabling logging

Below are examples of code that enables logging in different YDB SDKs.

Go

Native SDK

There are several ways to enable logs in an application that uses `ydb-go-sdk`:

Using the `YDB_LOG_SEVERITY_LEVEL` environment variable

This environment variable enables the built-in `ydb-go-sdk` logger (synchronous, non-block) and prints to the standard output stream.
You can set the environment variable as follows:

```
export YDB_LOG_SEVERITY_LEVEL=info
```

(possible values: `trace`, `debug`, `info`, `warn`, `error`, `fatal`, and `quiet`, defaults to `quiet`).

Enable a third-party logger `go.uber.org/zap`

```
package main

import (
  "context"
  "os"

  "go.uber.org/zap"

  ydbZap "github.com/ydb-platform/ydb-go-sdk-zap"
  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  var log *zap.Logger // zap-logger with init out of this scope
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydbZap.WithTraces(
      log,
      trace.DetailsAll,
    ),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  ...
}
```

Enable a third-party logger `github.com/rs/zerolog`

```
package main

import (
  "context"
  "os"

  "github.com/rs/zerolog"

  ydbZerolog "github.com/ydb-platform/ydb-go-sdk-zerolog"
  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  var log zerolog.Logger // zap-logger with init out of this scope
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydbZerolog.WithTraces(
      &log,
      trace.DetailsAll,
    ),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  ...
}
```

Enable a custom logger implementation `github.com/ydb-platform/ydb-go-sdk/v3/log.Logger`

```
package main

import (
  "context"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/log"
  "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  var logger log.Logger // logger implementation with init out of this scope
  db, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithLogger(
      logger,
      trace.DetailsAll,
    ),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  ...
}
```

Implement your own logging package

You can implement your own logging package based on the driver events in the `github.com/ydb-platform/ydb-go-sdk/v3/trace` tracing package. The `github.com/ydb-platform/ydb-go-sdk/v3/trace` tracing package describes all logged driver events.

Iterate over server errors with `IterateByIssues`

When using YDB through the Go SDK, you can enable logging of requests and responses and also obtain detailed information about server errors (issues) — extra messages YDB returns in the response when operations fail. To iterate over issues in the server response, use IterateByIssues.

database/sql

There are several ways to enable logs in an application that uses `ydb-go-sdk`:

Using the `YDB_LOG_SEVERITY_LEVEL` environment variable

This environment variable enables the built-in `ydb-go-sdk` logger (synchronous, non-block) and prints to the standard output stream.
You can set the environment variable as follows:

```
export YDB_LOG_SEVERITY_LEVEL=info
```

(possible values: `trace`, `debug`, `info`, `warn`, `error`, `fatal`, and `quiet`, defaults to `quiet`).

Enable a third-party logger `go.uber.org/zap`

```
package main

import (
  "context"
  "database/sql"
  "os"

  "go.uber.org/zap"

  ydbZap "github.com/ydb-platform/ydb-go-sdk-zap"
  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  var log *zap.Logger // zap-logger with init out of this scope
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydbZap.WithTraces(
      log,
      trace.DetailsAll,
    ),
  )
  if err != nil {
```

```
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  defer connector.Close()

  db := sql.OpenDB(connector)
  defer db.Close()
  ...
}
```

Enable a third-party logger `github.com/rs/zerolog`

```
package main

import (
  "context"
  "database/sql"
  "os"

  "github.com/rs/zerolog"

  ydbZerolog "github.com/ydb-platform/ydb-go-sdk-zerolog"
  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  var log zerolog.Logger // zap-logger with init out of this scope
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydbZerolog.WithTraces(
      &log,
      trace.DetailsAll,
    ),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)

  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  defer connector.Close()

  db := sql.OpenDB(connector)
  defer db.Close()
  ...
}
```

Enable a custom logger implementation `github.com/ydb-platform/ydb-go-sdk/v3/log.Logger`

```
package main

import (
  "context"
  "database/sql"
  "os"

  "github.com/ydb-platform/ydb-go-sdk/v3"
  "github.com/ydb-platform/ydb-go-sdk/v3/log"
  "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
  ctx, cancel := context.WithCancel(context.Background())
  defer cancel()
  var logger log.Logger // logger implementation with init out of this scope
  nativeDriver, err := ydb.Open(ctx,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithLogger(
      logger,
      trace.DetailsAll,
    ),
  )
  if err != nil {
    panic(err)
  }
  defer nativeDriver.Close(ctx)
```

```
  connector, err := ydb.Connector(nativeDriver)
  if err != nil {
    panic(err)
  }
  defer connector.Close()

  db := sql.OpenDB(connector)
  defer db.Close()
  ...
}
```

Implement your own logging package

> You can implement your own logging package based on the driver events in the `github.com/ydb-platform/ydb-go-sdk/v3/trace` tracing package. The `github.com/ydb-platform/ydb-go-sdk/v3/trace` tracing package describes all logged driver events.

Java

Native SDK

For logging purposes, the YDB Java SDK uses the slf4j library, which supports multiple logging levels (`error`, `warn`, `info`, `debug`, `trace`) for one or many loggers. The current implementation supports the following loggers:

- The `tech.ydb.core.grpc` logger provides information about the internal gRPC implementation
- The `debug` level logs all gRPC operations; use it only for debugging
- The `info` level is recommended by default
- On the `debug` level, the `tech.ydb.table.impl` logger lets you track the internal state of the YDB driver, including the session pool
- On the `debug` level, the `tech.ydb.table.SessionRetryContext` logger reports the number of retries, query results, per-retry duration, and total operation time
- On the `debug` level, the `tech.ydb.table.Session` logger provides the query text, response status, and execution time for session operations

Enabling and configuring the Java SDK loggers depends on the `slf4j-api` implementation you use.
Here is an example `log4j2` configuration for the `log4j-slf4j-impl` library:

```
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>

  <Loggers>
    <Logger name="io.netty" level="warn" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
    <Logger name="io.grpc.netty" level="warn" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
    <Logger name="tech.ydb.core.grpc" level="info" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
    <Logger name="tech.ydb.table.impl" level="info" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
    <Logger name="tech.ydb.table.SessionRetryContext" level="debug" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
    <Logger name="tech.ydb.table.Session" level="debug" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>

    <Root level="debug" >
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

JDBC

The JDBC driver uses the same logging stack via `slf4j`; configure the `tech.ydb.*` loggers the same way as in the native SDK.

The same debug logs are available in other stacks built on JDBC (Spring Boot, ORMs, connection pools, and so on): they reach YDB through this driver, so it is enough to attach the same `slf4j` / log4j2 / logback configuration in the application.

PHP

For logging purposes, you need to use a class, that implements `\Psr\Log\LoggerInterface`.
`ydb-php-sdk` has build-in loggers in `YdbPlatform\Ydb\Logger` namespace:

- `NullLogger` - default logger, which writes nothing
- `SimpleStdLogger($level)` - logger, which writes to logs in stderr.

Usage example:

```php
$config = [
  'logger' => new \YdbPlatform\Ydb\Logger\SimpleStdLogger(\YdbPlatform\Ydb\Logger\SimpleStdLogger::INFO)
]
$ydb = new \YdbPlatform\Ydb\Ydb($config);
```

Python

The Python SDK uses the standard `logging` library. To enable a specific logging level:

```python
import logging

logging.getLogger('ydb').setLevel(logging.DEBUG)
```

JavaScript

The SDK uses the debug library for logging.
To enable logs, set the `DEBUG` environment variable to filter SDK events, for example `DEBUG=ydbjs:*`.

Enabling metrics in Prometheus

Below are examples of the code for enabling metrics in Prometheus in different YDB SDKs.

Go

Native SDK

```
package main

import (
    "context"

    "github.com/prometheus/client_golang/prometheus"
    metrics "github.com/ydb-platform/ydb-go-sdk-prometheus/v2"
    "github.com/ydb-platform/ydb-go-sdk/v3"
    "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
    ctx := context.Background()
    registry := prometheus.NewRegistry()
    db, err := ydb.Open(ctx,
        os.Getenv("YDB_CONNECTION_STRING"),
        metrics.WithTraces(
            registry,
            metrics.WithDetails(trace.DetailsAll),
            metrics.WithSeparator("_"),
        ),
    )
    if err != nil {
        panic(err)
    }
    defer db.Close(ctx)
    ...
}
```

database/sql

```
package main

import (
    "context"
    "database/sql"

    "github.com/prometheus/client_golang/prometheus"
    metrics "github.com/ydb-platform/ydb-go-sdk-prometheus/v2"
    "github.com/ydb-platform/ydb-go-sdk/v3"
    "github.com/ydb-platform/ydb-go-sdk/v3/trace"
)

func main() {
    ctx := context.Background()
    registry := prometheus.NewRegistry()
    nativeDriver, err := ydb.Open(ctx,
        os.Getenv("YDB_CONNECTION_STRING"),
        metrics.WithTraces(
            registry,
            metrics.WithDetails(trace.DetailsAll),
            metrics.WithSeparator("_"),
        ),
    )
    if err != nil {
        panic(err)
    }
    defer nativeDriver.Close(ctx)

    connector, err := ydb.Connector(nativeDriver)
    if err != nil {
        panic(err)
    }

    db := sql.OpenDB(connector)
    defer db.Close()
    ...
}
```

Java

This functionality is not currently supported.

Python

This functionality is not currently supported.

JavaScript

This section is under development.

Rust

This functionality is not currently supported.

Track progress or vote for Rust SDK support: ydb-rs-sdk#267

Tracing with OpenTelemetry

YDB SDK instruments Query Service operations with OpenTelemetry spans, providing distributed tracing from application code to each gRPC call to YDB. The spans are exported via the standard OTLP protocol and are compatible with Jaeger, Grafana Tempo, Zipkin, and any other backend that supports OpenTelemetry.

Created spans

Currently, spans are supported for Query Service operations. Support for topics and other services is planned for future SDK versions.

The following spans are created:

Span	Type	Description
`ydb.RunWithRetry`	`Internal`	Covers the entire retry cycle for a single operation
`ydb.Try`	`Internal`	One span per attempt, including the first; child RPC spans are attached to it
`ydb.CreateSession`	`Client`	Creating a session via gRPC `CreateSession` and `AttachStream`
`ydb.ExecuteQuery`	`Client`	Execution of one YQL query
`ydb.BeginTransaction`	`Client`	Explicit call to start a transaction
`ydb.Commit`	`Client`	Transaction commit
`ydb.Rollback`	`Client`	Transaction rollback
`ydb.Driver.Initialize`	`Internal`	Initial driver initialization: cluster discovery and authentication

A typical span tree for a transactional operation with retry looks as follows:

```
ydb.RunWithRetry  (Internal)
├─ ydb.Try        (Internal)   ← 1st attempt: ERROR
│  ├─ ydb.ExecuteQuery (Client)
│  ├─ ydb.ExecuteQuery (Client)
│  └─ ydb.Commit       (Client) ← ERROR: Transaction Lock Invalidated
└─ ydb.Try        (Internal)   ← 2nd attempt: SUCCESS, ydb.retry.backoff_ms=50
   ├─ ydb.ExecuteQuery (Client)
   ├─ ydb.ExecuteQuery (Client)
   └─ ydb.Commit       (Client)
```

`ydb.RunWithRetry` is the parent span for the entire retry operation. For each execution attempt, a separate child span `ydb.Try` is created: the first `ydb.Try` corresponds to the first attempt, the second to the first **retry** attempt, and so on. RPC spans of a specific attempt, for example `ydb.ExecuteQuery` and `ydb.Commit`, are created inside the corresponding `ydb.Try`.

> **Note**
>
> If an attempt ends with an error, its span `ydb.Try` ends with an error status. On a retry, a new `ydb.Try` is created; starting from the second attempt, the attribute `ydb.retry.backoff_ms` — the wait time before this attempt in milliseconds — is set on it. This wait time is included in the duration of the next span `ydb.Try`: the span starts before the backoff pause, then after the pause, the RPC calls of this attempt are executed.

Span attributes

The SDK uses both standard OpenTelemetry semantic conventions attributes and YDB-specific extensions.

Standard OpenTelemetry attributes

The following attributes belong to the stable OpenTelemetry semantic conventions and can be processed by tracing backends as standard:

Attribute	Where installed	Description
`db.system.name`	RPC spans	Always `"ydb"`
`db.namespace`	RPC spans	Path to YDB database
`server.address`	RPC spans	Main host from the connection string
`server.port`	RPC spans	Main port from the connection string
`network.peer.address`	RPC spans	Actual gRPC endpoint used for the call
`network.peer.port`	RPC spans	Actual gRPC endpoint port used for the call
`error.type`	Spans that failed	Error type. For example: `"transport_error"`, `"ydb_error"` or the full exception class name
`db.response.status_code`	RPC spans during `YdbException`	Textual name of the YDB status from the error, for example `ABORTED`, `UNAVAILABLE`, `OVERLOADED`

YDB-specific attributes

The following attributes are YDB extensions on top of the standard semantic conventions:

Attribute	Where it is installed	Description

`ydb.node.id`	RPC spans	ID of the YDB node that processed the query
`ydb.node.dc`	RPC spans	Data center of the YDB node that processed the query
`ydb.retry.backoff_ms`	Spans `ydb.Try`, starting from the second attempt	Wait time before retry in milliseconds

W3C trace context

The SDK automatically propagates the W3C `traceparent` header in every outgoing gRPC call. This allows the YDB server to trace internal operations within the same trace, without additional configuration. For more details about server-side tracing, see the section Passing an external trace-id to YDB.

Connecting to the SDK

Go

Install the OpenTelemetry adapter for the YDB Go SDK:

```
go get github.com/ydb-platform/ydb-go-sdk-otel
```

Configure `TracerProvider` and pass the adapter to `ydb.Open`:

```
package main

import (
    "context"
    "os"

    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracegrpc"
    "go.opentelemetry.io/otel/sdk/resource"
    sdktrace "go.opentelemetry.io/otel/sdk/trace"

    "github.com/ydb-platform/ydb-go-sdk/v3"
    ydbOtel "github.com/ydb-platform/ydb-go-sdk-otel"
)

func main() {
    ctx := context.Background()

    exporter, err := otlptracegrpc.New(ctx,
        otlptracegrpc.WithEndpoint("localhost:4317"),
        otlptracegrpc.WithInsecure(),
    )
    if err != nil {
        panic(err)
    }
    res, _ := resource.Merge(resource.Default(), resource.NewSchemaless(
        attribute.String("service.name", "my-service"),
    ))
    tp := sdktrace.NewTracerProvider(
        sdktrace.WithBatcher(exporter),
        sdktrace.WithResource(res),
    )
    defer tp.Shutdown(ctx)
    otel.SetTracerProvider(tp)

    db, err := ydb.Open(ctx,
        os.Getenv("YDB_CONNECTION_STRING"),
        ydbOtel.WithTraces(
            ydbOtel.WithTracer(tp.Tracer("ydb-go-sdk")),
        ),
    )
    if err != nil {
        panic(err)
    }
    defer db.Close(ctx)
}
```

Python

Install additional dependencies `opentelemetry` and the OTLP exporter:

```
pip install ydb[opentelemetry]
pip install opentelemetry-exporter-otlp-proto-grpc
```

Call `enable_tracing()` after configuring the global `TracerProvider`:

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExporter
from opentelemetry.sdk.resources import Resource

import ydb
from ydb.opentelemetry import enable_tracing
```

```
resource = Resource(attributes={"service.name": "my-service"})
provider = TracerProvider(resource=resource)
provider.add_span_processor(
    BatchSpanProcessor(OTLPSpanExporter(endpoint="http://localhost:4317"))
)
trace.set_tracer_provider(provider)

enable_tracing()

with ydb.Driver(endpoint="grpc://localhost:2136", database="/local") as driver:
    driver.wait(timeout=5)
    with ydb.QuerySessionPool(driver) as pool:
        pool.execute_with_retries("SELECT 1")

provider.shutdown()
```

C#

Add the NuGet package:

```
dotnet add package Ydb.Sdk.OpenTelemetry
```

Register the YDB instrumentation when configuring OpenTelemetry in your service:

```
services.AddOpenTelemetry()
    .WithTracing(builder => builder
        .AddYdb()
        .AddOtlpExporter());
```

Java

Add YDB SDK and OpenTelemetry dependencies (example for Maven):

```
<dependency>
    <groupId>tech.ydb</groupId>
    <artifactId>ydb-sdk-core</artifactId>
    <version>${ydb.sdk.version}</version>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-sdk</artifactId>
    <version>${otel.version}</version>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-otlp</artifactId>
    <version>${otel.version}</version>
</dependency>
```

Create an instance of the OpenTelemetry SDK and pass it to the transport via `OpenTelemetryTracer`:

```
import io.opentelemetry.api.OpenTelemetry;
import io.opentelemetry.exporter.otlp.trace.OtlpGrpcSpanExporter;
import io.opentelemetry.sdk.OpenTelemetrySdk;
import io.opentelemetry.sdk.resources.Resource;
import io.opentelemetry.sdk.trace.SdkTracerProvider;
import io.opentelemetry.sdk.trace.export.BatchSpanProcessor;
import io.opentelemetry.semconv.resource.attributes.ResourceAttributes;
import tech.ydb.core.auth.CloudAuthHelper;
import tech.ydb.core.grpc.GrpcTransport;
import tech.ydb.core.opentelemetry.OpenTelemetryTracer;
import tech.ydb.query.QueryClient;

Resource resource = Resource.getDefault().toBuilder()
    .put(ResourceAttributes.SERVICE_NAME, "my-service")
    .build();

SdkTracerProvider tracerProvider = SdkTracerProvider.builder()
    .setResource(resource)
    .addSpanProcessor(BatchSpanProcessor.builder(
        OtlpGrpcSpanExporter.builder()
            .setEndpoint("http://localhost:4317")
            .build()
    ).build())
    .build();

OpenTelemetry openTelemetry = OpenTelemetrySdk.builder()
    .setTracerProvider(tracerProvider)
    .build();

try (GrpcTransport transport = GrpcTransport.forConnectionString(connectionString)
        .withAuthProvider(CloudAuthHelper.getAuthProviderFromEnviron())
        .withTracer(OpenTelemetryTracer.fromOpenTelemetry(openTelemetry))
        .build();
     QueryClient queryClient = QueryClient.newClient(transport).build()) {
    // Use queryClient here
}
```

When using the JDBC driver, simply add the `enableOpenTelemetryTracer=true` parameter to the connection string — the driver will pick up the global OTel provider automatically:

```
jdbc:ydb://<host>:<port>/<database>?enableOpenTelemetryTracer=true
```

C++

Include the OpenTelemetry tracing header from the YDB C++ SDK and add a dependency on the OTel C++ SDK:

```
#include <ydb-cpp-sdk/client/driver/driver.h>
#include <ydb-cpp-sdk/open_telemetry/trace.h>

#include <opentelemetry/exporters/otlp/otlp_http_exporter_factory.h>
#include <opentelemetry/exporters/otlp/otlp_http_exporter_options.h>
#include <opentelemetry/sdk/trace/tracer_provider.h>
#include <opentelemetry/sdk/trace/simple_processor_factory.h>
#include <opentelemetry/sdk/resource/resource.h>
#include <opentelemetry/trace/provider.h>

namespace sdktrace = opentelemetry::sdk::trace;
namespace otlp     = opentelemetry::exporter::otlp;
namespace resource = opentelemetry::sdk::resource;
using namespace NYdb;

// 1. Initialize the OTel tracing provider
otlp::OtlpHttpExporterOptions opts;
opts.url = "http://localhost:4318/v1/traces";
auto exporter  = otlp::OtlpHttpExporterFactory::Create(opts);
auto processor = sdktrace::SimpleSpanProcessorFactory::Create(std::move(exporter));
auto res       = resource::Resource::Create({{"service.name", "my-service"}});
auto otelProvider = std::make_shared<sdktrace::TracerProvider>(
    std::move(processor), res);
opentelemetry::trace::Provider::SetTracerProvider(otelProvider);

// 2. Wrap in the YDB tracing provider
auto ydbTraceProvider = NTrace::CreateOtelTraceProvider(otelProvider);

// 3. Create a YDB driver with tracing enabled
auto driverConfig = TDriverConfig()
    .SetEndpoint("localhost:2136")
    .SetDatabase("/local")
    .SetTraceProvider(ydbTraceProvider);

TDriver driver(driverConfig);
```

JavaScript

Install `@ydbjs/telemetry` together with the OpenTelemetry Node SDK and OTLP exporter:

```
npm install @ydbjs/telemetry @opentelemetry/sdk-node @opentelemetry/exporter-trace-otlp-http
```

Initialize `NodeSDK` before creating the driver and call `register()` from `@ydbjs/telemetry`:

```
import { NodeSDK } from '@opentelemetry/sdk-node'
import { OTLPTraceExporter } from '@opentelemetry/exporter-trace-otlp-http'
import { Driver } from '@ydbjs/core'
import { query } from '@ydbjs/query'
import { register } from '@ydbjs/telemetry'

const sdk = new NodeSDK({
    serviceName: 'my-service',
    traceExporter: new OTLPTraceExporter({ url: 'http://localhost:4318/v1/traces' }),
})
sdk.start()

// Must be called BEFORE creating the Driver — W3C propagation middleware
// trace context is set once during driver construction.
const instrumentation = register()

using driver = new Driver(process.env.YDB_CONNECTION_STRING)
await driver.ready()
await using sql = query(driver)
// ...

instrumentation.disable()
await sdk.shutdown()
```

Alternatively, use `--import` for auto-loading before starting the application:

```
node --import @opentelemetry/sdk-node/register --import @ydbjs/telemetry/register your-app.js
```

Enabling tracing in Jaeger

Below are code examples for enabling tracing in Jaeger in different YDB SDK.

C++

This functionality is not currently supported.

Go

Native SDK

```
package main

import (
    "context"
    "time"

    "github.com/opentracing/opentracing-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"

    "github.com/ydb-platform/ydb-go-sdk/v3"
    "github.com/ydb-platform/ydb-go-sdk/v3/trace"

    tracing "github.com/ydb-platform/ydb-go-sdk-opentracing"
)

const (
    tracerURL   = "localhost:5775"
    serviceName = "ydb-go-sdk"
)

func main() {
    tracer, closer, err := jaegerConfig.Configuration{
        ServiceName: serviceName,
        Sampler: &jaegerConfig.SamplerConfig{
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{
            LogSpans:            true,
            BufferFlushInterval: 1 * time.Second,
            LocalAgentHostPort:  tracerURL,
        },
    }.NewTracer()
    if err != nil {
        panic(err)
    }

    defer closer.Close()

    // set global tracer of this application
    opentracing.SetGlobalTracer(tracer)

    span, ctx := opentracing.StartSpanFromContext(context.Background(), "client")
    defer span.Finish()

    db, err := ydb.Open(ctx,
        os.Getenv("YDB_CONNECTION_STRING"),
        tracing.WithTraces(tracing.WithDetails(trace.DetailsAll)),
    )
    if err != nil {
        panic(err)
    }
    defer db.Close(ctx)
    ...
}
```

database/sql

```
package main

import (
    "context"
    "database/sql"
    "time"

    "github.com/opentracing/opentracing-go"
    jaegerConfig "github.com/uber/jaeger-client-go/config"

    "github.com/ydb-platform/ydb-go-sdk/v3"
    "github.com/ydb-platform/ydb-go-sdk/v3/trace"

    tracing "github.com/ydb-platform/ydb-go-sdk-opentracing"
)

const (
    tracerURL   = "localhost:5775"
    serviceName = "ydb-go-sdk"
)
```

```
func main() {
    tracer, closer, err := jaegerConfig.Configuration{
        ServiceName: serviceName,
            Sampler: &jaegerConfig.SamplerConfig{
            Type:  "const",
            Param: 1,
        },
        Reporter: &jaegerConfig.ReporterConfig{
            LogSpans:            true,
            BufferFlushInterval: 1 * time.Second,
            LocalAgentHostPort:  tracerURL,
        },
    }.NewTracer()
    if err != nil {
        panic(err)
    }

    defer closer.Close()

    // set global tracer of this application
    opentracing.SetGlobalTracer(tracer)

    span, ctx := opentracing.StartSpanFromContext(context.Background(), "client")
    defer span.Finish()

    nativeDriver, err := ydb.Open(ctx,
        os.Getenv("YDB_CONNECTION_STRING"),
        tracing.WithTraces(tracing.WithDetails(trace.DetailsAll)),
    )
    if err != nil {
        panic(err)
    }
    defer nativeDriver.Close(ctx)

    connector, err := ydb.Connector(nativeDriver)
    if err != nil {
        panic(err)
    }

    db := sql.OpenDB(connector)
    defer db.Close()
    ...
}
```

Java

This functionality is not currently supported.

Python

This functionality is not currently supported.

C#

This functionality is not currently supported.

JavaScript

This functionality is not currently supported.

Rust

This functionality is not currently supported.

PHP

This functionality is not currently supported.

YDB CLI recipes

This section contains recipes for various tasks that can be solved with YDB CLI.

Table of contents:

- Convert a table between row-oriented and column-oriented
- Conducting load testing
- Configuring Time to Live (TTL)

See also:

- YDB CLI
- YDB for Application Developers / Software Engineers

Convert a table between row-oriented and column-oriented

YDB supports two main types of tables: row-oriented and column-oriented. The chosen table type determines the physical representation of data on disks, so changing the type in place is impossible. However, you can create a new table of a different type and copy the data. This recipe consists of the following steps:

1. Prepare a new table
2. Copy data
3. Switch the workload *(optional)*

These instructions assume that the source table is row-oriented, and the goal is to obtain a similar column-oriented destination table; however, the table roles could be swapped.

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Prepare a new table

Take a copy of the original `CREATE TABLE` statement used for the source table. Modify the following to create a file with the `CREATE TABLE` query for the destination table:

1. Change the table name to a desired new name.
2. Set the `STORE` setting value to `COLUMN` to make it a column-oriented table.

Run this query (assuming it is saved in a file named `create_column_oriented_table.yql`):

```
$ ydb -p quickstart yql -f create_column_oriented_table.yql
```

Example test data and table schemas

Row-oriented source table:

```
CREATE TABLE `row_oriented_table` (
    id Int64 NOT NULL,
    metric_a Double,
    metric_b Double,
    metric_c Double,
    PRIMARY KEY (id)
)
```

Column-oriented destination table:

```
CREATE TABLE `column_oriented_table` (
    id Int64 NOT NULL,
    metric_a Double,
    metric_b Double,
    metric_c Double,
    PRIMARY KEY (id)
)
PARTITION BY HASH(id)
WITH (STORE = COLUMN)
```

> **Note**
>
> Refer to the documentation for application developers to learn more about partitioning column-oriented tables and choosing a partitioning key (`PARTITION BY` clause).

Fill the source row-oriented table with random data:

```
INSERT INTO `row_oriented_table` (id, metric_a, metric_b, metric_c)
SELECT
    id,
    Random(id + 1),
    Random(id + 2),
    Random(id + 3)
FROM (
    SELECT ListFromRange(1, 1000) AS id
) FLATTEN LIST BY id
```

Copy data

Currently, the recommended way to copy data between YDB tables of different types is to export and import:

1. Export data to the local filesystem:

```
$ ydb -p quickstart dump -p row_oriented_table -o tmp_backup/
```

2. Import it back into another YDB table:

```
ydb -p quickstart import file csv -p column_oriented_table tmp_backup/row_oriented_table/*.csv
```

Make sure you have enough free space in the file system to store all the data.

Switch the workload

It is currently impossible to seamlessly replace the original table with a newly created column-oriented one. However, if necessary, you can gradually switch your queries to work with the new table by replacing the original table path in the queries with the new one.

If the original table is no longer needed, it can be dropped with `ydb -p quickstart table drop row_oriented_table` or `yql -p quickstart yql -s "DROP TABLE row_oriented_table"`.

See also

- YDB CLI
- YDB for Application Developers / Software Engineers

Conducting load testing

YDB CLI has a built-in toolkit for performing load testing using standard benchmarks:

Benchmark	Reference
TPC-C	tpcc
TPC-H	tpch
TPC-DS	tpcds
ClickBench	clickbench

Also, there is a possibility to run user-defined testing scenarios, which are initiated via the `ydb workload query` command, see the description. The details are provided in the corresponding section.

All mentioned methods emulate the user's workload under the given scenarios. The detailed description of each method is provided in the corresponding sections, the links to which are provided above.

All commands for working with benchmarks are organized into corresponding groups, and the database path is specified in the same way for all commands:

```
ydb workload tpcc --path path/in/database ...
ydb workload tpch --path path/in/database ...
ydb workload tpcds --path path/in/database ...
ydb workload clickbench --path path/in/database ...
ydb workload query --path path/in/database ...
```

Load testing consists of three stages:

1. Data preparation
2. Testing
3. Cleanup

Data preparation

It consists of two steps: initializing tables and filling them with data.

Initialization

Initialization is performed by the `init` command:

```
ydb workload tpcc --path tpcc/10wh init
ydb workload clickbench --path clickbench/hits init --store=row
ydb workload tpch --path tpch/s1 init --store=column
ydb workload tpcds --path tpcds/s1 init --store=external-s3
ydb workload query --path user/suite1 init --suite-path /home/user/user_suite
```

At the stage of creating tables, for `tpch`, `tpcds`, and `clickbench`, you can configure the tables to be created:

- Select the type of tables to be used: row, column, external, etc. (parameter `--store`);
- Select the types of columns to be used: some data types from the original benchmarks can be represented by multiple YDB data types. In such cases, it is possible to select a specific one with `--string`, `--datetime`, and `--float-mode` parameters.

You can also specify that tables should be deleted before creation if they already exist using the `--clear` parameter.

For more details, see the relevant reference sections:

- tpcc init
- clickbench init
- tpch init
- tpcds init
- query init

Loading data into the tables

The `import` command is used to load data into the benchmark tables. This command is specific to each benchmark, and its behavior depends on the subcommands. However, there are also parameters common to all benchmarks.

For more details, see the relevant reference sections:

- tpcc import
- tpch import
- tpcds import
- clickbench import

Examples:

```
ydb workload tpcc --path tpcc/10wh import
ydb workload clickbench --path clickbench/hits import files --input hits.csv.gz
ydb workload tpch --path tpch/s1 import generator --scale 1
ydb workload tpcds --path tpcds/s1 import generator --scale 1
ydb workload query --path user/suite1 import --suite-path /home/user/user_suite
```

Testing

The performance testing is performed using the `run` command. Its behavior is mostly the same across different benchmarks, though some differences do exist.

Examples:

```
ydb workload tpcc --path tpcc/10wh run
ydb workload clickbench --path clickbench/hits run --include 1-5,8
ydb workload tpch --path tpch/s1 run --exclude 3,4 --iterations 3
ydb workload tpcds --path tpcds/s1 run --plan ~/query_plan --include 2 --iterations 5
ydb workload query --path user/suite1 run --plan ~/query_plan --include first_query_set.1.sql,second_query_se
t.2.sql --iterations 5
```

The command 'run' for each benchmark has a number of additional parameters for configuring the types of reports generated, collecting statistics, and other results of load testing.

For more details, see the relevant reference sections:

- tpcc run
- clickbench run
- tpch run
- tpcds run
- query run

Cleanup

After all necessary testing has been completed, the benchmark's data can be removed from the database using the `clean` command:

```
ydb workload tpcc --path tpcc/10wh clean
ydb workload clickbench --path clickbench/hits clean
ydb workload tpch --path tpch/s1 clean
ydb workload tpcds --path tpcds/s1 clean
ydb workload query --path user/suite1 clean
```

For more details, see the relevant reference sections:

- tpcc clean
- clickbench clean
- tpch clean
- tpcds clean
- query clean

Configuring Time to Live (TTL)

This section contains recipes for configuration of table's TTL with YDB CLI.

Enabling TTL for an existing table

In the example below, the items of the `mytable` table will be deleted an hour after the time set in the `created_at` column:

```
$ ydb -e <endpoint> -d <database> table ttl set --column created_at --expire-after 3600 mytable
```

The example below shows how to use the `modified_at` column with a numeric type (`Uint32`) as a TTL column. The column value is interpreted as the number of seconds since the Unix epoch:

```
$ ydb -e <endpoint> -d <database> table ttl set --column modified_at --expire-after 3600 --unit seconds mytab
le
```

Enabling data eviction to S3-compatible external storage

Warning

Supported only for column-oriented tables. Support for row-oriented tables is currently under development.

To enable data eviction, an external data source object that describes a connection to the external storage is needed. Refer to YQL recipe for examples of creating an external data source.

The example below shows how to enable data eviction by executing a YQL-query from YDB CLI. Rows of the table `mytable` will be moved to the bucket described in the external data source `/Root/s3_cold_data` one hour after the time recorded in the column `created_at` and will be deleted after 24 hours.

```
$ ydb -e <endpoint> -d <database> sql -s '
    ALTER TABLE `mytable` SET (
        TTL =
            Interval("PT1H") TO EXTERNAL DATA SOURCE `/Root/s3_cold_data`,
            Interval("PT24H") DELETE
        ON modified_at AS SECONDS
    );
'
```

Disabling TTL

```
$ ydb -e <endpoint> -d <database> table ttl reset mytable
```

Getting TTL settings

The current TTL settings can be obtained from the table description:

```
$ ydb -e <endpoint> -d <database> scheme describe mytable
```

YQL recipes

This section contains query recipes for various tasks that can be solved with YQL, YDB's SQL dialect.

Table of contents:

- Accessing values inside JSON with YQL
- Modifying JSON with YQL
- Configuring Time to Live (TTL)

See also:

- YQL - Overview
- YDB for Application Developers / Software Engineers

Accessing values inside JSON with YQL

YQL provides two main ways to retrieve values from JSON:

- Using **JSON functions from the SQL standard**. This approach is recommended for simple cases and for teams that are familiar with them from other DBMSs.
- Using **Yson UDF**, list and dict builtins, and lambdas. This approach is more flexible and tightly integrated with YDB's data type system, thus recommended for complex cases.

Below are the recipes that will use the same input JSON to demonstrate how to use each option to check whether a key exists, get a specific value, and retrieve a subtree.

JSON functions

```
$json = @@{
    "friends": [
        {
            "name": "James Holden",
            "age": 35
        },
        {
            "name": "Naomi Nagata",
            "age": 30
        }
    ]
}@@j;

SELECT
    JSON_EXISTS($json, "$.friends[*].name"), -- True
    CAST(JSON_VALUE($json, "$.friends[0].age") AS Int32), -- 35
    JSON_QUERY($json, "$.friends[0]"); -- {"name": "James Holden", "age": 35}
```

`JSON_*` functions expect the `Json` data type as an input to run. In this example, the string literal has the suffix `j`, marking it as `Json`. In tables, data could be stored in either JSON format or as a string representation. To convert data from `String` to `JSON` data type, use the `CAST` function, such as `CAST(my_string AS JSON)`.

Yson UDF

This approach typically combines multiple functions and expressions, so a query might leverage different specific strategies.

Convert the whole JSON to YQL containers

```
$json = @@{
    "friends": [
        {
            "name": "James Holden",
            "age": 35
        },
        {
            "name": "Naomi Nagata",
            "age": 30
        }
    ]
}@@j;

$containers = Yson::ConvertTo($json, Struct<friends:List<Struct<name:String?,age:Int32?>>>);
$has_name = ListAny(
    ListMap($containers.friends, ($friend) -> {
        return $friend.name IS NOT NULL;
    })
);
$get_age = $containers.friends[0].age;
$get_first_friend = Yson::SerializeJson(Yson::From($containers.friends[0]));

SELECT
    $has_name, -- True
    $get_age, -- 35
    $get_first_friend; -- {"name": "James Holden", "age": 35}
```

It is **not** necessary to convert the whole JSON object to a structured combination of containers. Some fields can be omitted if not used, while some subtrees could be left in an unstructured data type like `Json`.

Work with in-memory representation

```
$json = @@{
    "friends": [
        {
            "name": "James Holden",
            "age": 35
        },
        {
            "name": "Naomi Nagata",
            "age": 30
        }
    ]
}@@j;

$has_name = ListAny(
    ListMap(Yson::ConvertToList($json.friends), ($friend) -> {
```

```
        return Yson::Contains($friend, "name");
    })
);
$get_age = Yson::ConvertToInt64($json.friends[0].age);
$get_first_friend = Yson::SerializeJson($json.friends[0]);

SELECT
    $has_name, -- True
    $get_age, -- 35
    $get_first_friend; -- {"name": "James Holden", "age": 35}
```

See also

- Modifying JSON with YQL

Modifying JSON with YQL

In memory, YQL operates on immutable values. Thus, when a query needs to change something inside a JSON value, the mindset should be about constructing a new value from pieces of the old one.

This example query takes an input JSON named `$fields`, parses it, substitutes key `a` with 0, drops key `d`, and adds a key `c` with value 3:

```
$fields = '{"a": 1, "b": 2, "d": 4}'j;
$pairs = DictItems(Yson::ConvertToInt64Dict($fields));
$result_pairs = ListExtend(ListNotNull(ListMap($pairs, ($item) -> {
    $item = if ($item.0 == "a", ("a", 0), $item);
    return if ($item.0 == "d", null, $item);
})), [("c", 3)]);
$result_dict = ToDict($result_pairs);
SELECT Yson::SerializeJson(Yson::From($result_dict));
```

See also

- Yson
- Functions for lists
- Functions for dictionaries
- Accessing values inside JSON with YQL

Configuring Time to Live (TTL)

This section contains recipes for configuration of table's TTL with YQL.

Enabling TTL for an existing table

In the example below, the items of the `mytable` table will be deleted an hour after the time set in the `created_at` column:

```
ALTER TABLE `mytable` SET (TTL = Interval("PT1H") ON created_at);
```

Tip

An `Interval` is created from a string literal in ISO 8601 format with some restrictions.

The example below shows how to use the `modified_at` column with a numeric type (`Uint32`) as a TTL column. The column value is interpreted as the number of seconds since the Unix epoch:

```
ALTER TABLE `mytable` SET (TTL = Interval("PT1H") ON modified_at AS SECONDS);
```

Enabling data eviction to S3-compatible external storage

Warning

Supported only for column-oriented tables. Support for row-oriented tables is currently under development.

In the following example, rows of the table `mytable` will be moved to the bucket described in the external data source `/Root/s3_cold_data` one hour after the time recorded in the column `created_at` and will be deleted after 24 hours:

```
ALTER TABLE `mytable` SET (
  TTL =
      Interval("PT1H") TO EXTERNAL DATA SOURCE `/Root/s3_cold_data`,
      Interval("PT24H") DELETE
  ON modified_at AS SECONDS
);
```

Warning

Supported only for column-oriented tables. Support for row-oriented tables is currently under development.

To enable data eviction, an external data source object that describes a connection to the external storage is needed.
In the example below, an external data source `/Root/s3_cold_data` is created. It describes a connection to bucket `test_cold_data` located in Yandex Object Storage with authorization by static access keys provided via secrets `access_key` and `secret_key` .

```
CREATE SECRET access_key WITH (value="...");
CREATE SECRET secret_key WITH (value="...");

CREATE EXTERNAL DATA SOURCE `/Root/s3_cold_data` WITH (
    SOURCE_TYPE="ObjectStorage",
    AUTH_METHOD="AWS",
    LOCATION="http://storage.yandexcloud.net/test_cold_data",
    AWS_ACCESS_KEY_ID_SECRET_PATH="access_key",
    AWS_SECRET_ACCESS_KEY_SECRET_PATH="secret_key",
    AWS_REGION="ru-central1"
)
```

Follow examples below to enable data eviction using an external data source.

In the following example, rows of the table `mytable` will be moved to the bucket described in the external data source `/Root/s3_cold_data` one hour after the time recorded in the column `created_at` and will be deleted after 24 hours:

```
ALTER TABLE `mytable` SET (
    TTL =
        Interval("PT1H") TO EXTERNAL DATA SOURCE `/Root/s3_cold_data`,
        Interval("PT24H") DELETE
    ON modified_at AS SECONDS
);
```

In the following example, rows of the table `mytable` will be moved to buckets `/Root/s3_cold` and `/Root/s3_frozen` one hour and 30 days respectively after the time recorded in the column `created_at` :

```
ALTER TABLE `mytable` SET (
    TTL =
        Interval("PT1H") TO EXTERNAL DATA SOURCE `/Root/s3_cold`,
        Interval("PT30D") TO EXTERNAL DATA SOURCE `/Root/s3_frozen`
    ON modified_at AS SECONDS
);
```

````
## Enabling TTL for a newly created table {#enable-for-new-table}

For a newly created table, you can pass TTL settings along with the table description:

```yql
CREATE TABLE `mytable` (
  id Uint64,
  expire_at Timestamp,
  PRIMARY KEY (id)
) WITH (
  TTL = Interval("PT1H") ON expire_at
);
````

Disabling TTL

```
ALTER TABLE `mytable` RESET (TTL);
```
```

Vector index recipes

This section of YDB documentation contains vector index recipes:

- Vector Index — Quick Start

Vector Index — Quick Start

This article will help you quickly get started with vector indexes in YDB using a simple model example.

The article will cover the following steps for working with vector indexes:

- creating a table with vectors;
- populating the table with data;
- building a vector index;
- performing vector search without an index;
- performing vector search with an index.

Step 1. Creating a table with vectors

First, you need to create a table in YDB that will store vectors. This can be done using an SQL query:

```
CREATE TABLE Vectors (
  id Uint64,
  embedding String,
  PRIMARY KEY (id)
);
```

This `Vectors` table has two columns:

- `id` — a unique identifier for each vector
- `embedding` — a vector of real numbers packed into a string

Step 2. Populating the table with data

After creating the table, you should add vectors to it using an `UPSERT INTO` query:

```
UPSERT INTO Vectors(id, embedding)
VALUES
    (1, Untag(Knn::ToBinaryStringFloat([1.f, 1.f, 1.f, 1.f, 1.f]), "FloatVector")),
    (2, Untag(Knn::ToBinaryStringFloat([1.f, 1.f, 1.f, 1.f, 1.25f]), "FloatVector")),
    (3, Untag(Knn::ToBinaryStringFloat([1.f, 1.f, 1.f, 1.f, 1.5f]), "FloatVector")),
    (4, Untag(Knn::ToBinaryStringFloat([-1.f, -1.f, -1.f, -1.f, -1.f]), "FloatVector")),
    (5, Untag(Knn::ToBinaryStringFloat([-2.f, -2.f, -2.f, -2.f, -4.f]), "FloatVector")),
    (6, Untag(Knn::ToBinaryStringFloat([-3.f, -3.f, -3.f, -3.f, -6.f]), "FloatVector"));
```

For a description of the `Knn::ToBinaryStringFloat` function, see KNN.

Step 3. Building a vector index

To create a vector index `EmbeddingIndex` on the `Vectors` table, use the following command:

```
ALTER TABLE Vectors
ADD INDEX EmbeddingIndex
GLOBAL USING vector_kmeans_tree
ON (embedding)
WITH (
  distance=cosine)
```

This command creates an index of the `vector_kmeans_tree` type. For more information about indexes of this type, see Vector Index Type `vector_kmeans_tree`. In this model example, only the `distance=cosine` parameter is specified — it sets the distance function used by the index. This is an important parameter: the distance function must match the one the embedding model was trained for. The parameters `clusters`, `levels`, and `overlap_clusters` are autodetected and not shown here; if you have a large dataset and want to tune the index, see the detailed parameter reference in Vector index.

For general information about vector indexes, their creation parameters, and current limitations, see the section Vector Indexes.

Step 4. Searching the table without using a vector index

At this step, an exact search for the 3 nearest neighbors for the given vector `[1.f, 1.f, 1.f, 1.f, 4.f]` is performed **without** using an index.

First, the target vector is encoded into a binary representation using `Knn::ToBinaryStringFloat`.

Then, the cosine distance from the `embedding` of each row to the target vector is calculated.

Records are sorted by increasing distance, and the first three (`$K`) records are selected, which are the nearest ones.

```
$K = 3;
$TargetEmbedding = Knn::ToBinaryStringFloat([1.f, 1.f, 1.f, 1.f, 4.f]);

SELECT id, Knn::CosineDistance(embedding, $TargetEmbedding) As CosineDistance
FROM Vectors
ORDER BY Knn::CosineDistance(embedding, $TargetEmbedding)
LIMIT $K;
```

Query execution result:

```
id CosineDistance
3  0.1055728197
2  0.1467181444
1  0.1999999881
```

For detailed information about exact vector search without using vector indexes, see KNN.

Step 5. Searching the table using a vector index

To search for the 3 nearest neighbors of the vector `[1.f, 1.f, 1.f, 1.f, 4.f]` using the `EmbeddingIndex` index that was created in step 3, execute the following query:

```
$K = 3;
$TargetEmbedding = Knn::ToBinaryStringFloat([1.f, 1.f, 1.f, 1.f, 4.f]);

SELECT id, Knn::CosineDistance(embedding, $TargetEmbedding) As CosineDistance
FROM Vectors VIEW EmbeddingIndex
ORDER BY Knn::CosineDistance(embedding, $TargetEmbedding)
LIMIT $K;
```

Note that in this query, after the table name, it is specified that record selection should be performed using the vector index: `FROM Vectors VIEW EmbeddingIndex`.

Query execution result:

```
id CosineDistance
3  0.1055728197
2  0.1467181444
1  0.1999999881
```

Thanks to using the index, searching for the nearest vectors happens significantly faster on large datasets.

Conclusion

This article provides a simple example of working with vector indexes: creating a table with vectors, populating the table with vectors, building a vector index for such a table, and searching for vectors in the table using a vector index or without it.

In the case of a small table, as in this model example, it's impossible to see the difference in query performance. These examples are intended to illustrate the syntax when working with vector indexes.

For more information about vector indexes, see Vector Indexes.

Vector index with a prepared external dataset

This article walks you through working with vector indexes in YDB. It uses an English Wikipedia text dataset (485,859 rows) prepared by the Hugging Face community.

The article covers:

- creating a table for loading data;
- downloading the dataset;
- importing data into the table;
- building the vector index;
- running vector search without the index;
- running vector search with the index.

Prerequisites

To run the examples in this article you need:

- A YDB database with vector index support enabled. For a simple single-node cluster setup, see here. For production deployment options, see here.
- On your machine:
  - The YDB CLI;
  - Python 3;
- Network access from your machine to the hosts where YDB is running.

Step 1. Creating a table

First, create a table in YDB to store the data. You can do this with an SQL query:

```
CREATE TABLE wikipedia (
  id Uint64 NOT NULL,
  title Utf8,
  text Utf8,
  url Utf8,
  wiki_id Uint32,
  views Float,
  paragraph_id Uint32,
  langs Uint32,
  emb Utf8,
  embedding String,
  PRIMARY KEY (id)
);
```

Step 2. Downloading the dataset

Download the dataset from Hugging Face. It contains English Wikipedia texts split into paragraphs, with an embedding vector for each paragraph.

In your working directory, create a file `import_dataset.py` with this Python script:

```
#!/usr/bin/env python
from datasets import load_dataset

# Load the dataset
dataset = load_dataset("Cohere/wikipedia-22-12-simple-embeddings")

# Save it as CSV or another format locally
dataset['train'].to_csv('wikipedia_embeddings_train.csv', index=False)
```

Install the `datasets` package and run the script:

```
pip3 install datasets
python3 import_dataset.py
```

This creates `wikipedia_embeddings_train.csv` in your working directory with the Wikipedia texts and embeddings.

Step 3. Importing data from file into the table

Use the YDB CLI to import the file `wikipedia_embeddings_train.csv` into the table `wikipedia` created in Step 1.

In the command below:

- `<endpoint>` — endpoint of your YDB database
- `<database>` — database path of your YDB database

```
ydb -e <endpoint> -d <database> -v import file csv --path wikipedia --header wikipedia_embeddings_train.csv --timeout 30
```

In this example the test YDB server runs without authentication, so no user credentials are passed to the CLI. For CLI authentication, see here.

The file `wikipedia_embeddings_train.csv` is about 6.26 GB, so the upload may take a while. When it finishes, you can verify that all rows were loaded into the `wikipedia` table from Step 1 with:

```
SELECT COUNT(*) AS row_count FROM wikipedia
```

Query result:

```
row_count
485859
```

For YDB vector indexes, vectors must be in string format (see data types). The embeddings in the imported dataset (stored in the `emb` column) can be converted into the `embedding` column in the required format by running this script on your machine:

```
for (( i=0; i <= 490000; i+=10000 ))
do
echo $i
echo "{\"begin\":$i}" | ydb -e <endpoint> -d <database> table query execute -q 'declare $begin As Int32; UPDATE wikipedia SET embedding = Unwrap(Untag(Knn::ToBinaryStringFloat(Cast(String::SplitToList(String::ReplaceAll(String::RemoveAll(String::RemoveFirst(String::RemoveLast(emb, "]"), "["), "\n"), "  ", " "), " ") AS List<Float>)), "FloatVector")) WHERE id>=$begin AND id < $begin + 10000;'
done
```

Step 4. Building the vector index

To create the vector index `idx_vector` on the `wikipedia` table, run:

```
ALTER TABLE wikipedia
ADD INDEX idx_vector
GLOBAL USING vector_kmeans_tree
ON (embedding)
WITH (
  distance=cosine);
```

This creates a `vector_kmeans_tree` index. For more on this index type, see here.

For general information on vector indexes, creation parameters, and current limitations, see Vector Indexes.

Step 5. Search in the table without using the vector index

This step runs exact search for the 3 nearest neighbors of a given vector **without** using the index. The query vector is assumed to be produced from the search text by an encoder model (e.g. Embed from cohere.com).

First, the target vector is encoded to binary with `Knn::ToBinaryStringFloat`.

Then cosine distance is computed from each row's `embedding` to the target vector.

Rows are ordered by distance ascending, and the first three (`$K`) rows are returned as the nearest neighbors.

```
$K = 3;
$TargetEmbedding = Knn::ToBinaryStringFloat(Cast([0.1961289,0.51426697,...] AS List<Float>));

SELECT id, title, text, wiki_id, Knn::CosineDistance(embedding, $TargetEmbedding) as CosineDistance
FROM wikipedia VIEW PRIMARY KEY
ORDER BY Knn::CosineDistance(embedding, $TargetEmbedding)
LIMIT $K;
```

For details on exact vector search without vector indexes, see the Knn UDF documentation.

Step 6. Search in the table using the vector index

To search for the 3 nearest neighbors of a given vector using the index `idx_vector` created in Step 4, run:

```
$K = 3;
$TargetEmbedding = Knn::ToBinaryStringFloat(Cast([0.1961289,0.51426697,...] AS List<Float>));

SELECT id, title, text, wiki_id, Knn::CosineDistance(embedding, $TargetEmbedding) as CosineDistance
FROM wikipedia VIEW idx_vector
ORDER BY Knn::CosineDistance(embedding, $TargetEmbedding)
LIMIT $K;
```

The result when using the vector index can differ from the result without it.

Vector indexes provide **approximate** search and are optimized for speed. To improve performance, some data is pruned from consideration, so a few vectors that would have been among the nearest may be missed. Thus approximate search results can differ from exact search results.

Depending on data size, the same query can be orders of magnitude faster with the vector index than without it.

Conclusion

This article demonstrated working with a vector index and an external dataset: creating a table for vectors, loading it from an external dataset, building a vector index on that table, and running vector search with and without the index.

Exact vector search returns the true nearest neighbors but can be expensive, especially on large data. Approximate search returns results much faster with some loss in accuracy, which is often acceptable in practice.

For more on vector indexes, see here.

Fulltext search recipes

This section contains recipes for using fulltext indexes and fulltext search functions.

- Fulltext index - quick start

Fulltext index - quick start

This quick start shows how to create a fulltext index and execute fulltext queries in YDB.

Create a table and a fulltext index

```
CREATE TABLE articles (
  id Uint64,
  title String,
  body String,
  PRIMARY KEY (id)
);

ALTER TABLE articles ADD INDEX ft_idx
  GLOBAL USING fulltext_relevance
  ON (body) COVER (title)
  WITH (tokenizer=standard, use_filter_lowercase=true);
```

Insert sample data

```
UPSERT INTO articles (id, title, body) VALUES
  (1, "Intro",  "Machine learning is a field of AI"),
  (2, "Search", "Fulltext search finds documents by words"),
  (3, "Notes",  "Neural networks are used in machine learning");
```

Filter documents

```
SELECT id, title
FROM articles VIEW ft_idx
WHERE FulltextMatch(body, "machine learning")
LIMIT 20;
```

Query execution result:

```
id title
1  Intro
3  Notes
```

Rank documents

```
SELECT id, title, FulltextScore(body, "machine learning") AS relevance
FROM articles VIEW ft_idx
WHERE FulltextScore(body, "machine learning") > 0
ORDER BY relevance DESC
LIMIT 10;
```

Query execution result:

```
id title relevance
1  Intro  0.8762718444094963
3  Notes  0.8762718444094963
```

For more details, see:

- Fulltext indexes
- VIEW (Fulltext index)
- Fulltext built-in functions

Recipes for searching JSON documents

This section contains ready-made recipes for using JSON indexes to efficiently select data based on the contents of columns of type `Json` and `JsonDocument`.

The recipes are built around the JSON_EXISTS and JSON_VALUE functions with JsonPath expressions. To guarantee index usage, all examples use explicit access via `VIEW IndexName`.

- JSON index – quick start — minimal scenario: creating a table, an index, inserting data, and basic queries.
- Catalog with nested attributes — example of a product catalog with nested attributes, range conditions, and searching nested arrays.
- Parameterized queries and JsonPath variables — parameterized queries and passing variables to JsonPath via the `PASSING` section.
- Checking field type and path existence — checking the field type and path existence using JsonPath methods.

Before running the examples, make sure that JSON index support is enabled on the cluster.

JSON index – quick start

This guide shows how to create a JSON index and run queries using the JSON_EXISTS and JSON_VALUE functions in YDB.

Create a table and a JSON index

```
CREATE TABLE documents (
    id Uint64,
    payload JsonDocument,
    PRIMARY KEY (id),
    INDEX json_idx GLOBAL USING json ON (payload)
);
```

The column type `JsonDocument` stores JSON in a compact binary format and is preferred for an indexed column. Alternatively, the type `Json` (text representation) can be used.

The primary key of the table must consist of a single column of an integer type (`Uint64`, `Uint32`, `Int64`, or `Int32`) — this is a current limitation of the JSON index implementation.

Add test data

```
UPSERT INTO documents (id, payload) VALUES
    (1, JsonDocument(@@{"user": {"id": 100, "name": "Alice"}, "active": true}@@)),
    (2, JsonDocument(@@{"user": {"id": 101, "name": "Bob"}, "active": false}@@)),
    (3, JsonDocument(@@{"user": {"id": 102, "name": "Charlie"}, "archived": true}@@));
```

Here, the `@@...@@` construct is a multiline string literal, convenient for writing JSON without escaping quotes. The `JsonDocument(...)` function converts text into a value of type `JsonDocument`.

Filter by the presence of a path in the document

The JSON_EXISTS function checks whether a path specified by a JsonPath expression exists in the document.

```
SELECT id
FROM documents VIEW json_idx
WHERE JSON_EXISTS(payload, '$.user.id');
```

Result:

```
id
1
2
3
```

The path token `$.user.id` is used for index search. The index returns the result without scanning the main table.

Selecting rows with a specific document field value

The JSON_VALUE function extracts a scalar value by JsonPath; to use the index, you must specify the return type in the `RETURNING` clause:

```
SELECT id
FROM documents VIEW json_idx
WHERE JSON_VALUE(payload, '$.user.name' RETURNING Utf8) = "Alice"u;
```

Result:

```
id
1
```

When checking equality, the token «path + value» (`$.user.name = "Alice"`) is placed into the index, which ensures the highest selectivity.

Combination of conditions

Multiple calls `JSON_EXISTS` / `JSON_VALUE` on a single indexed JSON column can be combined using the `AND` and `OR` operators:

```
SELECT id
FROM documents VIEW json_idx
WHERE JSON_EXISTS(payload, '$.user.id')
  AND JSON_VALUE(payload, '$.active' RETURNING Bool);
```

Result:

```
id
1
```

See also

- JSON indexes — a complete overview of capabilities and limitations.
- VIEW (JSON index) — query syntax through `VIEW`.
- INDEX (CREATE TABLE) — syntax for creating a JSON index.

- Catalog with nested attributes — example of a product catalog with nested attributes.

Catalog with nested attributes

This recipe shows how to use a JSON index to speed up access to a product catalog where product characteristics are stored as a JSON document with an arbitrary set of fields. This schema is convenient when:

- The set of product attributes is unknown in advance or differs for different categories.
- Adding new attributes via `ALTER TABLE ADD COLUMN` is undesirable.
- You need to efficiently filter products by various combinations of attributes.

A JSON index enables efficient retrieval by any path and value inside a JSON document without a full table scan.

Create a table and index

```
CREATE TABLE products (
    sku_id Uint64,
    attrs JsonDocument,
    PRIMARY KEY (sku_id),
    INDEX attrs_json_idx GLOBAL USING json ON (attrs)
);
```

In this schema:

- `sku_id` — numeric product identifier.
- `attrs` — product attributes in `JsonDocument` format. Storing as `JsonDocument` saves space and speeds up deserialization compared to `Json`.
- `attrs_json_idx` — JSON index on column `attrs`. It is updated synchronously together with the main table.

Load test data

```
UPSERT INTO products (sku_id, attrs) VALUES
    (10, JsonDocument(@@{
        "brand": "ACME",
        "price": 49.90,
        "category": "tools",
        "warehouses": [{"id": 1, "stock": 12}, {"id": 2, "stock": 0}]
    }@@)),
    (11, JsonDocument(@@{
        "brand": "ACME",
        "price": 199.00,
        "category": "electronics",
        "warehouses": [{"id": 1, "stock": 3}]
    }@@)),
    (12, JsonDocument(@@{
        "brand": "Globex",
        "price": 25.00,
        "category": "tools",
        "warehouses": [{"id": 2, "stock": 0}]
    }@@));
```

Filter by brand and price range

```
SELECT sku_id, attrs
FROM products VIEW attrs_json_idx
WHERE JSON_VALUE(attrs, '$.brand' RETURNING Utf8) = "ACME"u
  AND JSON_VALUE(attrs, '$.price' RETURNING Double) BETWEEN 10.0 AND 100.0;
```

What happens:

- For the `$.brand = "ACME"` condition, the token «path + value» goes into the index — this gives an exact match and maximum selectivity.
- For the `BETWEEN 10.0 AND 100.0` condition, only the path token `$.price` goes into the index. This narrows the set of rows to those where the `price` field is present, after which the query execution engine checks the range with an exact comparison (post-filter).
- The conditions are combined via `AND` — this allows the index to be used to check both fragments at once, resulting in minimal index reads.

Result:

```
sku_id attrs
10     {"brand":"ACME","price":49.9,...}
```

Search by nested array

JsonPath supports accessing array elements and filters within the path. For example, you can find products that have stock in at least one warehouse:

```
SELECT sku_id
FROM products VIEW attrs_json_idx
WHERE JSON_EXISTS(attrs, '$.warehouses ? (@.stock > 0)');
```

For index search, the path token `$.warehouses.stock` is used, and the `@.stock > 0` condition is checked by a post-filter on each found record. This is efficient when the corresponding field is present in a relatively small portion of documents.

Result:

```
sku_id
10
11
```

Search by category and attribute presence

Conditions can be combined with any `AND` and `OR` operators. Search for products in category `tools` that have the `price` field specified:

```
SELECT sku_id
FROM products VIEW attrs_json_idx
WHERE JSON_VALUE(attrs, '$.category' RETURNING Utf8) = "tools"u
  AND JSON_EXISTS(attrs, '$.price');
```

Result:

```
sku_id
10
12
```

See also

- Supported JSON index predicates — full rules for which expressions are indexed.
- AND and OR handling — nuances of combining conditions, including with non-indexable predicates.
- Parameterized queries and JsonPath variables — parameterized versions of the same queries.

Parameterized queries and JsonPath variables

In most applications, query input data is not substituted into SQL text but is passed via parameters. Typical use cases for parameters when working with JSON indexes:

1. direct comparison of the result of `JSON_VALUE` with a parameter.
2. checking whether the result is in a list of values using the `IN` expression.
3. passing a parameter to a JsonPath expression via the `PASSING` clause.

Prerequisites

Below is an example of a table, a JSON index, and data population.

```
CREATE TABLE documents (
    id Uint64,
    payload JsonDocument,
    PRIMARY KEY (id),
    INDEX json_idx GLOBAL USING json ON (payload)
);

UPSERT INTO documents (id, payload) VALUES
    (1, JsonDocument(@@{
        "owner_id": 100,
        "tag": "active",
        "archived": false,
        "content": {"x": 1, "y": 1}
    }@@)),
    (2, JsonDocument(@@{
        "owner_id": 100,
        "tag": "draft",
        "archived": false,
        "content": {"x": 1, "y": 2}
    }@@)),
    (3, JsonDocument(@@{
        "owner_id": 101,
        "tag": "active",
        "archived": true,
        "content": {"x": 2, "y": 1}
    }@@)),
    (4, JsonDocument(@@{
        "owner_id": 102,
        "tag": "pending",
        "archived": false,
        "content": {"x": 2, "y": 2}
    }@@));
```

Direct comparison with a parameter

The most common case is that the parameter is substituted as the right operand of the comparison, and the left operand is a call to `JSON_VALUE` with an explicit `RETURNING`:

```
DECLARE $owner_id AS Int64;

SELECT id, payload
FROM documents VIEW json_idx
WHERE JSON_VALUE(payload, '$.owner_id' RETURNING Int64) = $owner_id
  AND JSON_EXISTS(payload, '$.archived ? (@ == false)');
```

The index receives a token «path + value» (`$.owner_id = $owner_id`) — the parameter is treated as a regular value. This allows performing a query with the same selectivity as when comparing with a literal.

Running with `$owner_id = 100` returns rows `1` and `2`.

Search by a list of values

To search by multiple values of a single field, you can use `IN`:

```
DECLARE $tags AS List<Utf8>;

SELECT id
FROM documents VIEW json_idx
WHERE JSON_VALUE(payload, '$.tag' RETURNING Utf8) IN $tags;
```

The behavior of the index differs depending on whether it is a literal or a parameter:

- `IN ("active"u, "pending"u)` — a list of literals is converted into `OR` of several «path + value» tokens, one for each value in the list.
- `IN $tags` — for each parameter value during query execution, a «path + value» token is formed (`$.tag` + value from the list). At the compilation stage, the parameter values are not yet known, so in the query plan (`EXPLAIN`) such a condition is displayed as a «path + parameter» pair (`{"path": "$.tag", "param": "$tags"}`).

Any collection of scalar values can serve as a parameter for `IN`: `List<T>`, `Tuple<T, ...>`, `Dict<K, V>`, or `Set<T>`.

Running with `$tags = ["active"u, "pending"u]` returns rows `1`, `3`, and `4`.

Parameters inside JsonPath (PASSING)

If a parameter must be used inside a JsonPath filter (`? (...)`), it is passed in the `PASSING` clause:

```
DECLARE $min_stock AS Int64;

SELECT id
FROM documents VIEW json_idx
WHERE JSON_EXISTS(
    payload,
    '$.content ? (@.y > $threshold)'
    PASSING $min_stock AS threshold
);
```

For index search, the path token `$.content.y` is used, and the condition `@.y > $threshold` is checked by a post-filter.

Similarly, `PASSING` works in `JSON_VALUE`:

```
DECLARE $v AS Int64;

SELECT id
FROM documents VIEW json_idx
WHERE JSON_VALUE(
    payload,
    '$.content ? (@.y == $val)'
    PASSING $v AS val
    RETURNING Int64
) = 10;
```

Supported parameter types

For all three methods, parameters with the following types are supported: `Int8` … `Int64`, `Uint8` … `Uint64`, `Float`, `Double`, `Bytes` (`String`), `Text` (`Utf8`), `Bool`. Optional types (`Optional<T>`) are not supported in parameters.

For more details about types, see JSON_VALUE.

See also

- JSON index – quick start — basic use case for a JSON index.
- Passing parameters to JSON index predicates — detailed description of all options.
- JsonPath — JsonPath language syntax.

Checking field type and path existence

This recipe shows how a JSON index is used to check document structure: the presence of a specific path, the type of value at the path, etc. These tasks are common when working with heterogeneous JSON documents where some fields are optional or may have different types.

Prerequisites

```
CREATE TABLE documents (
    id Uint64,
    payload JsonDocument,
    PRIMARY KEY (id),
    INDEX json_idx GLOBAL USING json ON (payload)
);

UPSERT INTO documents (id, payload) VALUES
    (1, JsonDocument(@@{"archived": false, "value": 1,    "data": [1, 2, 3]}@@)),
    (2, JsonDocument(@@{"archived": false, "value": 2,    "data": "plain text"}@@)),
    (3, JsonDocument(@@{"archived": true,  "value": 3,    "data": {"nested": true}}@@)),
    (4, JsonDocument(@@{"archived": false, "value": null, "meta": "no data field"}@@));
```

Find documents where a field contains an array

The JsonPath method `.type()` returns a string name of the value type at the specified path. This allows filtering documents by content type:

```
SELECT id
FROM documents VIEW json_idx
WHERE JSON_VALUE(payload, '$.data.type()' RETURNING Utf8) = "array"u;
```

The path token `$.data` is added to the index — the method `.type()` completes the path token construction. The exact check of the string value `"array"` is performed by a post-filter.

Result:

```
id
1
```

The full list of values returned by the method `.type()` is given in the JsonPath syntax description.

Find documents where a field is a non-empty array

The method `.size()` returns the number of array elements (or 1 for a scalar, 0 for a missing path). In combination with a type filter:

```
SELECT id
FROM documents VIEW json_idx
WHERE JSON_VALUE(payload, '$.data.type()' RETURNING Utf8) = "array"u
  AND JSON_VALUE(payload, '$.data.size()' RETURNING Int64) > 0;
```

Both fragments are indexed by the corresponding paths, and the exact values are checked by a post-filter. Result: `id = 1`.

Find documents with value `false` or `null`

To check for "value equals `false`" or "value equals `null`", use JsonPath inside `JSON_EXISTS`, not `JSON_VALUE(...) IS NULL`:

```
-- The value of the 'archived' field is false
SELECT id
FROM documents VIEW json_idx
WHERE JSON_EXISTS(payload, '$.archived ? (@ == false)');

-- The value of the 'value' field is null
SELECT id
FROM documents VIEW json_idx
WHERE JSON_EXISTS(payload, '$.value ? (@ == null)');
```

The index search operation includes the path `$.archived` (or `$.value`) and the values `false` or `null`, respectively.

> **Note**
>
> The above conditions select documents where the specified attribute is explicitly set to false or null. Checking for the absence of an attribute cannot be done using a JSON index.

See also

- JSON_EXISTS — what is allowed in JsonPath expressions.
- JSON_VALUE — what is allowed when extracting values.
- JsonPath: methods — list of methods (`type` , `size` , `keyvalue` , ...) and JsonPath predicates.

Local index recipes

Practical examples of local indexes.

- Bloom skip index quickstart
- Quick start with the min_max index

Bloom skip index quickstart

Column-oriented (OLAP) table: bloom_filter

Below is a minimal example: a column-oriented table with a primary key and a local `bloom_filter` index on a column that is frequently used in filters.

```
CREATE TABLE events (
    id Uint64 NOT NULL,
    resource_id Utf8 NOT NULL,
    payload String,
    message Utf8,
    PRIMARY KEY (id),
    INDEX idx_res LOCAL USING bloom_filter
        ON (resource_id)
        WITH (false_positive_probability = 0.01)
)
WITH (
    STORE = COLUMN
);
```

Extending the example: n-gram index

You can add `bloom_ngram_filter` on a string column to the same table from the example above (column-oriented tables):

```
ALTER TABLE events
  ADD INDEX idx_msg LOCAL USING bloom_ngram_filter
  ON (message)
  WITH (
    ngram_size = 3,
    false_positive_probability = 0.01,
    case_sensitive = true
  );
```

Queries and the effect

After you load data, selective queries that filter on indexed columns may read less data: while scanning storage, the Bloom skip index skips fragments that cannot contain the requested value (compared to reading the full column without this filter).

Sample data and queries for the column-oriented table above:

```
INSERT INTO events (id, resource_id, payload, message) VALUES
    (1, "res-1", "{}", "started"),
    (2, "res-42", "{}", "error: timeout"),
    (3, "res-2", "{}", "done");
```

Equality on the column protected by `bloom_filter` — the engine can prune irrelevant fragments when reading `resource_id` and related columns:

```
SELECT id, message
FROM events
WHERE resource_id = "res-42";
```

Substring search on the column with `bloom_ngram_filter` — the n-gram index helps drop fragments that cannot contain matching n-grams in `message`:

```
SELECT id, message
FROM events
WHERE message LIKE '%timeout%';
```

Row-oriented (OLTP) table: prefix bloom filter

On a row-oriented table, a `bloom_filter` index is built over a left prefix of the primary key. The indexed columns must be a contiguous leading subset of the primary key columns:

```
CREATE TABLE orders (
    customer_id Utf8 NOT NULL,
    order_id Utf8 NOT NULL,
    amount Decimal(10,2),
    PRIMARY KEY (customer_id, order_id),
    -- Prefix bloom filter on the first key column
    INDEX idx_customer LOCAL USING bloom_filter
        ON (customer_id)
        WITH (false_positive_probability = 0.001),
    -- Prefix bloom filter on the full primary key
    INDEX idx_full_key LOCAL USING bloom_filter
        ON (customer_id, order_id)
);
```

Point lookups and range scans that constrain the leading key columns can skip irrelevant data fragments. A query that filters on `customer_id` only uses the prefix bloom filter `idx_customer`:

```
SELECT amount FROM orders WHERE customer_id = "cust-42";
```

A query that filters on the full primary key uses `idx_full_key`:

```
SELECT amount FROM orders WHERE customer_id = "cust-42" AND order_id = "ord-1001";
```

How to verify index effectiveness

To check that the Bloom skip index actually helps, run the same selective query on a table with enough data before and after creating the index and compare execution time.

Further reading:

- Details and limitations: Bloom skip indexes
- Parameter tuning: Parameter tuning
- Full syntax: ALTER TABLE ADD INDEX

Quick start with the min_max index

Creating a table with the min_max index

Below is a minimal example: a columnar table with a primary key and a local index of type `min_max` on columns that are often used in filters.

```
CREATE TABLE events (
    id Uint64 NOT NULL,
    created_at Timestamp,
    level Int32,
    resource_id Utf8,
    PRIMARY KEY (id),
    INDEX idx_created_at LOCAL USING min_max
        ON (created_at),
    INDEX idx_level LOCAL USING min_max
        ON (level)
)
WITH (
    STORE = COLUMN
);
```

Queries and effect

After data is loaded, selective queries with filters on indexed columns can read less data: during a storage scan, the min_max index skips fragments whose value range cannot contain matching rows.

Example data and queries for the table above:

```
INSERT INTO events (id, created_at, level, resource_id) VALUES
    (1, Timestamp("2024-01-01T00:00:00.000000Z"), 1, "res-1"),
    (2, Timestamp("2024-01-01T00:01:00.000000Z"), 3, "res-42"),
    (3, Timestamp("2024-01-02T00:00:00.000000Z"), 5, "res-2");
```

Range filter on the timestamp column:

```
SELECT id, resource_id
FROM events
WHERE created_at >= Timestamp("2024-01-01T00:00:00.000000Z")
  AND created_at <  Timestamp("2024-01-02T00:00:00.000000Z");
```

Range filter on the numeric column:

```
SELECT id, resource_id
FROM events
WHERE level BETWEEN 2 AND 4;
```

How to check index efficiency

To verify that the min_max index really helps, run the same selective query on a table with a sufficient amount of data before and after creating the index and compare the execution time.

Additionally:

- Details and limitations: min_max index
- Full syntax: ALTER TABLE ADD INDEX

Topic-to-Table Data Transfer Recipes

This section provides practical examples for setting up a transfer from topics to tables.

Contents:

- Transfer — quick start
- Transfer — streaming NGINX access logs to a table

Transfer — quick start

This guide helps you get started with transfer in YDB using a basic example.

This guide covers the following steps for working with transfers:

- creating a topic, for the transfer to read from;
- creating a table, for the transfer to write data to;
- creating the transfer;
- populating the topic with data;
- verifying the table contents.

Step 1. Create a topic

First, you need to create a topic in YDB that the transfer will read data from. You can do this using a YQL query:

```
CREATE TOPIC `transfer_recipe/source_topic`;
```

The `transfer_recipe/source_topic` topic lets you transfer any unstructured data.

Step 2. Create a table

After creating the topic, you need to create a table that will receive data from the `source_topic` topic. You can do this using a YQL query:

```
CREATE TABLE `transfer_recipe/target_table` (
  partition Uint32 NOT NULL,
  offset Uint64 NOT NULL,
  data String,
  PRIMARY KEY (partition, offset)
);
```

The `transfer_recipe/target_table` table has three columns:

- `partition` — the ID of the topic partition the message was received from;
- `offset` — the sequence number that identifies the message within its partition;
- `data` — the message body.

Step 3. Create a transfer

After creating the topic and the table, you need to create a data transfer that will move messages from the topic to the table. You can do this using a YQL query:

```
$transformation_lambda = ($msg) -> {
    return [
        <|
            partition: $msg._partition,
            offset: $msg._offset,
            data: $msg._data
        |>
    ];
};

CREATE TRANSFER `transfer_recipe/example_transfer`
  FROM `transfer_recipe/source_topic` TO `transfer_recipe/target_table`
  USING $transformation_lambda;
```

In this example:

- `$transformation_lambda` - a transformation rule for converting a topic message into table columns. In this case, the topic message is transferred to the table without any changes. To learn more about configuring transformation rules, see the documentation;
- `$msg` - a variable that contains the topic message being processed.

Step 4. Populate the topic with data

After creating the transfer, you can write messages to the topic, for example, using the YDB CLI.

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

```
echo "Message 1" | ydb --profile quickstart topic write 'transfer_recipe/source_topic'
echo "Message 2" | ydb --profile quickstart topic write 'transfer_recipe/source_topic'
echo "Message 3" | ydb --profile quickstart topic write 'transfer_recipe/source_topic'
```

Step 5. Verify the table contents

After writing messages to the `source_topic` topic, records will appear in the `transfer_recipe/target_table` table after a short delay. You can verify this using a YQL query:

```
SELECT *
FROM `transfer_recipe/target_table`;
```

Query result:

partition	offset	data
0	0	Message 1
0	1	Message 2
0	2	Message 3

Rows are not added to the table for each message received from the topic; instead, they are buffered and inserted in batches. By default, data is written to the table every 60 seconds or when the volume of accumulated data reaches 8 MB. These parameters can be explicitly configured when creating a transfer or modified later.

Conclusion

This guide provides a basic example of working with a transfer: creating a topic, table, and transfer, writing data to the topic, and verifying the result.

These examples are designed to illustrate the syntax for working with transfers. For a more realistic example, see the article that describes how to stream NGINX access logs.

See Also

- Data transfer
- Transfer — streaming NGINX access logs to a table

Transfer — streaming NGINX access logs to a table

This guide explains how to set up streaming of NGINX access logs to a YDB table for further analysis. It covers the default NGINX access log format. To learn more about the NGINX log format and how to configure it, see the NGINX documentation.

The default NGINX access log format is as follows:

```
$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$http_referer" "$http_user_age
nt"
```

Example:

```
::1 - - [01/Sep/2025:15:02:47 +0500] "GET /favicon.ico HTTP/1.1" 404 181 "-" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/136.0.0.0 YaBrowser/25.6.0.0 Safari/537.36"
::1 - - [01/Sep/2025:15:02:51 +0500] "GET / HTTP/1.1" 200 409 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKi
t/537.36 (KHTML, like Gecko) Chrome/136.0.0.0 YaBrowser/25.6.0.0 Safari/537.36"
::1 - - [01/Sep/2025:15:02:51 +0500] "GET /favicon.ico HTTP/1.1" 404 181 "http://localhost/" "Mozilla/5.0 (X1
1; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/136.0.0.0 YaBrowser/25.6.0.0 Safari/537.36"
```

This guide covers the following steps:

- creating a table to write the data to;
- creating the transfer;
- verifying the table contents.

Prerequisites

To complete the examples in this guide, it needs:

- A YDB cluster, version 25.2 or later. For instructions on how to install a single-node YDB cluster, see the guide. For recommendations on deploying YDB for production use, see this guide.
- An installed NGINX HTTP server with access logging enabled, or access to NGINX access logs from another server.
- A configured process to stream NGINX access logs from a file to the `transfer_recipe/access_log_topic` topic. For example, you can use Kafka Connect configured to stream data from a file to a topic.

Step 1. Create a table

Create a table that will receive data from the `transfer_recipe/access_log_topic` topic. You can do this using a YQL query:

```
CREATE TABLE `transfer_recipe/access_log` (
  partition Uint32 NOT NULL,
  offset Uint64 NOT NULL,
  line Uint64 NOT NULL,
  remote_addr String,
  remote_user String,
  time_local Timestamp,
  request_method String,
  request_path String,
  request_protocol String,
  status Uint32,
  body_bytes_sent Uint64,
  http_referer String,
  http_user_agent Utf8,
  PRIMARY KEY (partition, offset, line)
);
```

The `transfer_recipe/access_log` table has three service columns:

- `partition` — the ID of the topic partition the message was received from;
- `offset` — the sequence number that identifies the message within its partition;
- `line` — the sequence number of the log line within the message.

Together the `partition`, `offset` and `line` columns uniquely identify each line from the access log file.

If you need to store access log data for a limited time, you can configure the automatic deletion of old table rows. For example, to set a retention period of 24 hours, you can use a YQL query:

```
ALTER TABLE `transfer_recipe/access_log` SET (TTL = Interval("PT24H") ON time_local);
```

Step 2. Create a transfer

After creating the topic and the table, you need to create a data transfer that will move messages from the topic to the table. You can do this using a YQL query:

```
$transformation_lambda = ($msg) -> {
    -- Function to convert a log line into a table row
    $line_lambda = ($line) -> {
        -- First, split the line by the " (double quote) character to separate strings that may contain space
s.
        -- The strings themselves cannot contain double quotes; instead, they are replaced with the \x22 char
acter sequence.
        $parts = String::SplitToList($line.1, '"');
        -- Split each resulting part that isn't a quoted string by spaces.
        $info_parts = String::SplitToList($parts[0], " ");
        $request_parts = String::SplitToList($parts[1], " ");
        $response_parts = String::SplitToList($parts[2], " ");
```

```
        -- Convert the date to the Datetime type
        $dateParser = DateTime::Parse("%d/%b/%Y:%H:%M:%S");
        $date = $dateParser(SUBSTRING($info_parts[3], 1));

        -- Return a structure where each named field corresponds to a table column.
        -- Important: The data types of the named fields must match the data types of the table columns. For
example, if a column is of type Uint32,
        -- the value of the named field must also be of type Uint32. Otherwise, an explicit CAST is required.
        -- Values for NOT NULL columns must be extracted from optional types using the Unwrap function.
        return <|
            partition: $msg._partition,
            offset: $msg._offset,
            line: $line.0,
            remote_addr: $info_parts[0],
            remote_user: $info_parts[2],
            time_local: DateTime::MakeTimestamp($date),
            request_method: $request_parts[0],
            request_path: $request_parts[1],
            request_protocol: $request_parts[2],
            status: CAST($response_parts[1] AS Uint32),
            body_bytes_sent: CAST($response_parts[2] AS Uint64),
            http_referer: $parts[3],
            http_user_agent: CAST(String::CgiUnescape($parts[5]) AS Utf8) -- Explicitly cast to Utf8, because
the http_user_agent column is of type Utf8, not String
        |>;
    };


    $split = String::SplitToList($msg._data, "\n"); -- If a message contains multiple log lines, split it int
o individual lines
    $lines = ListFilter($split, ($line) -> { -- Filter out empty lines, which can be caused by a trailing \n
character
        return LENGTH($line) > 0;
    });

    -- Convert each access log line into a table row
    return ListMap(ListEnumerate($lines), $line_lambda);
};

CREATE TRANSFER `transfer_recipe/access_log_transfer`
  FROM `transfer_recipe/access_log_topic` TO `transfer_recipe/access_log`
  USING $transformation_lambda;
```

In this example:

- `$transformation_lambda` - a transformation rule for converting a topic message into table columns. Each access log line within the message is processed individually using `line_transformation_lambda`;
- `$line_lambda` - a transformation rule for converting a single access log line into a table row;
- `$msg` - variable that contains the topic message being processed.

Step 3. Verify the table contents

After writing messages to the `transfer_recipe/access_log_topic` topic, records will appear in the `transfer_recipe/access_log` table after a short delay. You can verify this using a YQL query:

```
SELECT *
FROM `transfer_recipe/access_log`;
```

Query result:

#	partition	offset	line	remote_addr	remote_user	time_local	request_method	request_path	request_protocol	status	bo
1	0	2	0	::1	-	2025-09-01T15:02:51.000000Z	GET	/favicon.ico	HTTP/1.1	404	18
2	0	1	0	::1	-	2025-09-01T15:02:51.000000Z	GET	/	HTTP/1.1	200	40
3	0	0	0	::1	-	2025-09-01T15:02:47.000000Z	GET	/favicon.ico	HTTP/1.1	404	18

Rows are not added to the table for each message received from the topic; instead, they are buffered and inserted in batches. By default, data is written to the table every 60 seconds or when the volume of accumulated data reaches 8 MB. These parameters can be explicitly configured when creating a transfer or modified later.

Conclusion

This guide demonstrates how to stream NGINX access logs to a YDB table. You can process logs of any other text format in a similar way: create a table to store the required data from the log, and write a lambda function to correctly transform the log lines into table rows.

See Also

- Data transfer
- Transfer — quick start
- Lambda function
- CREATE TABLE
- CREATE TOPIC
- CREATE TRANSFER
- UNWRAP function
- COALESCE function

Backup and restore via NFS

This recipe describes preparing NFS for the server commands `export nfs` and `import nfs`: cluster requirements, access rights, step-by-step configuration of the NFS server and clients, feature flags, performance, and diagnostics. See the reference for command syntax: export to NFS, import from NFS.

Prerequisites

The `export nfs` / `import nfs` operations are performed by **server processes** YDB, not by the CLI client. This means the file system must be accessible from each cluster node:

- The NFS directory must be mounted on **all nodes** of the YDB cluster at the same absolute path.
- The YDB process must have **write** permissions for export and **read** permissions for import to this directory.
- The `--fs-path` path must be absolute. Relative paths, paths containing `..` or `.`, and paths with null bytes will be rejected by the server.
- The functionality must be enabled using the feature flag `enable_fs_backups`.

> **Warning**
>
> This functionality is compatible with the NFSv4 network file system protocol.

Access rights

File operations are performed by server processes YDB, not by the CLI client. Therefore, permissions to the NFS directory must be granted to the OS user under which the `ydbd` process runs.

POSIX permissions:

- For **export** (writing files): `rwx` on the directory (read for traversal, write for creating files and subdirectories, execute for navigation) and `rw` on files.
- For **import** (reading files): `rx` on the directory (read for traversal, execute for navigation) and `r` on files.

NFSv4 ACL:

NFSv4 uses an extended permission model instead of POSIX `rwx`. The same letter in an NFSv4 ACE has different meanings for files and directories:

Letter	For files	For directories
`r`	`read-data` — read content	`list-directory` — view content
`w`	`write-data` — write data	`create-file` — create files
`a`	`append-data` — append data	`create-subdirectory` — create subdirectories
`x`	`execute` — execute	`change-directory` — navigate
`D`	—	`delete-child` — delete child objects
`t`	`read-attributes`	`read-attributes`

Permissions on the NFS directory for **export** (creating files and subdirectories):

- On the directory itself: `w` (create-file), `a` (create-subdirectory), `x` (change-directory), `t` (read-attributes), `D` (delete-child).
- On created files: `w` (write-data), `a` (append-data), `t` (read-attributes).

Permissions on the NFS directory for **import** (reading files):

- On the directory itself: `r` (list-directory), `x` (change-directory), `t` (read-attributes).
- On files: `r` (read-data), `t` (read-attributes).

Since files are created by the `ydbd` process during export, permissions on them cannot be set in advance. In NFSv4, **ACE inheritance** is used for this — the `f` (file-inherit) and `d` (directory-inherit) flags on the parent directory's ACE. When a new file or subdirectory is created, the ACE with these flags is automatically copied to the new object.

Step-by-step setup NFS-server and clients

Below are instructions for configuring the NFS server and YDB client nodes for two permission management options: standard POSIX permissions and extended NFSv4 ACL.

Option 1: POSIX permissions

Configuring the NFS server

1. Install the NFS server package:

```
sudo apt install nfs-kernel-server
```

2. Create a directory for export and assign the owner:

```
sudo mkdir -p /mnt/ydb-backup
sudo chown ydb:ydb /mnt/ydb-backup
sudo chmod 755 /mnt/ydb-backup
```

Options of the `chmod` command:

- `7` (rwx) — the owner (`ydb`) can read, write, and enter the directory.
- `5` (r-x) — the group and others can read and enter, but not write.

3. Add the directory to `/etc/exports`:

   ```
   /mnt/ydb-backup ydb-node-*.example.com(rw,sync,root_squash)
   ```

   Export options:

   - `rw` — allow read and write.
   - `sync` — synchronous write (data is written to disk before acknowledging the client).
   - `root_squash` — map `root` on the client to an anonymous user.
4. Apply the changes and start the NFS server:

   ```
   sudo exportfs -arv
   sudo systemctl enable --now nfs-server
   ```

   Options of `exportfs`:

   - `-a` — export all directories from `/etc/exports`.
   - `-r` — re-export all directories (synchronization with `/etc/exports`).
   - `-v` — verbose output.

Configuring clients (YDB nodes)

1. Install the NFS client package:

   ```
   sudo apt install nfs-common
   ```
2. Create a mount point:

   ```
   sudo mkdir -p /mnt/ydb-backup
   ```
3. Mount the NFS directory:

   ```
   sudo mount -t nfs4 -o rw nfs-server.example.com:/mnt/ydb-backup /mnt/ydb-backup
   ```

   Mount options:

   - `-t nfs4` — NFSv4 file system type.
   - `-o rw` — mount with read and write permissions.

   In this example, the host server does not belong to the YDB cluster, but backup and restore will work correctly if one of the cluster hosts is designated as the NFS server.
4. To enable automatic mounting at boot, add a line to `/etc/fstab`:

   ```
   nfs-server.example.com:/mnt/ydb-backup /mnt/ydb-backup nfs4 rw,hard,timeo=600,retrans=2,_netdev 0 0
   ```

   `fstab` options:

   - `rw` — read and write.
   - `hard` — retry requests indefinitely if the server is unavailable (do not lose data).
   - `timeo=600` — request timeout in tenths of a second (60 seconds).
   - `retrans=2` — number of retries before reporting an error.
   - `_netdev` — mount only after network initialization.

Verification

On the server (or any client), create a test file as user `ydb`:

```
sudo -u ydb touch /mnt/ydb-backup/test-file
```

On all client nodes, verify that the file is visible:

```
sudo -u ydb ls /mnt/ydb-backup/test-file
```

After verification, delete the test file:

```
sudo -u ydb rm /mnt/ydb-backup/test-file
```

Option 2: NFSv4 ACL

NFSv4 ACLs allow you to set more granular access permissions and configure permission inheritance for newly created files and directories.

Configuring the NFS server

1. Install the NFS server package and utilities for working with NFSv4 ACLs:

   ```
   sudo apt install nfs-kernel-server nfs4-acl-tools
   ```
2. Create a directory for export:

```
sudo mkdir -p /mnt/ydb-backup-server
```

3. Add the directory to `/etc/exports`:

```
/mnt/ydb-backup-server ydb-node-*.example.com(rw,sync,no_subtree_check,root_squash)
```

4. Apply the changes:

```
sudo exportfs -arv
sudo systemctl enable --now nfs-server
```

Configuring clients (YDB nodes)

Unlike POSIX permissions, which are set on the NFS server, NFSv4 ACLs are configured on each client after mounting the directory. If the NFS server is also a YDB cluster node, all the steps described below must be performed on it.

1. Install the NFS client packages and utilities for working with NFSv4 ACLs:

```
sudo apt install nfs-common nfs4-acl-tools
```

2. Create a mount point:

```
sudo mkdir -p /mnt/ydb-backup
```

3. Mount the NFS directory using the NFSv4 protocol:

```
sudo mount -t nfs4 -o rw nfs-server.example.com:/mnt/ydb-backup /mnt/ydb-backup
```

   Options:

   - `-t nfs4` — explicitly specify the NFSv4 protocol (required for correct ACL operation).
   - `-o rw` — mount with read and write permissions.

4. To enable automatic mounting, add a line to `/etc/fstab`:

```
nfs-server.example.com:/mnt/ydb-backup-server /mnt/ydb-backup nfs4 rw,hard,timeo=600,retrans=2,_netde
v 0 0
```

5. Determine the UID of user `ydb`:

```
id -u ydb
```

   Example output: `1871`

6. Configure NFSv4 ACLs on the directory:

```
sudo nfs4_setfacl -s A::<uid>:rwaxtD,A:fi:OWNER@:rwat,A:di:OWNER@:rwaxtD /mnt/ydb-backup
```

   ACE structure (`A:flags:who:permissions`):

   - `A` — ACE type: Allow.
   - Inheritance flags:
     - `f` (file-inherit) — inherit to created files.
     - `d` (directory-inherit) — inherit to created subdirectories.
     - `i` (inherit-only) — apply only to descendants, not to the directory itself.
   - `who` — who the permissions are granted to:
     - `1871` — UID of user `ydb`.
     - `OWNER@` — object owner.
   - Permissions:
     - `r` — read-data / list-directory.
     - `w` — write-data / create-file.
     - `a` — append-data / create-subdirectory.
     - `t` — read-attributes.
     - `D` — delete-child.

   This command sets:

   - Permissions for user `ydb` on the directory itself.
   - Inherited permissions for files and subdirectories (`fi` , `di`), which will apply only to descendants.
   - Owner permissions (`OWNER@`) with inheritance to files and directories.

Verification

Check the configured ACLs:

```
nfs4_getfacl /mnt/ydb-backup
```

Create a test file as user `ydb`:

```
sudo -u ydb touch /mnt/ydb-backup/test-file
```

Verify that the file inherited the ACL:

```
nfs4_getfacl /mnt/ydb-backup/test-file
```

On all client nodes, check access:

```
sudo -u ydb ls /mnt/ydb-backup/test-file
```

After verification, delete the test file:

```
sudo -u ydb rm /mnt/ydb-backup/test-file
```

Feature flags

To use export and import via NFS, you need to enable the `enable_fs_backups` feature flag in the cluster configuration. By default, this flag is **disabled**.

Performance configuration

Two mechanisms affect the performance of export and import operations via NFS: the **resource broker** (manages the parallelism of scan and data load tasks) and the **actor system IO pool** (handles file write and read operations).

Resource broker

Export and import via NFS use the same resource broker queues as S3: `queue_backup` for export and `queue_restore` for import. Increasing the limits speeds up operations but increases CPU load. Default values and configuration method are described in the resource_broker_config section.

Actor system IO pool

File operations (write during export, read during import) are performed by actors in the pool of the actor system. By default, this pool contains `1` thread. Since file system operations are blocking, the `IO` pool can become a bottleneck during intensive export or import.

To increase the number of `threads` in the IO pool, change the parameter in the actor system configuration:

```
actor_system_config:
  executor:
  - name: IO
    threads: 3
    time_per_mailbox_micro_secs: 100
    type: IO
```

Tip

Recommendations for choosing the number of IO pool threads when using NFS export/import:

- Start with `2` threads and monitor the load.
- Keep in mind that the IO pool handles not only NFS operations but also other blocking tasks (e.g., writing logs).
- Monitor the pool load via the YDB UI.

Logging

The progress of export and import operations is reflected in the YDB server logs. The main logging components are:

- `EXPORT` — logging of the export process on SchemeShard: creating directories, copying tables, starting backup tasks, loading the schema.
- `IMPORT` — logging of the import process on SchemeShard: reading metadata, creating tables, restoring data.
- `FS_WRAPPER` — logging of file operations on each node. This component records I/O errors (file unavailability, insufficient space, locking errors), which helps identify issues with NFS mounting on specific nodes.

When diagnosing issues, it is recommended to:

- Check the operation status via `ydb operation get`.
- Search the server logs for entries with components `EXPORT` or `IMPORT` and the operation ID.
- For file write or read errors, search the logs for entries with component `FS_WRAPPER` — they contain the file path and error description, which helps determine on which host and with which file the problem occurred.
- Verify that the NFS directory is accessible and has enough free space on **all** cluster nodes.

Limitations

- Export and import via NFS support the same object types as export to S3. A detailed list of supported objects is provided in the export command documentation.
- Import always creates objects from scratch — you cannot import into existing tables.
- The `--list` parameter for listing export objects is available only for import from S3 in the current version.
- The operation is not supported on the Windows platform.

Recipes for streaming queries

This section contains recipes for working with streaming queries.

Contents:

- Quick start: reading and writing to topics
- Debug reads from a topic

Quick start: reading and writing to topics

In this guide, you will create your first streaming query.

The query will:

- read events from the input topic
- filter only errors
- count the number of errors per server over 10 minutes
- write the result to the output topic.

Events arrive in JSON format with fields: time, logging level, and server name.

You will perform the following steps:

- Create topics;
- Create the streaming query;
- Check query state;
- Produce sample input;
- Read the output topic;
- Delete the streaming query.

Prerequisites

To run the examples, you will need:

- A running YDB database — see quick start;
- The `enable_streaming_queries` feature flag enabled.

Docker

```
docker run -d --rm --name ydb-local -h localhost \
  --platform linux/amd64 \
  -p 2135:2135 -p 2136:2136 -p 8765:8765 -p 9092:9092 \
  -v $(pwd)/ydb_certs:/ydb_certs \
  -e GRPC_TLS_PORT=2135 -e GRPC_PORT=2136 -e MON_PORT=8765 \
  -e YDB_FEATURE_FLAGS=enable_streaming_queries \
  ydbplatform/local-ydb:25.4
```

local_ydb

```
./local_ydb deploy \
  --ydb-working-dir=/absolute/path/to/working/directory \
  --ydb-binary-path=/path/to/kikimr/driver \
  --enable-feature-flag=enable_streaming_queries
```

Note

The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Step 1. Creating topics

Create the input and output topics:

```
CREATE TOPIC input_topic;
CREATE TOPIC output_topic;
```

Verify that the topics are created:

```
./ydb --profile quickstart scheme ls
```

Step 2. Create the streaming query

Create a streaming query with CREATE STREAMING QUERY:

```
CREATE STREAMING QUERY query_example AS
DO BEGIN

$number_errors = SELECT
    Host,
    COUNT(*) AS ErrorCount,
    CAST(HOP_START() AS String) AS Ts  -- Start time of the window corresponding to the aggregation result
FROM
    input_topic
WITH (
    FORMAT = json_each_row,
    SCHEMA = (
        Time String NOT NULL,
        Level String NOT NULL,
        Host String NOT NULL
    )
)
WHERE
    Level = "error"
```

```
GROUP BY
    HOP(CAST(Time AS Timestamp), "PT600S", "PT600S", "PT0S"),  -- Number of errors on non-overlapping windows
of 10 minutes length
    Host;

INSERT INTO
    output_topic
SELECT
    ToBytes(Unwrap(Yson::SerializeJson(Yson::From(TableRow()))))  -- Serialization of all columns into JSON
FROM
    $number_errors;

END DO
```

More details:

- `GROUP BY HOP` aggregation and `HOP_START` — GROUP BY ... HOP.
- Writing to a topic — Formats for writing data.
- JSON serialization: TableRow, Yson::From, Yson::SerializeJson, Unwrap, ToBytes.

Step 3. Check query state

Check query state in the streaming_queries system table:

```
SELECT
    Path,
    Status,
    Issues,
    Run
FROM
    `.sys/streaming_queries`
```

Make sure that the `Status` field has the value `RUNNING`. Otherwise, check the `Issues` field.

If the query is in `SUSPENDED` status or the `Issues` field contains errors, see the error diagnostics section.

Step 4. Produce sample input

Write test messages to the topic using the YDB CLI:

```
echo '{"Time": "2025-01-01T00:00:00.000000Z", "Level": "error", "Host": "host-1"}' | ./ydb --profile quicksta
rt topic write input_topic
echo '{"Time": "2025-01-01T00:04:00.000000Z", "Level": "error", "Host": "host-2"}' | ./ydb --profile quicksta
rt topic write input_topic
echo '{"Time": "2025-01-01T00:08:00.000000Z", "Level": "error", "Host": "host-1"}' | ./ydb --profile quicksta
rt topic write input_topic
echo '{"Time": "2025-01-01T00:12:00.000000Z", "Level": "error", "Host": "host-2"}' | ./ydb --profile quicksta
rt topic write input_topic
echo '{"Time": "2025-01-01T00:12:00.000000Z", "Level": "error", "Host": "host-1"}' | ./ydb --profile quicksta
rt topic write input_topic
```

Results appear in the output topic after the 10-minute aggregation window closes.

Step 5. Read the output topic

Read data from the output topic:

```
./ydb --profile quickstart topic read output_topic --partition-ids 0 --start-offset 0 --limit 10 --format new
line-delimited
```

Expected result:

```
{"ErrorCount":1,"Host":"host-2","Ts":"2025-01-01T00:00:00Z"}
{"ErrorCount":2,"Host":"host-1","Ts":"2025-01-01T00:00:00Z"}
```

Step 6. Delete the query

Delete the query with DROP STREAMING QUERY:

```
DROP STREAMING QUERY query_example;
```

What's next

- Explore the data formats supported in streaming queries.
- Learn how to enrich data with a reference from a local table or from S3.
- Learn how to write results to tables.

See also

- Streaming queries
- Data formats for reading/writing from topics.

Debug reads from a topic

When developing streaming queries, it is often useful to inspect what arrives in a topic without creating a full streaming query. Run a regular `SELECT` with `STREAMING = "TRUE"`.

> **Warning**
>
> For debugging and inspection only. For production, create streaming queries with CREATE STREAMING QUERY.

> **Note**
>
> In the examples, `ydb_source` is a pre-created external data source, and `topic_name` / `input_topic` are topics available through it.

Raw reads

Simplest option — read messages in `raw` format without parsing:

```
SELECT
    Data
FROM
    ydb_source.topic_name
WITH (
    FORMAT = raw,
    SCHEMA = (
        Data String
    ),
    STREAMING = "TRUE"
)
LIMIT 1
```

`LIMIT` is required; without it the query never completes because it waits for new messages indefinitely.

JSON parsing

If the topic stores JSON, parse fields directly:

```
SELECT
    *
FROM
    ydb_source.topic_name
WITH (
    FORMAT = json_each_row,
    SCHEMA = (
        Time String NOT NULL,
        Level String NOT NULL,
        Host String NOT NULL
    ),
    STREAMING = "TRUE"
)
LIMIT 5
```

See also

- Streaming queries
- Data formats for reading/writing from topics — supported data formats
- Streaming read from a topic — `STREAMING = "TRUE"` in the YQL reference

Troubleshooting

This section of the YDB documentation provides guidance on troubleshooting issues related to YDB databases and the applications that interact with them.

- Troubleshooting performance issues

Troubleshooting performance issues

Addressing database performance issues often requires a holistic approach, which includes optimizing queries, properly configuring hardware resources, and ensuring that both the database and the application are well-designed. Regular monitoring and maintenance are essential for proactively identifying and resolving these issues.

Tools to troubleshoot performance issues

Troubleshooting performance issues in YDB involves the following tools:

- YDB metrics

  Diagnostic steps for most performance issues involve analyzing Grafana dashboards that use YDB metrics collected by Prometheus For information about how to set up Grafana and Prometheus, see Setting Up YDB Cluster Monitoring.
- YDB logs
- Tracing
- YDB CLI
- YDB UI
- Query plans
- Parameterized queries and recompilation
- Third-party observability tools

Classification of YDB performance issues

Database performance issues can be classified into several categories based on their nature. This documentation section provides a high-level overview of these categories, starting with the lowest layers of the system and going all the way to the client. Below is a separate section for the actual performance troubleshooting instructions.

Hardware infrastructure issues

- **Network issues**. Network congestion in data centers and especially between data centers can significantly affect YDB performance.
- **Data center outages**: Disruptions in data center operations that can cause service or data unavailability. To address this concern, YDB cluster can be configured to span three data centers or availability zones, but the performance aspect needs to be taken into account too.
- **Data center maintenance and drills**. Planned maintenance or drills, exercises conducted to prepare personnel for potential emergencies or outages, can also affect query performance. Depending on the maintenance scope or drill scenario, some YDB servers might become unavailable, which leads to the same impact as an outage.
- **Server hardware issues**. Malfunctioning CPU, memory modules, and network cards, until replaced, significantly impact database performance or lead to the unavailability of the affected server.

Insufficient resource issues

These issues refer to situations when the workload demands more physical resources — such as CPU, memory, disk space, and network bandwidth — than allocated to a database. In some cases, suboptimal allocation of resources, for example misconfigured control groups (cgroups) or actor system pools, may also result in insufficient resources for YDB and increase query latencies even though physical hardware resources are still available on the database server.

- **CPU bottlenecks**. High CPU usage can result in slow query processing and increased response times. When CPU resources are limited, the database may struggle to handle complex queries or large transaction loads.
- **Insufficient disk space**. A lack of available disk space can prevent the database from storing new data, resulting in the database becoming read-only. This might also cause slowdowns as the system tries to reclaim disk space by compacting existing data more aggressively.
- **Insufficient memory (RAM)**. Queries require memory to temporarily store various intermediate data during execution. A lack of available memory can negatively impact database performance in multiple ways.
- **Insufficient disk I/O bandwidth**. A high rate of read/write operations can overwhelm the disk subsystem, causing increased data access latencies. When the distributed storage cannot read or write data quickly enough, queries requiring disk access will take longer to execute.

Operating system issues

- **System clock drift**. If the system clocks on the YDB servers start to drift apart, it will lead to increased distributed transaction latencies. In severe cases, YDB might even refuse to process distributed transactions and return errors.
- Other processes running on the same servers or virtual machines as YDB, such as antiviruses, observability agents, etc.
- Kernel misconfiguration.

YDB-related issues

- **Updating YDB versions**. There are two main related aspects: restarting all nodes within a relatively short timeframe, and the behavioral differences between versions.
- Actor system pools misconfiguration.

Schema design issues

- **Overloaded shards**. Data shards serving row-oriented tables may become overloaded for several reasons. Such overload leads to increased latencies for the transactions processed by the affected data shards.
- **Excessive tablet splits and merges**. YDB supports automatic splitting and merging of data shards, which allows it to seamlessly adapt to changes in workloads. However, these operations are not free and might have a short-term negative impact on query latencies.

Client application-related issues

- **Query design issues**. Inefficiently designed database queries may execute slower than expected.
- **SDK usage issues**. Issues related to improper or suboptimal use of the SDK.

Instructions

To troubleshoot YDB performance issues, treat each potential cause as a hypothesis. Systematically review the list of hypotheses and verify whether they apply to your situation. The documentation for each cause provides a description, guidance on how to check diagnostics, and recommendations on what to do if the hypothesis is confirmed.

If any known changes occurred in the system around the time the performance issues first appeared, investigate those first. Otherwise, follow this recommended order for evaluating potential root causes. This order is loosely based on the descending frequency of their occurrence on large production YDB clusters.

1. Overloaded shards and errors
2. Excessive tablet splits and merges
3. Frequent tablet moves between nodes
4. Insufficient hardware resources:
    - Disk I/O bandwidth
    - Disk space
    - Insufficient CPU
    - Insufficient memory
5. Hardware issues and data center outages
6. Network issues
7. Rolling restart
8. System clock drift
9. Transaction lock invalidation
10. Data center maintenance and drills

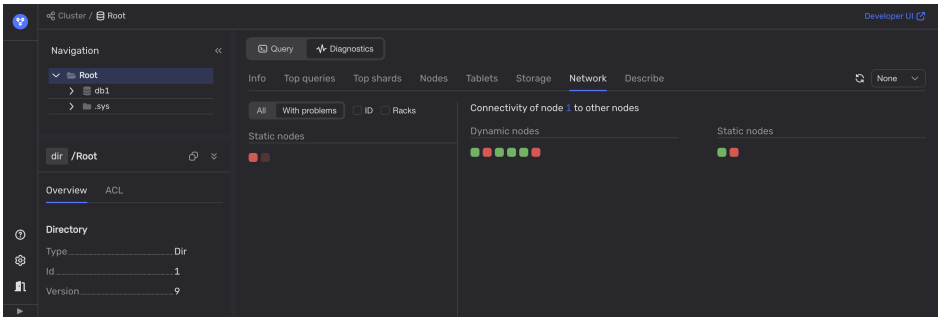
Network issues

Network performance issues, such as limited bandwidth, packet loss, and connection instability, can severely impact database performance by slowing query response times and leading to retriable errors like timeouts.

Diagnostics

To diagnose network issues, use the healthcheck in the YDB UI:

1. Open the YDB UI:
   1.1. Navigate to the **Databases** tab and click on the desired database.
   1.2. In the **Navigation** tab, confirm the required database is selected.
   1.3. Switch to the **Diagnostics** tab.
   1.4. Under the **Network** tab, apply the **With problems** filter.



2. Use available third-party tools to monitor network performance metrics such as latency, jitter, packet loss, throughput, and others.

Recommendations

Contact the responsible party for the network infrastructure the YDB cluster uses. If you are part of a larger organization, this could be an in-house network operations team. Otherwise, contact the cloud service or hosting provider's support service.

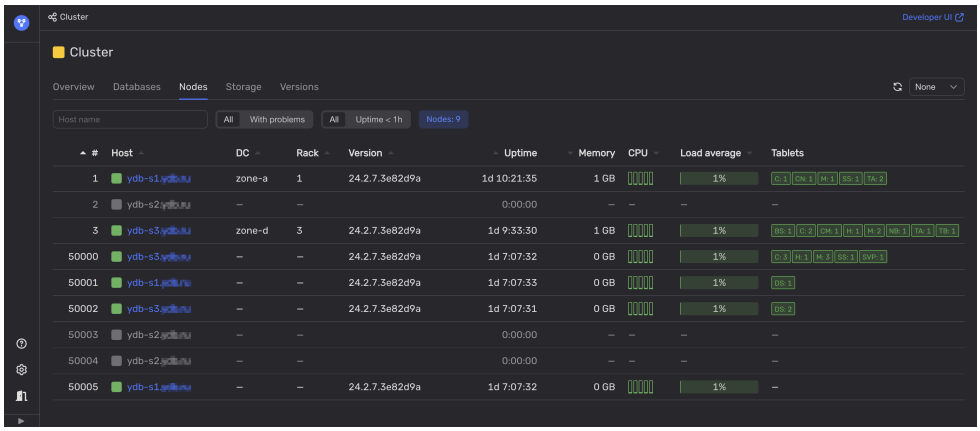
Data center outages

Data center outages are disruptions in data center operations that could cause service or data unavailability, but YDB has means to avoid it. Various factors, such as power failures, natural disasters, or cyberattacks, may cause these outages. A common fault-tolerant setup for YDB spans three data centers or availability zones (AZs). In this case, YDB can maintain uninterrupted operation even if one data center and a server rack in another are lost. However, it will initiate the relocation of tablets from the offline AZ to the remaining online nodes, temporarily leading to higher query latencies.

Diagnostics

To determine if one of the data centers of the YDB cluster is not available, follow these steps:

1. Open YDB UI.
2. On the **Nodes** tab, analyze the health indicators in the **Host** and **DC** columns.



   If all of the nodes in one of the data centers (DC) are not available, this data center is most likely offline.

   If not, review the **Rack** column to check if all YDB nodes are unavailable in one or more server racks. This could indicate that these racks are offline, which could be treated as a partial data center outage.

Recommendations

Contact the responsible party for the affected data center to resolve the underlying issue. If you are part of a larger organization, this could be an in-house team managing low-level infrastructure. Otherwise, contact the cloud service or hosting provider's support service. Meanwhile, check the data center's status page if it has one.

Additionally, consider potential data center outages in the capacity planning process. YDB nodes in each data center should have sufficient spare hardware resources to take over the full workload typically handled by any data center experiencing an outage.

Data center maintenance and drills

Planned maintenance or drills, exercises conducted to prepare personnel for potential emergencies or outages, can also affect query performance. Depending on the maintenance scope or drill scenario, some YDB nodes might become unavailable, which leads to the same impact as an outage.

Diagnostics

Check the planned maintenance and drills schedules to see if their timelines match with observed performance issues, otherwise, check the datacenter outage recommendations.

Recommendations

Contact the person responsible for the current maintenance or drill to discuss whether the performance impact is severe enough for it to be finished/canceled early, if possible.

Hardware issues

Malfunctioning storage drives and network cards, until replaced, significantly impact database performance up to total unavailability of the affected server. CPU issues might lead to server failure and higher load on the remaining YDB nodes.

Diagnostics

Use the hardware monitoring tools that your operating system and data center provide to diagnose hardware issues.

You can also use the **Healthcheck** in YDB UI to diagnose some hardware issues:

- **Storage issues**
  1. On the **Storage** tab, select the **Degraded** filter to list storage groups or nodes that contain degraded or failed storage.
  2. Check for any degradation in the storage system performance on the **Distributed Storage Overview** and **PDisk Device single disk** dashboards in Grafana.
- **Network issues**

  Refer to Network issues.

Recommendations

Contact the responsible party for the affected hardware to resolve the underlying issue. If you are part of a larger organization, this could be an in-house team managing low-level infrastructure. Otherwise, contact the cloud service or hosting provider's support service.

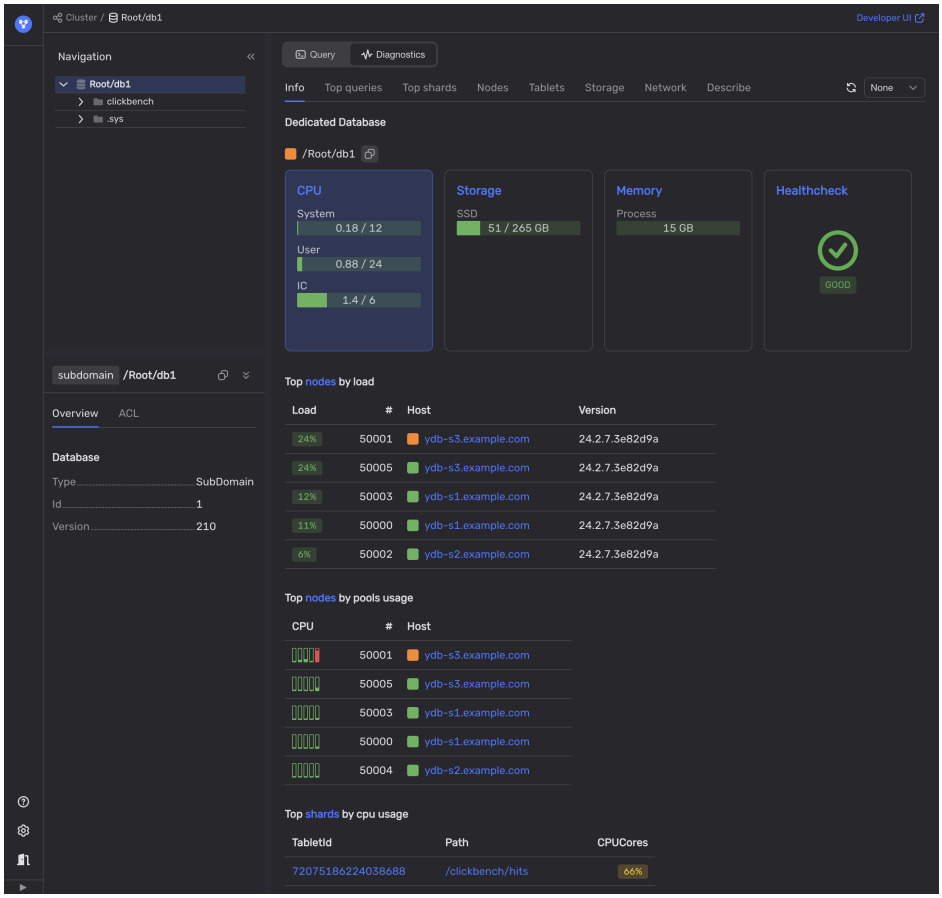
CPU bottleneck

High CPU usage can lead to slow query processing and increased response times. When CPU resources are constrained, the database may have difficulty handling complex queries or large transaction volumes.

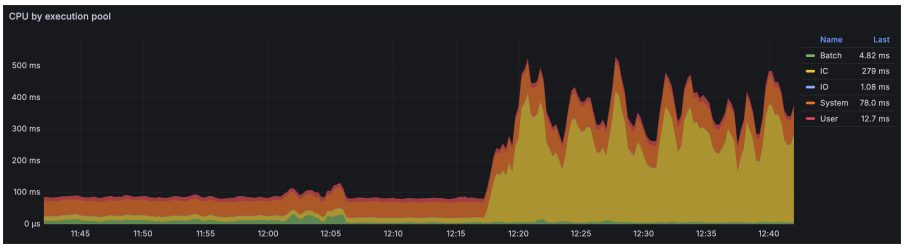
YDB nodes primarily consume CPU resources for running actors. On each node, actors are executed using multiple actor system pools. The resource consumption of each pool is measured separately which allows to identify what kind of activity changed its behavior.

Diagnostics

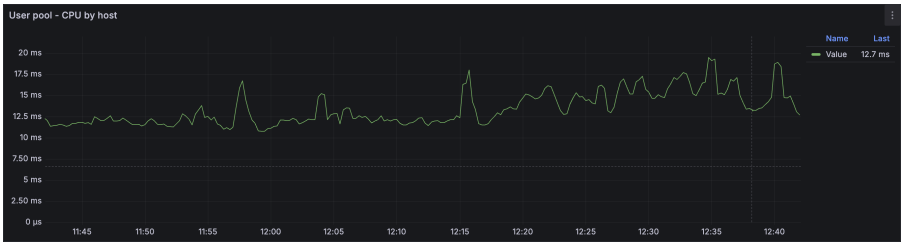
1. Use **Diagnostics** in the YDB UI to analyze CPU utilization in all pools:
   1.1. In the YDB UI, go to the **Databases** tab and click on the database.
   1.2. On the **Navigation** tab, ensure the required database is selected.
   1.3. Open the **Diagnostics** tab.
   1.4. On the **Info** tab, click the **CPU** button and see if any pools show high CPU usage.



2. Use Grafana charts to analyze CPU utilization in all pools:
   2.1. Open the **CPU** dashboard in Grafana.
   2.2. See if the following charts show any spikes:
   - **CPU by execution pool** chart

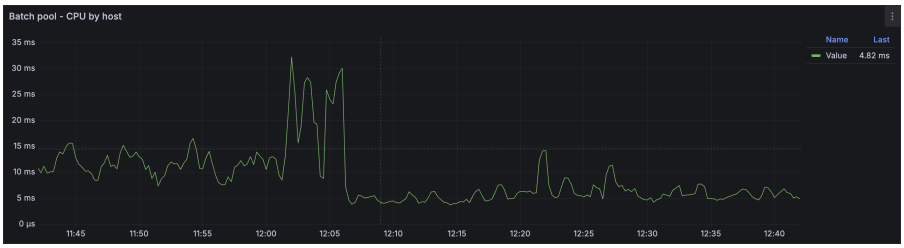


   - **User pool - CPU by host** chart



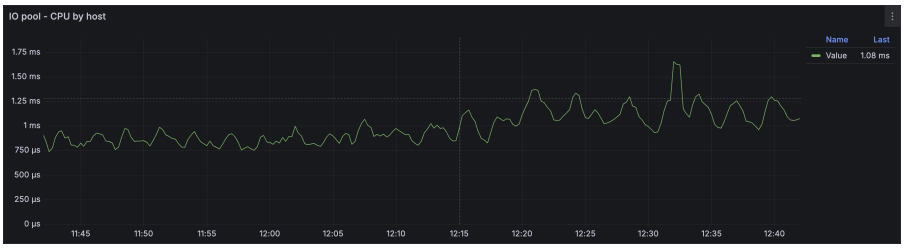
   - **System pool - CPU by host** chart

- **Batch pool - CPU by host** chart

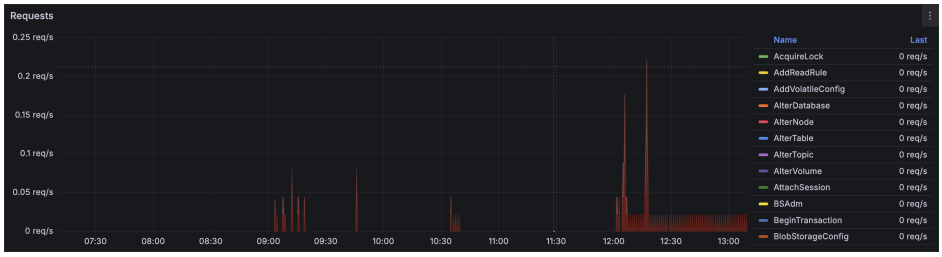


- **IC pool - CPU by host** chart

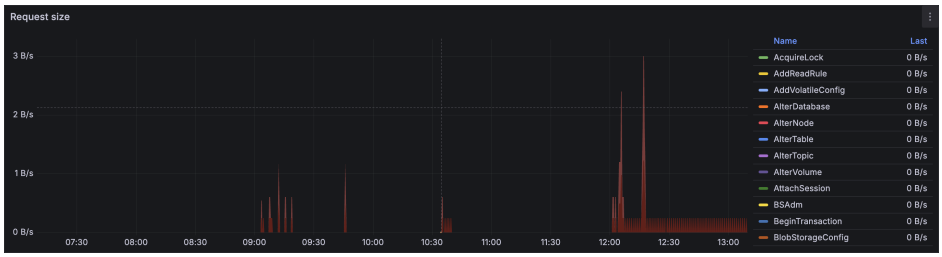
- **IO pool - CPU by host** chart



3. If the spike is in the user pool, analyze changes in the user load that might have caused the CPU bottleneck. See the following charts on the **DB overview** dashboard in Grafana:
   - **Requests** chart



   - **Request size** chart



   - **Response size** chart

   Also, see all of the charts in the **Operations** section of the **DataShard** dashboard.
4. If the spike is in the batch pool, check if there are any backups running.

Recommendation

Add additional database nodes to the cluster or allocate more CPU cores to the existing nodes. If that's not possible, consider distributing CPU cores between pools differently.

Insufficient memory (RAM)

If swap (paging of anonymous memory) is disabled on the server running YDB, insufficient memory activates another kernel feature called the OOM killer, which terminates the most memory-intensive processes (often the database itself). This feature also interacts with cgroups if multiple cgroups are configured.

If swap is enabled, insufficient memory may cause the database to rely heavily on disk I/O, which is significantly slower than accessing data directly from memory.

> **Warning**
>
> If YDB nodes are running on servers with swap enabled, disable it. YDB is a distributed system, so if a node restarts due to lack of memory, the client will simply connect to another node and continue accessing data as if nothing happened. Swap would allow the query to continue on the same node but with degraded performance from increased disk I/O, which is generally less desirable.

Even though the reasons and mechanics of performance degradation due to insufficient memory might differ, the symptoms of increased latencies during query execution and data retrieval are similar in all cases.

Additionally, which components within the YDB process consume memory may also be significant.

Diagnostics

1. Determine whether any YDB nodes recently restarted for unknown reasons. Exclude cases of YDB version upgrades and other planned maintenance. This could reveal nodes terminated by OOM killer and restarted by `systemd`.
   1.1. Open YDB UI.
   1.2. On the **Nodes** tab, look for nodes that have low uptime.
   1.3. Chose a recently restarted node and log in to the server hosting it. Run the `dmesg` command to check if the kernel has recently activated the OOM killer mechanism.

   Look for the lines like this:

   ```
   [ 2203.393223] oom-kill:constraint=CONSTRAINT_NONE,nodemask=(null),cpuset=user.slice,mems_allowed=0,
   global_oom,task_memcg=/user.slice/user-1000.slice/session-1.scope,task=ydb,pid=1332,uid=1000
   [ 2203.393263] Out of memory: Killed process 1332 (ydb) total-vm:14219904kB, anon-rss:1771156kB, fil
   e-rss:0kB, shmem-rss:0kB, UID:1000 pgtables:4736kB oom_score_adj:0
   ```

   Additionally, review the `ydbd` logs for relevant details.
2. Determine whether memory usage reached 100% of capacity.
   2.1. Open the **DB overview** dashboard in Grafana.
   2.2. Analyze the charts in the **Memory** section.
3. Determine whether the user load on YDB has increased. Analyze the following charts on the **DB overview** dashboard in Grafana:
   - **Requests** chart
   - **Request size** chart
   - **Response size** chart
4. Determine whether new releases or data access changes occurred in your applications working with YDB.

Recommendation

Consider the following solutions for addressing insufficient memory:

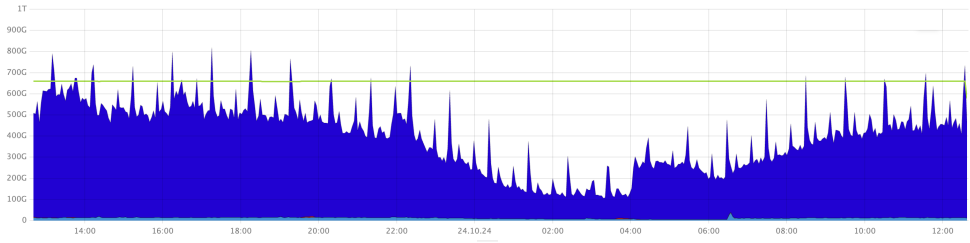
- If the load on YDB has increased due to new usage patterns or increased query rate, try optimizing the application to reduce the load on YDB or add more YDB nodes.
- If the load on YDB has not changed but nodes are still restarting, consider adding more YDB nodes or raising the hard memory limit for the nodes. For more information about memory management in YDB, see memory_controller_config.

I/O bandwidth

A high rate of read and write operations can overwhelm the disk subsystem, leading to increased data access latencies. When the system cannot read or write data quickly enough, queries that rely on disk access will experience delays.

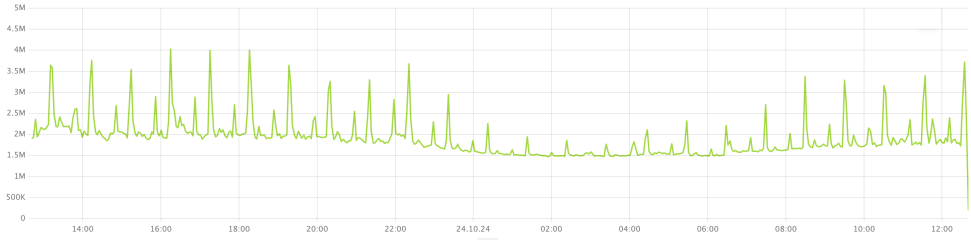
Diagnostics

1. Open the **Distributed Storage Overview** dashboard in Grafana.
2. On the **DiskTimeAvailable and total Cost relation** chart, see if the **Total Cost** spikes cross the **DiskTimeAvailable** level.



This chart shows the estimated total bandwidth capacity of the storage system in conventional units (green) and the total usage cost in conventional units (blue). When the total usage cost exceeds the total bandwidth capacity, the YDB storage system becomes overloaded, leading to increased latencies.

3. On the **Total burst duration** chart, check for any load spikes on the storage system. This chart displays microbursts of load on the storage system, measured in microseconds.



> **Note**
>
> This chart might show microbursts of the load that are not detected by the average usage cost in the **Cost and DiskTimeAvailable relation** chart.

Recommendations

Add more storage groups to the database.

In cases of high microburst rates, balancing the load across storage groups might help.

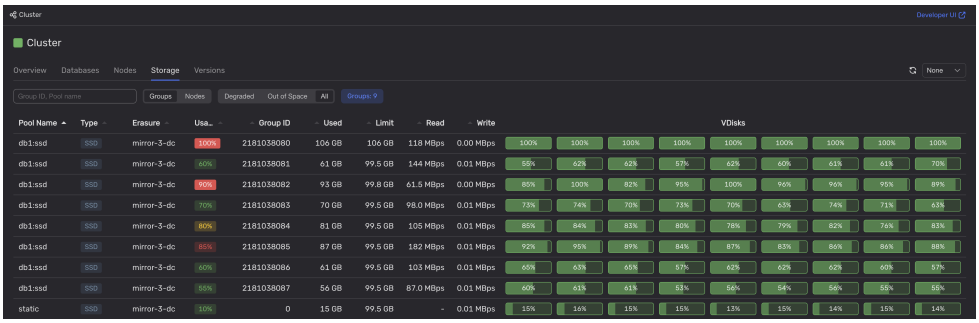
Disk space

A lack of available disk space can prevent the database from storing new data, resulting in the database becoming read-only. This can also cause slowdowns as the system tries to reclaim disk space by compacting existing data more aggressively.

Diagnostics

1. See if the **DB overview > Storage** charts in Grafana show any spikes.
2. In YDB UI, on the **Storage** tab, analyze the list of available storage groups and nodes and their disk usage.

> **Tip**
>
> Use the **Out of Space** filter to list only the storage groups with full disks.



> **Note**
>
> It is also recommended to use the Healthcheck API to get this information.

Recommendations

Add more storage groups to the database.

If the cluster doesn't have spare storage groups, configure them first. Add additional storage nodes, if necessary.

System clock drift

Synchronized clocks are critical for distributed databases. If system clocks on the YDB servers drift excessively, distributed transactions will experience increased latencies.

> **Alert**
>
> It is important to keep system clocks on the YDB servers in sync, to avoid high latencies.

Symptoms and thresholds

Typical symptoms of clock skew across nodes:

- **Slower distributed transactions** — end-to-end latency can increase roughly by the amount of clock skew (a node with a "faster" clock waits for coordination with nodes that lag behind).
- **Timeouts and deadline issues** — skew can cause spurious client/server timeouts and deadline failures.
- **Authentication issues** — large skew can break token validity windows and related checks.

When monitoring clock skew between nodes, use these thresholds (typical levels for cluster observability):

- **Less than 1 millisecond** — normal level for routine cluster operation.
- **1—5 milliseconds** — a range where it already makes sense to look for the cause of desynchronization and increase observability.
- **More than 5 milliseconds** — critical level: urgent diagnosis and time alignment are required.
- **25 milliseconds and above** — growing performance degradation; users see widespread latency issues and timeouts on distributed transactions.
- **30 seconds** — distributed transactions stop working (see below).

If the system clocks of the nodes running the coordinator tablets differ noticeably from one another, transaction latencies increase by the time difference between the fastest and slowest system clocks. This occurs because a transaction planned on a node with a faster system clock can only be executed once the node with the slowest clocks reaches the same time.

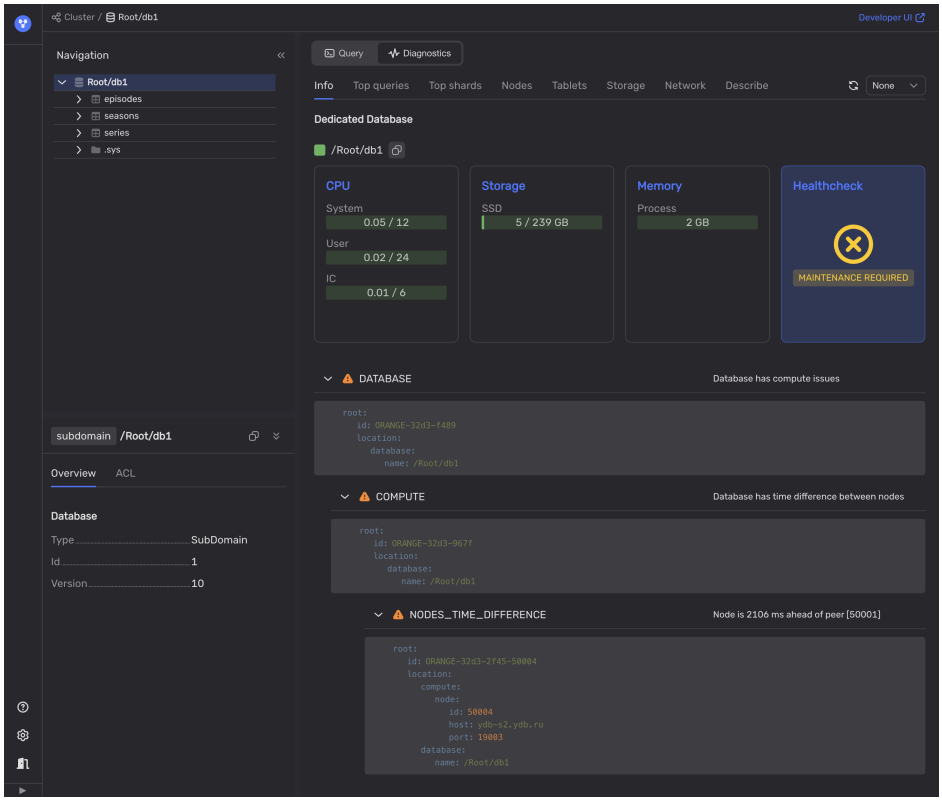
Furthermore, if the system clock drift exceeds 30 seconds, YDB will refuse to process distributed transactions. Before coordinators start planning a transaction, affected data shards determine an acceptable range of timestamps for the transaction. The start of this range is the current system time of the mediator tablet, and the end is defined by the 30-second planning timeout. If the coordinator's system clock falls outside this time range, it cannot plan a distributed transaction, resulting in errors for such queries.

Diagnostics

To diagnose system clock drift on YDB servers, use the following methods:

1. Use **Healthcheck** in the YDB UI:
   1.1. In the YDB UI, go to the **Databases** tab and click on the database.
   1.2. On the **Navigation** tab, ensure the required database is selected.
   1.3. Open the **Diagnostics** tab.
   1.4. On the **Info** tab, click the **Healthcheck** button.

   If the **Healthcheck** button displays a `MAINTENANCE REQUIRED` status, the YDB cluster might be experiencing issues, such as system clock drift. Any identified issues will be listed in the **DATABASE** section below the **Healthcheck** button.
   1.5. To see the diagnosed problems, expand the **DATABASE** section.



System clock drift problems are listed under `NODES_TIME_DIFFERENCE`.

> **Note**
>
> For more information, see Health Check API

2. Open the Interconnect overview page in the YDB UI. Interconnect metrics (including indicators related to clock skew across nodes) help assess the scope of the issue alongside the overall picture of connectivity latency and errors.

> **Note**
>
> An increase in system clock skew according to interconnect monitoring (as shown in the YDB UI and collected via cluster metrics) may be caused by actual clock drift, exhaustion of resources in the interconnect CPU pool, or network equipment overload.

3. Use such tools as `pssh` or `ansible` to run the command (for example, `date +%s%N`) on all YDB nodes to display the system clock value.

> **Warning**
>
> Network delays between the host that runs `pssh` or `ansible` and YDB hosts will influence the results.

If you use time synchronization utilities, you can also request their status instead of requesting the current timestamps. For example:

```
chronyc sources -v
```

It is also useful to review cluster monitoring metrics (if you collect them): transaction latency and gRPC success rates may degrade together with interconnect indicators when skew is present (in particular, the `interconnect.ClockSkewMicrosec` metric).

Recommendations

1. Manually synchronize the system clocks of servers running YDB nodes. For instance, use `pssh` or `ansible` to run the clock sync command across all nodes.
2. Ensure that system clocks on all YDB servers are regularly synchronized using `ntpd`, `chrony`, or a similar tool. Use the same logical time source for every server in the cluster (the same set of NTP servers or the same NTP hierarchy) and configure multiple independent upstream NTP sources.

> **Warning**
>
> Using the **systemd-timesyncd** service is not recommended, because this type of NTP client does not provide sufficient accuracy for system time synchronization. Use other tools instead, such as `chrony` or `ntpd`.

NTP configuration examples

Below are example settings for `chrony`. Replace the server list with values appropriate for your environment and security policy.

chrony (`/etc/chrony/chrony.conf`):

```
server ntp1.example.net iburst
server ntp2.example.net iburst
server ntp3.example.net iburst
server ntp4.example.net iburst
maxslewrate 10000
local stratum 10
```

Monitoring and alerting

Configure alerts on clock skew between nodes and related symptoms (including YDB UI data and external monitoring). See Symptoms and thresholds for threshold guidance.

Emergency recovery

If skew becomes critical, you may need forced synchronization (perform this deliberately and according to your cluster's agreed procedure):

```
# Example for chrony: stop the service and perform a one-shot sync
sudo systemctl stop chrony
sudo chronyd -q 'server ntp1.example.net iburst'
sudo systemctl start chrony
```

Suggested sequence on a cluster:

1. Verify NTP servers are correct and reachable.
2. Align time across nodes (without divergent per-node configuration).
3. Confirm convergence via Healthcheck and monitoring.
4. Identify the root cause of the synchronization failure (network, DNS, blocked NTP egress, etc.).

Prevention

- Periodically review metrics and Healthcheck reports for growing skew.
- Validate NTP fault tolerance (multiple NTP servers; UDP/123 reachability if applicable).
- Keep the configuration of the NTP client you use consistent across all cluster nodes.

Rolling restart

YDB clusters can be updated without downtime, which is possible because YDB normally has redundant components and supports rolling restart procedure. To ensure continuous data availability, YDB includes Cluster Management System (CMS) that tracks all outages and nodes taken offline for maintenance, such as restarts. CMS halts new maintenance requests if they might risk data availability.

However, even if data is always available, the restart of all nodes in a relatively short period of time might have a noticeable impact on overall performance. Each tablet running on a restarted node is relaunched on a different node. Moving a tablet between nodes takes time and may affect latencies of queries involving it. See recommendations for rolling restart.

Furthermore, a new YDB version may handle queries differently. While performance generally improves with each update, certain corner cases may occasionally end up with degraded performance. See recommendations for new version performance.

Diagnostics

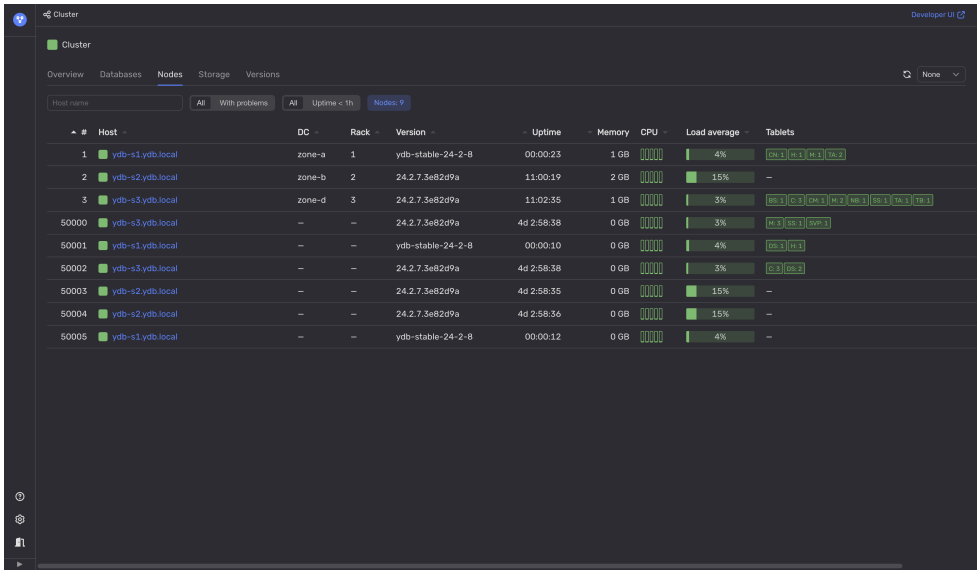
Warning

Diagnostics of YDB rolling restarts and updates relies only on secondary symptoms. To be absolutely sure, contact your database administrator.

To check if the YDB cluster is currently being updated:

1. Open YDB UI.
2. On the **Nodes** tab, see if YDB versions of the nodes differ.

   Also, see if the nodes with the higher YDB version have the lower uptime value.



Alert

Low uptime value of a YDB node might also indicate other problems. For example, see Insufficient memory (RAM).

Recommendations

For rolling restart

If the ongoing YDB cluster rolling restart significantly impacts applications to the point where they can no longer meet their latency requirements, consider slowing down the restart process:

1. If nodes are restarted in batches, reduce the batch size, up to one node at a time.
2. Space out in time the restarts for each data center and/or server rack.
3. Inject artificial pauses between restarts.

For new version performance

The goal is to detect any negative performance impacts from the new YDB version on specific queries in your particular workload as early as possible:

1. Review the YDB server changelog for any performance-related notes relevant to your workload.
2. Use a dedicated pre-production and/or testing YDB cluster to run a workload that closely mirrors your production workload. Always deploy the new YDB version to these clusters first. Monitor both client-side latencies and server-side metrics to identify any potential performance issues.
3. Implement canary deployment by updating only one node initially to observe any changes in its behavior. If everything appears stable, gradually expand the update to more nodes, such as an entire server rack or data center, and repeat checks for anomalies. If any issues arise, immediately roll back to the previous version and attempt to reproduce the issue in a non-production environment.

Report any identified performance issues on YDB's GitHub. Provide context and all the details that could help reproduce it.

Frequent tablet moves between nodes

YDB automatically balances the load by moving tablets from overloaded nodes to other nodes. This process is managed by Hive. When Hive moves tablets, queries affecting those tablets might experience increased latencies while they wait for the tablet to get initialized on the new node.

YDB considers usage of the following hardware resources for balancing nodes:

- CPU
- Memory
- Network
- Counter[14]

Autobalancing occurs in the following cases:

- **Imbalanced Hardware Resource Usage**

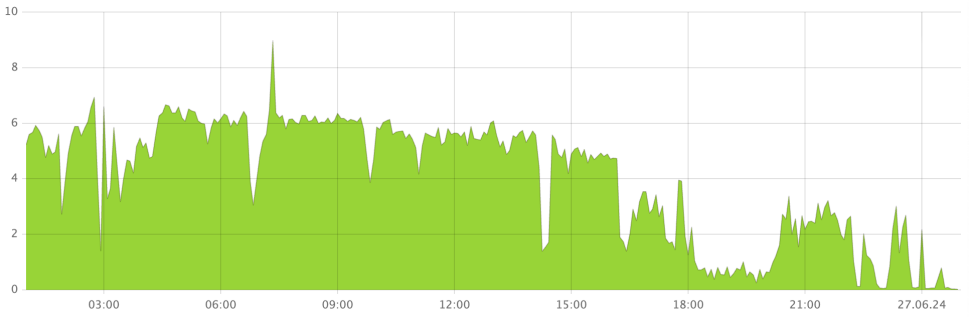
  YDB uses the Scatter metric to evaluate the balance of hardware resource usage. For more details on the Scatter metric's calculation logic and balancing triggers, see the Resource Usage Imbalance section.
- **Overloaded nodes (CPU and memory usage)**

  Hive initiates balancing in case of a significant load asymmetry (for example, > 90% on one node and < 70% on another). Learn more here: Node Overload.
- **Uneven distribution of database objects**

  For tablets with no explicit resource consumption, Hive uses a fake **Counter** resource to ensure their even distribution. Balancing is triggered if this distribution becomes skewed. Learn more: Even Distribution for a Specific Object.

Diagnostics

1. See if the **Tablets moved by Hive** chart in the **DB status** Grafana dashboard shows any spikes.




```
This chart displays the time-series data for the number of tablets moved per second.
```

2. See the Hive balancer stats.
   2.1. Open YDB UI.
   2.2. Click **Developer UI** in the upper right corner of the YDB UI.
   2.3. In the **Developer UI**, navigate to **Tablets > Hive > App**.

   See the balancer stats in the upper right corner.

Balancer	Runs	Moves	Last run	Last moves	Progress
Counter	0	0			
CPU	14	14	2024-10-18T12:05:44.342466Z	1	
Memory	0	0			
Network	0	0			
Emergency	0	0			
Spread	0	0			
Scatter	0	0			
Manual	0	0			
Storage	0	0			

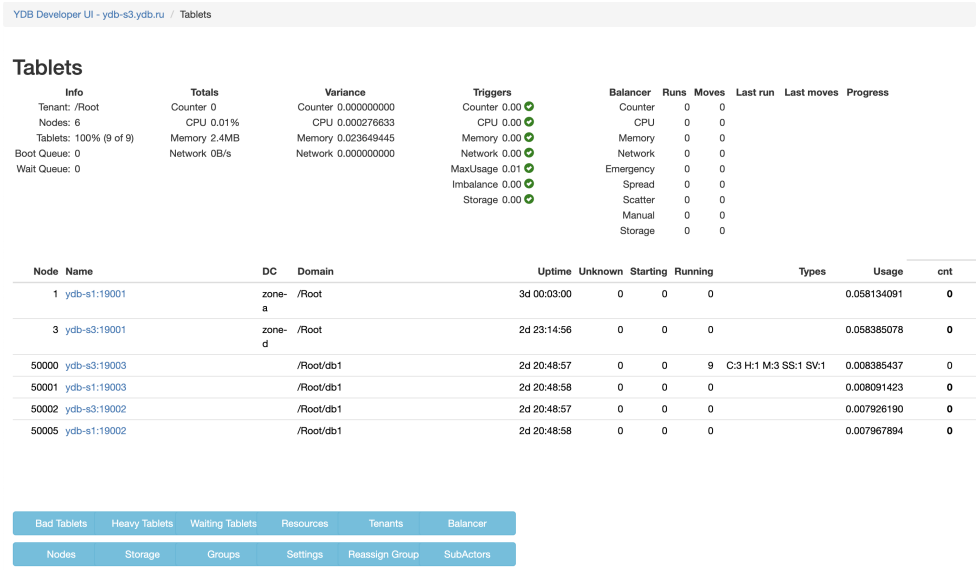
   2.4. Additionally, to see the recently moved tablets, click the **Balancer** button.

   The **Balancer** window will appear. The list of recently moved tablets is displayed in the **Latest tablet moves** section.

Recommendations

Adjust Hive balancer settings:

1. Open YDB UI.
2. Click **Developer UI** in the upper right corner of the YDB UI.
3. In the **Developer UI**, navigate to **Tablets > Hive > App**.



4. Click **Settings**.
5. To reduce the likelihood of overly frequent balancing, increase the following Hive balancer thresholds:

Parameter	Description	Default value
MinCounterScatterToBalance	The threshold for the counter scatter value. When this value is reached, Hive starts balancing the load.	0.02
MinCPUScatterToBalance	The threshold for the CPU scatter value. When this value is reached, Hive starts balancing the load.	0.5
MinMemoryScatterToBalance	The threshold for the memory scatter value. When this value is reached, Hive starts balancing the load.	0.5
MinNetworkScatterToBalance	The threshold for the network scatter value. When this value is reached, Hive starts balancing the load.	0.5
MaxNodeUsageToKick	The threshold for the node resource usage. When this value is reached, Hive starts emergency balancing.	0.9
ObjectImbalanceToBalance	The threshold for the database object imbalance metric.	0.02

Note

These parameters use relative values, where 1.0 represents 100% and effectively disables balancing. If the total hardware resource value can exceed 100%, adjust the ratio accordingly.

[14] Counter - a fake resource representing a count of tablets of a certain type on a node, used to ensure such tablets are distributed evenly across nodes.

Overloaded shards

Data shards serving row-oriented tables may become overloaded for the following reasons:

- A table is created without the AUTO_PARTITIONING_BY_LOAD clause.

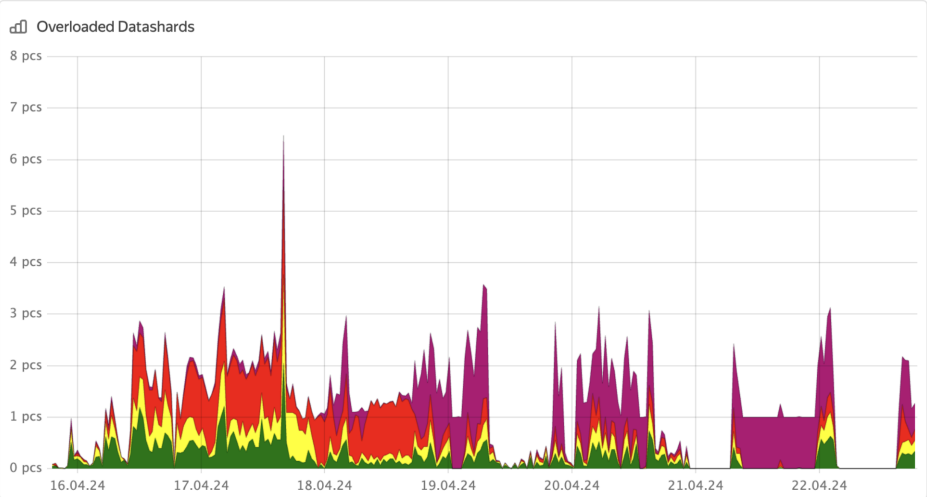
  In this case, YDB does not split overloaded shards.

  Data shards are single-threaded and process queries sequentially. Each data shard can accept up to 10,000 operations. Accepted queries wait for their turn to be executed. So the longer the queue, the higher the latency.

  If a data shard already has 10000 operations in its queue, new queries will return an "overloaded" error. Retry such queries using a randomized exponential back-off strategy. For more information, see Overloaded errors.
- A table was created with the AUTO_PARTITIONING_MAX_PARTITIONS_COUNT setting and has already reached its partition limit.
- An inefficient primary key that causes an imbalance in the distribution of queries across shards. A typical example is ingestion with a monotonically increasing primary key, which may lead to the overloaded "last" partition. For example, this could occur with an autoincrementing primary key using the serial data type.

Diagnostics

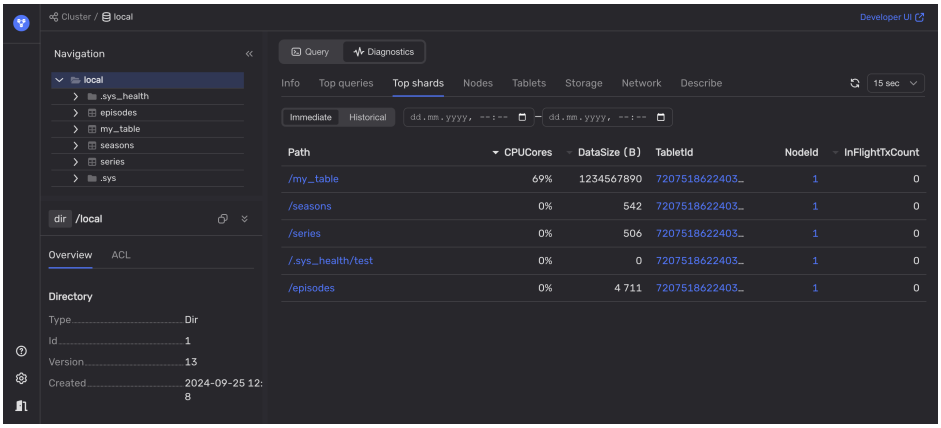
1. Use the YDB UI or Grafana to see if the YDB nodes are overloaded:
   - In the **DB overview** Grafana dashboard, analyze the **Overloaded shard count** chart.



The chart indicates whether the YDB cluster has overloaded shards, but it does not specify which table's shards are overloaded.

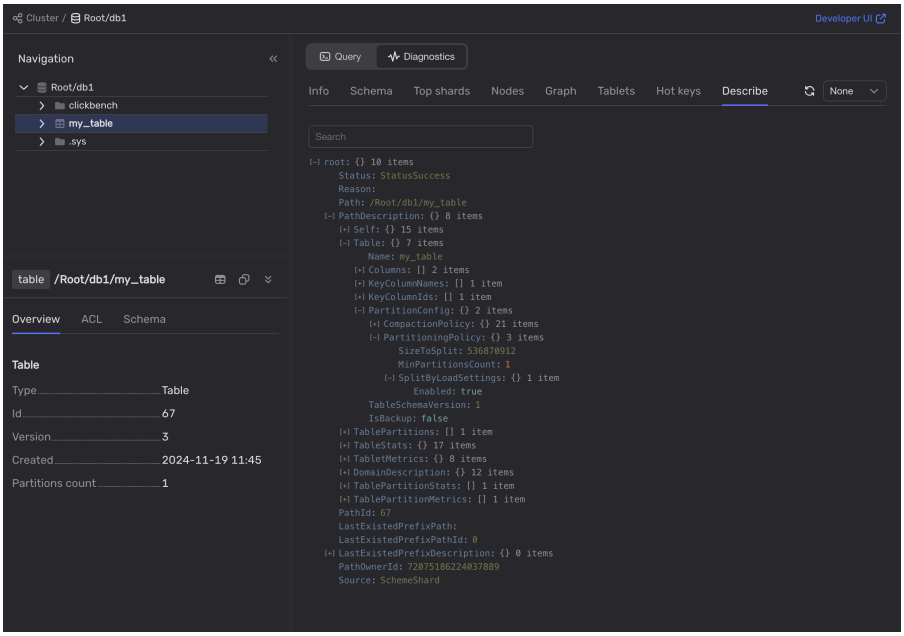
> **Tip**
>
> Use Grafana to set up alert notifications when YDB data shards get overloaded.

   - In the YDB UI:
     1. Go to the **Databases** tab and click on the database.
     2. On the **Navigation** tab, ensure the required database is selected.
     3. Open the **Diagnostics** tab.
     4. Open the **Top shards** tab.
     5. In the **Immediate** and **Historical** tabs, sort the shards by the **CPUCores** column and analyze the information.



Additionally, the information about overloaded shards is provided as a system table. For more information, see History of overloaded partitions.

2. To pinpoint the schema issue, use the YDB UI or YDB CLI:
   - In the YDB UI:
     1. On the **Databases** tab, click on the database.
     2. On the **Navigation** tab, select the required table.
     3. Open the **Diagnostics** tab.
     4. On the **Describe** tab, navigate to `root > PathDescription > Table > PartitionConfig > PartitioningPolicy`.



5. Analyze the **PartitioningPolicy** values:
   - `SizeToSplit`
   - `SplitByLoadSettings`
   - `MaxPartitionsCount`

   If the table does not have these options, see Recommendations for table configuration.

> **Note**
>
> You can also find this information on the **Diagnostics > Info** tab.

- In the YDB CLI:
  1. To retrieve information about the problematic table, run the following command:

     ```
     ydb scheme describe <table_name>
     ```

  2. In the command output, analyze the **Auto partitioning settings**:
     - `Partitioning by size`
     - `Partitioning by load`
     - `Max partitions count`

     If the table does not have these options, see Recommendations for table configuration.

3. Analyze whether primary key values increment monotonically:
   - Check the data type of the primary key column. `Serial` data types are used for autoincrementing values.
   - Check the application logic.
   - Calculate the difference between the minimum and maximum values of the primary key column. Then compare this value to the number of rows in a given table. If these values match, the primary key might be incrementing monotonically.

   If primary key values do increase monotonically, see Recommendations for the imbalanced primary key.

Recommendations

For table configuration

Consider the following solutions to address shard overload:

- If the problematic table is not partitioned by load, enable partitioning by load.

> **Tip**
>
> A table is not partitioned by load, if you see the `Partitioning by load: false` line on the **Diagnostics > Info** tab in the **YDB UI** or the `ydb scheme describe` command output.

- If the table has reached the maximum number of partitions, increase the partition limit.

> **Tip**
>
> To determine the number of partitions in the table, see the `PartCount` value on the **Diagnostics > Info** tab in the **YDB UI**.

Both operations can be performed by executing an `ALTER TABLE ... SET` query.

For the imbalanced primary key

Consider modifying the primary key to distribute the load evenly across table partitions. You cannot change the primary key of an existing table. To do that, you will have to create a new table with the modified primary key and then migrate the data to the new table.

> **Note**
>
> Also, consider changing your application logic for generating primary key values for new rows. For example, use hashes of values instead of values themselves.

Example

For a practical demonstration of how to follow these instructions, see Overloaded shard example.

Excessive tablet splits and merges

> **Warning**
>
> Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Each row-oriented table partition in YDB is processed by a data shard tablet. YDB supports automatic splitting and merging of data shards which allows it to seamlessly adapt to changes in workloads. However, these operations are not free and might have a short-term negative impact on query latencies.

When YDB splits a partition, it replaces the original partition with two new partitions covering the same range of primary keys. Now, two data shards process the range of primary keys that was previously handled by a single data shard, thereby adding more computing resources for the table.

By default, YDB splits a table partition when it reaches 2 GB in size. However, it's recommended to also enable partitioning by load, allowing YDB to split overloaded partitions even if they are smaller than 2 GB.

A scheme shard takes approximately 15 seconds to assess whether a data shard requires splitting. By default, the CPU usage threshold for splitting a data shard is set at 50%.

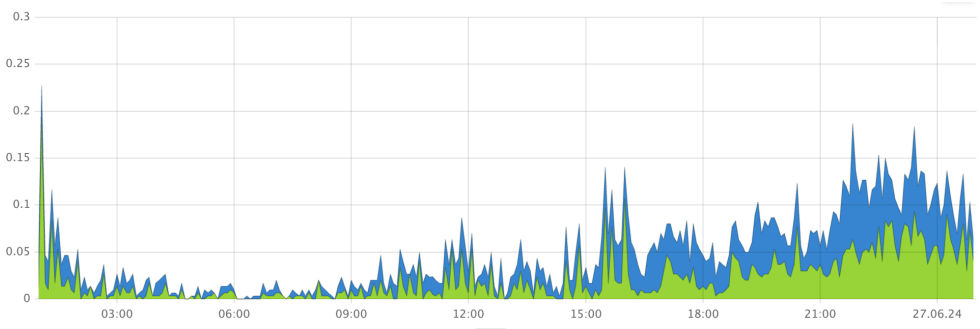
When YDB merges adjacent partitions in a row-oriented table, they are replaced with a single partition that covers their range of primary keys. TThe corresponding data shards are also consolidated into a single data shard to manage the new partition.

For merging to occur, data shards must have existed for at least 10 minutes, and their CPU usage over the last hour must not exceed 35%.

When configuring table partitioning, you can also set limits for the minimum and maximum number of partitions. If the difference between the minimum and maximum limits exceeds 20% and the table load varies significantly over time, Hive may start splitting overloaded tables and then merging them back during periods of low load.

Diagnostics

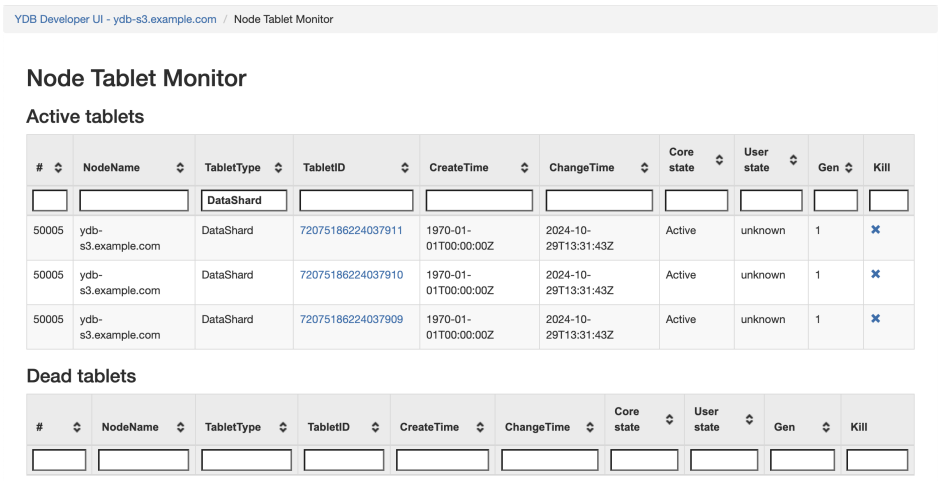
1. See if the **Split / Merge partitions** chart in the **DB status** Grafana dashboard shows any spikes.




```
This chart displays the time-series data for the following values:

- Number of split table partitions per second (blue)
- Number of merged table partitions per second (green)
```

2. Check whether the user load increased when the tablet splits and merges spiked.
   - Review the diagrams on the **DataShard** dashboard in Grafana for any changes in the volume of data read or written by queries.
   - Examine the **Requests** chart on the **Query engine** dashboard in Grafana for any spikes in the number of requests.
3. To identify recently split or merged tablets, follow these steps:
   3.1. In the YDB UI, click the **Developer UI** link in the upper right corner.
   3.2. Navigate to **Node Table Monitor** > **All tablets of the cluster**.
   3.3. To show only data shard tablets, in the **TabletType** filter, specify `DataShard`.



   3.4. Sort the tablets by the **ChangeTime** column and review tablets, which change time values coincide with the spikes on the **Split / Merge partitions** chart.
   3.5. To identify the table associated with the data shard, in the data shard row, click the link in the **TabletID** column.
   3.6. On the **Tablets** page, click the **App** link.

The information about the table is displayed in the **User table <table-name>** section.

4. To pinpoint the schema issue, follow these steps:

   4.1. Retrieve information about the problematic table using the YDB CLI. Run the following command:

   ```
   ydb scheme describe <table_name>
   ```

   4.2. In the command output, analyze the **Auto partitioning settings**:

   - `Partitioning by load`
   - `Max partitions count`
   - `Min partitions count`

Recommendations

If the user load on YDB has not changed, consider adjusting the gap between the min and max limits for the number of table partitions to the recommended 20% difference. Use the `ALTER TABLE table_name SET (key = value)` YQL statement to update the `AUTO_PARTITIONING_MIN_PARTITIONS_COUNT` and `AUTO_PARTITIONING_MAX_PARTITIONS_COUNT` parameters.

If you want to avoid splitting and merging data shards, you can set the min limit to the max limit value or disable partitioning by load.

Transaction lock invalidation

Transaction lock invalidation (TLI) occurs when one transaction (the breaker) writes data and breaks the optimistic locks of another transaction (the victim). The victim detects this at commit time and receives a `transaction locks invalidated` error. The transaction must be retried. Frequent retries degrade application performance.

> **Note**
>
> The YDB SDK provides a built-in mechanism for handling temporary failures. For more information, see Handling errors.

Preventing conflicts

- **Shorten transaction duration.** The longer a transaction holds locks, the higher the likelihood of a conflict. Where possible, avoid interactive transactions: the best approach is a single YQL query with `BEGIN;` and `COMMIT;` to read, modify, and commit data. If interactive transactions are necessary, execute `COMMIT` in the last query.
- **Reduce data overlap between transactions.** The fewer rows a transaction reads, the fewer locks it holds and the lower the chance of a conflict. Avoid reading unnecessary data. If multiple transactions compete for the same rows, reconsider your data model.
- **Use read-only transaction modes.** Transactions in `Snapshot Read-Only` mode read data from a consistent snapshot and do not acquire optimistic locks — such a transaction can never become a TLI victim. If a transaction does not modify data, explicitly set this mode in the SDK.

Example: transaction conflict during product reservation

In an online store during a flash sale, hundreds of buyers may try to purchase the same product at the same time. A purchase transaction reads the current stock, then the application calculates discounts, waits for payment confirmation, and only after that decrements the stock:

```
SELECT available FROM stock WHERE sku = $sku AND warehouse_id = $warehouse_id;
-- application checks: available > 0
-- application calculates discounts, waits for payment ...
UPDATE stock SET available = available - 1 WHERE sku = $sku AND warehouse_id = $warehouse_id;
COMMIT;
```

The `SELECT` acquires an optimistic lock on the row. While the transaction is waiting for payment, another transaction may update the same row and commit. This invalidates the lock, and the first transaction fails at commit time with a TLI error.

The solution is to reserve the product immediately in a short transaction, and perform all slower operations afterwards (discount calculation, payment, external service calls):

```
BEGIN;
UPDATE stock SET available = available - 1, reserved = reserved + 1
WHERE sku = $sku AND warehouse_id = $warehouse_id AND available > 0;

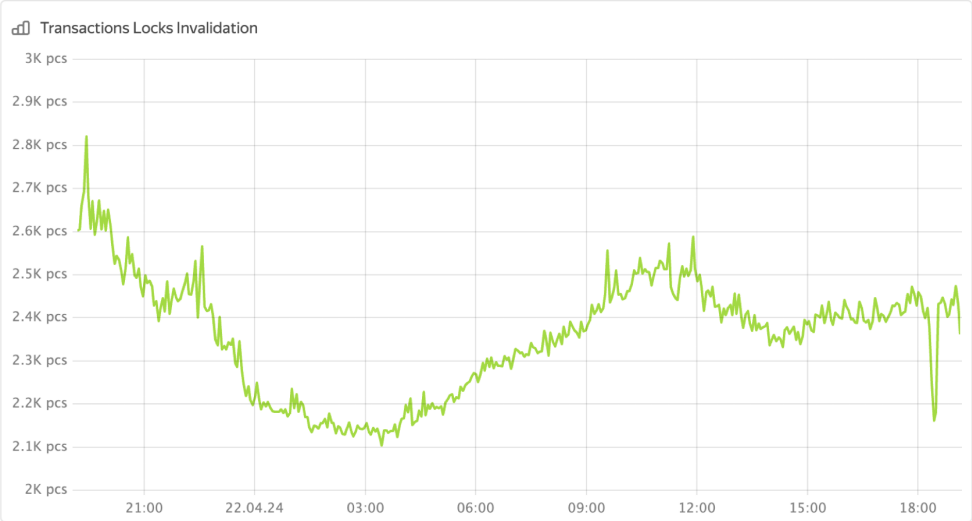
UPSERT INTO orders (order_id, sku, qty, status)
VALUES ($order_id, $sku, 1, "RESERVED");
COMMIT;
-- afterwards: calculate discounts, process payment, arrange delivery
```

If the payment succeeds, a separate transaction sets `status = "PAID"`. If it fails, another short transaction moves the item from `reserved` back to `available` and sets the status to `"CANCELLED"`. Because the reservation transaction completes quickly, it minimizes the time locks are held and reduces the probability of conflicts.

Diagnostics

Monitoring in Grafana

1. Open the **DB overview** Grafana dashboard.
2. See if the **Transaction Lock Invalidation** chart shows any spikes.



This chart shows the number of queries that returned the "transaction locks invalidated" error per second.

Diagnostics via TLI logs

When a transaction fails with a TLI error, the error message contains the victim query identifier:

```
Transaction locks invalidated. ... VictimQuerySpanId: 1111111111111111.
```

Using this `VictimQuerySpanId`, you can find the full conflict context in the server logs: which query acquired the locks and which one broke them. For details on enabling logging, log entry format, event correlation, and the `find_tli_chain` utility for automated log analysis, see TLI logging.

Analysis via system views

The following system views are available for analyzing lock conflicts:

Query-level analysis

To identify queries with the most conflicts, use the `.sys/query_metrics_one_minute` system view:

```
SELECT QueryText, LocksBrokenAsBreaker, LocksBrokenAsVictim
FROM `.sys/query_metrics_one_minute`
WHERE LocksBrokenAsBreaker > 0 OR LocksBrokenAsVictim > 0
ORDER BY LocksBrokenAsBreaker + LocksBrokenAsVictim DESC;
```

Column	Description
`LocksBrokenAsBreaker`	How many times this query broke other transactions' locks
`LocksBrokenAsVictim`	How many times this query's locks were broken

Queries with a high `LocksBrokenAsBreaker` are breakers: they cause other transactions to roll back. Queries with a high `LocksBrokenAsVictim` are victims.

Partition-level analysis

To analyze broken locks at the table partition level, use the following system views:

- `.sys/partition_stats` — current partition statistics, contains the cumulative `LocksBroken` field
- `.sys/top_partitions_by_tli_one_minute` — top 10 partitions with non-zero broken locks for one-minute intervals
- `.sys/top_partitions_by_tli_one_hour` — top 10 partitions with non-zero broken locks for one-hour intervals

Example query to find partitions with the most broken locks:

```
SELECT
    Path,
    SUM(LocksBroken) as TotalLocksBroken
FROM `.sys/partition_stats`
GROUP BY Path
ORDER BY TotalLocksBroken DESC
LIMIT 10;
```

Example query to view the history of broken locks by partition:

```
SELECT
    IntervalEnd,
    LocksBroken,
    Path
FROM `.sys/top_partitions_by_tli_one_hour`
WHERE IntervalEnd BETWEEN Timestamp("2000-01-01T00:00:00Z") AND Timestamp("2099-12-31T00:00:00Z")
ORDER BY IntervalEnd DESC, LocksBroken DESC
LIMIT 100;
```

TLI logging

Transaction lock invalidation (TLI) logging lets you identify which query had its locks broken **(the victim)** and which query broke them **(the breaker)**.

Enabling logging

To get detailed logs about lock conflicts, set the `INFO` level (numeric value `6`) for the `TLI` component.

Add the following to the cluster configuration:

```
log_config:
  entry:
    - component: "TLI"
      level: 6  # INFO
```

The `log_config` parameter supports dynamic updates without node restarts.

Once enabled, the server writes logs on every lock conflict, and `VictimQuerySpanId` begins appearing in SDK error messages.

> **Note**
>
> The base log size for a single TLI event is 10-20 KB. The actual volume depends on the size of the text of the logged requests (the breaker and the victim) and the transactions of which they are part, since all this data is recorded in the log. If conflicts are infrequent, the logging overhead is negligible. At high TLI rates, monitor the total log volume.
>
> To exclude tables with expected conflicts from TLI diagnostics, use the `tli_config.ignored_table_regexes` parameter.

Log structure

When TLI occurs, the server writes four entries — one from each component for each side of the conflict.

Example scenario:

Two sessions are established: victim and breaker.

Time	QuerySpanId	Role	Query
T1	`1111111111111111`	Victim	`SELECT * FROM Orders WHERE OrderId = 42` (acquires lock)
T2	`2222222222222222`	Breaker	`UPDATE Orders SET Status = 'done' WHERE OrderId = 42` (breaks lock)
T3	`3333333333333333`	Victim	`UPDATE Orders SET Amount = 100 WHERE OrderId = 42` (commit fails)

Breaker log (DataShard):

```
Component: DataShard, TabletId: <tablet-id>,
BreakerQuerySpanId: 2222222222222222, VictimQuerySpanIds: [1111111111111111],
Message: Write transaction broke other locks
```

Breaker log (SessionActor):

```
Component: SessionActor, Message: Query had broken other locks,
BreakerQuerySpanId: 2222222222222222,
BreakerQueryText: UPDATE Orders SET Status = 'done' WHERE OrderId = 42,
BreakerQueryTexts: [QuerySpanId=2222222222222222 QueryText=UPDATE Orders SET Status = 'done' WHERE OrderId =
42]
```

Victim log (DataShard):

```
Component: DataShard, TabletId: <tablet-id>,
VictimQuerySpanId: 1111111111111111, CurrentQuerySpanId: 3333333333333333,
Message: Write transaction was a victim of broken locks
```

Victim log (SessionActor):

```
Component: SessionActor, Message: Query was a victim of broken locks,
VictimQuerySpanId: 1111111111111111, CurrentQuerySpanId: 3333333333333333,
VictimQueryText: SELECT * FROM Orders WHERE OrderId = 42,
VictimQueryTexts: [QuerySpanId=1111111111111111 QueryText=SELECT * FROM Orders WHERE OrderId = 42 |
                   QuerySpanId=3333333333333333 QueryText=UPDATE Orders SET Amount = 100 WHERE OrderId = 42]
```

Log fields

Field	Description	Where it appears
`VictimQuerySpanId`	Identifier of the query whose locks were broken	Victim logs
`BreakerQuerySpanId`	Identifier of the query that broke the locks	Breaker logs
`CurrentQuerySpanId`	Identifier of the query at the time of the error (may differ from the victim)	Victim logs
`VictimQuerySpanIds`	Array of identifiers of all victim queries	Breaker DataShard logs

`VictimQueryText`	SQL of the victim query that acquired the locks	Victim SessionActor logs
`BreakerQueryText`	SQL of the breaker query	Breaker SessionActor logs
`VictimQueryTexts`	All queries of the victim transaction	Victim SessionActor logs
`BreakerQueryTexts`	All queries of the breaker transaction	Breaker SessionActor logs

Log analysis

Using the `VictimQuerySpanId` from the SDK error message, you can find all related events:

1. **Find the victim query**: search for `VictimQuerySpanId: <value from the error>` in the logs — this shows which SELECT acquired the broken locks, with the query text in the `VictimQueryText` field.
2. **Find the breaker query**: the breaker DataShard log's `VictimQuerySpanIds` field contains the same value. From this log, take `BreakerQuerySpanId` and find the breaker's SessionActor log — it contains `BreakerQueryText` with the full query text.
3. **Get full transaction context**: `VictimQueryTexts` and `BreakerQueryTexts` contain all queries of the respective transactions in execution order.

find_tli_chain utility

For automatic TLI log analysis, the YDB repository includes the `find_tli_chain.py` utility.

This utility takes the following parameters:

- `VictimQuerySpanId` - identifier of the broken query from the SDK error message;
- `LogFile` - name of the log file for log analysis.

It is assumed that TLI logging was previously enabled in the configuration, log files were collected from database nodes and merged into one file.

```
python3 find_tli_chain.py <VictimQuerySpanId> <LogFile>
```

Example:

```
python3 find_tli_chain.py 1111111111111111 ydb.log
```

```
================================================
  TLI Chain
================================================

VictimQuerySpanId: 1111111111111111

VictimQueryText: SELECT * FROM Orders WHERE OrderId = 42

BreakerQuerySpanId: 2222222222222222

BreakerQueryText: UPDATE Orders SET Status = 'done' WHERE OrderId = 42

================================================
  VictimTx
================================================

SELECT * FROM Orders WHERE OrderId = 42

UPDATE Orders SET Amount = 100 WHERE OrderId = 42

================================================
  BreakerTx
================================================

UPDATE Orders SET Status = 'done' WHERE OrderId = 42
```

The utility outputs:

- **TLI Chain** — `VictimQuerySpanId`, `VictimQueryText`, `BreakerQuerySpanId`, `BreakerQueryText`;
- **VictimTx** — all queries of the victim transaction in execution order;
- **BreakerTx** — all queries of the breaker transaction.

Additional parameters:

Parameter	Description
`--window-sec N`	Time window for searching around the event (default: 10 s)
`--no-color`	Disable colored output

> **Note**
>
> The utility correctly handles unsorted and mixed log files: filtering is based on timestamp values, not line positions.

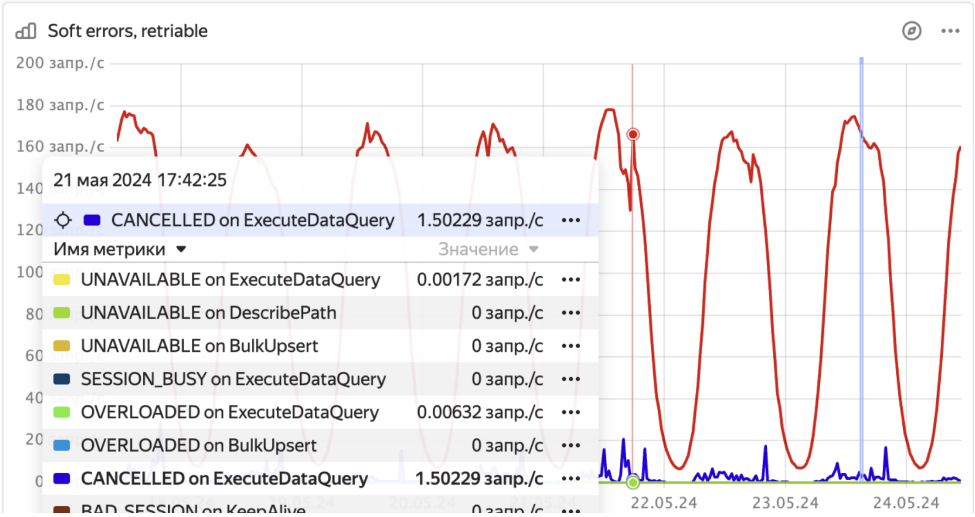
Overloaded errors

YDB returns `OVERLOADED` errors in the following cases:

- Overloaded table partitions with over 15000 queries in their queue.
- The outbound CDC queue exceeds the limit of 10000 elements or 125 MB.
- Table partitions in states other than normal, for example partitions in the process of splitting or merging.
- The number of sessions with a YDB node has reached the limit of 1000.

Diagnostics

1. Open the **DB overview** Grafana dashboard.
2. In the **API details** section, see if the **Soft errors (retriable)** chart shows any spikes in the rate of queries with the `OVERLOADED` status.



3. To check if the spikes in overloaded errors were caused by exceeding the limit of 15000 queries in table partition queues:
   3.1. In the YDB UI, go to the **Databases** tab and click on the database.
   3.2. On the **Navigation** tab, ensure the required database is selected.
   3.3. Open the **Diagnostics** tab.
   3.4. Open the **Top shards** tab.
   3.5. In the **Immediate** and **Historical** tabs, sort the shards by the **InFlightTxCount** column and see if the top values reach the 15000 limit.
4. To check if the spikes in overloaded errors were caused by tablet splits and merges, see Excessive tablet splits and merges.
5. To check if the spikes in overloaded errors were caused by exceeding the 1000 limit of open sessions, in the Grafana **DB status** dashboard, see the **Session count by host** chart.
6. See the overloaded shards issue.

Recommendations

If a YQL query returns an `OVERLOADED` error, retry the query using a randomized exponential backoff strategy. The YDB SDK provides a built-in mechanism for handling temporary failures. For more information, see Handling errors.

Exceeding the limit of open sessions per node may indicate a problem in the application logic.

Retry cascade effect

A cascade effect occurs when a client starts frequently retrying queries whose execution time exceeds the configured timeout.

Mechanism

Query execution time exceeds client timeout. Client starts repeating requests, but original queries continue executing on the server. Each subsequent retry takes longer than the previous one as the server processes more concurrent requests. At some point, query latency permanently exceeds client timeout. Client continues retrying, increasing load and further raising latency.

YDB specifics

In YDB's actor model, canceling in-flight requests has limitations. This is especially evident with client timeouts - when a client closes the gRPC stream before receiving a response, the request may continue executing on the server.

Recovery

A shard can recover normal operation only through temporary complete load removal, shard restart, or load-based shard splitting.

Recommendations

Avoid retrying timeouts or use them with caution. Apply exponential backoff with sufficiently large delays. Ensure the load actually decreases before retrying. Aggressive retries in an overloaded state have the opposite effect - the load on the shard increases by times.
For more information on proper error handling, see Handling errors.

Spilling Troubleshooting

This section provides troubleshooting information for common spilling issues in YDB. Spilling is a memory management mechanism that temporarily saves intermediate computation data to disk when the system runs out of RAM. These errors can occur during query execution when the system attempts to use spilling functionality and can be observed in logs and query responses.

Common Issues

- Permission denied - Insufficient access permissions to the spilling directory
- Spilling Service not started - Attempt to use spilling when the Spilling Service is disabled
- Total size limit exceeded - Maximum total size of spilling files exceeded

See Also

- Spilling Configuration
- Spilling Concept
- Memory Controller Configuration
- YDB Monitoring
- Performance Diagnostics

Permission Denied

Insufficient access permissions to the spilling directory prevent YDB from writing data to disk during spilling operations. This can cause queries to fail when they require spilling to handle large data volumes.

Diagnostics

Check if the spilling directory exists and has proper permissions:

- Verify that the spilling directory exists (see Spilling Configuration for information on how to find the spilling directory)
- Ensure the directory has read and write permissions for the user under which `ydbd` is running
- Check access permissions to the spilling directory
- Verify that the user under which `ydbd` runs can read and write to the directory

Recommendations

If permissions are incorrect:

1. Change the directory owner to the user under which `ydbd` runs.
2. Ensure read/write permissions are set for the directory owner.
3. Restart the `ydbd` process to apply the changes.

> **Note**
>
> The spilling directory is automatically created by YDB when the process starts. If the directory doesn't exist, check that the `root` parameter in the spilling configuration is set correctly.

Spilling Service Not Started

Spilling is disabled, but the system ran out of memory during query execution. When spilling is disabled, YDB cannot offload intermediate data to disk, which leads to query failure when available memory is exhausted.

Diagnostics

Check the spilling service configuration:

- Verify that `table_service_config.spilling_service_config.local_file_config.enable` is set to `true`.

Recommendations

To fix this error, enable spilling:

1. Set `table_service_config.spilling_service_config.local_file_config.enable`: `true` in your configuration.

For more details on enabling spilling, see the Enable section.

> **Note**
>
> Read more about the spilling architecture in Spilling in YDB.

Total Size Limit Exceeded

The maximum total size of spilling files has been exceeded (parameter `max_total_size`). This occurs when the total size of all spilling files reaches the configured limit, preventing new spilling operations.

Diagnostics

Check the current spilling usage:

- Monitor the total size of spilling files in the spilling directory.
- Check the current value of the `max_total_size` parameter.
- Review available disk space in the spilling directory location.
- Check if there are any stuck spilling files that should have been cleaned up.

Recommendations

To resolve this issue:

1. **Increase the spilling size limit:**
   - If there is sufficient free disk space, increase the `max_total_size` parameter in the configuration.
   - Increase the value by 20–50% from the current one.
2. **Expand disk space:**
   - If there is insufficient free disk space, add additional disk space.
   - Ensure that the spilling directory is located on a disk with sufficient capacity.
3. **Try repeating the query:**
   - Wait for other resource-intensive queries to complete.
   - Repeat the query execution during less busy times.

Overloaded shard example

This article describes an example of how to diagnose overloaded shards and resolve the issue.

For more information about overloaded shards and their causes, see Overloaded shards.

The article begins by stating the problem. Then, we'll examine diagrams in Grafana and information on the **Diagnostics** tab in the YDB UI to solve the problem and observe the solution in action.

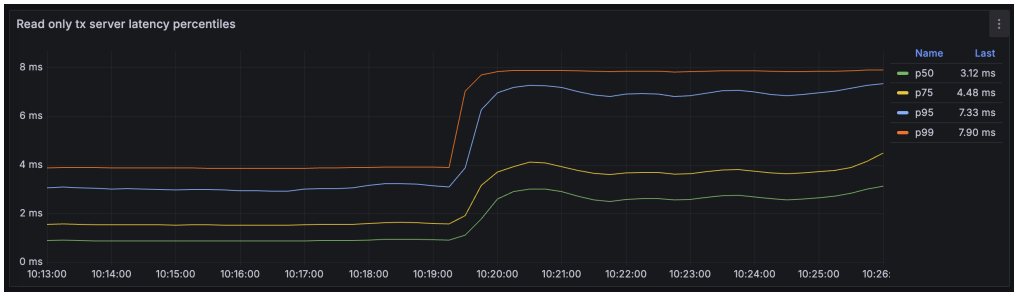
At the end of the article, you can find the steps to reproduce the situation.

Initial issue

You were notified that your system has started taking too long to process user requests.

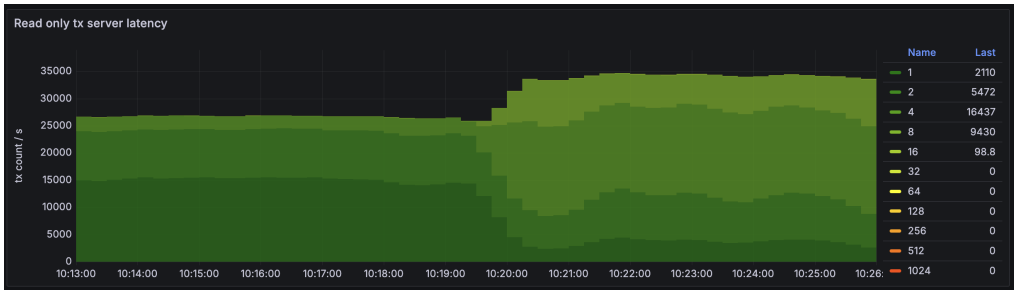
> **Note**
>
> These requests access a row-oriented table, which is managed by data shards.

Let's examine the **Latency** diagrams in the DB overview Grafana dashboard to determine whether the problem is related to the YDB cluster:



See the diagram description

The diagram shows transaction latency percentiles. At approximately `10:19:30`, these values increased by two to three times.



See the diagram description

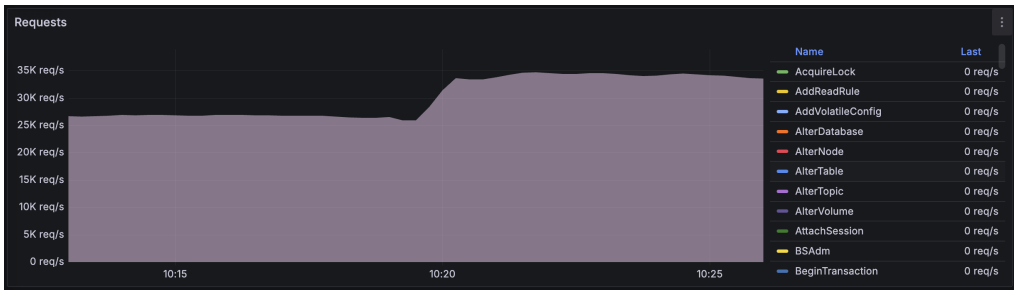
The diagram shows a heatmap of transaction latencies. Transactions are grouped into buckets based on their latency, with each bucket represented by a different color. This diagram displays both the number of transactions processed by YDB per second (on the vertical axis) and the latency distribution among them (with color).

By `10:20:30`, the share of transactions with the lowest latencies (`Bucket 1`, dark green) had dropped by four to five times. `Bucket 4` grew by approximately five times, and a new group of slower transactions, `Bucket 8`, appeared.

Indeed, the latencies have increased. Now, we need to localize the problem.

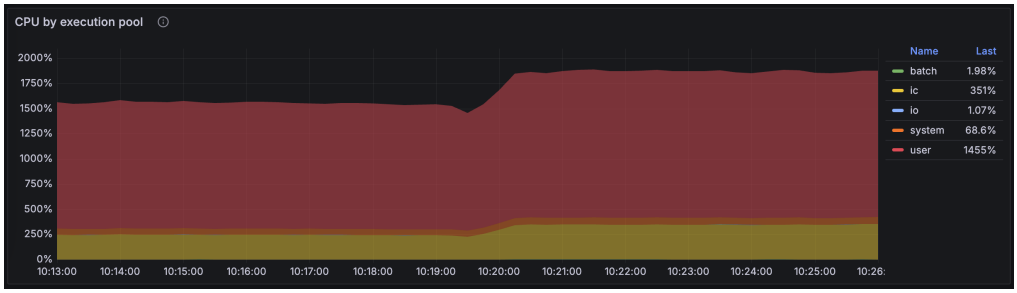
Diagnostics

Let's determine why the latencies increased. Could the cause be an increased workload? Here is the **Requests** diagram from the **API details** section of the DB overview Grafana dashboard:



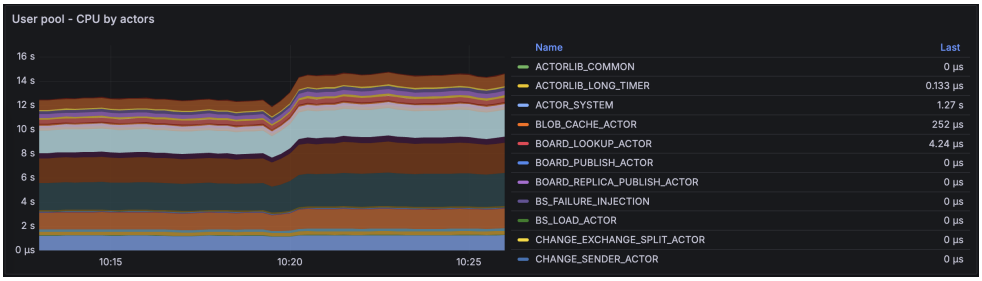
The number of user requests increased from approximately 27,000 to 35,000 at around `10:20:00`. But can YDB handle the increased load without additional hardware resources?

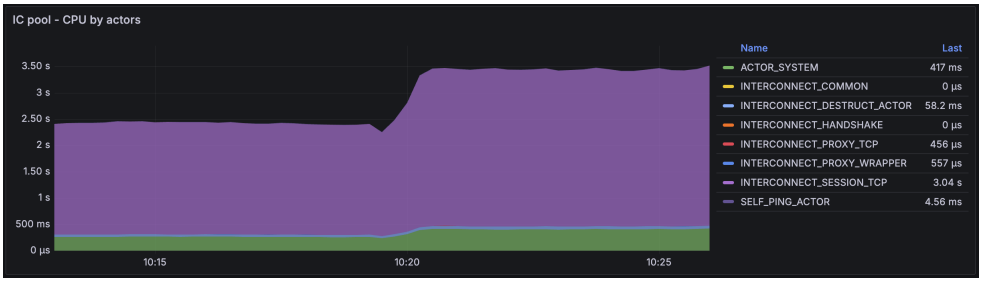
The CPU load has increased, as shown in the **CPU by execution pool** diagram.

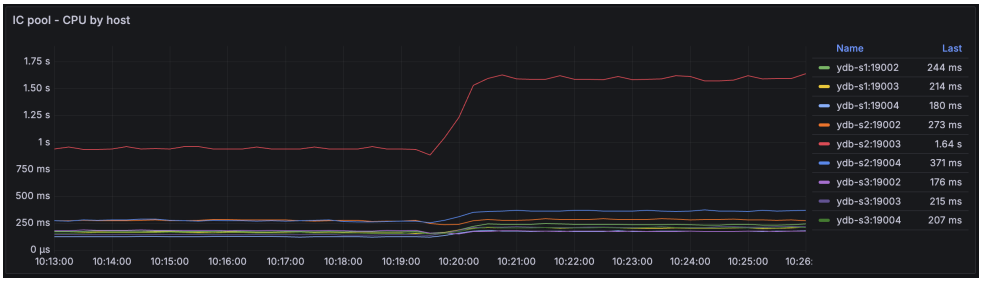


See the details on the CPU Grafana dashboard

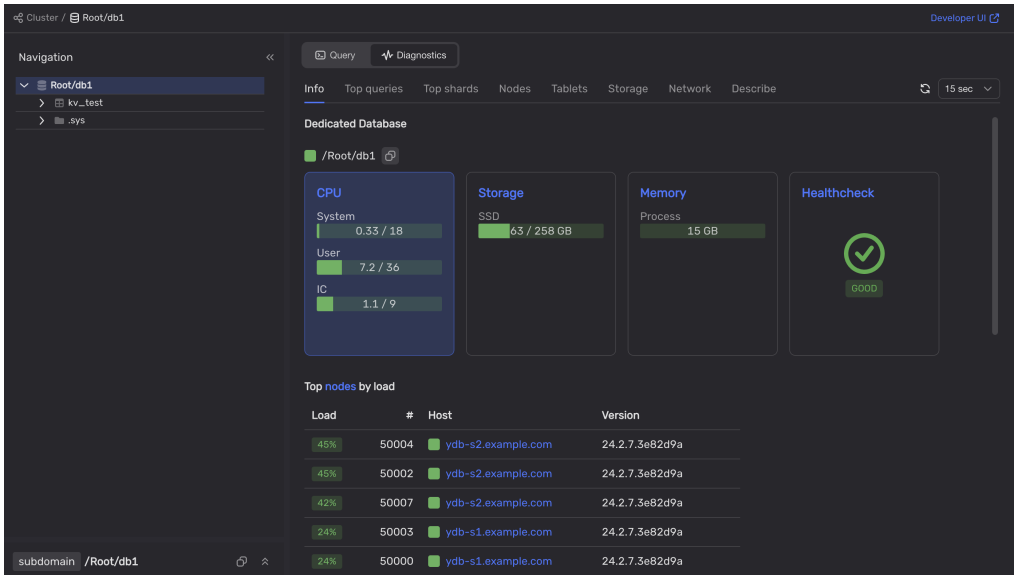
Examining the **CPU** Grafana dashboard reveals that CPU usage increased in the user pool and the interconnect pool:





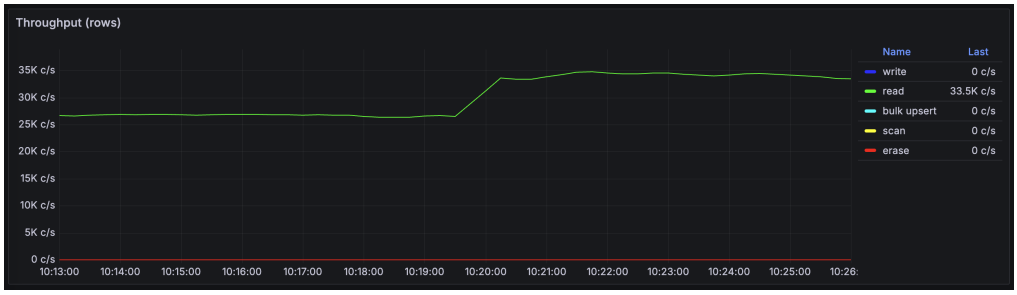


We can also observe overall CPU usage on the **Diagnostics** tab of the YDB UI:



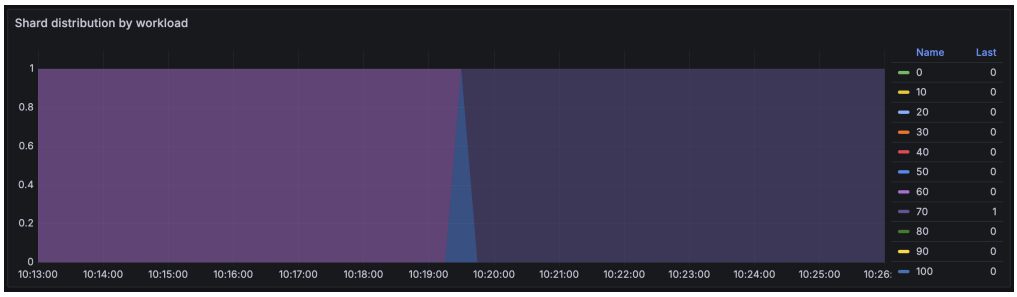
The YDB cluster appears not to utilize all of its CPU capacity.

By inspecting the **DataShard** and **DataShard details** sections of the DB overview Grafana dashboard, we can see that after the cluster load increased, one of its data shards became overloaded.



See the diagram description

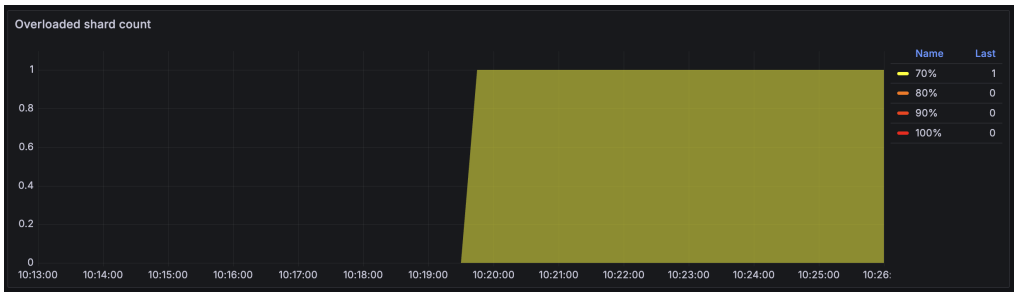
This diagram shows that the number of rows read per second in the YDB database increased from approximately 26,000 to 33,500 rows per second.



See the diagram description

This diagram shows a heatmap of data shard distribution by workload. Data shards are grouped into ten buckets based on the ratio of their current workload to full computing capacity. This allows you to see how many data shards your YDB cluster currently runs and how loaded they are.

The diagram shows only one data shard whose workload changed at approximately `10:19:30`—the data shard moved to `Bucket 70`, which contains shards loaded to between 60% and 70% of their capacity.

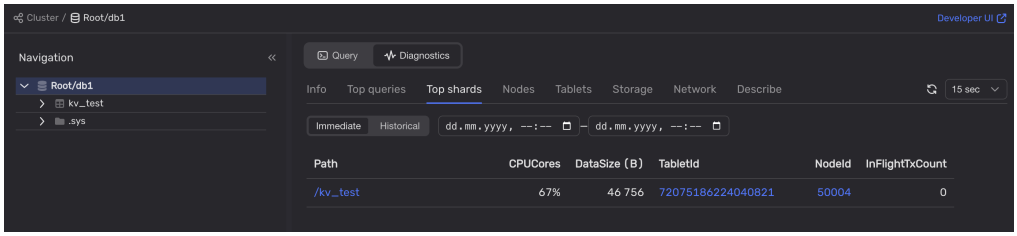


See the diagram description

Similar to the previous diagram, the **Overloaded shard count** is a heatmap of data shard distribution by load. However, it displays only data shards with a workload exceeding 60%.

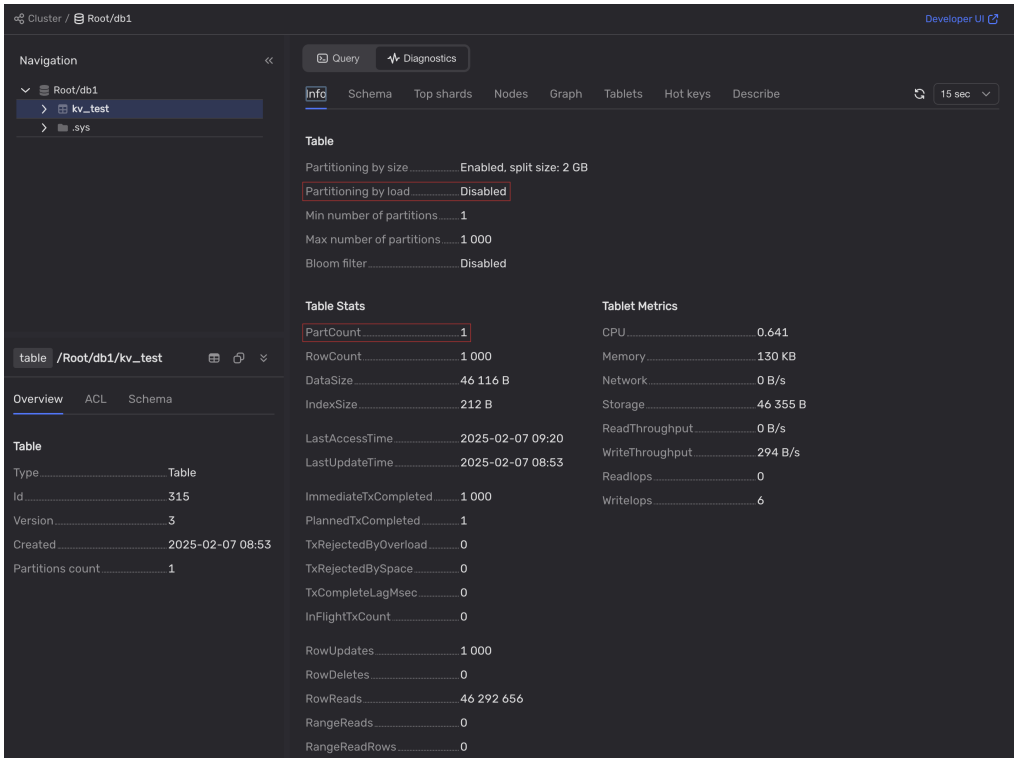
This diagram shows that the workload on one data shard increased to 70% at approximately `10:19:30`.

To determine which table the overloaded data shard is processing, let's open the **Diagnostics > Top shards** tab in the YDB UI:



We can see that one of the data shards processing queries for the `kv_test` table is loaded at 67%.

Next, let's examine the `kv_test` table on the **Info** tab:



Warning

The `kv_test` table was created with partitioning by load disabled and has only one partition.

This means that a single data shard processes all requests to this table. Since data shards are single-threaded and thus can handle only one request at a time, this is a poor practice.

Solution

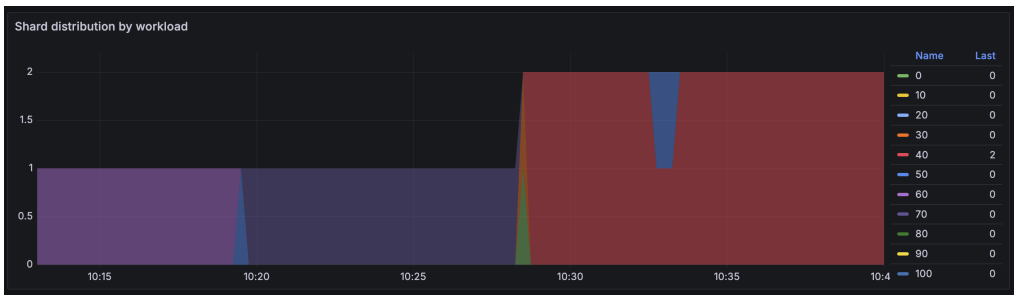
We should enable partitioning by load for the `kv_test` table:

1. In the YDB UI, select the database.
2. Open the **Query** tab.
3. Run the following query:

```
ALTER TABLE kv_test SET (
    AUTO_PARTITIONING_BY_LOAD = ENABLED
);
```

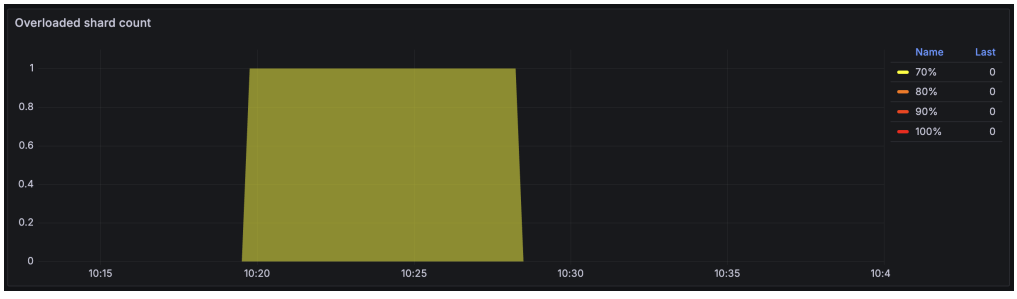
Aftermath

When we enable automatic partitioning for the `kv_test` table, the overloaded data shard splits into two.



See the diagram description

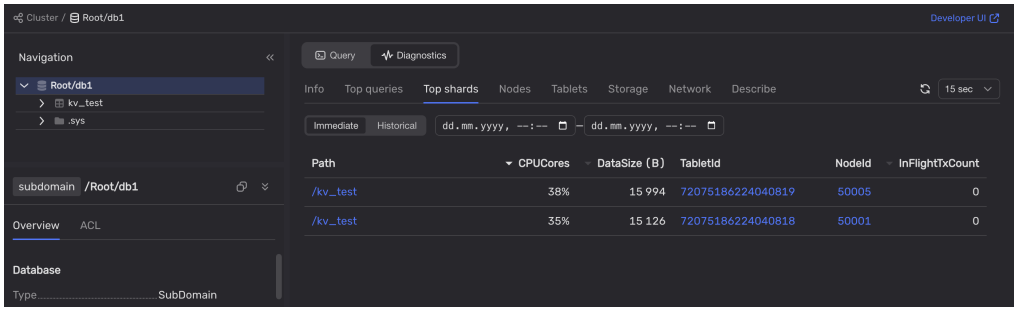
The diagram shows that the number of data shards increased at about `10:28:00`. Based on the bucket color, their workload does not exceed 40%.



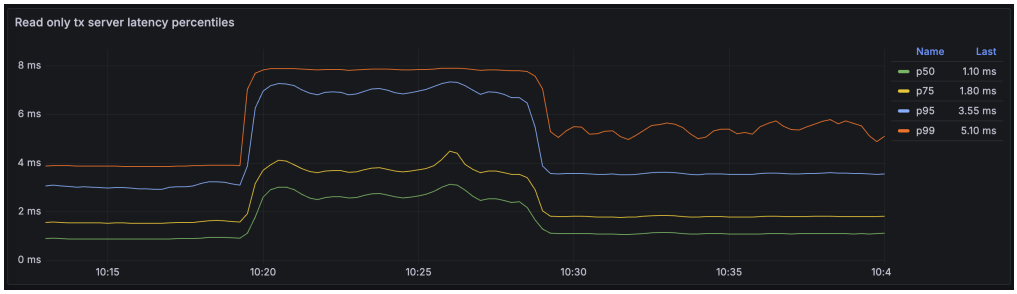
See the diagram description

The overloaded shard disappeared from the diagram at approximately `10:28:00`.

Now, two data shards are processing queries to the `kv_test` table, and neither is overloaded:

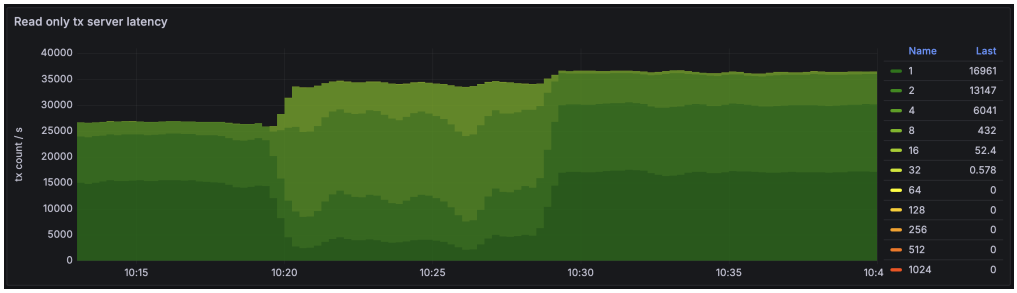


Let's confirm that latencies have returned to normal:



See the diagram description

At approximately `10:28:00`, the p50, p75, and p95 latency percentiles dropped almost to their original levels. The decrease in p99 latency is less pronounced but still shows a twofold reduction.



See the diagram description

The diagram shows that transactions are now grouped into six buckets. Approximately half of the transactions have returned to `Bucket 1`, meaning their latency is less than one millisecond. More than a third of the transactions are in `Bucket 2`, with latencies between one and two milliseconds. One-sixth of the transactions are in `Bucket 4`. The sizes of the other buckets are insignificant.

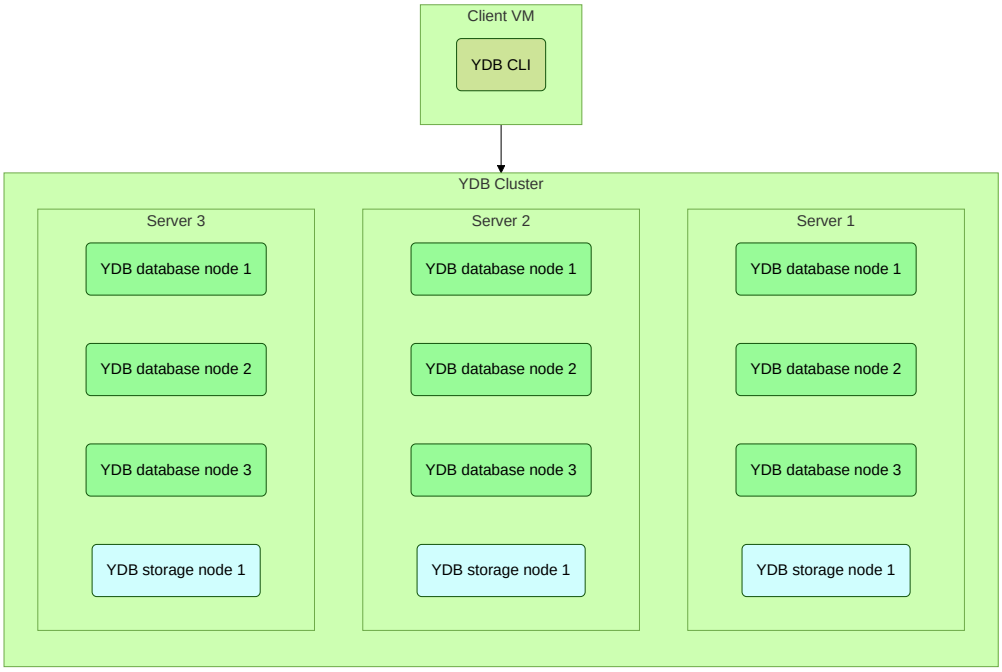
The latencies are almost as low as they were before the workload increased. We did not increase the system costs by introducing additional hardware resources. We've only enabled automatic partitioning by the load, which allowed us to use the existing resources more efficiently.

Bucket name	Latencies, ms	Single overloaded data shard, transactions per second	Multiple data shards, transactions per second
1	0-1	2110	▲ 16961
2	1-2	5472	▲ 13147
4	2-4	16437	▼ 6041
8	4-8	9430	▼ 432
16	8-16	98.8	▼ 52.4
32	16-32	—	▲ 0.578

Testbed

Topology

For this example, we used a YDB cluster consisting of three servers running Ubuntu 22.04 LTS. Each server runs one storage node and three database nodes belonging to the same database.



Hardware configuration

The servers are virtual machines with the following computing resources:

- Platform: Intel Broadwell
- Guaranteed vCPU performance: 100%
- vCPU: 28
- RAM: 32 GB
- Storage:
  - 3 x 93 GB SSD per storage node
  - 20 GB HDD for the operating system

Test

The load on the YDB cluster was generated using the `ydb workload` CLI command. For more information, see Load testing.

To reproduce the load, follow these steps:

1. Initialize the tables for the workload test:

   ```
   ydb workload kv init --min-partitions 1 --auto-partition 0
   ```

   We deliberately disable automatic partitioning for the created tables by using the `--min-partitions 1 --auto-partition 0` options.

2. Emulate the standard workload on the YDB cluster:

   ```
   ydb workload kv run select -s 600 -t 100
   ```

   We ran a simple load type using a YDB database as a key-value storage. Specifically, we used the `select` load to create SELECT queries and retrieve rows based on an exact match of the primary key.

   The `-t 100` parameter is used to run the test in 100 threads.

3. Overload the YDB cluster:

   ```
   ydb workload kv run select -s 1200 -t 250
   ```

   As soon as the first test ended, we ran the same load test in 250 threads to simulate the overload.

See also

- Troubleshooting performance issues
- Overloaded shards
- Row-oriented tables

Questions and answers about YDB

- General
- SDK
- Errors
- YQL
- Analytics
- Known issues

General questions about YDB

What is YDB?

YDB is a distributed fault-tolerant SQL DBMS. YDB offers high availability and scalability, while ensuring strong consistency and support for ACID transactions. Queries are made using an SQL dialect (YQL).

YDB is a fully managed database. DB instances are created through the YDB database management service.

What features does YDB provide?

YDB provides high availability and data security through synchronous replication in three availability zones. YDB also ensures even load distribution across available hardware resources. This means you don't need to order resources, YDB automatically provisions and releases resources based on the user load.

What consistency model does YDB use?

To read data, YDB uses a model of strict data consistency.

How do I design a primary key?

To design a primary key properly, follow the rules below.

- Avoid situations where most of the load falls on a single partition of a table. With even load distribution, it's easier to achieve high overall performance.This rule implies that you shouldn't use a monotonically increasing sequence, such as timestamp, as a table's primary key.
- The fewer table partitions a query uses, the faster it runs. For greater performance, follow the one query — one partition rule.
- Avoid situations where a small part of the DB is under much heavier load than the rest of the DB.

For more information, see choosing a primary key.

How do I evenly distribute load across table partitions?

You can use the following techniques to distribute the load evenly across table partitions and increase overall DB performance.

- To avoid using uniformly increasing primary key values, you can:
  - Change the order of its components.
  - use a hash of the key column values as the primary key.
- Reduce the number of partitions used in a single query.

For more information, see choosing a primary key.

Can I use NULL in a key column?

In YDB, all columns, including key ones, may contain a `NULL` value, but we don't recommend using `NULL` as values in key columns.

Per the SQL standard (ISO/IEC 9075), you can't compare `NULL` with other values. Therefore, the use of concise SQL statements with simple comparison operators may result in rows containing NULL being skipped during filtering, for example.

Is there an optimal size of a database row?

To achieve high performance, we don't recommend writing rows larger than 8 MB and key columns larger than 2 KB to the DB.

For more information about limits, see Database limits.

How are secondary indexes used in YDB?

Secondary indexes in YDB are global and can be non-unique.

For more information, see Secondary indexes.

How are paginated results printed?

To print paginated results, we recommend selecting data sorted by primary key sequentially, limiting the number of rows with the `LIMIT` keyword. We do not recommend using the `OFFSET` keyword to solve this problem.

For more information, see Paginated results.

How do I delete expired data?

To efficiently delete outdated data, we recommend using TTL.

Syncing two data centers in geographically distributed clusters

The lead tablet writes data to a distributed network storage that saves copies to several data centers. YDB does not commit a user query until after the required number of copies are saved to the required number of data centers.

SDK

What should I do if the SDK crashes when shutting down an application?

Make sure not to wrap SDK components in a singleton, since their lifetime shouldn't exceed the execution time of the `main()` function. When a client is destroyed, session pools are emptied, so network navigation is required. But gRPC contains global static variables that might already be destroyed by this time. This disables gRPC. If you need to declare a driver as a global object, invoke the `Stop(true)` function on the driver before exiting the `main()` function.

What should I do if, when using a `fork()` system call, a program does not work properly in a child process?

Using `fork()` in multithreaded applications is an antipattern. Since both the SDK and the gRPC library are multithreaded applications, their stability is not guaranteed.

What do I do if I get the "Active sessions limit exceeded" error even though the current number of active sessions is within limits?

The limit applies to the number of active sessions. An active session is a session passed to the client to be used in its code. A session is returned to the pool in a destructor. In this case, the session itself is a replicated object. You may have saved a copy of the session in the code.

Is it possible to make queries to different databases from the same application?

Yes, the C++ SDK lets you override the DB parameters and token when creating a client. There is no need to create separate drivers.

What should I do if a VM has failed and it's impossible to make a query?

To detect that a VM is unavailable, set a client timeout. All queries contain the client timeout parameters. The timeout value should be an order of magnitude greater than the expected query execution time.

Errors

Possible causes for "Status: OVERLOADED Error: Pending previous query completion" in the C++ SDK

Q: When running two queries, I try to get a response from the future method of the second one. It returns: `Status: OVERLOADED Why: <main>: Error: Pending previous query completion`.

A: Sessions in the SDK are single-threaded. To run multiple queries at once, you need to create multiple sessions.

What do I do if I frequently get the "Transaction locks invalidated" error?

Typically, if you get this error, repeat a transaction, as YDB uses optimistic locking. If this error occurs frequently, this is the result of a transaction reading a large number of rows or of many transactions competing for the same "hot" rows. It makes sense to view the queries running in the transaction and check if they're reading unnecessary rows.

What causes the "Exceeded maximum allowed number of active transactions" error?

The logic on the client side should try to keep transactions as short as possible.

No more than 10 active transactions are allowed per session. When starting a transaction, use either the commit flag for autocommit or an explicit commit/rollback.

What do I do if I get the Datashard: Reply size limit exceeded error in response to a query?

This error means that, as a query was running, one of the participating data shards attempted to return over 50 MB of data, which exceeds the allowed limit.

Recommendations:

- A general recommendation is to reduce the amount of data processed in a transaction.
- If a query involves a `Join`, it's a good idea to make sure that the method used is Index lookup Join.
- If a simple selection is performed, make sure that it is done by keys, or add `LIMIT` in the query.

What do I do is I get the "Datashard program size limit exceeded" in response to a query?

This error means that the size of a program (including parameter values) exceeded the 50-MB limit for one of the data shards. In most cases, this indicates an attempt to write over 50 MB of data to database tables in a single transaction. All modifying operations in a transaction such as `UPSERT`, `REPLACE`, `INSERT`, or `UPDATE` count as records.

You need to reduce the total size of records in one transaction. Normally, we don't recommend combining queries that logically don't require transactionality in a single transaction. When adding/updating data in batches, we recommend reducing the size of one batch to values not exceeding a few megabytes.

YQL

General questions

How do I select table rows by a list of keys?

You can select table rows based on a specified list of table primary key (or key prefix) values using the `IN` operator:

```
DECLARE $keys AS List<UInt64>;

SELECT * FROM some_table
WHERE Key1 IN $keys;
```

If a selection is made using a composite key, the query parameter must have the type of a list of tuples:

```
DECLARE $keys AS List<Tuple<UInt64, String>>;

SELECT * FROM some_table
WHERE (Key1, Key2) IN $keys;
```

To select rows effectively, make sure that the value types in the parameters match the key column types in the table.

Is search by index performed for conditions containing the LIKE operator?

You can only use the `LIKE` operator to search a table index if it specifies a row prefix:

```
SELECT * FROM string_key_table
WHERE Key LIKE "some_prefix%";
```

Why does a query return only 1000 rows?

1000 rows is the response size limit per YQL query. If a response is shortened, it is flagged as `Truncated`. To output more table rows, you can use paginated output or the `ReadTable` operation.

How to escape quotes of JSON strings when adding them to a table?

Consider an example with two possible options for adding a JSON string to a table:

```
UPSERT INTO test_json(id, json_string)
VALUES
    (1, Json(@@[{"name":"Peter \"strong cat\" Kourbatov"}]@@)),
    (2, Json('[{"name":"Peter \\\"strong cat\\\" Kourbatov"}]'))
;
```

To insert a value in the first line, use `raw string` and the escape method using `\"`. To insert the second line, escaping through `\\\"` is used.

We recommend using `raw string` and the escape method using `\"`, as it is more visual.

How do I update a JSON value in a table?

Partial updates of a JSON field are not supported, so it is necessary to construct a new value from parts of the old and new data. This is done using the `Yson` UDF combined with dictionary functions.

The example below adds a new field `new_field` to the JSON object stored in the `column` column:

```
UPDATE table
SET column = Yson::SerializeJson(
    Yson::From(
        SetUnion(
            Yson::ConvertTo(Yson::ParseJson(column), Dict<String, Yson>),
            {'new_field': Yson('"value"')},
            ($K, $v1, $v2) -> { RETURN COALESCE($v2, $v1); }
        )
    )
)
WHERE id = 1;
```

How do I update only those values whose keys are not in the table?

You can use the `LEFT JOIN` operator to identify the keys a table is missing and update their values:

```
DECLARE $values AS List<Struct<Key: UInt64, Value: String>>;

UPSERT INTO kv_table
SELECT v.Key AS Key, v.Value AS Value
FROM AS_TABLE($values) AS v
LEFT JOIN kv_table AS t
ON v.Key = t.Key
WHERE t.Key IS NULL;
```

Join operations

Are there any specific features of Join operations?

A `Join` in YDB is performed using one of the two methods below:

- Common Join.
- Index Lookup Join.

Common Join

The contents of both tables (to the left and to the right of `Join`) are sent to the requesting node which applies the operation to the totality of the data. This is the most generic way of performing a `Join` that is used whenever other optimizations are unavailable. For large tables, this method is either slow or doesn't work in general due to exceeding the data transfer limits.

Index Lookup Join

For rows on the left of `Join`, relevant values are looked up to the right. You use this method whenever the right part is a table and the `Join` key is its primary or secondary index key prefix. In this method, limited selections are made from the right table instead of full reads. This lets you use it when working with large tables.

> **Note**
>
> For most OLTP queries, we recommend using Index Lookup Join with a small size of the left part. These operations read little data and can be performed efficiently.

How do I Join data from query parameters?

You can use query parameter data as a constant table. To do this, use the `AS_TABLE` modifier with a parameter whose type is a list of structures:

```
DECLARE $data AS List<Struct<Key1: UInt64, Key2: String>>;

SELECT * FROM AS_TABLE($data) AS d
INNER JOIN some_table AS t
ON t.Key1 = d.Key1 AND t.Key2 = d.Key2;
```

There is no explicit limit on the number of entries in the constant table, but mind the standard limit on the total size of query parameters (50 MB).

What's the best way to implement a query like (key1, key2) IN ((v1, v2), (v3, v4), ...)?

You can use the `WHERE ... IN` syntax to query by a set of keys (including composite keys):

```
DECLARE $key_pairs AS List<Tuple<Uint64, Uint64>>;

SELECT * FROM some_table
WHERE (Key1, Key2) IN $key_pairs;
```

Transactions

How efficient is it to run multiple queries in a transaction?

When multiple queries are run sequentially, the total transaction latency may be greater than when the same operations are executed within a single query. This is primarily due to additional network latency for each query. Therefore, if a transaction doesn't need to be interactive, we recommend formulating all operations in a single YQL query.

Is a separate query atomic?

In general, YQL queries can be executed in multiple consecutive phases. For example, a Join query can be executed in two phases: reading data from the left and right table, respectively. This aspect is important when you run a query in a transaction with a low isolation level (`online_read_only`), as in this case, data between execution phases can be updated by other transactions.

Questions and answers about analytics in YDB

Can YDB be used for analytical workloads (OLAP)?

Yes, it can. If this is the primary type of workload for a given table, make sure it is column-oriented.

How to choose between row-oriented and column-oriented tables?

Similarly to choosing between transactional (OLTP) and analytical (OLAP) database management systems, this question comes to a number of trade-offs that need to be considered:

- **What's the main use case for the table?** For mostly transactional (OLTP) workloads, use row-oriented tables. For analytical workloads (OLAP), use column-oriented tables. Transactional workloads are characterized by a high rate of queries affecting a small number of rows each. Analytical workloads are characterized by processing large volumes of data to produce relatively small query results.
- **How is the table modified?** As a rule of thumb, row-oriented tables work better when data is frequently modified in place, while column-oriented tables work better when data is mostly appended by adding new rows. Thus, row-oriented tables usually reflect the current state of a dataset, while column-oriented tables often store a history of some sort of immutable events.
- **Which features are needed?** Even though YDB strives for feature parity between row-oriented and column-oriented tables, there might be current limitations to consider. Check the documentation for details on specific features intended to be used with a given table.

Unlike most other database management systems, YDB supports both row-oriented and column-oriented tables in the same database. However, keep in mind that transactional and analytical workloads have different resource consumption patterns and might affect each other when the cluster is overloaded.

Known issues

Table partitions are not merging when autopartitioning is enabled

For tables created in earlier versions of YDB, the autopartitioning mechanism does not merge partitions even when merge conditions based on load or size are met. This can result in an excessive number of partitions in the table.

This issue is specific to tables that meet the following two criteria:

- The table was created on a YDB cluster prior to version 22.2.
- The table had no minimum partition count value set, either at creation time or afterwards.

To resolve the issue, explicitly set a minimum partition count value for the table. For example, use the AUTO_PARTITIONING_MIN_PARTITIONS_COUNT parameter:

```
ALTER TABLE `my_table` SET (AUTO_PARTITIONING_MIN_PARTITIONS_COUNT = 1);
```

All questions on one page

General questions

What is YDB?

YDB is a distributed fault-tolerant SQL DBMS. YDB offers high availability and scalability, while ensuring strong consistency and support for ACID transactions. Queries are made using an SQL dialect (YQL).

YDB is a fully managed database. DB instances are created through the YDB database management service.

What features does YDB provide?

YDB provides high availability and data security through synchronous replication in three availability zones. YDB also ensures even load distribution across available hardware resources. This means you don't need to order resources, YDB automatically provisions and releases resources based on the user load.

What consistency model does YDB use?

To read data, YDB uses a model of strict data consistency.

How do I design a primary key?

To design a primary key properly, follow the rules below.

- Avoid situations where most of the load falls on a single partition of a table. With even load distribution, it's easier to achieve high overall performance.This rule implies that you shouldn't use a monotonically increasing sequence, such as timestamp, as a table's primary key.
- The fewer table partitions a query uses, the faster it runs. For greater performance, follow the one query — one partition rule.
- Avoid situations where a small part of the DB is under much heavier load than the rest of the DB.

For more information, see choosing a primary key.

How do I evenly distribute load across table partitions?

You can use the following techniques to distribute the load evenly across table partitions and increase overall DB performance.

- To avoid using uniformly increasing primary key values, you can:
  - Change the order of its components.
  - use a hash of the key column values as the primary key.
- Reduce the number of partitions used in a single query.

For more information, see choosing a primary key.

Can I use NULL in a key column?

In YDB, all columns, including key ones, may contain a `NULL` value, but we don't recommend using `NULL` as values in key columns.

Per the SQL standard (ISO/IEC 9075), you can't compare `NULL` with other values. Therefore, the use of concise SQL statements with simple comparison operators may result in rows containing NULL being skipped during filtering, for example.

Is there an optimal size of a database row?

To achieve high performance, we don't recommend writing rows larger than 8 MB and key columns larger than 2 KB to the DB.

For more information about limits, see Database limits.

How are secondary indexes used in YDB?

Secondary indexes in YDB are global and can be non-unique.

For more information, see Secondary indexes.

How are paginated results printed?

To print paginated results, we recommend selecting data sorted by primary key sequentially, limiting the number of rows with the `LIMIT` keyword. We do not recommend using the `OFFSET` keyword to solve this problem.

For more information, see Paginated results.

How do I delete expired data?

To efficiently delete outdated data, we recommend using TTL.

Syncing two data centers in geographically distributed clusters

The lead tablet writes data to a distributed network storage that saves copies to several data centers. YDB does not commit a user query until after the required number of copies are saved to the required number of data centers.

SDK

What should I do if the SDK crashes when shutting down an application?

Make sure not to wrap SDK components in a singleton, since their lifetime shouldn't exceed the execution time of the `main()` function. When a client is destroyed, session pools are emptied, so network navigation is required. But gRPC contains global static variables that might already be destroyed by this time. This disables gRPC. If you need to declare a driver as a global object, invoke the `Stop(true)` function on the driver before exiting the `main()` function.

What should I do if, when using a `fork()` system call, a program does not work properly in a child process?

Using `fork()` in multithreaded applications is an antipattern. Since both the SDK and the gRPC library are multithreaded applications, their stability is not guaranteed.

What do I do if I get the "Active sessions limit exceeded" error even though the current number of active sessions is within limits?

The limit applies to the number of active sessions. An active session is a session passed to the client to be used in its code. A session is returned to the pool in a destructor. In this case, the session itself is a replicated object. You may have saved a copy of the session in the code.

Is it possible to make queries to different databases from the same application?

Yes, the C++ SDK lets you override the DB parameters and token when creating a client. There is no need to create separate drivers.

What should I do if a VM has failed and it's impossible to make a query?

To detect that a VM is unavailable, set a client timeout. All queries contain the client timeout parameters. The timeout value should be an order of magnitude greater than the expected query execution time.

Errors

Possible causes for "Status: OVERLOADED Error: Pending previous query completion" in the C++ SDK

Q: When running two queries, I try to get a response from the future method of the second one. It returns: `Status: OVERLOADED Why: <main>: Error: Pending previous query completion`.

A: Sessions in the SDK are single-threaded. To run multiple queries at once, you need to create multiple sessions.

What do I do if I frequently get the "Transaction locks invalidated" error?

Typically, if you get this error, repeat a transaction, as YDB uses optimistic locking. If this error occurs frequently, this is the result of a transaction reading a large number of rows or of many transactions competing for the same "hot" rows. It makes sense to view the queries running in the transaction and check if they're reading unnecessary rows.

What causes the "Exceeded maximum allowed number of active transactions" error?

The logic on the client side should try to keep transactions as short as possible.

No more than 10 active transactions are allowed per session. When starting a transaction, use either the commit flag for autocommit or an explicit commit/rollback.

What do I do if I get the Datashard: Reply size limit exceeded error in response to a query?

This error means that, as a query was running, one of the participating data shards attempted to return over 50 MB of data, which exceeds the allowed limit.

Recommendations:

- A general recommendation is to reduce the amount of data processed in a transaction.
- If a query involves a `Join`, it's a good idea to make sure that the method used is Index lookup Join.
- If a simple selection is performed, make sure that it is done by keys, or add `LIMIT` in the query.

What do I do is I get the "Datashard program size limit exceeded" in response to a query?

This error means that the size of a program (including parameter values) exceeded the 50-MB limit for one of the data shards. In most cases, this indicates an attempt to write over 50 MB of data to database tables in a single transaction. All modifying operations in a transaction such as `UPSERT`, `REPLACE`, `INSERT`, or `UPDATE` count as records.

You need to reduce the total size of records in one transaction. Normally, we don't recommend combining queries that logically don't require transactionality in a single transaction. When adding/updating data in batches, we recommend reducing the size of one batch to values not exceeding a few megabytes.

YQL

General questions

How do I select table rows by a list of keys?

You can select table rows based on a specified list of table primary key (or key prefix) values using the `IN` operator:

```
DECLARE $keys AS List<UInt64>;

SELECT * FROM some_table
WHERE Key1 IN $keys;
```

If a selection is made using a composite key, the query parameter must have the type of a list of tuples:

```
DECLARE $keys AS List<Tuple<UInt64, String>>;

SELECT * FROM some_table
WHERE (Key1, Key2) IN $keys;
```

To select rows effectively, make sure that the value types in the parameters match the key column types in the table.

Is search by index performed for conditions containing the LIKE operator?

You can only use the `LIKE` operator to search a table index if it specifies a row prefix:

```
SELECT * FROM string_key_table
WHERE Key LIKE "some_prefix%";
```

Why does a query return only 1000 rows?

1000 rows is the response size limit per YQL query. If a response is shortened, it is flagged as `Truncated`. To output more table rows, you can use paginated output or the `ReadTable` operation.

How to escape quotes of JSON strings when adding them to a table?

Consider an example with two possible options for adding a JSON string to a table:

```
UPSERT INTO test_json(id, json_string)
VALUES
    (1, Json(@@[{"name":"Peter \"strong cat\" Kourbatov"}]@@)),
    (2, Json('[{"name":"Peter \\\"strong cat\\\" Kourbatov"}]'))
;
```

To insert a value in the first line, use `raw string` and the escape method using `\"`. To insert the second line, escaping through `\\\"` is used.

We recommend using `raw string` and the escape method using `\"`, as it is more visual.

How do I update a JSON value in a table?

Partial updates of a JSON field are not supported, so it is necessary to construct a new value from parts of the old and new data. This is done using the `Yson` UDF combined with dictionary functions.

The example below adds a new field `new_field` to the JSON object stored in the `column` column:

```
UPDATE table
SET column = Yson::SerializeJson(
    Yson::From(
        SetUnion(
            Yson::ConvertTo(Yson::ParseJson(column), Dict<String, Yson>),
            {'new_field': Yson('"value"')},
            ($K, $v1, $v2) -> { RETURN COALESCE($v2, $v1); }
        )
    )
)
WHERE id = 1;
```

How do I update only those values whose keys are not in the table?

You can use the `LEFT JOIN` operator to identify the keys a table is missing and update their values:

```
DECLARE $values AS List<Struct<Key: UInt64, Value: String>>;

UPSERT INTO kv_table
SELECT v.Key AS Key, v.Value AS Value
FROM AS_TABLE($values) AS v
LEFT JOIN kv_table AS t
ON v.Key = t.Key
WHERE t.Key IS NULL;
```

Join operations

Are there any specific features of Join operations?

A `Join` in YDB is performed using one of the two methods below:

- Common Join.
- Index Lookup Join.

Common Join

The contents of both tables (to the left and to the right of `Join`) are sent to the requesting node which applies the operation to the totality of the data. This is the most generic way of performing a `Join` that is used whenever other optimizations are unavailable. For large tables, this method is either slow or doesn't work in general due to exceeding the data transfer limits.

Index Lookup Join

For rows on the left of `Join`, relevant values are looked up to the right. You use this method whenever the right part is a table and the `Join` key is its primary or secondary index key prefix. In this method, limited selections are made from the right table instead of full reads. This lets you use it when working with large tables.

> **Note**
>
> For most OLTP queries, we recommend using Index Lookup Join with a small size of the left part. These operations read little data and can be performed efficiently.

How do I Join data from query parameters?

You can use query parameter data as a constant table. To do this, use the `AS_TABLE` modifier with a parameter whose type is a list of structures:

```
DECLARE $data AS List<Struct<Key1: UInt64, Key2: String>>;

SELECT * FROM AS_TABLE($data) AS d
```

```
INNER JOIN some_table AS t
ON t.Key1 = d.Key1 AND t.Key2 = d.Key2;
```

There is no explicit limit on the number of entries in the constant table, but mind the standard limit on the total size of query parameters (50 MB).

What's the best way to implement a query like (key1, key2) IN ((v1, v2), (v3, v4), ...)?

You can use the `WHERE ... IN` syntax to query by a set of keys (including composite keys):

```
DECLARE $key_pairs AS List<Tuple<Uint64, Uint64>>;

SELECT * FROM some_table
WHERE (Key1, Key2) IN $key_pairs;
```

Transactions

How efficient is it to run multiple queries in a transaction?

When multiple queries are run sequentially, the total transaction latency may be greater than when the same operations are executed within a single query. This is primarily due to additional network latency for each query. Therefore, if a transaction doesn't need to be interactive, we recommend formulating all operations in a single YQL query.

Is a separate query atomic?

In general, YQL queries can be executed in multiple consecutive phases. For example, a Join query can be executed in two phases: reading data from the left and right table, respectively. This aspect is important when you run a query in a transaction with a low isolation level (`online_read_only`), as in this case, data between execution phases can be updated by other transactions.

Analytics

Can YDB be used for analytical workloads (OLAP)?

Yes, it can. If this is the primary type of workload for a given table, make sure it is column-oriented.

How to choose between row-oriented and column-oriented tables?

Similarly to choosing between transactional (OLTP) and analytical (OLAP) database management systems, this question comes to a number of trade-offs that need to be considered:

- **What's the main use case for the table?** For mostly transactional (OLTP) workloads, use row-oriented tables. For analytical workloads (OLAP), use column-oriented tables. Transactional workloads are characterized by a high rate of queries affecting a small number of rows each. Analytical workloads are characterized by processing large volumes of data to produce relatively small query results.
- **How is the table modified?** As a rule of thumb, row-oriented tables work better when data is frequently modified in place, while column-oriented tables work better when data is mostly appended by adding new rows. Thus, row-oriented tables usually reflect the current state of a dataset, while column-oriented tables often store a history of some sort of immutable events.
- **Which features are needed?** Even though YDB strives for feature parity between row-oriented and column-oriented tables, there might be current limitations to consider. Check the documentation for details on specific features intended to be used with a given table.

Unlike most other database management systems, YDB supports both row-oriented and column-oriented tables in the same database. However, keep in mind that transactional and analytical workloads have different resource consumption patterns and might affect each other when the cluster is overloaded.

Known issues

Table partitions are not merging when autopartitioning is enabled

For tables created in earlier versions of YDB, the autopartitioning mechanism does not merge partitions even when merge conditions based on load or size are met. This can result in an excessive number of partitions in the table.

This issue is specific to tables that meet the following two criteria:

- The table was created on a YDB cluster prior to version 22.2.
- The table had no minimum partition count value set, either at creation time or afterwards.

To resolve the issue, explicitly set a minimum partition count value for the table. For example, use the AUTO_PARTITIONING_MIN_PARTITIONS_COUNT parameter:

```
ALTER TABLE `my_table` SET (AUTO_PARTITIONING_MIN_PARTITIONS_COUNT = 1);
```

SDK

What should I do if the SDK crashes when shutting down an application?

Make sure not to wrap SDK components in a singleton, since their lifetime shouldn't exceed the execution time of the `main()` function. When a client is destroyed, session pools are emptied, so network navigation is required. But gRPC contains global static variables that might already be destroyed by this time. This disables gRPC. If you need to declare a driver as a global object, invoke the `Stop(true)` function on the driver before exiting the `main()` function.

What should I do if, when using a `fork()` system call, a program does not work properly in a child process?

Using `fork()` in multithreaded applications is an antipattern. Since both the SDK and the gRPC library are multithreaded applications, their stability is not guaranteed.

What do I do if I get the "Active sessions limit exceeded" error even though the current number of active sessions is within limits?

The limit applies to the number of active sessions. An active session is a session passed to the client to be used in its code. A session is returned to the pool in a destructor. In this case, the session itself is a replicated object. You may have saved a copy of the session in the code.

Is it possible to make queries to different databases from the same application?

Yes, the C++ SDK lets you override the DB parameters and token when creating a client. There is no need to create separate drivers.

What should I do if a VM has failed and it's impossible to make a query?

To detect that a VM is unavailable, set a client timeout. All queries contain the client timeout parameters. The timeout value should be an order of magnitude greater than the expected query execution time.

Videos

This section contains video recordings from conferences and webinars:

- Videos 2025
- Videos 2024
- Videos 2023
- Videos 2022

The materials are divided by categories and tagged:

Overview	– overview materials that introduce YDB and the technologies used in it.
Use cases	– use cases of YDB.
Practice	– best practices for using YDB.
Database internals	– a detailed analysis of the internal implementation of YDB or its individual parts and mechanisms.
Releases	– an overview of new features and released versions of YDB.
Testing	– performance testing cases of YDB and comparisons with other similar-class DBMSs.
General	– generic materials.

Videos 2025

Implementing an MCP Server for DBMS in Python — YDB's Experience

Database internals

Discover how YDB leverages the Model Context Protocol (MCP) to bridge AI models with database systems. Ivan Blinkov (VP, Product and Open Source) shares the story of the development of a Python-based MCP server that enables seamless interactions between large language models and YDB's open-source Distributed DBMS. Learn about the design decisions, challenges faced, and the practical benefits of integrating MCP into data architecture. Ideal for developers and AI practitioners aiming to enhance AI capabilities with direct database access.

The presentation will be of interest to developers who want to integrate their products with AI via the MCP protocol.

Slides

Designing YDB: Constructing a Distributed cloud-native DBMS for OLTP and OLAP from the Ground Up

Database internals

Distributed systems offer multiple advantages: they are built to be fault-tolerant and reliable, can scale almost infinitely, provide low latency in geo-distributed scenarios, and, finally, are geeky and fun to explore. YDB is an open-source distributed SQL database that has been running in production for years. Some installations include thousands of servers storing petabytes of data. To provide these capabilities, any distributed DBMS must achieve consistency and consensus while tolerating unreliable networks, faulty hardware, and the absence of a global clock.

In this session, we will provide a gentle introduction to the problems, challenges, and fallacies of distributed computing, explaining why sharded systems like Citus are not always ACID-compliant and how they differ from truly distributed systems. Then, we will dive deep into the design decisions made by YDB to address these difficulties and outline YDB's architecture layer by layer: from bare metal disks and distributed storage to OLTP and OLAP functionalities. Finally, we will briefly compare our approach with that of Calvin, which originally inspired YDB, and Spanner.

Evgenii Ivanov (Senior developer) discussed the architecture of YDB, focusing on building a unified platform for fault-tolerant and reliable OLTP and OLAP processing.

The presentation will be of interest to developers of high-load systems and platform developers for various purposes.

Slides

Designing YDB: Constructing a Distributed cloud-native DBMS for OLTP and OLAP from the Ground Up

Database internals

Distributed systems are great in multiple aspects: they are built to be fault-tolerant and reliable, can scale almost infinitely, provide low latency in geo-distributed scenarios, and, finally, they are geeky and fun to explore. YDB is a distributed SQL database that has been running in production for years. There are installations with thousands of servers storing petabytes of data. To provide these capabilities, any distributed DBMS must achieve consistency and consensus while tolerating unreliable networks, faulty hardware, and the absence of a global clock.

In this session, we will briefly introduce the problems, challenges, and fallacies of distributed computing, explaining why sharded systems like Citus are not always ACID and differ from truly distributed systems. Then, we will dive deep into the design decisions made by YDB to address these difficulties and outline YDB's architecture layer by layer, from the bare metal disks and distributed storage up to OLTP and OLAP functionalities. Ultimately, we will briefly compare our approach with Calvin's, which initially inspired YDB, and Spanner.

Evgenii Ivanov (Senior developer) discussed the architecture of YDB, focusing on building a unified platform for fault-tolerant and reliable OLTP and OLAP processing.

The presentation will be of interest to developers of high-load systems and platform developers for various purposes.

Slides

YDB: How to implement streaming RAG in a distributed database

Database internals

Extracting real-time insights from multi-modal data streams across diverse domains presents an ongoing challenge. A promising solution lies in the implementation of Streaming Retrieval-Augmented Generation (RAG) techniques. YDB enhances this approach by offering robust services for both vector search and streaming, facilitating more efficient and effective data processing and retrieval. YDB is a versatile open-source Distributed SQL Database that combines high availability and scalability with strong consistency and ACID transactions.

Alexander Zevaykin (Team leader) and Elena Kalinina (Technical Project Manager) discussed an approach to implementing streaming RAG in YDB.

The presentation will be of interest to developers of high-load systems and platform developers for various purposes.

Slides

Sharded and Distributed Are Not the Same: What You Must Know When PostgreSQL Is Not Enough

Testing

It's no secret that PostgreSQL is extremely efficient and scales vertically well. At the same time, it isn't a secret that PostgreSQL scales only vertically, meaning its performance is limited by the capabilities of a single server. Most Citus-like solutions allow the database to be sharded, but a sharded database is not distributed and does not provide ACID guarantees for distributed transactions. The common opinion about distributed DBMSs is diametrically opposed: they are believed to scale well horizontally and have ACID distributed transactions but have lower efficiency in smaller installations.

When comparing monolithic and distributed DBMSs, discussions often focus on architecture but rarely provide specific performance metrics. This presentation, on the other hand, is entirely based on an empirical study of this issue. Our approach is simple: Evgenii Ivanov (Senior developer) installed PostgreSQL and distributed DBMSs on identical clusters of three physical servers and compared them using the popular TPC-C benchmark.

The presentation will be of interest to developers of high-load systems and platform developers for various purposes.

Slides

Videos 2024

Enhancing a Distributed SQL Database Engine: A Case Study on Performance Optimization

Database internals

Learn how we optimized a distributed SQL database engine, focusing on benchmark-driven improvements, and pivotal testing strategies. Alexey Ozeritskiy (Lead Software Engineer) will talk about performance optimization of distributed SQL engine. He will discuss background information about YDB engine itself and where it is used. The final part of his talk will be about containerization and performance.

The presentation is suitable for DBA.

Slides

YDB: extending a Distributed SQL DBMS with PostgreSQL compatibility

Database internals

PostgreSQL is an implementation of SQL standard with one of the most vibrant ecosystems around it. To leverage all the tools and libraries that already know how to work with PostgreSQL, emerging database management systems that bring something new to the market need to learn how to mimic PostgreSQL. In this talk at COSCUP 2024 Ivan Blinkov (VP, Product and Open Source) explores possible approaches to this and related trade-offs, as well as reasoning why YDB chose a unique approach to bring serializable consistency and seamless scalability to the PostgreSQL ecosystem.

> **Note**
>
> The video will be available later.

The presentation is suitable for people interested in trade-offs during implementation of PostgreSQL-compatible DBMS.

Slides

Breaking out of the cage: move complex development to GitHub

General

Alexander Smirnov (Technology Expert at Nebius) shows how the YDB team moved its primary development branch from an in-house repository to GitHub, set up independent commodity on-demand cloud infrastructure, CI processes with GitHub Actions, test management with open source and cloud tools. Special attention will be paid to the complexities of decoupling from the corporate monorepository and build system.

The presentation is suitable for DevOps engineers (CI/CD).

Slides

Introducing YDB, a Distributed SQL DBMS for mission-critical workloads

Overview

YDB is a versatile open-source Distributed SQL Database that combines high availability and scalability with strong consistency and ACID transactions. It accommodates transactional (OLTP), analytical (OLAP), and streaming workloads simultaneously. It is publicly available under Apache 2.0, one of the most permissive open-source licenses. In this talk at IndiaFOSS 2024, Ivan Blinkov (VP, Product and Open Source) introduces the system and explains how it can be used to build reliable data-driven applications that implement business-critical processes.

Slides

Working with Raw Disk Drives in Kubernetes — YDB's Experience | 在Kubernetes中使用原始磁盘驱动器——YDB的经验

Database internals

YDB is an open-source distributed database management system that, for performance reasons, uses raw disk drives (block devices) to store all data without any filesystem. It was relatively straightforward to manage such a setup in the bare-metal world of the past, but the dynamic nature of cloud-native environments introduced new challenges to keep this performance benefit. In this talk at KubeCon + CloudNativeCon + Open Source Summit Hong Kong, Ivan Blinkov (VP, Product and Open Source) explores how to leverage Kubernetes and the Operator design pattern to modernize how stateful distributed database clusters are managed without changing the primary approach to how the data is physically stored.

YDB是一个开源的分布式数据库管理系统，为了性能考虑，使用原始磁盘驱动器（块设备）存储所有数据，而不使用任何文件系统。在过去的裸金属世界中管理这样的设置相对比较简单，但云原生环境的动态特性引入了新的挑战，以保持这种性能优势。在这次演讲中，我们将探讨如何利用Kubernetes和运算符设计模式来现代化管理有状态的分布式数据库集群，而不改变数据物理存储的主要方法。

Slides

YDB: dealing with Big Data and moving towards AI

General

Alexander Zevaykin (Team leader) talks about YDB — a versatile, open-source Distributed SQL database management system that combines high availability and scalability with strong consistency and ACID transactions. It provides services for machine learning products and goes beyond traditional vector search capabilities.

> **Note**
>
> The video will be available later.

This database is used for industrial operations within Yandex. Among its clients are Yandex Market, Yandex Alice, and Yandex Taxi, which are some of the largest and most demanding AI-based applications.

The database offers true elastic scalability, capable of scaling up or down by several orders of magnitude.

Simultaneously, the database is fault-tolerant. It is designed to operate across three availability zones, ensuring continuous operation even if one of the zones becomes unavailable. The database automatically recovers from disk failures, server failures, or data center failures, with minimal latency disruptions to applications.

Currently, work is underway to implement accurate and approximate nearest neighbour search for machine learning purposes.

Takeaways:

- Architecture of a distributed, fault-tolerant database.
- Approaches to implementing vector search on large datasets.

Slides

An approach to unite tables and persistent queues in one system

General Database internals

People need databases to store their data and persistent queues to transfer their data from one system to another. We've united tables and persisted queues within one data platform. Now you have a possibility to take your data from a queue, then process it and keep the result in a database within a single transaction. So your application developers don't need to think about data inconsistency in cases of connection failures or other errors.

Elena Kalinina (Technical Project Manager) tell you about an open-source platform called YDB which allows you to work with tables and queues within a single transaction. Elena walk you through architecture decisions, possible scenarios, and performance aspects of this approach.

Slides

YDB vs. TPC-C: the Good, the Bad, and the Ugly behind High-Performance Benchmarking

Database internals

Modern distributed databases scale horizontally with great efficiency, making them almost limitless in capacity. This implies that benchmarks should be able to run on multiple machines and be very efficient to minimize the number of machines required. This talk will focus on benchmarking high-performance databases, particularly emphasizing YDB and our implementation of the TPC-C benchmark, the de facto gold standard in the database field.

First, we will speak about benchmarking strategies from a user's perspective. We will dive into key details related to benchmark implementations, which could be useful when you create a custom benchmark to mirror your production scenarios. Throughout our performance journey, we have identified numerous anti-patterns: there are things you should unequivocally avoid in your benchmark implementations. We'll highlight these "bad" and "ugly" practices with illustrative examples.

Next, we'll briefly discuss the popular key-value benchmark YCSB, which is a prerequisite for robust performance in distributed transactions. We'll then explore the TPC-C benchmark in greater detail, sharing valuable insights derived from our own implementation.

We'll conclude our talk by presenting performance results from the TPC-C benchmark, comparing YDB and CockroachDB with PostgreSQL to illustrate situations where PostgreSQL might not be enough and when you might want to consider a distributed DBMS instead.

Evgenii Ivanov (Senior developer) discussed best high-performance benchmarking practices and some pitfalls found during TPC-C implementation, then demonstrated TPC-C results of PostgreSQL, CockroachDB, and YDB.

The presentation will be of interest to developers of high-load systems and developers of platforms for various purposes.

Slides

Videos 2023

YDB — an open-source distributed SQL database

Overview

YDB is used as a mission-critical database for many Internet-scale services. YDB has been designed as a platform for various data storage and processing systems and is aimed at solving a wide range of problems. Oleg Bondar (CPO YDB) spoke about the structure of YDB, its main features, and benefits.

The presentation is suitable for everyone who is not yet familiar with YDB.

Slides

Scale it easy: YDB's high performance in a nutshell

Database internals

Implementing a distributed database with strong consistency isn't difficult; ensuring speed and scalability is the challenge. YDB excels in these aspects. In this talk, we'll discuss YDB's architecture and high performance, present benchmark results, and compare YDB to top competitors.

Evgenii Ivanov (Senior developer) discussed the architecture of YDB, demonstrated its high performance through benchmark results, and compared YDB with its competitors.

The presentation will be of interest to developers of high-load systems and developers of platforms for various purposes.

Slides

YDB — a Distributed SQL Database

Overview

This is a recording of a guest lecture in Belgrade University at the faculty of Mathematics. In this video, Anton Kovalenko (Technical Project Manager) describes the reasons why distributed SQL databases were created. He illustrates a brief history of Distributed SQL DBMS development, which products have appeared first.

Slides

Videos 2022

Parallel asynchronous replication between YDB database instances

Database internals

In this talk, we present an approach to asynchronous replication in YDB that provides the following characteristics: changefeed from the source database is sharded among multiple persistent queues, sharded changefeed is applied to the target database in a manner that guarantees the target database consistency.

Slides

Scalability and Fault Tolerance in YDB

In this talk, we will cover two layers of YDB: Tablet and BlobStorage, which together provide fault tolerance, scalability, and user isolation.

Articles

This section contains articles about YDB:

- Articles 2024
- Articles 2023

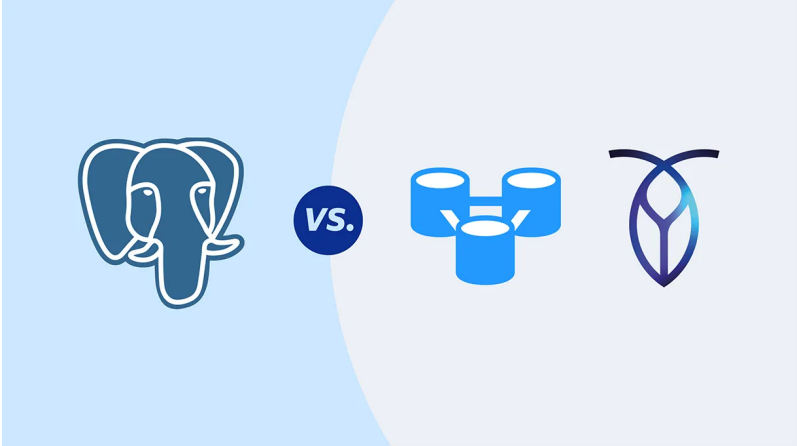
The materials are divided by categories and tagged:

Overview	– overview materials that introduce YDB and the technologies used in it.
Use cases	– use cases of YDB.
Practice	– best practices for using YDB.
Database internals	– a detailed analysis of the internal implementation of YDB or its individual parts and mechanisms.
Releases	– an overview of new features and released versions of YDB.
Testing	– performance testing cases of YDB and comparisons with other similar-class DBMSs.
General	– generic materials.

Articles 2024

When Postgres is not enough: performance evaluation of PostgreSQL vs. Distributed DBMSs

Database internals



The research presented is the result of our joint effort and close collaboration with Evgeny Efimkin, an expert in PostgreSQL who doesn't work on YDB.

How we switched to Java 21 virtual threads and got a deadlock in TPC-C for PostgreSQL

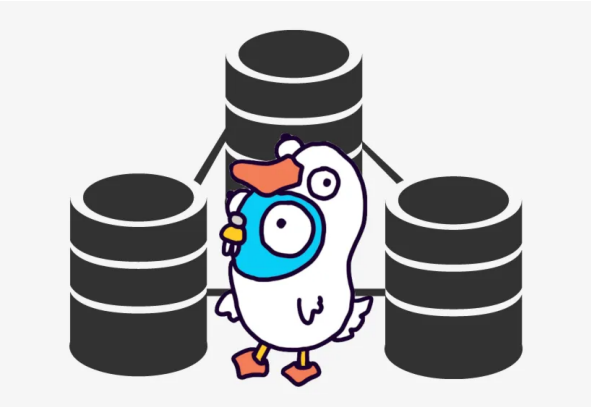
Database internals

In this post, we present a case study on how we encountered a deadlock with virtual threads in TPC-C for PostgreSQL, even without the dining philosophers problem. This post might be helpful for Java developers who are considering switching to virtual threads.

Articles 2023

Migrations in YDB using “goose”

Database internals



Any production process that works with a database will require a schema migration sooner or later. The migration updates the database’s table structure from one version to the next. Schema migrations can be done manually by executing an `ALTER TABLE` query or by using specialized tools. One such tool is called goose. In this article we see how goose provides schema management in a project and has supported YDB (a distributed open-source database) since v3.16.0.

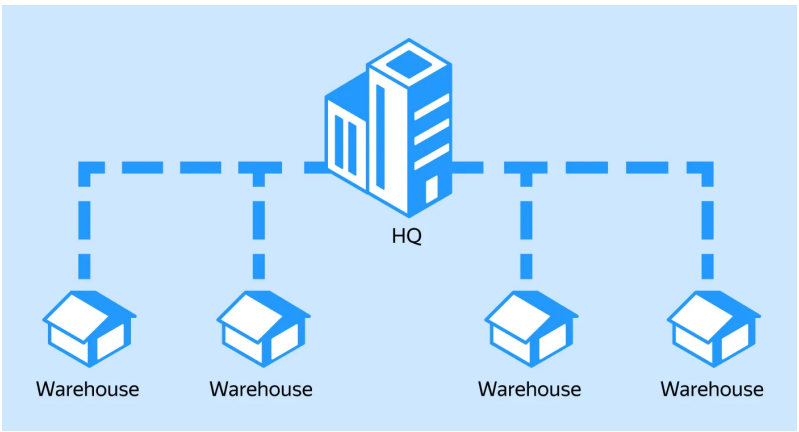
About prepared statements, server-side compiled query cache, or how to efficiently cache queries in YDB

Database internals

There are various ways to reduce the cost of SQL query execution in modern DBMS. The most common approaches are using prepared statements and query caching. Both methods are available in YDB. Their functionality and benefits are discussed in this article.

YDB meets TPC-C: distributed transactions performance now revealed

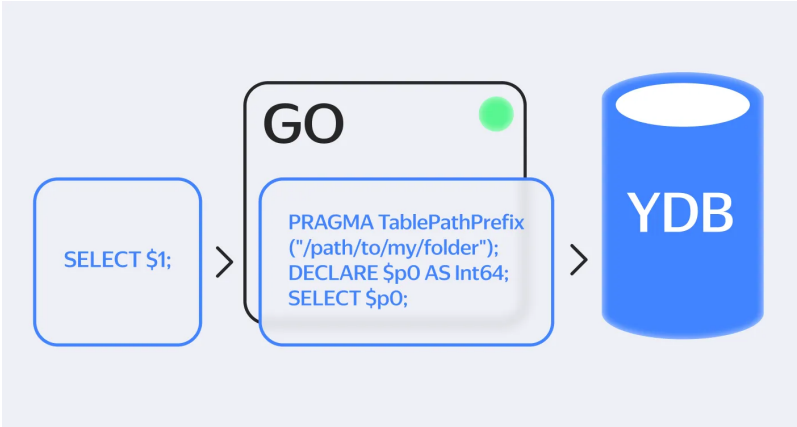
Database internals



We are excited to present our first results of TPC-C*, which is industry-standard On-Line Transaction Processing (OLTP) benchmark. According to these results, there are scenarios in which YDB slightly outperforms CockroachDB, another trusted and well-known distributed SQL database.

database/sql bindings for YDB in Go

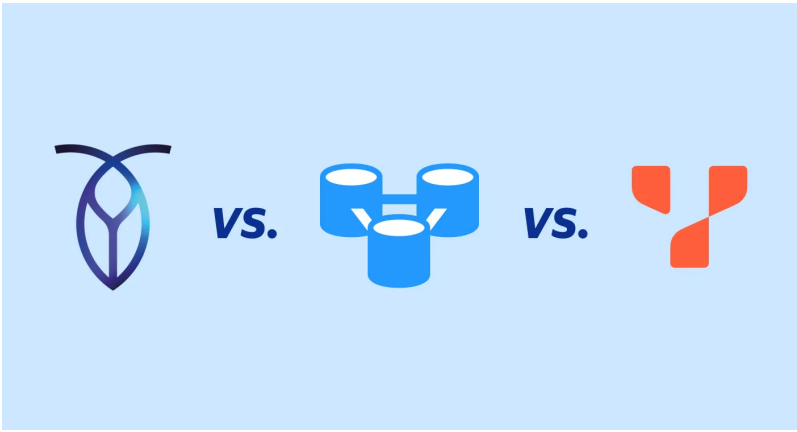
Database internals



YQL is a SQL dialect with YDB specific strict types. This is great for performance and correctness, but sometimes can be a bit daunting to express in a query, especially when they need to be parametrized externally from the application side. For instance, when a YDB query needs to be parametrized, each parameter has name and type provided via `DECLARE` statement. To explore more about this and see practical examples, read the detailed explanation in this article.

YCSB performance series: YDB, CockroachDB, and YugabyteDB

Database internals



It's a challenge to implement a distributed database with strong consistency, ensuring high speed and scalability. YDB excels in these aspects, and our customers can attest to this through their own experiences. Unfortunately, we have never presented any performance numbers to a broader audience. We recognize the value of this information, and we are preparing more benchmark results to share.

YDB Downloads

This section provides instructions for downloading various YDB builds and related tools:

- YDB server (`ydbd`):
  - YDB Open-Source Database
  - Yandex Enterprise Database
- Command-line client utility (`ydb`)
- Disk substem management utility (`ydb-dstool`)
- Cluster management utility (`ydbops`)
- Ansible playbooks

Download YDB Open-Source Database

YDB Open-Source Database (`ydbd`) is an executable file for running a node in the YDB cluster. It is distributed under the Apache 2.0 license.

See also Download Yandex Enterprise Database.

Linux

Version	Release date	Download	Changelog
v26.3			
v26.3.1.16-rc	18.09.26	Binary file	See list
v26.2			
v.26.2.1.14	16.09.26	Binary file	See list
v26.1			
v.26.1.1.22	27.07.26	Binary file	See list
v.26.1.1.20	02.07.26	Binary file	See list
v25.4			
v.25.4.1.15	24.04.26	Binary file	See list
v25.3			
v.25.3.1.27	20.05.26	Binary file	See list
v25.2			
v.25.2.1.26	12.05.26	Binary file	See list
v.25.2.1.10-rc	21.09.25	Binary file	See list
v25.1			
v.25.1.4.18	12.05.26	Binary file	See list
v.25.1.2.7-rc	14.07.25	Binary file	See list
v24.4			
v.24.4.4.12	03.06.25	Binary file	See list
v.24.4.4.2	15.04.25	Binary file	See list
v24.3			
v.24.3.15.5	06.02.25	Binary file	See list
v.24.3.11.14	09.01.25	Binary file	See list
v.24.3.11.13	24.12.24	Binary file	See list
v24.2			
v.24.2.7	20.08.24	Binary file	See list
v24.1			
v.24.1.18	31.07.24	Binary file	See list
v23.4			
v.23.4.11	14.05.24	Binary file	See list
v23.3			
v.23.3.17	14.12.23	Binary file	
v.23.3.13	12.10.23	Binary file	See list
v23.2			
v.23.2.12	14.08.23	Binary file	See list

Docker

Version	Release date	Docker `registry/image:tag`	Changelog
v26.3			
v26.3.1.16-rc	18.09.26	`cr.yandex/crptqonuodf51kdj7a7d/ydb:26.3.1.16-rc`	See list
v26.2			
v.26.2.1.14	16.09.26	`cr.yandex/crptqonuodf51kdj7a7d/ydb:26.2.1.14`	See list

v26.1			
v.26.1.1.22	27.07.26	`cr.yandex/crptqonuodf51kdj7a7d/ydb:26.1.1.22`	See list
v.26.1.1.20	02.07.26	`cr.yandex/crptqonuodf51kdj7a7d/ydb:26.1.1.20`	See list
v25.4			
v.25.4.1.15	24.04.26	`cr.yandex/crptqonuodf51kdj7a7d/ydb:25.4.1.15`	See list
v25.3			
v.25.3.1.27	20.05.26	`cr.yandex/crptqonuodf51kdj7a7d/ydb:25.3.1.27`	See list
v25.2			
v.25.2.1.26	12.05.26	`cr.yandex/crptqonuodf51kdj7a7d/ydb:25.2.1.26`	See list
v.25.2.1.10	21.09.25	`cr.yandex/crptqonuodf51kdj7a7d/ydb:25.2.1.10-rc`	See list
v25.1			
v.25.1.4.18	12.05.26	`cr.yandex/crptqonuodf51kdj7a7d/ydb:25.1.4.18`	See list
v.25.1.2.7-rc	14.07.25	`cr.yandex/crptqonuodf51kdj7a7d/ydb:25.1.2.7-rc`	See list
v24.4			
v.24.4.4.12	03.06.25	`cr.yandex/crptqonuodf51kdj7a7d/ydb:24.4.4.12`	See list
v.24.4.4.2	15.04.25	`cr.yandex/crptqonuodf51kdj7a7d/ydb:24.4.4.2`	See list
v24.3			
v.24.3.15.5	06.02.25	`cr.yandex/crptqonuodf51kdj7a7d/ydb:24.3.15.5`	See list
v.24.3.11.14	09.01.25	`cr.yandex/crptqonuodf51kdj7a7d/ydb:24.3.11.14`	See list
v.24.3.11.13	24.12.24	`cr.yandex/crptqonuodf51kdj7a7d/ydb:24.3.11.13`	See list
v24.2			
v.24.2.7	20.08.24	`cr.yandex/crptqonuodf51kdj7a7d/ydb:24.2.7`	See list
v24.1			
v.24.1.18	31.07.24	`cr.yandex/crptqonuodf51kdj7a7d/ydb:24.1.18`	See list
v23.4			
v.23.4.11	14.05.24	`cr.yandex/crptqonuodf51kdj7a7d/ydb:23.4.11`	See list
v23.3			
v.23.3.17	14.12.23	`cr.yandex/crptqonuodf51kdj7a7d/ydb:23.3.17`	
v.23.3.13	12.10.23	`cr.yandex/crptqonuodf51kdj7a7d/ydb:23.3.13`	See list
v23.2			
v.23.2.12	14.08.23	`cr.yandex/crptqonuodf51kdj7a7d/ydb:23.2.12`	See list

Source Code

Version	Release date	Link	Changelog
v26.3			
v26.3.1.16-rc	18.09.26	https://github.com/ydb-platform/ydb/tree/26.3.1.16-rc	See list
v26.2			
v.26.2.1.14	16.09.26	https://github.com/ydb-platform/ydb/tree/26.2.1.14	See list
v26.1			
v.26.1.1.22	27.07.26	https://github.com/ydb-platform/ydb/tree/26.1.1.22	See list
v.26.1.1.20	02.07.26	https://github.com/ydb-platform/ydb/tree/26.1.1.20	See list
v25.4			
v.25.4.1.15	24.04.26	https://github.com/ydb-platform/ydb/tree/25.4.1.15	See list
v25.3			
v.25.3.1.27	20.05.26	https://github.com/ydb-platform/ydb/tree/25.3.1.27	See list
v25.2			
v.25.2.1.26	12.05.26	https://github.com/ydb-platform/ydb/tree/25.2.1.26	See list

v.25.2.1.10-rc	21.09.25	https://github.com/ydb-platform/ydb/tree/25.2.1.10-rc	See list
v25.1			
v.25.1.4.18	12.05.26	https://github.com/ydb-platform/ydb/tree/25.1.4.18	See list
v.25.1.2.7-rc	14.07.25	https://github.com/ydb-platform/ydb/tree/25.1.2.7-rc	See list
v24.4			
v.24.4.4.12	03.06.25	https://github.com/ydb-platform/ydb/tree/24.4.4.12	See list
v.24.4.4.2	15.04.25	https://github.com/ydb-platform/ydb/tree/24.4.4.2	See list
v24.3			
v.24.3.15.5	06.02.25	https://github.com/ydb-platform/ydb/tree/24.3.15.5	See list
v.24.3.11.14	09.01.25	https://github.com/ydb-platform/ydb/tree/24.3.11.14	See list
v.24.3.11.13	24.12.24	https://github.com/ydb-platform/ydb/tree/24.3.11.13	See list
v24.2			
v.24.2.7	20.08.24	https://github.com/ydb-platform/ydb/tree/24.2.7	See list
v24.1			
v.24.1.18	31.07.24	https://github.com/ydb-platform/ydb/tree/24.1.18	See list
v23.4			
v.23.4.11	14.05.24	https://github.com/ydb-platform/ydb/tree/23.4.11	See list
v23.3			
v.23.3.17	14.12.23	https://github.com/ydb-platform/ydb/tree/23.3.17	
v.23.3.13	12.10.23	https://github.com/ydb-platform/ydb/tree/23.3.13	See list
v23.2			
v.23.2.12	14.08.23	https://github.com/ydb-platform/ydb/tree/23.2.12	See list

Download Yandex Enterprise Database

Yandex Enterprise Database is a commercial database management system based on the YDB core.

Yandex Enterprise Database Usage Terms

Yandex Enterprise Database is distributed under the terms of the license agreement.

Free Usage

Yandex Enterprise Database (Product) can be used free of charge under the terms of the license agreement for the following purposes:

- To evaluate and study the Product's features
- As part of software development and testing processes that use or embed Product functions
- For training in Product administration
- For learning the specifics of developing software that interacts with the Product

Commercial Usage

For commercial use of Yandex Enterprise Database, a license purchase is required, license agreement text. License prices are available upon request.

Downloading Distributions

Yandex Enterprise Database distributions are available for download via the links below.

Linux

YDB Enterprise Server (`ydbd`) is an executable file for running a Yandex Enterprise Database node.

Version	Release date	Download	Checksums	Changelog
v26.1				
v.26.1.1.ent.3	03.08.2026	Distribution Debug symbols	For distribution For debug symbols	See list
v25.4				
v.25.4.1.ent.2	17.06.2026	Distribution Debug symbols	For distribution For debug symbols	See list
v25.3				
v.25.3.1.ent.3	11.06.2026	Distribution Debug symbols	For distribution For debug symbols	See list
v25.2				
v.25.2.1.ent.13	11.06.2026	Distribution Debug symbols	For distribution For debug symbols	See list
v.25.2.1.ent.4	12.02.2026	Distribution Debug symbols	For distribution For debug symbols	See list
v25.1				
v.25.1.4.ent.8	12.02.2026	Distribution Debug symbols	For distribution For debug symbols	See list
v.25.1.4.ent.3	25.11.2025	Distribution Debug symbols	For distribution For debug symbols	See list
v24.4				
v.24.4.4.20	01.11.2025	Distribution Debug symbols	For distribution For debug symbols	See list
v.24.4.4.15	19.09.2025	Distribution Debug symbols	For distribution For debug symbols	See list
v.24.4.4.13	29.07.2025	Distribution Debug symbols	For distribution For debug symbols	See list
v24.3				
v.24.3.13.11	06.03.2024	Distribution Debug symbols	For distribution For debug symbols	See list
v.24.3.13.10	24.12.2024	Distribution Debug symbols	For distribution For debug symbols	See list
v24.2				
v.24.2.7.1	20.08.2024	Distribution Debug symbols	For distribution For debug symbols	See list
v24.1				
v.24.1.18.1	28.06.2024	Distribution Debug symbols	For distribution For debug symbols	See list

v23.4				
v.23.4.11.1	15.05.2024	Distribution	Checksums	See list
v23.3				
v.23.3.25.2	17.03.2024	Distribution	Checksums	See list

Docker

Version	Release date	Download	Changelog	
v26.1				
v.26.1.1.ent.3	03.08.2026	`cr.yandex/crptqonuodf51kdj7a7d/ydb-enterprise:26.1.1.ent.3`	See list	
v25.4				
v.25.4.1.ent.2	17.06.2026	`cr.yandex/crptqonuodf51kdj7a7d/ydb-enterpise:25.4.1.ent.2`	See list	
v25.3				
v.25.3.1.ent.3	11.06.2026	`cr.yandex/crptqonuodf51kdj7a7d/ydb-enterpise:25.3.1.ent.3`	See list	
v25.2				
v.25.2.1.ent.13	11.06.2026	`cr.yandex/crptqonuodf51kdj7a7d/ydb-enterpise:25.2.1.ent.13`	See list	
v.25.2.1.ent.4	12.02.2026	`cr.yandex/crptqonuodf51kdj7a7d/ydb-enterpise:25.2.1.ent.4`	See list	
v25.1				
v.25.1.4.ent.8	12.02.2026	`cr.yandex/crptqonuodf51kdj7a7d/ydb-enterpise:25.1.4.ent.8`	See list	
v.25.1.4.ent.3	25.11.2025	`cr.yandex/crptqonuodf51kdj7a7d/ydb-enterpise:25.1.4.ent.3`	See list	
v24.4				
v.24.4.4.20	01.11.2025	`cr.yandex/crptqonuodf51kdj7a7d/ydb-enterpise:24.4.4.ent.20`	See list	
v.24.4.4.15	19.09.2025	`cr.yandex/crptqonuodf51kdj7a7d/ydb-enterpise:24.4.4.ent.15`	See list	
v.24.4.4.13	29.07.2025	`cr.yandex/crptqonuodf51kdj7a7d/ydb-enterpise:2025-08-08-73940688fea2aeb2e109f01a7827f2d7`	See list	
v24.3				
v.24.3.13	05.12.2024	`cr.yandex/crptqonuodf51kdj7a7d/ydb:24.3.11.13`	See list	
v24.2				
v.24.2.7	20.08.2024	`cr.yandex/crptqonuodf51kdj7a7d/ydb:24.2.7`	See list	
v24.1				
v.24.1.18	31.07.2024	`cr.yandex/crptqonuodf51kdj7a7d/ydb:24.1.18`	See list	
v23.4				
v.23.4.11	14.05.24	`cr.yandex/crptqonuodf51kdj7a7d/ydb:23.4.11`	See list	
v23.3				
v.23.3.17	14.12.23	`cr.yandex/crptqonuodf51kdj7a7d/ydb:23.3.17`	See list	
v.23.3.13	12.10.23	`cr.yandex/crptqonuodf51kdj7a7d/ydb:23.3.13`	See list	

Certified version

Certified versions of Yandex Enterprise Database are not available for public download. Access is provided upon request.

The certified version of Yandex Enterprise Database 25.1 has a technical support end date of 30 October 2030.

Download YDB CLI

YDB CLI (`ydb`) is a command-line utility for working with YDB databases. It is distributed under the Apache 2.0 license.

> **Tip**
>
> The recommended way to install YDB CLI is by using the installation script. The script automatically downloads the appropriate binary and adds it to your `PATH` .

Linux (amd64)

Version	Release date	Download	Changelog
v.2.33.0	23.06.2026	Binary file	See list
v.2.32.0	22.06.2026	Binary file	See list
v.2.31.0	20.04.2026	Binary file	See list
v.2.30.0	07.04.2026	Binary file	See list
v.2.29.0	11.02.2026	Binary file	See list
v.2.28.0	19.12.2025	Binary file	See list
v.2.27.0	30.10.2025	Binary file	See list
v.2.26.0	25.09.2025	Binary file	See list
v.2.25.0	01.09.2025	Binary file	See list
v.2.24.1	28.07.2025	Binary file	See list
v.2.24.0	23.07.2025	Binary file	See list
v.2.23.0	16.07.2025	Binary file	See list
v.2.22.1	17.06.2025	Binary file	See list
v.2.22.0	04.06.2025	Binary file	See list
v.2.21.0	22.05.2025	Binary file	See list
v.2.20.0	05.03.2025	Binary file	See list
v.2.19.0	05.02.2025	Binary file	See list
v.2.18.0	24.12.24	Binary file	See list
v.2.17.0	04.12.24	Binary file	See list
v.2.16.0	26.11.24	Binary file	See list
v.2.15.0	17.10.24	Binary file	
v.2.14.0	03.10.24	Binary file	
v.2.13.0	23.09.24	Binary file	
v.2.12.0	19.09.24	Binary file	
v.2.11.0	15.07.24	Binary file	
v.2.10.0	24.06.24	Binary file	See list

Linux (arm64)

Version	Release date	Download	Changelog
v.2.33.0	23.06.2026	Binary file	See list
v.2.32.0	22.06.2026	Binary file	See list
v.2.31.0	20.04.2026	Binary file	See list
v.2.30.0	07.04.2026	Binary file	See list
v.2.29.0	11.02.2026	Binary file	See list
v.2.28.0	19.12.2025	Binary file	See list
v.2.27.0	30.10.2025	Binary file	See list
v.2.26.0	25.09.2025	Binary file	See list
v.2.25.0	01.09.2025	Binary file	See list
v.2.24.1	28.07.2025	Binary file	See list
v.2.24.0	23.07.2025	Binary file	See list

v.2.23.0	16.07.2025	Binary file	See list
v.2.22.1	17.06.2025	Binary file	See list
v.2.22.0	04.06.2025	Binary file	See list
v.2.21.0	22.05.2025	Binary file	See list
v.2.20.0	05.03.2025	Binary file	See list
v.2.19.0	05.02.2025	Binary file	See list
v.2.18.0	24.12.24	Binary file	See list
v.2.17.0	04.12.24	Binary file	See list
v.2.16.0	26.11.24	Binary file	See list
v.2.15.0	17.10.24	Binary file	
v.2.14.0	03.10.24	Binary file	

macOS (Intel)

Use the amd64 binary file if your Mac is based on an Intel processor.

Version	Release date	Download	Changelog
v.2.33.0	23.06.2026	Binary file	See list
v.2.32.0	22.06.2026	Binary file	See list
v.2.31.0	20.04.2026	Binary file	See list
v.2.30.0	07.04.2026	Binary file	See list
v.2.29.0	11.02.2026	Binary file	See list
v.2.28.0	19.12.2025	Binary file	See list
v.2.27.0	30.10.2025	Binary file	See list
v.2.26.0	25.09.2025	Binary file	See list
v.2.25.0	01.09.2025	Binary file	See list
v.2.24.1	28.07.2025	Binary file	See list
v.2.24.0	23.07.2025	Binary file	See list
v.2.23.0	16.07.2025	Binary file	See list
v.2.22.1	17.06.2025	Binary file	See list
v.2.22.0	04.06.2025	Binary file	See list
v.2.21.0	22.05.2025	Binary file	See list
v.2.20.0	05.03.2025	Binary file	See list
v.2.19.0	05.02.2025	Binary file	See list
v.2.18.0	24.12.24	Binary file	See list
v.2.17.0	04.12.24	Binary file	See list
v.2.16.0	26.11.24	Binary file	See list
v.2.15.0	17.10.24	Binary file	
v.2.14.0	03.10.24	Binary file	
v.2.13.0	23.09.24	Binary file	
v.2.12.0	19.09.24	Binary file	
v.2.11.0	15.07.24	Binary file	
v.2.10.0	24.06.24	Binary file	See list

macOS (Apple Silicon)

Use the arm64 binary file if your Mac is based on an Apple Silicon processor (M1, M2, or later).

Version	Release date	Download	Changelog
v.2.33.0	23.06.2026	Binary file	See list
v.2.32.0	22.06.2026	Binary file	See list
v.2.31.0	20.04.2026	Binary file	See list

v.2.30.0	07.04.2026	Binary file	See list
v.2.29.0	11.02.2026	Binary file	See list
v.2.28.0	19.12.2025	Binary file	See list
v.2.27.0	30.10.2025	Binary file	See list
v.2.26.0	25.09.2025	Binary file	See list
v.2.25.0	01.09.2025	Binary file	See list
v.2.24.1	28.07.2025	Binary file	See list
v.2.24.0	23.07.2025	Binary file	See list
v.2.23.0	16.07.2025	Binary file	See list
v.2.22.1	17.06.2025	Binary file	See list
v.2.22.0	04.06.2025	Binary file	See list
v.2.21.0	22.05.2025	Binary file	See list
v.2.20.0	05.03.2025	Binary file	See list
v.2.19.0	05.02.2025	Binary file	See list
v.2.18.0	24.12.24	Binary file	See list
v.2.17.0	04.12.24	Binary file	See list
v.2.16.0	26.11.24	Binary file	See list
v.2.15.0	17.10.24	Binary file	
v.2.14.0	03.10.24	Binary file	
v.2.13.0	23.09.24	Binary file	
v.2.12.0	19.09.24	Binary file	
v.2.11.0	15.07.24	Binary file	
v.2.10.0	24.06.24	Binary file	See list

Windows

Version	Release date	Download	Changelog
v.2.33.0	23.06.2026	Binary file	See list
v.2.32.0	22.06.2026	Binary file	See list
v.2.31.0	20.04.2026	Binary file	See list
v.2.30.0	07.04.2026	Binary file	See list
v.2.29.0	11.02.2026	Binary file	See list
v.2.28.0	19.12.2025	Binary file	See list
v.2.27.0	30.10.2025	Binary file	See list
v.2.26.0	25.09.2025	Binary file	See list
v.2.25.0	01.09.2025	Binary file	See list
v.2.24.1	28.07.2025	Binary file	See list
v.2.24.0	23.07.2025	Binary file	See list
v.2.23.0	16.07.2025	Binary file	See list
v.2.22.1	17.06.2025	Binary file	See list
v.2.22.0	04.06.2025	Binary file	See list
v.2.21.0	22.05.2025	Binary file	See list
v.2.20.0	05.03.2025	Binary file	See list
v.2.19.0	05.02.2025	Binary file	See list
v.2.18.0	24.12.24	Binary file	See list
v.2.17.0	04.12.24	Binary file	See list
v.2.16.0	26.11.24	Binary file	See list
v.2.15.0	17.10.24	Binary file	
v.2.14.0	03.10.24	Binary file	

v.2.13.0	23.09.24	Binary file	
v.2.12.0	19.09.24	Binary file	
v.2.11.0	15.07.24	Binary file	
v.2.10.0	24.06.24	Binary file	See list

Download YDB DSTool

YDB DSTool (`ydb-dstool`) is a command-line utility for managing the disk subsystem of a YDB cluster.

To use the utility, install ydb-dstool.

Linux (amd64)

Version	Release date	Download	Changelog
v.0.0.14	18.07.2025	Binary file	
v.0.0.13	20.11.2024	Binary file	
v.0.0.12	25.10.2024	Binary file	

Linux (arm64)

Version	Release date	Download	Changelog
v.0.0.14	18.07.2025	Binary file	

macOS (Intel)

Use the amd64 binary file if your Mac is based on an Intel processor.

Version	Release date	Download	Changelog
v.0.0.14	18.07.2025	Binary file	
v.0.0.13	20.11.2024	Binary file	
v.0.0.12	25.10.2024	Binary file	

macOS (Apple Silicon)

Use the arm64 binary file if your Mac is based on an Apple Silicon processor (M1, M2, or later).

Version	Release date	Download	Changelog
v.0.0.14	18.07.2025	Binary file	

Windows

Version	Release date	Download	Changelog
v.0.0.13	20.11.2024	Binary file	
v.0.0.12	25.10.2024	Binary file	

Download YDB Ops

YDB Ops (`ydbops`) is a command-line utility for managing YDB clusters.

Linux

Version	Release date	Download	Changelog
v0.0.25	03/12/2026	Binary file	

macOS (Intel)

Version	Release date	Download	Changelog
v0.0.25	03/12/2026	Binary file	

macOS (Apple Silicon)

Version	Release date	Download	Changelog
v0.0.25	03/12/2026	Binary file	

Download Ansible Playbooks for YDB

A set of automated playbooks for installing and maintaining the server side of open-source YDB or Yandex Enterprise Database using Ansible is available for download via the links below. Their source code is published on GitHub under Apache 2.0 license.

Version	Release date	Download	Changelog
v2.0.0	23.12.2025	ydb-ansible-2.0.0.tar.gz	
v1.3.2	02.12.2025	ydb-ansible-1.3.2.tar.gz	
v1.3.1	01.12.2025	ydb-ansible-1.3.1.tar.gz	
v1.3.0	26.11.2025	ydb-ansible-1.3.0.tar.gz	
v1.2	06.07.2025	ydb-ansible-1.2.tar.gz	
v1.1	21.06.2025	ydb-ansible-1.1.tar.gz	
v1.0	05.03.2025	ydb-ansible-1.0.zip	
v0.15	13.12.2024	ydb-ansible-0.15.zip	
v0.14	30.11.2024	ydb-ansible-0.14.zip	
v0.10	01.08.2024	ydb-ansible-0.10.zip	
v0.9	20.07.2024	ydb-ansible-0.9.zip	
v0.8	10.07.2024	ydb-ansible-0.8.zip	
v0.7	03.05.2024	ydb-ansible-0.7.zip	

YDB Server changelog

Version 26.3

Release candidate 26.3.1.16

Release date: 18.09.26

Functionality

- Backup export and import are available for column-oriented tables, including S3-compatible storage.
- Column-oriented table columns support dictionary encoding. Use `ENCODING(DICT)` for low-cardinality values.
- Local min_max indexes are enabled for column-oriented tables. They skip data fragments outside a query range, reducing the amount of data read.
- Added storage group decommissioning through virtual storage groups. Data moves to virtual groups in the background while applications continue reading and writing data.
- Added authentication through external OpenID Connect identity providers. YDB validates JWT tokens using the provider's JSON Web Key Set (JWKS) and periodically refreshes authentication data.
- Kafka API supports mutual TLS authentication. A client certificate is mapped to a security identifier and SASL authentication is not required.
- Columnar engine optimization: column-oriented tables use an updated compaction strategy that organizes data more efficiently.
- Authentication and authorization subsystem optimization: bulk authorization requests to AccessService are enabled by default, reducing authorization request overhead.
- Streaming YQL queries can access system virtual attributes such as `__ydb_create_time`, `__ydb_write_time`, and others, as well as user attributes `__ydb_user_attributes`. Feature documentation.
- Distributed Storage optimization: full VDisk synchronization is faster because processed SyncLog data is removed locally by default.
- Added transfer metrics and statistics to `DescribeTransfer` for monitoring and diagnostics.
- Added a configurable limit for stored forced-compaction operations. Completed and cancelled operations can be removed automatically when the limit is reached.
- Change Data Capture records can include the OpenTelemetry trace ID of the request that produced the change.
- Topic reads that start from a timestamp filter out messages with earlier write timestamps, including messages stored in the same blob as newer messages.

Disabled functionality

The following functionality is not enabled by default.

- For column-oriented tables, `ALTER TABLE ... COMPACT` can start forced compaction.
- Column-oriented and row-oriented tables now have parity in the set of YQL data types (`Interval`, `Uuid`, and `DyNumber` are supported).
- Added hybrid search, combining full-text relevance and vector similarity into one ranked result.
- Topics can be accessed through the Amazon SQS API, allowing SQS-compatible clients to read and write messages.
- Added JSON indexes for accelerating `JSON_EXISTS` and `JSON_VALUE` queries.
- Full-text indexes support filter columns, allowing search within a logical table partition.
- Full-text indexes can be created for tables with arbitrary primary-key types.

Version 26.2

Version 26.2.1.14

Release date: September 16, 2026.

Functionality

- Full-text indexes are enabled by default.
- Streaming queries can read from local topics, write to local topics, read local tables, and contain multiple `INSERT` statements.
- Streaming queries support watermarks.
- Added Bloom skip indexes: Bloom and Bloom n-gram indexes for column-oriented tables, and prefix Bloom indexes for row-oriented tables.
- Column compression settings for column-oriented tables are available by default.
- The parallelism level can now be configured for index builds.
- For row-oriented tables, `ALTER TABLE` statements `ALTER COLUMN SET DEFAULT` and `ALTER COLUMN DROP DEFAULT` are available by default.
- For row-oriented tables, the YQL statement `TRUNCATE TABLE` is available by default.
- The YQL statement `DISCARD SELECT` is available by default.
- QueryService can return query results in Apache Arrow format; this capability is enabled by default.
- Added forced compaction for row-oriented tables using `ALTER TABLE ... COMPACT`.
- Added automatic storage balancing between groups and background validation of disk placement.
- Table split and merge operations are faster for tables with many partitions: SchemeShard updates only the affected partitions instead of rebuilding the entire partition list.
- Added audit logging for topic operations.
- Added built-in minidump collection based on Google Breakpad for Linux nodes.
- Added the `ydb-dstool pdisk populate` subcommand for reproducing a PDisk workload on another device.

Disabled functionality

This functionality is present in the core to allow rollback from the future 26.3 release, but is not enabled by default. It will be enabled by default in the next major release. It may also be enabled in some managed YDB services.

- Added support for incremental backups, which store only changes relative to the preceding backup in a collection.
- Column-oriented tables can be exported and imported using S3-compatible storage.
- Added export and import of row-oriented tables using a local file system, including file systems mounted over NFS.

- Added snapshot retention for long-running analytical queries over column-oriented tables, preventing snapshot data from being removed before a query completes.
- QueryService can notify SDKs when a node or session is shutting down, allowing clients to stop sending new queries there.
- Added database-level limits on the count and volume of small blobs for column-oriented tables. New writes are rejected when the hard limit is exceeded.
- Watermark expressions can be evaluated outside the context of an individual message.
- Added min-max skip indexes for column-oriented tables.
- Added dictionary encoding for columns in column-oriented tables.
- Added online construction of unique secondary indexes.
- Transactions between topics and tables can use optimized conflict checking.

Bug Fixes

- Fixed incorrect results from some scan queries over column-oriented tables.
- Fixed handling of malformed Kafka requests that could cause excessive memory use or out-of-bounds access.
- Fixed topic reads hanging after a read balancer restart.
- Fixed race conditions in the server-side topic read session and in the Topic SDK.
- Fixed a crash and a hang in streaming query checkpointing.
- Fixed race conditions when cancelling and planning distributed transactions.
- Fixed handling of oversized blocks during encrypted export and a false data corruption error during encrypted restore.
- Fixed a race condition when collecting statistics with `ydb workload topic`.
- Fixed a double free during `DqHashCombine` spilling teardown.
- Fixed lost `ReadSet` acknowledgements that could prevent a transaction involving topics from completing.
- Fixed handling of `NODATA` responses in the KeyValue API: `NOT_FOUND` or `INTERNAL_ERROR` is returned instead of terminating the process.
- Fixed IAM authentication for external data sources in Generic Provider and error handling for provider responses.
- Fixed a memory leak when loading external data source metadata.
- Fixed parsing of tri-state feature flags in YAML configuration that could cause subsequent settings to be lost.
- Fixed copying and exporting tables with secondary indexes after index implementation tables were removed.
- Fixed object filtering and list operations during file-system export.
- Fixed Kafka API Metadata responses that could return an empty broker list or an inconsistent controller ID, causing Kafka AdminClient and Kafka Streams operations to time out.
- Fixed a leak of script execution records created by streaming queries.
- Fixed `local-ydb` overwriting a user-provided `config.yaml` mounted at the default path during the first Docker deployment.
- Fixed a Hive crash on restart when a tablet lock and its persisted leader pointed to different nodes.

Version 26.1

Version 26.1.1.22

Release date: July 27, 2026.

Bug Fixes

- Fixed Kafka API authentication for local users: with enabled `DomainLoginOnly` setting, users could not access tenant databases.
- Fixed a crash (use-after-free) when updating a vector index caused by asynchronous ReadActor destruction.

Version 26.1.1.20

Release date: July 2, 2026.

Functionality

- YQL statements `SHOW CREATE TABLE` and `SHOW CREATE VIEW` return the DDL required to recreate a table or view.
- `ALTER TABLE` supports default values in `ADD COLUMN` (`DEFAULT`).
- Shuffle Elimination is enabled in production: the optimizer can remove unnecessary data shuffles in joins.
- Backup and restore now cover asynchronous replications, external data sources, external tables, and transfers.
- The cluster keeps running when CMS is unavailable.
- Dynamic nodes can be registered using client TLS certificates.
- LDAP service account authentication supports the SASL EXTERNAL mechanism — see `enable_sasl_external_bind`.
- Asynchronous replication mirroring supports auto-partitioned topics; see also topic partitions in CDC.
- TLI (Transaction Lock Invalidation) diagnostics were extended: `tli_config`, logging, and system views.
- Load-based auto-partitioning considers CPU load on the partition leader and all its replicas.
- Streaming queries support writing results to local tables.
- Changefeeds (CDC) can export user security identifiers (`USER_SIDS`) — see `ALTER TABLE` `CHANGEFEED`.
- External data sources support `AUTH_METHOD=IAM`.
- CLI supports token file authentication (`--token-file`).
- Transaction handling between topics and tables has been optimized.

Bug Fixes

- Fixed `RETURNING` in streaming `UPDATE` and interactive queries.
- Fixed `SET DEFAULT` and `DROP DEFAULT` in `ALTER TABLE`.
- Fixed auth token expiry handling in the ticket parser.
- Fixed query execution in Workload Manager after tenant recreation.
- Fixed a use-after-free in the gRPC service layer.
- Fixed incremental restore surviving SchemeShard restarts and shard failures.
- Fixed streaming query metadata missing immediately after creation.
- Fixed async checkpointing stalling when input is full and the checkpoint is empty.

- Fixed cross-database quota interference in the Kesus quoter proxy.
- Fixed hangs in PQ read sessions.
- Fixed scan executor hangs on `SELECT … LIMIT` over empty tables.
- Fixed TLI deferred `LOCKS_BROKEN` flush accounting.
- Fixed streaming query timeout overflow.
- Fixed thread safety and token expiry parsing in the IAM credentials provider.
- Fixed errors in the HTTP gateway.
- Fixed integer overflow when handling null passwords.
- Fixed CDC collection disabled for `IsBuildInProgress` columns.
- Fixed segfault during updates.
- Fixed Shuffle Elimination with the `HashJoinMode` pragma.
- Fixed duplicated rows in scan queries after delivery issues.
- Fixed viewer HTTP endpoints restricted to viewer/admin SIDs; database scope enforced for database-only tokens.
- Fixed OOM when loading trash on blob depot.
- Fixed `TQueryBase` crash after cancelling a streaming query.
- Fixed local CDC reads from YQL.
- Fixed quick remote cancellation in the query service.

Version 25.4

Version 25.4.1.15

Release date: June 5, 2026.

Functionality

- YQL statements `BATCH UPDATE` and `BATCH DELETE FROM` are available for bulk updates and deletes in tables.
- The write execution path has changed substantially: writes now run in streaming mode without fully materializing data on the Query Processor side before sending it to DataShards, which improves performance for large write workloads. The change applies to a subset of scenarios; in some cases (for example, tables with secondary indexes) the previous approach is still used. For an overview of the execution pipeline, see Query execution.
- Lookup Join execution was optimized: it uses a streaming mode without materializing one side of the join, which lowers peak memory use, speeds up queries over large datasets, and removes previous limits on join side sizes. See Index lookup Join and the `JOIN` operator syntax.
- You can now configure access permissions for cluster and database system views.
- Row-oriented tables support configurable cache modes, including a new `in_memory` mode that preloads table data into RAM when sufficient memory is available.
- Topic consumers gained an `availability-period` parameter that extends retention of uncommitted messages beyond `retention-period`.
- Per-partition topic metrics and export to user shard quotas are available for accounting and observability.
- Faster queries with `LIMIT` on column-oriented tables through early result limiting on storage nodes (for queries with no sort or with sort by primary key). See `LIMIT` and `OFFSET` in YQL.
- Column-oriented tables support the `Bool` type in schema and queries — see primitive YQL types.
- A filtered vector index correctly returns rows inserted after the index was created when filter column values are new.
- Stream processing and data delivery are more tightly integrated into the core: topic → table transfer; streaming queries are available to users when `EnableStreamingQueries` is enabled.
- The `overlap_clusters` option substantially improves vector search quality by placing vectors in multiple index clusters (index settings) — see Vector indexes.
- Vector index search is significantly faster across all index types because distances are computed locally on each DataShard before data is sent over the network — see VIEW (vector index) and Vector indexes.
- Full vector search without an ANN index is faster thanks to pushdown (vector search, KNN UDF) — see Vector search and the KNN module.
- Database-stored secrets are fully supported (create, alter, drop, and use) — see Secrets. Note that the legacy syntax is deprecated.
- `UNION ALL` execution was improved with parallel execution, improving performance of analytical queries.

Bug Fixes

- Fixed an LDAP authentication vulnerability: knowing the login and password of any LDAP user (including one who is not a member of a group allowed to access YDB), an attacker could bypass group membership checks and gain access to the cluster (LDAP search filter injection; special characters are now escaped per RFC 2254).

Version 25.3

Version 25.3.1.27

Release date: May 20, 2026.

Functionality

- Added support for two–data center configuration with synchronous data writes (Bridge mode). Available in YDB Enterprise.
- Topic improvements:
  - In Kafka API compacted topics can now be created, and YDB automatically creates and removes the internal service consumer used for topic compaction;
  - Topic APIs were extended with new `DescribeConsumer` and per-partition topic metrics can now be exported into user quotas.
- Implemented backup and restore of topic configuration to and from S3;
- Implemented backup and restore (`VIEW`) to S3 and from S3.

Bug Fixes

- Fixed an LDAP authentication vulnerability: knowing the login and password of any LDAP user (including one who is not a member of a group allowed to access YDB), an attacker could bypass group membership checks and gain access to the cluster

(LDAP search filter injection; special characters are now escaped per RFC 2254).

- Fixed an issue that caused a server-side session leak.
- Fixed an issue where, in rare cases, reads from a table could block its deletion.
- Fixed a race condition when updating the CPU soft limit.
- Fixed behavior, where `ALTER TABLE` could fail for tables with a vector index.
- Fixed nconsistent results in some read-write transactions — conflicting writes no longer overwrite uncommitted changes.
- Fixed serializability violations in read-write transactions after shard restarts.
- Fixed a memory management issue when committing offsets in topics with automatic partitioning enabled.
- Added checks for enabled encryption in zero-copy transfers.
- Fixed an issue that could cause a VDisk to hang in local recovery after a ChunkRead error.
- Eliminated phantom VDisk appearances caused by race conditions between group creation and deletion operations.
- Improved phantom VDisks caused by races between group creation and deletion operations.
- When a session ends via attach stream, a notification is now sent.
- The coordination service now correctly returns `SCHEME_ERROR` for non-existent resources instead of the incorrect `INTERNAL_ERROR` code.
- Fixed memory handling issues and internal data consistency violations in Workload Manager and related scheduler code.
- Fixed an issue where PDisk info requests could time out when the target node was disabled or unavailable.

Version 25.2

Version 25.2.1.26

Release date: May 12, 2026.

Bug Fixes

- Fixed an LDAP authentication vulnerability: knowing the login and password of any LDAP user (including one not in a group with access to YDB), an attacker could bypass group membership checks and gain access to the cluster (injection into the LDAP user search filter; added escaping of special characters per RFC 2254).
- Fixed an issue where tablet deletion might get stuck
- Fixed an issue that caused an error when changing a table's follower
- Fixed a couple of changefeed related issues:
  - Fixed an issue where importing a table with a Utf8 primary key and an enabled changefeed could fail
  - Fixed an issue where importing a table without changefeeds could fail due to incorrect changefeed file lookup.
- Fixed an issue that could cause errors during UPSERT operations in column tables.
- Fixed an error that could cause a crash due to accessing freed memory
- Fixed an issue with duplicates in unique secondary index
- Fixed an issue with checksum mismatch error on restoration compressed backup from s3
- Fixed an issue where some queries from the TPC-H 1000 benchmark could fail
- Fixed a couple of cluster bootstrap related issues:
  - Fixed an issue where cluster bootstrap could hang when mandatory authorization was enabled.
  - Fixed an issue where it was impossible to create new databases for several minutes immediately after cluster deployment
- Fixed an issue where race condition could occur and clients receive `Could not find correct token validator` error when mising newly issued tokens before `LoginProvider` state is updated.
- Fixed an issue where named expression containing another named expression caused incorrect `VIEW` backup

Release candidate 25.2.1.10

Release date: September 21, 2025.

Functionality

- Analytical capabilities are available by default: column-oriented tables can be created without special flags, using LZ4 compression and hash partitioning. Supported operations include a wide range of DML operations (UPDATE, DELETE, UPSERT, INSERT INTO ... SELECT) and CREATE TABLE AS SELECT. Integration with dbt, Apache Airflow, Jupyter, Superset, and federated queries to S3 enables building end-to-end analytical pipelines in YDB.
- Cost-Based Optimizer is enabled by default for queries involving at least one column-oriented table but can also be enabled manually for other queries. The Cost-Based Optimizer improves query performance by determining the optimal join order and join types based on table statistics; supported hints allow fine-tuning execution plans for complex analytical queries.
- Added YDB Transfer – an asynchronous mechanism for transferring data from a topic to a table. You can create a transfer, update or delete it using YQL commands.
- Added spilling, a memory management mechanism, that temporarily offloads intermediate data arising from computations and exceeding available node RAM capacity to external storage. Spilling allows executing user queries that require processing large data volumes exceeding available node memory.
- Increased the maximum amount of time allowed for a single query to execute from 30 minutes to 2 hours.
- Added support for a user-defined Certificate Authority (CA) and Yandex Cloud Identity and Access Management (IAM) authentication in asynchronous replication.
- Enabled by default:
  - vector index for approximate vector similarity search,
  - support for client-side consumer balancing, compacted topics and transactions in YDB Topics Kafka API,
  - support for auto-partitioning topics for row-oriented tables in CDC,
  - support for auto-partitioning topics in asynchronous replication,
  - support for parameterized Decimal type,
  - support for Datetime64 data type,
  - automatic cleanup of temporary tables and directories during export to S3,
  - support for changefeeds in backup and restore operations,
  - the ability to enable followers (read replicas) for covered secondary indexes,
  - system views with history of overloaded partitions.

Bug Fixes

- Fixed CPU resource limiting for column-oriented tables in Workload Manager. Previously CPU consumption could exceed the configured limits.

Version 25.1

Version 25.1.4.18

Release date: May 12, 2026.

Functionality

- Added support for the Kafka frameworks, such as Kafka Connect, Kafka Streams, Confluent Schema Registry, Kafka Streams, Apache Flink, etc. Now YDB Topics Kafka API supports the following features:
  - client-side consumer balancing. To enable it, use the `enable_kafka_native_balancing` flag in the cluster configuration. For for information, see How consumer balancing works in Apache Kafka. When enabled, consumer balancing will work the same way in YDB Topics.
  - compacted topics. To enable topic compaction, use the `enable_topic_compactification_by_key` flag.
  - transactions. To enable transactions, use the `enable_kafka_transactions` flag.
- Added a new protocol to Node Broker that eliminates the long startup of nodes on large clusters (more than 1000 servers).

YDB UI

- Fixed an issue where not all tablets are shown for pers queue group on the tablets tab in diagnostics.
- Fixed an issue where the storage tab on the diagnostics page displayed nodes of other types in addition to storage nodes.
- Fixed a serialization issue that caused an error when opening query execution statistics.
- Changed the logic for nodes transitioning to critical state – the CPU pool, which is 75-99% full, now triggers a warning, not a critical state.

Performance

- Optimized processing of empty inputs when performing JOIN operations.

Bug fixes

- Fixed an LDAP authentication vulnerability: knowing the login and password of any LDAP user (including one not in a group with access to YDB), an attacker could bypass group membership checks and gain access to the cluster (injection into the LDAP user search filter; added escaping of special characters per RFC 2254).
- Added support for a new kind of change record in asynchronous replication — `reset` record (in addition to `update` & `erase` records).
- Fixed an issue where a replication instance with an unspecified `COMMIT_INTERVAL` option caused the process to crash.
- Fixed rare errors when reading from a topic during partition balancing.
- Fixed an issue where dedicated database deletion might leave database system tablets improperly cleaned.
- Fixed an issue that caused tablets to hang when nodes experienced critical memory shortage. Now tablets will automatically start as soon as any of the nodes frees up sufficient resources.
- Fixed an issue where only the first message from a batch was saved when writing Kafka messages, with all other messages in the batch being ignored.

Release candidate 25.1.2.7

Release date: July 14, 2025.

Functionality

- Implemented a vector index for approximate vector similarity search.
- Added support for consistent asynchronous replication.
- Added configuration mechanism V2 that simplifies the deployment of new YDB clusters and further work with them. Comparison of configuration mechanisms V1 and V2.
- Added support for the parameterized Decimal type.
- Added the ability to omit the `DECLARE` operator for query parameter type declarations. Parameter types are now automatically inferred from the provided values.
- Implemented client balancing of partitions when reading using the Kafka protocol (like Kafka itself). Previously, balancing took place on the server. This mode is enabled by setting the `enable_kafka_native_balancing` flag in the cluster configuration.
- Added support for auto-partitioning topics for row-oriented tables in CDC. This mode is enabled by setting the `enable_topic_autopartitioning_for_cdc` flag in the cluster configuration.
- Added the ability to alter the retention period of CDC topics using the `ALTER TOPIC` statement.
- Added support for the DEBEZIUM_JSON format for CDC.
- Added the ability to create changefeed streams to index tables.
- Added the ability to enable followers (read replicas) for covered secondary indexes. This mode is enabled by setting the `enable_access_to_index_impl_tables` flag in the cluster configuration.
- The scope of supported objects in backup and restore operations has been expanded:
  - Support for changefeeds (enabled with the `enable_changefeeds_export` and `enable_changefeeds_import` flags).
  - Support for views (enabled with the `enable_view_export` flag).
- Added automatic cleanup of temporary tables and directories during export to S3. This mode is enabled by setting the `enable_export_auto_dropping` flag in the cluster configuration.
- Added automatic integrity checks of backups during import, which prevent restoration from corrupted backups and protect against data loss.
- Added the ability to create views that refer to UDFs in queries.
- Added system views with information about access right settings, history of overloaded partitions - enabled by setting the `enable_followers_stats` flag in the cluster configuration, history of partitions with broken locks.
- Added new parameters to the CREATE USER and ALTER USER operators:
  - `HASH` — sets a password in encrypted form.
  - `LOGIN` and `NOLOGIN` — unlocks and blocks a user, respectively.
- Enhanced account security:

    - Added user password complexity verification.
    - Implemented automatic user lockout after a specified number of failed attempts to enter the correct password.
    - Added the ability for users to change their own passwords.
- Implemented the ability to toggle functional flags at runtime. Changes to flags that do not specify `(RequireRestart) = true` in the proto file are applied without a cluster restart.
- Changed lock behavior when shard locks exceed the limit. Once the limit is exceeded, the oldest locks (rather than the newest) are converted into full-shard locks.
- Implemented a mechanism to preserve optimistic locks in memory during graceful datashard restarts, reducing `ABORTED` errors caused by lock loss during table balancing.
- Implemented a mechanism to abort volatile transactions with the `ABORTED` status during graceful datashard restarts.
- Added support for removing `NOT NULL` constraints from a table column using the `ALTER TABLE ... ALTER COLUMN ... DROP NOT NULL` statement.
- Added a limit of 100,000 concurrent session-creation requests in the coordination service.
- Increased the maximum number of columns in the primary key from 20 to 30.
- Improved diagnostics and introspection of memory errors (#10419, #11968).
- ***(Experimental)*** Added an experimental mode with strict access control checks. This mode is enabled by setting these flags:
    - `enable_strict_acl_check` – do not allow granting rights to non-existent users and delete users with permissions;
    - `enable_strict_user_management` – enables strict checks for local users (i.e. only the cluster or database administrator can administer local users);
    - `enable_database_admin` – add the role of database administrator;

Backward Incompatible Changes

- If you are using queries that access named expressions as tables using the AS_TABLE function, update temporary over YDB to version v1.23.0-ydb-compat before updating YDB to the current version to avoid errors in query execution.

YDB UI

- Query Editor was redesigned to support partial results load - it starts displaying results when receives a chunk from the server, doesn't have to wait until the query completion. This approach allows application developers to see query results faster.
- Security Improvement: controls that are could not be activated by current user due to lack of permissions are not displayed. Users won't click and experience Access Denied error.
- Added search by tablet id on Tablets tab.
- HotKeys help tab accessible by ⌘+K key is added.
- Operations tab is added to Database page. Operations allow to list operations and cancel them.
- Cluster dashboard redesign and make it collapsable.
- JsonViewer: handle case sensitive search.
- Added code snippets for YDB SDK to connect to selected database. Such snippets must speed up development.
- Rows on Queries tab were sorted by string values after proper backend sort.
- QueryEditor: removed extra confirmation requests on leaving browser page – do not ask confirmation when it's irrelevant.

Performance

- Added support for constant folding in the query optimizer by default. This feature enhances query performance by evaluating constant expressions at compile time, thereby reducing runtime overhead and enabling faster, more efficient execution of complex static expressions.
- Added a granular timecast protocol for distributed transactions, ensuring that slowing one shard does not affect the performance of others.
- Implemented in-memory state migration on a graceful restart, preserving locks and improving transaction success rates. This reduces the execution time of long transactions by decreasing the number of retries.
- Implemented pipeline processing of internal transactions in Node Broker, accelerating the startup of dynamic nodes in the cluster.
- Improved Node Broker resilience under increased cluster load.
- Enabled evictable B-Tree indexes by default instead of non-evictable SST indexes, reducing memory consumption when storing cold data.
- Optimized memory consumption by storage nodes.
- Reduced Hive startup times to 30%.
- Optimized the distributed storage replication process.
- Optimized the header size of large binary objects in VDisk.
- Reduced memory consumption through allocator page cleaning.

Bug Fixes

- Fixed an error in the Interconnect configuration that caused performance degradation.
- Fixed an out-of-memory error that occurred when deleting very large tables by limiting the number of tablets that process this operation concurrently.
- Fixed an issue that caused accidental duplicate entries in the system tablet configuration.
- Fixed an issue where data reads took too long (seconds) during frequent table resharding operations.
- Fixed an error reading from asynchronous replicas that caused failures.
- Fixed an issue that caused rare CDC initial scan freezes.
- Fixed an issue handling incomplete schema transactions in datashards during system restart.
- Fixed an issue causing inconsistent reads from a topic when explicitly confirming a message read within a transaction; users now receive an error when attempting to confirm a message.
- Fixed an issue in which topic auto-partitioning functioned incorrectly within a transaction.
- Fixed an issue in which transactions hang when working with topics during tablet restarts.
- Fixed the "Key is out of range" error when importing data from S3-compatible storage.
- Fixed an issue in which the end of the metadata field in the cluster configuration.
- Improved the secondary index build process: the system now retries on certain errors instead of interrupting the build.
- Fixed an error executing the `RETURNING` expression in `INSERT` and `UPSERT` operations.
- Fixed an issue causing Drop Tablet operations in PQ tablets to hang during Interconnect delays.
- Fixed an error during VDisk compaction.

- Fixed an issue in which long topic-reading sessions ended with "too big inflight" errors.
- Fixed an issue where reading a topic by multiple consumers hangs if at least one partition has no incoming data.
- Fixed a rare issue with PQ tablet restarts.
- Fixed an issue in which, after updating the cluster version, Hive started subscribers in data centers without running database nodes.
- Fixed an issue in which the `Failed to set up listener on port 9092 errno# 98 (Address already in use)` error occurred during version updates.
- Fixed an error that led to a segmentation fault when a healthcheck request and a cluster-node disable request executed simultaneously.
- Fixed an issue that caused partitioning of row-oriented tables to fail when a split key was selected from access samples containing a mix of full-key and key-prefix operations (such as exact and range reads).
- Fixed an issue where the index type defaulted to `GLOBAL SYNC` despite `UNIQUE` being explicitly specified.
- Fixed an issue where topic auto-partitioning did not work when the `max_active_partition` parameter was set via `ALTER TOPIC`.
- Fixed an issue that caused `db scheme describe` to return columns out of their original creation order.

Version 24.4

Version 24.4.4.12

Release date: June 3, 2025.

Performance

- Limited the number of internal inflight configuration updates.
- Optimized memory consumption by PQ tablets.
- Optimized CPU consumption of Scheme shard and reduced query latencies by checking operation count limits before performing tablet split and merge operations.

Bug Fixes

- Fixed a rare issue of client applications hanging during transaction commit where deleting partition had been done before write quota update.
- Fixed an error in copying tables with Decimal type, which caused failures when rolling back to a previous version.
- Fixed an issue where a commit without confirmation of writing to a topic led to the blocking of the current and subsequent transactions with topics.
- Fixed transaction hanging when working with topics during tablet restart or deletion.
- Fixed issues with reading messages larger than 6Mb via Kafka API.
- Fixed memory leak during writing to the topic.
- Fixed errors in processing nullable columns and columns with UUID type in row tables.

Version 24.4.4.2

Release date: April 15, 2025

Functionality

- Enabled by default:
  - support for views
  - auto-partitioning mode for topics
  - transactions involving topics and row-oriented tables simultaneously
  - volatile distributed transactions
- Added the ability to read and write to a topic using the Kafka API without authentication.

Performance

- Enabled by default automatic secondary index selection for queries.

Bug Fixes

- Fixed an error that led to a significant decrease in reading speed from tablet followers.
- Fixed an error that caused volatile distributed transactions to sometimes wait for confirmations until the next reboot.
- Fixed a rare assertion failure (server process crash) when followers attached to leaders with an inconsistent snapshot.
- Fixed a rare datashard crash when a dropped table shard is restarted with uncommitted persistent changes.
- Fixed an error that could disrupt the order of message processing in a topic.
- Fixed a rare error that could stop reading from a topic partition.
- Fixed an issue where a transaction could hang if a user performed a control plane operation on a topic (for example, adding partitions or a consumer) while the PQ tablet is moving to another node.
- Fixed a memory leak issue with the UserInfo counter value. Because of the memory leak, a reading session would eventually return a "too big in flight" error.
- Fixed a proxy crash due to duplicate topics in a request.
- Fixed a rare bug where a user could write to a topic without any account quota being applied or consumed.
- Fixed an issue where topic deletion returned "OK" while the topic tablets persisted in a functional state. To remove such tablets, follow the instructions from the pull request.
- Fixed a rare issue that prevented the restoration of a backup for a large secondary indexed table.
- Fixed an issue that caused errors when inserting data using `UPSERT` into row-oriented tables with default values.
- Resolved a bug that caused failures when executing queries to tables with secondary indexes that returned result lists using the RETURNING * expression.

Version 24.3

Version 24.3.15.5

Release date: February 6, 2025

Functionality

- Added the ability to register a database node using a certificate. In the Node Broker the flag `AuthorizeByCertificate` has been added to enable certificate-based registration.
- Added priorities for authentication ticket through a third-party IAM provider, with the highest priority given to requests from new users. Tickets in the cache update their information with a lower priority.

Performance

- Improved tablet startup time on large clusters: 210 ms → 125 ms (SSD), 260 ms → 165 ms (HDD).

Bug Fixes

- Removed the restriction on writing values greater than 127 to the Uint8 type.
- Fixed an issue where reading small messages from a topic in small chunks significantly increased CPU load, which could lead to delays in reading and writing to the topic.
- Fixed an issue with restoring from a backup stored in S3 with path-style addressing.
- Fixed an issue with restoring from a backup that was created during an automatic table split.
- Fixed an issue with Uuid serialization for CDC.
- Fixed an issue with "frozen" locks, which could be caused by bulk operations (e.g., TTL-based deletions).
- Fixed an issue where reading from a follower of tablets sometimes caused crashes during automatic table splits.
- Fixed an issue where the coordination node successfully registered proxy servers despite a connection loss.
- Fixed an issue that occurred when opening the Embedded UI tab with information about distributed storage groups.
- Fixed an issue where the Health Check did not report time synchronization issues.
- Fixed a rare issue that caused errors during read queries.
- Fixed an uncommitted changes leak and cleaned them up on startup.
- Fixed consistency issues related to caching deleted ranges.

Version 24.3.11.14

Release date: January 9, 2025.

Functionality

- Added support for restart without downtime in a minimal fault-tolerant configuration of a cluster that uses the three-node variant of `mirror-3-dc`.
- Added new UDF Roaring Bitmap functions: AndNotWithBinary, FromUint32List, RunOptimize.

Version 24.3.11.13

Release date: December 24, 2024.

Functionality

- Introduced query tracing, a tool that allows you to view the detailed path of a request through a distributed system.
- Added support for asynchronous replication, that allows synchronizing data between YDB databases in near real time. It can also be used for data migration between databases with minimal downtime for applications interacting with these databases.
- Added support for views, which can be enabled by the cluster administrator using the `enable_views` setting in dynamic configuration.
- Extended federated query capabilities to support new external data sources: MySQL, Microsoft SQL Server, and Greenplum.
- Published documentation on deploying YDB with federated query functionality (manual setup).
- Added a new launch parameter `FQ_CONNECTOR_ENDPOINT` for YDB Docker containers that specifies an external data source connector address. Added support for TLS encryption for connections to the connector and the ability to expose the connector service port locally on the same host as the dynamic YDB node.
- Added an auto-partitioning mode for topics, where partitions can dynamically split based on load while preserving message read-order and exactly-once guarantees. The mode can be enabled by the cluster administrator using the settings `enable_topic_split_merge` and `enable_pqconfig_transactions_at_scheme_shard` in dynamic configuration.
- Added support for transactions involving topics and row-based tables, enabling transactional data transfer between tables and topics, or between topics, ensuring no data loss or duplication. Transactions can be enabled by the cluster administrator using the settings `enable_topic_service_tx` and `enable_pqconfig_transactions_at_scheme_shard` in dynamic configuration.
- Implemented Change Data Capture (CDC) for synchronous secondary indexes.
- Added support for changing record retention periods in CDC topics.
- Added support for auto-increment columns as part of a table's primary key.
- Added audit logging for user login events in YDB, session termination events in the user interface, and backup/restore operations.
- Added a system view with information about sessions installed from the database using a query.
- Added support literal default values for row-oriented tables. When inserting a new row in YDB Query default values will be assigned to the column if specified.
- Added the `version()` built-in function.
- Added support for `RETURNING` clause in queries.
- Added start/end times and authors in the metadata for backup/restore operations from S3-compatible storage.
- Added support for backup/restore of ACL for tables from/to S3-compatible storage.
- Included paths and decompression methods in query plans for reading from S3.
- Added new parsing options for timestamp/datetime fields when reading data from S3.
- Added support for the `Decimal` type in partitioning keys.
- Improved diagnostics for storage issues in HealthCheck.
- ***(Experimental)*** Added a cost-based optimizer for complex queries, involving column-oriented tables. The cost-based optimizer considers a large number of alternative execution plans for each query and selects the best one based on the cost estimate for each option. Currently, this optimizer only works with plans that contain JOIN operations.
- ***(Experimental)*** Initial version of the workload manager was implemented. It allows to create resource pools with CPU, memory and active queries count limits. Resource classifiers were implemented to assign queries to specific resource pool.
- ***(Experimental)*** Implemented automatic index selection for queries, which can be enabled via the `index_auto_choose_mode setting` in `table_service_config` in dynamic configuration.

YDB UI

- Added support for creating and viewing information on asynchronous replication instances.
- Added an indicator for auto-increment columns.
- Added a tab with information about tablets.
- Added a tab with details about distributed storage groups.
- Added a setting to trace all queries and display tracing results.
- Enhanced the PDisk page with attributes, disk space consumption details, and a button to initiate disk decommissioning.
- Added information about currently running queries.
- Added a row limit setting for query editor output and a notification when results exceed the limit.
- Added a tab to display top CPU-consuming queries over the last hour.
- Added a control to search the history and saved queries pages.
- Added the ability to cancel query execution.
- Added a shortcut to save queries in the editor.
- Separated donor disks from other disks in the UI.
- Added support for InterruptInheritance ACL and improved visualization of active ACLs.
- Added a display of the current UI version.
- Added a tab with information about the status of settings for enabling experimental functionality.

Performance

- Accelerated recovery of tables with secondary indexes from backups up to 20% according to our tests.
- Optimized Interconnect throughput.
- Improved the performance of CDC topics with thousands of partitions.
- Enhanced the Hive tablet balancing algorithm.

Bug fixes

- Fixed an issue that caused databases with a large number of tables or partitions to become non-functional during restoration from a backup. Now, if database size limits are exceeded, the restoration operation will fail, but the database will remain operational.
- Implemented a mechanism to forcibly trigger background compaction when discrepancies between the data schema and stored data are detected in DataShard. This resolves a rare issue with delays in schema changes.
- Resolved duplication of authentication tickets, which led to an increased number of requests to authentication providers.
- Fixed an invariant violation issue during the initial scan of CDC, leading to an abnormal termination of the `ydbd` server process.
- Prohibited schema changes for backup tables.
- Fixed an issue with an initial scan freezing during CDC when the table is frequently updated.
- Excluded deleted indexes from the count against the maximum index limit.
- Fixed a bug in the display of the scheduled execution time for a set of transactions (planned step).
- Fixed a problem with interruptions in blue–green deployment in large clusters caused by frequent updates to the node list.
- Resolved a rare issue that caused transaction order violations.
- Fixed an issue in the EvWrite API that resulted in incorrect memory deallocation.
- Resolved a problem with volatile transactions hanging after a restart.
- Fixed a bug in the CDC, which in some cases leads to increased CPU consumption, up to a core per CDC partition.
- Eliminated read delays occurring during and after the splitting of certain partitions.
- Fixed issues when reading data from S3.
- Corrected the calculation of the AWS signature for S3 requests.
- Resolved false positives in the HealthCheck system during database backups involving a large number of shards.

Version 24.2

Release date: August 20, 2024.

Functionality

- Added the ability to set maintenance task priorities in the cluster management system.
- Added a setting to enable stable names for cluster nodes within a tenant.
- Enabled retrieval of nested groups from the LDAP server, improved host parsing in the LDAP-configuration, and added an option to disable built-in authentication via login and password.
- Added support for authenticating dynamic nodes using SSL-certificates.
- Implemented the removal of inactive nodes from Hive without a restart.
- Improved management of inflight pings during Hive restarts in large clusters.
- Changed the order of establishing connections with nodes during Hive restarts.

YDB UI

- Added the option to set a TTL for user sessions in the configuration file.
- Added an option to sort the list of queries by `CPUTime`.
- Fixed precision loss when working with `double`, `float` data types.
- Added support for creating directories in the UI.
- Added an auto-refresh control on all pages.
- Improved ACL display.
- Enabled autocomplete in the queries editor by default.
- Added support for views.

Bug fixes

- Added a check on the size of the local transaction prior to its commit to fix errors in scheme shard operations when exporting/backing up large databases.
- Fixed an issue with duplicate results in SELECT queries when reducing quotas in DataShard.
- Fixed errors occurring during coordinator state changes.

- Fixed issues during the initial CDC scan.
- Resolved race conditions in asynchronous change delivery (asynchronous indexes, CDC).
- Fixed a crash that sometimes occurred during TTL-based deletions.
- Fixed an issue with PDisk status display in the CMS.
- Fixed an issue that might cause soft tablet transfers (drain) from a node to hang.
- Resolved an issue with the interconnect proxy stopping on a node that is running without restarts. The issue occurred when adding another node to the cluster.
- Corrected string escaping in error messages.
- Fixed an issue with managing free memory in the interconnect.
- Corrected UnreplicatedPhantoms and UnreplicatedNonPhantoms counters in VDisk.
- Fixed an issue with handling empty garbage collection requests on VDisk.
- Resolved issues with managing TVDiskControls settings through CMS.
- Fixed an issue with failing to load the data created by newer versions of VDisk.
- Fixed an issue with executing the `REPLACE INTO` queries with default values.
- Fixed errors in queries with multiple LEFT JOINs to a single string table.
- Fixed precision loss for `float`, `double` types when using CDC.

Version 24.1

Release date: July 31, 2024.

Functionality

- The Knn UDF function for precise nearest vector search has been implemented.
- The gRPC Query service has been developed, enabling the execution of all types of queries (DML, DDL) and retrieval of unlimited amounts of data.
- Integration with the LDAP protocol has been implemented, allowing the retrieval of a list of groups from external LDAP directories.

Embedded UI

- The database information tab now includes a resource consumption diagnostic dashboard, which allows users to assess the current consumption of key resources: processor cores, RAM, and distributed storage space.
- Charts for monitoring the key performance indicators of the YDB cluster have been added.

Performance

- Session timeouts for the coordination service between server and client have been optimized. Previously, the timeout was 5 seconds, which could result in a 10-second delay in identifying an unresponsive client and releasing its resources. In the new version, the check interval depends on the session's wait time, allowing for faster responses during leader changes or when acquiring distributed locks.
- CPU consumption by SchemeShard replicas has been optimized, particularly when handling rapid updates for tables with a large number of partitions.

Bug fixes

- A possible queue overflow error has been fixed. Change Data Capture now reserves the change queue capacity during the initial scan.
- A potential deadlock between receiving and sending CDC records has been fixed.
- An issue causing the loss of the mediator task queue during mediator reconnection has been fixed. This fix allows processing of the mediator task queue during resynchronization.
- A rarely occurring error has been fixed, where with volatile transactions enabled, a successful transaction confirmation result could be returned before the transaction was fully committed. Volatile transactions remain disabled by default and are still under development.
- A rare error that led to the loss of established locks and the successful confirmation of transactions that should have failed with a "Transaction Locks Invalidated" error has been fixed.
- A rare error that could result in a violation of data integrity guarantees during concurrent read and write operations on a specific key has been fixed.
- An issue causing read replicas to stop processing requests has been fixed.
- A rare error that could cause abnormal termination of database processes if there were uncommitted transactions on a table during its renaming has been fixed.
- An error in determining the status of a static group, where it was not marked as non-working when it should have been, has been fixed.
- An error involving partial commits of a distributed transaction with uncommitted changes, caused by certain race conditions with restarts, has been fixed.
- Anomalies related to reading outdated data, detected using Jepsen, have been fixed.

Version 23.4

Release date: May 14, 2024.

Performance

- Fixed an issue of increased CPU consumption by a topic actor `PERSQUEUE_PARTITION_ACTOR`.
- Optimized resource usage by SchemeBoard replicas. The greatest effect is noticeable when modifying the metadata of tables with a large number of partitions.

Bug fixes

- Fixed a bug of possible partial commit of accumulated changes when using persistent distributed transactions. This error occurs in an extremely rare combination of events, including restarting tablets that service the table partitions involved in the transaction.
- Fixed a bug involving a race condition between the table merge and garbage collection processes, which could result in garbage collection ending with an invariant violation error, leading to an abnormal termination of the `ydbd` server process.
- Fixed a bug in Blob Storage, where information about changes to the composition of a storage group might not be received in a timely manner by individual cluster nodes. As a result, reads and writes of data stored in the affected group could become blocked in rare cases, requiring manual intervention.

- Fixed a bug in Blob Storage, where data storage nodes might not start despite the correct configuration. The error occurred on systems with the experimental "blob depot" feature explicitly enabled (this feature is disabled by default).
- Fixed a bug that sometimes occurred when writing to a topic with an empty `producer_id` with turned off deduplication. It could lead to abnormal termination of the `ydbd` server process.
- Fixed a bug that caused the `ydbd` process to crash due to an incorrect session state when writing to a topic.
- Fixed a bug in displaying the metric of number of partitions in a topic, where it previously displayed an incorrect value.
- Fixed a bug causing memory leaks that appeared when copying topic data between clusters. These could cause `ydbd` server processes to terminate due to out-of-memory issues.

Version 23.3

Release date: October 12, 2023.

Functionality

- Implemented visibility of own changes. With this feature enabled you can read changed values from the current transaction, which has not been committed yet. This functionality also allows multiple modifying operations in one transaction on a table with secondary indexes.
- Added support for column tables. It is now possible to create analytical reports based on stored data in YDB with performance comparable to specialized analytical DBMS.
- Added support for Kafka API for topics. YDB topics can now be accessed via a Kafka-compatible API designed for migrating existing applications. Support for Kafka protocol version 3.4.0 is provided.
- Added the ability to write to a topic without deduplication. This is important in cases where message processing order is not critical.
- YQL has added the capabilities to create, modify, and delete topics.
- Added support of assigning and revoking access rights using the YQL `GRANT` and `REVOKE` commands.
- Added support of DML-operations logging in the audit log.
- ***(Experimental)*** When writing messages to a topic, it is now possible to pass metadata. To enable this functionality, add `enable_topic_message_meta: true` to the configuration file.
- ***(Experimental)*** Added support for reading from topics in a transaction. It is now possible to read from topics and write to tables within a transaction, simplifying the data transfer scenario from a topic to a table. To enable this functionality, add `enable_topic_service_tx: true` to the configuration file.
- ***(Experimental)*** Added support for PostgreSQL compatibility. This involves executing SQL queries in PostgreSQL dialect on the YDB infrastructure using the PostgreSQL network protocol. With this capability, familiar PostgreSQL tools such as psql and drivers (e.g., pq for Golang and psycopg2 for Python) can be used. Queries can be developed using the familiar PostgreSQL syntax and take advantage of YDB's benefits such as horizontal scalability and fault tolerance.
- ***(Experimental)*** Added support for federated queries. This enables retrieving information from various data sources without the need to move the data into YDB. Federated queries support interaction with ClickHouse and PostgreSQL databases, as well as S3 class data stores (Object Storage). YQL queries can be used to access these databases without duplicating data between systems.

Embedded UI

- A new option `PostgreSQL` has been added to the query type selector settings, which is available when the `Enable additional query modes` parameter is enabled. Also, the query history now takes into account the syntax used when executing the query.
- The YQL query template for creating a table has been updated. Added a description of the available parameters.
- Now sorting and filtering for Storage and Nodes tables takes place on the server. To use this functionality, you need to enable the parameter `Offload tables filters and sorting to backend` in the experiments section.
- Buttons for creating, changing and deleting topics have been added to the context menu.
- Added sorting by criticality for all issues in the tree in `Healthcheck`.

Performance

- Implemented read iterators. This feature allows to separate reads and computations. Read iterators allow datashards to increase read queries throughput.
- The performance of writing to YDB topics has been optimized.
- Improved tablet balancing during node overload.

Bug fixes

- Fixed an error regarding potential blocking of reading iterators of snapshots, of which the coordinators were unaware.
- Memory leak when closing the connection in Kafka proxy has been fixed.
- Fixed an issue where snapshots taken through reading iterators may fail to recover on restarts.
- Fixed an issue with an incorrect residual predicate for the `IS NULL` condition on a column.
- Fixed an occurring verification error: `VERIFY failed: SendResult(): requirement ChunksLimiter.Take(sendBytes) failed`.
- Fixed `ALTER TABLE` for TTL on column-based tables.
- Implemented a `FeatureFlag` that allows enabling/disabling work with `CS` and `DS`.
- Fixed a 50ms time difference between coordinator time in 23-2 and 23-3.
- Fixed an error where the storage endpoint was returning extra groups when the `viewer backend` had the `node_id` parameter in the request.
- Added a usage filter to the `/storage` endpoint in the `viewer backend`.
- Fixed an issue in Storage v2 where an incorrect number was returned in the `Degraded field`.
- Fixed an issue with cancelling subscriptions from sessions during tablet restarts.
- Fixed an error where `healthcheck alerts` for storage were flickering during rolling restarts when going through a load balancer.
- Updated `CPU usage metrics` in YDB.
- Fixed an issue where `NULL` was being ignored when specifying `NOT NULL` in the table schema.
- Implemented logging of `DDL` operations in the common log.
- Implemented restriction for the YDB table attribute `add/drop` command to only work with tables and not with any other objects.
- Disabled `CloseOnIdle` for interconnect.

- Fixed the doubling of read speed in the UI.
- Fixed an issue where data could be lost on block-4-2.
- Added a check for topic name validity.
- Fixed a possible deadlock in the actor system.
- Fixed the `KqpScanArrowInChanels::AllTypesColumns` test.
- Fixed the `KqpScan::SqlInParameter` test.
- Fixed parallelism issues for OLAP queries.
- Fixed the insertion of `ClickBench` parquet files.
- Added a missing call to `CheckChangesQueueOverflow` in the general `CheckDataTxReject`.
- Fixed an error that returned an empty status in `ReadRows` API calls.
- Fixed incorrect retry behavior in the final stage of export.
- Fixed an issue with infinite quota for the number of records in a `CDC topic`.
- Fixed the import error of `string` and `parquet` columns into an `OLAP string column`.
- Fixed a crash in `KqpOlapTypes.Timestamp` under `tsan`.
- Fixed a `viewer backend` crash when attempting to execute a query against the database due to version incompatibility.
- Fixed an error where the viewer did not return a response from the `healthcheck` due to a timeout.
- Fixed an error where incorrect `ExpectedSerial` values could be saved in `Pdisks`.
- Fixed an error where database nodes were crashing due to segfault in the S3 actor.
- Fixed a race condition in `ThreadSanitizer: data race KqpService::ToDictCache-UseCache`.
- Fixed a race condition in `GetNextReadId`.
- Fixed an issue with an inflated result in `SELECT COUNT(*)` immediately after import.
- Fixed an error where `TEvScan` could return an empty dataset in the case of shard splitting.
- Added a separate `issue/error` code in case of available space exhaustion.
- Fixed a `GRPC_LIBRARY Assertion` failed error.
- Fixed an error where scanning queries on secondary indexes returned an empty result.
- Fixed validation of `CommitOffset` in `TopicAPI`.
- Reduced shared cache consumption when approaching OOM.
- Merged scheduler logic from data executer and scan executer into one class.
- Added discovery and `proxy` handlers to the query execution process in the `viewer backend`.
- Fixed an error where the `/cluster` endpoint returned the root domain name, such as `/ru`, in the `viewer backend`.
- Implemented a seamless table update scheme for `QueryService`.
- Fixed an issue where `DELETE` returned data and did not delete it.
- Fixed an error in `DELETE ON` operation in query service.
- Fixed an unexpected batching disablement in `default` schema settings.
- Fixed a triggering check `VERIFY failed: MoveUserTable(): requirement move.ReMapIndexesSize() == newTableInfo->Indexes.size()`.
- Increased the `default` timeout for `grpc-streaming`.
- Excluded unused messages and methods from `QueryService`.
- Added sorting by `Rack` in /nodes in the `viewer backend`.
- Fixed an error where sorting queries returned an error in descending order.
- Improved interaction between `QP` and `NodeWhiteboard`.
- Removed support for old parameter formats.
- Fixed an error where `DefineBox` was not being applied to disks with a static group.
- Fixed a `SIGSEGV` error in the dinnode during `CSV` import via `YDB CLI`.
- Fixed an error that caused a crash when processing `NGRpcService::TRefreshTokenImpl`.
- Implemented a `gossip protocol` for exchanging cluster resource information.
- Fixed an error in `DeserializeValuePickleV1(): requirement data.GetTransportVersion() == (ui32) NDqProto::DATA_TRANSPORT_UV_PICKLE_1_0 failed`.
- Implemented `auto-increment` columns.
- Use `UNAVAILABLE` status instead of `GENERIC_ERROR` when shard identification fails.
- Added support for rope payload in `TEvVGet`.
- Added ignoring of deprecated events.
- Fixed a crash of write sessions on an invalid topic name.
- Fixed an error in `CheckExpected(): requirement newConstr failed, message: Rewrite error, missing Distinct((id)) constraint in node FlatMap`.
- Enabled `self-heal` by default.

Version 23.2

Release date: August 14, 2023.

Functionality

- ***(Experimental)*** Implemented visibility of own changes. With this feature enabled you can read changed values from the current transaction, which has not been committed yet. This functionality also allows multiple modifying operations in one transaction on a table with secondary indexes. To enable this feature add `enable_kqp_immediate_effects: true` under `table_service_config` section into configuration file.
- ***(Experimental)*** Implemented read iterators. This feature allows to separate reads and computations. Read iterators allow datashards to increase read queries throughput. To enable this feature add `enable_kqp_data_query_source_read: true` under `table_service_config` section into configuration file.

Embedded UI

- Navigation improvements:

    - Diagnostics and Development mode switches are moved to the left panel.
    - Every page has breadcrumbs.
    - Storage groups and nodes info are moved from left buttons to tabs on the database page.
- Query history and saved queries are moved to tabs over the text editor area in query editor.
- Info tab for scheme objects displays parameters using terms from `CREATE` or `ALTER` statements.
- Added column tables support.

Performance

- For scan queries, you can now effectively search for individual rows using a primary key or secondary indexes. This can bring you a substantial performance gain in many cases. Similarly to regular queries, you need to explicitly specify its name in the query text using the `VIEW` keyword to use a secondary index.
- ***(Experimental)*** Added an option to give control of the system tablets of the database (SchemeShard, Coordinators, Mediators, SysViewProcessor) to its own Hive instead of the root Hive, and do so immediately upon creating a new database. Without this flag, the system tablets of the new database are created in the root Hive, which can negatively impact its load. Enabling this flag makes databases completely isolated in terms of load, that may be particularly relevant for installations, consisting from a roughly hundred or more databases. To enable this feature add `alter_database_create_hive_first: true` under `feature_flags` section into configuration file.

Bug fixes

- Fixed a bug in the autoconfiguration of the actor system, resulting in all the load being placed on the system pool.
- Fixed a bug that caused full scanning when searching by prefix of the primary key using `LIKE`.
- Fixed bugs when interacting with datashard followers.
- Fixed bugs when working with memory in column tables.
- Fixed a bug in processing conditions for immediate transactions.
- Fixed a bug in the operation of iterator-based reads on datasharrd followers.
- Fixed a bug that caused cascading reinstallation of data delivery sessions to asynchronous indexes.
- Fixed bugs in the optimizer for scanning queries.
- Fixed a bug in the incorrect calculation of storage consumption by Hive after expanding the database.
- Fixed a bug that caused operations to hang on non-existent iterators.
- Fixed bugs when reading a range on a `NOT NULL` column.
- Fixed a bug in the replication of VDisks.
- Fixed a bug in the work of the `run_interval` option in TTL.

Version 23.1

Release date: May 5, 2023. To update to version 23.1, select the Downloads section.

Functionality

- Added initial table scan when creating a CDC changefeed. Now, you can export all the data existing at the time of changefeed creation.
- Added atomic index replacement. Now, you can atomically replace one pre-defined index with another. This operation is absolutely transparent for your application. Indexes are replaced seamlessly, with no downtime.
- Added the audit log: Event stream including data about all the operations on YDB objects.

Performance

- Improved formats of data exchanged between query stages. As a result, we accelerated SELECTs by 10% on parameterized queries and by up to 30% on write operations.
- Added autoconfiguring for the actor system pools based on the workload against them. This improves performance through more effective CPU sharing.
- Optimized the predicate logic: Processing of parameterized OR or IN constraints is automatically delegated to DataShard.
- (Experimental) For scan queries, you can now effectively search for individual rows using a primary key or secondary indexes. This can bring you a substantial gain in performance in many cases. Similarly to regular queries, to use a secondary index, you need to explicitly specify its name in the query text using the `VIEW` keyword.
- The query's computational graph is now cached at query runtime, reducing the CPU resources needed to build the graph.

Bug fixes

- Fixed bugs in the distributed data warehouse implementation. We strongly recommend all our users to upgrade to the latest version.
- Fixed the error that occurred on building an index on NOT NULL columns.
- Fixed statistics calculation with MVCC enabled.
- Fixed errors with backups.
- Fixed the race condition that occurred at splitting and deleting a table with SDC.

Version 22.5

Release date: March 7, 2023. To update to version **22.5**, select the Downloads section.

What's new

- Added changefeed configuration parameters to transfer additional information about changes to a topic.
- You can now rename tables that have TTL enabled.
- You can now manage the record retention period.

Bug fixes and improvements

- Fixed an error inserting 0 rows with a BulkUpsert.
- Fixed an error importing Date/DateTime columns from CSV.
- Fixed an error importing CSV data with line breaks.
- Fixed an error importing CSV data with NULL values.
- Improved Query Processing performance (by replacing WorkerActor with SessionActor).

- DataShard compaction now starts immediately after a split or merge.

Version 22.4

Release date: October 12, 2022. To update to version **22.4**, select the Downloads section.

What's new

- YDB Topics and Change Data Capture (CDC):
  - Introduced the new Topic API. YDB Topic is an entity for storing unstructured messages and delivering them to various subscribers.
  - Added support for the Topic API to the YDB CLI and SDK. The Topic API provides methods for message streaming writes and reads as well as topic management.
  - Added the ability to capture table updates and send change messages to a topic.
- SDK:
  - Added the ability to handle topics in the YDB SDK.
  - Added official support for the database/sql driver for working with YDB in Golang.
- Embedded UI:
  - The CDC changefeed and the secondary indexes are now displayed in the database schema hierarchy as separate objects.
  - Improved the visualization of query explain plan graphics.
  - Problem storage groups have more visibility now.
  - Various improvements based on UX research.
- Query Processing:
  - Added Query Processor 2.0, a new subsystem to execute OLTP queries with significant improvements compared to the previous version.
  - Improved write performance by up to 60%, and by up to 10% for reads.
  - Added the ability to include a NOT NULL restriction for YDB primary keys when creating tables.
  - Added support for renaming a secondary index online without shutting the service down.
  - Improved the query explain view that now also includes fields for the physical operators.
- Core:
  - For read only transactions, added consistent snapshot support that does not conflict with write transactions.
  - Added BulkUpsert support for tables with asynchronous secondary indexes.
  - Added TTL support for tables with asynchronous secondary indexes.
  - Added compression support for data export to S3.
  - Added an audit log for DDL statements.
  - Added support for authentication with static credentials.
  - Added system tables for query performance troubleshooting.

Yandex Enterprise Database changelog

Version 26.1

Version 26.1.1.ent.3

Release date: August 3, 2026.

This version includes all improvements from YDB 26.1.1.22; see the changelog. It also includes the enterprise-specific improvements listed below.

Enterprise-specific Improvements

The following changes are available in Yandex Enterprise Database in addition to the corresponding YDB build:

- Added an optimization that allows filtering rows by index columns before querying the main table, reducing the number of accesses to the main table when executing certain types of queries.
- Implemented a set of fixes in index access (StreamIndexLookup) that eliminates the possibility of rare situations where executed queries could hang, and reduces RAM consumption during query execution.
- Invalid views can now be restored from a backup. This allows restoring backups created from databases containing such views without additional actions from the administrator.
- Added support for mutual certificate-based authentication (mTLS) in the Kafka API.
- Added the `TraceId` column with the query trace identifier to the `.sys/top_queries_*` and `.sys/query_sessions` system views.

Version 25.4

Version 25.4.1.ent.2

Release date: June 17, 2026.

This version includes all improvements from YDB 25.4.1.15; see the changelog. It also includes all additional fixes listed below for version 25.2.1.ent.13.

Version 25.3

Version 25.3.1.ent.3

Release date: June 11, 2026.

This version includes all improvements from YDB 25.3.1.27; see the changelog. It also includes all additional fixes listed below for version 25.2.1.ent.13.

Version 25.2

Version 25.2.1.ent.13

Release date: June 11, 2026.

This version includes all improvements from YDB 25.2.1.26; see the changelog. It also includes a number of additional improvements ported from the current 26.1 version.

Additional Fixes

The following changes were ported from version 26.1 into supported stable versions of Yandex Enterprise Database:

- Fixed a bug that violated the sort order specified in the query when accessing system tables.
- Fixed a bug in internal state integrity check logic that in rare cases could cause a single (not mass) restart of storage nodes.
- Added an optimization that allows filtering rows by index columns before querying the main table, reducing the number of accesses to the main table when executing certain types of queries.
- Implemented a set of fixes in index access (StreamIndexLookup) that eliminates the possibility of rare situations where executed queries could hang, and reduces RAM consumption during query execution.
- Added an optimization that reduces memory consumption when processing queries with the TopSort operation (`SELECT ... ORDER BY x LIMIT n`).
- Added support for index materialization during backup and restore.
- TLI (Transaction Locks Invalidated) error messages now always include either an identifier or the path of the affected table.
- Lock metrics have been added to query statistics provided through the `.sys/query_metrics_*` system tables.
- Invalid views can now be restored from a backup. This allows restoring backups created from databases containing such views without additional actions from the administrator.

Version 25.2.1.ent.4

Release date: February 12, 2026.

New Features

- Analytical capabilities are available by default: column-oriented tables can be created without special flags, using LZ4 compression and hash partitioning. Supported operations include a wide range of DML operations (UPDATE, DELETE, UPSERT, INSERT INTO ... SELECT) and CREATE TABLE AS SELECT. Integration with dbt, Apache Airflow, Jupyter, Superset, and federated queries to S3 enables building end-to-end analytical pipelines in YDB.
- Cost-Based Optimizer is enabled by default for queries involving at least one column-oriented table but can also be enabled manually for other queries. The Cost-Based Optimizer improves query performance by determining the optimal join order and join types based on table statistics; supported hints allow fine-tuning execution plans for complex analytical queries.
- Added YDB Transfer – an asynchronous mechanism for transferring data from a topic to a table. You can create a transfer, update or delete it using YQL commands.
- Added spilling, a memory management mechanism, that temporarily offloads intermediate data arising from computations and exceeding available node RAM capacity to external storage. Spilling allows executing user queries that require processing large data volumes exceeding available node memory.
- Increased the maximum amount of time allowed for a single query to execute from 30 minutes to 2 hours.
- Added support for a user-defined Certificate Authority (CA) and Yandex Cloud Identity and Access Management (IAM) authentication in asynchronous replication.

- Enabled by default:
  - vector index for approximate vector similarity search,
  - support for client-side consumer balancing, compacted topics and transactions in YDB Topics Kafka API,
  - support for auto-partitioning topics for row-oriented tables in CDC,
  - support for auto-partitioning topics in asynchronous replication,
  - support for parameterized Decimal type,
  - support for Datetime64 data type,
  - automatic cleanup of temporary tables and directories during export to S3,
  - support for changefeeds in backup and restore operations,
  - the ability to enable followers (read replicas) for covered secondary indexes,
  - system views with history of overloaded partitions.

Bug Fixes

- Fixed CPU resource limiting for column-oriented tables in Workload Manager. Previously CPU consumption could exceed the configured limits.
- Fixed an issue where tablet deletion might get stuck
- Fixed an issue that caused an error when changing a table's follower
- Fixed a couple of changefeed related issues:
  - Fixed an issue where importing a table with a Utf8 primary key and an enabled changefeed could fail
  - Fixed an issue where importing a table without changefeeds could fail due to incorrect changefeed file lookup.
- Fixed an issue that could cause errors during UPSERT operations in column tables.
- Fixed an error that could cause a crash due to accessing freed memory
- Fixed an issue with duplicates in unique secondary index
- Fixed an issue with checksum mismatch error on restoration compressed backup from s3
- Fixed an issue where some queries from the TPC-H 1000 benchmark could fail
- Fixed a couple of cluster bootstrap related issues:
  - Fixed an issue where cluster bootstrap could hang when mandatory authorization was enabled.
  - Fixed an issue where it was impossible to create new databases for several minutes immediately after cluster deployment
- Fixed an issue where race condition could occur and clients receive `Could not find correct token validator` error when mising newly issued tokens before `LoginProvider` state is updated.

Version 25.1

Version 25.1.4.ent.8

Release date: February 12, 2026.

Bug Fixes

- Fixed an issue where named expression containing another named expression caused incorrect `VIEW` backup
- Fixed descending sorting not working in queries to system views.

Version 25.1.4.ent.3

Release date: November 25, 2025.

New Features

- Implemented a vector index for approximate vector similarity search.
- Added support for consistent asynchronous replication.
- Added configuration mechanism V2 that simplifies the deployment of new YDB clusters and further work with them. Comparison of configuration mechanisms V1 and V2.
- Added support for the parameterized Decimal type.
- Implemented client balancing of partitions when reading using the Kafka protocol (like Kafka itself). Previously, balancing took place on the server. This mode is enabled by setting the `enable_kafka_native_balancing` flag in the cluster configuration.
- Added support for auto-partitioning topics for row-oriented tables in CDC. This mode is enabled by setting the `enable_topic_autopartitioning_for_cdc` flag in the cluster configuration.
- Added the ability to alter the retention period of CDC topics using the `ALTER TOPIC` statement.
- Added support for the DEBEZIUM_JSON format for CDC.
- Added the ability to create changefeed streams to index tables.
- Added the ability to enable followers (read replicas) for covered secondary indexes. This mode is enabled by setting the `enable_access_to_index_impl_tables` flag in the cluster configuration.
- Changefeeds are now supported in backup and restore operations. To use this feature, set the `enable_changefeeds_export` and `enable_changefeeds_import` flags in the `feature_flags` section of the database or cluster configuration.
- Added automatic cleanup of temporary tables and directories during export to S3. This mode is enabled by setting the `enable_export_auto_dropping` flag in the cluster configuration.
- Added automatic integrity checks of backups during import, which prevent restoration from corrupted backups and protect against data loss.
- Added the ability to create views that refer to UDFs in queries.
- Added system views with information about access right settings, history of overloaded partitions - enabled by setting the `enable_followers_stats` flag in the cluster configuration, history of partitions with broken locks.
- Added new parameters to the CREATE USER and ALTER USER operators:
  - `HASH` — sets a password in encrypted form.
  - `LOGIN` and `NOLOGIN` — unlocks and blocks a user, respectively.
- Enhanced account security:
  - Added user password complexity verification.
  - Implemented automatic user lockout after a specified number of failed attempts to enter the correct password.
  - Added the ability for users to change their own passwords.
- Implemented the ability to toggle functional flags at runtime. Changes to flags that do not specify `(RequireRestart) = true` in the proto file are applied without a cluster restart.

- Changed lock behavior when shard locks exceed the limit. Once the limit is exceeded, the oldest locks (rather than the newest) are converted into full-shard locks.
- Implemented a mechanism to preserve optimistic locks in memory during graceful datashard restarts, reducing `ABORTED` errors caused by lock loss during table balancing.
- Implemented a mechanism to abort volatile transactions with the `ABORTED` status during graceful datashard restarts.
- Added support for removing `NOT NULL` constraints from a table column using the `ALTER TABLE ... ALTER COLUMN ... DROP NOT NULL` statement.
- Added a limit of 100,000 concurrent session-creation requests in the coordination service.
- Increased the maximum number of columns in the primary key from 20 to 30.
- Improved diagnostics and introspection of memory errors (#10419, #11968).
- ***(Experimental)*** Added an experimental mode with strict access control checks. This mode is enabled by setting these flags:
  - `enable_strict_acl_check` – do not allow granting rights to non-existent users and delete users with permissions;
  - `enable_strict_user_management` – enables strict checks for local users (i.e. only the cluster or database administrator can administer local users);
  - `enable_database_admin` – add the role of database administrator;
- Added support for the Kafka frameworks, such as Kafka Connect, Kafka Streams, Confluent Schema Registry, Kafka Streams, Apache Flink, etc. Now YDB Topics Kafka API supports the following features:
  - client-side consumer balancing. To enable it, use the `enable_kafka_native_balancing` flag in the cluster configuration. For for information, see How consumer balancing works in Apache Kafka. When enabled, consumer balancing will work the same way in YDB Topics.
  - compacted topics. To enable topic compaction, use the `enable_topic_compactification_by_key` flag.
  - transactions. To enable transactions, use the `enable_kafka_transactions` flag.
- Added a new protocol to Node Broker that eliminates the long startup of nodes on large clusters (more than 1000 servers).

Backward Incompatible Changes

- If you are using queries that access named expressions as tables using the AS_TABLE function, update temporary over YDB to version v1.23.0-ydb-compat before updating YDB to the current version to avoid errors in query execution.

YDB UI

- Query Editor was redesigned to support partial results load - it starts displaying results when receives a chunk from the server, doesn't have to wait until the query completion. This approach allows application developers to see query results faster.
- Security Improvement: controls that are could not be activated by current user due to lack of permissions are not displayed. Users won't click and experience Access Denied error.
- Added search by tablet id on Tablets tab.
- HotKeys help tab accessible by ⌘+K key is added.
- Operations tab is added to Database page. Operations allow to list operations and cancel them.
- Cluster dashboard redesign and make it collapsable.
- JsonViewer: handle case sensitive search.
- Added code snippets for YDB SDK to connect to selected database. Such snippets must speed up development.
- Rows on Queries tab were sorted by string values after proper backend sort.
- QueryEditor: removed extra confirmation requests on leaving browser page – do not ask confirmation when it's irrelevant.
- Fixed an issue where not all tablets are shown for pers queue group on the tablets tab in diagnostics.
- Fixed an issue where the storage tab on the diagnostics page displayed nodes of other types in addition to storage nodes.
- Fixed a serialization issue that caused an error when opening query execution statistics.
- Changed the logic for nodes transitioning to critical state – the CPU pool, which is 75-99% full, now triggers a warning, not a critical state.

Performance

- Added support for constant folding in the query optimizer by default. This feature enhances query performance by evaluating constant expressions at compile time, thereby reducing runtime overhead and enabling faster, more efficient execution of complex static expressions.
- Added a granular timecast protocol for distributed transactions, ensuring that slowing one shard does not affect the performance of others.
- Implemented in-memory state migration on a graceful restart, preserving locks and improving transaction success rates. This reduces the execution time of long transactions by decreasing the number of retries.
- Implemented pipeline processing of internal transactions in Node Broker, accelerating the startup of dynamic nodes in the cluster.
- Improved Node Broker resilience under increased cluster load.
- Enabled evictable B-Tree indexes by default instead of non-evictable SST indexes, reducing memory consumption when storing cold data.
- Optimized memory consumption by storage nodes.
- Reduced Hive startup times to 30%.
- Optimized the distributed storage replication process.
- Optimized the header size of large binary objects in VDisk.
- Reduced memory consumption through allocator page cleaning.
- Optimized processing of empty inputs when performing JOIN operations.

Bug Fixes

- Fixed an error in the Interconnect configuration that caused performance degradation.
- Fixed an out-of-memory error that occurred when deleting very large tables by limiting the number of tablets that process this operation concurrently.
- Fixed an issue that caused accidental duplicate entries in the system tablet configuration.
- Fixed an issue where data reads took too long (seconds) during frequent table resharding operations.
- Fixed an error reading from asynchronous replicas that caused failures.
- Fixed an issue that caused rare CDC initial scan freezes.
- Fixed an issue handling incomplete schema transactions in datashards during system restart.
- Fixed an issue causing inconsistent reads from a topic when explicitly confirming a message read within a transaction; users now receive an error when attempting to confirm a message.

- Fixed an issue in which topic auto-partitioning functioned incorrectly within a transaction.
- Fixed an issue in which transactions hang when working with topics during tablet restarts.
- Fixed the "Key is out of range" error when importing data from S3-compatible storage.
- Fixed an issue in which the end of the metadata field in the cluster configuration.
- Improved the secondary index build process: the system now retries on certain errors instead of interrupting the build.
- Fixed an error executing the `RETURNING` expression in `INSERT` and `UPSERT` operations.
- Fixed an issue causing Drop Tablet operations in PQ tablets to hang during Interconnect delays.
- Fixed an error during VDisk compaction.
- Fixed an issue in which long topic-reading sessions ended with "too big inflight" errors.
- Fixed an issue where reading a topic by multiple consumers hangs if at least one partition has no incoming data.
- Fixed a rare issue with PQ tablet restarts.
- Fixed an issue in which, after updating the cluster version, Hive started subscribers in data centers without running database nodes.
- Fixed an issue in which the `Failed to set up listener on port 9092 errno# 98 (Address already in use)` error occurred during version updates.
- Fixed an error that led to a segmentation fault when a healthcheck request and a cluster-node disable request executed simultaneously.
- Fixed an issue that caused partitioning of row-oriented tables to fail when a split key was selected from access samples containing a mix of full-key and key-prefix operations (such as exact and range reads).
- Fixed an issue where the index type defaulted to `GLOBAL SYNC` despite `UNIQUE` being explicitly specified.
- Fixed an issue where topic auto-partitioning did not work when the `max_active_partition` parameter was set via `ALTER TOPIC`.
- Fixed an issue that caused `db scheme describe` to return columns out of their original creation order.
- Added support for a new kind of change record in asynchronous replication — `reset` record (in addition to `update` & `erase` records).
- Fixed an issue where a replication instance with an unspecified `COMMIT_INTERVAL` option caused the process to crash.
- Fixed rare errors when reading from a topic during partition balancing.
- Fixed an issue where dedicated database deletion might leave database system tablets improperly cleaned.
- Fixed an issue that caused tablets to hang when nodes experienced critical memory shortage. Now tablets will automatically start as soon as any of the nodes frees up sufficient resources.
- Fixed an issue where only the first message from a batch was saved when writing Kafka messages, with all other messages in the batch being ignored.

Version 24.4

Version 24.4.4.20

Release date: November 1, 2025.

New Features

- Views are now supported in backup and restore operations. To use this feature, set the `enable_view_export` flag in the `feature_flags` section of the database or cluster configuration.
- Additional identifiers — the object path ID (`PathId`) and tablet ID (`TabletId`) — are now included in Transaction locks invalidated error messages when the table cannot be identified (Unknown table).

Version 24.4.4.15

Release date: September 19, 2025.

Performance

- Columns in `ORDER BY` statement are now considered by the optimizer when automatically selecting a secondary index. This optimization is limited to queries that reference only one table and do not include any `JOIN` operations with other tables.

Bug Fixes

- When receiving an `OperationAborted` error from S3, the export operation does not terminate with an error, but retries writing to S3.

Version 24.4.4.13

Release date: July 29, 2025.

New Features

- Added support for restart without downtime in a minimal fault-tolerant configuration of a cluster that uses the three-node variant of `mirror-3-dc`.
- Added new UDF Roaring Bitmap functions: AndNotWithBinary, FromUint32List, RunOptimize.
- Added the ability to register a database node using a certificate. In the Node Broker the flag `AuthorizeByCertificate` has been added to enable certificate-based registration.
- Added priorities for authentication ticket through a third-party IAM provider, with the highest priority given to requests from new users. Tickets in the cache update their information with a lower priority.
- Added the ability to read and write to a topic using the Kafka API without authentication.
- Enabled by default:
  - support for views
  - auto-partitioning mode for topics
  - transactions involving topics and row-oriented tables simultaneously
  - volatile distributed transactions

Performance

- Improved tablet startup time on large clusters: 210 ms → 125 ms (SSD), 260 ms → 165 ms (HDD).
- Limited the number of internal inflight configuration updates.
- Optimized memory consumption by PQ tablets.

- Optimized CPU consumption of Scheme shard and reduced query latencies by checking operation count limits before performing tablet split and merge operations.
- Automated secondary index selection is now enabled by default.

Bug Fixes

- Fixed an issue where reading small messages from a topic in small chunks significantly increased CPU load, which could lead to delays in reading and writing to the topic.
- Fixed an issue with restoring from a backup that was created during an automatic table split.
- Fixed an issue with Uuid serialization for CDC.
- Fixed an issue where reading from a follower of tablets sometimes caused crashes during automatic table splits.
- Fixed an issue where the coordination node successfully registered proxy servers despite a connection loss.
- Fixed an issue that occurred when opening the Embedded UI tab with information about distributed storage groups.
- Fixed an issue where the Health Check did not report time synchronization issues.
- Fixed a rare issue of client applications hanging during transaction commit where deleting partition had been done before write quota update.
- Fixed an error in copying tables with Decimal type, which caused failures when rolling back to a previous version.
- Fixed an issue where a commit without confirmation of writing to a topic led to the blocking of the current and subsequent transactions with topics.
- Fixed transaction hanging when working with topics during tablet restart or deletion.
- Fixed issues with reading messages larger than 6Mb via Kafka API.
- Fixed memory leak during writing to the topic.
- Fixed errors in processing nullable columns and columns with UUID type in row tables.
- Fixed an error that led to a significant decrease in reading speed from tablet followers.
- Fixed an error that caused volatile distributed transactions to sometimes wait for confirmations until the next reboot.
- Fixed a rare assertion failure (server process crash) when followers attached to leaders with an inconsistent snapshot.
- Fixed a rare datashard crash when a dropped table shard is restarted with uncommitted persistent changes.
- Fixed an error that could disrupt the order of message processing in a topic.
- Fixed a rare error that could stop reading from a topic partition.
- Fixed an issue where a transaction could hang if a user performed a control plane operation on a topic (for example, adding partitions or a consumer) while the PQ tablet is moving to another node.
- Fixed a memory leak issue with the UserInfo counter value. Because of the memory leak, a reading session would eventually return a "too big in flight" error.
- Fixed a proxy crash due to duplicate topics in a request.
- Fixed a rare bug where a user could write to a topic without any account quota being applied or consumed.
- Fixed an issue where topic deletion returned "OK" while the topic tablets persisted in a functional state. To remove such tablets, follow the instructions from the pull request.
- Fixed a rare issue that prevented the restoration of a backup for a large secondary indexed table.
- Fixed an issue that caused errors when inserting data using `UPSERT` into row-oriented tables with default values.
- Resolved a bug that caused failures when executing queries to tables with secondary indexes that returned result lists using the RETURNING * expression.

Version 24.3

Version 24.3.13.11

Release date: March 6, 2025.

Bug Fixes

- Fixed an uncommitted changes leak and cleaned them up on startup.
- Fixed consistency issues related to caching deleted ranges.
- Fixed the issue of caching negative responses to authentication requests from an LDAP-compatible user and group directory for too long.

Version 24.3.13.10

Release date: December 24, 2024.

New Features

- Introduced query tracing, a tool that allows you to view the detailed path of a request through a distributed system.
- Added support for asynchronous replication, that allows synchronizing data between YDB databases in near real time. It can also be used for data migration between databases with minimal downtime for applications interacting with these databases.
- Added support for views, which can be enabled by the cluster administrator using the `enable_views` setting in dynamic configuration.
- Extended federated query capabilities to support new external data sources: MySQL, Microsoft SQL Server, and Greenplum.
- Published documentation on deploying YDB with federated query New Features (manual setup).
- Added a new launch parameter `FQ_CONNECTOR_ENDPOINT` for YDB Docker containers that specifies an external data source connector address. Added support for TLS encryption for connections to the connector and the ability to expose the connector service port locally on the same host as the dynamic YDB node.
- Added an auto-partitioning mode for topics, where partitions can dynamically split based on load while preserving message read-order and exactly-once guarantees. The mode can be enabled by the cluster administrator using the settings `enable_topic_split_merge` and `enable_pqconfig_transactions_at_scheme_shard` in dynamic configuration.
- Added support for transactions involving topics and row-based tables, enabling transactional data transfer between tables and topics, or between topics, ensuring no data loss or duplication. Transactions can be enabled by the cluster administrator using the settings `enable_topic_service_tx` and `enable_pqconfig_transactions_at_scheme_shard` in dynamic configuration.
- Implemented Change Data Capture (CDC) for synchronous secondary indexes.
- Added support for changing record retention periods in CDC topics.
- Added support for auto-increment columns as part of a table's primary key.
- Added audit logging for user login events in YDB, session termination events in the user interface, and backup/restore operations.
- Added a system view with information about sessions installed from the database using a query.

- Added support literal default values for row-oriented tables. When inserting a new row in YDB Query default values will be assigned to the column if specified.
- Added the `version()` built-in function.
- Added support for `RETURNING` clause in queries.
- Added start/end times and authors in the metadata for backup/restore operations from S3-compatible storage.
- Added support for backup/restore of ACL for tables from/to S3-compatible storage.
- Included paths and decompression methods in query plans for reading from S3.
- Added new parsing options for timestamp/datetime fields when reading data from S3.
- Added support for the `Decimal` type in partitioning keys.
- Improved diagnostics for storage issues in HealthCheck.
- ***(Experimental)*** Added a cost-based optimizer for complex queries, involving column-oriented tables. The cost-based optimizer considers a large number of alternative execution plans for each query and selects the best one based on the cost estimate for each option. Currently, this optimizer only works with plans that contain JOIN operations.
- ***(Experimental)*** Initial version of the workload manager was implemented. It allows to create resource pools with CPU, memory and active queries count limits. Resource classifiers were implemented to assign queries to specific resource pool.
- ***(Experimental)*** Implemented automatic index selection for queries, which can be enabled via the `index_auto_choose_mode setting` in `table_service_config` in dynamic configuration.

YDB UI

- Added support for creating and viewing information on asynchronous replication instances.
- Added an indicator for auto-increment columns.
- Added a tab with information about tablets.
- Added a tab with details about distributed storage groups.
- Added a setting to trace all queries and display tracing results.
- Enhanced the PDisk page with attributes, disk space consumption details, and a button to initiate disk decommissioning.
- Added information about currently running queries.
- Added a row limit setting for query editor output and a notification when results exceed the limit.
- Added a tab to display top CPU-consuming queries over the last hour.
- Added a control to search the history and saved queries pages.
- Added the ability to cancel query execution.
- Added a shortcut to save queries in the editor.
- Separated donor disks from other disks in the UI.
- Added support for InterruptInheritance ACL and improved visualization of active ACLs.
- Added a display of the current UI version.
- Added a tab with information about the status of settings for enabling experimental New Features.

Performance

- Accelerated recovery of tables with secondary indexes from backups up to 20% according to our tests.
- Optimized Interconnect throughput.
- Improved the performance of CDC topics with thousands of partitions.
- Enhanced the Hive tablet balancing algorithm.

Bug fixes

- Fixed an issue that caused databases with a large number of tables or partitions to become non-functional during restoration from a backup. Now, if database size limits are exceeded, the restoration operation will fail, but the database will remain operational.
- Implemented a mechanism to forcibly trigger background compaction when discrepancies between the data schema and stored data are detected in DataShard. This resolves a rare issue with delays in schema changes.
- Resolved duplication of authentication tickets, which led to an increased number of requests to authentication providers.
- Fixed an invariant violation issue during the initial scan of CDC, leading to an abnormal termination of the `ydbd` server process.
- Prohibited schema changes for backup tables.
- Fixed an issue with an initial scan freezing during CDC when the table is frequently updated.
- Excluded deleted indexes from the count against the maximum index limit.
- Fixed a bug in the display of the scheduled execution time for a set of transactions (planned step).
- Fixed a problem with interruptions in blue–green deployment in large clusters caused by frequent updates to the node list.
- Resolved a rare issue that caused transaction order violations.
- Fixed an issue in the EvWrite API that resulted in incorrect memory deallocation.
- Resolved a problem with volatile transactions hanging after a restart.
- Fixed a bug in the CDC, which in some cases leads to increased CPU consumption, up to a core per CDC partition.
- Eliminated read delays occurring during and after the splitting of certain partitions.
- Fixed issues when reading data from S3.
- Corrected the calculation of the AWS signature for S3 requests.
- Resolved false positives in the HealthCheck system during database backups involving a large number of shards.
- Removed the restriction on writing values greater than 127 to the Uint8 type.
- Fixed an issue with restoring from a backup stored in S3 with path-style addressing.
- Fixed an issue with "frozen" locks, which could be caused by bulk operations (e.g., TTL-based deletions).
- Fixed a rare issue that caused errors during read queries.

Version 24.2

Version 24.2.7.1

Release date: August 20, 2024.

New Features

- Added the ability to set maintenance task priorities in the cluster management system.
- Added a setting to enable stable names for cluster nodes within a tenant.

- Enabled retrieval of nested groups from the LDAP server, improved host parsing in the LDAP-configuration, and added an option to disable built-in authentication via login and password.
- Added support for authenticating dynamic nodes using SSL-certificates.
- Implemented the removal of inactive nodes from Hive without a restart.
- Improved management of inflight pings during Hive restarts in large clusters.
- Changed the order of establishing connections with nodes during Hive restarts.

YDB UI

- Added the option to set a TTL for user sessions in the configuration file.
- Added an option to sort the list of queries by `CPUTime`.
- Fixed precision loss when working with `double`, `float` data types.
- Added support for creating directories in the UI.
- Added an auto-refresh control on all pages.
- Improved ACL display.
- Enabled autocomplete in the queries editor by default.
- Added support for views.

Bug fixes

- Added a check on the size of the local transaction prior to its commit to fix errors in scheme shard operations when exporting/backing up large databases.
- Fixed an issue with duplicate results in SELECT queries when reducing quotas in DataShard.
- Fixed errors occurring during coordinator state changes.
- Fixed issues during the initial CDC scan.
- Resolved race conditions in asynchronous change delivery (asynchronous indexes, CDC).
- Fixed a crash that sometimes occurred during TTL-based deletions.
- Fixed an issue with PDisk status display in the CMS.
- Fixed an issue that might cause soft tablet transfers (drain) from a node to hang.
- Resolved an issue with the interconnect proxy stopping on a node that is running without restarts. The issue occurred when adding another node to the cluster.
- Corrected string escaping in error messages.
- Fixed an issue with managing free memory in the interconnect.
- Corrected UnreplicatedPhantoms and UnreplicatedNonPhantoms counters in VDisk.
- Fixed an issue with handling empty garbage collection requests on VDisk.
- Resolved issues with managing TVDiskControls settings through CMS.
- Fixed an issue with failing to load the data created by newer versions of VDisk.
- Fixed an issue with executing the `REPLACE INTO` queries with default values.
- Fixed errors in queries with multiple LEFT JOINs to a single string table.
- Fixed precision loss for `float`, `double` types when using CDC.

Version 24.1

Version 24.1.18.1

Release date: July 31, 2024.

New Features

- The Knn UDF function for precise nearest vector search has been implemented.
- The gRPC Query service has been developed, enabling the execution of all types of queries (DML, DDL) and retrieval of unlimited amounts of data.
- Integration with the LDAP protocol has been implemented, allowing the retrieval of a list of groups from external LDAP directories.

Embedded UI

- The database information tab now includes a resource consumption diagnostic dashboard, which allows users to assess the current consumption of key resources: processor cores, RAM, and distributed storage space.
- Charts for monitoring the key performance indicators of the YDB cluster have been added.

Performance

- Session timeouts for the coordination service between server and client have been optimized. Previously, the timeout was 5 seconds, which could result in a 10-second delay in identifying an unresponsive client and releasing its resources. In the new version, the check interval depends on the session's wait time, allowing for faster responses during leader changes or when acquiring distributed locks.
- CPU consumption by SchemeShard replicas has been optimized, particularly when handling rapid updates for tables with a large number of partitions.

Bug fixes

- A possible queue overflow error has been fixed. Change Data Capture now reserves the change queue capacity during the initial scan.
- A potential deadlock between receiving and sending CDC records has been fixed.
- An issue causing the loss of the mediator task queue during mediator reconnection has been fixed. This fix allows processing of the mediator task queue during resynchronization.
- A rarely occurring error has been fixed, where with volatile transactions enabled, a successful transaction confirmation result could be returned before the transaction was fully committed. Volatile transactions remain disabled by default and are still under development.
- A rare error that led to the loss of established locks and the successful confirmation of transactions that should have failed with a "Transaction Locks Invalidated" error has been fixed.
- A rare error that could result in a violation of data integrity guarantees during concurrent read and write operations on a specific key has been fixed.

- An issue causing read replicas to stop processing requests has been fixed.
- A rare error that could cause abnormal termination of database processes if there were uncommitted transactions on a table during its renaming has been fixed.
- An error in determining the status of a static group, where it was not marked as non-working when it should have been, has been fixed.
- An error involving partial commits of a distributed transaction with uncommitted changes, caused by certain race conditions with restarts, has been fixed.
- Anomalies related to reading outdated data, detected using Jepsen, have been fixed.

Version 23.4

Version 23.4.11.1

Release date: May 14, 2024.

Performance

- Fixed an issue of increased CPU consumption by a topic actor `PERSQUEUE_PARTITION_ACTOR`.
- Optimized resource usage by SchemeBoard replicas. The greatest effect is noticeable when modifying the metadata of tables with a large number of partitions.

Bug fixes

- Fixed a bug of possible partial commit of accumulated changes when using persistent distributed transactions. This error occurs in an extremely rare combination of events, including restarting tablets that service the table partitions involved in the transaction.
- Fixed a bug involving a race condition between the table merge and garbage collection processes, which could result in garbage collection ending with an invariant violation error, leading to an abnormal termination of the `ydbd` server process.
- Fixed a bug in Blob Storage, where information about changes to the composition of a storage group might not be received in a timely manner by individual cluster nodes. As a result, reads and writes of data stored in the affected group could become blocked in rare cases, requiring manual intervention.
- Fixed a bug in Blob Storage, where data storage nodes might not start despite the correct configuration. The error occurred on systems with the experimental "blob depot" feature explicitly enabled (this feature is disabled by default).
- Fixed a bug that sometimes occurred when writing to a topic with an empty `producer_id` with turned off deduplication. It could lead to abnormal termination of the `ydbd` server process.
- Fixed a bug that caused the `ydbd` process to crash due to an incorrect session state when writing to a topic.
- Fixed a bug in displaying the metric of number of partitions in a topic, where it previously displayed an incorrect value.
- Fixed a bug causing memory leaks that appeared when copying topic data between clusters. These could cause `ydbd` server processes to terminate due to out-of-memory issues.

Version 23.3

Version 23.3.25.2

Release date: October 12, 2023.

New Features

- Implemented visibility of own changes. With this feature enabled you can read changed values from the current transaction, which has not been committed yet. This New Features also allows multiple modifying operations in one transaction on a table with secondary indexes.
- Added support for column tables. It is now possible to create analytical reports based on stored data in YDB with performance comparable to specialized analytical DBMS.
- Added support for Kafka API for topics. YDB topics can now be accessed via a Kafka-compatible API designed for migrating existing applications. Support for Kafka protocol version 3.4.0 is provided.
- Added the ability to write to a topic without deduplication. This is important in cases where message processing order is not critical.
- YQL has added the capabilities to create, modify, and delete topics.
- Added support of assigning and revoking access rights using the YQL `GRANT` and `REVOKE` commands.
- Added support of DML-operations logging in the audit log.
- ***(Experimental)*** When writing messages to a topic, it is now possible to pass metadata. To enable this New Features, add `enable_topic_message_meta: true` to the configuration file.
- ***(Experimental)*** Added support for reading from topics in a transaction. It is now possible to read from topics and write to tables within a transaction, simplifying the data transfer scenario from a topic to a table. To enable this New Features, add `enable_topic_service_tx: true` to the configuration file.
- ***(Experimental)*** Added support for PostgreSQL compatibility. This involves executing SQL queries in PostgreSQL dialect on the YDB infrastructure using the PostgreSQL network protocol. With this capability, familiar PostgreSQL tools such as psql and drivers (e.g., pq for Golang and psycopg2 for Python) can be used. Queries can be developed using the familiar PostgreSQL syntax and take advantage of YDB's benefits such as horizontal scalability and fault tolerance.
- ***(Experimental)*** Added support for federated queries. This enables retrieving information from various data sources without the need to move the data into YDB. Federated queries support interaction with ClickHouse and PostgreSQL databases, as well as S3 class data stores (Object Storage). YQL queries can be used to access these databases without duplicating data between systems.

Embedded UI

- A new option `PostgreSQL` has been added to the query type selector settings, which is available when the `Enable additional query modes` parameter is enabled. Also, the query history now takes into account the syntax used when executing the query.
- The YQL query template for creating a table has been updated. Added a description of the available parameters.
- Now sorting and filtering for Storage and Nodes tables takes place on the server. To use this New Features, you need to enable the parameter `Offload tables filters and sorting to backend` in the experiments section.
- Buttons for creating, changing and deleting topics have been added to the context menu.
- Added sorting by criticality for all issues in the tree in `Healthcheck`.

Performance

- Implemented read iterators. This feature allows to separate reads and computations. Read iterators allow datashards to increase read queries throughput.
- The performance of writing to YDB topics has been optimized.
- Improved tablet balancing during node overload.

Bug fixes

- Fixed an error regarding potential blocking of reading iterators of snapshots, of which the coordinators were unaware.
- Memory leak when closing the connection in Kafka proxy has been fixed.
- Fixed an issue where snapshots taken through reading iterators may fail to recover on restarts.
- Fixed an issue with an incorrect residual predicate for the `IS NULL` condition on a column.
- Fixed an occurring verification error: `VERIFY failed: SendResult(): requirement ChunksLimiter.Take(sendBytes) failed`.
- Fixed `ALTER TABLE` for TTL on column-based tables.
- Implemented a `FeatureFlag` that allows enabling/disabling work with `CS` and `DS`.
- Fixed a 50ms time difference between coordinator time in 23-2 and 23-3.
- Fixed an error where the storage endpoint was returning extra groups when the `viewer backend` had the `node_id` parameter in the request.
- Added a usage filter to the `/storage` endpoint in the `viewer backend`.
- Fixed an issue in Storage v2 where an incorrect number was returned in the `Degraded field`.
- Fixed an issue with cancelling subscriptions from sessions during tablet restarts.
- Fixed an error where `healthcheck alerts` for storage were flickering during rolling restarts when going through a load balancer.
- Updated `CPU usage metrics` in YDB.
- Fixed an issue where `NULL` was being ignored when specifying `NOT NULL` in the table schema.
- Implemented logging of `DDL` operations in the common log.
- Implemented restriction for the YDB table attribute `add/drop` command to only work with tables and not with any other objects.
- Disabled `CloseOnIdle` for interconnect.
- Fixed the doubling of read speed in the UI.
- Fixed an issue where data could be lost on block-4-2.
- Added a check for topic name validity.
- Fixed a possible deadlock in the actor system.
- Fixed the `KqpScanArrowInChanels::AllTypesColumns` test.
- Fixed the `KqpScan::SqlInParameter` test.
- Fixed parallelism issues for OLAP queries.
- Fixed the insertion of `ClickBench` parquet files.
- Added a missing call to `CheckChangesQueueOverflow` in the general `CheckDataTxReject`.
- Fixed an error that returned an empty status in `ReadRows` API calls.
- Fixed incorrect retry behavior in the final stage of export.
- Fixed an issue with infinite quota for the number of records in a `CDC topic`.
- Fixed the import error of `string` and `parquet` columns into an `OLAP string column`.
- Fixed a crash in `KqpOlapTypes.Timestamp` under `tsan`.
- Fixed a `viewer backend` crash when attempting to execute a query against the database due to version incompatibility.
- Fixed an error where the viewer did not return a response from the `healthcheck` due to a timeout.
- Fixed an error where incorrect `ExpectedSerial` values could be saved in `Pdisks`.
- Fixed an error where database nodes were crashing due to segfault in the S3 actor.
- Fixed a race condition in `ThreadSanitizer: data race KqpService::ToDictCache-UseCache`.
- Fixed a race condition in `GetNextReadId`.
- Fixed an issue with an inflated result in `SELECT COUNT(*)` immediately after import.
- Fixed an error where `TEvScan` could return an empty dataset in the case of shard splitting.
- Added a separate `issue/error` code in case of available space exhaustion.
- Fixed a `GRPC_LIBRARY Assertion` failed error.
- Fixed an error where scanning queries on secondary indexes returned an empty result.
- Fixed validation of `CommitOffset` in `TopicAPI`.
- Reduced shared cache consumption when approaching OOM.
- Merged scheduler logic from data executer and scan executer into one class.
- Added discovery and `proxy` handlers to the query execution process in the `viewer backend`.
- Fixed an error where the `/cluster` endpoint returned the root domain name, such as `/ru`, in the `viewer backend`.
- Implemented a seamless table update scheme for `QueryService`.
- Fixed an issue where `DELETE` returned data and did not delete it.
- Fixed an error in `DELETE ON` operation in query service.
- Fixed an unexpected batching disablement in `default` schema settings.
- Fixed a triggering check `VERIFY failed: MoveUserTable(): requirement move.ReMapIndexesSize() == newTableInfo->Indexes.size()`.
- Increased the `default` timeout for `grpc-streaming`.
- Excluded unused messages and methods from `QueryService`.
- Added sorting by `Rack` in /nodes in the `viewer backend`.
- Fixed an error where sorting queries returned an error in descending order.
- Improved interaction between `QP` and `NodeWhiteboard`.
- Removed support for old parameter formats.

- Fixed an error where `DefineBox` was not being applied to disks with a static group.
- Fixed a `SIGSEGV` error in the dinnode during `CSV` import via `YDB CLI`.
- Fixed an error that caused a crash when processing `NGRpcService::TRefreshTokenImpl`.
- Implemented a `gossip protocol` for exchanging cluster resource information.
- Fixed an error:

```
DeserializeValuePickleV1(): requirement data.GetTransportVersion() ==
(ui32) NDqProto::DATA_TRANSPORT_UV_PICKLE_1_0 failed
```

- Implemented `auto-increment` columns.
- Use `UNAVAILABLE` status instead of `GENERIC_ERROR` when shard identification fails.
- Added support for rope payload in `TEvVGet`.
- Added ignoring of deprecated events.
- Fixed a crash of write sessions on an invalid topic name.
- Fixed an error:

```
CheckExpected(): requirement newConstr failed, message: Rewrite error,
missing Distinct((id)) constraint in node FlatMap
```

- Enabled `self-heal` by default.

YDB CLI changelog

Version 2.33.0

Released on June 23, 2026. To update to version **2.33.0**, select the Downloads section.

Features

- The AI assistant in the `ydb` interactive mode can now search the YDB documentation, providing more accurate and context-aware answers.

Version 2.32.0

Released on June 22, 2026. To update to version **2.32.0**, select the Downloads section.

Features

- Added an AI mode to the `ydb` interactive mode. Press `Ctrl+T` to switch to it.
- Added the `ydb workload fulltext` command for load testing and measuring the quality of fulltext indexes.
- Added the `SET resource_pool` command to the `ydb` interactive mode.
- Added the `--resource-pool` option to the `ydb sql` command.
- Added the `--no-consumer` option to the `ydb topic read` command for reading from a topic without a consumer.
- Added the `--partition-write-speed-mps` and `--partition-write-burst-messages` options to the `ydb topic create` command and `ydb topic alter` command.
- Added the `--partition-max-inflight-bytes` option to the `ydb workload topic` command.
- Added the `--codec kafka-batch`, `--batch-inner-codec`, `--batch-flush-interval`, `--batch-flush-size`, and `--batch-flush-message-count` options to the `ydb workload topic run write|full` commands.
- Added CPU time statistics to the `ydb workload * run` commands.

Improvements

- The `ydb scheme describe` command now prints a human-readable description for external data sources (source type, location, authentication method, database, properties, and creation time) instead of empty output.

Version 2.31.0

Released on April 20, 2026. To update to version **2.31.0**, select the Downloads section.

Features

- Added the `--stats` option to the `ydb workload * run` commands, enabling extended execution statistics collection (e.g., `--stats profile`).

Version 2.30.0

Released on April 7, 2026. To update to version **2.30.0**, select the Downloads section.

Features

- Added the `ydb config completion` command to generate shell completion scripts for bash and zsh.
- Added the `ydb export nfs` and `ydb import nfs` commands, allowing users to create and restore backups directly to/from a shared NFS directory mounted on every host in the cluster.
- Added the `--compact` option to the `ydb workload tpcc import` command.
- Added the `--tx-mode` option to the `ydb workload * run` commands, which allows setting the transaction mode (e.g., `no-tx`, `serializable-rw`, `snapshot-rw`).
- Added support for the new compaction operation in the `ydb operation` subcommands.

Improvements

- When a profile is explicitly specified with the `-p` / `--profile` option, the active profile is no longer used: all options are taken only from the specified profile, environment variables, and command line. This avoids confusion when the chosen profile was unexpectedly supplemented with settings from the active profile.

Version 2.29.0

Released on February 11, 2026. To update to version **2.29.0**, select the Downloads section.

Features

- Enhancements to the `ydb` interactive mode:
  - Introduced the `/help` command for interactive command guidance.
  - Introduced the `/config` command, providing an interactive dialog to view and customize YDB CLI settings, including:
    - Enabling or disabling autocompletion hints.
    - Enabling or disabling color output.
    - Interactively selecting a color theme from a set of predefined options, with support for cloning and customizing your own theme.
- Added a download progress bar to the `ydb update` command.
- Added the `--include-index-data` option to the `ydb export s3` command, enabling index data export.
- Added the `--index-population-mode` option to the `ydb import s3` command, allowing selection of the index population mode (e.g., `build` or `import`).
- Added the `Created by`, `Create time`, and `End time` fields to the "build index" and "execute script" operations in the `ydb operation` subcommands.
- Added unified time interval format support across YDB CLI commands. Options accepting time durations now support explicit time units (e.g., `5s`, `2m`, `1h`) while maintaining backward compatibility with plain numbers interpreted using their original default units.

- Replaced the deprecated "Keep in memory" field with the "Cache mode" field in the column families description of the `ydb scheme describe` command.

Improvements

- Improved the `ydb init` and `ydb config profile` commands with interactive menus.
- Improved progress bars: consistent MiB/GiB units, stable speed display, and a dual progress bar for the `ydb import file` command showing both in-progress and confirmed bytes.

Bug fixes

- Fixed an out-of-memory issue in the `ydb workload query run` command for queries with large result sets.
- Fixed static credentials parsing to avoid using a profile password when the username comes from another source.

Version 2.28.0

Released on December 19, 2025. To update to version **2.28.0**, select the Downloads section.

Features

- Added `snapshot-ro` and `snapshot-rw` transaction modes to the `--tx-mode` option of the `ydb table query execute` command.
- Added `NO_COLOR` environment variable support to disable ANSI colors in YDB CLI (see no-color.org).
- Added a simple progress bar for non-interactive stderr.
- Added the `omit-indexes` property to the `--item` option of the `ydb tools copy` command, allowing tables to be copied without their indexes.
- Added the `import files` subcommand to the `ydb workload vector` command to populate the table from CSV or Parquet files.
- Added the `import generate` subcommand to the `ydb workload vector` command to populate the table with random data.
- ***(Requires server v26.1+)*** Changes to previously added `ydb admin cluster state fetch` command:
  - Renamed to `ydb admin cluster diagnostics collect`.
  - Added the `--no-sanitize` option, which disables sanitization and preserves sensitive data in the output.
  - Added the `--output` option to specify the path to the output `.tar.bz2` file.

Bug fixes

- Fixed a bug where the `ydb tools restore` command could crash with a `mutex lock failure (Invalid argument)` error due to an internal race condition.
- Fixed restoration of views containing named expressions and views that access secondary indexes in the `ydb tools restore` command.

Version 2.27.0

Released on October 30, 2025. To update to version **2.27.0**, select the Downloads section.

Features

- Added the `--exclude` option to the `ydb import s3` command, allowing schema objects to be excluded from the import if their names match a pattern.
- Added transfer objects support to the `ydb tools dump` command and `ydb tools restore` command.
- Added a new `--retention-period` option to the `ydb topic` subcommands. Usage of the legacy `--retention-period-hours` option is discouraged.
- The `ydb topic consumer add` command now has a new `--availability-period` option, which overrides the consumer's retention guarantee.
- The `ydb workload vector` commands now support `build-index` and `drop-index` subcommands.
- ***(Requires server v26.1+)*** Added the `ydb admin cluster state fetch` command to collect information about cluster nodes' state and metrics.

Bug fixes

- Fixed a bug where the `ydb debug ping` command crashed on any error.

Version 2.26.0

Released on September 25, 2025. To update to version **2.26.0**, select the Downloads section.

Features

- Added the `--no-merge` and `--no-cache` options to the `ydb monitoring healthcheck` command.
- Added query compilation time statistics to the `ydb workload * run` commands.
- Added the `--retries` option to the `ydb tools restore` command, which allows setting the number of retries for every upload data request.
- ***(Requires server v25.4+)*** Added the `--replace-sys-acl` option to the `ydb tools restore` command, which specifies whether to replace the ACL for system objects.

Version 2.25.0

Released on September 1, 2025. To update to version **2.25.0**, select the Downloads section.

Features

- Added final execute statistics to the `ydb workload * run` commands.
- Added the `--start-offset` option to the `ydb topic read` command, which specifies a starting position for reading from the selected partition.
- ***(Requires server v25.3+)*** Added a new paths approach in the `ydb export s3` and `ydb import s3` commands with the new `--include` option instead of the `--item` option.

- ***(Requires server v25.3+)*** Added support for encryption features in the `ydb export s3` and `ydb import s3` commands.
- ***(Requires server v25.3+) (Experimental)*** Added the `ydb admin cluster bridge` commands to manage a cluster in bridge mode: `list`, `switchover`, `failover`, `takedown`, `rejoin`.

Improvements

- User and password authentication options are now parsed independently, allowing them to be sourced from different priority levels. For example, the username can be specified via the `--user` option while the password is retrieved from the `YDB_PASSWORD` environment variable.
- Changed the default logging level from `EMERGENCY` to `WARN` for commands that support multiple verbosity levels.

Backward incompatible changes

- Removed the `--float-mode` option from the `ydb workload tpch run` and `ydb workload tpcds run` commands. Float mode is now inferred automatically from the table schema created during the `init` phase.

Bug fixes

- Fixed a bug where the `ydb import file csv` command with the `--newline-delimited` option could get stuck when processing input with invalid data.
- Fixed an issue with the progress bar display in the `ydb workload clickbench import files` command: incorrect percentage values and excessive line breaks caused duplicate progress lines.
- Fixed a bug where the `ydb workload topic write` command could crash with an `Unknown AckedMessageId` error due to an internal race condition.
- Fixed decimal type comparison in the `ydb workload * run` commands.

Version 2.24.1

Released on July 28, 2025. To update to version **2.24.1**, select the Downloads section.

Bug fixes

- Fixed a bug where the `ydb tools dump` command silently skipped schema objects of unsupported types and created empty directories for them in the destination folder on the file system.

Version 2.24.0

Released on July 23, 2025. To update to version **2.24.0**, select the Downloads section.

Features

- Added the ability for the `ydb workload tpch` and `ydb workload tpcds` commands to use a fractional value for the `--scale` option, specifying a percentage of the full benchmark's data size and workload.
- Added the `ydb workload tpcc check` command to check TPC-C data consistency.

Improvements

- Changed the default storage type in `ydb workload * init` commands to `column` (from `row`), and the default datetime mode to `datetime32` (from `datetime64`).

Bug fixes

- Fixed an issue where the `ydb import file csv` command could get stuck during execution.

Version 2.23.0

Released on July 16, 2025. To update to version **2.23.0**, select the Downloads section.

Features

- Added the `ydb workload tpcc` command to run a TPC-C benchmark.
- Added the `ydb workload vector select` command to benchmark vector index performance and recall.
- Added the `ydb tools infer csv` command to generate a `CREATE TABLE` SQL query from a CSV data file.

Improvements

- Enhanced processing of special values (`null`, `/dev/null`, `stdout`, `cout`, `console`, `stderr`, and `cerr`) for the `--output` option in the `ydb workload * run` commands.
- The `ydb workload` commands now work with absolute paths for database scheme objects.
- Improvements in the `ydb` interactive mode:
  - Added server connection check and hotkeys description.
  - Improved inline hints.
  - Added table column names completion.
  - Added schema caching.

Bug fixes

- Fixed an issue where the `ydb tools restore` command was not working on Windows.

Version 2.22.1

Released on June 17, 2025. To update to version **2.22.1**, select the Downloads section.

Bug fixes

- Fixed an issue where the certificate was not read from a file if the path to the file was specified in the profile with the `ca-file` field.
- Fixed an issue where the `ydb workload query import` and `ydb workload clickbench import files` commands displayed number of rows instead of number of bytes in progress state.

Version 2.22.0

Released on June 4, 2025. To update to version **2.22.0**, select the Downloads section.

Features

- Added scheme object names completion in interactive mode.
- Enhanced the capabilities of the `ydb workload query` command: added `ydb workload query init`, `ydb workload query import`, and `ydb workload query clean` commands, and modified the `ydb workload query run` command. Using these commands, you can initialize tables, populate them with data, perform load testing, and clean up the data afterwards.
- Added the `--threads` option to the `ydb workload clickbench run`, `ydb workload tpch run`, and `ydb workload tpcds run` commands. This option allows to specify the number of threads sending the queries.
- ***(Requires server v25.1+) (Experimental)*** Added the `ydb admin cluster config version` command to show the configuration version (V1/V2) on nodes.

Backward incompatible changes

- Removed the `--executor` option from the `ydb workload * run` commands. The `generic` executor is now always used.

Bug fixes

- Fixed an issue where the `ydb workload * clean` commands were deleting all contents from the target directory, instead of just the tables created by the init command.

Version 2.21.0

Released on May 22, 2025. To update to version **2.21.0**, select the Downloads section.

Features

- Added the `--no-discovery` global option, which allows skipping discovery and connecting to a user-provided endpoint directly.
- Added new options for workload commands:
  - Added the `--scale` option to the `ydb workload tpch init` and `ydb workload tpcds init` commands to set the percentage of the benchmark's data size and workload to use, relative to full scale.
  - Added the `--retries` option to the `ydb workload <clickbench|tpch|tpcds> run` commands to specify maximum retry count for every request.
  - Added the `--partition-size` option to the `ydb workload <clickbench|tpcds|tpch> init` commands to set maximum partition size in megabytes for row tables.
  - Added date range parameters (`--date-to`, `--date-from`) to the `ydb workload log run` command to support uniform primary key distribution.
- Enhanced backup and restore functionality:
  - Added the `--replace` and `--verify-existence` options to the `ydb tools restore` command to control the removal of existing objects that match those in the backup before restoration.
  - Improved the `ydb tools dump` command by not saving replica tables with ASYNC REPLICATION and their changefeeds to local backups. It prevents duplication of changefeeds and reduces the amount of space the backup takes on disk.
- Enhanced CLI usability:
  - Detailed help message (`-hh`) now shows the whole subcommand tree.
  - Added automatic pair insertion for brackets in `ydb` interactive mode.
  - Added support for files with BOM (Byte Order Mark) in the `ydb import file` commands.
- ***(Requires server v25.1+) (Experimental)*** Improved the `ydb debug latency` command:
  - Added the `--min-inflight` parameter to set minimum number of concurrent requests (default: 1).
  - Added the `--percentile` option to specify custom latency percentiles.
  - Enhanced the output with additional gRPC ping measurements.

Bug fixes

- The `ydb operation get` command now properly handles running operations.
- Fixed errors in the `ydb scheme rmdir` command:
  - Fixed an issue where the command was trying to delete subdomains.
  - Fixed deletion order: external tables are now deleted before external data sources due to possible dependencies between them.
  - Added support for coordination nodes in recursive removal.
- Fixed return code of the `ydb workload * run --check-canonical` command when results differ from canonical ones.
- Fixed an issue where CLI was attempting to read parameters from stdin even without available data.
- ***(Requires server v25.1+) (Experimental)*** Fixed an authorization error in the `ydb admin database restore` command when restoring from a backup containing multiple database administrator accounts.

Version 2.20.0

Released on March 5, 2025. To update to version **2.20.0**, select the Downloads section.

Features

- Added topics support in the `ydb tools dump` and `ydb tools restore` commands. In this release, only topic settings are retained; messages are not included in the backup.
- Added coordination nodes support in the `ydb tools dump` and `ydb tools restore` commands.
- Added the new `ydb workload log import generator` command.
- Added new global options for client certificates in SSL/TLS connections:
  - `--client-cert-file`: File containing a client certificate for SSL/TLS connections (PKCS#12 or PEM-encoded).
  - `--client-cert-key-file`: File containing a PEM-encoded private key for the client certificate.
  - `--client-cert-key-password-file`: File containing a password for the private key (if the key is encrypted).
- Queries in the `ydb workload run` command are now executed in random order.

- ***(Requires server v25.1+)*** Added support for external data sources and external tables in the `ydb tools dump` and `ydb tools restore` commands.
- ***(Experimental)*** Added the `ydb admin node config init` command to initialize a directory with node configuration files.
- ***(Requires server v25.1+) (Experimental)*** Added the `ydb admin cluster config generate` command to generate a dynamic configuration file from a cluster static configuration file.
- ***(Requires server v25.1+) (Experimental)*** Added the command `ydb admin cluster dump` and the command `ydb admin cluster restore` for dumping all cluster-level data. These dumps contain a list of databases with metadata, users, and groups but do not include schema objects.
- ***(Requires server v25.1+) (Experimental)*** Added the `ydb admin database dump` and `ydb admin database restore` commands for dumping all database-level data. These dumps contain database metadata, schema objects, their data, users, and groups.
- ***(Requires server v25.1+) (Experimental)*** Added the `--dedicated-storage-section` and `--dedicated-cluster-section` options to the `ydb admin cluster config fetch` command, allowing cluster and storage config sections to be fetched separately.

Bug fixes

- Fixed a bug where the `ydb auth get-token` command attempted to authenticate twice: once while listing endpoints and again while executing the actual token request.
- Fixed a bug where the `ydb import file csv` command was saving progress even if a batch upload had failed.
- Fixed a bug where some errors could be ignored when restoring from a local backup with the `ydb tools restore` command.
- Fixed a memory leak in the data generator for the `ydb workload tpcds` benchmark.

Version 2.19.0

Released on February 5, 2025. To update to version **2.19.0**, select the Downloads section.

Features

- Added changefeeds support in the `ydb tools dump` and `ydb tools restore` commands.
- Added `CREATE TABLE` text suggestion on schema error during the `ydb import file csv` command.
- Added statistics output on the current progress of the query in the `ydb workload` command.
- Added query text to the error message if a query fails in the `ydb workload run` command.
- Added a message if the global timeout expired in the `ydb workload run` command.
- ***(Requires server v25.1+)*** Added views support in the `ydb export s3` and `ydb import s3`. Views are exported as `CREATE VIEW` YQL statements, which are executed on import.
- ***(Requires server v25.1+)*** Added the `--skip-checksum-validation` option to the `ydb import s3` command to skip server-side checksum validation.
- ***(Requires server v25.1+) (Experimental)*** Added new options for the `ydb debug ping` command: `--chain-length`, `--chain-work-duration`, `--no-tail-chain`.
- ***(Requires server v25.1+) (Experimental)*** Added new options for the `ydb admin storage fetch` command: `--dedicated-storage-section` and `--dedicated-cluster-section`.
- ***(Requires server v25.1+) (Experimental)*** Added new options for the `ydb admin storage replace` command: `--filename`, `--dedicated-cluster-yaml`, `--dedicated-storage-yaml`, `--enable-dedicated-storage-section` and `--disable-dedicated-storage-section`.

Bug fixes

- Fixed a bug where the arm64 YDB CLI binary was downloading the amd64 binary to replace itself during the `ydb update` command. To update already installed binaries to the latest arm64 version, YDB CLI should be reinstalled.
- Fixed the return code of the `ydb workload run` command.
- Fixed a bug where the `ydb workload tpch import generator` and `ydb workload tpcds import generator` commands were failing because not all tables had been created.
- Fixed a bug with backslashes in the `ydb workload` commands paths on Windows.

Version 2.18.0

Released on December 24, 2024. To update to version **2.18.0**, select the Downloads section.

Features

- Added support for views in local backups: `ydb tools dump` and `ydb tools restore`. Views are backed up as `CREATE VIEW` queries saved in the `create_view.sql` files, which can be executed to recreate the original views.
- Added new options to the `ydb workload topic run` command: `--tx-commit-interval` and `--tx-commit-messages`, allowing you to specify the interval between transaction commits in milliseconds or in the number of messages written, respectively.
- Made the `--consumer` flag in the `ydb topic read` command optional. In the non-subscriber reading mode, the partition IDs must be specified with the `--partition-ids` option. In this case, the read is performed without saving the offset commit.
- The `ydb import file csv` command now saves the import progress. Relaunching the import command will resume the process from the row where it was interrupted.
- In the `ydb workload kv` and `ydb workload stock` commands, the default value of the `--executer` option has been changed to `generic`, which makes them no longer rely on the legacy query execution infrastructure.
- Replaced the CSV format with Parquet for filling tables in the `ydb workload` benchmarks.
- ***(Requires server v25.1+) (Experimental)*** Added new `ydb admin storage` command with `fetch` and `replace` subcommands to manage server storage configuration.

Backward incompatible changes

- Replaced the `--query-settings` option with `--query-prefix` in the `ydb workload * run` command.

Bug fixes

- Fixed a bug where the `ydb workload * run` command could crash in `--dry-run` mode.

- Fixed a bug in the `ydb import file csv` where multiple columns with escaped quotes in the same row were parsed incorrectly.

Version 2.17.0

Released on December 4, 2024. To update to version **2.17.0**, select the Downloads section.

Features

- ***(Requires server v25.1+) (Experimental)*** Added the `ydb debug ping` command for performance and connectivity debugging.

Performance

- Improved performance of parallel importing data from the file system using the `ydb tools restore` command.

Bug fixes

- Fixed a bug in the table schema created by the `ydb workload tpch` command where the `partsupp` table contained an incorrect list of key columns.
- Resolved an issue where the `ydb tools restore` command failed with the error "Too much data" if the maximum value of the `--upload-batchbytes` option was set to 16 MB.

Version 2.16.0

Released on November 26, 2024. To update to version **2.16.0**, select the Downloads section.

Features

- Improved throughput of the `ydb import file csv` command by up to 3 times.
- Added support for running the stock benchmark with column-oriented tables.
- Added support for ISO 8601–formatted timestamps in the `ydb topic` commands.
- Added the `--explain-ast` option to the `ydb sql` command, which prints the query AST.
- Added ANSI SQL syntax highlighting in interactive mode.
- Added support for PostgreSQL syntax in the `ydb workload tpch` and `ydb workload tpcds` benchmarks.
- Introduced the `-c` option for the `ydb workload tpcds run` command to compare results with expected values and display differences.
- Added log events for the `ydb tools dump` and `ydb tools restore` commands.
- Enhanced the `ydb tools restore` command to display error locations.

Backward incompatible changes

- Changed the default value of the `ydb topic write` command's `--codec` option to `RAW`.

Bug fixes

- Fixed the progress bar in the `ydb workload import` command.
- Resolved an issue where restoring from a backup using the `--import-data` option could fail if the table's partitioning had changed.

Version 2.10.0

Released on June 24, 2024. To update to version **2.10.0**, select the Downloads section.

Features

- Added the `ydb sql` command that runs over QueryService and can execute any DML/DDL command.
- Added `notx` support for the `--tx-mode` option in the `ydb table query execute` command.
- Added start and end times for long-running operation descriptions (export, import).
- Added replication description support in the `ydb scheme describe` and `ydb scheme ls` commands.
- Added big datetime types support: `Date32`, `Datetime64`, `Timestamp64`, `Interval64`.
- `ydb workload` commands rework:
  - Added the `--clear` option to the `init` subcommand, allowing tables from previous runs to be removed before workload initialization.
  - Added the `ydb workload * import` command to prepopulate tables with initial content before executing benchmarks.

Backward incompatible changes

- `ydb workload` commands rework:
  - The `--path` option was moved to a specific workload level. For example: `ydb workload tpch --path some/tables/path init ...`.
  - The `--store=s3` option was changed to `--store=external-s3` in the `init` subcommand.

Bug fixes

- Fixed colors in the `PrettyTable` format

Version 2.9.0

Released on April 25, 2024. To update to version **2.9.0**, select the Downloads section.

Features

- Improved query logical plan tables: added colors, more information, fixed some bugs.
- The verbose option `-v` is supported for the `ydb workload` commands to provide debug information.
- Added an option to run the `ydb workload tpch` command with an S3 source to measure federated queries performance.
- Added the `--rate` option for `ydb workload` commands to control the transactions (or requests) per second limit.

- Added the `--use-virtual-addressing` option for S3 import/export, allowing the switch to virtual hosting of buckets for the S3 path layout.
- Improved the `ydb scheme ls` command performance due to listing directories in parallel.

Bug fixes

- Resolved an issue where extra characters were truncated during line transfers in CLI tables.
- Fixed invalid memory access in `tools restore`.
- Fixed the issue of the `--timeout` option being ignored in generic and scan queries, as well as in the import command.
- Added a 60-second timeout to version checks and CLI binary downloads to prevent infinite waiting.
- Minor bug fixes.

Version 2.8.0

Released on January 12, 2024. To update to version **2.8.0**, select the Downloads section.

Features

- Added new `ydb admin config` and `ydb admin volatile-config` commands for cluster configuration management.
- Added support for loading PostgreSQL-compatible data types by ydb import file csv|tsv|json command. Only for row-oriented tables.
- Added support for directory load from an S3-compatible storage in the ydb import s3 command. Currently only available on Linux and Mac OS.
- Added support for outputting the results of ydb table query execute, ydb yql and ydb scripting yql commands in the Apache Parquet format.
- In the ydb workload commands, the `--executer` option has been added, which allows to specify which type of queries to use.
- Added a column with median benchmark execution time in the statistics table of the ydb workload clickbench command.
- ***(Experimental)*** Added the `generic` request type to the ydb table query execute command, which allows performing DDL and DML operations, returning arbitrarily-sized results, and supporting MVCC. The command uses an experimental API, compatibility is not guaranteed.
- ***(Experimental)*** In the `ydb table query explain` command, the `--collect-diagnostics` option has been added to collect query diagnostics and save it to a file. The command uses an experimental API, compatibility is not guaranteed.

Bug fixes

- Fixed an error displaying tables in `pretty` format with Unicode characters.
- Fixed an error substituting the wrong primary key in the command `ydb tools pg-convert`.

Version 2.7.0

Released on October 23, 2023. To update to version **2.7.0**, select the Downloads section.

Features

- Added the `ydb tools pg-convert` command, which prepares a dump obtained by the pg_dump utility for loading into the YDB postgres-compatible layer.
- Added the `ydb workload query` load testing command, which loads the database with script execution queries in multiple threads.
- Added new `ydb scheme permissions list` command to list permissions.
- In the commands ydb table query execute, ydb table query explain, ydb yql, and ydb scripting yql, the `--flame-graph` option has been added, specifying the path to the file in which you need to save the visualization of query execution statistics.
- Special commands in the interactive query execution mode are now case-insensitive.
- Added validation for special commands and their parameters.
- Added table reading in the scenario with transactions in the command ydb workload transfer topic-to-table run.
- Added the `--commit-messages` option to the command ydb workload transfer topic-to-table run, specifying the number of messages in a single transaction.
- Added the options `--only-table-in-tx` and `--only-topic-in-tx` in the command ydb workload transfer topic-to-table run, specifying restrictions on the types of queries in a single transaction.
- Added new columns `Select time` and `Upsert time` in the statistics table in the command ydb workload transfer topic-to-table run.

Bug fixes

- Fixed an error when loading an empty JSON list by commands: ydb table query execute, ydb scripting yql and ydb yql.

Version 2.6.0

Released on September 7, 2023. To update to version **2.6.0**, select the Downloads section.

Features

- Added `--path` option to ydb workload tpch run, which contains the path to the directory with tables created by the ydb workload tpch init command.
- Added ydb workload transfer topic-to-table run command, which loads the database with read requests from topics and write requests to the table.
- Added the option `--consumer-prefix` in the commands ydb workload topic init, ydb workload topic run read|full, specifying prefixes of consumer names.
- Added the `--partition-ids` option in the ydb topic read command, which specifies a comma-separated list of topic partition identifiers to read from.
- Added support for CSV and TSV parameter formats in YQL query execution commands.
- The interactive mode of query execution has been redesigned. Added new interactive mode specific commands: `SET`, `EXPLAIN`, `EXPLAIN AST`. Added saving history between CLI launches and auto-completion of YQL queries.
- Added the command ydb config info, which outputs the current connection parameters without connecting to the database.
- Added the command ydb workload kv run mixed, which loads the database with write and read requests.

- The `--percentile` option in the ydb workload topic run write|read|full commands can now take floating point values.
- The default values for the `--seconds` and `--warmup` options in the ydb workload topic run write|read|full commands have been increased to 60 seconds and 5 seconds, respectively.
- Changed the default value for the `--supported-codecs` option to `RAW` in the ydb topic create and ydb topic consumer add commands.

Bug fixes

- Fixed string loss when loading with the ydb import file json command.
- Fixed ignored statistics during the warm-up of commands ydb workload topic run write|read|full.
- Fixed incomplete statistics output in the ydb scripting yql and ydb yql commands.
- Fixed incorrect output of progress bar in ydb tools dump and ydb tools restore commands.
- Fixed loading large files with the header in the ydb import file csv|tsv command.
- Fixed hanging of the ydb tools restore --import-data command.
- Fixed error `Unknown value Rejected` when executing the ydb operation list build index command.

Version 2.5.0

Released on June 20, 2023. To update to version **2.5.0**, select the Downloads section.

Features

- For the `ydb import file` command, a parameter --timeout has been added that specifies the time within which the operation should be performed on the server.
- Added a progress bar in commands ydb scheme rmdir --recursive and ydb import file.
- Added the command ydb workload kv run read-rows, which loads the database with requests to read rows using a new experimental API call ReadRows (implemented only in the main branch), which performs faster key reading than select.
- New parameters `--warmup-time`, `--percentile`, `--topic` have been added to the ydb workload topic, setting the test warm-up time, the percentile in the statistics output and the topic name, respectively.
- Added the ydb workload tpch command to run the TPC-H benchmark.
- Added the `--ordered` flag in the command ydb tools dump, which preserves the order by primary key in tables.

Performance

- The data loading speed in the `ydb import file` command has been increased by adding parallel loading. The number of threads is set by the new parameter --threads.
- A performance of the ydb import file json command has been increased by reducing the number of data copies.

Version 2.4.0

Released on May 24, 2023. To update to version **2.4.0**, select the Downloads section.

Features

- Added the ability to upload multiple files in parallel with the command ydb import file.
- Added support for deleting column tables for the command ydb scheme rmdir --recursive.
- Improved stability of the command ydb workload topic.

Version 2.3.0

Release date: May 1, 2023. To update to version **2.3.0**, select the Downloads section.

Features

- Added the interactive mode of query execution. To switch to the interactive mode, run ydb yql without arguments. This mode is experimental: backward compatibility is not guaranteed yet.
- Added the ydb index rename command for atomic replacement or renaming of a secondary index.
- Added the `ydb workload topic` command for generating the load that reads messages from topics and writes messages to topics.
- Added the --recursive option for the `ydb scheme rmdir` command. Use it to delete a directory recursively, with all its content.
- Added support for the `topic` and `coordination node` types in the ydb scheme describe command.
- Added the --commit option for the `ydb topic consumer` command. Use it to commit messages you have read.
- Added the --columns option for the `ydb import file csv|tsv` command. Use it as an alternative to the file header when specifying a column list.
- Added the --newline-delimited option for the `ydb import file csv|tsv` command. Use it to make sure that your data is newline-free. This option streamlines import by reading data from several file sections in parallel.

Bug fixes

- Fixed the bug that resulted in excessive memory and CPU utilization when executing the `ydb import file` command.

Version 2.2.0

Release date: March 3, 2023. To update to version **2.2.0**, select the Downloads section.

Features

- Fixed the error that didn't allow specifying supported compression algorithms when adding a topic consumer.
- Added support for streaming YQL scripts and queries based on options transferred via `stdin`.
- You can now use a file to provide YQL query options
- Password input requests are now output to `stderr` instead of `stdout`.
- You can now save the root CA certificate path in a profile.
- Added a global option named --profile-file to use the specified file as storage for profile settings.
- Added a new type of load testing: ydb workload clickbench.

Version 2.1.1

Release date: December 30, 2022. To update to version **2.1.1**, select the Downloads section.

Improvements

- Added support for the `--stats` option of the ydb scheme describe command for column-oriented tables.
- Added support for Parquet files to enable their import with the ydb import command.
- Added support for additional logging and retries for the ydb import command.

Version 2.1.0

Release date: November 18, 2022. To update to version **2.1.0**, select the Downloads section.

Features

- You can now create a profile non-interactively.
- Added the ydb config profile update and ydb config profile replace commands to update and replace profiles, respectively.
- Added the `-1` option for the ydb scheme ls command to enable output of a single object per row.
- You can now save the IAM service URL in a profile.
- Added support for username and password-based authentication without specifying the password.
- Added support for AWS profiles in the ydb export s3 command.
- You can now create profiles using `stdin`. For example, you can pass the YC CLI `yc ydb database get information` command output to the `ydb config profile create` command input.

Bug fixes

- Fixed the error when request results were output in JSON-array format incorrectly if they included multiple server responses.
- Fixed the error that disabled profile updates so that an incorrect profile was used.

Version 2.0.0

Release date: September 20, 2022. To update to version **2.0.0**, select the Downloads section.

Features

- Added the ability to work with topics:
  - `ydb topic create`: Create a topic.
  - `ydb topic alter`: Update a topic.
  - `ydb topic write`: Write data to a topic.
  - `ydb topic read`: Read data from a topic.
  - `ydb topic drop`: Delete a topic.
- Added a new type of load testing:
  - `ydb workload kv init`: Create a table for kv load testing.
  - `ydb workload kv run`: Apply one of three types of load: run multiple `UPSERT` sessions, run multiple `INSERT` sessions, or run multiple sessions of GET requests by primary key.
  - `ydb workload kv clean`: Delete a test table.
- Added the ability to disable current active profile (see the `ydb config profile deactivate` command).
- Added the ability to delete a profile non-interactively with no commit (see the `--force` option under the `ydb config profile remove` command).
- Added CDC support for the `ydb scheme describe` command.
- Added the ability to view the current DB status (see the `ydb monitoring healthcheck` command).
- Added the ability to view authentication information (token) to be sent with DB queries under the current authentication settings (see the `ydb auth get-token` command).
- Added the ability for the `ydb import` command to read data from stdin.
- Added the ability to import data in JSON format from a file or stdin (see the `ydb import file json` command).

Improvements

- Improved command processing. Improved the accuracy of user input parsing and validation.

Version 1.9.1

Release date: June 25, 2022. To update to version **1.9.1**, select the Downloads section.

Features

- Added the ability to compress data when exporting it to S3-compatible storage (see the `--compression` option of the ydb export s3 command).
- Added the ability to manage new YDB CLI version availability auto checks (see the `--disable-checks` and `--enable-checks` options of the ydb version command).

Security changelog

Fixed in YDB 22.4.44, 2022-11-28

CVE-2022-28228

Out-of-bounds read was discovered in YDB server. An attacker could construct a query with an insert statement that would allow them to access confidential information or cause a crash.

Link to CVE: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-28228.

Credits: Maxim Arnold.

Fixed in YDB Go SDK v3.53.3, 2023-10-17

CVE-2023-45825

Token in custom credentials object can leak through logs.

Link to CVE: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-45825.

Credits: Sergey Foster.

Fixed in YDB 25.3.1.25, 2026-04-03

CVE-2026-10549

LDAP filter injection vulnerability allows an attacker with valid LDAP credentials to bypass group membership checks and gain unauthorized access to the database.

Link to CVE: https://www.cve.org/CVERecord?id=CVE-2026-10549.