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

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.

> **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.

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
```

5. Apply the manifest for creating a YDB cluster:

```
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. The created database will be accessible from inside the Kubernetes cluster by the `database-minikube-sample` DNS name on port 2135.
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
```

6. Apply the manifest for creating a storage:

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

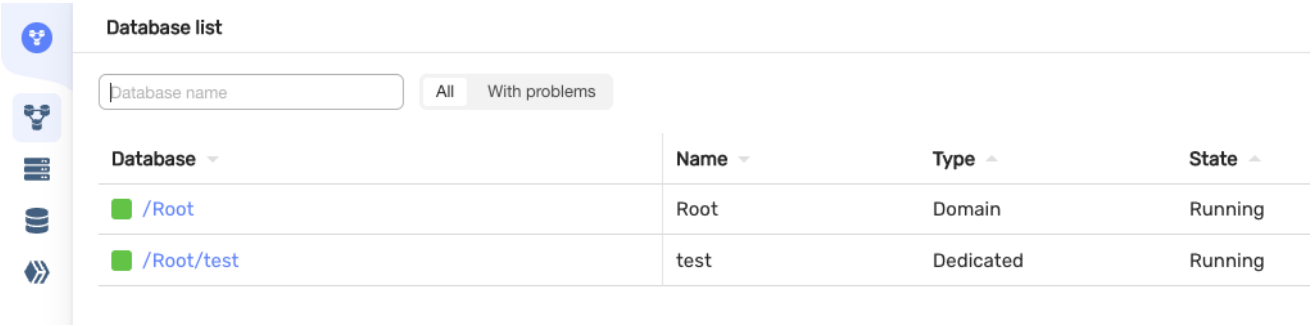
7. Wait for `kubectl get storages.ydb.tech` to become `Ready`.
8. Apply the manifest for creating a database:

```
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. The created database will be accessible from inside the Kubernetes cluster by the `database-kind-sample` DNS name on port 2135.
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

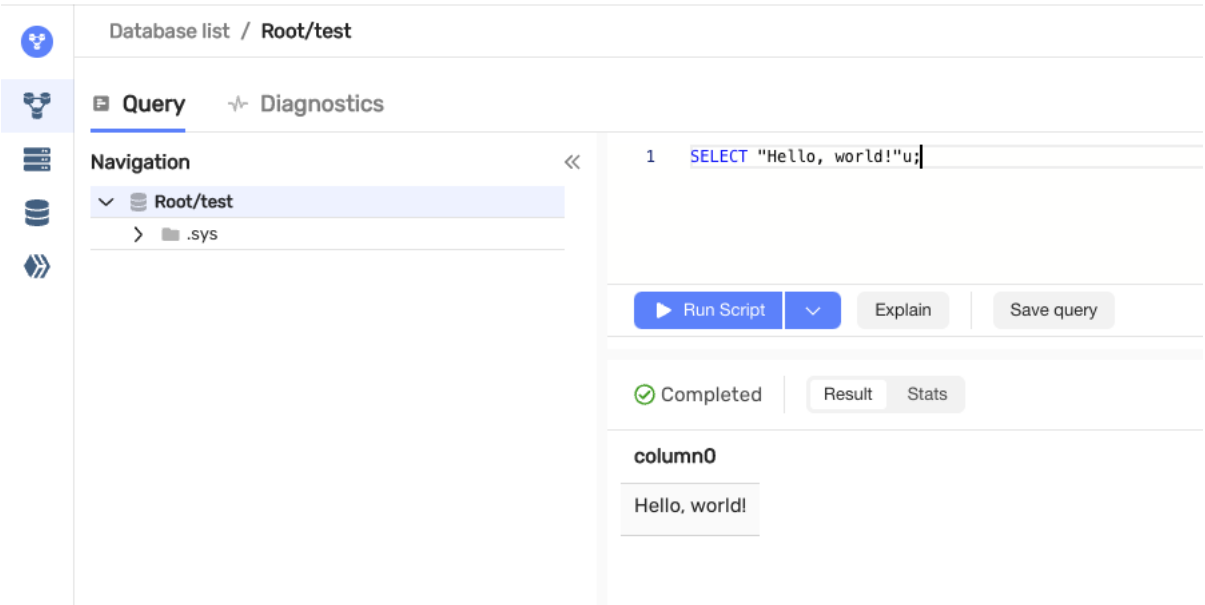
The simplest way to launch your first YDB query is via the built-in web interface. It is launched by default on port 8765 of the YDB server. If you have launched it locally, open localhost:8765 in your web browser. If not, replace `localhost` with your server's hostname in this URL or use `ssh -L 8765:localhost:8765 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, while `/Root/test` or `/local` (depending on the chosen installation method) is a playground created during installation in the previous step. Click on either `/Root/test` or `/local`, 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.

YDB Overview

YDB is a horizontally scalable, distributed, fault-tolerant DBMS. It is designed for high performance, with a typical server capable of handling tens of thousands of queries per second. The system is designed to handle hundreds of petabytes of data. YDB can operate in both single data center and geo-distributed (cross data center) modes on a cluster of thousands of servers.

YDB provides:

- Strict consistency, which can be relaxed to increase performance.
- Support for queries written in YQL, an SQL dialect for working with big data.
- Automatic data replication.
- High availability with automatic failover if a server, rack, or availability zone goes offline.
- Automatic data partitioning as data or load grows.

To interact with YDB, you can use the YDB CLI and SDK for C++, C#, Go, Java, Node.js, PHP, Python, and Rust.

YDB supports a relational data model and manages row-oriented and column-oriented tables with a predefined schema. Directories can be created like in a file system to organize tables. In addition to tables, YDB supports topics for storing unstructured messages and delivering them to multiple subscribers.

Database commands are mainly written in YQL, an SQL dialect, providing a powerful and familiar way to interact with the database.

YDB supports high-performance distributed ACID transactions that may affect multiple records in different tables. It provides the serializable isolation level, the strictest transaction isolation, with the option to reduce the isolation level to enhance performance.

YDB natively supports different processing options, such as OLTP and OLAP. The current version offers limited analytical query support, which is why YDB is currently considered an OLTP database.

YDB is an open-source system. The source code is available under the Apache License 2.0. Client applications interact with YDB based on gRPC, which has an open specification, allowing for SDK implementation in any programming language.

Use Cases

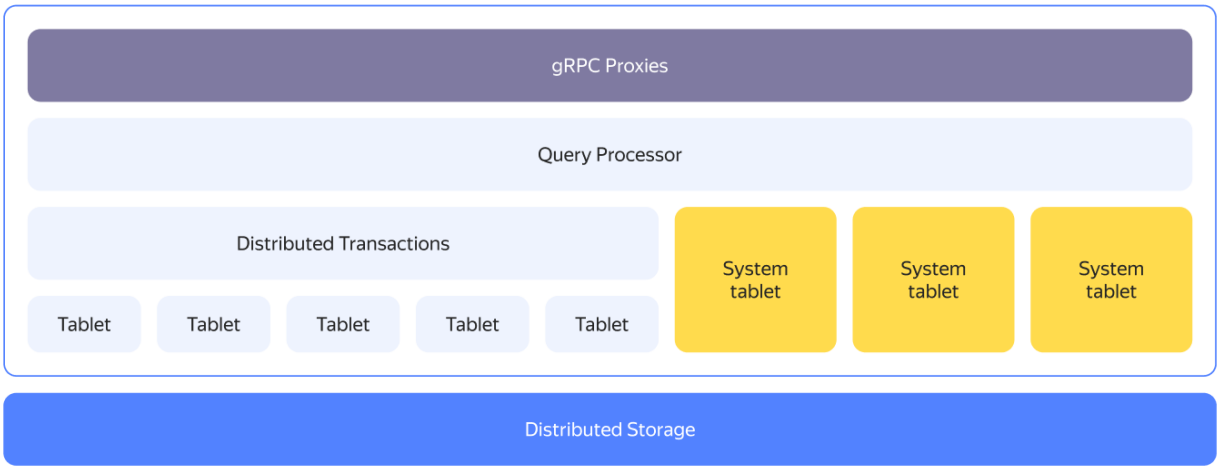
YDB can be used as an alternative solution in the following cases:

- When using NoSQL systems, if strong data consistency is required.
- When using NoSQL systems, if you need to make transactional updates to data stored in different rows of one or more tables.
- In systems that need to process and store large amounts of data and allow for virtually unlimited horizontal scalability (using industrial clusters of 5000+ nodes, processing millions of RPS, and storing petabytes of data).
- In low-load systems, when supporting a separate DB instance would be a waste of money (consider using YDB in serverless mode instead).
- In systems with unpredictable or seasonally fluctuating load (you can add/reduce computing resources on request and/or in serverless mode).
- In high-load systems that shard load across relational DB instances.
- When developing a new product with no reliable load forecast or with an expected high load beyond the capabilities of conventional relational databases.
- In projects where the simultaneous handling of transactional and analytical workloads is required.

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

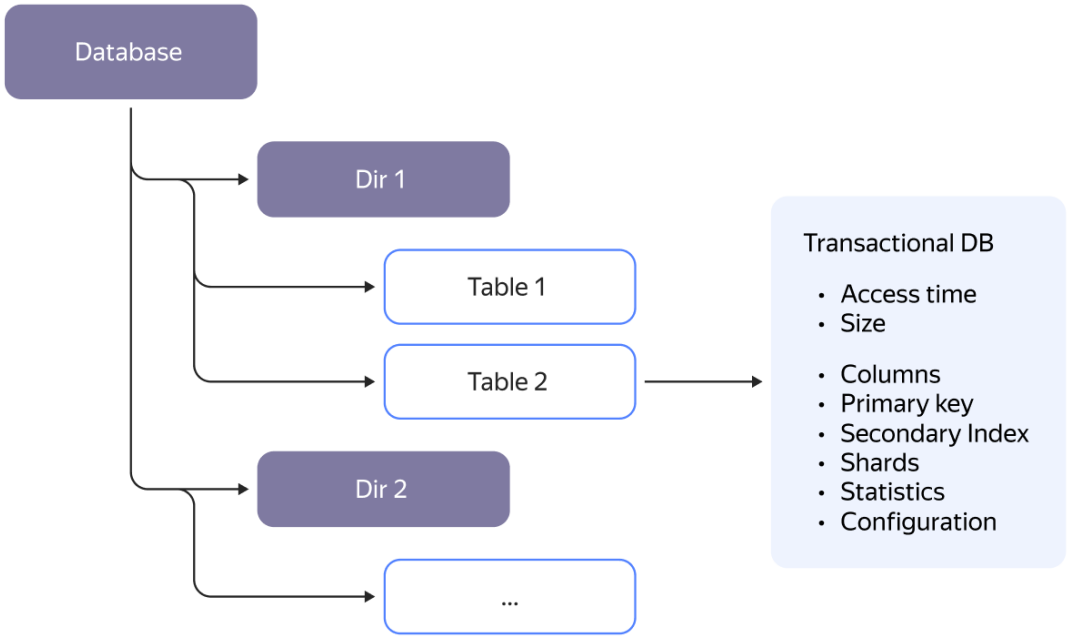


YDB clusters typically run on commodity hardware with a shared-nothing architecture. From a bird's eye view, YDB exhibits a layered architecture. The compute and storage layers are disaggregated; they can either run on separate sets of nodes or be co-located.

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.

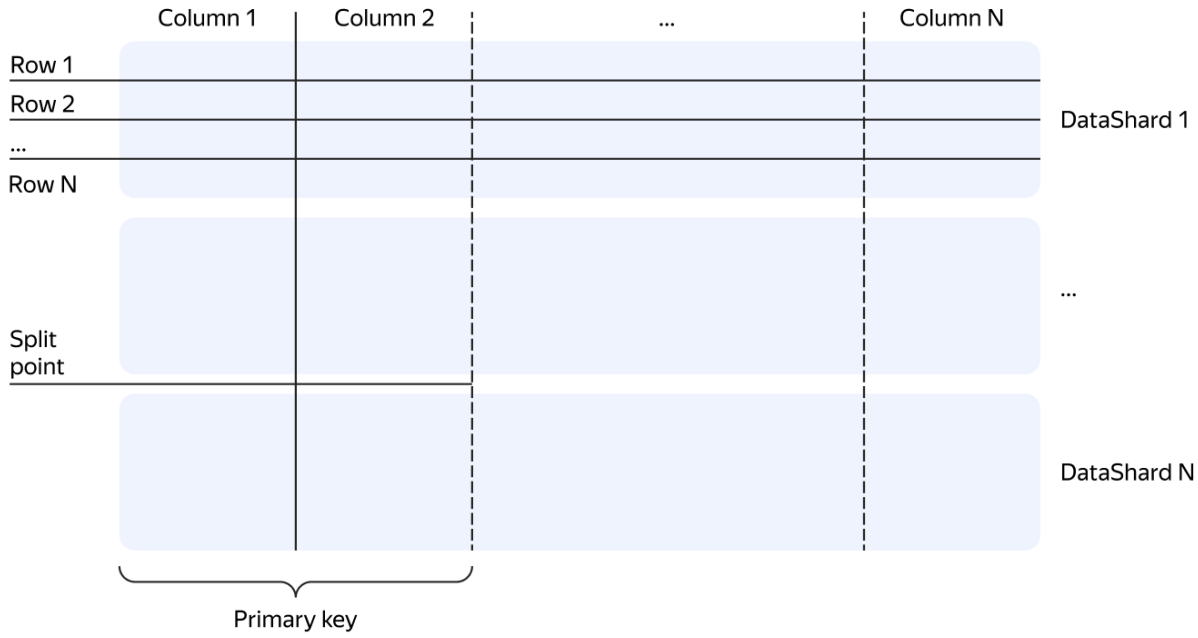
The next level of detail of the overall YDB architecture is explained in the General YDB schema article.

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.

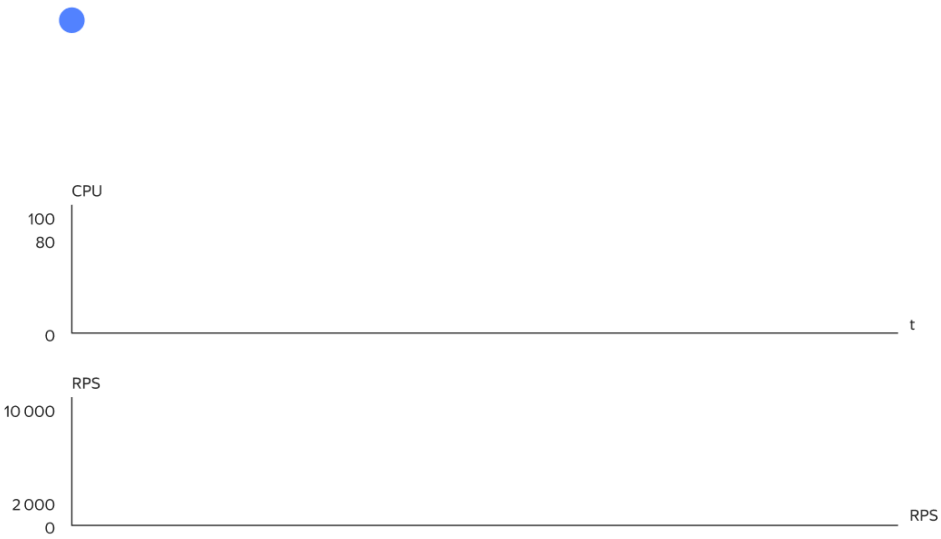
Regardless of the type, each table must have a primary key. Column-oriented tables can only have `NOT NULL` columns in primary keys. Table data is physically sorted by the primary key.

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



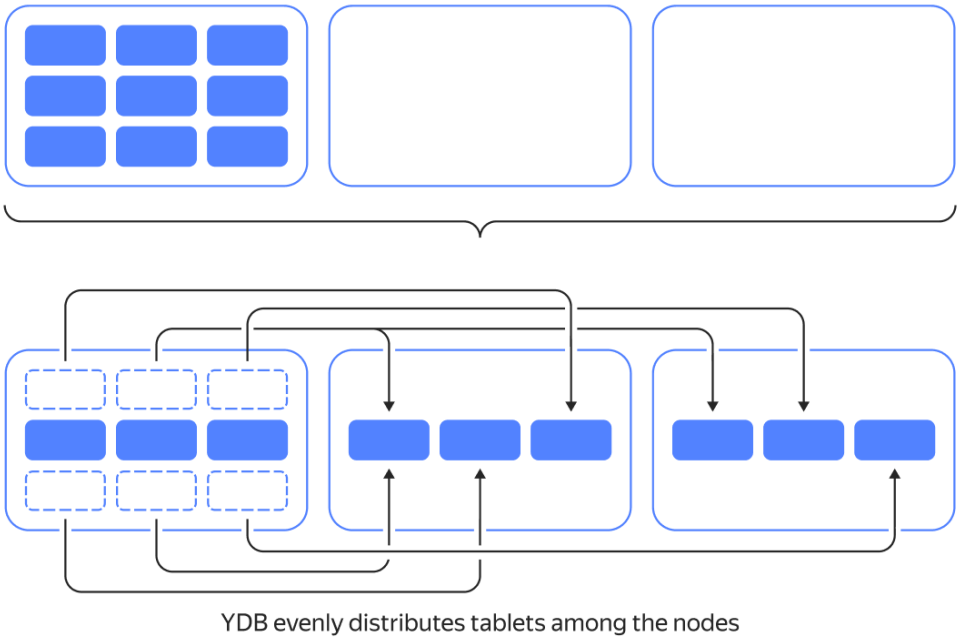
Data shards will automatically split into more as the load increases. They automatically merge back to the appropriate number when the peak load subsides.

Split by Size



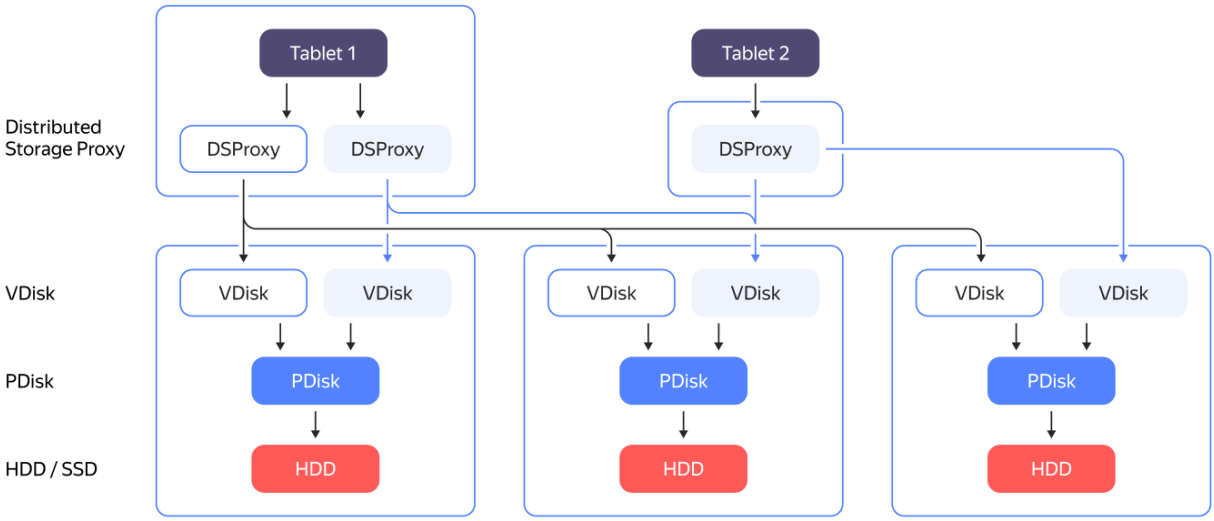
Data shards will also automatically split when the data size increases. They automatically merge back if enough data is deleted.

Automatic Balancing



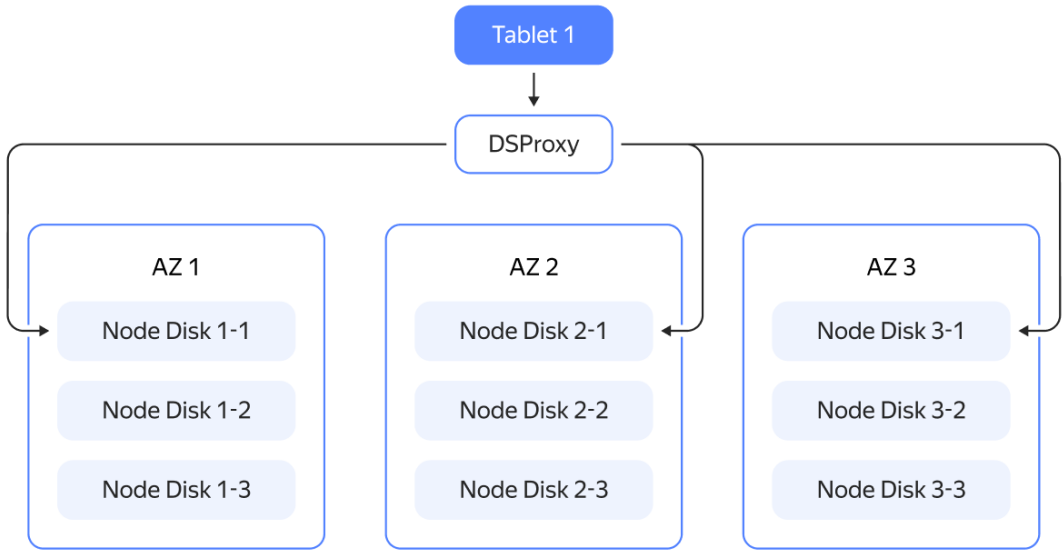
YDB evenly distributes tablets among available nodes. It moves heavily loaded tablets from overloaded nodes. CPU, memory, and network metrics are tracked to facilitate this.

Distributed Storage Internals



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 for DevOps Engineers

This section of YDB documentation covers everything you need to know to run a production-grade YDB cluster. It is subdivided based on what's your preferred approach to managing infrastructure:

- **Ansible**: for deployments on bare metal and virtual machines.
- **Kubernetes**: for containerized deployments.
- **Manual**: generic instructions.

Regardless of the chosen infrastructure management method, it is recommended to familiarize yourself with YDB system requirements first.

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
- YQL Tutorial - Overview
- Choosing a primary key for:
  - Row-oriented tables
  - Column-oriented tables
- Secondary indexes
- Uploading data to YDB
- Paginated output
- Using timeouts
- Database system views
- Change Data Capture
- Custom attributes in tables
- Reference:
  - YQL - Overview
  - YDB SDK reference
  - YDB CLI
  - YDB compatibility with PostgreSQL
  - Kafka API

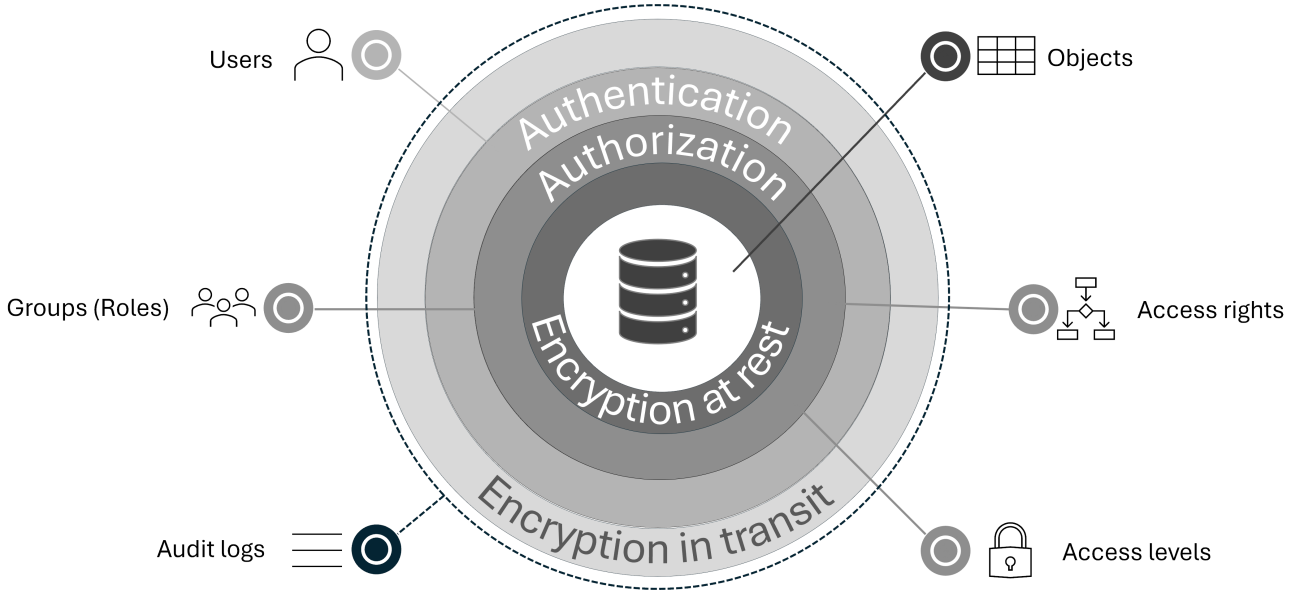
If you're interested in developing YDB core or satellite projects, refer to the documentation for contributors.

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 three access levels:
  - **Viewer** allows viewing the cluster state, which is not publicly accessible.
  - **Operator** grants additional privileges to monitor and modify the cluster state.
  - **Administrator** grants privileges to administer the YDB cluster and its databases.

  Similarly to access rights, one or more access levels can be granted to a user or a group. An access subject that does not have any access levels can view only publicly available information about the cluster. Each access level adds privileges to the access subject. For the maximum level of privileges, an access subject must have all three access levels.

  For more information about access levels, see Configuring administrative and other privileges.
- **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` configuration section.
  - **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.

YDB Development

This section contains guidelines for YDB developers and contributors.

- Working on a change
- Build and test using Ya Make build system
- Releases

YDB reference

This documentation section contains reference information about various aspects of YDB, including:

- YQL - Overview
- YDB compatibility with PostgreSQL
- Using the embedded web UI
- Integrations YDB
- YDB CLI
- YDB SDK reference
- Languages and APIs
- Kafka API
- YDB cluster configuration
- Reference on YDB observability
- YDB DSTool overview
- ydbops utility overview
- YDB Docker container reference

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:

- YDB SDK code recipes
- YDB CLI recipes
- YQL recipes

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

Questions and answers about YDB

- General
- SDK
- Errors
- YQL
- Analytics

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

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 explains terms that are useful to any person working with YDB regardless of their role and use case.

Cluster

A YDB **cluster** is a set of interconnected YDB nodes that communicate with each other to serve user queries and reliably store user data. These nodes form one of the supported cluster topologies, which directly affects the cluster's reliability and performance characteristics.

YDB clusters are multitenant and can contain multiple 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

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

Given YDB follows the approach of separated storage and compute layers, `ydbd` has multiple operation modes that determine the node type. The available node types are explained below.

Database node

Database nodes (also known as **tenant nodes** or **compute nodes**) serve user queries addressed to a specific logical database. Their state is only in memory and can be recovered from the Distributed Storage. All database nodes of a given YDB cluster can be considered its compute layer. Thus, adding database nodes and allocating extra CPU and RAM to them are the main ways to increase the database's compute resources.

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 and responsible for long-term persisting pieces of data. All storage nodes of a given YDB cluster are called Distributed Storage and can be considered the cluster's storage layer. Thus, adding extra storage nodes and their disks are the main ways to increase the cluster's storage capacity and input/output throughput.

Hybrid node

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

Static node

Static nodes are manually configured during the initial cluster initialization or re-configuration. Typically, they play the role of storage nodes, but technically, it is possible to configure them to be database nodes as well.

Dynamic node

Dynamic nodes are added and removed from the cluster on the fly. They can only play the role of 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

A **storage group**, **Distributed storage group**, or **Blob storage group** is a location for reliable data storage similar to RAID, but using disks of multiple servers. Depending on the chosen 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 group

A **static group** is a special storage group created during the initial cluster deployment. Its primary role is to store system tablet's data, which can be considered cluster-wide metadata.

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

Dynamic group

Regular storage groups that are not static are called **dynamic groups**. They are called dynamic because they can be created and decommissioned on the fly during cluster operation.

Storage pool

Storage pool is a collection of data storage devices with similar characteristics. Each storage pool is assigned a unique name within a YDB cluster. Technically, each storage pool consists of multiple PDisks. Each storage group is created in a particular storage pool, which determines the performance characteristics of the storage group through the selection of appropriate storage devices. It is typical to have separate storage pools for NVMe, SSD, and HDD devices or particular models of those devices with different capacities and speeds.

Actor

The actor model is one of the main approaches for concurrent programming, which is employed by YDB. In this model, **actors** are lightweight user-space processes that may have and modify their private state but can only affect each other indirectly through message passing. YDB has its own implementation of this model, which is covered below.

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

Tablet

A **tablet** is one of YDB's primary building blocks and abstractions. It is an entity responsible for a relatively small segment of user or system data. Typically, a tablet manages up to single-digit gigabytes of data, but some kinds of tablets can handle more.

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 a persistent state reliably saved in Distributed Storage. This state allows the tablet to continue operating on a different database node if the previous one is down or 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 tablet infrastructure allow tablets to manipulate their state using **local transactions** with serializable isolation level. Technically, they aren't really local to a single node as such a state persists remotely in Distributed Storage.
- In the context of YDB, the term **distributed transactions** usually refers to transactions involving multiple tablets. For example, cross-table or even cross-row transactions are often distributed.
- **Single-shard** transactions span a single tablet and are faster to complete. For example, transactions between rows in 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.

Interactive transactions

The term **interactive transactions** refers to transactions that are split into multiple queries and involve data processing by an application between these queries. For example:

1. Select some data.
2. Process the selected data in the application.
3. Update some data in the database.
4. Commit the transaction in a separate query.

Multi-version concurrency control

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

Topology

YDB supports several cluster topologies, described in more detail in a separate article YDB Cluster Topology. A few related terms are explained below.

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 multiple availability zones. The distance between availability zones in the same region is expected to be around 500 km or less. YDB performs synchronous data writes to each availability zone in a region, ensuring reasonable latencies and uninterrupted performance if an availability zone fails.

Rack

A **rack** or **server rack** is a piece of equipment used to mount multiple servers in an organized manner. Servers in the same rack are more likely to become unavailable simultaneously due to rack-wide issues related to electricity, cooling, etc. Thus, YDB can consider information about which server is located in which rack when placing each piece of data in bare-metal environments.

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 RAM or on disk drives: row-oriented (row-by-row) and column-oriented (column-by-column). The chosen approach greatly impacts the performance characteristics of operations with this data, with the former more suitable for transaction workloads (OLTP) and the latter for analytical (OLAP). YDB supports both.

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 independently. They are optimized for building aggregates over a small number of columns but are less suitable for accessing particular rows, as rows need to be reconstructed from their cells on the fly. They are explained 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.

The guidelines on choosing primary keys are provided in Choosing a primary key.

Primary index

A **primary index** or **primary key index** is the main data structure used to locate rows in a table. It is built based on the chosen primary key and determines the physical order of rows in a 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 locate rows in a table, typically when it can't be done efficiently using the primary index. Unlike the primary index, secondary indexes are managed independently from the main table data. Thus, a table might have multiple secondary indexes for different use cases. YDB's capabilities in terms of secondary indexes are covered in a separate article Secondary Indexes. Secondary indexes can be either unique or non-unique.

A special type of **secondary index** is singled out separately - [vector index] (#vector-index).

Vector Index

Vector index is an additional data structure used to speed up the vector search when there is a large amount of data, and the exact vector search without an index does not perform satisfactorily. The capabilities of YDB regarding vector indexes are described in a separate article Vector Indexes.

Vector index is distinct from a secondary index as it solves other tasks.

Column family

A **column family** or **column group** is a feature that allows storing a subset of row-oriented table columns separately in a distinct family or group. The primary use case is to store some columns on different kinds of disk drives (offload less important columns to HDD) or with various compression settings. If the workload requires many column families, consider using column-oriented tables instead.

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

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.

There are user-defined and system-defined views.

User-defined view

A **user-defined view** is created by a user with the CREATE VIEW statement. For more information, see View.

System view

A **system view** is for monitoring the DB status. System views are located in the .sys directory in the root of the database tree. It is explained in a separate article Database system views.

Topic

A **topic** is a persistent queue that can be used for reliable asynchronous communications between various systems via message passing. YDB provides the infrastructure to ensure "exactly once" semantics in such communications, which ensures that there are both no lost messages and no accidental duplicates.

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

Partition

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

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

Offset

An **offset** is a sequence number that identifies a message inside a partition.

Producer

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

Consumer

A **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 **stream of changes** to a given table. Technically, it is implemented on top of topics. It is described in more detail in a separate article Change Data Capture (CDC).

Changefeed

Changefeed or **stream of changes** is an ordered list of changes in a given table published via a topic.

Asynchronous replication instance

Asynchronous replication instance is a named entity that stores asynchronous replication settings (connection properties, a list of replicated objects, etc.) It can also be used to retrieve the status of asynchronous replication, such as the initial synchronization process, replication lag, errors, and more.

Replicated object

Replicated object is an object, for example, a table, that is asynchronously replicated to the target database.

Replica object

Replica object is a mirror copy of the replicated object, automatically created by an asynchronous replication instance. Replica objects are typically read-only.

Coordination node

A **coordination node** is a schema object that allows client applications to create semaphores for coordinating their actions. Learn more about coordination nodes.

Semaphore

A **semaphore** is an object within a coordination node that provides a synchronization mechanism for distributed applications. Semaphores can be persistent or ephemeral and support operations like creation, acquisition, release, and monitoring. Learn more about semaphores in YDB.

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 querying 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 Queries.

External data source

An **external data source** or **external connection** is a piece of metadata that describes how to connect to a supported external system for federated query execution.

External table

An **external table** is a piece of metadata that describes a particular dataset that can be retrieved from an external data source.

Secret

A **secret** is a sensitive piece of metadata that requires special handling. For example, secrets can be used in external data source definitions and represent things like passwords and tokens.

Authentication token

An **authentication token** or **auth token** is a token that YDB uses for authentication.

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

A **scheme root** is a root element of a YDB cluster scheme. Children elements of the cluster scheme root can be databases or other scheme objects.

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** in the context of authorization is an entity for which access rights and restrictions are configured. In YDB, access objects are scheme objects.
Each access object has an owner and an access control list.

Access subject

An **access subject** is an entity that can interact with access objects or perform specific actions within the system. Access to these interactions and actions depends on configured access control lists.

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 right inheritance refers to the mechanism by which access rights are automatically passed down from parent access objects to child access objects within a database structure. This ensures that permissions granted at a higher level in the hierarchy are applied to all sub-levels beneath it, unless explicitly overridden.

Access control list

An **access control list** or **ACL** is a list of all rights granted to access subjects (users and groups) for 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.

YDB uses three access levels:

- viewer
- operator
- administrator

An access level is granted by adding an access subject to an access level list.

Access level list

An **access level list** is a list of SIDs that grants a certain access level to the associated access subjects.

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

Owner

An **owner** is an access subject (user or group) having full rights over a specific access object.

User

A **user** is an individual utilizing YDB to perform a specific function.

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

- local users in YDB databases
- external users from third-party directory services

YDB users are identified by their SIDs.

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

An **external user** is an individual whose YDB account is created in a third-party directory service, for example, in LDAP or IAM.

Group

A **group** or **access group** is a named collection of users with identical access rights to certain access objects.

Role

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

Roles in YDB are implemented as groups that are created during the initial cluster deployment and granted a set of access rights on the root of the cluster scheme.

SID

SID (**Security Identifier**) is a string in the format `<login>[@<subsystem>]`, identifying an access subject in access control lists.

Query optimizer

Query optimizer is a YDB component that takes a logical plan as input and produces the most efficient physical plan with the lowest estimated resource consumption among the alternatives. The YDB query optimizer is described in the Query Optimization in YDB section.

Advanced terminology

This section explains terms that are useful to YDB contributors and users who want to get a deeper understanding of what's going on inside the system.

Actors implementation

Actor system

An **actor system** is a C++ library with YDB's implementation of the Actor model.

Actor service

An **actor service** is an actor that has a well-known name and is usually run in a single instance on a node.

ActorId

An **ActorId** is a unique identifier of the actor or tablet in the cluster.

Actor system interconnect

The **actor system interconnect** or **interconnect** is the [cluster's](#cluster internal network layer. All actors interact with each other within the system via the interconnect.

Local

A **Local** is an actor service running on each node. It directly manages the tablets on its node and interacts with Hive. It registers with Hive and receives commands to launch tablets.

Actor system pool

The **actor system pool** is a thread pool used to run actors. Each node operates multiple pools to coarsely separate resources between different types of activities. A typical set of pools includes:

- **System**: A pool that handles internal operations within YDB node. It serves system tablets, state storage, distributed storage I/O, and so on.
- **User**: A pool dedicated to user-generated load, such as running non-system tablets or queries executed by the KQP.
- **Batch**: A pool for tasks without strict execution deadlines, including heavy queries handled by the KQP background operations like backups, data compaction, and garbage collection.
- **IO**: A pool for tasks involving blocking operations, such as authentication or writing logs to files.
- **IC**: A pool for interconnect, responsible for system calls related to data transfers across the network, data serialization, message splitting and merging.

Tablet implementation

A **tablet** is an actor with a persistent state. It includes a set of data for which this tablet is responsible and a finite state machine through which the tablet's data (or state) changes. The tablet is a fault-tolerant entity because tablet data is stored in a Distributed storage that survives disk and node failures. The tablet is automatically restarted on another node if the previous one is down or overloaded. The data in the tablet changes in a consistent manner because the system infrastructure ensures that there is no more than one tablet leader through which changes to the tablet data are carried out.

The tablet solves the same problem as the Paxos and Raft algorithms in other systems, namely the distributed consensus task. From a technical point of view, the tablet implementation can be described as a Replicated State Machine (RSM) over a shared log, as the tablet state is completely described by an ordered command log stored in a distributed and fault-tolerant storage.

During execution, the tablet state machine is managed by three components:

1. The generic tabular part ensures the log's consistency and recovery in case of failures.
2. **Executor** is an abstraction of a local database, namely data structures and code that arrange work with the data stored by the tablet.
3. An actor with a custom code that implements the specific logic of a specific 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.

Usually, there are orders of magnitude more tablets running in a YDB cluster compared to processes or threads that other systems would use for a similarly sized cluster. There can easily be hundreds of thousands to millions of tablets working simultaneously in a YDB cluster.

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

Tablet leader

A **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 there is no more than one leader for a given tablet at any moment.

Tablet candidate

A **tablet candidate** is one of the election participants who wants to become a leader for a given tablet. If a candidate wins the election, it assumes the tablet leader role.

Tablet follower

A **tablet follower** or **hot standby** is a copy of a tablet leader that applies the log of commands accepted by the leader (with some lag). A tablet can have zero or more followers. Followers serve two primary purposes:

- In case of the leader's termination or failure, followers are the preferred candidates to become the new leader because they can become the leader much faster than other candidates since they have applied most of the log.
- Followers can respond to read-only queries if a client explicitly chooses the optional relaxed transaction mode that allows for stale reads.

Tablet generation

A **tablet generation** is a number identifying the reincarnation of the tablet leader. It changes only when a new leader is chosen and always grows.

Tablet local database

A **tablet local database** or **local database** is a set of data structures and related code that manages the tablet's state and the data it stores. Logically, the local database state is represented by a set of tables very similar to relational tables. Modification of the state of the local database is performed by local tablet transactions generated by the tablet's user actor.

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

Log-structured merge-tree

A **log-structured merge-tree** or **LSM tree**, is a data structure designed to optimize write and read performance in storage systems. It is used in YDB for storing local database tables and VDisks data.

MemTable

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

Sorted string table

A **sorted string table** or **SST** is an immutable data structure that stores table rows sorted by key, facilitating efficient key lookups and range queries. Each SST is composed of a contiguous series of small data pages, typically around 7 KiB in size each, which further optimizes the process of reading data from disk. An SST typically represents a part of LSM tree.

Tablet pipe

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

TabletID

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

Bootstrapper

The **bootstrapper** is the primary mechanism for launching tablets, used for service tablets (for example, for Hive, DS controller, root SchemeShard). The Hive tablet initializes the rest of the tablets.

Shared cache

A **shared cache** is an actor that stores data pages recently accessed and read from distributed storage. Caching these pages reduces disk I/O operations and accelerates data retrieval, enhancing overall system performance.

Memory controller

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

Tablet types

Tablets can be considered a framework for building reliable components operating in a distributed system. YDB has multiple components implemented using this framework, listed below.

Scheme shard

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

Additionally, there is a **root scheme shard**, which stores information about databases created in a cluster.

Data shard

A **data shard** or **DataShard** is a tablet that manages a segment of a row-oriented user table. The logical user table is divided into segments by continuous ranges of the primary key of the table. Each such range is managed by a separate DataShard tablet instance. Such ranges are also called partitions. DataShard tablets store 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

A **KV Tablet** or **key-value tablet** is a tablet that implements a simple key→value mapping, where keys and values are strings. It also has a number of specific features, like 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

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

A **Hive** is a system tablet responsible for launching and managing other tablets. It also moves tablets between nodes in case of node failures or overload. You can learn more about Hive in a dedicated article.

Cluster management system

The **cluster management system** or **CMS** is a system tablet responsible for managing the information about the current YDB cluster state. This information is used to perform cluster rolling restarts without affecting user workloads, maintenance, cluster re-configuration, etc.

Node Broker

The **Node Broker** is a system tablet that registers dynamic nodes in the cluster.

Slot

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

- **Slot** is a portion of a server's resources allocated to running a single YDB node. A common slot size is 10 CPU cores and 50 GB of RAM. Slots are used if a YDB cluster is deployed on servers or virtual machines with sufficient resources to host multiple slots.
- VDisk **slot** or **VSlot** is a fraction of PDisk that can be allocated to one of the VDisks.

State storage

A **State storage** or **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 name service for resolving tablets, i.e., getting ActorId by TabletID. StateStorage is also used in the process of electing the tablet leader.

Information in state storage is volatile. Thus, it is lost when the power is turned off, or the process is restarted. Despite the name, this service is not persistent storage. It contains only information that is easily recoverable and does not have to be durable. However, state storage keeps information on several nodes to minimize the impact of node failures. Through this service, it is possible to gather a quorum, which is used to elect tablet leaders.

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

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

A **gRPC Proxy** is the client proxy system for external user requests. Client requests enter the system via the gRPC protocol, then the proxy component translates them into internal calls for executing these requests, passed around via Interconnect. This proxy provides an interface for both request-response and bidirectional streaming.

Distributed storage implementation

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

Distributed storage persists immutable data, with each immutable blob identified by a specific `LogoBlobID` key. The distributed storage API is very specific, designed only for use by tablets to store their data and log changes, not for general-purpose data storage. Data in distributed storage is deleted using special barrier commands. Due to the lack of mutations in its interface, distributed storage can be implemented without implementing distributed consensus. Moreover, distributed storage is just a building block tablets use to implement distributed consensus.

LogoBlob

A **LogoBlob** is a piece of binary immutable data identified by LogoBlobID and stored in Distributed storage. The blob size is limited at the VDisk level and higher on the stack. Currently, the maximum blob size VDisks are ready to process is 10 MB.

LogoBlobID

A **LogoBlobID** is the LogoBlob identifier in the Distributed storage. It has a structure of the form `[TabletID, Generation, Step, Channel, Cookie, BlobSize, PartID]`. The key elements of LogoBlobID are:

- `TabletID` is an ID of the tablet that the LogoBlob belongs to.
- `Generation` is the generation of the tablet in which the blob was recorded.
- `Channel` is the tablet channel where the LogoBlob is recorded.
- `Step` is an incremental counter, usually within the tablet generation.
- `Cookie` is a unique blob identifier within a single `Step`. A cookie is usually used when writing several blobs within a single `Step`.
- `BlobSize` is the LogoBlob size.
- `PartID` is the identifier of the blob part. It is crucial when the original LogoBlob is broken into parts using erasure coding, and the parts are written to the corresponding VDisks and storage groups.

Replication

Replication is a process that ensures there are always enough copies (replicas) of data to maintain the desired availability characteristics of a YDB cluster. Typically, it is used in geo-distributed YDB clusters.

Erasure Coding

Erasure coding is a method of data encoding in which the original data is supplemented with redundancy and divided into several fragments, providing the ability to restore the original data if one or more fragments are lost. It is widely used in single-AZ YDB clusters as opposed to replication with 3 replicas. For example, the most popular 4+2 scheme provides the same reliability as three replicas, with space redundancy of 1.5 versus 3.

PDisk

PDisk or **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 simulating a block device for testing purposes). PDisk provides data integrity controls (including erasure encoding of sector groups for data recovery on single bad sectors, integrity control with checksums), transparent data-at-rest encryption of all disk data, and transactional guarantees of disk operations (write confirmation strictly after `fsync`).

PDisk contains a scheduler that provides device bandwidth sharing between several clients (VDisks). PDisk divides a block device into chunks called slots (about 128 megabytes in size; smaller chunks are allowed). No more than 1 VDisk can own each slot at a time. PDisk also supports a recovery log shared between PDisk service records and all VDisks.

VDisk

VDisk or **Virtual disk** is a component that implements the persistence of distributed storage LogoBlobs on PDisks. VDisk stores all its data on PDisks. One VDisk corresponds to one PDisk, but usually, several VDisks are linked to one PDisk. Unlike PDisk, which hides chunks and logs behind it, VDisk provides an interface at the LogoBlob and LogoBlobID level, like writing LogoBlob, reading LogoBlobID data, and deleting a set of LogoBlob 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. The VDisks in a group synchronize the data with each other and replicate the data in case of loss. A 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 chunks and logs, reserve chunks, delete chunks, and transactionally receive and return ownership of chunks. In some contexts, Yard can be considered to be 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

The **distributed storage proxy**, **DS proxy**, or **BS proxy** plays the role of a client library for performing operations with Distributed storage. DS Proxy users 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 the quorum of the VDisks, make retries if necessary, and control the write/read flow to avoid overloading VDisks.

Technically, DS Proxy is implemented as an actor service launched by the node warden on each node for each storage group, processing all requests to the group (writing, reading, and deleting LogoBlobs, blocking the group). When writing data, DS proxy performs erasure encoding of data by dividing LogoBlobs into parts, which are then sent to the corresponding VDisks. DS Proxy performs the reverse process when reading, receiving parts from VDisks, and restoring LogoBlobs from them.

Node warden

Node warden or `BS_NODE` is an actor service on each node of the cluster, launching PDisks, VDisks, and DS proxies of static storage groups at the node start. Also, it interacts with the DS controller to launch PDisk, VDisk, and DS proxies of dynamic groups. The DS proxy of dynamic groups is launched on request: node warden processes "undelivered" messages to the DS proxy, launching the corresponding DS proxies and receiving the group configuration from the DS controller.

Fail realm

A **fail realm** is a set of fail domains that are likely to fail simultaneously. The correlated failure of two VDisks within the same fail realm is more probable than that of two VDisks from different fail realms.

An example of a fail realm is a set of hardware located in the same data center or availability zone that can all fail together due to a natural disaster, major power outage, or similar event.

Fail 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.

For example, a fail domain includes disks on the same server, as all server disks may become unavailable if the server's PSU or network controller fails. A fail domain also typically includes servers located in the same server rack, as all the hardware in the rack may become unavailable if there is a power outage or an issue with the network hardware in the same rack. Thus, the typical fail domain corresponds to a server rack if the cluster is configured to be rack-aware, or to a server otherwise.

Domain failures are handled automatically by YDB without shutting down the cluster.

Distributed storage channel

A **channel** is a logical connection between a tablet and Distributed storage group. The tablet can write data to different channels, and each channel is mapped to a specific storage group. Having multiple channels allows the tablet to:

- Record more data than one storage group can contain.
- Store different LogoBlobs on different storage groups, with different properties like erasure encoding or on different storage media (HDD, SSD, NVMe).

Distributed transactions implementation

Terms related to the implementation of distributed transactions are explained below. The implementation itself is described in a separate article DataShard: distributed transactions.

Deterministic transactions

YDB distributed transactions 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 for highly efficient distributed processing of transactions. The original paper imposed limitations on what kinds of operations can be executed in this manner. As these limitations interfered with real-world user scenarios, YDB evolved its algorithms to overcome them by using deterministic transactions as stages of executing user transactions with additional orchestration and locking.

Optimistic locking

As in many other database management systems, YDB queries can put locks on certain pieces of data, like table rows, to ensure that concurrent access does not modify them into an inconsistent state. However, YDB checks these locks not at the beginning of transactions but during commit attempts. The former is called **pessimistic locking** (used in PostgreSQL, for example), while the latter is called **optimistic locking** (used in YDB).

Prepare stage

The **prepare stage** is a phase of distributed transaction execution, during which the transaction body is registered on all participating shards.

Execute stage

The **execute stage** is a phase of distributed query execution in which the scheduled transaction is executed and the 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

The **read-write set** or **RW set** is a set of data that will participate in executing a distributed transaction. It combines the read set, the data that will be read, and the write set, the data modifications to be carried out.

Read set

The **read set** or **ReadSet data** is what participating shards forward during the transaction execution. In the case of data transactions, it may contain information about the state of optimistic locks, the readiness of the shard for commit, or the decision to cancel the transaction.

Transaction proxy

The **transaction proxy** or `TX_PROXY` is a service that orchestrates the execution of many distributed transactions: sequential phases, phase execution, planning, and aggregation of results. In the case of direct orchestration by other actors (for example, KQP data transactions), it is used for caching and allocation of unique TxIDs.

Transaction flags

Transaction flags or **TxFlags** is a bitmask of flags that modify the execution of a transaction in some way.

Transaction ID

Transaction ID or **TxID** is a unique identifier assigned to each transaction when it is accepted by YDB.

Transaction order ID

A **transaction order ID** is a unique identifier assigned to each transaction during planning. It consists of PlanStep and Transaction ID.

PlanStep

PlanStep or **step** is the logical time for which a set of transactions is planned to be executed.

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 the expression of 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 calculations on the read data, and writing the results to the database and/or to a special document representing the query result (shown to the user). The MiniKQL transaction must explicitly set its read set (readable data) and assume a deterministic selection of execution branches (for example, there is no random).

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.

KQP

KQP or **Query Processor** is a YDB component responsible for the orchestration of user query execution and generating the final response.

Global schema

Global Schema, **Global Scheme**, or **Database Schema** is a schema of all the data stored in a database. It consists of tables and other entities, such as topics. The metadata about these entities is called a 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 only work with the global schema.

KiKiMR

KiKiMR is the legacy name of YDB that was used long before it became an open-source product. It can still be occasionally found in the source code, old articles and videos, etc.

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.

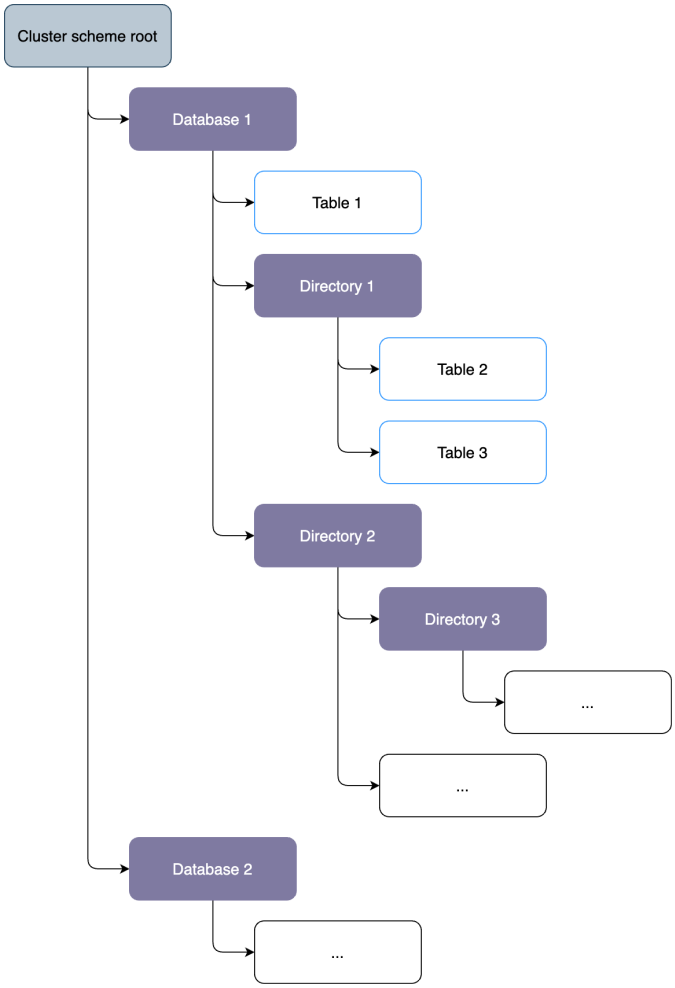
The OS that the client runs on already includes a set of root certificates from the world's major certification authorities. However, the YDB cluster owner can use its own CA that is not associated with any of the global ones, which is often the case in corporate environments, and is almost always used 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).

Cluster structure

This section describes the YDB cluster scheme entities.

YDB cluster scheme

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 a database can use nested directories to form a hierarchy.



YDB scheme objects

Scheme objects in YDB databases:

- Folder
- Table
- View
- Topic
- Secret
- External Table
- External Data Source

Scheme objects in YDB all follow the same naming rules described in the section below.

Database Object Naming Rules

Every scheme object in YDB has a name. In YQL statements, object names are specified by identifiers that can be enclosed in backticks or not. For more information on identifiers, refer to Keywords and identifiers.

Scheme object names in YDB must meet the following requirements:

- Object names can include the following characters:
  - Uppercase Latin characters
  - Lowercase Latin characters
  - Digits
  - Special characters: `.`, `-`, and `_`
- Object name length must not exceed 255 characters.
- Objects cannot be created in folders which names start with a dot, such as `.sys`, `.metadata`, and `.sys_health`.

Column Naming Rules

Column names in YDB must meet the following requirements:

- Column names can include the following characters:
  - Uppercase Latin characters
  - Lowercase Latin characters
  - Digits
  - Special characters: `-` and `_`
- Column names must not start with the system prefix `__ydb_`.

YDB Cluster Topology

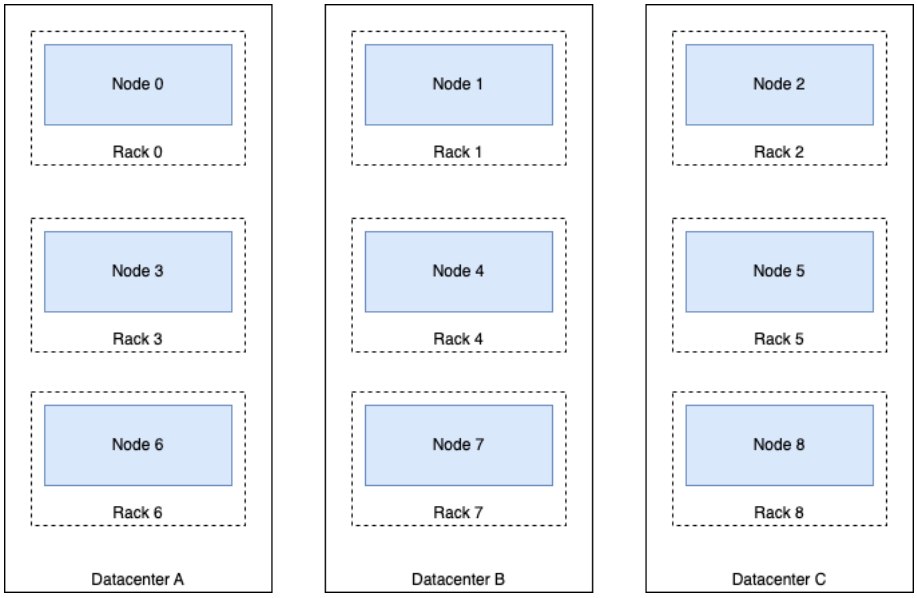
A YDB cluster consists of storage and database nodes. As the data stored in YDB is available only via queries and API calls, both types of nodes are essential for database availability. However, distributed storage consisting of storage nodes has the most impact on the cluster's fault tolerance and ability 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

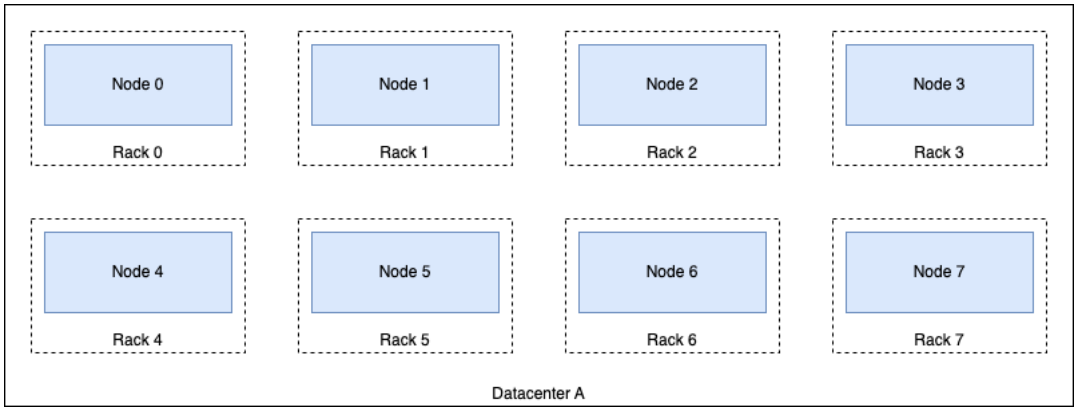
Cluster topology is based on the chosen distributed storage operating mode, which needs to be determined according to the fault tolerance requirements. YDB's failure model is based on the concepts of fail domain and fail realm.

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)	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)	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.

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

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:

- *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.
- *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.
- *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.

The transaction execution mode is specified in its settings when creating the transaction. See the examples for the YDB SDK in the Setting up the transaction execution mode.

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

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 with Topics and Working with topics.

Secondary Indexes

YDB automatically creates a primary key index, which is why selection by primary key is always efficient, affecting only the rows needed. Selections by criteria applied to one or more non-key columns typically result in a full table scan. To make these selections efficient, use *secondary indexes*.

The current version of YDB implements *synchronous* and *asynchronous* global secondary indexes. Each index is a hidden table that is updated:

- For synchronous indexes: Transactionally when the main table changes.
- For asynchronous indexes: In the background while getting 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 may have multiple secondary indexes. An index may include multiple columns, and the sequence of columns in an index matters. A single column may be included in multiple indexes. In addition to the specified columns, every index implicitly stores the table primary key columns to enable navigation from an index record to the table row.

Synchronous Secondary Index

A synchronous index is updated simultaneously with the table that it indexes. This index ensures strict consistency through distributed transactions. While reads and blind writes to a table with no index can be performed without a planning stage, significantly reducing delays, such optimization is impossible when writing data to a table with a synchronous index.

Asynchronous Secondary Index

Unlike a synchronous index, an asynchronous index doesn't use distributed transactions. Instead, it receives changes from an indexed table in the background. Write transactions to a table using this index are performed with no planning overheads due to reduced guarantees: an asynchronous index provides eventual consistency, but no strict consistency. You can only use asynchronous indexes in read transactions in Stale Read Only mode.

Covering Secondary Index

You can copy the contents of columns into a covering index. This eliminates the need to read data from the main table when performing reads by index and significantly reduces delays. At the same time, such denormalization leads to increased usage of disk space and may slow down inserts and updates due to the need for additional data copying.

Unique Secondary Index

This type of index enforces unique constraint behavior and, like other indexes, allows efficient point lookup queries. YDB uses it to perform additional checks, ensuring that each distinct value in the indexed column set appears in the table no more than once. If a modifying query violates the constraint, it will be aborted with a `PRECONDITION_FAILED` status. Therefore, client code must be prepared to handle this status.

A unique secondary index is a synchronous index, so the update process is the same as in the Synchronous Secondary Index section described above from a transaction perspective.

Limitations

Currently, a unique index cannot be added to an existing table.

Vector Index

Vector Index is a special type of secondary index.

Unlike secondary indexes, which optimize equality or range searches, vector indexes allow vector search based on distance or similarity functions.

Creating a Secondary Index Online

YDB lets you create new and delete existing secondary indexes without stopping the service. For a single table, you can only create one index at a time.

Online index creation consists of the following steps:

1. Taking a snapshot of a data table and creating an index table marked that writes are available.

   After this step, write transactions are distributed, writing to the main table and the index, respectively. The index is not yet available to the user.
2. Reading the snapshot of the main table and writing data to the index.

   "Writes to the past" are implemented: situations where data updates in step 1 change the data written in step 2 are resolved.
3. Publishing the results and deleting the snapshot.

   The index is ready to use.

Possible impact on user transactions:

- There may be an increase in delays because transactions are now distributed (when creating a synchronous index).

- There may be an enhanced background of `OVERLOADED` errors because index table automatic shard splitting is actively running during data writes.

The rate of data writes is selected to minimize their impact on user transactions. To quickly complete the operation, we recommend running the online creation of a secondary index when the user load is minimum.

Creating an index is an asynchronous operation. If the client-server connection is interrupted after the operation has started, index building continues. You can manage asynchronous operations using the YDB CLI.

Creating and Deleting Secondary Indexes

A secondary index can be:

- Created when creating a table with the YQL `CREATE TABLE` statement.
- Added to an existing table with the YQL `ALTER TABLE` statement or the YDB CLI `table index add` command.
- Deleted from an existing table with the YQL `ALTER TABLE` statement or the YDB CLI `table index drop` command.
- Deleted together with the table using the YQL `DROP TABLE` statement or the YDB CLI `table drop` command.

Using Secondary Indexes

For detailed information on using secondary indexes in applications, refer to the relevant article in the documentation section for developers.

Vector search

Concept of vector search

Vector search, also known as nearest neighbor search (NN), is an optimization problem where the goal is to find the nearest vector (or a set of vectors) in a given dataset relative to a specified query vector. The proximity between vectors is determined using distance or similarity metrics.

Vector search is actively used in the following areas:

- recommendation systems
- semantic search
- search for similar images
- 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.

Methods for solving vector search tasks can be divided into three major categories:

- exact methods
- approximate methods without index
- approximate methods with index

The choice of a method depends on the number of vectors and the nature of the workload. Exact methods search slowly but update quickly, whereas indexes do 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.

Learn more about exact vector search.

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 construction may take considerable time
- in the current version, data modification operations such as insertion, update, and deletion are not supported

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

- The number of topic partitions is fixed as of changefeed creation and remains unchanged (unlike tables, topics are not elastic).
- Changefeeds support records of the following types of operations:
  - Updates
  - Erases

Adding rows is a special update case, and a record of adding a row in a changefeed will look similar to an update record.

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).

> **Note**
>
> By default, virtual timestamps are not uploaded to the changefeed. To enable them, 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 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>},
    "erase": {},
    "newImage": {<columns>},
    "oldImage": {<columns>},
    "ts": [<step>, <txId>]
}
```

- `key`: An array of primary key component values. Always present.
- `update`: Update flag. Present if a record matches the update operation. In `UPDATES` mode, it also contains the names and values of updated columns.
- `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`).

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]
}
```

Note

- The same record may not contain the `update` 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 or an erase).
- In `UPDATES` mode, the `update` field for update operations is an operation flag (update) and contains the names and values of updated columns.
- JSON object fields containing column names and values (`newImage`, `oldImage`, and `update` 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>
        }
    }
}
```

- `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.

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 number of topic partitions is equal to the number of table partitions. The number of topic partitions can be redefined by specifying the TOPIC_MIN_ACTIVE_PARTITIONS parameter when creating a 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`;
- 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.

Scan Queries in YDB

Scan Queries is a separate data access interface designed primarily for running analytical ad hoc queries against a DB.

This method of executing queries has the following unique features:

- Only *Read-Only* queries.
- In `SERIALIZABLE_RW` mode, a data snapshot is taken and then used for all subsequent operations. As a result, the impact on OLTP transactions is minimal (only taking a snapshot).
- The output of a query is a data stream (gRPC stream). This means scan queries have no limit on the number of rows in the result.
- Due to the high overhead, it is only suitable for ad hoc queries.

> **Note**
>
> From the *Scan Queries* interface, you can query system tables.

Scan queries cannot currently be considered an effective solution for running OLAP queries due to their technical limitations (which will be removed in time):

- The query duration is limited to 5 minutes.
- Many operations (including sorting) are performed entirely in memory, which may lead to resource shortage errors when running complex queries.
- A single strategy is currently in use for joins: *MapJoin* (a.k.a. *Broadcast Join*) where the "right" table is converted to a map; and therefore, must be no more than a few gigabytes in size.
- Prepared form isn't supported, so for each call, a query is compiled.
- There is no optimization for point reads or reading small ranges of data.
- The SDK doesn't support automatic retry.

For handling OLAP workloads in YDB, there is a specialized type of table — column-oriented tables. These tables store the data of each column separately from other columns. This allows only the columns directly involved in the query to be read during execution.

Despite the fact that *Scan Queries* obviously don't interfere with the execution of OLTP transactions, they still use common DB resources: CPU, memory, disk, and network. Therefore, running complex queries **may lead to resource hunger**, which will affect the performance of the entire DB.

How Do I Use It?

Like other types of queries, *Scan Queries* are available via the CLI, and SDK.

C++ SDK

To run a query using *Scan Queries*, use two methods from the `Ydb::TTableClient` class:

```
class TTableClient {
    ...
    TAsyncScanQueryPartIterator StreamExecuteScanQuery(const TString& query,
        const TStreamExecScanQuerySettings& settings = TStreamExecScanQuerySettings());

    TAsyncScanQueryPartIterator StreamExecuteScanQuery(const TString& query, const TParams& params,
        const TStreamExecScanQuerySettings& settings = TStreamExecScanQuerySettings());
    ...
};
```

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
Database	Maximum path	32	Maximum number of	MaxDepth	SCHEME_ERROR
	number of paths (schema objects)		number of path elements (directories, a		GENERIC_ERROR
			(table shards and system tablets) that can run in	MaxShards	GENERIC_ERROR
			automatic table sharding takes place.		
Database	Maximum object name length	255	Limits the number of characters in the name	MaxPathElementLength	SCHEME_ERROR

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

Parameter	Value
Maximum row size	8 MB
Maximum size of an inserted data block	8 MB

Parameter	Default	Explanation	Status in case of a violation of the limit
Query duration	1800 seconds (30 minutes)	The maximum amount of time allowed for a single query to execute.	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.	OVERLOADED
Maximum query text length	10 KB	The maximum allowable length of YQL query text.	BAD_REQUEST
Maximum size of parameter values	50 MB	The maximum total size of parameters passed when executing a previously prepared query.	BAD_REQUEST

owing limitations,

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

D	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
			all access control rules that can be saved for the schema object in question.		
Directory	Maximum	100,000	Maximum created in a directory.	MaxChildrenInDir	SCHEME_ERROR
Table	Maximum number of table shards	35,000	Maximum number of table shards.	MaxShardsInPath	GENERIC_ERROR
Table	Maximum number of columns	200	Limits the total number of columns in a table.	MaxTableColumns	GENERIC_ERROR
Table	Maximum column name length	255	Limits the number of characters in a column name.	MaxTableColumnNameLength	GENERIC_ERROR
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	GENERIC_ERROR
Table	Maximum number of indexes	20	Maximum number of indexes other than the primary key index that can be created in a table.	MaxTableIndices	GENERIC_ERROR
Table	Maximum number of followers	3	Maximum number of read-only replicas that can be specified when creating a table with followers.	MaxFollowersCount	GENERIC_ERROR
Table	Maximum number of tables to copy	10,000	Limit on the size of the table list for persistent table copy operations.	MaxConsistentCopyTargets	GENERIC_ERROR

Parameter	Value
Maximum size of a transmitted message	12 MB

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.

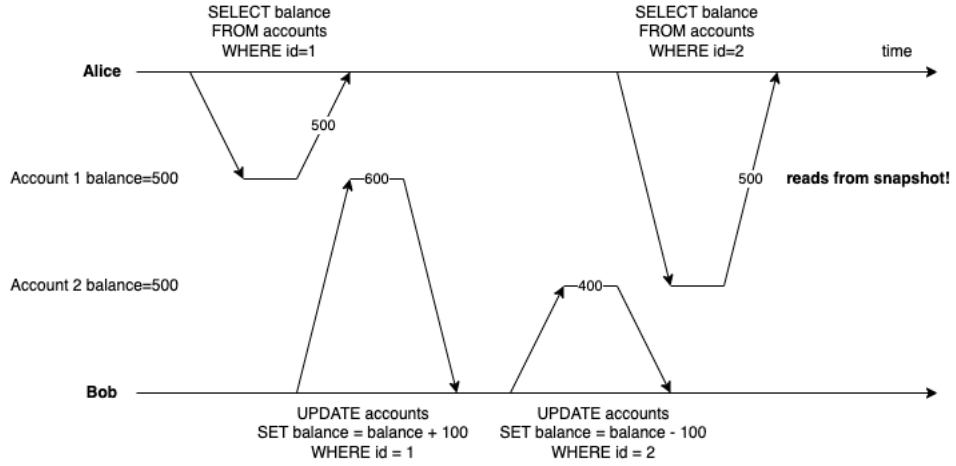
Why YDB Needs MVCC

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.

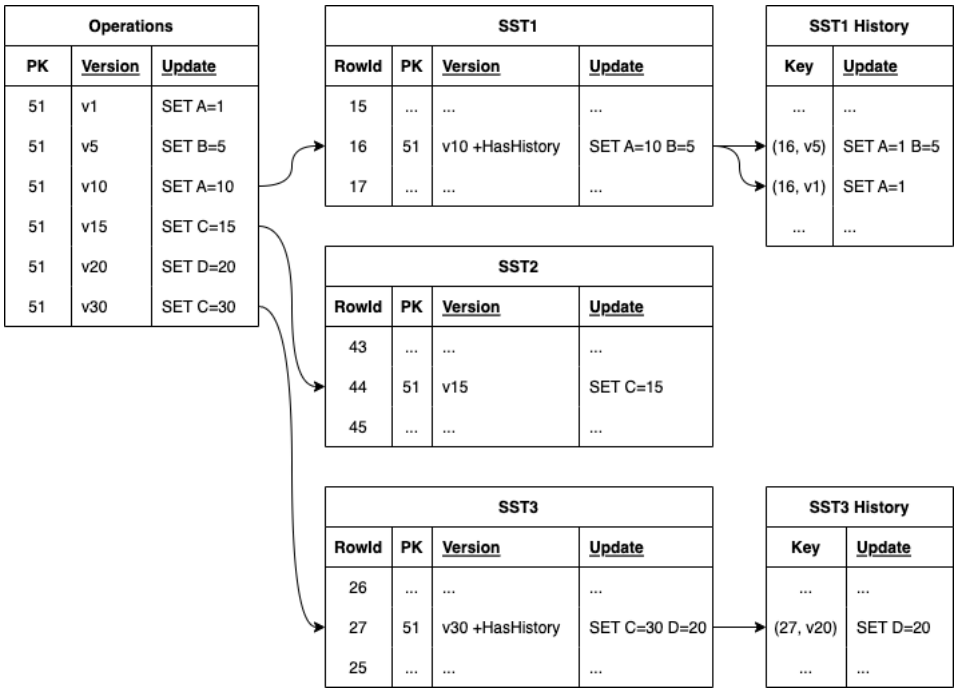
After implementing MVCC using global versions (shared with deterministic distributed transactions), it became possible to perform reads in a transaction using a global snapshot. This means pure read-only transactions no longer fail with serializability errors and rarely need to be retried by the user, 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

DataShard tablets currently store a single table partition in a write-optimized LSM tree, where for each primary key we store row operation with a set of column updates. During searches, we merge updates from multiple levels and get the final row state. Compactions similarly merge updates from multiple levels and write a resulting aggregate row update.

One of our design goals when adding MVCC was minimal degradation to existing workloads, and that meant queries, especially range queries, with the most recent version needed to be fast. That meant using common approaches like adding a version suffix to keys was out of the question. Instead, when a row in an SST (sorted string table, part of an LSM tree) has multiple versions, we only store the most recent version in the main data page, marking it with a flag signaling "history" data is present. Older row versions are stored in a special "history" companion SST, where for each marked row id we store row versions in descending order. When we read from a snapshot, we detect if the most recent row version is too recent and perform a binary search in the history SST instead. Once we find a row version corresponding to a snapshot, we apply its updates to the final row state. We also use the fact that LSM tree levels roughly correspond to their write time, allowing us to stop searching once the first matching row is found for a given snapshot. For each level below that, we only need to apply the most recent row to the final row state, which limits the number of merges to at most the number of levels, which is usually small.



Rows in SSTs are effectively deltas; nonetheless, they are stored as pre-merged from the viewpoint of a given SST, which helps with both search and compaction complexity. Let's imagine a hypothetical situation where the user writes 1 million updates to some key K, each time modifying one of a multitude of columns. As a write-optimized storage, we prefer blind writes and don't read the full row before updating and writing a new updated row; instead, we write an update that says "update column C for key K". If we didn't store pre-merged state at each level, soon there would have been 1 million deltas for the key K, each at a different version. Then each read would potentially need to consider applying all 1 million deltas to the row. Instead, we merge updates at the same level into aggregate updates, starting with memtable (where the previous row state is always in memory and we don't need to read from disk). When compacting several levels into a new SST, we only need to iterate over each update version and merge it with the most recent version in the SSTs below; this limits either merge complexity at compaction (the number of merges for each version is limited by the number of levels) and at read time, while still allowing us to perform blind writes.

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).

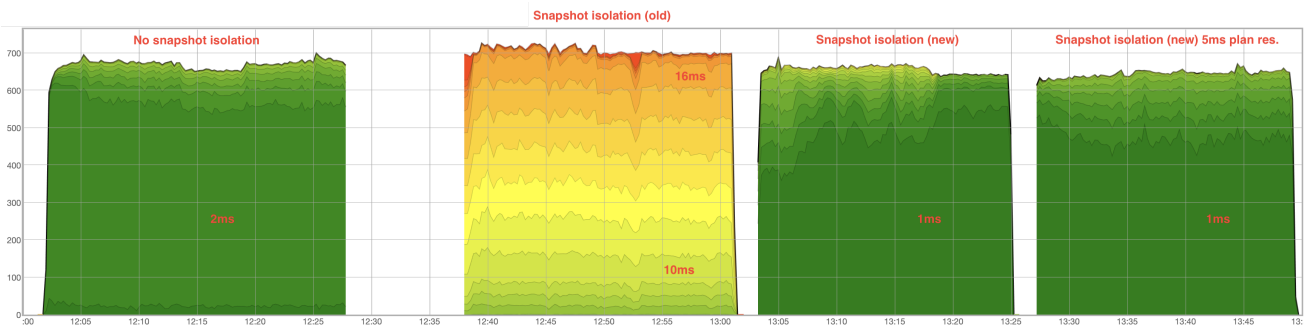
Assigning such non-unique timestamps to immediate transactions guarantees there's always some new non-decreasing timestamp that allows them to execute without delay (subject to other out-of-order execution restrictions). It also helps with compactions, as we may have thousands of these transactions executed 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

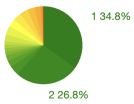
An interesting conundrum happened when we tried to enable MVCC snapshots for the first time: there was a severe degradation in read latency under some workloads! This was because time in YDB usually "ticks" every 10ms (tuned to be similar to typical multi-datacenter commit time of 4ms), and single-shard 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:



	bin	Min	Max	Last	Avg	Sum
☑	all (35)					
☑	1	0.000	562.600	0.000	225.416	87010.733
☑	2	0.000	594.467	0.000	173.325	66903.600
☑	3	0.000	60.333	0.000	25.600	9881.733



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.

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.

To drop an asynchronous replication instance, use the DROP ASYNC REPLICATION YQL expression.

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:

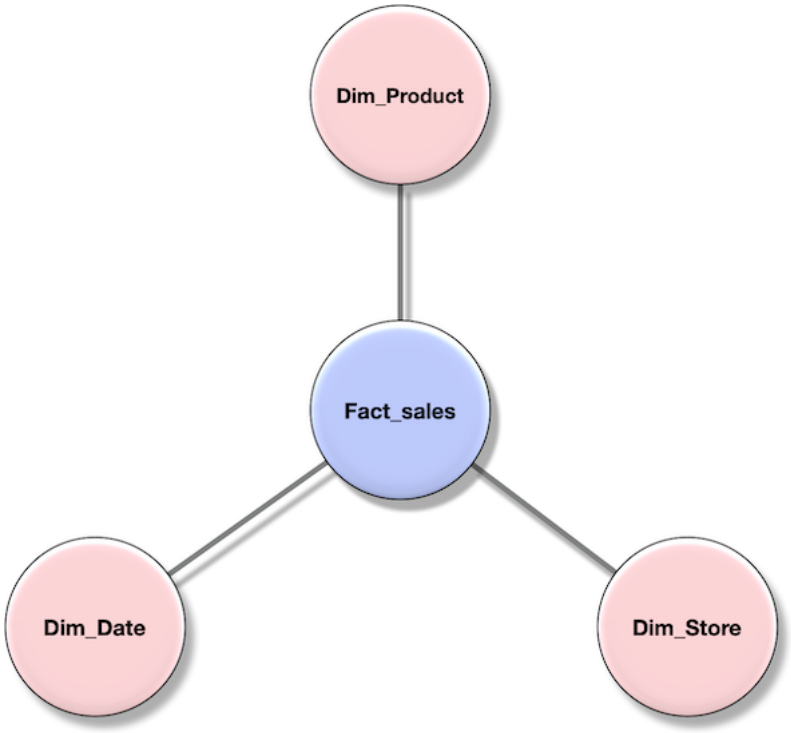
```
SELECT
    P.Brand,
    S.Country AS Countries,
    SUM(F.Units_Sold)

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.

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`, `body` and so on.

Rows in the table hold the data, while columns define the 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,
    body String,
    PRIMARY KEY (id)
)
```

Please note that currently, the `NOT NULL` constraint can only be applied to columns that are part of the primary key.

YDB supports the creation of both row-oriented and column-oriented tables. The primary difference between them lies in their use cases and how data is stored on the disk drive. In row-oriented tables, data is stored sequentially in the form of rows, while in column-oriented tables, data is stored in the form of columns. Each table type has its own specific purpose.

Row-Oriented Tables

Row-oriented tables are well-suited for transactional queries generated by Online Transaction Processing (OLTP) systems, such as weather service backends or online stores. Row-oriented tables offer efficient access to a large number of columns simultaneously. Lookups in row-oriented tables are optimized due to the utilization of indexes.

An index is a data structure that improves the speed of data retrieval operations based on one or several columns. It's analogous to an index in a book: instead of scanning every page of the book to find a specific chapter, you can refer to the index at the back of the book and quickly navigate to the desired page.

Searching using an index allows you to swiftly locate the required rows without scanning through all the data. For instance, if you have an index on the "author" column and you're looking for articles written by "Gray," the DBMS leverages this index to quickly identify all rows associated with that surname.

You can create a row-oriented table through the YDB web interface, CLI, or SDK. Regardless of the method you choose to interact with YDB, it's important to keep in mind the following rule: the table must have at least one primary key column, and it's permissible to create a table consisting solely of primary key columns.

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-oriented database table can be partitioned by primary key value ranges. Each partition of the table is responsible for a specific section of primary keys. Key ranges served by different partitions do not overlap. Different table partitions can be served by different cluster nodes (including ones in different locations). Partitions can also move independently between servers to enable rebalancing or ensure partition operability if servers or network equipment goes offline.

If there is not a lot of data or load, the table may consist of a single shard. As the amount of data served by the shard or the load on the shard grows, YDB automatically splits this shard into two shards. The data is split by the median value of the primary key if the shard size exceeds the threshold. If partitioning by load is used, the shard first collects a sample of the requested keys (that can be read, written, and deleted) and, based on this sample, selects a key for partitioning to evenly distribute the load across new shards. So in the case of load-based partitioning, the size of new shards may significantly vary.

The size-based shard split threshold and automatic splitting can be configured (enabled/disabled) individually for each database table.

In addition to automatically splitting shards, you can create an empty table with a predefined number of shards. You can manually set the exact shard key split range or evenly split it into a predefined number of shards. In this case, ranges are created based on the first component of the primary key. You can set even splitting for tables that have a `Uint64` or `Uint32` integer as the first component of the primary key.

Partitioning parameters refer to the table itself rather than to secondary indexes built on its data. Each index is served by its own set of shards, and decisions to split or merge its partitions are made independently based on the default settings. These settings may become available to users in the future like the settings of the main table.

A split or a merge usually takes about 500 milliseconds. During this time, the data involved in the operation becomes temporarily unavailable for reads and writes. Without raising it to the application level, special wrapper methods in the YDB SDK make automatic retries when they discover that a shard is being split or merged. Please note that if the system is overloaded for some reason (for example, due to a general shortage of CPU or insufficient DB disk throughput), split and merge operations may take longer.

The following table partitioning parameters are defined in the data schema:

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`.

Automatic partitioning by load. If a shard consumes more than 50% of the CPU for a few dozens of seconds, it is enqueued for splitting. If the total load on two or more adjacent shards uses less than 35% of a single CPU core within an hour, they are enqueued for merging.

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 one table partition can only be hosted on one server and use no more than 1 CPU core for data update operations. Hence, you can set the minimum number of partitions for a table on which a high load is expected to at least the number of nodes (servers) or, preferably, to 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. This setting takes effect when the `AUTO_PARTITIONING_BY_SIZE` mode is enabled.

This value serves as a recommendation for partitioning. Partitioning may sometimes not occur even if the configured size is exceeded.

AUTO_PARTITIONING_MIN_PARTITIONS_COUNT

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

Partitions are only merged if their actual number exceeds the value specified by this parameter. When using automatic partitioning by load, we recommend that you set this parameter to a value other than 1, so that periodic load drops don't lead to a decrease in the number of partitions below the required one.

AUTO_PARTITIONING_MAX_PARTITIONS_COUNT

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

Partitions are only split if their number doesn't exceed the value specified by this parameter. With any automatic partitioning mode enabled, we recommend that you set a meaningful value for this parameter and monitor when the actual number of partitions approaches this value; otherwise, splitting of partitions will stop sooner or later under an increase in data or load, which will lead 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 making queries in YDB, the actual execution of a query to each shard is performed at a single point serving the distributed transaction protocol. By storing data in shared storage, you can run one or more shard followers without allocating additional storage space: the data is already stored in replicated format, and you can serve more than one reader (but there is still only one writer at any given moment).

Reading data from followers allows you:

- To serve clients demanding minimal delay, which is otherwise unachievable in a multi-DC cluster. This is accomplished by executing queries soon after they are formulated, which eliminates the delay associated with inter-DC transfers. As a result, you can both preserve all the storage reliability guarantees of a multi-DC cluster and respond to point read queries in milliseconds.
- To handle read queries from followers without affecting modifying queries running on a shard. This can be useful both for isolating different scenarios and for increasing the partition bandwidth.
- To ensure continued service when moving a partition leader (both in a planned manner for load balancing and in an emergency). It lets the processes in the cluster survive without affecting the reading clients.
- To increase the overall shard read performance if many read queries access the same keys.

You can enable running read replicas for each shard of the table in the table data schema. The read replicas (followers) are typically accessed without leaving the data center network, which ensures response delays in milliseconds.

Parameter name	Description	Type	Acceptable values	Update capability	Reset capability
`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	Yes	No

The internal state of each of the followers is restored exactly and fully consistently from the leader state.

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 followers, their delay from the leader may vary: although each follower of each of the shards retains internal consistency, artifacts may be observed from shard to shard. Please provide for this in your application code. For that same reason, it's currently impossible to perform cross-shard transactions from followers.

Deleting Expired Data (TTL)

YDB supports automatic background deletion of expired data. A table data schema may define a column containing a `Datetime` or a `Timestamp` value. A comparison of this value with the current time for all rows will be performed in the background. Rows for which the current time becomes greater than the column value plus specified delay will be deleted.

Parameter name	Type	Acceptable values	Update capability	Reset capability
`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 lets you rename an existing table, move it to another directory of the same database, or replace one table with another, deleting the data in the replaced table. Only the metadata of the table is changed by operations (for example, its path and name). The table data is neither moved nor 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 renaming is determined by the type of data transactions currently running against the table and doesn't depend on the table size.

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

Bloom Filter

Using a Bloom filter lets you more efficiently determine if some keys are missing in a table when making multiple point queries by primary key. This reduces the number of required disk I/O operations but increases the amount of memory consumed.

Parameter name	Type	Acceptable values	Update capability	Reset capability

`KEY_BLOOM_FILTER`	Enum	`ENABLED`, `DISABLED`	Yes	No

Column Groups

YDB allows grouping columns in a table to optimize their storage and usage. The column group mechanism improves performance for partial row reads by separating table columns into multiple storage groups. The most commonly used scenario is the organization of storing rarely used attributes in a separate column group. Then you can enable data compression and/or store it on slower drives.

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 may contain any number of columns from its table. Each table column belongs to one and only one column group (column groups don't 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.

The following storage attributes are configured for column groups:

- Storage device type (SSD or HDD, availability depends on YDB cluster configuration);
- Data compression mode (no compression or LZ4 algorithm compression).

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.

Accessing data in primary column group fields is faster and less resource-intensive than accessing the same table row's data stored in additional column groups. Primary key lookups always occur in the primary column group. Accessing fields in other column groups requires additional search operations to locate specific storage positions after the primary key lookup.

Thus, moving some columns into a separate group accelerates reads for critical, frequently used columns (in the primary group) while slightly slowing access to other columns. Additionally, column groups enable storage parameter management - selecting device types and compression modes.

Column-Oriented Tables

> **Warning**
>
> Column-oriented YDB tables are in the Preview mode.

YDB's column-oriented tables store data of each column separately (independently) from each other. This data storage principle is optimized for handling Online Analytical Processing (OLAP) workloads, as only the columns directly involved in the query are read during its execution. One of the key advantages of this approach is the high data compression ratios since columns often contain repetitive or similar data. A downside, however, is that operations on whole rows become more resource-intensive.

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 YDB column-oriented tables is similar to working with row 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, to evenly distribute the data across the hosts.
- Column-oriented 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: `Bool`, `Decimal`, `Double`, `Float`, `Int8`, `Int16`, `Interval`, `JsonDocument`, `Json`, `Uuid`, `Yson`.
- Column-oriented tables support column groups, but only for compression settings.

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,
    body String,
    PRIMARY KEY (id)
)
WITH (STORE = COLUMN);
```

At the moment, not all functionality of column-oriented tables is implemented. The following features are not currently supported:

- Reading from replicas.
- Secondary indexes.
- Vector indexes.
- Bloom filters.
- Change Data Capture.
- Table renaming.
- Custom table attributes.
- Modifying the list of columns.
- Adding data to column-oriented tables using the SQL INSERT statement.
- Deleting data from column-oriented tables using the SQL DELETE statement. In fact, deletion is only possible after the TTL data retention time has expired.

Partitioning Column-Oriented Tables

Unlike row-oriented YDB tables, you cannot partition column-oriented tables by primary keys but only by specially designated partitioning keys. Partitioning keys constitute a subset of the table's primary keys.

Example of column-oriented partitioning:

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

Unlike data partitioning in row-oriented YDB tables, key values are not used to partition data in column-oriented tables. This way, you can uniformly distribute data across all your existing partitions. This kind of partitioning enables you to avoid hotspots at data insertion and speeds up analytical queries that process (that is, read) large amounts of data.

How you select partitioning keys substantially affects the performance of queries to your column-oriented tables. Learn more in Choosing keys for maximum column-oriented table performance.

To manage data partitioning, use the `AUTO_PARTITIONING_MIN_PARTITIONS_COUNT` additional parameter. The system ignores other partitioning parameters for column-oriented tables.

`AUTO_PARTITIONING_MIN_PARTITIONS_COUNT` sets the minimum physical number of partitions used to store data.

- Type: `Uint64`.
- The default value is `1`.

Because it ignores all the other partitioning parameters, the system uses the same value as the upper partition limit.

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 and delivering them to multiple subscribers. Basically, a topic is a named set of messages.

A producer app writes messages to a topic. Consumer apps are independent of each other, they receive and read messages from the topic in the order they were written there. Topics implement the publish-subscribe architectural pattern.

YDB topics have the following properties:

- At-least-once delivery guarantees when messages are read by subscribers.
- Exactly-once delivery guarantees when publishing messages (to ensure there are no duplicate messages).
- FIFO message processing guarantees for messages published with the same source ID.
- Message delivery bandwidth scaling for messages published with different sequence IDs.

Messages

Data is transferred as message streams. A message is the minimum atomic unit of user information. A message consists of a body, attributes, and additional system properties. The content of a message is an array of bytes which is not interpreted by YDB in any way.

Messages may contain user-defined attributes in "key-value" format. They are returned along with the message body when reading the message. User-defined attributes let the consumer decide whether it should 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 reading them.

Partitioning

To enable horizontal scaling, a topic is divided into `partitions` that are units of parallelism. Each partition has a limited throughput. The recommended write speed is 1 MBps.

> **Note**
>
> As of now, you can only reduce the number of partitions in a topic by deleting and recreating a topic with a smaller number of partitions.

Partitions can be:

- **Active.** By default, all partitions are active. Both read and write operations are allowed on an active partition.
- **Inactive.** An inactive partition is read-only. A partition becomes inactive after splitting for autopartitioning. It is automatically deleted once all messages are removed due to the expiration of the retention period.

Offset

All messages within a partition have a unique sequence number called an `offset`. An offset monotonically increases as new messages are written.

Autopartitioning

Total topic throughput is determined by the number of partitions in the topic and the throughput of each partition. The number of partitions and the throughput of each partition are set at the time of topic creation. If the maximum required write speed for a topic is unknown at the creation time, autopartitioning allows the topic to be scaled automatically. If autopartitioning is enabled for a topic, the number of partitions will increase automatically as the write speed increases (see Autopartitioning Strategies).

Guarantees

1. The SDK and server provide an exactly-once guarantee in the case of writing during a partition split. This means that any message will be written either to the parent partition or to one of the child partitions but never to both simultaneously. Additionally, a message cannot be written to the same partition multiple times.
2. The SDK and server maintain the reading order. Data is read from the parent partition first, followed by the child partitions.
3. As a result, the exactly-once writing guarantee and the reading order guarantee are preserved for a specific producer identifier.

Autopartitioning Strategies

The following autopartitioning strategies are available for any topic:

DISABLED

Autopartitioning is disabled for this topic. The number of partitions remains constant, and there is no automatic scaling.

The initial number of partitions is set during topic creation. If the partition count is manually adjusted, new partitions are added. Both previously existing and new partitions are active.

UP

Upwards autopartitioning is enabled for this topic. This means that if the write speed to the topic increases, the number of partitions will automatically increase. However, if the write speed decreases, the number of partitions remains unchanged.

The partition count increase algorithm works as follows: if the write speed for a partition exceeds a defined threshold (as a percentage of the maximum write speed for that partition) during a specified period, the partition is split into two child partitions. The original partition becomes inactive, allowing only read operations. When the retention period expires, and all messages in the original partition are deleted, the partition itself is also deleted. The two new child partitions become active, allowing both read and write operations.

PAUSED

Autopartitioning is paused for this topic, meaning that the number of partitions does not increase automatically. If needed, you can re-enable autopartitioning for this topic.

Examples of YQL queries for switching between different autopartitioning strategies can be found here.

Autopartitioning Constraints

The following constraints apply when using autopartitioning:

1. Once autopartitioning is enabled for a topic, it cannot be stopped, only paused.
2. When autopartitioning is enabled for a topic, it is impossible to read from or write to it using the Kafka API.
3. Autopartitioning can only be enabled on topics that use the reserved capacity mode.

Message Sources and Groups

Messages are ordered using the `producer_id` and `message_group_id`. The order of written messages is maintained within pairs: `<producer ID, message group ID>`.

When used for the first time, a pair of `<producer ID, message group ID>` is linked to a topic's partition using the round-robin algorithm and all messages with this pair of IDs get into the same partition. The link is removed if there are no new messages using this producer ID for 14 days.

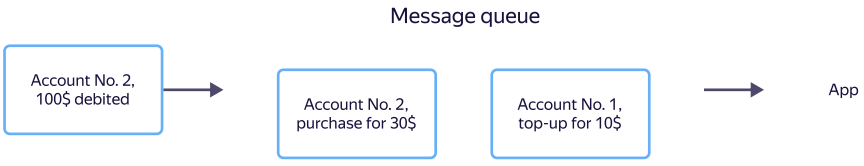
> **Warning**
>
> The recommended maximum number of `<producer ID, message group ID>` pairs is up to 100 thousand per partition in the last 14 days.

Why and When the Message Processing Order Is Important

When the Message Processing Order Is Important

Let's consider a finance application that calculates the balance on a user's account and permits or prohibits debiting the funds.

For such tasks, you can use a message queue. When you top up your account, debit funds, or make a purchase, a message with the account ID, amount, and transaction type is registered in the queue. The application processes incoming messages and calculates the balance.

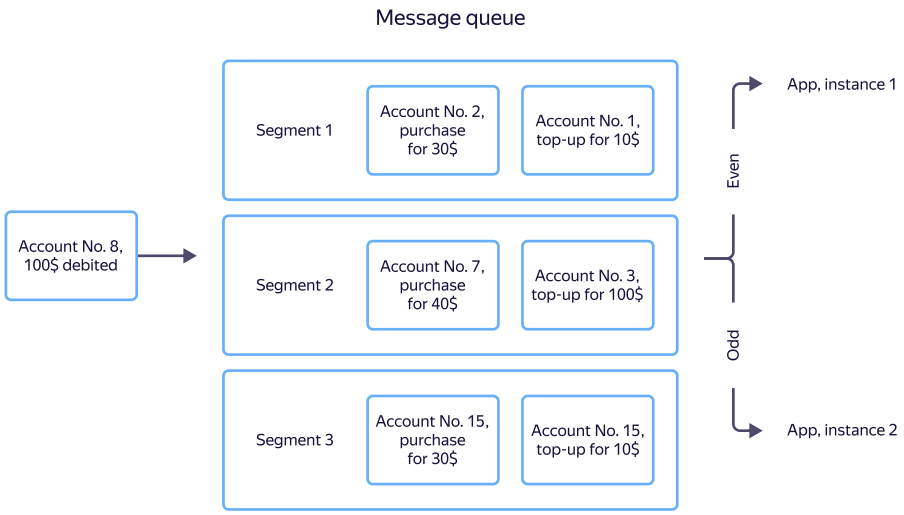


To accurately calculate the balance, the message processing order is crucial. If a user first tops up their account and then makes a purchase, messages with details about these transactions must be processed by the app in the same order. Otherwise, there may be an error in the business logic and the app will reject the purchase as a result of insufficient funds. There are guaranteed delivery order mechanisms, but they cannot ensure a message order within a single queue on an arbitrary data amount.

When several application instances read messages from a stream, a message about account top-ups can be received by one instance and a message about debiting by another. In this case, there's no guaranteed instance with accurate balance information. To avoid this issue, you can, for example, save data in the DBMS, share information between application instances, and implement a distributed cache.

YDB can write data so that messages from one source (for example, about transactions from one account) arrive at the same application instance. The source of a message is identified by the source_id, while the sequence number of a message from the source is used to ensure there are no duplicate messages. YDB arranges data streams so that messages from the same source arrive at the same partition. As a result, transaction messages for a given account will always arrive at the same partition and be processed by the application instance linked to this partition. Each of the instances processes its own subset of partitions and there's no need to synchronize the instances.

Below is an example when all transactions on accounts with even IDs are transferred to the first instance of the application, and with odd ones — to the second.



When the Processing Order Is Not Important

For some tasks, the message processing order is not critical. For example, it's sometimes important to simply deliver data that will then be ordered by the storage system.

For such tasks, the 'no-deduplication' mode can be used. In this scenario, neither `producer_id` nor `source_id` are specified in write session setup and `sequence numbers` are also not used for messages. The no-deduplication mode offers better performance and requires fewer server resources, but there is no message ordering or deduplication on the server side, which means that a message sent to the server multiple times (for example, due to network instability or writer process crash) may also be written to the topic multiple times.

> **Warning**
>
> We strongly recommend that you don't use random or pseudo-random source IDs. We recommend using a maximum of 100 thousand different source IDs per partition.

Source ID

A source ID is an arbitrary string up to 2048 characters long. This is usually the ID of a file server or some other ID.

Sample Source IDs

Type	ID	Description
File	Server ID	Files are used to store application logs. In this case, it's convenient to use the server ID as a source ID.
User actions	ID of the class of user actions, such as "viewing a page", "making a purchase", and so on.	It's important to handle user actions in the order they were performed by the user. At the same time, there is no need to handle every single user action in one application. In this case, it's convenient to group user actions by class.

Message Group ID

A message group ID is an arbitrary string up to 2048 characters long. This is usually a file name or user ID.

Sample Message Group IDs

Type	ID	Description
File	Full file path	All data from the server and the file it hosts will be sent to the same partition.

User actions	User ID	It's important to handle user actions in the order they were performed. In this case, it's convenient to use the user ID as a source ID.

Message Sequence Numbers

All messages from the same source have a `sequence number` used for their deduplication. A message sequence number should monotonically increase within a `topic`, `source` pair. If the server receives a message whose sequence number is less than or equal to the maximum number written for the `topic`, `source` pair, the message will be skipped as a duplicate. Some sequence numbers in the sequence may be skipped. Message sequence numbers must be unique within the `topic`, `source` pair.

Sequence numbers are not used if no-deduplication mode is enabled.

Sample Message Sequence Numbers

Type	Example	Description
File	Offset of transferred data from the beginning of a file	You can't delete lines from the beginning of a file, since this will lead to skipping some data as duplicates or losing some data.
DB table	Auto-increment record ID	

Message Retention Period

The message retention period is set for each topic. After it expires, messages are automatically deleted. An exception is data that hasn't been read by an important consumer: this data will be stored until it's read.

Data Compression

When transferring data, the producer app indicates that a message can be compressed using one of the supported codecs. The codec name is passed while writing a message, saved along with it, and returned when reading the message. Compression applies to each individual message, no batch message compression is supported. Data is compressed and decompressed on the producer and consumer apps' end.

Supported codecs are explicitly listed in each topic. When making an attempt to write data to a topic with a codec that is not supported, a write error occurs.

Codec	Description
`raw`	No compression.
`gzip`	Gzip compression.
`lzop`	lzop compression.
`zstd`	zstd compression.

Consumer

A consumer is a named entity that reads data from a topic. A consumer contains committed consumer offsets for each topic read on their behalf.

Consumer Offset

A consumer offset is a saved offset of a consumer by each topic partition. It's saved by a consumer after sending commits of the data read. When a new read session is established, messages are delivered to the consumer starting with the saved consumer offset. This lets users avoid saving the consumer offset on their end.

Important Consumer

A consumer may be flagged as "important". This flag indicates that messages in a topic won't be removed until the consumer reads and confirms them. You can set this flag for most critical consumers that need to handle all data even if there's a long idle time.

Warning

As a long timeout of an important consumer may result in full use of all available free space by unread messages, be sure to monitor important consumers' data read lags.

Topic Protocols

To work with topics, the YDB SDK is used (see also Reference).

Kafka API version 3.4.0 is also supported with some restrictions (see Work with Kafka API).

Transactions with Topics

YDB supports working with topics within transactions.

Read from a Topic Within a Transaction

Topic data does not change during a read operation. Therefore, within transactional reads from a topic, only the offset commit is a true transactional operation. The postponed offset commit occurs automatically at the transaction commit, and the SDK handles this transparently for the user.

Write into a Topic Within a Transaction

During transactional writes to a topic, data is stored outside the partition until the transaction is committed. At the transaction commit, the data is published to the partition and appended to the end of the partition with sequential offsets. Changes made within the transaction are not visible in transactions with topics in YDB.

Topic Transaction Constraints

There are no additional constraints when working with topics within a transaction. It is possible to write large amounts of data to a topic, write to multiple partitions, and read with multiple consumers.

However, it is recommended to consider that data is published only at transaction commit. Therefore, if a transaction is long-running, the data will become visible only after a significant delay.

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

> **Warning**
>
> This functionality is in "Preview" mode.

To work with external data sources in YDB, federated queries are used. Federated queries utilize various access credentials 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.

> **Warning**
>
> The current syntax for working with secrets is temporary and will be changed in future releases of YDB.

Creating Secrets

Secrets are created using an SQL query:

```
CREATE OBJECT `MySecretName` (TYPE SECRET) WITH value=`MySecretData`;
```

Access Management

All rights to use the secret belong to its creator. The creator can grant another user read access to the secret through access management for secrets.

Special objects called `SECRET_ACCESS` are used to manage access to secrets. To grant permission to use the secret `MySecretName` to the user `another_user`, a `SECRET_ACCESS` object named `MySecretName:another_user` must be created:

```
CREATE OBJECT `MySecretName:another_user` (TYPE SECRET_ACCESS)
```

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 the 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. In the case of S3 (Object Storage), it describes the access credentials and the path to the bucket.

The following example demonstrates creating 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_NAME="test_password_name",
  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;
```

External data sources allow execution of federated queries for cross-system data analytics tasks.

The following data sources can be used:

- ClickHouse
- PostgreSQL
- Connections to S3 (Object Storage)

Federated Queries

Federated queries allow retrieving information from various data sources without needing to transfer the data from these sources into YDB storage. Currently, federated queries support interaction with ClickHouse, PostgreSQL, and S3-compatible data stores. Using YQL queries, you can access these databases without the need to duplicate data between systems.

To work with data stored in external DBMSs, it is sufficient to create an external data source. To work with unstructured data stored in S3 buckets, you additionally need to create an external table. In both cases, it is necessary to create secrets objects first that store confidential data required for authentication in external systems.

You can learn about the internals of the federated query processing system in the architecture section. Detailed information on working with various data sources is provided in the corresponding sections:

- ClickHouse
- Greenplum
- MySQL
- PostgreSQL
- S3
- YDB

Federated Query Processing System Architecture

External Data Sources and External Tables

A key element of the federated query processing system in YDB is the concept of an external data source. Such sources can include relational DBMS, object storage, 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 located in external systems, YDB must have information about the internal structure of this data (e.g., the number, names, and types of columns in tables). Some sources provide such metadata along with the data itself, whereas for other unschematized sources, this metadata must be provided externally. This latter purpose is served by external tables.

Once external data sources and (if necessary) external tables are registered in YDB, the client can proceed to describe federated queries.

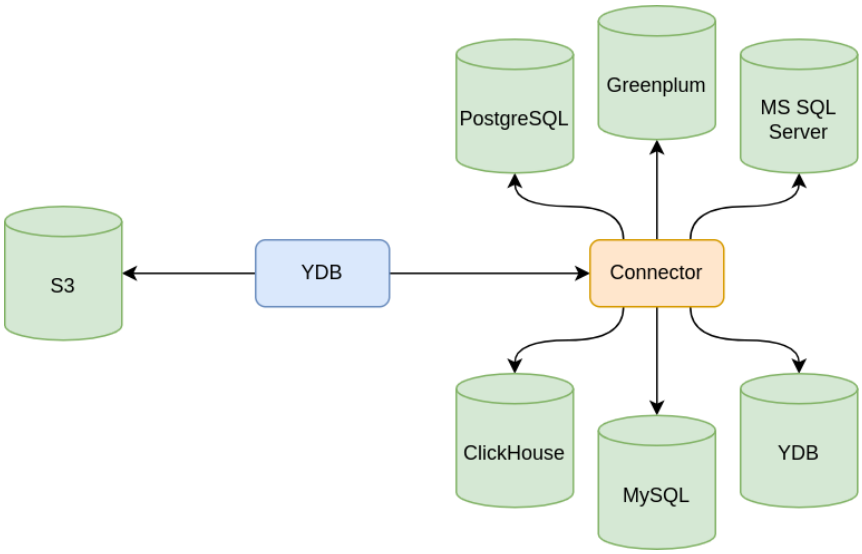
Connectors

While executing federated queries, YDB needs to access external data storage systems over the network, for which it uses their client libraries. Including such dependencies negatively affects the codebase size, compilation time, and binary file size of YDB, as well as the product's overall stability.

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 require support for all sources simultaneously. Support can be optionally enabled using *connectors* - special microservices implementing a unified interface for accessing external data sources.

The functions of connectors include:

- Translating YQL queries into queries in the language specific to the external source (e.g., into another SQL dialect or HTTP API calls).
- Establishing network connections with data sources.
- Converting data retrieved from external sources into a columnar format in Arrow IPC Stream format, supported by YDB.



Thus, connectors form an abstraction layer that hides the specifics of external data sources from YDB. The concise connector interface makes it easy to expand the list of supported sources with minimal changes to YDB's 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

Source	Support
ClickHouse	Via connector fq-connector-go
Greenplum	Via connector fq-connector-go
Microsoft SQL Server	Via connector fq-connector-go
MySQL	Via connector fq-connector-go
PostgreSQL	Via connector fq-connector-go
S3	Built into `ydbd`
YDB	Via connector fq-connector-go

Working with ClickHouse Databases

This section describes the basic information about working with the external ClickHouse database ClickHouse.

To work with the external ClickHouse database, the following steps must be completed:

1. Create a secret containing the password to connect to the database.

```
CREATE OBJECT clickhouse_datasource_user_password (TYPE SECRET) WITH (value = "<password>");
```

2. Create an external data source describing 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"`). To enable encryption for connections to the external database, use 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_NAME="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

To work with ClickHouse, use the following SQL query form:

```
SELECT * FROM clickhouse_datasource.<table_name>
```

Where:

- `clickhouse_datasource` is the identifier of the external data source;
- `<table_name>` is the table's name within the external data source.

Limitations

There are several limitations when working with ClickHouse clusters:

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 within it. This optimization, known as "predicate pushdown", significantly reduces the volume of data transferred from the source to the federated query processing engine. This reduces network load and saves computational resources for YDB.

   A specific case of predicate pushdown, where filtering expressions specified after the `WHERE` keyword are passed down, is called "filter pushdown". Filter pushdown is possible when using:

Description	Example
Filters like `IS NULL` / `IS NOT NULL`	`WHERE column1 IS NULL` or `WHERE column1 IS NOT NULL`
Logical conditions `OR` , `NOT` , `AND`	`WHERE column IS NULL OR column2 IS NOT NULL`
Comparison conditions `=` , `<>` , `<` , `<=` , `>` , `>=` with other columns or constants	`WHERE column3 > column4 OR column5 <= 10`

   Supported data types for filter pushdown:

YDB Data Type
`Bool`
`Int8`
`Uint8`
`Int16`
`Uint16`
`Int32`
`Uint32`
`Int64`
`Uint64`
`Float`
`Double`

Supported Data Types

By default, ClickHouse columns cannot physically contain `NULL` values. However, users can create tables with columns of optional or nullable types. The column types displayed in YDB when extracting data from the external ClickHouse database will depend on whether primitive or optional types are used in the ClickHouse table. Due to the previously discussed limitations of YDB types used to store dates and times, all similar ClickHouse types are displayed in YDB as optional.

Below are the mapping tables for ClickHouse and YDB types. All other data types, except those listed, are not supported.

Primitive Data Types

ClickHouse data type	YDB data type	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. Values outside this range return `NULL`.
`DateTime`	`Optional<DateTime>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. Values outside this range return `NULL`.
`DateTime64`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. Values outside this range return `NULL`.

`String`	`String`	
`FixedString`	`String`	Null bytes in `FixedString` are transferred to `String` unchanged.

Optional Data Types

ClickHouse data type	YDB data type	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. Values outside this range return `NULL`.
`Nullable(DateTime)`	`Optional<DateTime>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. Values outside this range return `NULL`.
`Nullable(DateTime64)`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. Values outside this range return `NULL`.
`Nullable(String)`	`Optional<String>`	
`Nullable(FixedString)`	`Optional<String>`	Null bytes in `FixedString` are transferred to `String` unchanged.

Working with Greenplum Databases

This section provides basic information on working with external Greenplum databases. Since Greenplum is based on PostgreSQL, integrations with them are similar, and some links below may lead to PostgreSQL documentation.

Follow these steps to work with an external Greenplum database:

1. Create a secret containing the password for connecting to the database.

   ```
   CREATE OBJECT greenplum_datasource_user_password (TYPE SECRET) WITH (value = "<password>");
   ```

2. Create an external data source that describes a specific database within the Greenplum cluster. In the `LOCATION` parameter, pass the network address of the master node of Greenplum. By default, the namespace `public` is used for reading, but this value can be changed using the optional `SCHEMA` parameter. The network connection is made using the standard Frontend/Backend Protocol over TCP transport (`PROTOCOL="NATIVE"`). 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_NAME="greenplum_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 format is used to work with Greenplum:

```
SELECT * FROM greenplum_datasource.<table_name>
```

where:

- `greenplum_datasource` - identifier of the external data source;
- `<table_name>` - table name within 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 within it. This optimization, known as "predicate pushdown", significantly reduces the volume of data transferred from the source to the federated query processing engine. This reduces network load and saves computational resources for YDB.

   A specific case of predicate pushdown, where filtering expressions specified after the `WHERE` keyword are passed down, is called "filter pushdown". Filter pushdown is possible when using:

Description	Example
Filters like `IS NULL` / `IS NOT NULL`	`WHERE column1 IS NULL` or `WHERE column1 IS NOT NULL`
Logical conditions `OR`, `NOT`, `AND`	`WHERE column IS NULL OR column2 IS NOT NULL`
Comparison conditions `=`, `<>`, `<`, `<=`, `>`, `>=` with other columns or constants	`WHERE column3 > column4 OR column5 <= 10`

Supported data types for filter pushdown:

YDB Data Type
`Bool`
`Int8`
`Int16`
`Int32`
`Int64`
`Float`
`Double`

Supported Data Types

In the Greenplum database, the optionality of column values (whether a column can 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 system catalog pg_attribute, i.e., at the metadata level of the table. Therefore, all basic Greenplum types can contain `NULL` values by default, and in the YDB type system, they should be mapped to optional types.

Below is a correspondence table between Greenplum and YDB types. All other data types, except those listed, are not supported.

Greenplum Data Type	YDB Data Type	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. Values outside this range return `NULL`.

`timestamp`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. Values outside this range return `NULL`.
`bytea`	`Optional<String>`	
`character`	`Optional<Utf8>`	Default collation rules, string 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

This section provides basic information about working with external Microsoft SQL Server databases.

To work with an external Microsoft SQL Server database, you need to follow these steps:

1. Create a secret containing the password for connecting to the database.

   ```
   CREATE OBJECT ms_sql_server_datasource_user_password (TYPE SECRET) 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 connect to. The `DATABASE_NAME` specifies the database name (for example, `master`). The `LOGIN` and `PASSWORD_SECRET_NAME` parameters are used for authentication to the external database. You can enable encryption for connections to the external database using the `USE_TLS="TRUE"` parameter.

   ```
   CREATE EXTERNAL DATA SOURCE ms_sql_server_datasource WITH (
       SOURCE_TYPE="Microsoft SQL Server",
       LOCATION="<host>:<port>",
       DATABASE_NAME="<database>",
       AUTH_METHOD="BASIC",
       LOGIN="user",
       PASSWORD_SECRET_NAME="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 to the database.

Query Syntax

The following SQL query format is used to work with Microsoft SQL Server:

```
SELECT * FROM ms_sql_server_datasource.<table_name>
```

where:

- `ms_sql_server_datasource` - the external data source identifier;
- `<table_name>` - the table name within 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 within it. This optimization, known as "predicate pushdown", significantly reduces the volume of data transferred from the source to the federated query processing engine. This reduces network load and saves computational resources for YDB.

   A specific case of predicate pushdown, where filtering expressions specified after the `WHERE` keyword are passed down, is called "filter pushdown". Filter pushdown is possible when using:

Description	Example
Filters like `IS NULL` / `IS NOT NULL`	`WHERE column1 IS NULL` or `WHERE column1 IS NOT NULL`
Logical conditions `OR`, `NOT`, `AND`	`WHERE column IS NULL OR column2 IS NOT NULL`
Comparison conditions `=`, `<>`, `<`, `<=`, `>`, `>=` with other columns or constants	`WHERE column3 > column4 OR column5 <= 10`

   Supported data types for filter pushdown:

YDB Data Type
`Bool`
`Int8`
`Int16`
`Int32`
`Int64`
`Float`
`Double`

Supported Data Types

In the Microsoft SQL Server database, the optionality of column values (whether the column can contain `NULL` values or not) is not a part of the data type system. The `NOT NULL` constraint for any column of any table is stored within the `IS_NULLABLE` column in the INFORMATION_SCHEMA.COLUMNS system table, i.e., at the table metadata level. Therefore, all basic Microsoft SQL Server types can contain `NULL` values by default, and in the YDB type system, they should be mapped to optional.

Below is a correspondence table between Microsoft SQL Server types and YDB types. All other data types, except those listed, are not supported.

Microsoft SQL Server Data Type	YDB Data Type	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. Values outside this range return `NULL`.
`smalldatetime`	`Optional<Datetime>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. Values outside this range return `NULL`.
`datetime`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. Values outside this range return `NULL`.
`datetime2`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. Values outside this range return `NULL`.
`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

This section provides basic information about working with external MySQL databases.

To work with an external MySQL database, you need to follow these steps:

1. Create a secret containing the password for connecting to the database.

   ```
   CREATE OBJECT mysql_datasource_user_password (TYPE SECRET) WITH (value = "<password>");
   ```

2. Create an external data source that describes a specific MySQL database. The `LOCATION` parameter contains the network address of the MySQL instance to connect to. The `DATABASE_NAME` specifies the database name (for example, `mysql`). The `LOGIN` and `PASSWORD_SECRET_NAME` parameters are used for authentication to the external database. You can enable encryption for 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_NAME="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 format is used to work with MySQL:

```
SELECT * FROM mysql_datasource.<table_name>
```

where:

- `mysql_datasource` - the external data source identifier;
- `<table_name>` - the table name within 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 within it. This optimization, known as "predicate pushdown", significantly reduces the volume of data transferred from the source to the federated query processing engine. This reduces network load and saves computational resources for YDB.

   A specific case of predicate pushdown, where filtering expressions specified after the `WHERE` keyword are passed down, is called "filter pushdown". Filter pushdown is possible when using:

Description	Example
Filters like `IS NULL` / `IS NOT NULL`	`WHERE column1 IS NULL` or `WHERE column1 IS NOT NULL`
Logical conditions `OR`, `NOT`, `AND`	`WHERE column IS NULL OR column2 IS NOT NULL`
Comparison conditions `=`, `<>`, `<`, `<=`, `>`, `>=` with other columns or constants	`WHERE column3 > column4 OR column5 <= 10`

   Supported data types for filter pushdown:

YDB Data Type
`Bool`
`Int8`
`Uint8`
`Int16`
`Uint16`
`Int32`
`Uint32`
`Int64`
`Uint64`
`Float`
`Double`

Supported Data Types

In the MySQL database, the optionality of column values (whether the column can contain `NULL` values or not) is not a part of the data type system. The `NOT NULL` constraint for any column of any table is stored within the `IS_NULLABLE` column in the INFORMATION_SCHEMA.COLUMNS system table, i.e., at the table metadata level. Therefore, all basic MySQL types can contain `NULL` values by default, and in the YDB type system they should be mapped to optional.

Below is a correspondence table between MySQL types and YDB types. All other data types, except those listed, are not supported.

MySQL Data Type	YDB Data Type	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. Values outside this range return `NULL`.
`datetime`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. Values outside this range return `NULL`.
`timestamp`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. Values outside this range return `NULL`.

`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

This section provides basic information on working with external PostgreSQL databases.

To work with an external PostgreSQL database, you need to follow these steps:

1. Create a secret containing the password for connecting to the database.

   ```
   CREATE OBJECT postgresql_datasource_user_password (TYPE SECRET) WITH (value = "<password>");
   ```

2. Create an external data source that describes a specific database within the PostgreSQL cluster. By default, the namespace `public` is used for reading, but this value can be changed using the optional `SCHEMA` parameter. The network connection is made using the standard (Frontend/Backend Protocol) over TCP transport (`PROTOCOL="NATIVE"`). You can enable encryption of connections to the external database using the `USE_TLS="TRUE"` parameter.

   ```
   CREATE EXTERNAL DATA SOURCE postgresql_datasource WITH (
       SOURCE_TYPE="PostgreSQL",
       LOCATION="<host>:<port>",
       DATABASE_NAME="<database>",
       AUTH_METHOD="BASIC",
       LOGIN="user",
       PASSWORD_SECRET_NAME="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 format is used to work with PostgreSQL:

```
SELECT * FROM postgresql_datasource.<table_name>
```

where:

- `postgresql_datasource` - identifier of the external data source;
- `<table_name>` - table name within 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 within it. This optimization, known as "predicate pushdown", significantly reduces the volume of data transferred from the source to the federated query processing engine. This reduces network load and saves computational resources for YDB.

   A specific case of predicate pushdown, where filtering expressions specified after the `WHERE` keyword are passed down, is called "filter pushdown". Filter pushdown is possible when using:

Description	Example
Filters like `IS NULL` / `IS NOT NULL`	`WHERE column1 IS NULL` or `WHERE column1 IS NOT NULL`
Logical conditions `OR`, `NOT`, `AND`	`WHERE column IS NULL OR column2 IS NOT NULL`
Comparison conditions `=`, `<>`, `<`, `<=`, `>`, `>=` with other columns or constants	`WHERE column3 > column4 OR column5 <= 10`

Supported data types for filter pushdown:

YDB Data Type
`Bool`
`Int8`
`Int16`
`Int32`
`Int64`
`Float`
`Double`

Supported Data Types

In the PostgreSQL database, the optionality of column values (whether a column can 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 system catalog pg_attribute, i.e., at the metadata level of the table. Therefore, all basic PostgreSQL types can contain `NULL` values by default, and in the YDB type system, they should be mapped to optional types.

Below is a correspondence table between PostgreSQL and YDB types. All other data types, except those listed, are not supported.

PostgreSQL Data Type	YDB Data Type	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. Values outside this range return `NULL`.
`timestamp`	`Optional<Timestamp>`	Valid time range from 1970-01-01 00:00:00 to 2105-12-31 23:59:59. Values outside this range return `NULL`.
`bytea`	`Optional<String>`	

`character`	`Optional<Utf8>`	Default collation rules, string padded with spaces to the required length.
`character varying`	`Optional<Utf8>`	Default collation rules.
`text`	`Optional<Utf8>`	Default collation rules.
`json`	`Optional<Json>`	

Working with YDB Databases

YDB can act as an external data source for another YDB database. This section discusses the organization of collaboration between two independent YDB databases in federated query processing mode.

To connect to an external YDB database from another YDB database acting as the federated query engine, the following steps need to be performed on the latter:

1. Prepare authentication data to access the remote YDB database. Currently, in federated queries to YDB, the only available authentication method is login and password (other methods are not supported). The password to the external database is stored as a secret:

   ```
   CREATE OBJECT ydb_datasource_user_password (TYPE SECRET) WITH (value = "<password>");
   ```

2. Create an external data source describing the external YDB database. The `LOCATION` parameter contains the network address of the YDB instance to which the network connection is made. The `DATABASE_NAME` specifies the name of the database (e.g., `local`). For authentication to the external database, the `LOGIN` and `PASSWORD_SECRET_NAME` parameters are used. Encryption of connections to the external database can be enabled using the `USE_TLS="TRUE"` parameter. If encryption is enabled, the `<port>` field in the `LOCATION` parameter should specify the gRPCs port of the external YDB; otherwise, the gRPC port should be specified.

   ```
   CREATE EXTERNAL DATA SOURCE ydb_datasource WITH (
       SOURCE_TYPE="Ydb",
       LOCATION="<host>:<port>",
       DATABASE_NAME="<database>",
       AUTH_METHOD="BASIC",
       LOGIN="user",
       PASSWORD_SECRET_NAME="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 the external data source.

Query Syntax

To retrieve data from tables of the external YDB database, the following form of SQL query is used:

```
SELECT * FROM ydb_datasource.`<table_name>`
```

Where:

- `ydb_datasource` - identifier of the external data source;
- `<table_name>` - full name of the table within the hierarchy of directories in the YDB database, e.g., `table`, `dir1/table1`, or `dir1/dir2/table3`.

If the table is at the top level of the hierarchy (not belonging to any directories), it is permissible not to enclose the table name in backticks "`":

```
SELECT * FROM ydb_datasource.<table_name>
```

Limitations

There are several limitations when working with external YDB data sources:

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 within it. This optimization, known as "predicate pushdown", significantly reduces the volume of data transferred from the source to the federated query processing engine. This reduces network load and saves computational resources for YDB.

   A specific case of predicate pushdown, where filtering expressions specified after the `WHERE` keyword are passed down, is called "filter pushdown". Filter pushdown is possible when using:

Description	Example

Filters like `IS NULL` / `IS NOT NULL`	`WHERE column1 IS NULL` or `WHERE column1 IS NOT NULL`
Logical conditions `OR`, `NOT`, `AND`	`WHERE column IS NULL OR column2 IS NOT NULL`
Comparison conditions `=`, `<>`, `<`, `<=`, `>`, `>=` with other columns or constants	`WHERE column3 > column4 OR column5 <= 10`

Supported data types for filter pushdown:

YDB Data Type
`Bool`
`Int8`
`Uint8`
`Int16`
`Uint16`
`Int32`
`Uint32`
`Int64`
`Uint64`
`Float`
`Double`
`String`
`Utf8`

Supported Data Types

When working with tables located in the external YDB database, users have access to a limited set of data types. All other types, except for those listed below, are not supported. In some cases the type conversion is performed, meaning that the columns of the table from the external YDB database may change their type after being read by the YDB database processing the federated query.

External YDB data type	Federated YDB data type
`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_NAME` – reference to the name of the secret where `AWS_ACCESS_KEY_ID` is stored.
- `AWS_SECRET_ACCESS_KEY_SECRET_NAME` – reference to the name of 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 OBJECT aws_access_id (TYPE SECRET) WITH (value=`<id>`);
CREATE OBJECT aws_access_key (TYPE SECRET) 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_NAME`, `AWS_SECRET_ACCESS_KEY_SECRET_NAME`, 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_NAME="aws_access_id",
  AWS_SECRET_ACCESS_KEY_SECRET_NAME="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
```

Data Formats and Compression Algorithms

This section describes the data formats supported in YDB for storage in S3 and the supported compression algorithms.

Supported Data Formats

The table below lists the data formats supported in YDB.

Format	Read	Write
`csv_with_names`	✓	✓
`tsv_with_names`	✓	
`json_list`	✓	
`json_each_row`	✓	
`json_as_string`	✓	
`parquet`	✓	✓
`raw`	✓	

Format csv_with_names

This format is based on the CSV format. Data is placed in columns separated by commas, with column names in the file's first row.

Example data:

```
Year,Manufacturer,Model,Price
1997,Man_1,Model_1,3000.00
1999,Man_2,Model_2,4900.00
```

Example query

```
SELECT
    AVG(Price)
FROM `connection`.`path`
WITH
(
    FORMAT = "csv_with_names",
    SCHEMA =
    (
        Year Int32,
        Manufacturer Utf8,
        Model Utf8,
        Price Double
    )
)
```

Query result:

#	Manufacturer	Model	Price	Year
1	Man_1	Model_1	3000	1997
2	Man_2	Model_2	4900	1999

Format tsv_with_names

This format is based on the `TSV` format. Data is placed in columns separated by tab characters (code `0x9`), with column names in the file's first row.

Example data:

```
Year    Manufacturer    Model   Price
1997    Man_1   Model_1    3000.00
```

```
1999    Man_2   Model_2    4900.00
```

Example query

```
SELECT
    AVG(Price)
FROM `connection`.`path`
WITH
(
    FORMAT = "tsv_with_names",
    SCHEMA =
    (
        Year Int32,
        Manufacturer Utf8,
        Model Utf8,
        Price Double
    )
)
```

Query result:

#	Manufacturer	Model	Price	Year
1	Man_1	Model_1	3000	1997
2	Man_2	Model_2	4900	1999

Format json_list

This format is based on the JSON representation of data. Each file must contain an array of JSON objects.

Example of valid data (data 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 **INVALID** data (each line contains a separate JSON object, but they 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 }
```

Format json_each_row

This format is based on the JSON representation of data. Each file must contain a JSON object on each line without combining them into a JSON array. This format is used for data streaming systems like Apache Kafka or YDB Topics.

Example of valid data (each line contains a separate JSON object):

```
{ "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
    AVG(Price)
FROM `connection`.`path`
WITH
(
    FORMAT = "json_each_row",
    SCHEMA =
    (
        Year Int32,
        Manufacturer Utf8,
        Model Utf8,
        Price Double
    )
)
```

Query result:

#	Manufacturer	Model	Price	Year
1	Man_1	Model_1	3000	1997
2	Man_2	Model_2	4900	1999

Format json_as_string

This format is based on the JSON representation of data. The `json_as_string` format does not split the input JSON document into fields but represents each file line as a single JSON object (or string). This format is helpful when the list of fields is not the same in all rows and may vary.

Each file must contain:

- A JSON object on each line, or
- JSON objects combined into an array.

Example of valid data (data 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 query

```
SELECT
    *
FROM `connection`.`path`
WITH
(
    FORMAT = "json_as_string",
    SCHEMA =
    (
        Data Json
    )
)
```

Query result:

#	Data
1	`{"Manufacturer": "Man_1", "Model": "Model_1", "Price": 3000, "Year": 1997}`
2	`{"Manufacturer": "Man_2", "Model": "Model_2", "Price": 4900, "Year": 1999}`

Format parquet

This format allows reading the contents of files in Apache Parquet format.

Supported data compression algorithms for Parquet files:

- Uncompressed
- SNAPPY
- GZIP
- LZO
- BROTLI
- LZ4
- ZSTD
- LZ4_RAW

Example query

```
SELECT
    AVG(Price)
FROM `connection`.`path`
WITH
(
    FORMAT = "parquet",
```

```
    SCHEMA =
    (
        Year Int32,
        Manufacturer Utf8,
        Model Utf8,
        Price Double
    )
)
```

Query result:

#	Manufacturer	Model	Price	Year
1	Man_1	Model_1	3000	1997
2	Man_2	Model_2	4900	1999

Format raw

This format allows reading the contents of files as is, in raw form. The data read in this way can be processed using YQL tools, splitting into rows and columns.

This format should be used if the built-in parsing capabilities in YDB are insufficient.

Example query

```
SELECT
    *
FROM `connection`.`path`
WITH
(
    FORMAT = "raw",
    SCHEMA =
    (
        Data String
    )
)
```

Query result:

```
Year,Manufacturer,Model,Price
1997,Man_1,Model_1,3000.00
1999,Man_2,Model_2,4900.00
```

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	Read	Write
Gzip	gzip	✓	✓
Zstd	zstd	✓	
LZ4	lz4	✓	✓
Brotli	brotli	✓	
Bzip2	bzip2	✓	
Xz	xz	✓	

For Parquet file format, the following internal compression algorithms are supported:

Compression Format	Name in YDB	Read	Write
Raw	raw	✓	
Snappy	snappy	✓	✓

YDB does not support working with externally compressed Parquet files, such as files named `<myfile>.parquet.gz` or similar. All files in Parquet format must be without external compression.

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`.

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.

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.

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. YDB uses the TCMalloc memory allocator. To make it efficient, enable Transparent Huge Pages and Memory overcommitment.

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, in the case of 128 CPU cores a viable approach would be to run four 32-CPU dynamic nodes, each in a dedicated taskset.

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.

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 YDB cluster with Ansible
- Deploy infrastructure for YDB cluster using Terraform
- Restarting YDB clusters deployed with Ansible
- Observability:
  - Logging on clusters deployed with Ansible

Working with YDB using Kubernetes

This section of YDB documentation contains a collection of articles intended for SRE's 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

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
- Overview of cluster disk subsystem management
- Setting up monitoring for a YDB cluster
- Logging in YDB
- Backup and recovery
- Using the embedded web UI
- Cluster system views

Deploying YDB cluster with Ansible

This guide outlines the process of deploying a YDB cluster on a group of servers using Ansible. The recommended setup to get started is 3 servers with 3 disk drives for user data each. For reliability purposes each server should have as independent infrastructure as possible: they'd better be each in a separate datacenter or availability zone, or at least in different server racks.

For large-scale setups, it is recommended to use at least 9 servers for highly available clusters (`mirror-3-dc`) or 8 servers for single-datacenter clusters (`block-4-2`). In these cases, servers can have only one disk drive for user data each, but they'd better have an additional small drive for the operating system. You can learn about redundancy models available in YDB from the YDB Cluster Topology article. During operation, the cluster can be expanded without suspending user access to the databases.

Note

Recommended server requirements:

- 16 CPUs (calculated based on the utilization of 8 CPUs by the storage node and 8 CPUs by the dynamic node).
- 16 GB RAM (recommended minimum RAM).
- Additional SSD drives for data, at least 120 GB each.
- SSH access.
- Network connectivity between machines in the cluster.
- OS: Ubuntu 18+, Debian 9+.
- Internet access is needed to update repositories and download necessary packages.

Download the GitHub repository with examples for installing YDB cluster – `git clone https://github.com/ydb-platform/ydb-ansible-examples.git` . This repository contains a few installation templates for deploying YDB clusters in subfolders, as well as scripts for generating TLS certificates and requirement files for installing necessary Python packages. In this article, we'll use the `3-nodes-mirror-3-dc` subfolder for the most simple setup. Alternatively, you can similarly use `8-nodes-block-4-2` or `9-nodes-mirror-3-dc` if you have the necessary number of suitable servers.

Repository Structure

```
├── 3-nodes-mirror-3-dc / 9-nodes-mirror-3-dc / 8-nodes-block-4-2
│   ├── ansible.cfg # An Ansible configuration file containing settings for connecting to servers and project structure options. It is essential for customizing Ansible's behavior and specifying default settings.
│   ├── ansible_vault_password_file # A file containing the password for decrypting encrypted data with Ansible Vault, such as sensitive variables or configuration details. This is crucial for securely managing secrets like the root user password.
│   ├── creds # A directory for environment variables that specify the username and password for YDB, facilitating secure access to the database.
│   ├── files
│   │   ├── config.yaml # A YDB configuration file, which contains settings for the database instances.
│   ├── inventory # A directory containing inventory files, which list and organize the servers Ansible will manage.
│   │   ├── 50-inventory.yaml # The main inventory file, specifying the hosts and groups for Ansible tasks.
│   │   └── 99-inventory-vault.yaml #  An encrypted inventory file storing sensitive information, such as the root user's password for YDB, using Ansible Vault.
├── README.md # A markdown file providing a description of the repository, including how to use it, prerequisites, and any other relevant information.
├── requirements.txt # A file listing Python package dependencies required for the virtual environment, ensuring all necessary tools and libraries are installed.
├── requirements.yaml # Specifies the Ansible collections needed, pointing to the latest versions or specific versions required for the project.
├── TLS #A directory intended for storing TLS (Transport Layer Security) certificates and keys for secure communication.
│   ├── ydb-ca-nodes.txt # Contains a list of Fully Qualified Domain Names (FQDNs) of the servers for which TLS certificates will be generated, ensuring secure connections to each node.
│   └── ydb-ca-update.sh # A script for generating TLS certificates from the ydb-ca-nodes.txt list, automating the process of securing communication within the cluster.
```

To work with the project on a local (intermediate or installation) machine, you will need: Python 3 version 3.10+ and Ansible core version 2.15.2 or higher. Ansible can be installed and run globally (installed in the system) or in a virtual environment. If Ansible is already installed – you can move on to the step "Configuring the Ansible project"; if Ansible is not yet installed, install it using one of the following methods:

Installing Ansible globally (Ubuntu 22.04 LTS)

- 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` (note that installing just `ansible` will lead to an unsuitable outdated version).
- Check the Ansible core version – `ansible --version`

Installing Ansible in a Python virtual environment

- Update the apt package list – `sudo apt-get update`.
- Install the `venv` package for Python3 – `sudo apt-get install python3-venv`
- Create a directory where the virtual environment will be created and where the playbooks will be downloaded. For example, `mkdir venv-ansible`.
- Create a Python virtual environment – `python3 -m venv venv-ansible`.
- Activate the virtual environment – `source venv-ansible/bin/activate`. All further actions with Ansible are performed inside the virtual environment. You can exit it with the command `deactivate`.
- Install the recommended version of Ansible using the command `pip3 install -r requirements.txt`, while in the root directory of the downloaded repository.
- Check the Ansible core version – `ansible --version`

Navigate to the root directory of the downloaded repository and execute the command `ansible-galaxy install -r requirements.yaml` – this will download the Ansible collections `ydb_platform.ydb` and `community.general`, which contain roles and plugins for installing YDB.

Configure the Ansible project

Edit the inventory files

Regardless of the chosen cluster topology (`3-nodes-mirror-3-dc`, `9-nodes-mirror-3-dc`, or `8-nodes-block-4-2`), the main parameters for installing and configuring YDB are contained in the inventory file `50-inventory.yaml`, which is located in the `inventory/` directory.

In the inventory file `50-inventory.yaml`, you need to specify the current list of FQDNs of the servers where YDB will be installed. By default, the list appears as follows:

```
all:
  children:
    ydb:
      static-node-1.ydb-cluster.com:
      static-node-2.ydb-cluster.com:
      static-node-3.ydb-cluster.com:
```

Next, you need to make the following changes in the `vars` section of the inventory file:

- `ansible_user` – specify the user for Ansible to connect via SSH.
- `ansible_ssh_common_args: "-o ProxyJump=<ansible_user>@<static-node-1-IP>"` – option for connecting Ansible to a server by IP, from which YDB will be installed (including ProxyJump server). It is used when installing YDB from a local machine not included in the private DNS zone.
- `ansible_ssh_private_key_file` – change the default private SSH-key path to the actual one: `"../<ssh-private-key-name>"`.
- Choose one of the available options for deploying YDB executables:
  - `ydb_version`: automatically download one of the YDB official releases by version number. For example, `23.4.11`.
  - `ydb_git_version`: automatically compile the YDB executables from the source code, downloaded from the official GitHub repository. The setting's value is a branch, tag, or commit name. For example, `main`.
  - `ydb_archive`: a local filesystem path for a YDB distribution archive downloaded or otherwise prepared in advance.
  - `ydbd_binary` and `ydb_cli_binary`: local filesystem paths for YDB server and client executables, downloaded or otherwise prepared in advance.

Installing fq-connector-go

Installing a connector may be necessary for using federated queries. The playbook can deploy the fq-connector-go to the hosts with dynamic nodes. Use the following settings:

- `ydb_install_fq_connector` - set `true` for installing the 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 the source code, downloaded from the official GitHub repository. The setting's value is a branch, tag, or commit name. For example, `main`.
  - `ydb_fq_connector_archive`: a local filesystem path for a fq-connector-go distribution archive downloaded or otherwise prepared in advance.

- `ydb_fq_connector_binary`: local filesystem paths for fq-connector-go executable, downloaded or otherwise prepared in advance.

Optional changes in the inventory files

Feel free to change these settings if needed, but it is not necessary in straightforward cases:

- `ydb_cores_static` – set the number of CPU cores allocated to static nodes.
- `ydb_cores_dynamic` – set the number of CPU cores allocated to dynamic nodes.
- `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 with names matching node hostnames, containing certificates for a given node. If omitted, self-signed TLS certificates will be generated automatically for the whole 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 of the `ydb_database_groups` setting in the `vars` section depends on available disk drives. Assuming only one database in the cluster, use the following logic:

- For production-grade deployments, use disks with a capacity of over 800 GB and high IOPS, then choose the value for this setting based on the cluster topology:
  - For `block-4-2`, set `ydb_database_groups` to 95% of your total disk drive count, rounded down.
  - For `mirror-3-dc`, set `ydb_database_groups` to 84% of your total disk drive count, 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 properties where the YDB cluster is being deployed. By default, `system_ntp_servers` includes a set of NTP servers without considering the geographical location of the infrastructure on which the YDB cluster will be deployed. 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.

No changes to other sections of the `50-inventory.yaml` configuration file are required.

Changing the root user password

Next, you can change the standard YDB root user password contained in the encrypted inventory file `99-inventory-vault.yaml` and in the file `ansible_vault_password_file.txt`. To change the password – specify the new password in the `ansible_vault_password_file.txt` file and duplicate it in the `99-inventory-vault.yaml` file in the format:

```
all:
  children:
    ydb:
      vars:
        ydb_password: <new-password>
```

To encrypt `99-inventory-vault.yaml`, execute the command `ansible-vault encrypt inventory/99-inventory-vault.yaml`.

After modifying the inventory files, you can proceed to prepare the YDB configuration file.

Prepare the YDB configuration file

The YDB configuration file contains the settings for YDB nodes and is located in the subdirectory `/files/config.yaml`. A detailed description of the configuration file settings for YDB can be found in the article YDB cluster configuration.

The default YDB configuration file already includes almost all the necessary settings for deploying the cluster. You need to replace the standard FQDNs of hosts with the current FQDNs in the `hosts` and `blob_storage_config` sections:

- `hosts` section:

```
...
hosts:
- host: static-node-1.ydb-cluster.com
  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
        pdisk_category: SSD
        path: /dev/disk/by-partlabel/ydb_disk_1
...
```

The rest of the sections and settings in the configuration file can remain unchanged.

fq-connector-go configuration file

Configuration file for fq-connector-go located in the `/files/fq-connector-go/config.yaml`. In straightforward cases, it can remain unchanged.

Deploying the YDB cluster

> **Note**
>
> The minimum number of servers in a YDB cluster is eight servers for the `block-4-2` redundancy model and nine servers for the `mirror-3-dc` redundancy model.
>
> In `mirror-3-dc` servers should be distributed across three availability zones or datacenters as evenly as possible.

The repository contains two ready sets of templates for deploying a YDB cluster of eight (redundancy model `block-4-2`) and nine servers (`mirror-3-dc`). Both options can be scaled to any required number of servers, considering a number of technical requirements.

To prepare your template, you can follow the instructions below:

1. Create a copy of the directory with the ready example (`3-nodes-mirror-3-dc`, `9-nodes-mirror-3-dc`, or `8-nodes-block-4-2`).
2. Specify the FQDNs of the servers in the file `TLS/ydb-ca-nodes.txt` and execute the script `ydb-ca-update.sh` to generate sets of TLS certificates.
3. Change the template's inventory files according to the instructions.
4. Make changes to the YDB configuration file according to the instructions.
5. In the directory of the cloned template, execute the command `ansible-playbook ydb_platform.ydb.initial_setup`.

Installation script execution plan for YDB

The sequence of role executions and their brief descriptions:

1. The `packages` role configures repositories, manages APT preferences and configurations, fixes unconfigured packages, and installs necessary software packages depending on the distribution version.
2. The `system` role sets up system settings, including clock and timezone configuration, time synchronization via NTP with `systemd-timesyncd`, configuring `systemd-journald` for log management, kernel module loading configuration, kernel parameter optimization through `sysctl`, and CPU performance tuning using `cpufrequtils`.
3. The `ydb` role performs tasks related to checking necessary variables, installing base components and dependencies, setting up system users and groups, deploying and configuring YDB, including managing TLS certificates and updating configuration files.
4. The `ydb_fq_connector` role (optional) performs tasks related to deploying and configuring fq-connector-go, including checking necessary variables, installing binaries, configuration files, creating and launching `systemd unit`.

5. The `ydb-static` role prepares and launches static nodes of YDB, including checking necessary variables and secrets, formatting and preparing disks, creating and launching `systemd unit` for the storage node, as well as initializing the storage and managing database access.
6. The `ydb-dynamic` role configures and manages dynamic nodes of YDB, including checking necessary variables, creating configuration and `systemd unit` files for each dynamic node, launching these nodes, obtaining a token for YDB access, and creating a database in YDB.

Detailed step-by-step installation process description

1. Role `packages`. Tasks:

- `check dpkg audit` – Verifies the dpkg state using the `dpkg --audit` command and saves the command results in the `dpkg_audit_result` variable. The task will terminate with an error if the `dpkg_audit_result.rc` command returns a value other than 0 or 1.
- `run the equivalent of "apt-get clean" as a separate step` – Cleans the apt cache, similarly to the `apt-get clean` command.
- `run the equivalent of "apt-get update" as a separate step` – Updates the apt cache, akin to the `apt-get update` command.
- `fix unconfigured packages` – Fixes packages that are not configured using the `dpkg --configure --pending` command.
- `set vars_for_distribution_version variables` – Sets variables for a specific Linux distribution version.
- `setup apt repositories` – Configures apt repositories from a specified list.
- `setup apt preferences` – Configures apt preferences (variable contents are specified in `roles/packages/vars/distributions/<distributive name>/<version>/main.yaml`).
- `setup apt configs` – Configures apt settings.
- `flush handlers` – Forcibly runs all accumulated handlers. In this context, it triggers a handler that updates the apt cache.
- `install packages` – Installs apt packages considering specified parameters and cache validity.

Links to the lists of packages that will be installed for Ubuntu 22.04 or Astra Linux 1.7:

- List of packages for Ubuntu 22.04;
- List of packages for Astra Linux 1.7.

1. Role `system`. Tasks:

- `configure clock` – A block of tasks for setting up system clocks:
  - `assert required variables are defined` – Checks for the existence of the `system_timezone` variable. This check ensures that the necessary variable is available for the next task in the block.
  - `set system timezone` – Sets the system timezone. The timezone is determined by the value of the `system_timezone` variable, and the hardware clock (`hwclock`) is set to UTC. After completing the task, a notification is sent to restart the `cron` service.
  - `flush handlers` – Forces the execution of accumulated handlers using the `meta` directive. This will restart the following processes: `timesyncd`, `journald`, `cron`, `cpufrequtils`, and execute the `sysctl -p` command.
- `configure systemd-timesyncd` – A task block for configuring `systemd-timesyncd`:
  - `assert required variables are defined` asserts that the number of NTP servers (`system_ntp_servers`) is more than one if the variable `system_ntp_servers` is defined. If the variable `system_ntp_servers` is not defined, the execution of the `configure systemd-timesyncd` task block will be skipped, including the check for the number of NTP servers and the configuration of `systemd-timesyncd`.
  - `create conf.d directory for timesyncd` - Creates the `/etc/systemd/timesyncd.conf.d` directory if the `system_ntp_servers` variable is defined.
  - `configure systemd-timesyncd` - Creates a configuration file `/etc/systemd/timesyncd.conf.d/ydb.conf` for the `systemd-timesyncd` service with primary and backup NTP servers. The task is executed if the `system_ntp_servers` variable is defined. After completing the task, a notification is sent to restart the `timesyncd` service.
  - `flush handlers` - Calls accumulated handlers. Executes the handler `restart timesyncd`, which restarts the `systemd-timesyncd.service`.
  - `start timesyncd` - Starts and enables the `systemd-timesyncd.service`. Subsequently, the service will start automatically at system boot.
- `configure systemd-journald` – A block of tasks for configuring the `systemd-journald` service:
  - `create conf.d directory for journald` - Creates the `/etc/systemd/journald.conf.d` directory for storing `systemd-journald` configuration files.
  - `configure systemd-journald` - Creates a configuration file `/etc/systemd/journald.conf.d/ydb.conf` for `systemd-journald`, specifying a `Journal` section with the option `ForwardToWall=no`. The `ForwardToWall=no` setting in the `systemd-journald` configuration means that system log messages will not be forwarded as "wall" messages to all logged-in users. After completing the task, a notification is sent to restart the `journald` service.
  - `flush handlers` - Calls accumulated handlers. Executes the handler `restart journald`, which restarts the `systemd-journald` service.
  - `start journald` - Starts and enables the `systemd-journald.service`. Subsequently, the service will start automatically at system boot.
- `configure kernel` – A block of tasks for kernel configuration:

    - `configure /etc/modules-load.d dir` - Creates the `/etc/modules-load.d` directory with owner and group permissions for the root user and `0755` permissions.
    - `setup conntrack module` - Copies the `nf_conntrack` line into the file `/etc/modules-load.d/conntrack.conf` to load the `nf_conntrack` module at system start.
    - `load conntrack module` - Loads the `nf_conntrack` module in the current session.
    - `setup sysctl files` - Applies templates to create configuration files in `/etc/sysctl.d/` for various system settings (such as security, network, and filesystem settings). The list of files includes `10-console-messages.conf`, `10-link-restrictions.conf`, and others. After completing this task, a notification is sent to apply the kernel settings changes.
    - `flush handlers` - Calls accumulated handlers. Executes the handler `apply kernel settings`, which runs the `sysctl -p` command to apply the kernel parameters specified in `/etc/sysctl.conf` or in other files in the `/etc/sysctl.d/` directory.
- `configure cpu governor` – A block of tasks for configuring the CPU frequency management mode:
    - `install cpufrequtils` - Installs the `cpufrequtils` package from apt. The task is set with cache check parameters and a task timeout of 300 seconds to expedite task execution and avoid an infinite loop waiting for apt package updates.
    - `use performance cpu governor` - Creates the file `/etc/default/cpufrequtils` with content "GOVERNOR=performance", which sets the CPU governor mode to "performance" (disabling power-saving mode when CPU cores are idle). After completing the task, a notification is sent to restart the `cpufrequtils` service.
    - `disable ondemand.service` - Disables the `ondemand.service` if it is present in the system. The service is stopped, its automatic start is disabled, and it is masked (preventing its start). After completing the task, a notification is sent to restart cpufrequtils.
    - `flush handlers` - Calls accumulated handlers. Executes the handler `restart cpufrequtils`, which restarts the `cpufrequtils` service.
    - `start cpufrequtils` - Starts and enables the `cpufrequtils.service`. Subsequently, the service will start automatically at system boot.

1. Role `ydbd`. Tasks:

- `check if required variables are defined` – Checks that the variables `ydb_archive`, `ydb_config`, `ydb_tls_dir` are defined. If any of these are undefined, Ansible will display an appropriate error message and stop the playbook execution.
- `set vars_for_distribution variables` – Sets variables from the specified file in the `vars_for_distribution_file` variable during playbook execution. This task manages a set of variables dependent on the specific Linux distribution.
- `ensure libaio is installed` – Ensures that the `libaio` package is installed.
- `install custom libidn from archive` – Installs a custom version of the `libidn` library from an archive.
- `create certs group` – Creates a system group `certs`.
- `create ydb group` – Creates a system group `ydb`.
- `create ydb user` – Creates a system user `ydb` with a home directory.
- `install YDB server binary package from archive` – Installs YDB from a downloaded archive.
- `create YDB audit directory` – Creates an `audit` subdirectory in the YDB installation directory.
- `setup certificates` – A block of tasks for setting up security certificates:
    - `create YDB certs directory` – Creates a `certs` subdirectory in the YDB installation directory.
    - `copy the TLS ca.crt` – Copies the root certificate `ca.crt` to the server.
    - `copy the TLS node.crt` – Copies the TLS certificate `node.crt` from the generated certificates directory.
    - `copy the TLS node.key` – Copies the TLS certificate `node.key` from the generated certificates directory.
    - `copy the TLS web.pem` – Copies the TLS pem key `web.pem` from the generated certificates directory.
- `copy configuration file` – Copies the configuration file `config.yaml` to the server.
- `add configuration file updater script` – Copies the `update_config_file.sh` script to the server.

1. Role `ydbd_static`. Tasks:

- `check if required variables are defined` – Checks that the variables `ydb_cores_static`, `ydb_disks`, `ydb_domain`, `ydb_user` are defined. If any of these variables are undefined, the task will fail and an appropriate error message will be displayed for each undefined variable.
- `check if required secrets are defined` – Verifies that the secret variable `ydb_password` is defined. If this variable is undefined, the task will fail and an error message will be displayed.
- `create static node configuration file` – Creates a static node configuration file by running the copied `update_config_file.sh` script with `ydbd-config.yaml` and `ydbd-config-static.yaml` configurations.
- `create static node systemd unit` – Creates a `ydbd-storage.service` file for the static node based on a template. After completing the task, a notification is sent to restart the `systemd` service.
- `flush handlers` – Executes accumulated handlers. Restarts all `systemd` services.
- `format drives confirmation block` – A block of tasks for formatting disks and interrupting playbook execution in case the user declines confirmation. A confirmation request to format the connected disk will be displayed in the terminal. Response options: `yes` – to continue executing the playbook with disk formatting. Any other value will be interpreted as a refusal to format. By default, disks are formatted automatically without asking the user for permission, as the variables `ydb_allow_format_drives` and `ydb_skip_data_loss_confirmation_prompt` are set to `true`. If user confirmation is required, the value of the `ydb_skip_data_loss_confirmation_prompt` variable should be changed to `false` in the inventory file `50-inventory.yaml`.
- `prepare drives` – A task for formatting connected disks. Calls the `drive_prepare` plugin – a specially developed Ansible module for YDB installation, which is part of the YDB collection and is located in the directory

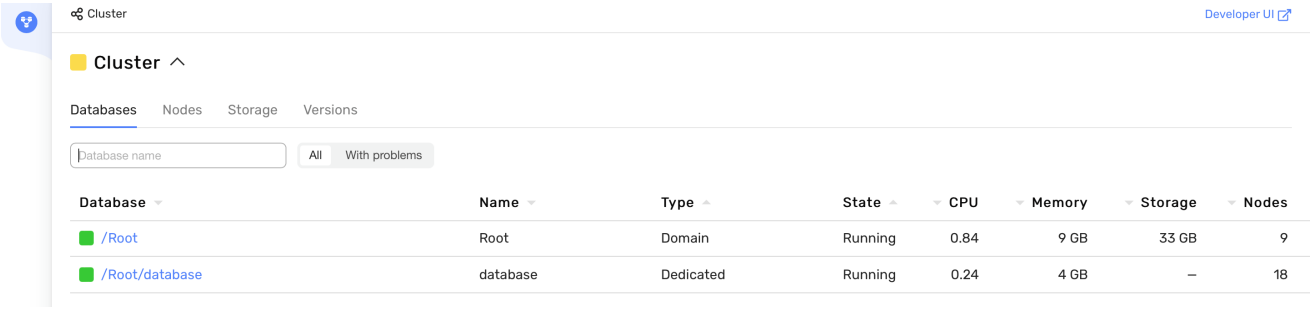
`.../.ansible/collections/ansible_collections/ydb_platform/ydb/plugins/action/drive_prepare.py`. The module will format the connected disk specified in the `ydb_disks` variable if the `ydb_allow_format_drives` variable is set to `true`.

- `start storage node` – Starts the storage node process using `systemd`. If any errors occur during service startup, playbook execution will be interrupted.
- `get ydb token` – Requests a YDB token to perform the storage initialization command. The token is stored in the `ydb_credentials` variable. The task calls the `get_token` module from the directory `.../.ansible/collections/ansible_collections/ydb_platform/ydb/plugins/modules`. If any errors occur at this step, playbook execution will be interrupted.
- `wait for ydb discovery to start working locally` – Calls the `wait_discovery` module, which performs a `ListEndpoints` request to YDB to check the operability of the cluster's basic subsystems. If the subsystems are working properly, storage initialization commands and database creation can be executed.
- `init YDB storage if not initialized` – Initializes the storage if it has not already been created. The task calls the `init_storage` plugin, which performs the storage initialization command using a grpcs request to the static node on port 2135. The command result is stored in the `init_storage` variable.
- `wait for ydb healthcheck switch to "GOOD" status` – Waits for the YDB healthcheck system to switch to a `GOOD` status. The task calls the `wait_healthcheck` plugin, which performs a health check command on YDB.
- `set cluster root password` – Sets the password for the YDB root user. The task is executed by the `set_user_password` plugin, which performs a grpcs request to YDB and sets a pre-defined password for the YDB root user. The password is specified in the `ydb_password` variable in the inventory file `/examples/9-nodes-mirror-3-dc/inventory/99-inventory-vault.yaml` in an encrypted form.

1. Role `ydbd_dynamic`. Tasks:

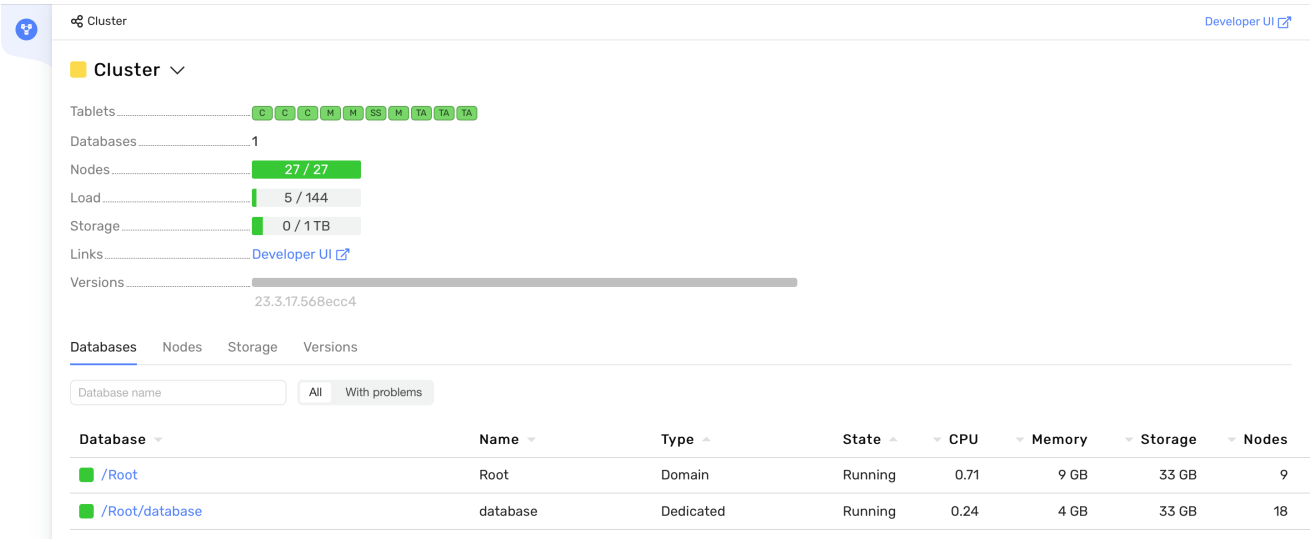
- `check if required variables are defined` – Verifies the presence of required variables (`ydb_domain`, `ydb_pool_kind`, `ydb_cores_dynamic`, `ydb_brokers`, `ydb_dbname`, `ydb_dynnodes`) and displays an error if any variable is missing.
- `create dynamic node configuration file` – Creates a configuration file for dynamic nodes.
- `create dynamic node systemd unit` – Creates a systemd service for dynamic nodes. After completing the task, a notification is sent to restart the `systemd` service.
- `flush handlers` – Executes accumulated handlers. This will restart `systemd`.
- `start dynamic nodes` – Starts the process of dynamic nodes using `systemd`.
- `get ydb token` – Obtains a token for creating a database.
- `create YDB database` – Creates a database. The task is executed by the `create_database` plugin, which performs a request to 99-inventory-vault.yaml to create the database.
- `wait for ydb discovery to start working locally` – Calls the `wait_discovery` module again to check the operability of the cluster's basic subsystems.

As a result of executing the playbook, a YDB cluster will be created, with a test database named `database`, a `root` user with maximum access rights created, and Embedded UI running on port 8765. To connect to the Embedded UI, you can set up SSH tunneling. For this, execute the command `ssh -L 8765:localhost:8765 -i <ssh private key> <user>@<first-ydb-static-node-ip>` on your local machine. After successfully establishing the connection, you can navigate to the URL localhost:8765:



Monitoring the cluster state

After successfully creating the YDB cluster, you can check its state using the Embedded UI – http://localhost:8765/monitoring/cluster/tenants:

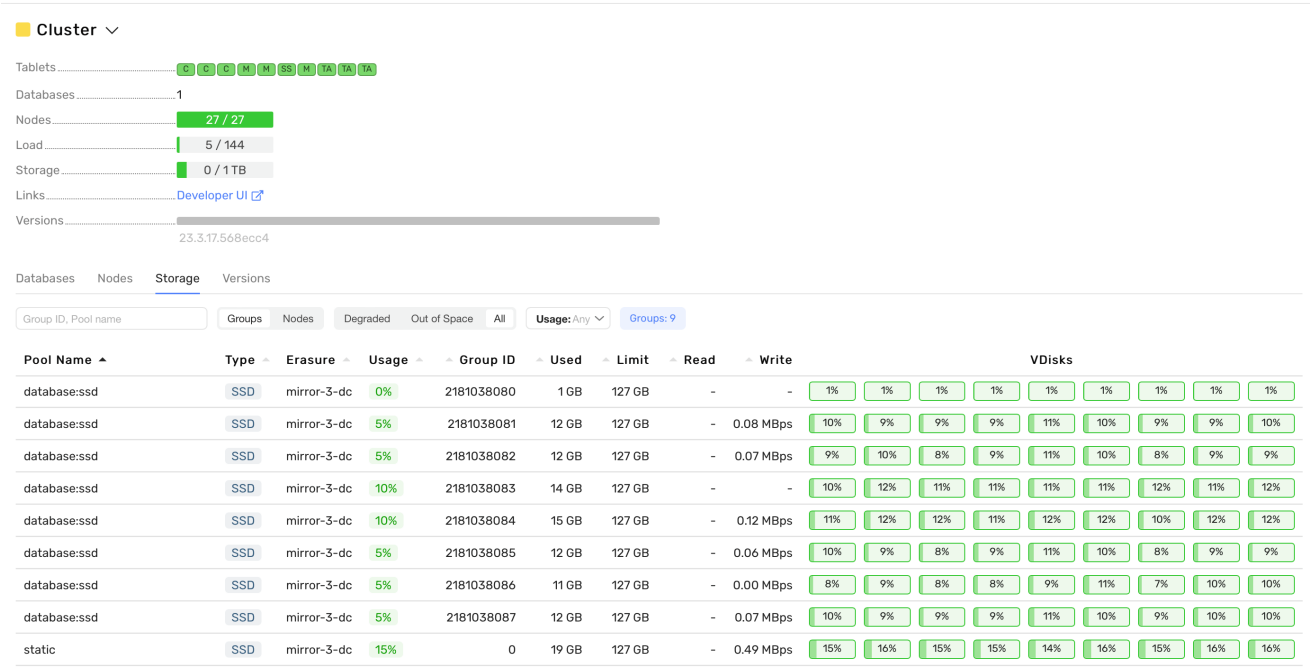


This section displays the following parameters of the YDB cluster, reflecting its state:

- `Tablets` – a list of running tablets. All tablet state indicators should be green;
- `Nodes` – the number and state of static and dynamic nodes launched in the cluster. The node state indicator should be green, and the ratio of created to launched nodes should be equal. For example, 27/27 for a nine-node cluster.

The `Load` indicators (amount of RAM used) and `Storage` (amount of disk space used) should also be green.

You can check the state of the storage group in the `storage` section – http://localhost:8765/monitoring/cluster/storage:



The `VDisks` indicators should be green, and the `state` status (found in the tooltip when hovering over the Vdisk indicator) should be `Ok`. More about the cluster state indicators and monitoring can be read in the article YDB Monitoring.

Cluster Testing

You can test the cluster using the built-in load tests in YDB CLI. To do this, download YDB CLI version 2.5.0 to the machine where Ansible is installed. For example, using wget: `wget https://storage.yandexcloud.net/yandexcloud-ydb/release/2.5.0/linux/amd64/ydb`.

Make the downloaded binary file executable – `chmod +x ydb` and execute the connection check command:

```
./ydb \
config profile create <profile name> \
-d /Root/database \
-e grpcs://< FQDN node >:2135 \
--ca-file <path to generated certs>/CA/certs/ca.crt \
--user root \
--password-file <path to vault password file>/ansible_vault_password_file
```

Command parameters and their values:

- `config profile create` – This command is used to create a connection profile. You specify the profile name. More detailed information on how to create and modify profiles can be found in the article Creating and updating profiles.
- `-e` – Endpoint, a string in the format `protocol://host:port`. You can specify the FQDN of any cluster node and omit the port. By default, port 2135 is used.
- `--ca-file` – Path to the root certificate for connections to the database using `grpcs`. The certificate is created by the `ydb-ca-update.sh` script in the `TLS` directory and is located at the path `TLS/CA/certs/` relative to the root of the `ydb-ansible-examples` repository.
- `--user` – The user for connecting to the database. By default, the user `root` is created when executing the `ydb_platform.ydb.initial_setup` playbook.
- `--password-file` – Path to the password file. In each folder with a YDB cluster deployment template, there is an `ansible_vault_password_file` that contains the password for the user `root`.

You can check if the profile has been created using the command `./ydb config profile list`, which will display a list of profiles. After creating a profile, you need to activate it with the command `./ydb config profile activate <profile name>`. To verify that the profile has been activated, you can rerun the command `./ydb config profile list` – the active profile will have an (active) mark.

To execute a YQL query, you can use the command `./ydb sql -s 'select 1;'`, which will return the result of the `select 1` command in table form to the terminal. After checking the connection, you can create a test table with the command: `./ydb workload kv init --init-upserts 1000 --cols 4`. This will create a test table `kv_test` consisting of 4 columns and 1000 rows. You can verify that the `kv_test` table was created and filled with test data by using the command `./ydb sql -s 'select * from kv_test limit 10;'`.

The terminal will display a table of 10 rows. Now, you can perform cluster performance testing. The article Key-Value load describes 5 types of workloads (`upsert`, `insert`, `select`, `read-rows`, `mixed`) and the parameters for their execution. An example of executing the `upsert` test workload with the parameter to print the execution time `--print-timestamp` and standard execution parameters is: `./ydb workload kv run upsert --print-timestamp`.

A report of the following type will be displayed in the terminal:

```
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
...
```

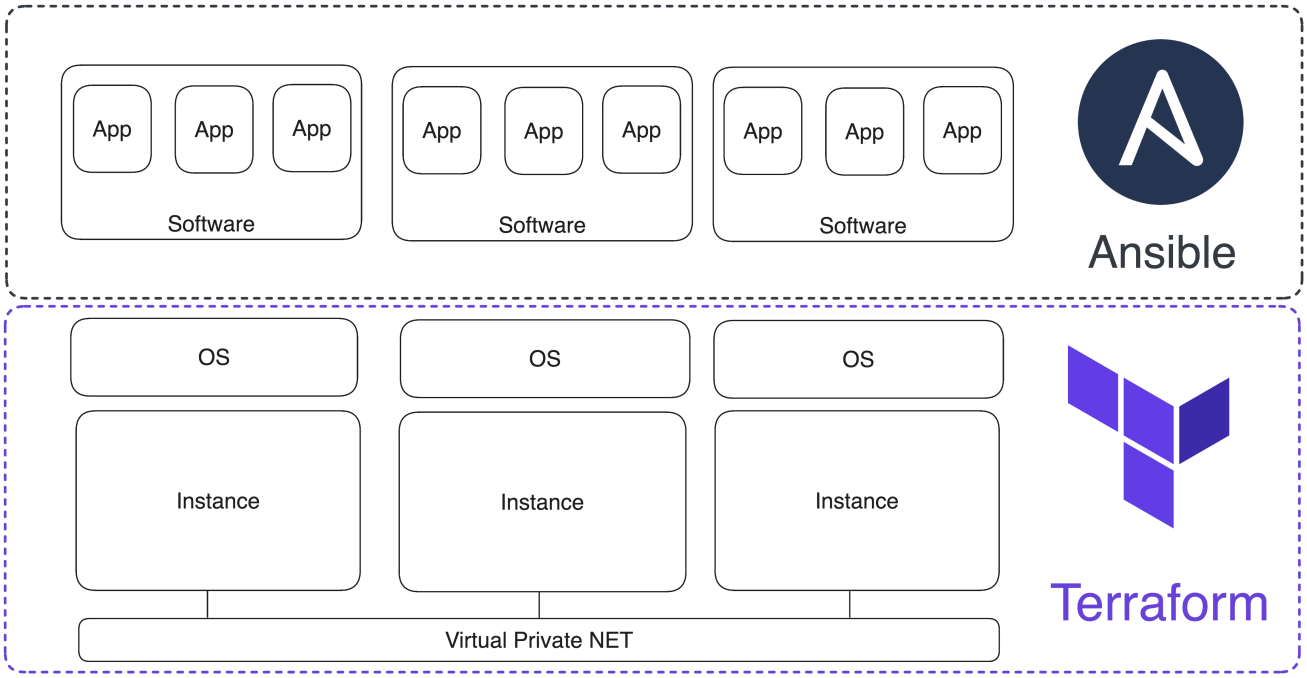
After completing the tests, the `kv_test` table can be deleted with the command: `./ydb workload kv clean`. More details on the options for creating a test table and tests can be read in the article Key-Value load.

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 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 init` – create resources in the cloud.

Next, use the commands `terraform plan`, `terraform init`, and `terraform destroy` (destruction of the created infrastructure) to apply further changes as necessary.

Create infrastructure in Azure to deploy 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 init` – create resources in the cloud.

Next, use the commands `terraform plan`, `terraform init`, and `terraform destroy` (destruction of the created infrastructure) to apply further changes as necessary.

Creating infrastructure in Google Cloud Platform to deploy 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 init` – create resources in the cloud.

Next, use the commands `terraform plan`, `terraform init`, and `terraform destroy` (destruction of the created infrastructure) to apply further changes as necessary.

Creating an infrastructure in Yandex Cloud for deploying the 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 init` – create resources in the cloud.

Next, use the commands `terraform plan`, `terraform init`, 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
```

Dynamic configuration

YDB has its own configuration management mechanism called dynamic configuration. It is agnostic of the chosen cluster deployment method and allows for flexible cluster configuration even in multitenant environments.

Consider switching to dynamic configuration if manually deploying config files and doing a complete rolling restart on any change is no longer feasible.

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     exited
   Configure Transparent Huge Pages (THP)
   ```

```
ydbd-database-a.service                                                  loaded active     running
YDB dynamic node / database / a
ydbd-database-b.service                                                  loaded active     running
YDB dynamic node / database / b
ydbd-storage.service                                                     loaded active     running
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.

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. We'll follow this process in more detail below. 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/storage-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/storage-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/database.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
...
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:22.4.44 --rm ydb-cli bash
```

3. Query the YDB database:

```
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 resources you don't use

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.

Deploying YDB cluster manually

This document describes how to deploy a multi-tenant YDB cluster on multiple bare-metal or virtual servers.

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 Embedded 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
```

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.

Prepare configuration files

Prepare a configuration file for YDB:

1. Download a sample config for the appropriate failure model of your cluster:
   - block-4-2: For a single-data center cluster.
   - mirror-3dc: For a cross-data center cluster consisting of 9 nodes.
   - mirror-3dc-3nodes: For a cross-data center cluster consisting of 3 nodes.
2. In the `host_configs` section, specify all disks and their types on each cluster node. Possible disk types:
   - ROT: Rotational, HDD.
   - SSD: SSD or NVMe.

   ```
   host_configs:
   - drive:
     - path: /dev/disk/by-partlabel/ydb_disk_ssd_01
       type: SSD
     host_config_id: 1
   ```

3. In the `hosts` section, specify the FQDN of each node, their configuration and location in a `data_center` or `rack`:

   ```
   hosts:
   - host: node1.ydb.tech
     host_config_id: 1
     walle_location:
       body: 1
       data_center: 'zone-a'
       rack: '1'
   - host: node2.ydb.tech
     host_config_id: 1
     walle_location:
       body: 2
       data_center: 'zone-b'
       rack: '1'
   - host: node3.ydb.tech
     host_config_id: 1
     walle_location:
       body: 3
   ```

```
    data_center: 'zone-c'
    rack: '1'
```

4. Under `blob_storage_config`, edit the FQDNs of all the nodes accommodating your static storage group:
   - For the `mirror-3-dc` scheme, specify FQDNs for nine nodes.
   - For the `block-4-2` scheme, specify FQDNs for eight nodes.
5. Enable user authentication (optional).

   If you plan to use authentication and user access differentiation features in the YDB cluster, add the following parameters to the `domains_config` section:

```
domains_config:
  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"
```

In the traffic encryption mode, make sure that the YDB configuration file specifies paths to key files and certificate files under `interconnect_config` and `grpc_config`:

```
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
```

Save the YDB configuration file as `/opt/ydb/cfg/config.yaml` on each cluster node.

For more detailed information about creating the configuration file, see YDB cluster configuration.

Copy the TLS keys and certificates to each server

Make sure to copy the generated TLS keys and certificates to a protected folder on each YDB cluster node. Below are sample commands that create a protected folder and copy files with keys and certificates.

```
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 a 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 `/etc/systemd/system/ydbd-storage.service` configuration file by 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
```

Initialize a cluster

The cluster initialization operation sets up static nodes listed in the cluster configuration file, for storing YDB data.

To initialize the cluster, you'll need the `ca.crt` file issued by the Certificate Authority. Use its path in the initialization commands. Before running the commands, copy `ca.crt` to the server where you will run the commands.

Cluster initialization actions depend on whether the user authentication mode is enabled in the YDB configuration file.

Authentication enabled

To execute administrative commands (including cluster initialization, database creation, disk management, and others) in a cluster with user authentication mode enabled, you must first get an authentication token using the YDB CLI client version 2.0.0 or higher. You must install the YDB CLI client on any computer with network access to the cluster nodes (for example, on one of the cluster nodes) by following the installation instructions.

When the cluster is first installed, it has a single `root` account with a blank password, so the command to get the token is the following:

```
ydb -e grpcs://<node1.ydb.tech>:2135 -d /Root --ca-file ca.crt \
  --user root --no-password auth get-token --force >token-file
```

You can specify any storage server in the cluster as an endpoint (the `-e` or `--endpoint` parameter).

If the command above is executed successfully, the authentication token will be written to `token-file`. Copy the token file to one of the storage servers in the cluster, then run the following commands on the server:

```
export LD_LIBRARY_PATH=/opt/ydb/lib
/opt/ydb/bin/ydbd -f token-file --ca-file ca.crt -s grpcs://`hostname -f`:2135 \
    admin blobstorage config init --yaml-file  /opt/ydb/cfg/config.yaml
echo $?
```

Authentication disabled

On one of the storage servers in the cluster, run these commands:

```
export LD_LIBRARY_PATH=/opt/ydb/lib
/opt/ydb/bin/ydbd --ca-file ca.crt -s grpcs://`hostname -f`:2135 \
    admin blobstorage config init --yaml-file  /opt/ydb/cfg/config.yaml
echo $?
```

You will see that the cluster was initialized successfully when the cluster initialization command returns a zero code.

Create a database

To work with row-oriented and column-oriented tables, you need to create at least one database and run a process (or processes) to serve this database (dynamic nodes):

To execute the administrative command for database creation, you will need the `ca.crt` certificate file issued by the Certificate Authority (see the above description of cluster initialization).

When creating your database, you set an initial number of storage groups that determine the available input/output throughput and maximum storage. For an existing database, you can increase the number of storage groups when needed.

The database creation procedure depends on whether you enabled user authentication in the YDB configuration file.

Authentication enabled

Get an authentication token. Use the authentication token file that you obtained when initializing the cluster or generate a new token.

Copy the token file to one of the storage servers in the cluster, then run the following commands on the server:

```
export LD_LIBRARY_PATH=/opt/ydb/lib
/opt/ydb/bin/ydbd -f token-file --ca-file ca.crt -s grpcs://`hostname -f`:2135 \
    admin database /Root/testdb create ssd:1
echo $?
```

Authentication disabled

On one of the storage servers in the cluster, run these commands:

```
export LD_LIBRARY_PATH=/opt/ydb/lib
/opt/ydb/bin/ydbd --ca-file ca.crt -s grpcs://$(hostname -f):2135 \
    admin database /Root/testdb create ssd:1
echo $?
```

You will see that the database was created successfully when the command returns a zero code.

The command example above uses the following parameters:

- `/Root`: Name of the root domain, must match the `domains_config`.`domain`.`name` setting in the cluster configuration file.
- `testdb`: Name of the created database.
- `ssd:1`: Name of the storage pool and the number of storage groups allocated. The pool name usually means the type of data storage devices and must match the `storage_pool_types`.`kind` setting inside the `domains_config`.`domain` element of the configuration file.

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 \
    --node-broker grpcs://<ydb1>:2135 \
    --node-broker grpcs://<ydb2>:2135 \
    --node-broker grpcs://<ydb3>:2135
```

In the command example above, `<ydbN>` is replaced by FQDNs of any three servers running the cluster's static nodes.

Using systemd

Create a systemd configuration file named `/etc/systemd/system/ydbd-testdb.service` by the following template: 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 \
    --node-broker grpcs://<ydb1>:2135 \
    --node-broker grpcs://<ydb2>:2135 \
    --node-broker grpcs://<ydb3>:2135
LimitNOFILE=65536
LimitCORE=0
LimitMEMLOCK=32212254720

[Install]
WantedBy=multi-user.target
```

In the file example above, `<ydbN>` is replaced by 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 to ensure database scalability and fault tolerance.

Initial account setup

If authentication mode is enabled in the cluster configuration file, initial account setup must be done before working with the YDB cluster.

The initial installation of the YDB cluster automatically creates a `root` account with a blank password, as well as a standard set of user groups described in the Built-in groups section.

To perform initial account setup in the created YDB cluster, run the following operations:

1. Install the YDB CLI as described in the documentation.
2. Set the password for the `root` account:

```
ydb --ca-file ca.crt -e grpcs://<node.ydb.tech>:2136 -d /Root/testdb --user root --no-password \
    yql -s 'ALTER USER root PASSWORD "passw0rd"'
```

Replace the value `passw0rd` with the required password. Save the password in a separate file. Subsequent commands as the root user will be executed using the password passed with the `--password-file <path_to_user_password>` option. Additionally, the password can be saved in the connection profile, as described in the documentation for YDB CLI.

1. Create additional accounts:

```
ydb --ca-file ca.crt -e grpcs://<node.ydb.tech>:2136 -d /Root/testdb --user root --password-file
<path_to_root_pass_file> \
    yql -s 'CREATE USER user1 PASSWORD "passw0rd"'
```

1. Set the account rights by including them in the integrated groups:

```
ydb --ca-file ca.crt -e grpcs://<node.ydb.tech>:2136 -d /Root/testdb --user root --password-file
<path_to_root_pass_file> \
    yql -s 'ALTER GROUP `ADMINS` ADD USER user1'
```

In the command examples listed above, `<node.ydb.tech>` is the FQDN of the server where any dynamic node servicing the `/Root/testdb` database is running. When connecting via SSH to a YDB node, it's convenient to use the `grpcs://$(hostname -f):2136` command to use the current server's FQDN.

Start using the created database

1. Install the YDB CLI as described in the documentation.
2. Create a test row (`test_row_table`) or column (`test_column_table`) oriented 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, title Utf8, PRIMARY KEY (id)) WITH (STORE =
COLUMN);'
```

Here, `<node.ydb.tech>` is the FQDN of the server running the dynamic node that serves the `/Root/testdb` database.

Checking access to the built-in web interface

To check access to the YDB built-in web interface, open in the browser the `https://<node.ydb.tech>:8765` URL, where `<node.ydb.tech>` is the FQDN of the server running any static YDB node.

In the web browser, set as trusted 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 web browser should prompt you for a login and password. Enter your credentials, and you'll see the built-in interface welcome page. The user interface and its features are described in Using the embedded web UI.

> **Note**
>
> A common way to provide access to the YDB built-in web interface is to set up a fault-tolerant HTTP balancer running `haproxy` , `nginx` , or similar software. A detailed description of the HTTP balancer is beyond the scope of the standard YDB installation guide.

Installing YDB in the unprotected mode

> **Warning**
>
> We do not recommend using the unprotected YDB mode for development or production environments.

The above installation procedure assumes that YDB was deployed in the standard protected mode.

The unprotected YDB mode is primarily intended for test scenarios associated with YDB software development and testing. In the unprotected mode:

- Traffic between cluster nodes and between applications and the cluster runs over an unencrypted connection.
- Users are not authenticated (it doesn't make sense to enable authentication when the traffic is unencrypted because the login and password in such a configuration would be transparently transmitted across the network).

When installing YDB to run in the unprotected mode, follow the above procedure, with the following exceptions:

1. When preparing for the installation, you do not need to generate TLS certificates and keys and copy the certificates and keys to the cluster nodes.
2. In the configuration files, remove the `security_config` subsection under `domains_config` . Remove the `interconnect_config` and `grpc_config` sections entirely.
3. Use simplified commands to run static and dynamic cluster nodes: omit the options that specify file names for certificates and keys; use the `grpc` protocol instead of `grpcs` when specifying the connection points.
4. Skip the step of obtaining an authentication token before cluster initialization and database creation because it's not needed in the unprotected mode.
5. 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 $?
```

1. 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
```

1. When accessing your database from the YDB CLI and applications, use grpc instead of grpcs and skip authentication.

Backup and recovery

Backups protect against data loss by letting you restore data.

YDB provides multiple solutions for backup and recovery:

- Backing up data to files and restoring it using the YDB CLI.
- Backing up data to S3-compatible storage and restoring it using the YDB CLI.

YDB CLI

Files

To back up data to a file, the following commands are used:

- `ydb admin cluster dump` — to back up a cluster' metadata
- `ydb admin database dump` — to back up a database
- `ydb tools dump` — to back up individual schema objects or directories

To learn more about these commands, see Exporting data to the file system.

To restore data from a backup, the following commands are used:

- `ydb admin cluster restore` — to restore a cluster' metadata from a backup
- `ydb admin database restore` — to restore a database from a backup
- `ydb tools restore` — to restore individual schema objects or directories from a backup

To learn more about these command, see Importing data from the file system.

S3-compatible storage

To back up data to S3-compatible storage (such as AWS S3), run the `ydb export s3` command. To learn more about this command, follow the link to the YDB CLI reference.

To restore data from a backup created in S3-compatible storage, run the `ydb import s3` command. To learn more about this command, follow the link to the YDB CLI reference.

< path="_includes/backup_and_recovery/others_overlay.md" keyword="undefined">

Setting up monitoring for a YDB cluster

This page provides instructions on how to set up monitoring for a YDB cluster.

YDB has multiple system health metrics. Instant metrics are available in the web interface:

```
http://<ydb-server-address>:<ydb-port>/counters/
```

- `<ydb-server-address>`: YDB server address.

  For a local YDB cluster that is deployed using Quick start use the `localhost` address.
- `<ydb-port>`: YDB port. Default value: 8765.

Linked metrics are grouped into subgroups (such as `counters auth`). To only view metrics for a particular subgroup, follow a URL like:

```
http://<ydb-server-address>:<ydb-port>/counters/counters=<servicename>/
```

- `<servicename>`: metrics subgroup name.

For example, data about the utilization of server hardware resources is available at the URL:

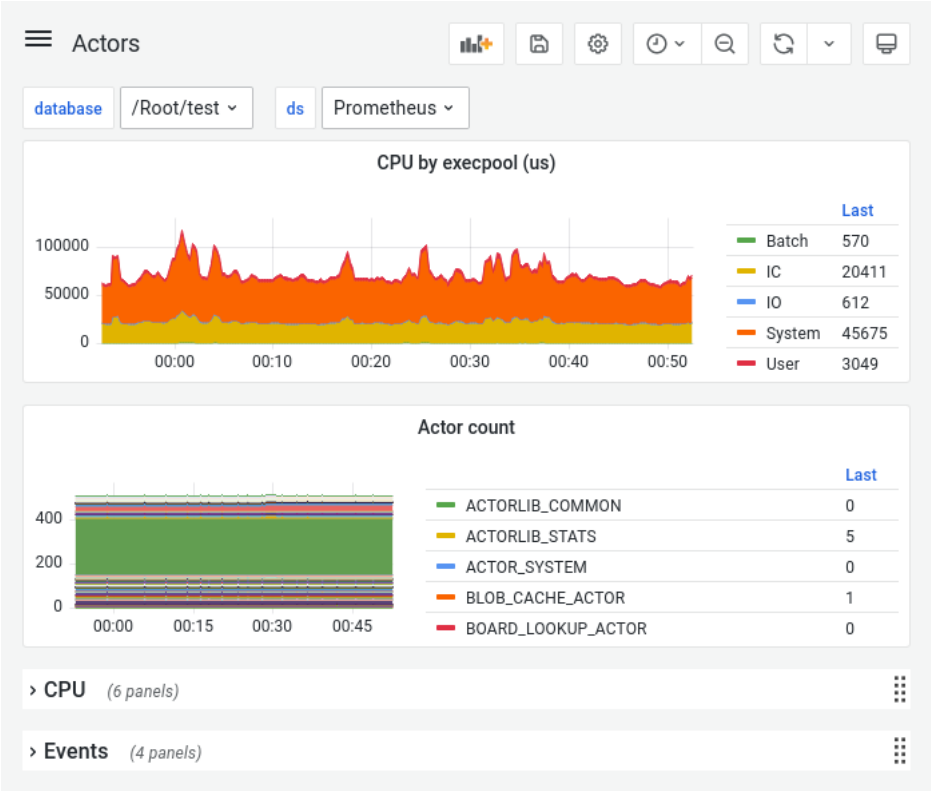
```
http://<ydb-server-address>:<ydb-port>/counters/counters=utils
```

You can collect metrics using Prometheus, a popular open-source observability tool, or any other system compatible with its format. YDB metrics in Prometheus format are available at a URL in the following format:

```
http://<ydb-server-address>:<ydb-port>/counters/counters=<servicename>/prometheus
```

- `<servicename>`: metrics subgroup name.

To visualize data, use any system that supports Prometheus, such as Grafana, Zabbix, or AWS CloudWatch:



Setting up monitoring with Prometheus and Grafana

To set up monitoring for a YDB cluster using Prometheus and Grafana:

1. Install Prometheus.
2. Edit the Prometheus configuration file:
   2.1. In the `targets` section specify addresses of all servers of the YDB cluster and ports for each storage and database node that runs on the server.

For example, for the YDB cluster that contains three servers, each server running one storage node on port 8765 and two database nodes on ports 8766 and 8767, specify nine addresses for all metrics subgroups except for the disk subgroups (for disk metrics subgroups, specify only storage node addresses):

```
static_configs:
- targets:
  - ydb-s1.example.com:8765
  - ydb-s1.example.com:8766
  - ydb-s1.example.com:8767
  - ydb-s2.example.com:8765
  - ydb-s2.example.com:8766
  - ydb-s2.example.com:8767
  - ydb-s3.example.com:8765
  - ydb-s3.example.com:8766
  - ydb-s3.example.com:8767
```

For a local single-node YDB cluster, specify one address in the `targets` section:

```
- targets: ["localhost:8765"]
```

2.2. If necessary, in the `tls_config` section, specify the CA-issued certificate used to sign the other TLS certificates of the YDB cluster:

```
tls_config:
    ca_file: '<ydb-ca-file>'
```

3. Run Prometheus using the edited configuration file.
4. Install and start Grafana.
5. Create a data source of the `prometheus` type in Grafana, and attach it to the running Prometheus instance.
6. Upload YDB dashboards to Grafana.

To upload dashboards, use the Grafana UI Import tool or run a script. Please note that the script uses basic authentication in Grafana. For other cases, modify the script.

Review the dashboard metric reference.

Logging in YDB

Each YDB component writes messages to logs at different levels. They can be used to detect severe issues or identify the root causes.

Logging setup

You can configure logging for the various components of the YDB in the Embedded UI.

There are currently two options for running YDB logging: manually or using systemd.

Manually

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.

Using systemd

Default logs are written to `journald` and can be retrieved via `journalctl -u ydbd`.

YDB updating

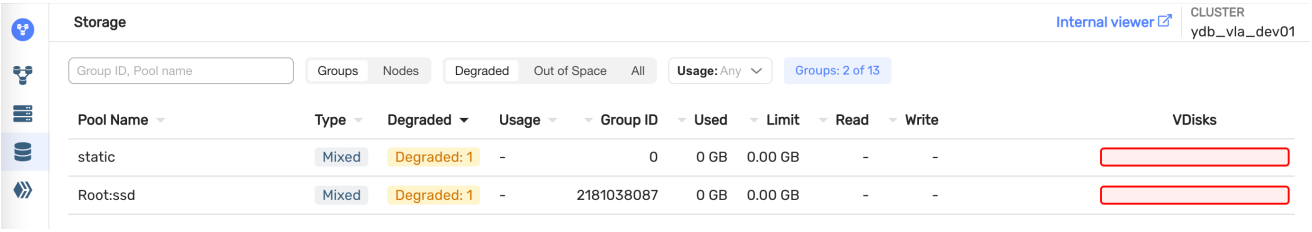
YDB is a distributed system that supports rolling restart without downtime or performance degradation.

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 controlled by an attacker 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 options are shown in the table below.

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. For internal technical reasons, the list must include the `root@builtin` element. Example:

```
domains_config:
    ...
    security_config:
        enforce_user_token_requirement: true
        monitoring_allowed_sids:
        ...
        administration_allowed_sids:
        ...
        viewer_allowed_sids:
        ...
        register_dynamic_node_allowed_sids:
        - "root@builtin" # required for internal technical reasons
        - "registerNode@cert"
```

For more detailed information on configuring cluster authentication parameters, see the relevant documentation section.

3. Deploy the static configuration files on all cluster nodes either manually, or using the Ansible playbook action.
4. Perform the rolling restart of storage nodes by using ydbops or Ansible playbook action.
5. Perform the rolling restart of database nodes through ydbops or Ansible playbooks.

Changing an actor system's 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
      }
    }
  }
}

```bash
ydbd -s <endpoint> admin console execute --domain=<domain> --retry=10 actorsystem.txt
```
```

Updating configurations via CMS

Get the current settings

The following command will let you get the current settings for a 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 the settings

First, you need to pull the desired config as indicated above and then prepare a protobuf file with an update 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 use settings for a specific tenant.

```
ydbd -s <endpoint> admin console configs update <protobuf-file>
```

Cluster system views

To enable internal introspection of the cluster state, the user can make queries to special system views. These tables are accessible from the cluster's root directory and use the `.sys` system path prefix.

Cloud database users usually don't have access to cluster system tables, as cluster support and timely diagnostics are the prerogatives of the cloud team.

Hereinafter, in the descriptions of available fields, the **Key** column contains the corresponding table's primary key field index.

> **Note**
>
> There are also similar system views describing what's happening inside a given database, they are covered in a separate article for DBAs.

Distributed Storage

Information about the operation of distributed storage is contained in several interconnected tables, each of which is responsible for describing its entity, such as:

- PDisk
- VSlot
- Group
- Storage Pool

In addition, there is a separate table that shows statistics on the use of the number of groups in different storage pools and whether these pools can be increased.

ds_pdisks

Field	Type	Key	Value
NodeId	Uint32	0	ID of the node where a PDisk is running.
PDiskId	Uint32	1	ID of the PDisk (unique within the node).
Type	String		Media type (ROT, SSD, NVME).
Kind	Uint64		A user-defined numeric ID that is needed to group disks with the same type of media into different subgroups.
Path	String		Path to the block device inside the machine.
Guid	Uint64		A unique ID that is generated randomly when adding a disk to the system and is designed to prevent data loss in the event of disk swapping.
BoxId	Uint64		ID of the Box that this PDisk belongs to.
SharedWithOs	Bool		Flag indicating if the "SharedWithOs" label is available. Set manually when creating a PDisk. You can use it to filter disks when creating new groups.
ReadCentric	Bool		Flag indicating if the "ReadCentric" label is available. Set manually when creating a PDisk. You can use it to filter disks when creating new groups.
AvailableSize	Uint64		The number of bytes that can be allocated on the PDisk.
TotalSize	Uint64		The total number of bytes on the PDisk.
Status	String		PDisk operation mode that affects its participation in the allocation of groups (ACTIVE, INACTIVE, BROKEN, FAULT, and TO_BE_REMOVED).
StatusChangeTimestamp	Timestamp		The time when the Status was last changed. NULL indicates that the Status hasn't changed since the creation of PDisk.
ExpectedSlotCount	Uint32		The maximum number of VSlots that can be created on this PDisk.
NumActiveSlots	Uint32		The number of slots that are currently active.

ds_vslots

Field	Type	Key	Value
NodeId	Uint32	0	ID of the node where a VSlot is running.
PDiskId	Uint32	1	ID of the PDisk inside the node where the VSlot is running.
VSlotId	Uint32	2	ID of the VSlot inside the PDisk.
GroupId	Uint32		Number of the storage group that this VSlot belongs to.
GroupGeneration	Uint32		Generation of the storage group configuration that this VSlot belongs to.
FailRealm	Uint32		Relative number of the fail realm of the VSlot within the storage group.
FailDomain	Uint32		Relative number of the fail domain of the VSlot within the fail realm.
VDisk	Uint32		Relative number of the VSlot inside the fail domain.
AllocatedSize	Uint64		The number of bytes that the VSlot occupies on the PDisk.
AvailableSize	Uint64		The number of bytes that can be allocated to this VSlot.
Status	String		Status of the VDisk running in this VSlot (INIT_PENDING, REPLICATING, READY, or ERROR).
Kind	String		Preset VDisk operation mode (Default, Log, ...).

Please not that the (NodeId, PDiskId) tuple creates an external key to the `ds_pdisks` table and the (GroupId) to the `ds_groups` table.

ds_groups

Field	Type	Key	Value
GroupId	Uint32	0	Number of the storage group in the cluster.
Generation	Uint32		Storage group configuration generation.
ErasureSpecies	String		Group redundancy coding mode (block-4-2, mirror-3-dc, mirror-3of4, ...).
BoxId	Uint64		ID of the Box that this group is created in.
StoragePoolId	Uint64		ID of the storage pool inside the Box that this group operates in.

EncryptionMode	Uint32		Group data encryption and its algorithm (if enabled).
LifeCyclePhase	Uint32		Availability of a generated encryption key (if encryption is enabled).
AllocatedSize	Uint64		The number of allocated bytes of data in the group (reduced to user bytes, that is, to redundancy).
AvailableSize	Uint64		The number of bytes of user data available for allocation (up to redundancy as well).
SeenOperational	Bool		A Boolean flag that indicates whether the group was operational after its creation.
PutTabletLogLatency	Interval		90th percentile of the PutTabletLog request execution time.
PutUserDataLatency	Interval		90th percentile of the PutUserData request execution time.
GetFastLatency	Interval		90th percentile of the GetFast request execution time.

In this table, the (BoxId, StoragePoolId) tuple creates an external key to the `ds_storage_pools` table.

ds_storage_pools

Field	Type	Key	Value
BoxId	Uint64	0	ID of the Box that this storage pool belongs to.
StoragePoolId	Uint64	1	ID of the storage pool inside the Box.
Name	String		User-defined storage pool name (used when linking tablets and storage pools).
Generation	Uint64		Storage pool configuration generation (number of changes).
ErasureSpecies	String		Redundancy coding mode for all groups within this storage pool.
VDiskKind	String		Preset operation mode for all VDisks in this storage pool.
Kind	String		A user-defined string description of the purpose of the pool, which 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		ID of the SchemeShard object of the schema that this storage pool belongs to (as of now, always NULL).
PathId	Uint64		ID of the node of the schema object inside the specified SchemeShard that this storage pool belongs to.

ds_storage_stats

Unlike other tables that show physical entities, the `ds_storage_stats` table shows aggregated storage statistics.

Field	Type	Key	Value
BoxId	Uint64	0	ID of the Box that statistics are calculated for.

PDiskFilter	String	1	A string description of filters that select a PDisk to create groups (for example, by media type).
ErasureSpecies	String	2	Redundancy coding mode that statistics are collected for.
CurrentGroupsCreated	Uint32		Number of groups created with the specified characteristics.
CurrentAllocatedSize	Uint64		Total space occupied by all groups from CurrentGroupsCreated.
CurrentAvailableSize	Uint64		Total space that is available to all groups from CurrentGroupsCreated.
AvailableGroupsToCreate	Uint32		Number of groups with the specified characteristics that can be created taking into account the need for a reserve.
AvailableSizeToCreate	Uint64		Number of available bytes that will be obtained when creating all groups from AvailableGroupsToCreate.

It should be noted that AvailableGroupsToCreate shows the maximum number of groups that can be created if no other types of groups are created. So when extending a storage pool, the count of AvailableGroupsToCreate in several rows of statistics may change.

Note

Loads caused by accessing system views are more analytical in nature. Making frequent queries to them in large DBs will consume a lot of system resources. The recommended load is no more than 1-2 RPS.

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.

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>
   ```

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.

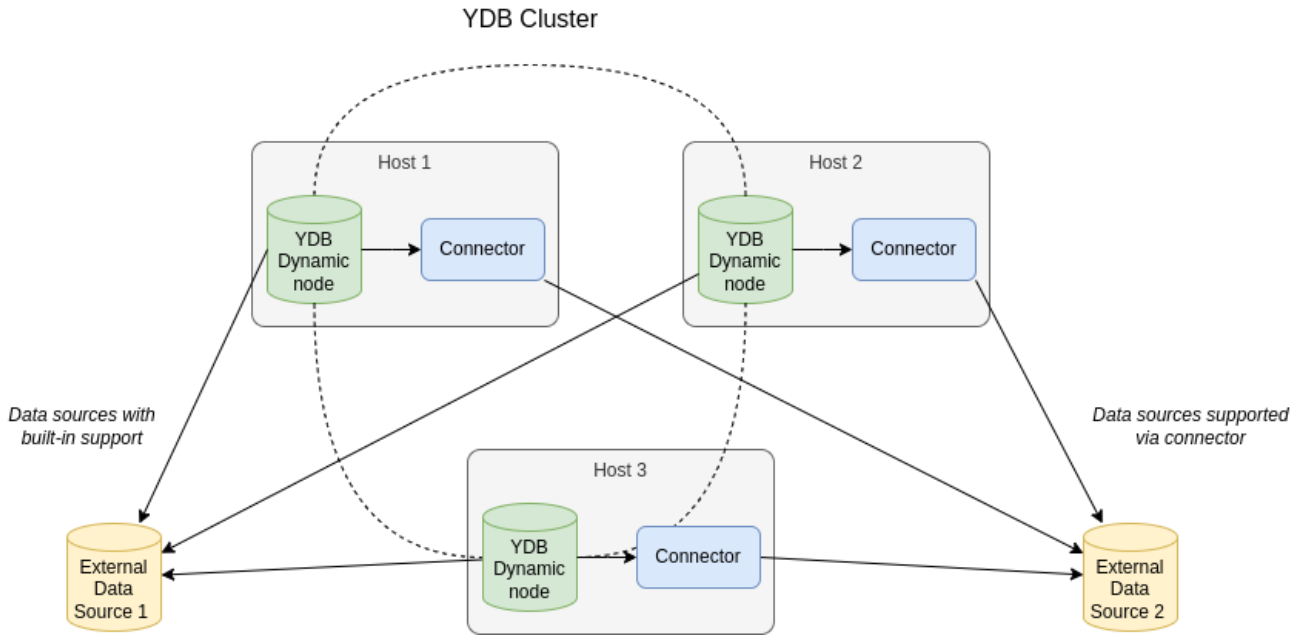
The YDB cluster and external data sources in a production installation should be deployed on different physical or virtual servers, including clouds. If access to a specific source requires a connector, it should be deployed on the same servers as the dynamic nodes of YDB. In other words, each `ydbd` 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.

Overview of cluster disk subsystem management

Managing a cluster's disk subsystem includes the following actions:

- Editing the cluster configuration:
  - Expanding a cluster
  - Adding storage groups
  - Moving a State Storage
  - Moving a static group
- Maintenance:
  - Safe restart and shutdown of nodes
  - Enabling/disabling Scrubbing
  - Working with SelfHeal
  - Decommissioning a cluster part
  - Moving VDisks
- Troubleshooting:
  - Staying within the failure model
  - Disk load balancing
  - Freeing up space on physical devices

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 built-in 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, to the configuration, the description of the added nodes (in the `hosts` section) and disks used by them (in the `host_configs` section).
   - 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 built-in 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 ItemConfigGenerationExpected#
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 built-in 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.

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.

Moving a State Storage

To decommission a YDB cluster host that accommodates a part of a State Storage, you need to move the group to another host.

> **Warning**
>
> The YDB cluster might become unavailable as a result of an invalid sequence of actions or a configuration error.

As an example, let's take 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
  ...
...
```

The static node of the cluster that serves a part of State Storage is set up and running on the host with `node_id:1`. Suppose that you want to decommission this host.

To replace `node_id:1`, we added to the cluster a new host with `node_id:10` and deployed a static node in it.

To move State Storage from the `node_id:1` host to the `node_id:10` host:

1. Stop the cluster's static nodes on the hosts with `node_id:1` and `node_id:10`.

   > **Note**
   >
   > YDB cluster is fault tolerant. Temporarily shutting down a node doesn't affect the cluster availability. For details, see YDB Cluster Topology.

2. In the `config.yaml` configuration file, change the `node` host list, replacing the ID of the removed host by the ID of the added host:

   ```
   domains_config:
   ...
     state_storage:
     - ring:
         node: [2, 3, 4, 5, 6, 7, 8, 9, 10]
         nto_select: 9
       ssid: 1
   ...
   ```

3. Update the `config.yaml` configuration files for all the cluster nodes, including dynamic nodes.
4. Use the rolling-restart procedure to restart all the cluster nodes (including dynamic nodes but excluding static nodes on the hosts with `node_id:1` and `node_id:10`). Please note that there must be at least a 15 second delay between host restarts.
5. Stop static cluster nodes on the hosts with `node_id:1` and `node_id:10`.

Moving a static group

To decommission a YDB cluster host that accommodates a part of a static group, you need to move the group to another host.

> **Warning**
>
> The YDB cluster might become unavailable as a result of an invalid sequence of actions or a configuration error.

As an example, let's take a YDB cluster where you set up and launched a static node on a host with `node_id:1`. This node serves a part of the static group.

Fragment of the static group configuration:

```
...
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 to the cluster a new host with `node_id:10` and deployed a static node in it.

To move a part of the static group from the `node_id:1` host to the `node_id:10` host:

1. Stop the static cluster node on the host with `node_id:1`.

   > **Note**
   >
   > YDB cluster is fault tolerant. Temporarily shutting down a node doesn't affect the cluster availability. For details, see YDB Cluster Topology.

2. In the `config.yaml` configuration file, change `node_id`, replacing the ID of the removed host by the ID of the added host:

   ```
   ...
   blob_storage_config:
     ...
     service_set:
       ...
       groups:
         ...
         rings:
           ...
           fail_domains:
           - vdisk_locations:
             - node_id: 10
               path: /dev/vda
               pdisk_category: SSD
           ...
         ...
       ...
     ...
   ...
   ```

   Edit the `path` and `pdisk_category` for the disk if these parameters are different on the host with `node_id: 10`.

3. Update the `config.yaml` configuration files for all the cluster nodes, including dynamic nodes.
4. Use the rolling-restart procedure to restart all the static cluster nodes.

5. Go to the Embedded UI monitoring page and make sure that the VDisk of the static group is visible on the target physical disk and its replication is in progress. For details, see Monitoring static groups.
6. Use the rolling-restart procedure to restart all the dynamic cluster nodes.

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
```

Working with SelfHeal

While a clusters are running, entire nodes or individual block devices that YDB runs on can fail.

SelfHeal ensures a cluster's continuous performance and fault tolerance if malfunctioning nodes or devices cannot be repaired quickly.

SelfHeal can:

- Detect faulty system elements.
- Transfer faulty elements carefully without data loss and disintegration of storage groups.

SelfHeal is enabled by default.

YDB component responsible for SelfHeal is called "Sentinel".

Enabling and disabling SelfHeal

You can enable and disable SelfHeal using YDB DSTool.

To enable SelfHeal, run the command:

```
ydb-dstool -e <bs_endpoint> cluster set --enable-self-heal
```

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 the initial settings, click **Create**. If you want to update the current settings, click ✎.

You can use the following settings:

Parameter	Description
Status	Enabling and disabling SelfHeal in CMS.
Dry run	Enables/disables the mode in which the CMS doesn't change the BSC setting.
Config update interval (sec.)	BSC configuration update interval.
Retry interval (sec.)	Interval of configuration update attempts.
State update interval (sec.)	PDisk state update interval.
Timeout (sec.)	PDisk state update timeout.
Change status retries	Number of retries to change the PDisk status for BSC (`ACTIVE`, `FAULTY`, `BROKEN`, and so on).
Change status retry interval (sec.)	Delay between retries to update the PDisk status in BSC. CMS monitors the status of the disk with the interval **State update inverval**. If the disk remains in one **Status update interval** state during several cycles, the CMS changes its status to BSC. Next are the settings for the number of update cycles after which the CMS changes the disk status. If the disk state is `Normal`, the disk status changes to `ACTIVE`. In other states, the disk switches to `FAULTY`. The `0` value disables status changes for the state (by default, this is set for `Unknown`). For example, with the default settings, if the CMS detects the `Initial` disk state for five `Status update interval` cycles which are 60 seconds each, the disk status changes to `FAULTY`.
Default state limit	For states with no setting specified, this value can be used by default. This value is also used for unknown PDisk states that don't have any settings. It's used if no value is set for states such as `Initial`, `InitialFormatRead`, `InitialSysLogRead`, `InitialCommonLogRead`, and `Normal`.
Initial	PDisk starts initializing. Transition to `FAULTY`.
InitialFormatRead	PDisk is reading its format. Transition to `FAULTY`.
InitialFormatReadError	PDisk received an error when reading its format. Transition to `FAULTY`.
InitialSysLogRead	PDisk is reading the system log. Transition to `FAULTY`.

InitialSysLogReadError	PDisk received an error when reading the system log. Transition to `FAULTY`.
InitialSysLogParseError	PDisk received an error when parsing and checking the consistency of the system log. Transition to `FAULTY`.
InitialCommonLogRead	PDisk is reading the common VDisk log. Transition to `FAULTY`.
InitialCommonLogReadError	PDisk received an error when reading the common VDisk log. Transition to `FAULTY`.
InitialCommonLogParseError	PDisk received an error when parsing and checking the consistency of the common log. Transition to `FAULTY`.
CommonLoggerInitError	PDisk received an error when initializing internal structures to be logged to the common log. Transition to `FAULTY`.
Normal	PDisk completed initialization and is running normally. Transition to `ACTIVE` will occur after a specified number of cycles (for example, if the disk is `Normal` for 5 minutes, it switches to `ACTIVE`).
OpenFileError	PDisk received an error when opening a disk file. Transition to `FAULTY`.
Missing	The node responds, but this PDisk is missing from its list. Transition to `FAULTY`.
Timeout	The node didn't respond within the specified timeout. Transition to `FAULTY`.
NodeDisconnected	The node has disconnected. Transition to `FAULTY`.
Stopped	PDisk has been stopped. Transition to `FAULTY`.
Unknown	Unexpected response, for example, `TEvUndelivered` to the state request. Transition to `FAULTY`.

Working with donor disks

The donor disk is the previous VDisk after the data transfer, which continues to store its data and only responds to read requests from the new VDisk. When data is transfered with donor disks enabled, previous VDisks continue to function until the data is fully moved to the new disks. To prevent data loss when moving a VDisk, enable donor disks:

```
ydb-dstool -e <bs_endpoint> cluster set --enable-donor-mode
```

To disable donor disks, run the command:

```
ydb-dstool -e <bs_endpoint> cluster set --disable-donor-mode
```

Decommissioning a cluster part

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 decommissioning

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` so that no data can be moved inside the DC-1.
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]"
```

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> GroupGeneration:
<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 }
}
```

Configuration overview

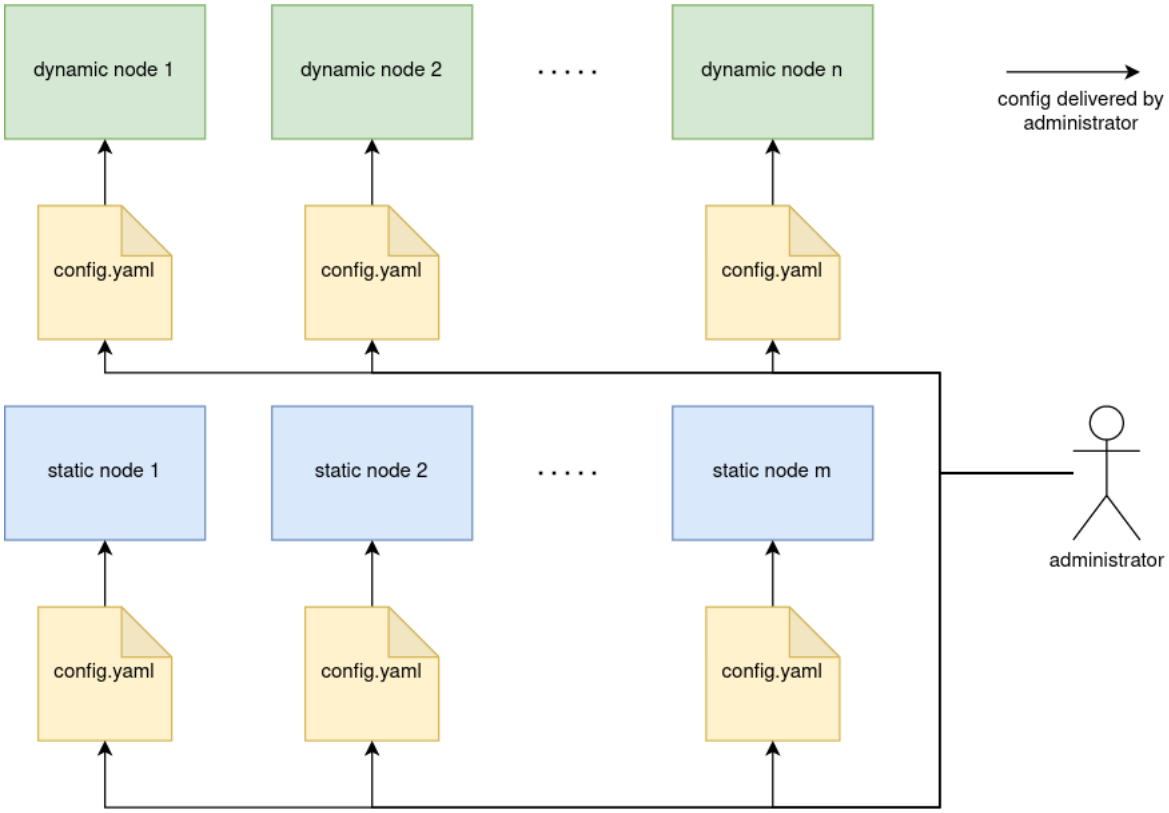
A YDB node requires configuration to run. There are two types of configurations:

- **Static** — a YAML file stored on the node's local disk.
- **Dynamic** — a YAML document stored in the YDB configuration repository.

Static nodes in the cluster use static configuration. Dynamic nodes can use static configuration, dynamic configuration, or a combination of both.

Static configuration

Static configuration is a YAML file stored on the cluster nodes. This file lists all the system settings. The path to the file is passed to the `ydbd` process at startup via a command-line parameter. Distributing the 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 section YDB cluster configuration. This configuration is **required** for running static nodes.

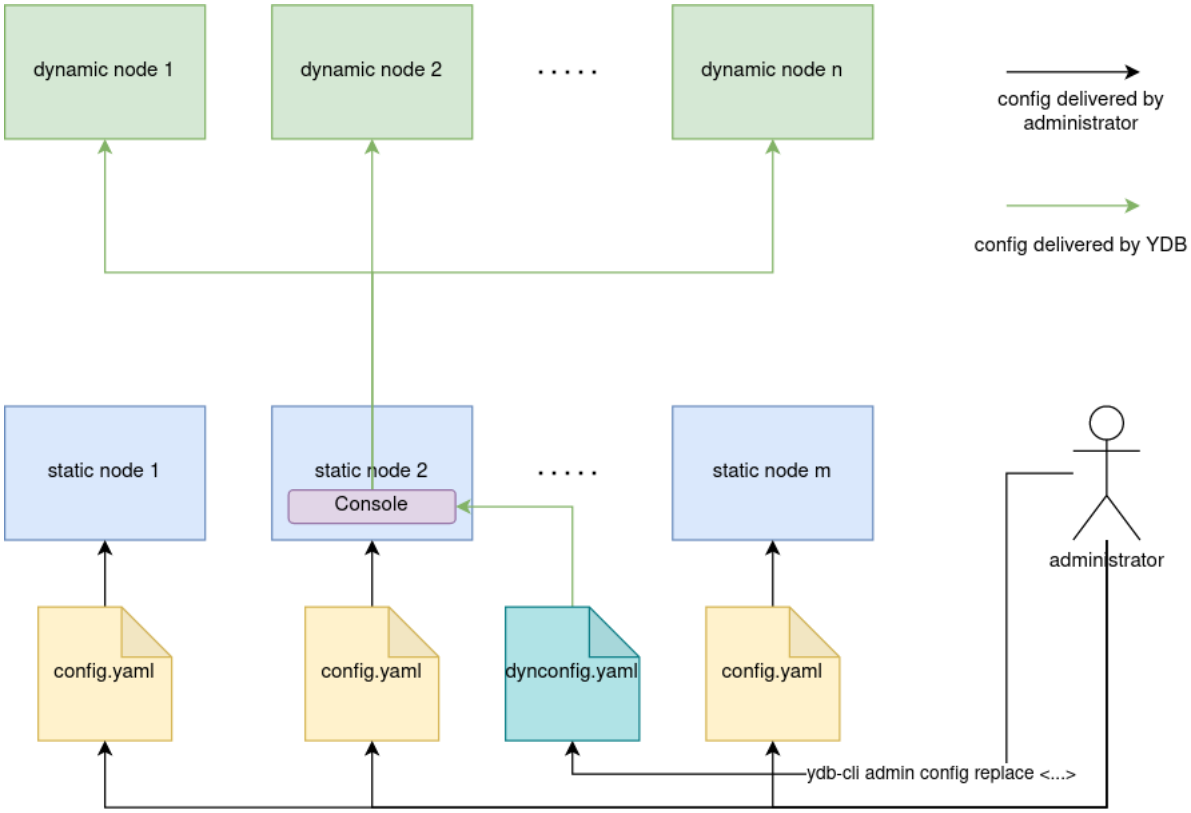


Basic usage scenario

1. Copy the 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 securely stored in the cluster in a Console tablet. Unlike static configuration, uploading it to the cluster is sufficient, as YDB will handle its distribution and maintenance in a consistent state. Dynamic configuration, using selectors, allows handling complex scenarios while remaining within a single configuration file. A description of the dynamic configuration is provided in the section Dynamic cluster configuration.



Basic usage scenario

1. Copy the 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. Extend the configuration file to the dynamic configuration format.
6. Upload the resulting configuration to the cluster using `ydb admin config replace -f dynconfig.yaml`.

Dynamic cluster configuration

Dynamic configuration allows running dynamic nodes by configuring them centrally without manually distributing files across the nodes. YDB acts as a configuration management system, providing tools for reliable storage, versioning, and delivery of configurations, as well as a DSL (Domain Specific Language) for overriding parts of the configuration for specific groups of nodes. The configuration is a YAML document and is an extended version of the static configuration:

- The configuration description is moved to the `config` field
- The `metadata` field is added for validation and versioning
- The `allowed_labels` and `selector_config` fields are added for granular overrides of settings

This configuration is uploaded to the cluster, where it is reliably stored and delivered to each dynamic node upon startup. Certain settings are updated on the fly without restarting nodes. Using dynamic configuration, you can centrally solve the following tasks:

- Change logging levels for all or specific components across the entire cluster or for specific groups of nodes.
- Enable experimental features (feature flags) on specific databases.
- Change actor system settings on individual nodes or groups of nodes.

Preparing to use the dynamic configuration

The following tasks should be performed before using the dynamic configuration in the cluster:

1. Enable database node authentication and authorization.
2. Export the current settings from the CMS in YAML format using the following command if CMS-based configuration management has been used in the cluster:

   ```
   ./ydbd -s grpcs://<node1.ydb.tech>:2135 --ca-file ca.crt --token-file ydbd-token \
        admin console configs dump-yaml > dynconfig.yaml
   ```

   Before running the command shown above, obtain the authentication token using the `ydb auth get-token` command, as detailed in the cluster initial deployment procedure.
3. Prepare the initial dynamic configuration file:
   - If there are non-empty CMS settings exported in the previous step, adjust the YAML file with the exported CMS settings:
     - Add the `metadata` section based on the configuration example.
     - Add the `yaml_config_enabled: true` parameter to the `config` section.
   - If there are no previous CMS-based settings, use the minimal configuration example.
   - For clusters using TLS encryption for actor system interconnect, add the interconnect TLS settings to the `config` section.
4. Apply the dynamic configuration settings file to the cluster:

   ```
   # Apply the dynconfig.yaml on the cluster
   ydb admin config replace -f dynconfig.yaml
   ```

> **Note**
>
> The legacy configuration management via CMS will become unavailable after enabling dynamic configuration support on the YDB cluster.

Configuration examples

Example of a minimal dynamic configuration for a single-datacenter cluster:

```
# Configuration metadata.
# This field is managed by the server.
metadata:
  # Cluster name from the cluster_uuid parameter set during cluster installation, or "", if the parameter is not
set.
  cluster: ""
  # Configuration file identifier, always increments by 1 starting from 0.
  # Automatically increases when a new configuration is uploaded to the server.
  version: 0
# Main cluster configuration. All values here are applied by default unless overridden by selectors.
# Content is similar to the static cluster configuration.
config:
  # It must always be set to true when using YAML configuration.
  yaml_config_enabled: true
  # Actor system configuration, as by default, this section is used only by dynamic nodes.
  # Configuration is set specifically for them.
  actor_system_config:
    # Automatic configuration selection for the node based on type and available cores.
```

```
    use_auto_config: true
    # HYBRID || COMPUTE || STORAGE — node type.
    node_type: COMPUTE
    # Number of cores.
    cpu_count: 14
allowed_labels: {}
selector_config: []
```

Detailed configuration parameters are described on the YDB cluster configuration page.

By default, the cluster configuration is assigned version 1. When applying a new configuration, the system compares the uploaded configuration's version with the value specified in the YAML file. If the versions match, the current version number is automatically incremented by one.

Below is a more comprehensive example of a dynamic configuration that defines typical global parameters as well as parameters specific to a particular database:

```
---
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 the dynamic configuration

```
# Fetch the cluster configuration
ydb admin config fetch > dynconfig.yaml
# Edit using any text editor
vim dynconfig.yaml
# Apply the configuration file dynconfig.yaml to the cluster
ydb admin config replace -f dynconfig.yaml
```

Additional configuration options are described on the selectors and temporary configuration pages.
All commands for working with configuration are described in the Fetch the cluster configuration section.

Operation mechanism

Configuration update from the administrator's perspective

1. The configuration file is uploaded by the user using a grpc call or YDB CLI to the cluster.
2. The file is checked for validity, basic constraints, version correctness, cluster name correctness, and the correctness of the configurations obtained after DSL transformation are verified.
3. The configuration version in the file is incremented by one.
4. The file is reliably stored in the cluster using the Console tablet.
5. File updates are distributed across the cluster nodes.

Configuration update from the cluster node's perspective

1. Each node requests the entire configuration at startup.
2. Upon receiving the configuration, the node generates the 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 updates, the local service receives it and transforms it for the node's labels.
5. All local services subscribed to updates receive the updated configuration.

Steps 1 and 2 are performed only for dynamic cluster nodes.

Configuration versioning

This mechanism prevents concurrent configuration modifications and makes updates idempotent. When a modification request is received, the server compares the version of the received modification with the stored one. If the version is one less, the configurations are compared: if they are identical, it means the user is attempting to upload the configuration again, the user receives OK, and the cluster configuration is not updated. If the version matches the current one on the cluster, the configuration is replaced with the new one, and the version field is incremented by one. In all other cases, the user receives an error.

Dynamically updated settings

Some system settings are updated without restarting nodes. To change them, upload a new configuration and wait for it to propagate across the cluster.

List of dynamically updated 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 validation delays of several seconds, as YDB needs to check the validity of each possible final configuration. The number of values for a single label has much less impact.
- Using large files (more than 500KiB for a cluster with 1000 nodes) can lead to increased network traffic in the cluster when updating the configuration. The traffic volume is directly proportional to the number of nodes and the configuration size.

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 described.

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
described in the section below
      # in this case, !inherit allows managing the actor_system_config.use_auto_config parameter for the entire
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:

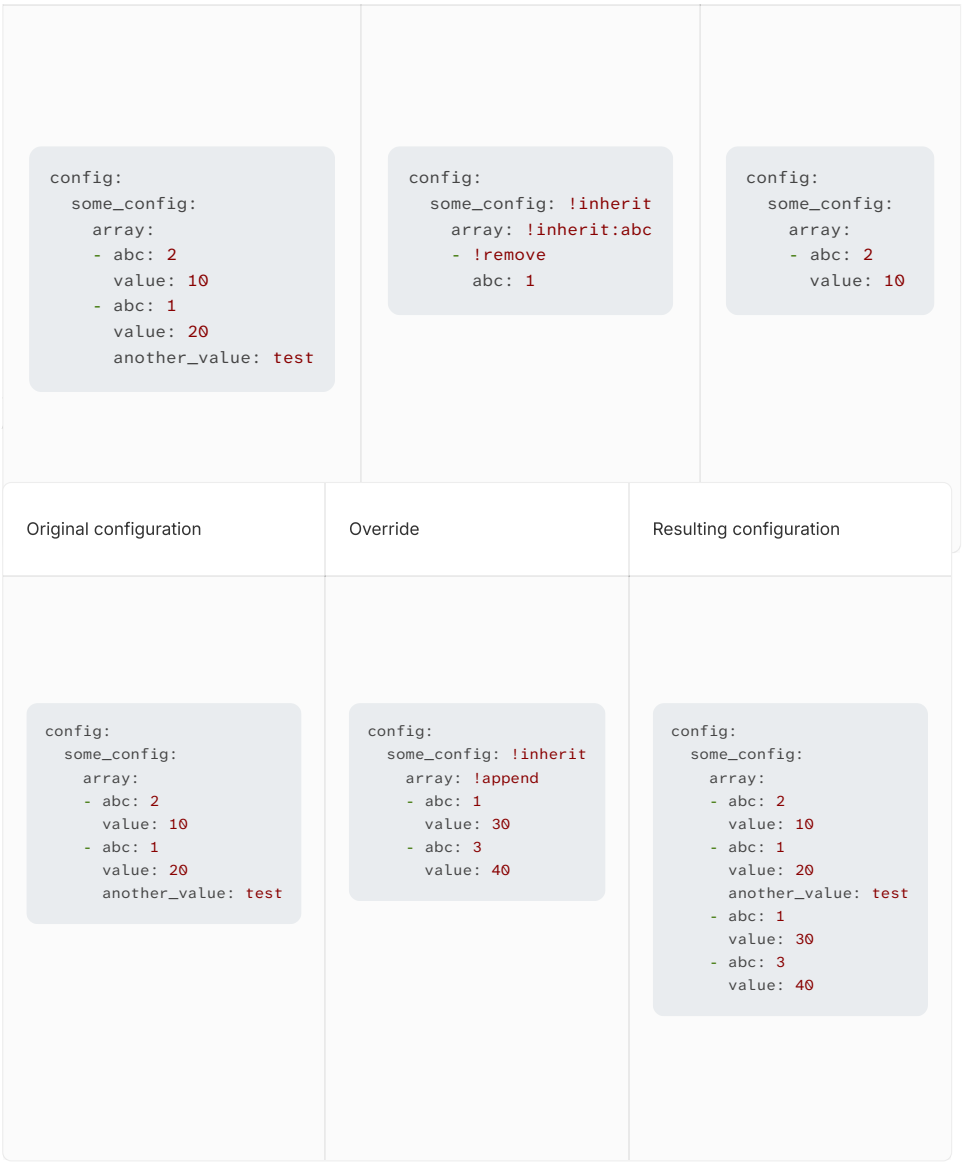
Original configuration	Override	Resulting configuration

`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



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 those
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
described 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
```

Volatile configurations

Volatile configurations are a special type of configuration that complements dynamic configurations while being non-persistent. These configurations are discarded when the Console tablet is moved or restarted, as well as when the main configuration is updated.

Primary use cases:

- Temporarily changing configuration for debugging or testing
- Trial activation of potentially dangerous settings. In the event of a cluster crash or restart, these settings will be automatically disabled

These configurations are added at the end of the selectors set, and the syntax for their description is identical to the selector syntax.

```
# Retrieve all volatile configurations uploaded to the cluster
ydb admin volatile-config fetch --all --output-directory <dir>
# Retrieve the volatile configuration with id=1
ydb admin volatile-config fetch --id 1
# Apply the volatile configuration volatile.yaml to the cluster
ydb admin volatile-config add -f volatile.yaml
# Delete volatile configurations with id=1 and id=3 on the cluster
ydb admin volatile-config drop --id 1 --id 3
# Delete all volatile configurations on the cluster
ydb admin volatile-config drop --all
```

Example of working with volatile configuration

Temporarily enabling logging settings for the `blobstorage` component to `DEBUG` on the node `host1.example.com`:

```
# Request current metadata to form a correct header for the volatile configuration
$ 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
# ...
# Delete the configuration
$ ydb admin volatile-config drop --id 2
```

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-linux-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-certificates.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-
   certificates.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.

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.
- Alternatively, YDB provides PostgreSQL-compatible API. It is intended to simplify migrating existing applications that have outgrown PostgreSQL. However, it is also useful for exotic programming languages that have a PostgreSQL client library but don't have a YDB SDK. Refer to PostgreSQL compatibility documentation to check if its completeness will suit your needs.
- 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 or a PostgreSQL driver depending on the route you have chosen above.

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
- Embedded UI
- Any SQL IDE that supports JDBC
- psql or pgAdmin for the PostgreSQL-compatible route.

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.

For PostgreSQL-compatibility route

- Learn how to connect PostgreSQL driver with YDB cluster.
- The rest should be similar to using vanilla PostgreSQL. Use your experience with it or refer to any favorite resources. However, refer to the list of functions and statements to adjust your expectations.

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.

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

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
- Node.js
- Python

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 | Node.js | PHP | Python

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 | Node.js | PHP | Python

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 | Node.js | PHP | Python

Retrieving data

Retrieve data using a `SELECT` statement in YQL. Handle the retrieved data selection in the app.

C++ | C# (.NET) | Go | Java | Node.js | PPHP | Python

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 | Node.js | PHP | Python

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 | Node.js | 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 | Node.js | PHP | Python

Error handling

For more information about error handling, see Error handling in the API.

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

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 it's 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`.

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.

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 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 `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

Alert

The functionality of vector indexes is available in the test mode in main. This functionality will be fully available in version 25.1.

The following features are not supported:

- modifying rows in tables with vector indexes
- building an index for vectors with bit quantization

These limitations may be removed in future versions.

Vector indexes are specialized data structures that enable efficient vector search in multidimensional spaces. Unlike secondary indexes, which 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.

Characteristics of Vector Indexes

Vector indexes in YDB address the nearest neighbor search problem using similarity or distance functions. Several distance/similarity functions are supported: "inner_product", "cosine" (similarity) and "cosine", "euclidean", "manhattan" (distance).

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)

Internally, a vector index consists of hidden index tables named `indexImpl*Table`. In selection queries using the vector index, the index tables will appear in query statistics.

Types of Vector Indexes

A vector index can be **covering**, meaning it includes additional columns to enable reading from the index without accessing the main table.

Alternatively, it can be **prefixed**, allowing for additional columns to be used for quick filtering during reading.

Below are examples of creating vector indexes of different types.

Basic Vector Index

Global vector index on the `embedding` column:

```
ALTER TABLE my_table
  ADD INDEX my_index
  GLOBAL USING vector_kmeans_tree
  ON (embedding)
  WITH (distance=cosine, vector_type="uint8", vector_dimension=512, levels=2, clusters=128);
```

Vector Index with Covering Columns

A covering vector index, including an additional column `data` to avoid reading from the main table during a search:

```
ALTER TABLE my_table
  ADD INDEX my_index
  GLOBAL USING vector_kmeans_tree
```

```
  ON (embedding) COVER (data)
  WITH (distance=cosine, vector_type="uint8", vector_dimension=512, levels=2, clusters=128);
```

Prefixed Vector Index

A prefixed vector index, allowing filtering by the prefix column `user` during vector search:

```
ALTER TABLE my_table
  ADD INDEX my_index
  GLOBAL USING vector_kmeans_tree
  ON (user, embedding)
  WITH (distance=cosine, vector_type="uint8", vector_dimension=512, levels=2, clusters=128);
```

Prefixed Vector Index with Covering Columns

A prefixed vector index with covering columns:

```
ALTER TABLE my_table
  ADD INDEX my_index
  GLOBAL USING vector_kmeans_tree
  ON (user, embedding) COVER (data)
  WITH (distance=cosine, vector_type="uint8", vector_dimension=512, levels=2, clusters=128);
```

Creating Vector Indexes

Vector indexes can be created:

- during table creation using the YQL operator CREATE TABLE;
- added to an existing table using the YQL operator ALTER TABLE.

Using Vector Indexes

Queries to vector indexes are executed using the `VIEW` syntax in YQL. For prefixed indexes, specify the prefix columns in the `WHERE` clause:

```
DECLARE $query_vector AS List<Uint8>;

SELECT user, data
FROM my_table VIEW my_index
ORDER BY Knn::CosineSimilarity(embedding, $query_vector) DESC
LIMIT 10;
```

For more details on executing `SELECT` queries using vector indexes, see the section VIEW VECTOR INDEX.

Note

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 statistics. In particular, ensure there is no full scan of the main table.

Query plan optimization

Before executing a query, it is essential to analyze its execution plan to detect and eliminate the reasons for possible excessive cluster resource consumption or abnormally high execution time. This article will discuss specific examples of query plan analysis.

Let's consider the following query that searches for episodes by title:

```
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 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:

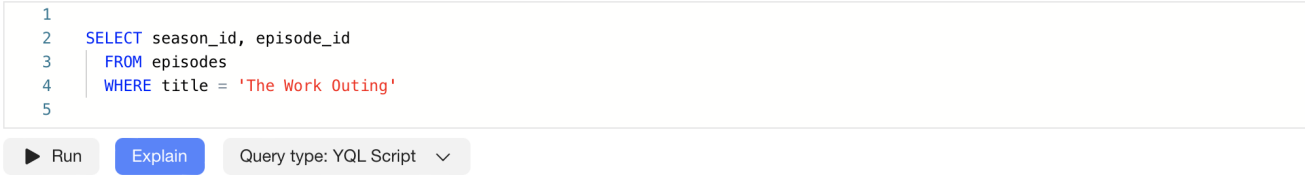
```
ydb -p <profile_name> table query explain \
  -q "SELECT season_id, episode_id
  FROM episodes
  WHERE title = 'The Work Outing'"
```

Result:

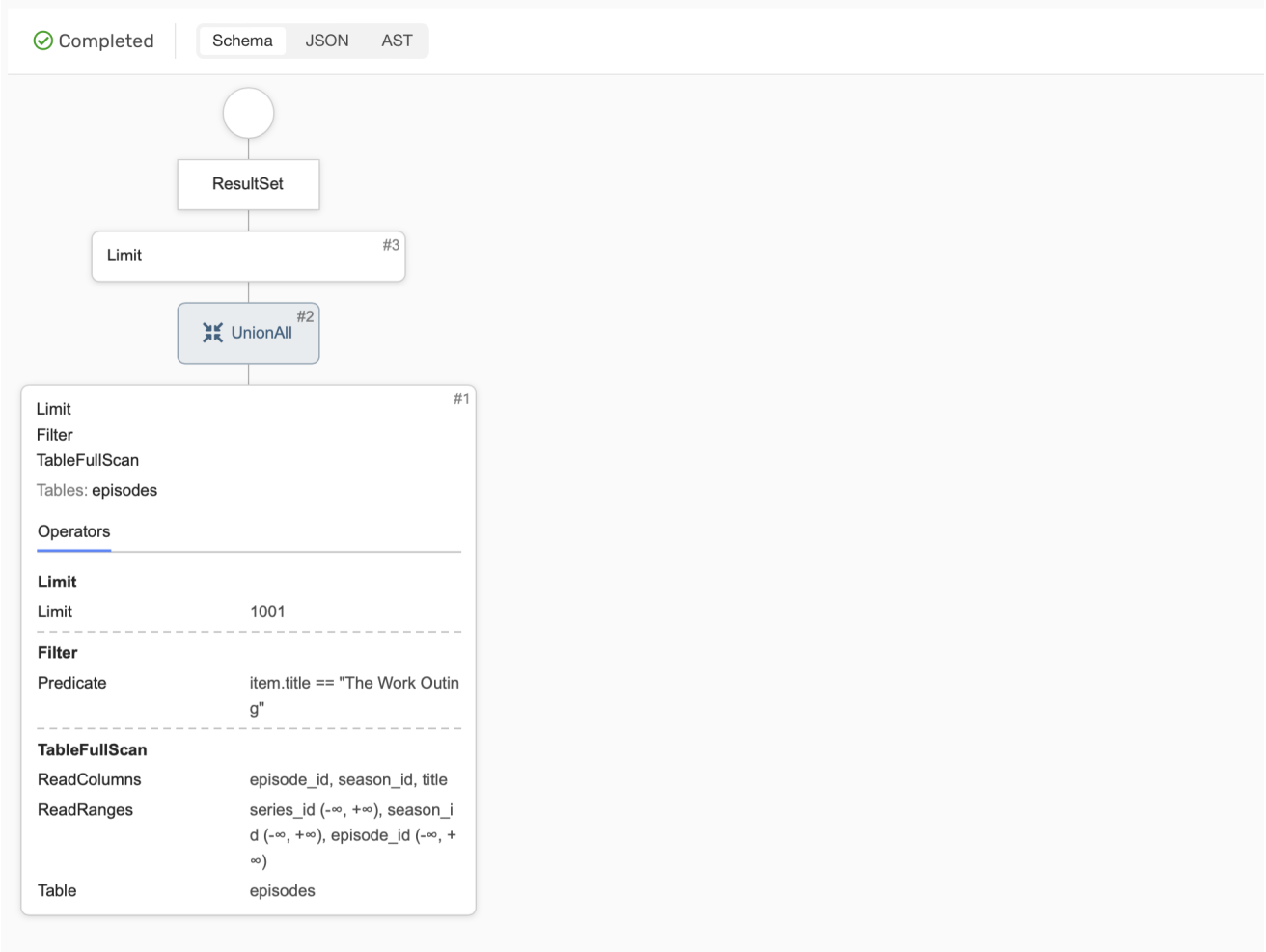
```
Query Plan:
ResultSet
└──Limit (Limit: 1001)
   └──<UnionAll>
      └──Limit (Limit: 1001)
      └──Filter (Predicate: item.title == "The Work Outing")
      └──TableFullScan (ReadRanges: ["series_id (-∞, +∞)","season_id (-∞, +∞)","episode_id (-∞, +∞)"],
ReadColumns: ["episode_id","season_id","title"], Table: episodes)
         Tables: ["episodes"]
```

Embedded UI

You can also build a query plan via Embedded 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:
```

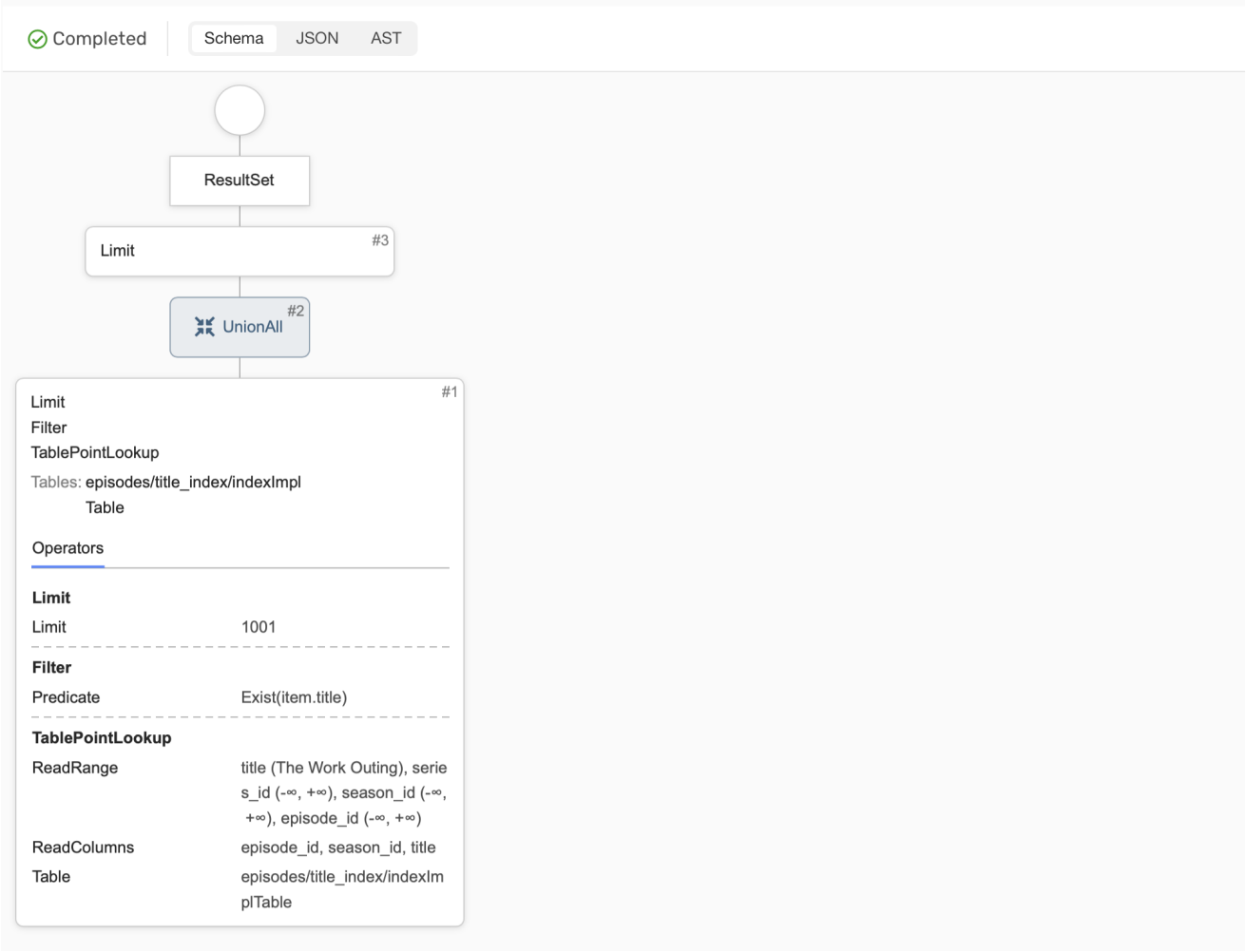
```
episodes/title_index/indexImplTable)
        Tables: ["episodes/title_index/indexImplTable"]
```

Embedded UI

```
1
2  SELECT season_id, episode_id
3    FROM episodes VIEW title_index
4    WHERE title = 'The Work Outing'
5
```

Run | Explain | Query type: YQL Script

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.

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

```
import (
  "context"

  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*300) // client and by default operation timeout
defer cancel()
ctx = ydb.WithOperationTimeout(ctx, time.Millisecond*400)     // operation timeout override
ctx = ydb.WithOperationCancelAfter(ctx, time.Millisecond*300) // cancel after timeout
tx := table.TxControl(table.BeginTx(table.WithSerializableReadWrite()), table.CommitTx())
_, res, err := s.Execute(ctx, tx, query, table.NewQueryParameters())
}
```

Database system views

To obtain service information about the state of the database, you can access system views. They are accessible 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 the frequency of 1 request per second is not recommended.

Partitions

The following system view stores detailed information about partitions of DB tables:

- `partition_stats`: Contains information about instant metrics and cumulative operation counters. Instant metrics are, for example, CPU load or count of in-flight transactions. Cumulative counters, for example, count the total number of rows read.

The system view is designed to detect various irregularities in the load on a table partition or show the size of table partition data.

Instant metrics (`NodeID`, `AccessTime`, `CPUCores`, etc.) contain instantaneous values.
Cumulative metrics (`RowReads`, `RowUpdate`, `LockAcquired`, etc.) store accumulated values since the last launch (`StartTime`) of the tablet serving the partition.

Table structure:

Column	Description	Data type	Instant/Cumulative
`OwnerId`	ID of the SchemeShard table. Key: `0`.	`Uint64`	Instant
`PathId`	ID of the SchemeShard path. Key: `1`.	`Uint64`	Instant
`PartIdx`	Partition sequence number. Key: `2`.	`Uint64`	Instant
`FollowerId`	ID of the partition tablet follower. A value of 0 means the leader. Key: `3`.	`Uint32`	Instant
`DataSize`	Approximate partition size in bytes.	`Uint64`	Instant
`RowCount`	Approximate number of rows.	`Uint64`	Instant
`IndexSize`	Partition index size in bytes.	`Uint64`	Instant
`CPUCores`	Instantaneous value of the load on the partition (the share of the CPU core time spent by the actor of the partition).	`Double`	Instant
`TabletId`	ID of the partition tablet.	`Uint64`	Instant
`Path`	Full path to the table.	`Utf8`	Instant
`NodeId`	ID of the partition node.	`Uint32`	Instant
`StartTime`	Last time of the launch of the partition tablet.	`Timestamp`	Instant
`AccessTime`	Last time of reading from the partition.	`Timestamp`	Instant
`UpdateTime`	Last time of writing to the partition.	`Timestamp`	Instant
`RowReads`	Number of point reads.	`Uint64`	Cumulative
`RowUpdates`	Number of rows written.	`Uint64`	Cumulative
`RowDeletes`	Number of rows deleted.	`Uint64`	Cumulative
`RangeReads`	Number of range reads.	`Uint64`	Cumulative
`RangeReadRows`	Number of rows read in ranges.	`Uint64`	Cumulative

`InFlightTxCount`	Number of in-flight 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 overload.	`Uint64`	Cumulative
`TxRejectedByOutOfStorage`	Number of transactions cancelled due to lack of storage space.	`Uint64`	Cumulative
`LastTtlRunTime`	Launch time of the last TTL erasure procedure	`Timestamp`	Instant
`LastTtlRowsProcessed`	Number of rows checked during the last TTL erasure procedure	`Uint64`	Instant
`LastTtlRowsErased`	Number of rows deleted during the last TTL erasure procedure	`Uint64`	Instant
`LocksAcquired`	Number of locks acquired.	`Uint64`	Cumulative
`LocksWholeShard`	The number of "whole shard" locks taken.	`Uint64`	Cumulative
`LocksBroken`	Number of broken locks.	`Uint64`	Cumulative

Example queries

Top 5 of most loaded partitions among all DB tables:

```
SELECT
    Path,
    PartIdx,
    CPUCores
FROM `.sys/partition_stats`
ORDER BY CPUCores DESC
LIMIT 5
```

List of DB tables with in-flight sizes and loads:

```
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 DB tables with the largest 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 the user queries.

Maximum total execution time:

- `top_queries_by_duration_one_minute`: data is split into one-minute intervals, contains the history for the last 6 hours;
- `top_queries_by_duration_one_hour`: data is split into one-hour intervals, contains the history for the last 2 weeks.

Maximum number of bytes read from the table:

- `top_queries_by_read_bytes_one_minute`: data is split into one-minute intervals, contains the history for the last 6 hours;
- `top_queries_by_read_bytes_one_hour`: Data is split into one-hour intervals, contains the history for the last 2 weeks.

Maximum CPU time:

- `top_queries_by_cpu_time_one_minute`: Data is split into one-minute intervals, contains the history for the last 6 hours;
- `top_queries_by_cpu_time_one_hour`: Data is split into one-hour intervals, contains the history for the last 2 weeks.

Different runs of a query with the same text are deduplicated. 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 completed in that time interval.

Fields that provide information about the used CPU time (... `CPUTime`) are expressed in microseconds.

Query text limit is 10 KB.

All tables have the same structure:

Column	Description
`IntervalEnd`	The end of the minute or hour interval for which statistics are collected. Type: `Timestamp`. Key: `0`.

`Rank`	Rank of a top query. Type: `Uint32`. Key: `1`.
`QueryText`	Query text. Type: `Utf8`.
`Duration`	Total query execution time. Type: `Interval`.
`EndTime`	Query execution end time. Type: `Timestamp`.
`Type`	Query type (data, scan, or 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 used during query execution. Type: `Uint64`.
`UserSID`	User Security ID. Type: `String`.
`ParametersSize`	Size of query parameters in bytes. Type: `Uint64`.
`CompileDuration`	Duration of query compilation. Type: `Interval`.
`FromQueryCache`	Shows whether the cache of prepared queries was used. Type: `Bool`.
`CPUTime`	Total CPU time used to execute the query (microseconds). Type: `Uint64`.
`ShardCount`	Number of shards used during query execution. Type: `Uint64`.
`SumShardCPUTime`	Total CPU time used in shards. Type: `Uint64`.
`MinShardCPUTime`	Minimum CPU time used in shards. Type: `Uint64`.
`MaxShardCPUTime`	Maximum CPU time used in shards. Type: `Uint64`.
`ComputeNodesCount`	Number of compute nodes used during query execution. Type: `Uint64`.
`SumComputeCPUTime`	Total CPU time used in compute nodes. Type: `Uint64`.
`MinComputeCPUTime`	Minimum CPU time used in compute nodes. Type: `Uint64`.

`MaxComputeCPUTime`	Maximum CPU time used in compute nodes. Type: `Uint64`.
`CompileCPUTime`	CPU time used to compile a query. Type: `Uint64`.
`ProcessCPUTime`	CPU time used for overall query handling. Type: `Uint64`.

Example queries

Top queries by execution time. The query is made to 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 made to 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
```

Query details

The following system view stores detailed information about queries:

- `query_metrics_one_minute`: Data is split into one-minute intervals, contains up to 256 queries for the last 6 hours.

Each table row contains information about a set of queries with identical text that were made during one minute. The table fields provide the minimum, maximum, and total values for each query metric tracked. Within the interval, queries are sorted in descending order of the total CPU time used.

Restrictions:

- Query text limit is 10 KB.
- Statistics may be incomplete if the database is under heavy load.

Table structure:

Column	Description
`IntervalEnd`	The end of the minute interval for which statistics are collected. Type: `Timestamp`. Key: `0`.
`Rank`	Query rank within an interval (by the `SumCPUTime` field). Type: `Uint32`. Key: `1`.
`QueryText`	Query text. Type: `Utf8`.
`Count`	Number of query runs. Type: `Uint64`.
`SumDuration`	Total duration of queries. Type: `Interval`.

`MinDuration`	Minimum query duration. Type: `Interval`.
`MaxDuration`	Maximum query duration. Type: `Interval`.
`SumCPUTime`	Total CPU time used. Type: `Uint64`.
`MinCPUTime`	Minimum CPU time used. Type: `Uint64`.
`MaxCPUTime`	Maximum CPU time used. 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`.

Example queries

Top 10 queries for the last 6 hours by the total number of rows updated per minute:

```
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
```

History of overloaded partitions

The following system views (tables) store the history of points in time when the load on individual DB table partitions was high:

- `top_partitions_one_minute`: The data is split into one-minute intervals, contains the history for the last 6 hours.
- `top_partitions_one_hour`: The data is split into one-hour intervals, contains the history for the last 2 weeks.

These views contain partitions with peak loads of more than 70% (`CPUCores` > 0.7). Partitions within a single interval are ranked by peak load value.

The keys of the views are:

- `IntervalEnd` - the moment when the interval is closed;
- `Rank` - the rank of the partition according to the peak load of `CPUCores` in this interval.

For example, if a table has 10 partitions than `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 a `Rank` from 1 to 10 and the same `IntervalEnd` `"20.12.2024 11:00"`.

All tables have the same structure:

Column	Description
`IntervalEnd`	The end of the minute or hour interval for which statistics are collected. Type: `Timestamp`. Key: `0`.
`Rank`	Partition rank within an interval (by `CPUCores`). Type: `Uint32`. Key: `1`.
`TabletId`	ID of the tablet serving the partition. Type: `Uint64`.
`FollowerId`	ID of the partition tablet follower.A value of 0 means the leader. Type: `Uint32`
`Path`	Full path to the table. Type: `Utf8`.
`PeakTime`	Peak time within an interval. Type: `Timestamp`.
`CPUCores`	Peak load per partition (share of the CPU core time spent by the actor of the partition). Type: `Double`.
`NodeId`	ID of the node where the partition was located during the peak load. Type: `Uint32`.
`DataSize`	Approximate partition size, in bytes, during the peak load. Type: `Uint64`.
`RowCount`	Approximate row count during the peak load. Type: `Uint64`.
`IndexSize`	Partition index size per tablet during the peak load. Type: `Uint64`.
`InFlightTxCount`	The number of in-flight transactions during the peak load. Type: `Uint32`.

Example queries

The following query returns partitions with CPU usage of more than 70% in the specified interval, with tablet IDs and sizes as of the time when the percentage was exceeded. The query is made to the `.sys/top_partitions_one_minute` view:

```
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 returns partitions with CPU usage of over 90% in the specified interval, with tablet IDs and sizes as of the time when the percentage was exceeded. The query is made to the `.sys/top_partitions_one_hour` view:

```
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
```

History of partitions with broken locks

The following system views contain a history of moments with a non-zero number of broken locks `LocksBroken` in individual partitions of DB tables:

- `top_partitions_by_tli_one_minute`: The data is split into one-minute intervals, contains the history for the last 6 hours.
- `top_partitions_by_tli_one_hour`: The data is split into one-hour intervals, contains the history for the last 2 weeks.

The views provide 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 keys of the views are:

- `IntervalEnd` - the moment of interval closure;
- `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 the hourly interval `"20.12.2024 10:00-11:00"` will output 10 rows, sorted in descending order by `LocksBroken`. They will have `Rank` from 1 to 10 and the same `IntervalEnd` `"20.12.2024 11:00"`.

All tables have the same structure:

Column	Description
`IntervalEnd`	The end of the minute or hour interval for which statistics are collected. Type: `Timestamp`. Key: `0`.
`Rank`	Partition rank within an interval (by `CPUCores`). Type: `Uint32`. Key: `1`.
`TabletId`	ID of the tablet serving the partition. Type: `Uint64`.
`FollowerId`	ID of the partition tablet follower.A value of 0 means the leader. Type: `Uint32`
`Path`	Full path to the table. Type: `Utf8`.

`LocksAcquired`	Number of locks acquired "on a range of keys" in this interval. Type: `Uint64`.
`LocksWholeShard`	Number of locks acquired "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 during the peak load. Type: `Uint32`.
`DataSize`	Approximate partition size, in bytes, during the peak load. Type: `Uint64`.
`RowCount`	Approximate row count during the peak load. Type: `Uint64`.
`IndexSize`	Partition index size per tablet during the peak load. Type: `Uint64`.

Example queries

The following query returns partitions in the specified time interval, with tablet identifiers and the number of broken locks. The query is made to 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
```

Auth users, groups, permissions

Auth users

The `auth_users` view lists local YDB users. It does not include users authenticated through external systems such as LDAP.

This view can be fully accessed by administrators, while regular users can only view their own details.

Table structure:

Column	Description
`Sid`	SID of the user. Type: `Utf8`. Key: `0`.
`IsEnabled`	Indicates if login is allowed; used for explicit administrator block. Independent of `IsLockedOut`. Type: `Bool`.
`IsLockedOut`	Automatically locked out due to exceeding failed login attempts. Independent of `IsEnabled`. Type: `Bool`.
`CreatedAt`	Timestamp of user creation. Type: `Timestamp`.
`LastSuccessfulAttemptAt`	Timestamp of the last successful login attempt. Type: `Timestamp`.
`LastFailedAttemptAt`	Timestamp of the last failed login attempt. Type: `Timestamp`.
`FailedAttemptCount`	Number of failed login attempts. Type: `Uint32`.
`PasswordHash`	JSON string containing password hash, salt, and hash algorithm. Type: `Utf8`.

Auth groups

The `auth_groups` view lists access groups.

This view can be accessed only by administrators.

Table structure:

Column	Description
`Sid`	SID of the group. Type: `Utf8`. Key: `0`.

Auth group members

The `auth_group_members` view lists membership details within access groups.

This view can be accessed only by administrators.

Table structure:

Column	Description
`GroupSid`	SID of the group. Type: `Utf8`. Key: `0`.
`MemberSid`	SID of the group member. Type: `Utf8`. Key: `1`.

Auth permissions

The auth permissions views list assigned access rights.

Contains two views:

- `auth_permissions`: Directly assigned access rights.
- `auth_effective_permissions`: Effective access rights, accounting for inheritance.

A user can view an access object in the results if they have the `ydb.granular.describe_schema` permission on it.

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 YDB access right. Type: `Utf8`. Key: `2`.

Example queries

All the directly assigned permissions for the table located at the path `my_table`:

```
SELECT *
FROM `.sys/auth_permissions`
WHERE Path = "my_table"
```

All the effective permissions for the table located at the path `my_table`, including inherited permissions:

```
SELECT *
FROM `.sys/auth_effective_permissions`
WHERE Path = "my_table"
```

All permissions directly assigned to the user identified as `user3`:

```
SELECT *
FROM `.sys/auth_permissions`
WHERE Sid = "user3"
```

Auth owners

The `auth_owners` view lists details of access objects ownership.

A user can view an access object in the results if they have the `ydb.granular.describe_schema` permission on it.

Table structure:

Column	Description
`Path`	Path to the access object. Type: `Utf8`. Key: `0`.
`Sid`	SID of the access object owner. Type: `Utf8`.

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":[23779788941838502 58]}
...
```

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
```

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 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 Dave
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;
```

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).GetValueSync();

        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()-
>FormatLocalTime("%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 active
        // 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 instead,
    // 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 driver initialization:

```
public static async Task Run(
    string endpoint,
    string database,
    ICredentialsProvider credentialsProvider)
{
    var config = new DriverConfig(
        endpoint: endpoint,
        database: database,
        credentials: credentialsProvider
    );

    using var driver = new Driver(
        config: config
    );

    await driver.Initialize();
}
```

App code snippet for creating a session:

```
using var queryClient = new QueryService(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.

To create tables, use the `queryClient.Exec` method with a DDL (Data Definition Language) YQL query.

```
await queryClient.Exec(@"
    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)
    );
");
```

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 queryClient.Exec(@"
    UPSERT INTO series (series_id, title, release_date) VALUES
        ($id, $title, $release_date);
    ",
    new Dictionary<string, YdbValue>
    {
        { "$id", YdbValue.MakeUint64(1) },
        { "$title", YdbValue.MakeUtf8("NewTitle") },
        { "$release_date", YdbValue.MakeDate(DateTime.UtcNow) }
    }
);
```

Retrieving data

Retrieve data using a `SELECT` statement in YQL. Handle the retrieved data selection in the app.

To execute YQL queries, use the `queryClient.ReadRow` или `queryClient.ReadAllRows` method. The SDK lets you explicitly control the execution of transactions and configure the transaction execution mode using the `TxMode` enum. In the code snippet below, a transaction with the `NoTx` mode and an automatic commit after executing the request is used. The values of the request parameters are passed in the form of a dictionary name-value in the `parameters` argument.

```
var row = await queryClient.ReadRow(@"
        SELECT
            series_id,
            title,
            release_date
        FROM series
        WHERE series_id = $id;
    ",
    new Dictionary<string, YdbValue>
    {
        { "$id", YdbValue.MakeUint64(id) }
    }
);
```

Processing execution results

The result of query execution (resultset) consists of an organized set of rows. Example of processing the query execution result:

```
foreach (var row in resultSet.Rows)
{
    Console.WriteLine($"> Series, " +
        $"series_id: {(ulong)row["series_id"]}, " +
        $"title: {(string?)row["title"]}, " +
        $"release_date: {(DateTime?)row["release_date"]}");
}
```

Scan queries

Execute a scan query to produce a data stream. Streaming allows to read an unlimited number of rows and an unlimited amount of data.

```
await queryClient.Stream(
    $"SELECT title FROM seasons ORDER BY series_id, season_id;",
    async stream =>
    {
        await foreach (var part in stream)
        {
            foreach (var row in part.ResultSet!.Rows)
            {
                Console.WriteLine(row[0].GetOptionalUtf8());
```

```
            }
        }
    });
```

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 program).
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_date",
&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
}
```

Scan queries

Execute a scan query to produce a data stream. Streaming allows to read an unlimited number of rows and an unlimited amount of data.

> **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://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:

```
<auth_mode_var>="<auth_mode_value>" java -jar ydb-java-examples/query-example/target/ydb-query-example.jar
grpcs://<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
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.

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, params))
            .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 Node.js

This page provides a detailed description of the code for
an example application,
which is available as part of the YDB Node.js 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:

Using connectionString

```
const authService = getCredentialsFromEnv();
logger.debug('Driver initializing...');
const driver = new Driver({connectionString, authService});
const timeout = 10000;
if (!await driver.ready(timeout)) {
    logger.fatal(`Driver did not become ready within ${timeout}ms!`);
    process.exit(1);
}
```

Using endpoint and database

```
const authService = getCredentialsFromEnv();
logger.debug('Driver initializing...');
const driver = new Driver({endpoint, database, authService});
const timeout = 10000;
if (!await driver.ready(timeout)) {
    logger.fatal(`Driver did not become ready within ${timeout}ms!`);
    process.exit(1);
}
```

App code snippet for creating a session:

```
const result = await driver.queryClient.do({
    ...
    fn: async (session) => {
        ...
    }
});
```

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.

```
async function createTables(driver: Driver, logger: Logger) {
    logger.info('Dropping old tables and creating new ones...');
    await driver.queryClient.do({
        fn: async (session) => {

          try {
              await session.execute({
                  text: `
                      DROP TABLE ${SERIES_TABLE};
                      DROP TABLE ${EPISODES_TABLE};
                      DROP TABLE ${SEASONS_TABLE};`,
              });
          } catch (err) { // Ignore if tables are missing
```

```
            if (err instanceof SchemeError) throw err;
        }

        await session.execute({
            text: `
                CREATE TABLE ${SERIES_TABLE}
                (
                    series_id    UInt64,
                    title        Utf8,
                    series_info  Utf8,
                    release_date DATE,
                    PRIMARY KEY (series_id)
                );

                CREATE TABLE ${SEASONS_TABLE}
                (
                    series_id   UInt64,
                    season_id   UInt64,
                    title UTF8,
                    first_aired DATE,
                    last_aired DATE,
                    PRIMARY KEY (series_id, season_id)
                );

                CREATE TABLE ${EPISODES_TABLE}
                (
                    series_id  UInt64,
                    season_id  UInt64,
                    episode_id UInt64,
                    title      UTf8,
                    air_date   DATE,
                    PRIMARY KEY (series_id, season_id, episode_id),
                    INDEX      episodes_index GLOBAL ASYNC ON (air_date)
                );`,
        });
    },
});
}
```

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:

```
async function upsertSimple(driver: Driver, logger: Logger): Promise<void> {
    logger.info('Making an upsert...');
    await driver.queryClient.do({
        fn: async (session) => {
             await session.execute({
                text: `
                   UPSERT INTO ${EPISODES_TABLE} (series_id, season_id, episode_id, title)
                   VALUES (2, 6, 1, "TBD");`,
           })
        }
    });
    logger.info('Upsert completed.')
}
```

Retrieving data

Retrieve data using a `SELECT` statement in YQL. Handle the retrieved data selection in the app.

The `QuerySession.execute()` method is used to execute YQL queries.

Depending on the rowMode parameter, the data can be retrieved in javascript form or as YDB structures.

rowMode: RowType.Native

```
async function selectNativeSimple(driver: Driver, logger: Logger): Promise<void> {
    logger.info('Making a simple native select...');
    const result = await driver.queryClient.do({
        fn: async (session) => {
            const {resultSets} =
                await session.execute({
```

```
                    // rowMode: RowType.Native, // Result set columns and rows are returned as native JavaScript
values. This is the default behavior.
                    text: `
                        SELECT series_id,
                               title,
                               release_date
                        FROM ${SERIES_TABLE}
                        WHERE series_id = 1;`,
                });
            const {value: resultSet1} = await resultSets.next();
            const rows: any[][] = []
            for await (const row of resultSet1.rows) rows.push(row);
            return {cols: resultSet1.columns, rows};
        }
    });
    logger.info(`selectNativeSimple cols: ${JSON.stringify(result.cols, null, 2)}`);
    logger.info(`selectNativeSimple rows: ${JSON.stringify(result.rows, null, 2)}`);
}
```

rowMode: RowType.Ydb

```
async function selectTypedSimple(driver: Driver, logger: Logger): Promise<void> {
    logger.info('Making a simple typed select...');
    const result = await driver.queryClient.do({
        fn: async (session) => {
            const {resultSets} =
                await session.execute({
                    rowMode: RowType.Ydb, // enables typedRows() on result sets
                    text: `
                        SELECT series_id,
                               title,
                               release_date
                        FROM ${SERIES_TABLE}
                        WHERE series_id = 1;`,
                });
            const {value: resultSet1} = await resultSets.next();
            const rows: Series[] = [];
            // Note: resultSet1.rows will iterate YDB IValue structures
            for await (const row of resultSet1.typedRows(Series)) rows.push(row);
            return {cols: resultSet1.columns, rows};
        }
    });
    logger.info(`selectTypedSimple cols: ${JSON.stringify(result.cols, null, 2)}`);
    logger.info(`selectTypedSimple rows: ${JSON.stringify(result.rows, null, 2)}`);
}
```

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.

```
async function selectWithParameters(driver: Driver, data: ThreeIds[], logger: Logger): Promise<void> {

    await driver.queryClient.do({
        fn: async (session) => {
            for (const [seriesId, seasonId, episodeId] of data) {
                const episode = new Episode({seriesId, seasonId, episodeId, title: '', airDate: new Date()});
                const {resultSets, opFinished} = await session.execute({
                    parameters: {
                        '$seriesId': episode.getTypedValue('seriesId'),
                        '$seasonId': episode.getTypedValue('seasonId'),
                        '$episodeId': episode.getTypedValue('episodeId')
                    },
                    text: `
                        DECLARE $seriesId AS Uint64;
                        DECLARE $seasonId AS Uint64;
                        DECLARE $episodeId AS Uint64;

                        SELECT title,
                               air_date
                        FROM episodes
                        WHERE series_id = $seriesId
                          AND season_id = $seasonId
                          AND episode_id = $episodeId;`
                });
```

```
                const {value: resultSet} = await resultSets.next();
                const {value: row} = await resultSet.rows.next();
                await opFinished;
                logger.info(`Parametrized select query ${JSON.stringify(row, null, 2)}`);
            }
        }
    });
}
```

Scan queries

Execute a scan query to produce a data stream. Streaming allows to read an unlimited number of rows and an unlimited amount of data.

The `QuerySession.execute()` method is used to retrieve data in a streaming manner.

```
async function selectWithParametrs(driver: Driver, data: ThreeIds[], logger: Logger): Promise<void> {
    logger.info('Selecting with a parametrized query...');
    await driver.queryClient.do({
        fn: async (session) => {
            for (const [seriesId, seasonId, episodeId] of data) {

                const episode = new Episode({seriesId, seasonId, episodeId, title: '', airDate: new Date()});

                const {resultSets, opFinished} = await session.execute({
                    parameters: {
                        '$seriesId': episode.getTypedValue('seriesId'),
                        '$seasonId': episode.getTypedValue('seasonId'),
                        '$episodeId': episode.getTypedValue('episodeId')
                    },
                    text: `
                        DECLARE $seriesId AS Uint64;
                        DECLARE $seasonId AS Uint64;
                        DECLARE $episodeId AS Uint64;

                        SELECT title,
                               air_date
                        FROM episodes
                        WHERE series_id = $seriesId
                          AND season_id = $seasonId
                          AND episode_id = $episodeId;`
                });
                const {value: resultSet} = await resultSets.next();
                const {value: row} = await resultSet.rows.next();
                await opFinished;
                logger.info(`Parametrized select query ${JSON.stringify(row, null, 2)}`);
            }
        }
    });
}
```

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.

Here's a code sample demonstrating how to explicitly use the `Session.beginTransaction()` and `Session.commitTransaction()` calls to execute a transaction:

do()

```
async function explicitTcl(driver: Driver, ids: ThreeIds, logger: Logger) {
    logger.info('Running a parametrized query with explicit transaction control...');
    await driver.queryClient.do({
        fn: async (session) => {
            await session.beginTransaction({serializableReadWrite: {}});
            const [seriesId, seasonId, episodeId] = ids;
            const episode = new Episode({seriesId, seasonId, episodeId, title: '', airDate: new Date()});
            await session.execute({
                parameters: {
                    '$seriesId': episode.getTypedValue('seriesId'),
                    '$seasonId': episode.getTypedValue('seasonId'),
                    '$episodeId': episode.getTypedValue('episodeId')
                },
                text: `
```

```
                    DECLARE $seriesId AS Uint64;
                    DECLARE $seasonId AS Uint64;
                    DECLARE $episodeId AS Uint64;

                    UPDATE episodes
                    SET air_date = CurrentUtcDate()
                    WHERE series_id = $seriesId
                      AND season_id = $seasonId
                      AND episode_id = $episodeId;`
            })
            const txId = session.txId;
            await session.commitTransaction();
            logger.info(`TxId ${txId} committed.`);
        }
    });
}
```

doTx()

```
async function transactionPerWholeDo(driver: Driver, ids: ThreeIds, logger: Logger) {
    logger.info('Running a query with one transaction per whole doTx()...');
    await driver.queryClient.doTx({
        txSettings: {serializableReadWrite: {}},
        fn: async (session) => {
            const [seriesId, seasonId, episodeId] = ids;
            const episode = new Episode({seriesId, seasonId, episodeId, title: '', airDate: new Date()});
            await session.execute({
                parameters: {
                    '$seriesId': episode.getTypedValue('seriesId'),
                    '$seasonId': episode.getTypedValue('seasonId'),
                    '$episodeId': episode.getTypedValue('episodeId')
                },
                text: `
                    DECLARE $seriesId AS Uint64;
                    DECLARE $seasonId AS Uint64;
                    DECLARE $episodeId AS Uint64;

                    UPDATE episodes
                    SET air_date = CurrentUtcDate()
                    WHERE series_id = $seriesId
                      AND season_id = $seasonId
                      AND episode_id = $episodeId;`
            })
            logger.info(`TxId ${session.txId} will be committed by doTx().`);
        }
    });
}
```

Error handling

For more information about error handling, see Error handling in the API.

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
reference/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,
$episode_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 special
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 special
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)
```

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.
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.

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.

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-short-name } 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`.

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 that includes data about all the operations that tried to change the YDB objects, successfully or unsuccessfully:

- Database: Creating, editing, and deleting databases.
- Directory: Creating and deleting.
- Table: Creating or editing table schema, changing the number of partitions, backup and recovery, copying and renaming, and deleting tables.
- Topic: Creating, editing, and deleting.
- ACL: Editing.

The data of the audit log stream can be delivered to:

- File on each YDB cluster node.
- Agent for delivering Unified Agent metrics.
- Standard error stream, `stderr`.

You can use any of the listed destinations or their combinations.

If you forward the stream to a file, access to the audit log is set by file-system rights. Saving the audit log to a file is recommended for production installations.

Forwarding the audit log to the standard error stream (`stderr`) is recommended for test installations. Further stream processing is determined by the YDB cluster logging settings.

Audit log events

The information about each operation is saved to the audit log as a separate event. Each event includes a set of attributes. Some attributes are common across events, while other attributes are determined by the specific YDB component that generated the event.

Attribute	Description
Common attributes	
`subject`	Event source SID (`<login>@<subsystem>` format). Unless mandatory authentication is enabled, the attribute will be set to `{none}`. Required.
`operation`	Names of operations or actions are similar to the YQL syntax (for example, `ALTER DATABASE`, `CREATE TABLE`). Required.
`status`	Operation completion status. Acceptable values: • `SUCCESS`: The operation completed successfully. • `ERROR`: The operation failed. • `IN-PROCESS`: The operation is in progress. Required.
`reason`	Error message. Optional.
`component`	Name of the YDB component that generated the event (for example, `schemeshard`). Optional.
`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. Optional.
`remote_address`	The IP of the client that delivered the request. Optional.
`detailed_status`	The status delivered by a YDB component (for example, `StatusAccepted`, `StatusInvalidParameter`, `StatusNameConflict`). Optional.
Ownership and permission attributes	
`new_owner`	The SID of the new owner of the object when ownership is transferred. Optional.

`acl_add`	List of added permissions in short notation (for example, `[+R:someuser]`). Optional.
`acl_remove`	List of revoked permissions in short notation (for example, `[-R:someuser]`). Optional.
Custom attributes	
`user_attrs_add`	List of custom attributes added when creating objects or updating attributes (for example, `[attr_name1: A, attr_name2: B]`). Optional.
`user_attrs_remove`	List of custom attributes removed when creating objects or updating attributes (for example, `[attr_name1, attr_name2]`). Optional.
Attributes of the SchemeShard component	
`tx_id`	Unique transaction ID. Similarly to `request_id`, this ID can be used to differentiate events related to different operations. Required.
`database`	Database path (for example, `/my_dir/db`). 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]`). Required.

Enabling audit log

Delivering events to the audit log stream is enabled for the entire YDB cluster. To enable it, add, to the cluster configuration, the `audit_config` section, and specify in it one of the stream destinations (`file_backend`, `unified_agent_backend`, `stderr_backend`) or their combination:

```
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
```

Key	Description
`file_backend`	Write the audit log to a file at each cluster node.Optional.
`format`	Audit log format. The default value is `JSON`. Acceptable values: • `JSON`: Serialized JSON. • `TXT`: Text format. 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. This parameter is required if you use `file_backend`.
`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.Optional.

`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"`. Optional.
`stderr_backend`	Forward the audit log to the standard error stream (`stderr`).Optional.

Sample configuration that saves the audit log text to `/var/log/ydb-audit.log`:

```
audit_config:
  file_backend:
    format: TXT
    file_path: "/var/log/ydb-audit.log"
```

Sample configuration that saves the audit log text to Yandex Unified Agent with the `audit` label and outputs it to `stderr` in JSON format:

```
audit_config:
  unified_agent_backend:
    format: TXT
    log_name: audit
  stderr_backend:
    format: JSON
```

Examples

Fragment of audit log file in `JSON` format.

```
2023-03-13T20:05:19.776132Z: {"paths":"
[/my_dir/db1/some_dir]","tx_id":"562949953476313","database":"/my_dir/db1","remote_address":"ipv6:
[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]:xxxxx","status":"SUCCESS","subject":"
{none}","detailed_status":"StatusAccepted","operation":"CREATE DIRECTORY","component":"schemeshard"}
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}","detailed_status":"StatusAlreadyExists","operation":"CREATE DIRECTORY","component":"schemeshard"}
2023-03-13T19:59:27.614731Z: {"paths":"
[/my_dir/db1/some_table]","tx_id":"562949953426315","database":"/my_dir/db1","remote_address":"
{none}","status":"SUCCESS","subject":"{none}","detailed_status":"StatusAccepted","operation":"CREATE
TABLE","component":"schemeshard"}
2023-03-13T20:10:44.345767Z: {"paths":"[/my_dir/db1/some_table,
/my_dir/db1/another_table]","tx_id":"562949953506313","database":"{none}","remote_address":"ipv6:
[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]:xxxxx","status":"SUCCESS","subject":"
{none}","detailed_status":"StatusAccepted","operation":"ALTER TABLE RENAME","component":"schemeshard"}
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","component":"schemeshard","acl_add":"[+
(ConnDB):subject:-]"}
```

Event that occurred at `2023-03-13T20:05:19.776132Z` in JSON-pretty:

```
{
  "paths": "[/my_dir/db1/some_dir]",
  "tx_id": "562949953476313",
  "database": "/my_dir/db1",
  "remote_address": "ipv6:[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]:xxxxx",
  "status": "SUCCESS",
  "subject": "{none}",
  "detailed_status": "StatusAccepted",
  "operation": "CREATE DIRECTORY",
  "component": "schemeshard"
}
```

The same events in `TXT` format will look as follows:

```
2023-03-13T20:05:19.776132Z: component=schemeshard, tx_id=844424930186969, 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=StatusAccepted
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/some_dir', error: path exist, request accepts it (id: [OwnerId: 72075186224037889, LocalPathId: 3],
type: EPathTypeDir, state: EPathStateNoChanges)
2023-03-13T19:59:27.614731Z: component=schemeshard, tx_id=562949953426315, remote_address={none}, subject={none},
database=/my_dir/db1, operation=CREATE TABLE, paths=[/my_dir/db1/some_table], status=SUCCESS,
detailed_status=StatusAccepted
2023-03-13T20:10:44.345767Z: component=schemeshard, tx_id=562949953506313, remote_address=ipv6:
[xxxx:xxx:xxx:xxx:x:xxxx:xxx:xxxx]:xxxxx, subject={none}, database={none}, operation=ALTER TABLE RENAME, paths=
[/my_dir/db1/some_table, /my_dir/db1/another_table], status=SUCCESS, detailed_status=StatusAccepted
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:-]
```

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.

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

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.
  - PostgreSQL wire protocol for external communication with client applications initially designed to work with PostgreSQL.
  - Kafka wire protocol for external communication with client applications initially designed to work with Apache Kafka.
  - HTTP for running the Embedded 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 configuration section.

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

- In general to connect to GitHub you can use: ssh/token/ssh from yubikey/password etc. Recommended method is ssh keys.
- If you don't have already created keys (or yubikey), then just create new keys. Full instructions are on this GitHub page.
- If you have personal keys and use skotty as ssh-agent:
  - Add keys to skotty with command ssh-add
  - Edit `~/.skotty/config.yaml` file by adding a section:

```
keys_order:
    - added
    - insecure
    - legacy
    - secure
```

- If you have a yubikey, you can use the legacy key from the yubikey:
  - Let's assume you have already configured yubikey (or configure yubikey locally)
  - On your laptop: `skotty ssh keys`
  - Upload `legacy@yubikey` ssh key to GitHub (via UI)
  - test connection on laptop: `ssh -T git@github.com`

Remote development

If you are developing on a remote dev host you can use the key from your laptop (generated keys or keys from yubikey). You need to configure key forwarding. (Full instructions are on this GitHub page).

Suppose your remote machine is dev123456.search.yandex.net.

- on your laptop add ssh forwarding (`~/.ssh/config`):

```
Host dev123456.search.yandex.net
    ForwardAgent yes
```

- on remote dev host add to `~/.bashrc` :

```
if [[ -S "$SSH_AUTH_SOCK" && ! -h "$SSH_AUTH_SOCK" ]]; then
    ln -sf "$SSH_AUTH_SOCK" ~/.ssh/ssh_auth_sock;
fi
export SSH_AUTH_SOCK=~/.ssh/ssh_auth_sock;
```

- test connection: `ssh -T git@github.com`

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

- {#T}
- {#T}
- 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.

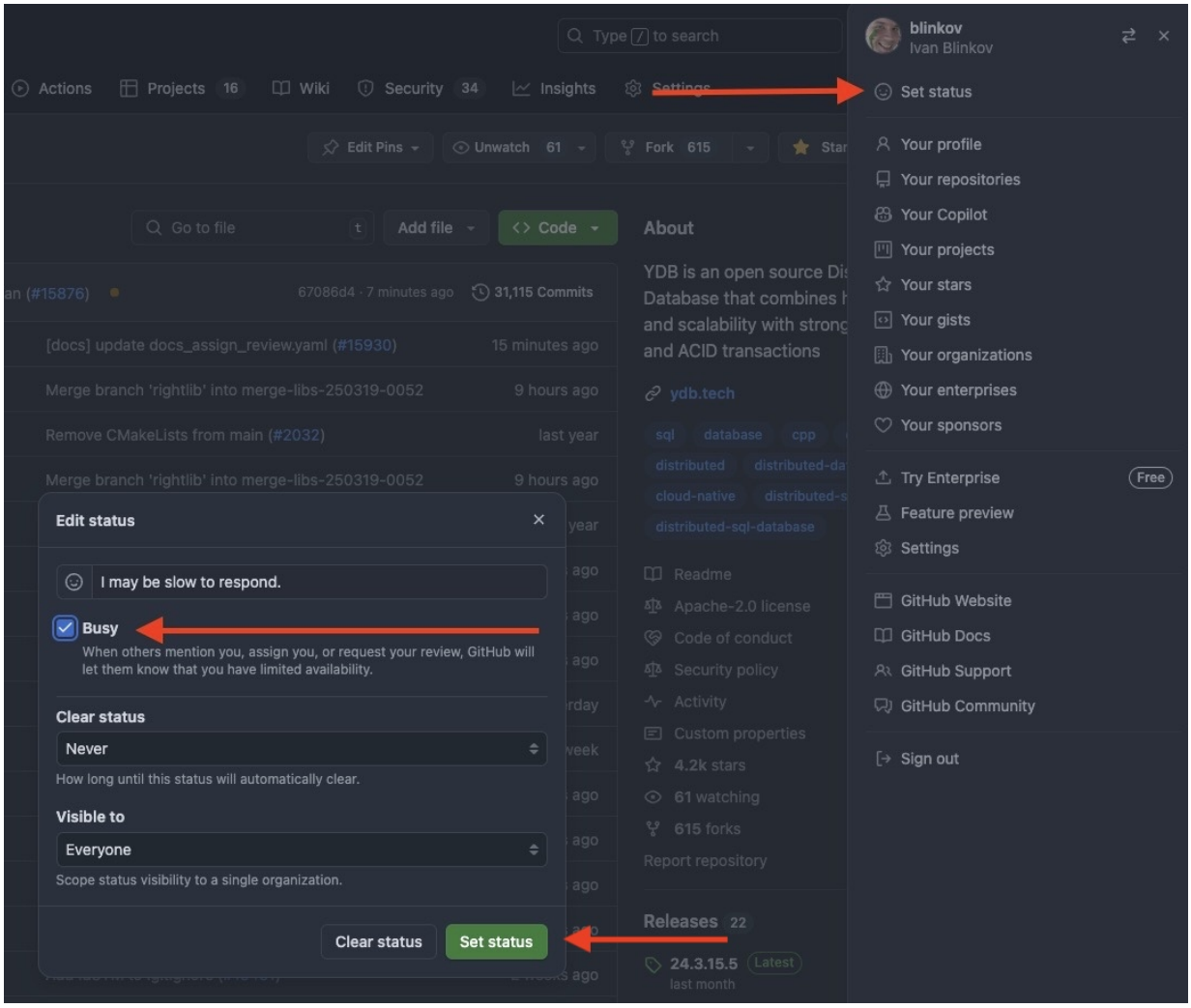
Roles

- **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

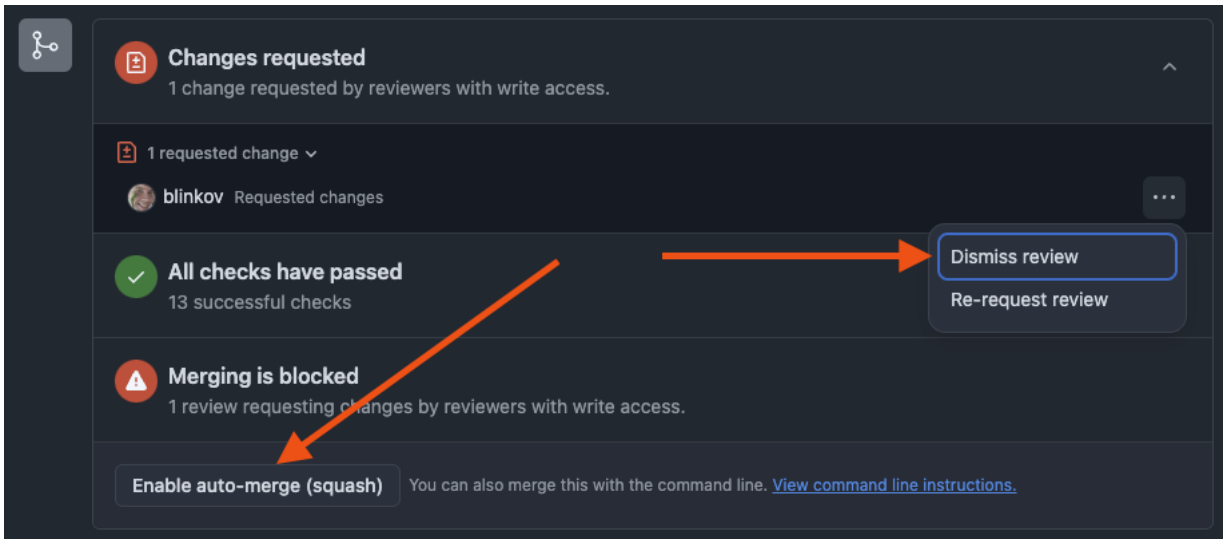
1. **Author** opens a GitHub pull request with suggested changes to the ydb/docs folder. Following the style guide from the beginning makes the review process smoother.
2. **Author** ensures the pull request is in a reviewable state by meeting all of the following criteria:
   2.1. The pull request has `* 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

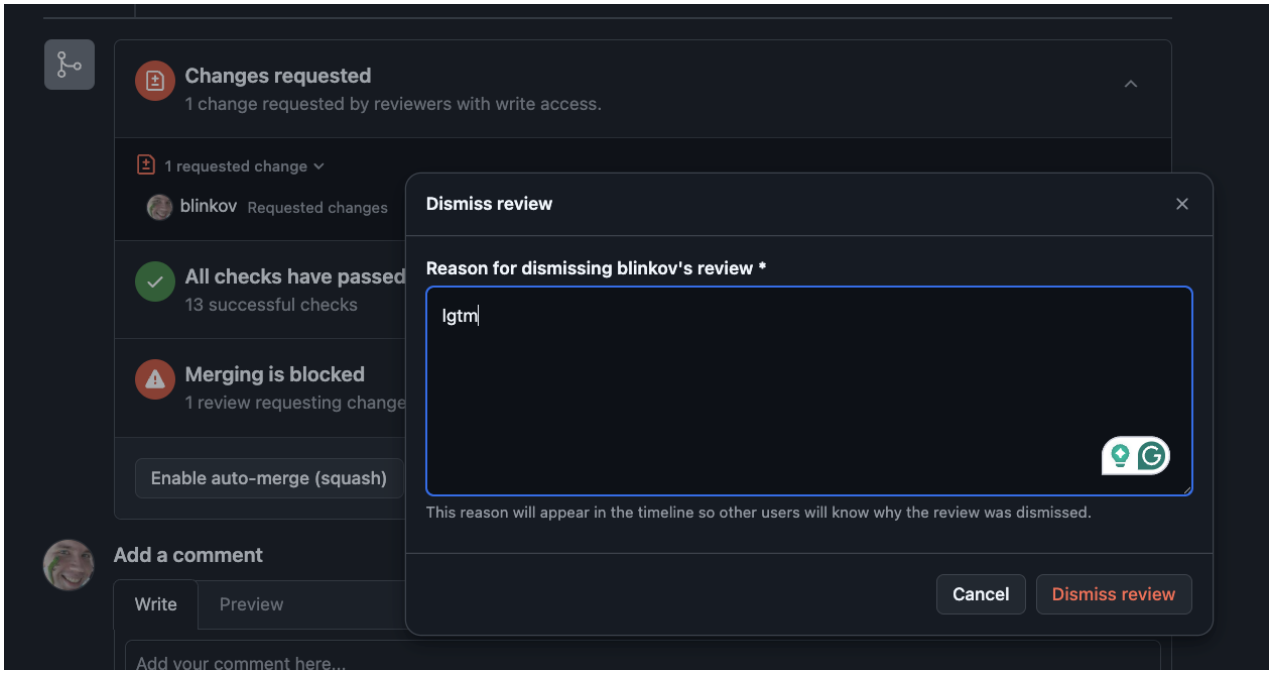


5. The **author** and **primary reviewer** iterate until the suggested change passes the checklist. The **primary reviewer** provides feedback via comments on the pull request, while the **author** addresses them. The expected turnaround time for each review iteration is two business days, up to a few 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

- [ ] 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.

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.
  - 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
- {#T}
- {#T}

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 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 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

DataShard


DataShard
DataShard
Hive
gRPC proxy/KQP
gRPC proxy/KQP
DataShard
DataShard
Dynamic
Node 1
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
BSC
Static...
Static
Node 1
gRPC proxy/KQP
gRPC proxy/KQP
CMS
CMS
Static...
Static
Node 2


NodeBroker
Tenant 1
Tenant 2
Static
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

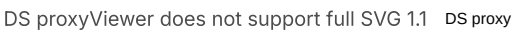
NodeWarden

NodeWarden

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).

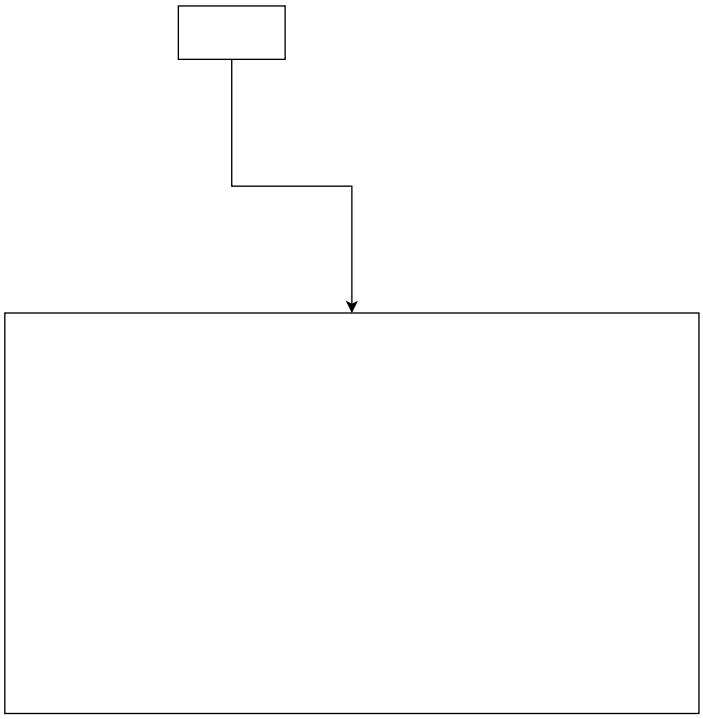
The tablet writes blobs of different types to different *channels*. A channel specifies the branch of storage in which to store blobs and performs various functions, such as:

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:



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

TTableStorageInfo

Channel 0

Channel 1

Channel 1

Channel 2

Channel N

TTabletChannelInfo

TTabletChannelInfo

0≤gen<10

0≤gen<10

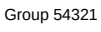
Group 12345

Group 12345

10≤gen<125


10≤gen<125


Group 54321


Group 54321


125≤gen


125≤gen

Group 12345

TabletViewer does not support full SVG 1.1

Tablet

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 1:1000

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 PDisk 4:1000

PDisk 4:1001 PDisk 4:1001

Node 1 Node 1

Node 2 Node 2

Node 3

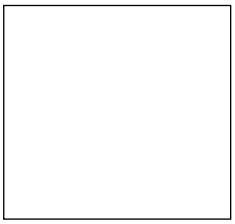
Node 4

DS proxyViewer does not support full SVG 1.1 DS proxy

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

5

5

6

6

7

7

fail realm 0

fail realm 0

fail...

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

VDISK[GroupId:Generation:0:6:0]

`VDISK[GroupId:Generation:0:6:0]`

0

0

1

2 2

3 3

4 4

5

5

6 6

7 7

8 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 VDisk ID.

`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` (3×3×1 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.

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.

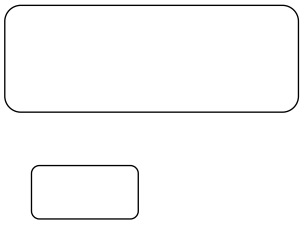
Overview of testing with load actors

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

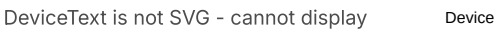
PDiskWriteLoad, PDis...

PDiskWriteLoad,
PDiskReadLoad,
PDiskLogLoad

Load actors

Load actors

YDB components **YDBcomponents**



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 Embedded 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.

Embedded 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 Embedded UI. For a description of output parameters, see the documentation of the respective actor.

Embedded 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

LocalDB: persistent uncommitted changes

Tablets may need to store a potentially large amount of data over a potentially long time, and then either commit or rollback all accumulated changes atomically without keeping them in-memory. To support this LocalDB allows table changes to be marked with a unique 64-bit transaction id (TxId), which are stored in the given table alongside committed data, but not visible until the given TxId is committed. The commit or rollback itself is atomic and very cheap, with committed data eventually integrated into the table as if written normally in the first place.

This feature is used as a building block for various other features:

- Storing uncommitted changes in long-running YQL transactions and observing them in subsequent queries in the same transaction
- Storing undecided side-effects in upcoming volatile distributed transactions
- Cross-region consistent replication, where table changes are streamed and applied in small batches, and periodically committed as consistent snapshots of the origin database state

Limitations

Current implementation has the following limitations:

1. Keys may have multiple changes written by different TxIds, however these transactions must be committed in the order of these writes. For example, when key K is updated by tx1 and then by tx2, it is possible to then commit tx1 first and then tx2, and observe both changes. However, if tx2 is committed first, then tx1 must be rolled back.
2. The number of uncommitted transactions and the amount of uncommitted data to a particular key must be limited by upper layers to some reasonably small value.
3. TxIds must not be reused after commit or rollback (even across different shards), and must be globally unique.

Logging uncommitted changes

Redo log (see flat_redo_writer.h) has the following events related to uncommitted changes:

- EvUpdateTx stores table changes with an uncommitted TxId. This event is generated by TDatabase::UpdateTx database method.
- EvRemoveTx is used for removing a given TxId (performing a rollback). This event is generated by TDatabase::RemoveTx database method.
- EvCommitTx is used for committing a given TxId at the specified MVCC version. This event is generated by TDatabase::CommitTx database method.

Storing uncommitted changes in MemTables

MemTable in LocalDB is a relatively small in-memory sorted tree that maps table keys to values. MemTable value is a chain of MVCC (partial) rows, each tagged with a row version (a pair of Step and TxId which is a global timestamp). Rows are normally pre-merged across the given MemTable. For example, let's suppose there have been the following operations for some 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 for key K in a single MemTable will look like this:

Version	Row
`v3000/12`	`SET A = 1, B = 2, C = 3`
`v2000/11`	`SET A = 1, B = 2`
`v1000/10`	`SET A = 1`

However, if the MemTable was split between updates, it may 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`

Changes are applied to the current MemTable, and uncommitted changes are no exception. However, they are tagged with a special version (where Step is the maximum possible number, as if they are in some "distant" future, and TxId is their uncommitted TxId), without

any pre-merging. For example, let's suppose we additionally performed the following operations:

TxId	Operation
15	`UPDATE ... SET C = 10`
13	`UPDATE ... SET B = 20`

The update chain for our key K will 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`

When reading iterator performs a lookup for changes with `Step == max` into an in-memory transaction map, which maps committed TxIds to their corresponding commit versions, and applies all committed deltas until it finds and applies a pre-merged row with `Step != max`.

Let's suppose we commit tx 13 at `v4000/20`. At that point transaction map is updated with `[13] => v4000/20`, and tx 13 is now committed. Any read afterwards will consult transaction map and apply deltas for tx 13, but skip tx 15, since it was not committed and treated as implicitly rolled back. MemTable chain for key K is not changed however.

Let's suppose we now perform an `UPDATE ... SET A = 30` at version `v5000/21`, the resulting chain 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`

Notice how the new record has its state pre-merged, including the previously committed delta for tx 13. Since tx 15 is not committed it was skipped and baked into a pre-merged state for `v5000/21`. It is important that tx 15 is not committed afterwards, and would result in a read anomaly otherwise: some versions would observe it as committed, and some won't.

Compacting uncommitted changes

Compaction takes some parts from the table, merges them in a sorted order, and writes as a new SST, which replaces compacted data. When compacting MemTable it also implies compacting the relevant redo log, and includes `EvRemoveTx` / `EvCommitTx` events, which affect change visibility and must also end up in persistent storage. LocalDB writes TxStatus blobs (see flat_page_txstatus.h), which store a list of committed and removed transactions, and replace the compacted redo log in regard to `EvRemoveTx` / `EvCommitTx` events. Compaction uses the latest transaction status maps, but it filters them leaving only those transactions that are mentioned in the relevant MemTables or previous TxStatus pages, so that it matches the compacted redo log.

Data pages (see flat_page_data.h) store uncommitted deltas from MemTables (or other SSTs) aggregated by their TxId in the same order just before the primary record. Records may have MVCC flags (HasHistory, IsVersioned, IsErased), which specify whether there is MVCC fields and data present. Delta records have an IsDelta flag, which is really a HasHistory flag without other MVCC flags. Since it was never used by previous versions (HasHistory flag was only ever used together with IsVersioned flag, you could not have history rows without a verioned record), it clearly identifies record as an uncommitted delta. Delta records have a TDelta info immediately after the fixed record data, which specifies TxId of the uncommitted delta.

One key may have several uncommitted delta records, as well as (optionally) the latest committed record data. Historically, data pages could only have one record (and one record pointer) per key, so the record pointer leads to the top of the delta chain, and other records are available via additional per-record offset table for other records:

Offset	Description
-X*8	offset of Main

...	...
-16	offset of Delta 2
-8	offset of Delta 1
0	header of Delta 0
...	...
offset of Delta 1	header of Delta 1
...	...
offset of Main	header of Main

Having a pointer to Delta 0, other records for the same key are available with the `GetAltRecord(size_t index)` method, where `index` is the record number (which is 1 for Delta 1). The chain of records ends either with a pointer to the record without an IsDelta flag (the Main record), or 0 (when there is no Main record for the key).

Let's suppose that after writing tx 13 above the MemTable was compacted. Entry for the 32-bit key K may look like this (offsets are relative to the record pointer on the table):

Offset	Value	Description
-16	58	offset of Main
-8	29	offset of Delta 1
0	0x21	Delta 0: IsDelta + ERowOp::Upsert
1	0x00	.. key column is not NULL
2	K	.. key column (32-bit)
6	0x00	.. column A is empty
7	0	.. column A (32-bit)
11	0x01	.. column B = ECellOp::Set
12	20	.. column B (32-bit)
16	0x00	.. column C is empty
17	0	.. column C (32-bit)
21	13	.. TDelta::TxId
29	0x21	Delta 1: IsDelta + ERowOp::Upsert
30	0x00	.. key column is not NULL
31	K	.. key column (32-bit)
35	0x00	.. column A is empty
36	0	.. column A (32-bit)
40	0x00	.. column B is 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 is not 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 HasHistory flag in the Main record shows that other two records are stored among history data with keys `(RowId, 2000, 11)` and `(RowId, 1000, 10)` respectively.

When compacting iterator runs in a special mode that enumerates all deltas and all versions for each key. The compaction scan implementation (see flat_ops_compact.h) first aggregates all uncommitted deltas by their TxId in the same order (in case changes from different TxIds overlap their order may change arbitrarily, which is OK since such transactions are not supposed to both commit). After uncommitted deltas are aggregated, they are flushed to the resulting SST (see flat_part_writer.h and flat_page_writer.h), and committed row versions for the same key are enumerated, which are written in decreasing version order.

When iterator positions to the first committed delta (i.e. the IsDelta record which has commit info in the transaction map), the commit info is used as the resulting row versions, with row state combined from all deltas below, including the first committed record from each LSM level participating in compaction. When positioning to the next version iterator skips delta with version at or below the last one and the process is repeated.

With a large number of compacted deltas for a key, when they are later committed with different versions, the process of generating committed records grows quadratically in the number of deltas. For this reason upper levels should control the number of deltas for each key and must not allow them to grow too large (deltas might be merged in the reverse order in the future to side step this problem). Other limitation is that uncommitted deltas for a given key need to all be in memory and on a single page, since each read must walk the entire delta chain and check whether each record is committed. In the future we may want to store deltas across multiple pages, but since they all need to be in memory anyway there is little reason to do so. Removing the requirement for all deltas to be in memory during reads is theoretically possible, but it requires storing them in a different form.

Uncommitted transaction stats

Optimistically most transactions are eventually committed, but sometimes transactions roll back, even after compaction. Since rollbacks may cause a large amount of data to become unreachable, SSTs store TxIdStats pages (see flat_page_txidstat.h) with the number of rows and bytes occupied by each TxId. As more and more transactions are rolled back, compaction strategy aggregates the number of unreachable bytes, and eventually runs garbage collecting compactions.

These pages are also used for keeping in-memory transaction maps small. When transaction is eventually committed or rolled back, its status is stored in an in-memory hashmap as long as the specified TxId has deltas anywhere. As SSTs are compacted, uncommitted deltas from committed transactions are rewritten into fully committed records, rolled back deltas are removed entirely, and eventually transactions stop being mentioned in TxIdStats pages of resulting SSTs. When a given TxId is no longer mentioned anywhere, it is safely removed from transaction maps and no longer occupies any memory.

The in-memory transaction map is limited in size by limiting the number of open (which are neither committed nor rolled back) transactions at the datashard level. When compacting SSTs may only generate deltas for currently open transactions, so the total transaction map size is limited by the maximum number of open transactions, multiplied by the number of SSTs.

Borrowing SSTs with uncommitted changes

When copying tables, and when datashards split or merge they use LocalDB borrowing, where source shard SSTs are merged into destination shard tables. When using uncommitted changes those may contain changes from open transactions, or those which have been committed or rolled back, but not compacted yet. To guarantee that destination shards have the same view of the data as the source shards, TxStatus blobs also need borrowing, modifying destination transaction maps.

Note that transaction may have different status at different shards. Let's review a hypothetical example:

- Transaction writes changes to key K with TxId at shard S, which are compacted into a large SST
- Shard S becomes too large and splits into shards L and R, so that SST is borrowed by both with row filters applied, key K ends up in shard L, but transaction TxId is also phantomly present in shard R, since it is mentioned in the borrowed SST
- Transaction commits changes at shard L, without a commit at shard R (since logically and from the writer's perspective there have been no changes at shard R)
- Since transaction has finished, this TxId would be eventually rolled back at shard R (which doesn't cause any visible side-effects)
- Let's supposed that shards L and R are eventually merged

When merged TxStatus from shard L would specify TxId as committed, but TxStatus from shard R would specify TxId as rolled back. Since transactions are supposed to commit all changes, and phantom rollbacks are possible, when conflicting TxStatus are merged commit "wins" over rollback.

Note that this is purely theoretical, in reality shards currently fully compact before splitting or merging, so conflicting TxStatus should not be possible, this is done purely as an additional safety net and must be taken into account. This means that shards are not allowed to partially commit transaction changes, all changes from a given TxId must be committed. This also means TxIds must never be reused, even across shards, and only globally unique TxIds are safe to use for uncommitted changes.

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 KQP, 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 KQP 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 KQP 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 KQP 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 KQP, 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 KQP 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 KQP 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 KQP 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. KQP 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, KQP 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. KQP 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), KQP 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 KQP 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 KQP 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 KQP 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.

KqpLoad

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 }
    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 }
```

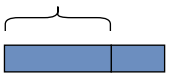
```
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    Chunks: { Slots: 4096 Weight: 1 }
    DurationSeconds: 120
    IntervalMsMin: 0
    IntervalMsMax: 0
    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...

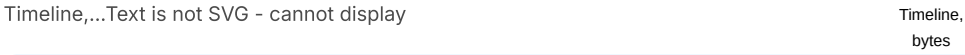
BurstInter... BurstInterval
VDisk2

StorageDuration VDisk2 StorageDuration VDisk2

BurstSize VDisk2

Data that is being deleted

Data that is being dele...



> **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 Embedded 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 Embedded 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
}
```

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.

YDB compatibility with PostgreSQL

> **Warning**
>
> At the moment, YDB's compatibility with PostgreSQL **is under development**, so not all PostgreSQL constructs and functions are supported yet. PostgreSQL compatibility is available for testing in the form of a Docker container, which can be deployed by following these instructions.

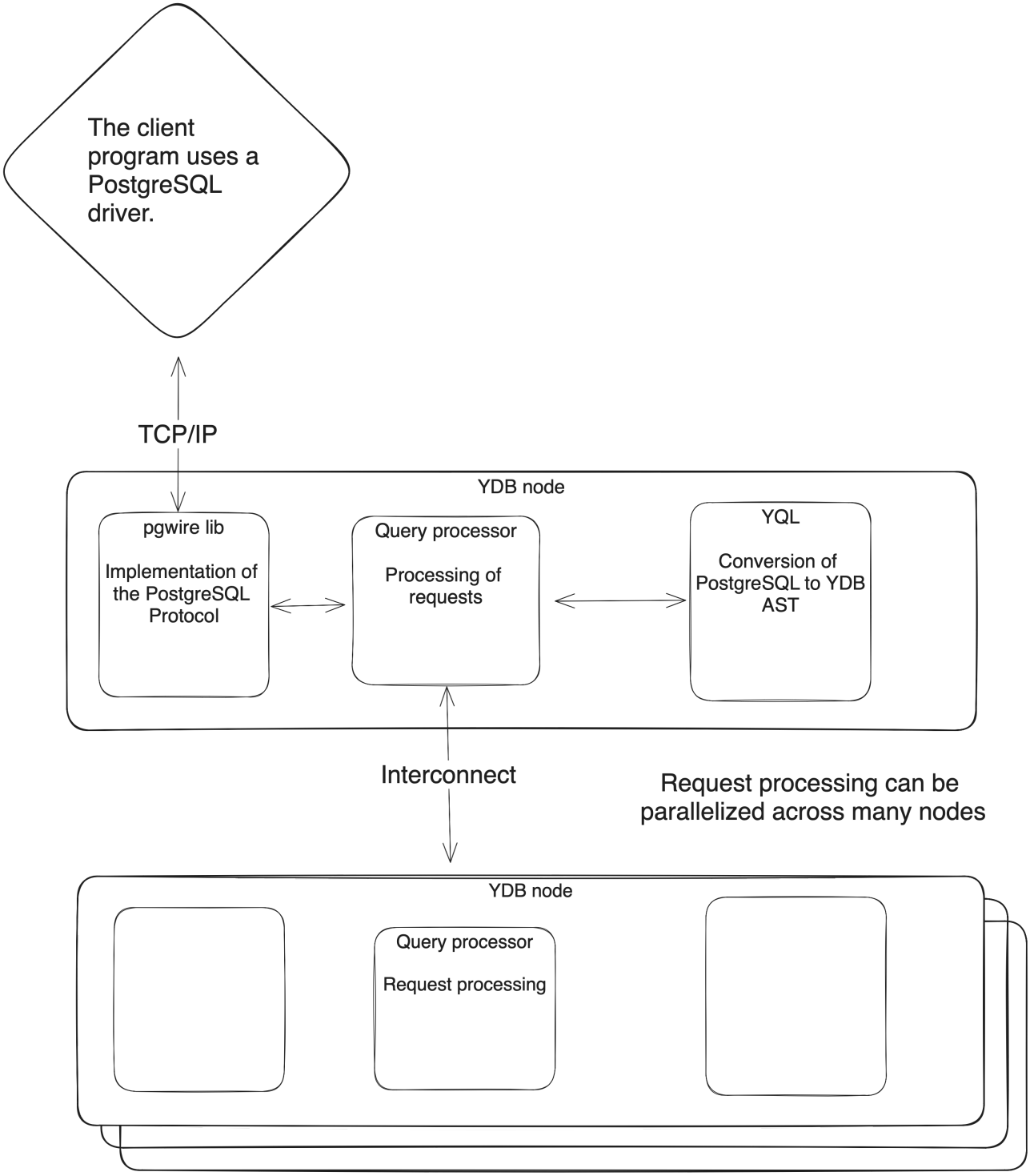
PostgreSQL compatibility is a mechanism for executing SQL queries in the PostgreSQL dialect on YDB infrastructure using the PostgreSQL wire protocol. This feature allows the use of familiar PostgreSQL tools such as psql and drivers (e.g., pq for Golang and psycopg2 for Python). Developers can write queries using the PostgreSQL syntax while benefiting from YDB's advantages such as horizontal scalability and fault tolerance.

YDB's compatibility with PostgreSQL simplifies the migration of applications that were previously operating within the PostgreSQL ecosystem. This feature allows for a smoother transition of database-driven applications to YDB. At present, a limited set of PostgreSQL 14 instructions and functions are supported. PostgreSQL compatibility enables switching from PostgreSQL to YDB without modifying the project code (provided that the SQL constructs used in the project are supported by YDB), by merely changing the connection parameters.

The operation of PostgreSQL compatibility can be described as follows:

1. The application sends queries to YDB, where they are processed by a component known as pgwire. Pgwire implements the wire protocol of PostgreSQL and forwards commands to the query processor.
2. The query processor translates the PostgreSQL queries into YQL AST.
3. After the queries are processed, the results are compiled and sent back to the application that issued the query via the PostgreSQL wire protocol. During query processing, it can be parallelized and executed on any number of YDB nodes.

The functionality of PostgreSQL compatibility can be graphically represented as follows:



Using the embedded web 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.

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 network protocols PostgreSQL or Apache Kafka. Due to the compatibility layer, many tools designed to work with these systems can also interact with YDB. The compatibility level of each specific application needs to be clarified separately.

Graphical user interfaces

Environment	Instruction	Compatibility level
Embedded 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	PostgreSQL wire protocol	Instruction
DataLens	Full	Instruction
FineBI	PostgreSQL wire protocol	Instruction
Grafana	Full	Instruction

Orchestration

System	Instruction
Apache Airflow™	Instruction

Data ingestion

Delivery System	Instruction
FluentBit	Instruction
LogStash	Instruction
Kafka Connect Sink	Instruction
Arbitrary JDBC data sources	Instruction

Streaming data ingestion

Delivery System	Instruction
Apache Kafka API	Instruction

Data migrations

Environment	Instruction
goose	Instruction
Liquibase	Instruction
Flyway	Instruction

Object–relational mapping (ORM)

Delivery System	Instruction
Hibernate	Instruction
Spring Data JDBC	Instruction
JOOQ	Instruction
Dapper	Instruction

Vector search

System	Instruction
LangChain	Instruction

See also

- YDB SDK reference
- YDB compatibility with PostgreSQL
- Kafka API

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 SDK reference

OpenSource SDKs in the following programming languages are available to work with YDB:

Language	GitHub repository	API reference
C++	ydb-platform/ydb/tree/main/ydb/public/sdk/cpp	N/A
C# (.NET)	ydb-platform/ydb-dotnet-sdk	N/A
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
Node.js	ydb-platform/ydb-nodejs-sdk	N/A
PHP	ydb-platform/ydb-php-sdk	N/A
Python	ydb-platform/ydb-python-sdk	https://ydb-platform.github.io/ydb-python-sdk
Rust	ydb-platform/ydb-rs-sdk	N/A

The SDK documentation contains the following sections:

- Installation
- Authentication
- Handling errors
- Code recipes
- Comparison of SDK features

See also:

- Documentation for Application Developers
- Example applications

Languages and APIs

- ADO.NET - .NET Access to YDB
- JDBC driver for YDB
- YDB Model Context Protocol Server

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:

- Usage examples
- Constraints

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 describes the main groups of configurable parameters in this file.

host_configs: Typical host configurations

A YDB cluster consists of multiple nodes, and one or more typical server configurations are usually used for their deployment. To avoid repeating its 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: Static cluster nodes

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'
```

Kubernetes features

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.

domains_config: Cluster domain

This section contains the configuration of the YDB cluster root domain, including the Blob Storage (binary object storage) and State Storage configurations.

Syntax

```
domains_config:
  domain:
  - name: <root domain name>
    storage_pool_types: <Blob Storage configuration>
  state_storage: <State Storage configuration>
  security_config: <authentication configuration>
```

Blob Storage configuration

This section defines one or more types of storage pools available in the cluster for the data in the databases 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

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.

Actor system

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). Configuring is 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 configuring.
- Number of CPU cores for each YDB cluster subsystem in the case of manual configuring.

Automatic configuring adapts to the current system workload. It is recommended in most cases.

You might opt for manual configuring 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 Embedded UI monitoring page.

Automatic configuring

Example of the `actor_system_config` section for automatic configuring of the actor system:

```
actor_system_config:
  use_auto_config: true
  node_type: STORAGE
  cpu_count: 10
```

Parameter	Description
`use_auto_config`	Enabling automatic configuring 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 configuring 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.

Memory controller

There are many components inside YDB database nodes that utilize memory. Most of them need a fixed amount, but some are flexible and can use varying amounts of memory, typically to improve performance. If YDB components allocate more memory than is physically available, the operating system is likely to terminate the entire YDB process, which is undesirable. The memory controller's goal is to allow YDB to avoid out-of-memory situations while still 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 and accelerate data retrieval.
- MemTable: holds data that has not yet been flushed to SST.
- KQP: stores intermediate query results.
- Allocator caches: keep memory blocks that have been released but not yet returned to the operating system.

Memory limits can be configured to control overall memory usage, ensuring the database operates efficiently within the available resources.

Hard memory limit

The hard memory limit specifies the total amount of memory available to YDB process.

By default, the hard memory limit for YDB process is set to its cgroups memory limit.

In environments without a cgroups memory limit, the default hard memory limit equals to the host's total available memory. This configuration allows the database to utilize all available resources but may lead to resource competition with other processes on the same host. Although the memory controller attempts to account for this external consumption, such a setup is not recommended.

Additionally, the hard memory limit can be specified in the configuration. Note that the database process may still exceed this limit. Therefore, it is highly recommended to use cgroups memory limits in production environments to enforce strict memory control.

Most of other memory limits can be configured either in absolute bytes or as a percentage relative to the hard memory limit. Using percentages is advantageous for managing clusters with nodes of varying capacities. If both absolute byte and percentage limits are specified, the memory controller uses a combination of both (maximum for lower limits and minimum for upper limits).

Example of the `memory_controller_config` section with a specified hard memory limit:

```
memory_controller_config:
  hard_limit_bytes: 16106127360
```

Soft memory limit

The soft memory limit specifies a dangerous threshold that should not be exceeded by YDB process under normal circumstances.

If the soft limit is exceeded, YDB gradually reduces the shared cache size to zero. Therefore, more database nodes should be added to the cluster as soon as possible, or per-component memory limits should be reduced.

Target memory utilization

The target memory utilization specifies a threshold for YDB process memory usage that is considered optimal.

Flexible cache sizes are calculated according to their limit thresholds to keep process consumption around this value.

For example, in a database that consumes a little memory on query execution, caches consume memory around this threshold, and other memory stays free. If query execution consumes more memory, caches start to reduce their sizes to their minimum threshold.

Per-component memory limits

There are two different types of components within YDB.

The first type, known as cache components, functions as caches, for example, by storing the most recently used data. Each cache component has minimum and maximum memory limit thresholds, allowing them to adjust their capacity dynamically based on the current YDB process consumption.

The second type, known as activity components, allocates memory for specific activities, such as query execution or the compaction process. Each activity component has a fixed memory limit. Additionally, there is a total memory limit for these activities from which they attempt to draw the required memory.

Many other auxiliary components and processes operate alongside the YDB process, consuming memory. Currently, these components do not have any memory limits.

Cache components memory limits

The cache components include:

- Shared cache
- MemTable

Each cache component's limits are dynamically recalculated every second to ensure that each component consumes memory proportionally to its limit thresholds while the total consumed memory stays close to the target memory utilization.

The minimum memory limit threshold for cache components isn't reserved, meaning the memory remains available until it is actually used. However, once this memory is filled, the components typically retain the data, operating within their current memory limit. Consequently, the sum of the minimum memory limits for cache components is expected to be less than the target memory utilization.

If needed, both the minimum and maximum thresholds should be overridden; otherwise, any missing threshold will have a 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
```

Activity components memory limits

The activity components include:

- KQP

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 further constrained by an additional limit known as the activities memory limit. If the total memory usage of the activity components exceeds this limit, any additional memory requests will be denied.

As a result, while the combined individual limits of the activity components might collectively exceed the activities memory limit, each component's individual limit should be less than this overall cap. Additionally, the sum of the minimum memory limits for the cache components, plus the activities memory limit, must be less than the soft memory limit.

There are some other activity components that currently do not have individual memory limits.

Example of the `memory_controller_config` section with a specified KQP limit:

```
memory_controller_config:
  query_execution_limit_percent: 25
```

Configuration parameters

Each configuration parameter applies within the context of a single database node.

As mentioned above, the sum of the minimum memory limits for the cache components plus the activities memory limit should be less than the soft memory limit.

This restriction can be expressed in a simplified form:

Or in a detailed form:

Parameter	Default	Description
`hard_limit_bytes`	CGroup memory limit / Host memory	Hard memory usage limit.
`soft_limit_percent` / `soft_limit_bytes`	75%	Soft memory usage limit.
`target_utilization_percent` / `target_utilization_bytes`	50%	Target memory utilization.
`activities_limit_percent` / `activities_limit_bytes`	30%	Activities memory limit.
`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%	KQP memory limit.

blob_storage_config: Static cluster group

Specify a static cluster group's configuration. A static group is necessary for the operation of the basic cluster tablets, including `Hive`, `SchemeShard`, and `BlobstorageContoller`.
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.

Enabling stable node names

Node names are assigned through the Node Broker, which is a system tablet that registers dynamic nodes in the YDB 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>
```

Configuring Health Check

This 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)

Sample cluster configurations

You can find model cluster configurations for deployment in the repository. Check them out before deploying a cluster.

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 in YDB

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.
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.

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.

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
- {#T}

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.

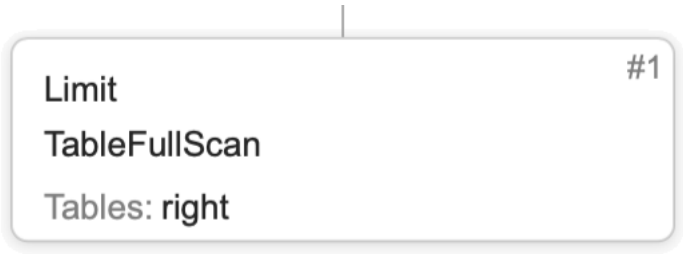
Below, you can find information about node types, and an example of analyzing a specific query plan can be found here.

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.

YQL data types

This section contains articles on YQL data types:

- Simple/Primitive 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 synonymously.

Numeric types

Type	Description	Notes
`Bool`	Boolean value.	
`Int8`	A signed integer. Acceptable values: from -2^7 to 2^7-1.	
`Int16`	A signed integer. Acceptable values: from -2^{15} to $2^{15}-1$.	
`Int32`	A signed integer. Acceptable values: from -2^{31} to $2^{31}-1$.	
`Int64`	A signed integer. Acceptable values: from -2^{63} to $2^{63}-1$.	
`Uint8`	An unsigned integer. Acceptable values: from 0 to 2^8-1.	
`Uint16`	An unsigned integer. Acceptable values: from 0 to $2^{16}-1$.	
`Uint32`	An unsigned integer. Acceptable values: from 0 to $2^{32}-1$.	
`Uint64`	An unsigned integer. Acceptable values: from 0 to $2^{64}-1$.	
`Float`	A real number with variable precision, 4 bytes in size.	Can't be used in the primary key
`Double`	A real number with variable precision, 8 bytes in size.	Can't be used in the primary key
`Decimal`	A real number with the specified precision, up to 35 decimal digits	When used in table columns, the precision is fixed: Decimal(22,9)
`DyNumber`	A binary representation of a real number with an accuracy of up to 38 digits. Acceptable values: positive numbers from 1×10^{-130} up to $1\times10^{126}-1$, negative numbers from $-1\times10^{126}-1$ to -1×10^{-130}, and 0. Compatible with the `Number` type in AWS DynamoDB. It's not recommended for ydb-native applications.	

String types

Type	Description	Notes
`String`	A string that can contain any binary data	
`Utf8`	Text encoded in UTF-8	
`Json`	JSON represented as text	Doesn't support matching, can't be used in the primary key
`JsonDocument`	JSON in an indexed binary representation	Doesn't support matching, can't be used in the primary key
`Yson`	YSON in a textual or binary representation.	Doesn't support matching, can't be used in the primary key
`Uuid`	Universally unique identifier UUID	

Cell size restrictions

The maximum value size for a non-key column cell with any string data type is 8 MB.

Unlike the `JSON` data type that stores the original text representation passed by the user, `JsonDocument` uses an indexed binary representation. An important difference from the point of view of semantics is that `JsonDocument` doesn't preserve formatting, the order of keys in objects, or their duplicates.

Thanks to the indexed view, `JsonDocument` lets you bypass the document model using `JsonPath` without the need to parse the full content. This helps efficiently perform operations from the JSON API, reducing delays and cost of user queries. Execution of `JsonDocument` queries can be up to several times more efficient depending on the type of load.

Due to the added redundancy, `JsonDocument` is less effective in storage. The additional storage overhead depends on the specific content, but is 20-30% of the original volume on average. Saving data in `JsonDocument` format requires additional conversion from the textual representation, which makes writing it less efficient. However, for most read-intensive scenarios that involve processing data from JSON, this data type is preferred and recommended.

Warning

To store numbers (JSON Number) in `JsonDocument`, as well as for arithmetic operations on them in the JSON API, the Double type is used. Precision might be lost when non-standard representations of numbers are used in the source JSON document.

Date and time

Type	Description	Notes
`Date`	Date, precision to the day	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
`Datetime`	Date/time, precision to the second	Internal representation: Unsigned 32-bit integer
`Timestamp`	Date/time, precision to the microsecond	Internal representation: Unsigned 64-bit integer
`Interval`	Time interval (signed), precision to microseconds	Value range: From -136 years to +136 years. Internal representation: Signed 64-bit integer. Can't be used in the primary key
`TzDate`	Date with time zone label, precision to the day	Not supported in table columns
`TzDateTime`	Date/time with time zone label, precision to the second	Not supported in table columns
`TzTimestamp`	Date/time with time zone label, precision to the microsecond	Not supported in table columns

Supporting types with a time zone label

Time zone label for the `TzDate`, `TzDatetime`, `TzTimestamp` types is an attribute that is used:

- When converting (CAST, DateTime::Parse, DateTime::Format) to a string and from a string.
- In DateTime::Split, a timezone component is added to `Resource<TM>`.

The point in time for these types is stored in UTC, and the timezone label doesn't participate in any other calculations in any way. For example:

```
SELECT -- these expressions are always true for any timezones: the timezone doesn't affect the point in time.
    AddTimezone(CurrentUtcDate(), "Europe/Moscow") ==
        AddTimezone(CurrentUtcDate(), "America/New_York"),
```

```
    AddTimezone(CurrentUtcDatetime(), "Europe/Moscow") ==
        AddTimezone(CurrentUtcDatetime(), "America/New_York");
```

Keep in mind that when converting between `TzDate` and `TzDatetime`, or `TzTimestamp` the date's midnight doesn't follow the local time zone, but midnight in UTC for the date in UTC.

Casting between data types

Explicit casting

Explicit casting using CAST:

Casting to numeric types

Type	Bool	Int	Uint	Float	Double	Decimal
Bool	—	Yes[1]	Yes[1]	Yes[1]	Yes[1]	No
INT	Yes[2]	—	Yes[3]	Yes	Yes	Yes
Uint	Yes[2]	Yes	—	Yes	Yes	Yes
Float	Yes[2]	Yes	Yes	—	Yes	No
Double	Yes[2]	Yes	Yes	Yes	—	No
Decimal	No	Yes	Yes	Yes	Yes	—
String	Yes	Yes	Yes	Yes	Yes	Yes
Utf8	Yes	Yes	Yes	Yes	Yes	Yes
Json	No	No	No	No	No	No
Yson	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes[4]	Yes[4]
Uuid	No	No	No	No	No	No
Date	No	Yes	Yes	Yes	Yes	No
Datetime	No	Yes	Yes	Yes	Yes	No
Timestamp	No	Yes	Yes	Yes	Yes	No
Interval	No	Yes	Yes	Yes	Yes	No

[1] `True` is converted to `1` and `False` to `0`.
[2] Any value other than `0` is converted to `True`, `0` is converted to `False`.
[3] Possible only in the case of a non-negative value.
[4] Using the built-in function Yson::ConvertTo.

Converting to date and time data types

Type	Date	Datetime	Timestamp	Interval
Bool	No	No	No	No
INT	Yes	Yes	Yes	Yes
Uint	Yes	Yes	Yes	Yes
Float	No	No	No	No
Double	No	No	No	No
Decimal	No	No	No	No
String	Yes	Yes	Yes	Yes
Utf8	Yes	Yes	Yes	Yes
Json	No	No	No	No
Yson	No	No	No	No

Uuid	No	No	No	No
Date	—	Yes	Yes	No
Datetime	Yes	—	Yes	No
Timestamp	Yes	Yes	—	No
Interval	No	No	No	—

Conversion to other data types

Type	String	Utf8	Json	Yson	Uuid
Bool	Yes	No	No	No	No
INT	Yes	No	No	No	No
Uint	Yes	No	No	No	No
Float	Yes	No	No	No	No
Double	Yes	No	No	No	No
Decimal	Yes	No	No	No	No
String	—	Yes	Yes	Yes	Yes
Utf8	Yes	—	No	No	No
Json	Yes	Yes	—	No	No
Yson	Yes[4]	No	No	No	No
Uuid	Yes	Yes	No	No	—
Date	Yes	Yes	No	No	No
Datetime	Yes	Yes	No	No	No
Timestamp	Yes	Yes	No	No	No
Interval	Yes	Yes	No	No	No

[4] 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, because it failed
    CAST(-1 AS Uint16) IS NULL, -- true, a negative integer cast to an unsigned integer
    CAST([-1, 0, 1] AS List<Uint8?>),            -- [null, 0, 1]
        --The item type is optional: the failed item is cast to null.
    CAST(["3.14", "bad", "42"] AS List<Float>),   -- [3.14, 42]
        --The item type is not optional: the failed item has been deleted.
    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 specify the operation result type, if the operation is possible:

Numeric types

Type	Int	Uint	Float	Double

INT	—	`INT`	`Float`	`Double`
Uint	`INT`	—	`Float`	`Double`
Float	`Float`	`Float`	—	`Double`
Double	`Double`	`Double`	`Double`	—

Date and time types

Type	Date	Datetime	Timestamp	Interval	TzDate	TzDatetime	TzTimestamp
Date	—	—	—	`Date`	—	—	—
Datetime	—	—	—	`Datetime`	—	—	—
Timestamp	—	—	—	`Timestamp`	—	—	—
Interval	`Date`	`Datetime`	`Timestamp`	—	`TzDate`	`TzDatetime`	`TzTimestamp`
TzDate	—	—	—	`TzDate`	—	—	—
TzDatetime	—	—	—	`TzDatetime`	—	—	—
TzTimestamp	—	—	—	`TzTimestamp`	—	—	—

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. Only Decimal(22, 9) is supported.

- 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
- ALTER TABLE
- UPDATE
- DELETE
- REPLACE
- UPSERT
- GROUP BY
- JOIN
- WINDOW
- FLATTEN
- ACTION
- INTO RESULT
- PRAGMA
- DECLARE
- CREATE ASYNC REPLICATION
- ALTER ASYNC REPLICATION
- DROP ASYNC REPLICATION

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 `||`.

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, because it failed
    CAST(-1 AS Uint16) IS NULL, -- true, a negative integer cast to an unsigned integer
    CAST([-1, 0, 1] AS List<Uint8?>),            -- [null, 0, 1]
        --The item type is optional: the failed item is cast to null.
```

```
    CAST(["3.14", "bad", "42"] AS List<Float>),   -- [3.14, 42]
        --The item type is not optional: the failed item has been deleted.
    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 $_
export $_; --- An error: Can not export 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;
```

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.

Examples

The following statement forces the asynchronous replication process to complete:

```
ALTER ASYNC REPLICATION my_replication SET (STATE = "DONE", FAILOVER_MODE = "FORCE");
```

See also

- CREATE ASYNC REPLICATION
- DROP ASYNC REPLICATION

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.

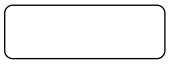
Built-in groups

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.

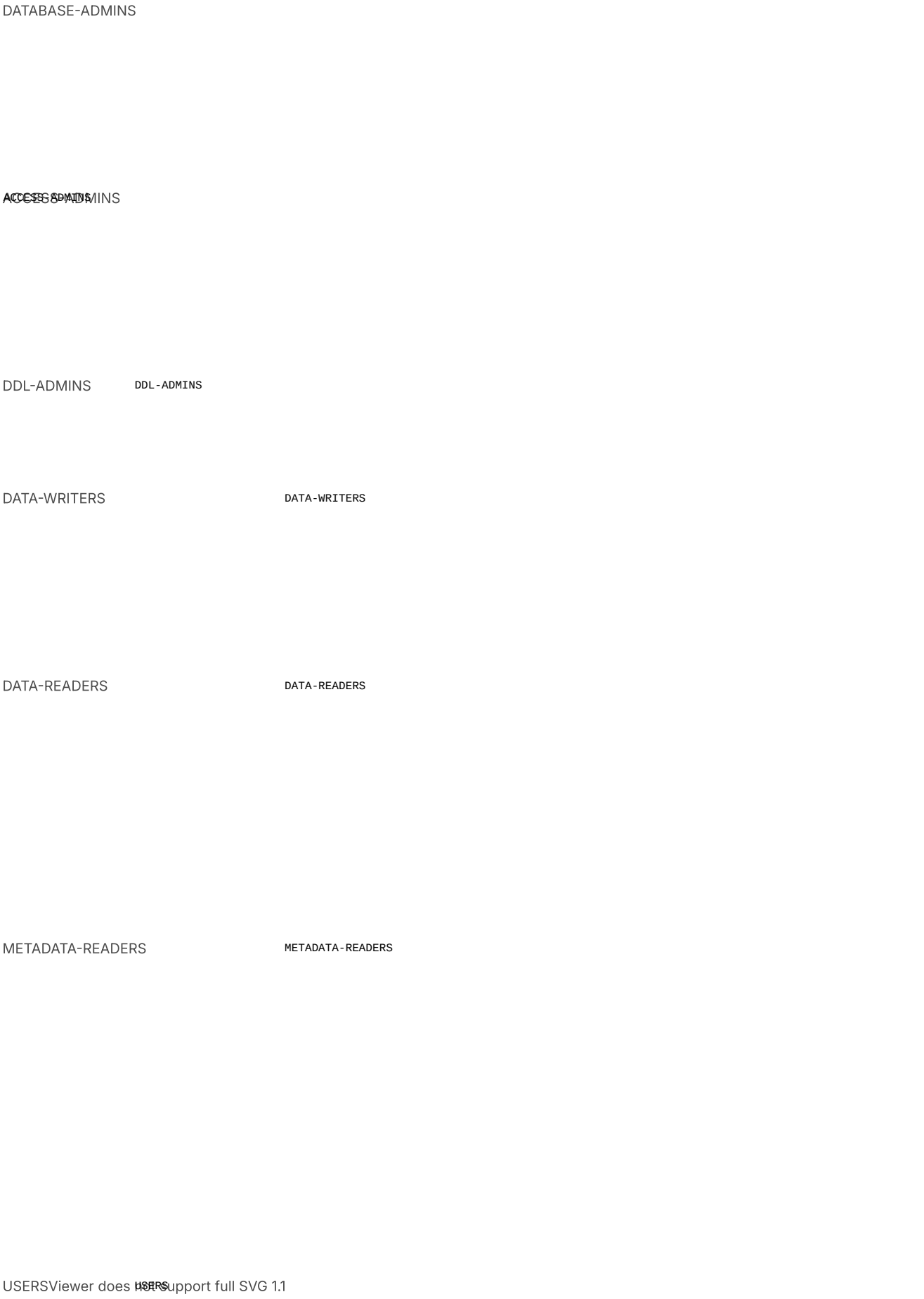
Below is a diagram demonstrating how groups inherit permissions from each other. For example, `DATA-WRITERS` includes all permissions of `DATA-READERS`:



ADMINS

`ADMINS`

`DATABASE-ADMINS`


DATABASE-ADMINS
ACCESS-ADMINS
ACCESS-ADMINS
DDL-ADMINS
DDL-ADMINS
DATA-WRITERS
DATA-WRITERS
DATA-READERS
DATA-READERS
METADATA-READERS
METADATA-READERS
USERS
USERS
Viewer does not support full SVG 1.1

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).

`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`, default value: `18h`.
- `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. `0` is interpreted as unlimited. Value type: `integer`, default value: `0`
- `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`.

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 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).

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.

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).
  - Authentication details for the source database (mandatory) depending on the authentication method:
    - Access token:
      - `TOKEN_SECRET_NAME` — the name of the secret that contains the token.
    - Login and password:
      - `USER` — a database user name.
      - `PASSWORD_SECRET_NAME` — the name of the secret that contains the password for the source database user.

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_NAME = '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_NAME = '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_NAME = '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_NAME = 'my_secret'
);
```

See also

- ALTER ASYNC REPLICATION
- DROP ASYNC REPLICATION

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 OBJECT (TYPE SECRET)

> **Warning**
>
> The syntax for managing secrets will change in future YDB releases.

The `CREATE OBJECT (TYPE SECRET)` statement creates a secret.

Syntax

```
CREATE OBJECT <secret_name> (TYPE SECRET) WITH value="<secret_value>";
```

Parameters

- `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 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.

Database Object Naming Rules

Every scheme object in YDB has a name. In YQL statements, object names are specified by identifiers that can be enclosed in backticks or not. For more information on identifiers, refer to Keywords and identifiers.

Scheme object names in YDB must meet the following requirements:

- Object names can include the following characters:
  - Uppercase Latin characters
  - Lowercase Latin characters
  - Digits
  - Special characters: `.`, `-`, and `_`
- Object name length must not exceed 255 characters.
- Objects cannot be created in folders which names start with a dot, such as `.sys`, `.metadata`, and `.sys_health`.

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 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).

Database Object Naming Rules

Every scheme object in YDB has a name. In YQL statements, object names are specified by identifiers that can be enclosed in backticks or not. For more information on identifiers, refer to Keywords and identifiers.

Scheme object names in YDB must meet the following requirements:

- Object names can include the following characters:
  - Uppercase Latin characters
  - Lowercase Latin characters
  - Digits
  - Special characters: `.`, `-`, and `_`
- Object name length must not exceed 255 characters.
- Objects cannot be created in folders which names start with a dot, such as `.sys`, `.metadata`, and `.sys_health`.

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).

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: `18h`.
- `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. `0` is interpreted as unlimited. Value type: `integer`, default value: `0`
- `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`.

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**
>
> Supported only for row-oriented tables. Support for column-oriented tables is currently under development.
>
> Instead of using `DELETE FROM` to delete data from colum-oriented tables, you can use the mechanism of deleting rows by time — TTL. TTL can be set when creating the table via `CREATE TABLE` or modified later via `ALTER TABLE`.

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;
```

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_NAME = '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 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 TABLE

Deletes the specified table.

If there is no such table, an error is returned.

Examples

```
DROP TABLE my_table;
```

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 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
`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;
```

GROUP BY

Group the `SELECT` results by the values of the specified columns or expressions. `GROUP BY` is often combined with aggregate functions (`COUNT`, `MAX`, `MIN`, `SUM`, `AVG`) to perform calculations in each group.

If `GROUP BY` is present in the query, then when selecting columns (between `SELECT ... FROM`), you can use the following constructs:

1. Columns by which grouping is performed (included in the `GROUP BY` argument).
2. Aggregate functions (see the next section). Columns that **are not** used for grouping can only be included as arguments for an aggregate function.
3. Functions that return the start and end times of the current window (`HOP_START` and `HOP_END`).
4. Arbitrary calculations combining items 1–3.

You can group by the result of an arbitrary expression computed from the source columns. In this case, to access the result of this expression, we recommend assigning a name to it 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 as 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 expression, you can set a name for it using AS,
                              -- and use it in SELECT
```

The query in the format `SELECT * FROM table GROUP BY k1, k2, ...` returns all columns listed in GROUP BY, i.e., is equivalent to `SELECT DISTINCT k1, k2, ... FROM table`.

An asterisk can also be used as an argument for the `COUNT` aggregate function. `COUNT(*)` means "the count of rows in the group".

> **Note**
>
> Aggregate functions ignore `NULL` in their arguments, except for `COUNT`.

YQL also provides aggregation factories implemented by 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: A key column
   COUNT(*) AS group_size,              -- OK: COUNT(*)
   SUM(key + subkey) AS sum1,           -- OK: An aggregate function
   CAST(SUM(1 + 2) AS String) AS sum2,  -- OK: An aggregate function with a constant argument
   SUM(SUM(1) + key) AS sum3,           -- ERROR: Nested aggregations are not allowed
   key AS k1,                           -- ERROR: Using a non-key column named key without aggregation
   key * 2 AS dk1,                      -- ERROR in YQL: using a non-key column named key without aggregation
FROM my_table
GROUP BY
  key * 2 AS double_key,
  subkey as sk,
```

Warning

Specifying a name for a column or expression in `GROUP BY .. AS foo` it is an extension on top of YQL. Such a name becomes visible in `WHERE` despite the fact that filtering by `WHERE` is executed before the grouping. For example, if the `T` table includes two columns, `foo` and `bar`, then the query `SELECT foo FROM T WHERE foo > 0 GROUP BY bar AS foo` would actually filter data by the `bar` column from the source table.

GROUP BY ... SessionWindow()

YQL supports grouping by session. To standard expressions in `GROUP BY`, you can add a special `SessionWindow` function:

```
SELECT
  user,
  session_start,
  SessionStart() AS same_session_start, -- It's 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 in this case:

1. The input table is partitioned by the grouping keys specified in `GROUP BY`, ignoring SessionWindow (in this case, it's based on `user`).

   If `GROUP BY` includes nothing more than SessionWindow, then the input table gets into one partition.
2. Each partition is split into disjoint subsets of rows (sessions).

   For this, the partition is sorted in the ascending order of the `time_expr` expression.
   The session limits are drawn between neighboring items of the partition, that differ in their `time_expr` values by more than `timeout_expr`.
3. The sessions obtained in this way are the final partitions on which aggregate functions are calculated.

The `SessionWindow()` key column (in the example, it's `session_start`) has the value "the minimum `time_expr` in the session".
If `GROUP BY` includes SessionWindow(), you can use a special aggregate function SessionStart.

An extended version of SessionWindow with four arguments is also supported:

`SessionWindow(<order_expr>, <init_lambda>, <update_lambda>, <calculate_lambda>)`

Where:

- `<order_expr>`: An expression used to sort the source partition
- `<init_lambda>`: A lambda function to initialize the state of session calculation. It has the signature `(TableRow())->State`. It's called once for the first (following the sorting order) element of the source partition
- `<update_lambda>`: A lambda function to update the status of session calculation and define the session limits. It has the signature `(TableRow(), State)->Tuple<Bool, State>`. It's called for every item of the source partition, except the first one. The new value of state is calculated based on the current row of the table and the previous state. If the first item in the return tuple is `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 item in the tuple.
- `<calculate_lambda>`: A lambda function for calculating the session key (the "value" of SessionWindow() that is also accessible via SessionStart()). The function has the signature `(TableRow(), State)->SessionKey`. It's called for the first item in the partition (after `<init_lambda>`) and those items for which `<update_lambda>` has returned `True` in the first item in the tuple. Please note that to start a new session, you should make sure that `<calculate_lambda>` has returned a value different from the previous session key. Sessions having the same keys are not merged. For example, if `<calculate_lambda>` returns the sequence `0, 1, 0, 1`, then there will be four different sessions.

Using the extended version of SessionWindow, you can, for example, do the following: divide a partition into sessions, as in the SessionWindow use case with two arguments, but with the maximum session length limited by a certain constant:

Example

```
$max_len = 1000; -- is the maximum session length.
$timeout = 100; -- is the timeout (timeout_expr in a simplified version of SessionWindow).

$init = ($row) -> (AsTuple($row.ts, $row.ts)); -- is the session status: tuple from 1) value of the temporary
column ts in the session's first line and 2) in the current line
$update = ($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, -- It's 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
```

You can use `SessionWindow` in `GROUP BY` only once.

ROLLUP, CUBE, and GROUPING SETS

The results of calculating the aggregate function as subtotals for the groups and overall totals over individual columns or whole table.

Syntax

```
SELECT
    c1, c2, -- the columns to group by

AGGREGATE_FUNCTION(c3) AS outcome_c  -- an aggregate function (SUM, AVG, MIN, MAX, COUNT)

FROM table_name

GROUP BY
    GROUP_BY_EXTENSION(c1, c2)       -- an extension of GROUP BY: ROLLUP, CUBE, or GROUPING SETS
```

- `ROLLUP` groups the column values in the order they are listed in the arguments (strictly from left to right), generates subtotals for each group and the overall total.
- `CUBE` groups the values for every possible combination of columns, generates the subtotals for each group and the overall total.
- `GROUPING SETS` sets the groups for subtotals.

You can combine `ROLLUP`, `CUBE` and `GROUPING SETS`, separating them by commas.

GROUPING

The values of columns not used in calculations are replaced with `NULL` in the subtotal. In the overall total, the values of all columns are replaced by `NULL`. `GROUPING`: A function that allows you to distinguish the source `NULL` values from the `NULL` values added while calculating subtotals and overall totals.

`GROUPING` returns a bit mask:

- `0`: If `NULL` is used for the original empty value.
- `1`: If `NULL` is added for a subtotal or overall 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 here (column2) as well, then together
        -- the ROLLUP and GROUPING SETS would produce a result
        -- similar to CUBE
    )
;
```

DISTINCT

Applying aggregate functions only to distinct values of the column.

> **Note**
>
> Applying `DISTINCT` to calculated values is not currently implemented. For this purpose, you can use a subquery or the expression `GROUP BY ... AS ...`.

Example

```
SELECT
  key,
  COUNT (DISTINCT value) AS count -- top 3 keys by the number of unique values
FROM my_table
GROUP BY key
ORDER BY count DESC
LIMIT 3;
```

You can also use `DISTINCT` to fetch distinct rows using `SELECT DISTINCT`.

GROUP COMPACT BY

Improves aggregation efficiency if the query author knows in advance that none of aggregation keys finds large amounts of data (i.e., with the order of magnitude exceeding a gigabyte or a million of rows). If this assumption fails to materialize, then the operation may fail with Out of Memory error or start running much slower compared to the non-COMPACT version.

Unlike the usual GROUP BY, the Map-side combiner stage and additional Reduce are disabled for each field with DISTINCT aggregation.

Example

```
SELECT
  key,
  COUNT (DISTINCT value) AS count -- top 3 keys by the number of unique values
FROM my_table
GROUP COMPACT BY key
ORDER BY count DESC
LIMIT 3;
```

GROUP BY ... HOP

Group the table by the values of the specified columns or expressions and subsets by time (the time window).

Among the columns used for grouping, make sure to use the `HOP` construct to define the time window for grouping.

```
HOP(time_extractor, hop, interval, delay)
```

The implemented version of the time window is called the **hopping window**. This is a window that moves forward in discrete intervals (the `hop` parameter). The total duration of the window is set by the `interval` parameter. To determine the time of each input event, the `time_extractor` parameter is used. This expression depends only on the input values of the columns and must have the `Timestamp` type. It specifies where to extract the time value from data.

The following happens in this case:

1. The input table is partitioned by the grouping keys specified in `GROUP BY`, ignoring HOP. If `GROUP BY` includes nothing more than HOP, then the input table gets into one partition.
2. Each partition is sorted in ascending order of the expression `time_extractor`.
3. Each partition is split into subsets of rows (possibly intersecting), on which aggregate functions are calculated.

In each partition defined by the values of all the grouping columns, the window moves forward independently of other streams. The advancement of the window depends entirely on the latest event in the partition.

The `interval` and `delay` parameters must be multiples of the `hop` parameter. Non-multiple intervals are prohibited in the current implementation.
The `interval` and `hop` parameters must be positive.

The `delay` parameter is ignored in the current implementation because the data in one partition is already sorted.

To set `hop`, `interval`, and `delay`, use a string expression compliant with ISO 8601. This format is used to construct the built-in `Interval` type from a string.

When selecting columns (between `SELECT ... FROM`) you can use the `HOP_START` and `HOP_END` functions (without parameters), which return a value of `Timestamp` type, corresponding to the start and end of the current window.

The **tumbling window**, known in other systems, is a special case of a **hopping window** where `interval` == `hop`.

Examples

```
SELECT
    key,
    COUNT(*)
FROM my_table
GROUP BY
    HOP(CAST(subkey AS Timestamp), "PT10S", "PT1M", "PT30S"),
    key;
-- hop = 10 seconds
-- interval = 1 minute
-- delay = 30 seconds
```

```
SELECT
    double_key,
    HOP_END() as time,
    COUNT(*) as count
FROM my_table
GROUP BY
    key + key AS double_key,
    HOP(ts, "PT1M", "PT1M", "PT1M");
```

HAVING

Filtering a `SELECT` based on the calculation results of aggregate functions. The syntax is similar to WHERE.

Example

```
SELECT
    key
FROM my_table
GROUP BY key
HAVING COUNT(value) > 100;
```

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;
```

INSERT INTO

Warning

Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Available methods for loading data into columnar tables:

- YDB CLI
- Bulk data upsert
- Yandex Data Transfer

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;
```

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`;
```

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  ...
```

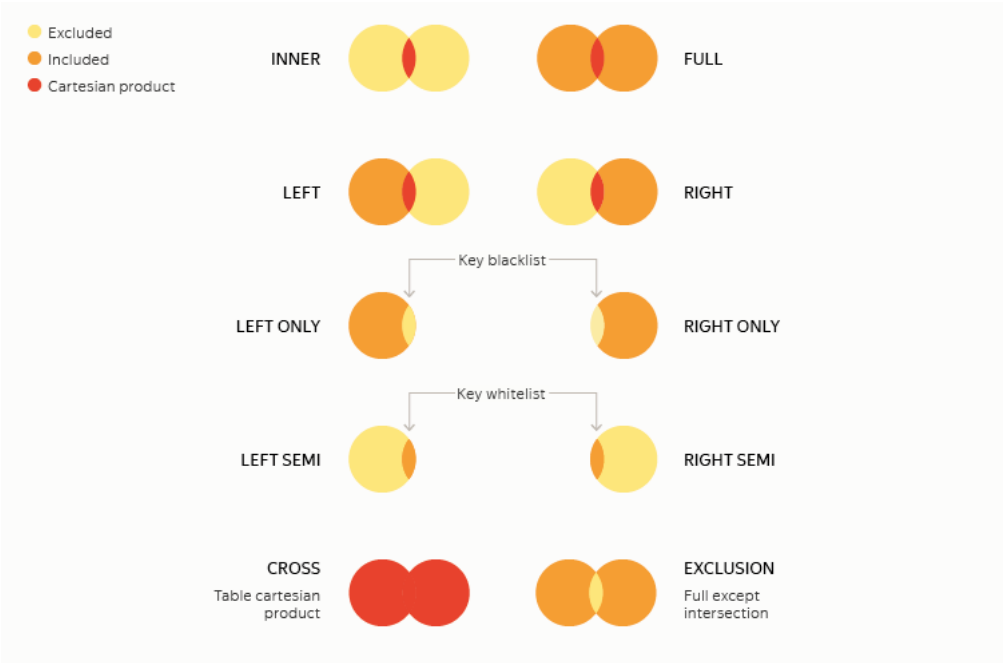
At each JOIN step, rules are used to establish correspondences between rows in the left and right data subqueries, creating a new subquery that includes every combination of rows that meet the JOIN conditions.

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.



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;
```

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 made for key columns: the query `SELECT * FROM T1 AS a JOIN T2 AS b USING(key)` in the SimpleColumns mode works same as `SELECT a.key ?? b.key AS key, ... FROM T1 AS a JOIN T2 AS b USING(key)`.

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

Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Available methods for loading data into columnar tables:

- YDB CLI
- Bulk data upsert
- Yandex Data Transfer

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 [, ...]} FROM
{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
`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;
```

UPDATE

Warning

Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Available methods for loading data into columnar tables:

- YDB CLI
- Bulk data upsert
- Yandex Data Transfer

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;
```

UPSERT INTO

Warning

Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Available methods for loading data into columnar tables:

- YDB CLI
- Bulk data upsert
- Yandex Data Transfer

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'))
```

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`.

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.

Classic SQL constructs not supported yet

[NOT] [EXISTS|INTERSECT|EXCEPT]

A syntactically available alternative is `EXISTS`, but it's not very useful as it doesn't support correlated subqueries. You can also rewrite it using `JOIN`.

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.

ALTER TABLE

Using the `ALTER TABLE` command, you can modify the columns and additional parameters of row and column tables. Multiple actions can be specified in a single command. Generally, the `ALTER TABLE` command looks like this:

```
ALTER TABLE table_name action1, action2, ..., actionN;
```

An action is any modification to the table, as described below:

- Renaming the table.
- Managing columns of row and column tables.
- Adding or removing a changefeed.
- Managing indexes.
- Managing column groups of a row table.
- Modifying additional table parameters.

Adding, removing, and renaming a index

Warning

Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Alert

The functionality of vector indexes is available in the test mode in main. This functionality will be fully available in version 25.1.

The following features are not supported:

- modifying rows in tables with vector indexes
- building an index for vectors with bit quantization

These limitations may be removed in future versions.

Adding an index

`ADD INDEX` : Adds an index with the specified name and type for a given set of columns. The code below adds a global index named `title_index` for the `title` column.

```
ALTER TABLE `series` ADD INDEX `title_index` GLOBAL ON (`title`);
```

You can specify any secondary index parameters from the `CREATE TABLE` command.
You can specify any vector index parameters from the `CREATE TABLE` command.

You can also add a secondary index using the YDB CLI table index command.

Altering an index

Indexes have type-specific parameters that can be tuned. Global indexes, whether synchronous or asynchronous, are implemented as hidden tables, and their automatic partitioning and followers settings can be adjusted just like those of regular tables.

Note

Currently, specifying secondary index partitioning settings during index creation is not supported in either the `ALTER TABLE ADD INDEX` or the `CREATE TABLE INDEX` statements.

```
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>`: The name of the table whose index is to be modified.
- `<index_name>`: The name of the index to be modified.
- `<setting_name>`: The name of the setting to be modified, which should be one of the following:
  - 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

Note

These settings cannot be reset.

- `<value>`: The new value for the setting. Possible values include:
  - `ENABLED` or `DISABLED` for the `AUTO_PARTITIONING_BY_SIZE` and `AUTO_PARTITIONING_BY_LOAD` settings
  - `"PER_AZ:<count>"` or `"ANY_AZ:<count>"` where `<count>` is the number of replicas for the `READ_REPLICAS_SETTINGS`
  - An integer of `Uint64` type for the other settings

Example

The query in the following example enables automatic partitioning by load for the index named `title_index` of the table `series`, sets its minimum partition count to 5, and enables one follower per AZ for every 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"
);
```

Deleting an index

`DROP INDEX`: Deletes the index with the specified name. The code below deletes the index named `title_index`.

```
ALTER TABLE `series` DROP INDEX `title_index`;
```

You can also remove a index using the YDB CLI table index command.

Renaming an index

`RENAME INDEX`: Renames the index with the specified name.

If an index with the new name exists, an error is returned.

Replacement of atomic indexes under load is supported by the command ydb table index rename in the YDB CLI and by YDB SDK ad-hoc methods.

Example of index renaming:

```
ALTER TABLE `series` RENAME INDEX `title_index` TO `title_index_new`;
```

Changing the composition of columns

YDB supports adding columns to row and column tables, as well as deleting non-key columns from tables.

`ADD COLUMN` — adds a column with the specified name and type. The code below will add a column named `views` with data type `Uint64` to the `episodes` table.

```
ALTER TABLE episodes ADD COLUMN views Uint64;
```

`DROP COLUMN` — deletes a column with the specified name. The code below will delete the column named `views` from the `episodes` table.

```
ALTER TABLE episodes DROP COLUMN views;
```

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.
- `RETENTION_PERIOD`: Record retention period. The value type is `Interval` and the default value is 24 hours (`Interval('PT24H')`).
- `TOPIC_MIN_ACTIVE_PARTITIONS`: The number of topic partitions. By default, the number of topic partitions is equal to the number of table partitions.
- `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
);
```

Example of creating a changefeed with initial scan:

```
ALTER TABLE `series` ADD CHANGEFEED `updates_feed` WITH (
    FORMAT = 'JSON',
    MODE = 'UPDATES',
    INITIAL_SCAN = TRUE
);
```

`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

> **Warning**
>
> Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

```
ALTER TABLE old_table_name RENAME TO new_table_name;
```

See table and column naming rules

Database Object Naming Rules

Every scheme object in YDB has a name. In YQL statements, object names are specified by identifiers that can be enclosed in backticks or not. For more information on identifiers, refer to Keywords and identifiers.

Scheme object names in YDB must meet the following requirements:

- Object names can include the following characters:
  - Uppercase Latin characters
  - Lowercase Latin characters
  - Digits
  - Special characters: `.`, `-`, and `_`
- Object name length must not exceed 255 characters.
- Objects cannot be created in folders which names start with a dot, such as `.sys`, `.metadata`, and `.sys_health`.

Column Naming Rules

Column names in YDB must meet the following requirements:

- Column names can include the following characters:
  - Uppercase Latin characters
  - Lowercase Latin characters
  - Digits
  - Special characters: `-` and `_`
- Column names must not start with the system prefix `__ydb_`.

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`;
```

Changing column groups

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

Alert

Supported only for row-oriented tables.

The code below changes the storage type to `hdd` for the `default` column group in the `series_with_families` table:

```
ALTER TABLE series_with_families ALTER FAMILY default SET DATA "hdd";
```

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 compression level of codec

Alert

Supported only for column-oriented tables.

The code below changes the compression level of codec if it supports different compression levels for the `default` column group in the `series_with_families` table:

```
ALTER TABLE series_with_families ALTER FAMILY default SET COMPRESSION_LEVEL 5;
```

You can specify any parameters of a group of columns from the `CREATE TABLE` command.

CREATE TABLE

CREATE TABLE syntax

The invocation of `CREATE TABLE` creates a table with the specified data schema and primary key columns (`PRIMARY KEY`). It also allows defining secondary indexes on the created table.

```
CREATE [TEMP | TEMPORARY] TABLE table_name (
    column1 type1,
    column2 type2 NOT NULL,
    ...
    columnN typeN,
    INDEX index1_name GLOBAL ON ( column ),
    INDEX index2_name GLOBAL ON ( column1, column2, ... ),
    PRIMARY KEY ( column, ... ),
    FAMILY column_family ( family_options, ... )
)
WITH ( key = value, ... )
```

YDB supports two types of tables:

- Row-oriented tables.
- Column-oriented tables.

The table type is specified by the `STORE` parameter in the `WITH` clause, where `ROW` indicates a row-oriented table and `COLUMN` indicates a 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.

Database Object Naming Rules

Every scheme object in YDB has a name. In YQL statements, object names are specified by identifiers that can be enclosed in backticks or not. For more information on identifiers, refer to Keywords and identifiers.

Scheme object names in YDB must meet the following requirements:

- Object names can include the following characters:
  - Uppercase Latin characters
  - Lowercase Latin characters
  - Digits
  - Special characters: `.`, `-`, and `_`
- Object name length must not exceed 255 characters.
- Objects cannot be created in folders which names start with a dot, such as `.sys`, `.metadata`, and `.sys_health`.

Column Naming Rules

Column names in YDB must meet the following requirements:

- Column names can include the following characters:
  - Uppercase Latin characters
  - Lowercase Latin characters
  - Digits
  - Special characters: `-` and `_`
- Column names must not start with the system prefix `__ydb_`.

Examples of table creation

Creating a row-oriented table

```
CREATE TABLE <table_name> (
  a Uint64,
  b Uint64,
  c Float,
```

```
  PRIMARY KEY (a, b)
);
```

For both key and non-key columns, only primitive data types are allowed.

Without additional modifiers, a column acquires an optional type and allows `NULL` values. To designate a non-optional type, use the `NOT NULL` constraint.

Specifying a `PRIMARY KEY` with a non-empty list of columns is mandatory. These columns become part of the key in the order they are listed.

Example of creating a row-oriented table using 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
);
```

Such code will create a row-oriented table with automatic partitioning by partition size (`AUTO_PARTITIONING_BY_SIZE`) enabled, and with the preferred size of each partition (`AUTO_PARTITIONING_PARTITION_SIZE_MB`) set to 512 megabytes. The full list of row-oriented table partitioning options can be found in the Partitioning Row-Oriented Tables section.

Creating a column-oriented 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
);
```

Example of creating a column-oriented table with an option to specify the minimum physical number of partitions for storing data:

```
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 columnar table with 10 partitions. The full list of column-oriented table partitioning options can be found in the Partitioning Column-Oriented Tables section.

When creating row-oriented tables, it is possible to specify:

- A secondary index.
- A vector index.
- Column groups.
- Additional parameters.

When creating column-oriented tables, it is possible to specify:

- Column groups.
- Additional parameters.

INDEX

> **Warning**
>
> Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

The INDEX construct is used to define a secondary index in a row-oriented table:

```
CREATE TABLE table_name (
    ...
    INDEX <index_name> GLOBAL [UNIQUE] [SYNC|ASYNC] [USING <index_type>] ON ( <index_columns> ) COVER (
<cover_columns> ),
    ...
)
```

Where:

- **Index_name** is the unique name of the index to be used to access data.
- **SYNC/ASYNC** indicates synchronous/asynchronous data writes to the index. If not specified, synchronous.
- **Index_type** is the index type. Only `secondary` is supported now.
- **UNIQUE** indicates that the index should guarantee the uniqueness of the indexed column set, thereby enforcing the unique constraint.
- **Index_columns** is a list of comma-separated names of columns in the created table to be used for a search in the index.
- **Cover_columns** is a list of comma-separated names of columns in the created table, which will be stored in the index in addition to the search columns, making it possible to fetch additional data without accessing the table for it.

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_bc GLOBAL UNIQUE SYNC ON (b, c),
    PRIMARY KEY (a)
)
```

Vector index

Warning

Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Alert

The functionality of vector indexes is available in the test mode in main. This functionality will be fully available in version 25.1.

The following features are not supported:

- modifying rows in tables with vector indexes
- building an index for vectors with bit quantization

These limitations may be removed in future versions.

Warning

It makes no sense to create an empty table with a vector index, because for now we don't allow mutations in tables with vector indexes.

You should use `ALTER TABLE ... ADD INDEX` command) to add a vector index to an existing table.

The INDEX construct is used to define a vector index in a row-oriented table:

```
CREATE TABLE table_name (
    ...
    INDEX <index_name> GLOBAL [SYNC] USING <index_type> ON ( <index_columns> ) COVER ( <cover_columns> ) WITH (
<index_parameters> ),
    ...
)
```

Where:

- **Index_name** is the unique name of the index to be used to access data.
- **SYNC** indicates synchronous data writes to the index. If not specified, synchronous.
- **Index_type** is the index type. Only `vector_kmeans_tree` is supported now.
- **Index_columns** is a list of comma-separated column names in the created table to be used for a search in the index. The last column in the list is used as embedding, the other columns are used as prefix columns.
- **Cover_columns** is a list of comma-separated column names in the created table, which will be stored in the index in addition to the search columns, making it possible to fetch additional data without accessing the table for it.
- **Index_parameters** is a list of comma-separated key-value parameters:
  - parameters for any vector **index_type**:
    - `vector_dimension` is a number of dimension in the indexed embedding (<= 16384)
    - `vector_type` is a type of value in the indexed embedding, can be `float`, `uint8`, `int8`, `bit`
    - `distance` is a type of the distance function which will be used for this index. Valid values: `cosine`, `manhattan`, `euclidean`.
    - `similarity` is a type of the similarity function which will be used for this index. Valid values: `inner_product`, `cosine`.
  - parameters specific to `vector_kmeans_tree`:
    - `clusters` is a `k` in each kmeans used for tree (values > 1000 can affect performance)
    - `levels` is a level count in the tree

Warning

The `distance` and `similarity` parameters can not be specified together.

Warning

The `vector_type=bit` vector index is not supported yet.

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),
    PRIMARY KEY (article_id)
)
```

Column groups

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`.

> **Alert**
>
> Supported only for row-oriented tables.

- `COMPRESSION`: A data compression codec. Acceptable values: `off`, `lz4`, `zstd`.

> **Alert**
>
> Codec `"zstd"` is supported only for column-oriented tables.

- `COMPRESSION_LEVEL` — compression level of codec if it supports different compression levels.

> **Alert**
>
> Supported only for column-oriented tables.

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.

Creating a row-oriented table

```
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"
    ),
    FAMILY family_large (
        DATA = "rot",
        COMPRESSION = "lz4"
    )
);
```

Creating a column-oriented table

```
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 (
        COMPRESSION = "lz4"
    ),
    FAMILY family_large (
        COMPRESSION = "zstd",
        COMPRESSION_LEVEL = 5
    )
)
WITH (STORE = COLUMN);
```

Note

Available types of storage devices depend on the YDB cluster configuration.

Creation of temporary tables (TEMPORARY)

> **Warning**
>
> Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

`TEMPORARY` / `TEMP` – a temporary table that is automatically deleted at the end of the session. If this parameter is not set (left empty), a permanent table is created. Any indexes created on a temporary table will also be deleted at the end of the session, which means that they are temporary as well. A temporary table and a permanent table with the same name are allowed, in which case a temporary table will be selected.

```
CREATE TEMPORARY TABLE table_name (
    ...
);
```

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
);
```

SELECT

Returns the result of evaluating the expressions specified after `SELECT`.

It can be used in combination with other operations to obtain other effect.

Examples

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

```
SELECT 2 + 2;
```

SELECT execution procedure

The `SELECT` query result is calculated as follows:

- Determine the set of input tables by evaluating the FROM clauses.
- Apply MATCH_RECOGNIZE to input tables.
- Evaluate SAMPLE/TABLESAMPLE.
- Execute FLATTEN COLUMNS or FLATTEN BY; aliases set in `FLATTEN BY` become visible after this point.
- Execute every JOIN.
- Add to (or replace in) the data the columns listed in GROUP BY ... AS ....
- Execute WHERE — Discard all the data mismatching the predicate.
- Execute GROUP BY, evaluate aggregate functions.
- Apply the filter HAVING.
- Evaluate window functions;
- Evaluate expressions in `SELECT`.
- Assign names set by aliases to expressions in `SELECT`.
- Apply top-level DISTINCT to the resulting columns.
- Execute similarly every subquery inside UNION ALL, combine them (see PRAGMA AnsiOrderByLimitInUnionAll).
- Perform sorting with ORDER BY.
- Apply OFFSET and LIMIT to the result.

Column order in YQL

The standard SQL is sensitive to the order of columns in projections (that is, in `SELECT`). While the order of columns must be preserved in the query results or when writing data to a new table, some SQL constructs use this order.
This applies, for example, to UNION ALL and positional ORDER BY (ORDER BY ordinal).

The column order is ignored in YQL by default:

- The order of columns in the output tables and query results is undefined
- The data scheme of the `UNION ALL` result is output by column names rather than positions

If you enable `PRAGMA OrderedColumns;`, the order of columns is preserved in the query results and is derived from the order of columns in the input tables using the following rules:

- `SELECT`: an explicit column enumeration dictates the result order.
- `SELECT` with an asterisk (`SELECT * FROM ...`) inherits the order from its input.
- The order of columns after JOIN: First output the left-hand columns, then the right-hand ones. If the column order in any of the sides in the `JOIN` output is undefined, the column order in the result is also undefined.
- The order in `UNION ALL` depends on the UNION ALL execution mode.
- The column order for AS_TABLE is undefined.

Warning

In the YT table schema, key columns always precede non-key columns. The order of key columns is determined by the order of the composite key.
When `PRAGMA OrderedColumns;` is enabled, non-key columns preserve their output order.

Combining queries

Results of several SELECT statements (or subqueries) can be combined using `UNION` and `UNION ALL` keywords.

```
query1 UNION [ALL] query2 (UNION [ALL] query3 ...)
```

Union of more than two queries is interpreted as a left-associative operation, that is

```
query1 UNION query2 UNION ALL query3
```

is interpreted as

```
(query1 UNION query2) UNION ALL query3
```

If the underlying queries have one of the `ORDER BY/LIMIT/DISCARD/INTO RESULT` operators, the following rules apply:

- `ORDER BY/LIMIT/INTO RESULT` is only allowed after the last query
- `DISCARD` is only allowed before the first query
- the operators apply to the `UNION [ALL]` as a whole, instead of referring to one of the queries
- to apply the operator to one of the queries, enclose the query in parantheses

Clauses supported in SELECT

- FROM
- FROM AS_TABLE
- FROM SELECT
- DISTINCT
- UNIQUE DISTINCT
- UNION
- WITH
- WITHOUT
- WHERE
- ORDER BY
- ASSUME ORDER BY
- LIMIT OFFSET
- SAMPLE
- TABLESAMPLE
- MATCH_RECOGNIZE
- VIEW secondary_index
- VIEW vector_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);
```

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 *;
```

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.

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)

> **Warning**
>
> Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

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)

> **Warning**
>
> Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

To make a `SELECT` by secondary index of row-oriented table statement, use the following:

```
SELECT *
    FROM TableName VIEW IndexName
    WHERE …
```

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)

Warning

Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

Alert

The functionality of vector indexes is available in the test mode in main. This functionality will be fully available in version 25.1.

The following features are not supported:

- modifying rows in tables with vector indexes
- building an index for vectors with bit quantization

These limitations may be removed in future versions.

To select data from a row-oriented table using a vector index, use the following statements:

```
SELECT ...
    FROM TableName VIEW IndexName
    WHERE ...
    ORDER BY Knn::SomeDistance(...)
    LIMIT ...
```

```
SELECT ...
    FROM TableName VIEW IndexName
    WHERE ...
    ORDER BY Knn::SomeSimilarity(...) DESC
    LIMIT ...
```

Note

The vector index will not be automatically selected by the optimizer, so it must be specified explicitly using the expression `VIEW IndexName'.

Examples

- Select all the fields from the `series` row-oriented table using the `views_index` vector index created for `embedding` and inner product similarity:

  ```
  SELECT series_id, title, info, release_date, views, uploaded_user_id, Knn::InnerProductSimilarity(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_index2` prefixed vector index created for `embedding` and inner product similarity with prefix column `release_date`:

  ```
  SELECT series_id, title, info, release_date, views, uploaded_user_id, Knn::InnerProductSimilarity(embedding,
  $target) as similarity
      FROM series VIEW views_index2
      WHERE release_date = "2025-03-31"
      ORDER BY Knn::InnerProductSimilarity(embedding, $TargetEmbedding) DESC
      LIMIT 10
  ```

WITH

It's set after the data source in `FROM` and is used for additional hints for row-oriented and column-oriented. You can't use hints for subqueries and named expressions.

The following values are supported:

- `INFER_SCHEMA`: Sets the flag for output of the table schema. The behavior is similar to the yt.inferSchema pragma, but for a specific data source. You can specify the number of rows to output (from 1 to 1000).
- `FORCE_INFER_SCHEMA`: Sets the flag for table schema output. The behavior is similar to the yt.ForceInferSchema pragma, but for a specific data source. You can specify the number of rows to output (from 1 to 1000).
- `DIRECT_READ`: Suppresses certain optimizers and enforces accessing table contents as is. The behavior is similar to the debug pragma DirectRead, but for a specific data source.
- `INLINE`: Hints that the table contents is small and you need to use its in-memory view to process the query. The actual size of the table is not controlled in this case, and if it's large, the query might fail with an out-of-memory error.
- `UNORDERED`: Suppresses original table sorting.
- `XLOCK`: Hints that you need to lock the table exclusively. It's useful when you read a table at the stage of processing the query metaprogram, and then update its contents in the main query. Avoids data loss if an external process managed to change the table between executing a metaprogram phase and the main part of the query.
- `SCHEMA` type: Hints that the specified table schema must be used entirely, ignoring the schema in the metadata.
- `COLUMNS` type: Hints that the specified types should be used for columns whose names match the table's column names in the metadata, as well as which columns are additionally present in the table.
- `IGNORETYPEV3`, `IGNORE_TYPE_V3`: Sets the flag to ignore type_v3 types in the table. The behavior is similar to the yt.IgnoreTypeV3 pragma, but for a specific data source.

When setting the `SCHEMA` and `COLUMNS` hints, the type must be a structure.

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>>;
```

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;
```

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 clicks,
and one button 3 click
    DEFINE
        B1 AS B1.button = 1, -- Defining the B1 variable as event of clicking button 1 (the button field equals 1)
        B2 AS B2.button = 2, -- Defining the B2 variable as event of clicking button 2 (the button field equals 2)
        B3 AS B3.button = 3  -- Defining the B3 variable as event of clicking button 3 (the button field equals 3)
);
```

Result:

b1	b3	device_id	zone_id
100	500	4	2
200	600	17	3

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.

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 digits, with 2 after the decimal point
```

```
  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("P1DT2H3M4.567890S"),
  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), -- "\u0001\u0000\u0000\u0000"
    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; -- actually, it is better to use TABLESAMPLE

$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]
```

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
```

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 this result, use in the subquery the built-in function IF with two arguments (to get `NULL` in else), and apply an outer COUNT(DISTINCT ...) to its result.

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 –
interval
```

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 Number
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

In the DateTime module, the main internal representation format is `Resource<TM>`, which stores the following date components:

- Year (12 bits).
- 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 primitive type to an internal representation. It's always successful on a non-empty input.

List of functions

- `DateTime::Split(Date{Flags:AutoMap}) -> Resource<TM>`
- `DateTime::Split(Datetime{Flags:AutoMap}) -> Resource<TM>`
- `DateTime::Split(Timestamp{Flags:AutoMap}) -> Resource<TM>`
- `DateTime::Split(TzDate{Flags:AutoMap}) -> Resource<TM>`
- `DateTime::Split(TzDatetime{Flags:AutoMap}) -> Resource<TM>`
- `DateTime::Split(TzTimestamp{Flags:AutoMap}) -> Resource<TM>`

Functions that accept `Resource<TM>` as input, can be called directly from the primitive date/time type. An implicit conversion will be made in this case by calling a relevant `Split` function.

Make...

Making a primitive type from an internal representation. It's always successful on a non-empty input.

List of functions

- `DateTime::MakeDate(Resource<TM>{Flags:AutoMap}) -> Date`
- `DateTime::MakeDatetime(Resource<TM>{Flags:AutoMap}) -> Datetime`
- `DateTime::MakeTimestamp(Resource<TM>{Flags:AutoMap}) -> Timestamp`
- `DateTime::MakeTzDate(Resource<TM>{Flags:AutoMap}) -> TzDate`
- `DateTime::MakeTzDatetime(Resource<TM>{Flags:AutoMap}) -> TzDatetime`
- `DateTime::MakeTzTimestamp(Resource<TM>{Flags:AutoMap}) -> TzTimestamp`

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 an internal representation.

List of functions

- `DateTime::GetYear(Resource<TM>{Flags:AutoMap}) -> Uint16`
- `DateTime::GetDayOfYear(Resource<TM>{Flags:AutoMap}) -> Uint16`

- `DateTime::GetMonth(Resource<TM>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetMonthName(Resource<TM>{Flags:AutoMap}) -> String`
- `DateTime::GetWeekOfYear(Resource<TM>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetWeekOfYearIso8601(Resource<TM>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetDayOfMonth(Resource<TM>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetDayOfWeek(Resource<TM>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetDayOfWeekName(Resource<TM>{Flags:AutoMap}) -> String`
- `DateTime::GetHour(Resource<TM>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetMinute(Resource<TM>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetSecond(Resource<TM>{Flags:AutoMap}) -> Uint8`
- `DateTime::GetMillisecondOfSecond(Resource<TM>{Flags:AutoMap}) -> Uint32`
- `DateTime::GetMicrosecondOfSecond(Resource<TM>{Flags:AutoMap}) -> Uint32`
- `DateTime::GetTimezoneId(Resource<TM>{Flags:AutoMap}) -> Uint16`
- `DateTime::GetTimezoneName(Resource<TM>{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:Uint8?, Second:Uint8?, Microsecond:Uint32?, Timezone:String? ]) -> Resource<TM>?
```

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($tm, "Europe/Moscow" as Timezone)), -- 2018-12-31T22:02:03.456789Z
(conversion to UTC)
    DateTime::MakeTzTimestamp(DateTime::Update($tm, "Europe/Moscow" as Timezone)); -- 2019-01-
01T01:02:03.456789,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::FromMilliseconds(Uint64{Flags:AutoMap}) -> Timestamp`
- `DateTime::FromMicroseconds(Uint64{Flags:AutoMap}) -> Timestamp`

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::ToMilliseconds(Date/DateTime/Timestamp/TzDate/TzDatetime/TzTimestamp{Flags:AutoMap}) -> Uint64`
- `DateTime::ToMicroseconds(Date/DateTime/Timestamp/TzDate/TzDatetime/TzTimestamp{Flags:AutoMap}) -> Uint64`

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}) -> Int16`
- `DateTime::ToHours(Interval{Flags:AutoMap}) -> Int32`
- `DateTime::ToMinutes(Interval{Flags:AutoMap}) -> Int32`
- `DateTime::ToSeconds(Interval{Flags:AutoMap}) -> Int32`
- `DateTime::ToMilliseconds(Interval{Flags:AutoMap}) -> Int64`
- `DateTime::ToMicroseconds(Interval{Flags:AutoMap}) -> Int64`
- `DateTime::IntervalFromDays(Int16{Flags:AutoMap}) -> Interval`
- `DateTime::IntervalFromHours(Int32{Flags:AutoMap}) -> Interval`
- `DateTime::IntervalFromMinutes(Int32{Flags:AutoMap}) -> Interval`
- `DateTime::IntervalFromSeconds(Int32{Flags:AutoMap}) -> Interval`
- `DateTime::IntervalFromMilliseconds(Int64{Flags:AutoMap}) -> Interval`
- `DateTime::IntervalFromMicroseconds(Int64{Flags:AutoMap}) -> Interval`

AddTimezone doesn't affect the output of ToSeconds() in any way, because ToSeconds() always returns GMT time.

You can also create an Interval from a string literal in the format ISO 8601. Time units up to a week are supported, inclusive.

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::EndOfYear(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::StartOfQuarter(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::EndOfQuarter(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::StartOfMonth(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::EndOfMonth(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::StartOfWeek(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::EndOfWeek(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::StartOfDay(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::EndOfDay(Resource<TM>{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::StartOf(Resource<TM>{Flags:AutoMap}, Interval{Flags:AutoMap}) -> Resource<TM>?`
- `DateTime::EndOf(Resource<TM>{Flags:AutoMap}, Interval{Flags:AutoMap}) -> Resource<TM>?`

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`

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::ShiftQuarters(Resource<TM>{Flags:AutoMap}, Int32) -> Resource<TM>?`
- `DateTime::ShiftMonths(Resource<TM>{Flags:AutoMap}, Int32) -> Resource<TM>?`

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<TM>{Flags:AutoMap}) -> String`

A set of specifiers is implemented for the formatting string:

- `%%`: % character.
- `%Y`: 4-digit year.

- `%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

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>?`

Implemented specifiers:

- `%%` : the % character.
- `%Y` : 4-digit year (1970).
- `%m` : 2-digit month (1).
- `%d` : 2-digit day (1).
- `%H` : 2-digit hour (0).
- `%M` : 2-digit minutes (0).
- `%S` : Seconds (0), can also accept microseconds in the formats from xx. up to xx.xxxxxx
- `%Z` : The IANA name of the timezone (GMT).
- `%b` : A short three-letter case-insensitive English name of the month (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");

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)
```

For the common formats, wrappers around the corresponding util methods are supported. You can only get TM with components in the UTC timezone.

Parse specific formats

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($datetime_parse_tz("2019-09-16 00:00:00" || " Europe/Moscow"))),  -
- 1568581200
    DateTime::ToSeconds(DateTime::MakeDatetime(DateTime::Update($datetime_parse("2019-09-16 00:00:00"),
"Europe/Moscow" as Timezone))), -- 1568581200

    -- INCORRECT (Date imports time as GMT, but AddTimezone has no effect on ToSeconds that always returns GMT
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). DateTime::ToSeconds() exports only to GMT. That's why we should put timezones aside for a while and use only GMT (as if we assumed for a while that Moscow is in GMT):

```
$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, -- 1568592000
    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, -- 1568581200
    DateTime::ToSeconds(DateTime::MakeDatetime($datetime_parse_tz("2019-09-16 00:00:00 Europe/Moscow"))) AS md_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
Europe/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 */
```

Starting from the Unix epoch, there is no valid value representing midnight on 01/01/1970 for the Europe/Moscow timezone. As a result, such a cast is impossible and fails at runtime.

At the same time, values with a negative timezone offset are converted correctly:

```
SELECT CAST(TzDatetime("1970-01-01T23:59:59,America/Los_Angeles") as TzDate);
/* 1970-01-01,America/Los_Angeles */
```

Daylight saving time

Please 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>?
```

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:
> CosineDistance, error: Arguments should have same tags, but 'FloatVector' is not equal to 'Uint8Vector'
> ```

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), "FloatVector"));
```

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), "FloatVector"),
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:Uint64?,NeverCa
->
Struct<CaseSensitive:Bool,DotNl:Bool,Literal:Bool,LogErrors:Bool,LongestMatch:Bool,MaxMem:Uint64,NeverCapture:Bool,Ne
```

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<roaring_bitmap>
Roaring::OrWithBinary(Resource<roaring_bitmap>{Flags:AutoMap}, String{Flags:AutoMap})->Resource<roaring_bitmap>

Roaring::And(Resource<roaring_bitmap>{Flags:AutoMap}, Resource<roaring_bitmap>{Flags:AutoMap})-
>Resource<roaring_bitmap>
Roaring::AndWithBinary(Resource<roaring_bitmap>{Flags:AutoMap}, String{Flags:AutoMap})->Resource<roaring_bitmap>

Roaring::AndNot(Resource<roaring_bitmap>{Flags:AutoMap}, Resource<roaring_bitmap>{Flags:AutoMap})-
>Resource<roaring_bitmap>
Roaring::AndNotWithBinary(Resource<roaring_bitmap>{Flags:AutoMap}, String{Flags:AutoMap})-
>Resource<roaring_bitmap>

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::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("ҫурт"); -- "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(99,111,100,101,32,112,111,105,110,116,115,32,99,111,110,118,101,114,116,101,114
-- "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, defaults 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
Max<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, defaults 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
Max<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}) ->
Resource<'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
ToBytes(...)
```

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

Connecting via PostgreSQL Protocol

Warning

At the moment, YDB's compatibility with PostgreSQL **is under development**, so not all PostgreSQL constructs and functions are supported yet. PostgreSQL compatibility is available for testing in the form of a Docker container, which can be deployed by following these instructions.

Running YDB with PostgreSQL compatibility enabled

Currently, the PostgreSQL compatibility feature is available for testing in the Docker image: `ghcr.io/ydb-platform/local-ydb:nightly`.

Commands for starting a local Docker container with YDB and open ports for PostgreSQL and Embedded UI:

Tip

In this example, the containers are intentionally created in a way that their state is deleted after they stop. This simplifies the instructions and allows you to repeatedly run tests in a known environment without worrying about past test failures.

To preserve the container's state, you need to remove the environment variable `YDB_USE_IN_MEMORY_PDISKS`.

Docker-compose

To launch using a Docker Compose configuration file, it must already be installed on your system.

docker-compose.yaml:

```
services:
    ydb:
        image: ghcr.io/ydb-platform/local-ydb:nightly
        ports:
        - "5432:5432"
        - "8765:8765"
        environment:
        - "YDB_USE_IN_MEMORY_PDISKS=true"
        - "POSTGRES_USER=${YDB_PG_USER:-root}"
        - "POSTGRES_PASSWORD=${YDB_PG_PASSWORD:-1234}"
        - "YDB_EXPERIMENTAL_PG=1"
```

Run:

```
docker-compose up -d --pull=always
```

Docker command

```
docker run --name ydb-postgres -d --pull always -p 5432:5432 -p 8765:8765 -e POSTGRES_USER=root -e
POSTGRES_PASSWORD=1234 -e YDB_EXPERIMENTAL_PG=1 -e YDB_USE_IN_MEMORY_PDISKS=true ghcr.io/ydb-platform/local-
ydb:nightly
```

After launching the container, you can connect to it via PostgreSQL clients on port 5432, the database `local`, or open the web interface on port 8765.

Connecting to the Running Container via psql

Upon executing this command, the interactive command-line interface of `psql`, the PostgreSQL client, will be launched. All subsequent queries should be entered within this client interface.

```
docker run --rm -it --network=host postgres:14 psql postgresql://root:1234@localhost:5432/local
```

Hello world example

```
SELECT 'Hello, world!';
```

Output:

```
    column0
---------------
 Hello, world!
(1 row)
```

Creating a Table

The primary purpose of database management systems is to store data for later retrieval. As an SQL-based system, the principal abstraction for storing data is the table. To create our first table, execute the following query:

```
CREATE TABLE example
(
    key int4,
    value text,
    PRIMARY KEY (key)
);
```

Adding test data

Now let's populate our table with some initial data. The simplest way to do this is by using literals.

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

Querying test data

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

Output:

```
 column0
---------
       2
(1 row)
```

Stopping the Container

This command will stop the running container and delete all data stored within it.

Docker-compose

In the directory containing the `docker-compose.yaml` file, execute the command that will stop the container and remove its data:

```
docker-compose down -vt 1
```

> **Note**
>
> To stop a Docker container and preserve its data, you should run it without the `YDB_USE_IN_MEMORY_PDISKS` environment variable and use the stop command:
>
> ```
> docker-compose stop
> ```

Docker command

This command will stop the container and remove the data:

```
docker rm -f ydb-postgres
```

PostgreSQL functions

> **Warning**
>
> At the moment, YDB's compatibility with PostgreSQL **is under development**, so not all PostgreSQL constructs and functions are supported yet. PostgreSQL compatibility is available for testing in the form of a Docker container, which can be deployed by following these instructions.

This section contains PostgreSQL functions supported in the YDB mode of compatibility with PostgreSQL. The original structure of the PostgreSQL documentation and examples of function application are preserved in the section. This article is automatically supplemented and tested.

9.1. Logical Operators

The usual logical operators are available:

- boolean AND boolean → boolean
- boolean OR boolean → boolean
- NOT boolean → boolean

9.2. Comparison Functions and Operators

The usual comparison operators are available, as shown in Table 9.1.

Table 9.1. Comparison Operators

Operator	Description
datatype < datatype → boolean	Less than
datatype > datatype → boolean	Greater than
datatype <= datatype → boolean	Less than or equal to
datatype >= datatype → boolean	Greater than or equal to
datatype = datatype → boolean	Equal
datatype <> datatype → boolean	Not equal
datatype != datatype → boolean	Not equal

There are also some comparison predicates, as shown in Table 9.2. These behave much like operators, but have special syntax mandated by the SQL standard.

Predicate	Description	Example(s)

datatype BETWEEN datatype AND datatype → boolean	Between (inclusive of the range endpoints).	`2 BETWEEN 1 AND 3 → true` `2 BETWEEN 3 AND 1 → false`
datatype NOT BETWEEN datatype AND datatype → boolean	Not between (the negation of BETWEEN).	`2 NOT BETWEEN 1 AND 3 → false`
datatype BETWEEN SYMMETRIC datatype AND datatype → boolean	Between, after sorting the two endpoint values.	`2 BETWEEN SYMMETRIC 3 AND 1 → true`
datatype NOT BETWEEN SYMMETRIC datatype AND datatype → boolean	Not between, after sorting the two endpoint values.	`2 NOT BETWEEN SYMMETRIC 3 AND 1 → false`

datatype IS DISTINCT FROM datatype → boolean	Not equal, treating null as a comparable value. (NOT SUPPORTED)	`#1 IS DISTINCT FROM NULL → true` `#NULL IS DISTINCT FROM NULL → false`
datatype IS NOT DISTINCT FROM datatype → boolean	Equal, treating null as a comparable value. (NOT SUPPORTED)	`#1 IS NOT DISTINCT FROM NULL → false` `#NULL IS NOT DISTINCT FROM NULL → true`
datatype IS NULL → boolean	Test whether value is null.	`1.5 IS NULL → false`
datatype IS NOT NULL → boolean	Test whether value is not null.	`'null' IS NOT NULL → true`
datatype ISNULL → boolean	Test whether value is null (nonstandard syntax)	`1.5 ISNULL → false`

datatype NOTNULL → boolean	Test whether value is not null (nonstandard syntax)	`1.5 NOTNULL → true`

Operator	Description	Example(s)
numeric_type + numeric_type → numeric_type	Addition	`2 + 3 → 5`
+ numeric_type → numeric_type	Unary plus (no operation)	`+ 3.5 → 3.5`
numeric_type - numeric_type → numeric_type	Subtraction	`2 - 3 → -1`
- numeric_type → numeric_type	Negation	`- (-4) → 4`

numeric_type * numeric_type → numeric_type	Multiplication	`2 * 3 → 6`

numeric_type / numeric_type → numeric_type	Division (for integral types, division truncates the result towards zero)	`5.0 / 2 → 2.5000000000000000`
		`5 / 2 → 2`
		`(-5) / 2 → -2`

numeric_type % numeric_type → numeric_type	Modulo (remainder); available for smallint, integer, bigint, and numeric	`5 % 4 → 1`

numeric ^ numeric → numeric
 	
double precision ^ double precision → double precision
	Exponentiation.
	
	Unlike typical mathematical practice, multiple uses of ^ will associate left to right by default.
		`2 ^ 3 → 8`
		`2 ^ 3 ^ 3 → 512`
		`2 ^ (3 ^ 3) → 134217728`

|/ double precision → double precision	Square root	`|/ 25.0 → 5`

||/ double precision → double precision	Cube root	`||/ 64.0 → 4`
@ numeric_type → numeric_type	Absolute value	`@ -5.0 → 5.0`
integral_type & integral_type → integral_type	Bitwise AND	`91 & 15 → 11`
integral_type | integral_type → integral_type	Bitwise OR	`32 | 3 → 35`
integral_type # integral_type → integral_type	Bitwise exclusive OR	`17 # 5 → 20`

~ integral_type → integral_type	Bitwise NOT	`~1 → -2`
integral_type << integer → integral_type	Bitwise shift left	`1 << 4 → 16`
integral_type >> integer → integral_type	Bitwise shift right	`8 >> 2 → 2`

Function	Description	Example(s)
abs (numeric_type) → numeric_type	Absolute value	`abs(-17.4) → 17.4`
cbrt (double precision) → double precision	Cube root	`cbrt(64.0) → 4`

ceil (numeric) → numeric ceil (double precision) → double precision	Nearest integer greater than or equal to argument	`ceil(42.2) → 43` `ceil(-42.8) → -42`
ceiling (numeric) → numeric ceiling (double precision) → double precision	Nearest integer greater than or equal to argument(same as ceil)	`ceiling(95.3) → 96`
degrees (double precision) → double precision	Converts radians to degrees	`degrees(0.5) → 28.64788975654116`
div (y numeric, x numeric) → numeric	Integer quotient of y/x (truncates towards zero)	`div(9, 4) → 2`
exp (numeric) → numeric exp (double precision) → double precision	Exponential (e raised to the given power)	`exp(1.0) → 2.7182818284590452`

factorial (bigint) → numeric	Factorial	`factorial(5) → 120`
floor (numeric) → numeric floor (double precision) → double precision	Nearest integer less than or equal to argument	`floor(42.8) → 42` `floor(-42.8) → -43`
gcd (numeric_type, numeric_type) → numeric_type	Greatest common divisor (the largest positive number that divides both inputs with no remainder); returns 0 if both inputs are zero; available for integer, bigint, and numeric	`gcd(1071, 462) → 21`
lcm (numeric_type, numeric_type) → numeric_type	Least common multiple (the smallest strictly positive number that is an integral multiple of both inputs); returns 0 if either input is zero; available for integer, bigint, and numeric	`lcm(1071, 462) → 23562`
ln (numeric) → numeric ln (double precision) → double precision	Natural logarithm	`ln(2.0) → 0.6931471805599453`

log (numeric) → numeric log (double precision) → double precision	Base 10 logarithm	`log(100) → 2`
log10 (numeric) → numeric log10 (double precision) → double precision	Base 10 logarithm (same as log)	`log10(1000) → 3`
log (b numeric, x numeric) → numeric	Logarithm of x to base b	`log(2.0, 64.0) → 6.0000000000000000`
min_scale (numeric) → integer	Minimum scale (number of fractional decimal digits) needed to represent the supplied value precisely	`min_scale(8.4100) → 2`
mod (y numeric_type, x numeric_type) → numeric_type	Remainder of y/x; available for smallint, integer, bigint, and numeric	`mod(9, 4) → 1`

pi ( ) → double precision	Approximate value of π	`pi() → 3.141592653589793`
power (a numeric, b numeric) → numeric power (a double precision, b double precision) → double precision	a raised to the power of b	`power(9, 3) → 729`
radians (double precision) → double precision	Converts degrees to radians	`radians(45.0) → 0.7853981633974483`
round (numeric) → numeric round (double precision) → double precision	Rounds to nearest integer. For numeric, ties are broken by rounding away from zero. For double precision, the tie-breaking behavior is platform dependent, but "round to nearest even" is the most common rule.	`round(42.4) → 42`

Function	Description	Example(s)
random ( ) → double precision	Returns a random value in the range 0.0 <= x < 1.0	
setseed (double precision) → void	Sets the seed for subsequent random() calls; argument must be between -1.0 and 1.0, inclusive (NOT SUPPORTED)	

Table 9.7 shows the available trigonometric functions. Each of these functions comes in two variants, one that measures angles in radians and one that measures angles in degrees.

Table 9.7. Trigonometric Functions

Function	Description	Example(s)

acos (double precision) → double precision	Inverse cosine, result in radians	`acos(1) → 0`
acosd (double precision) → double precision	Inverse cosine, result in degrees	`acosd(0.5) → 60`
asin (double precision) → double precision	Inverse sine, result in radians	`asin(1) → 1.5707963267948966`
asind (double precision) → double precision	Inverse sine, result in degrees	`asind(0.5) → 30`
atan (double precision) → double precision	Inverse tangent, result in radians	`atan(1) → 0.7853981633974483`

atand (double precision) → double precision	Inverse tangent, result in degrees	`atand(1) → 45`
atan2 (y double precision, x double precision) → double precision	Inverse tangent of y/x, result in radians	`atan2(1, 0) → 1.5707963267948966`
atan2d (y double precision, x double precision) → double precision	Inverse tangent of y/x, result in degrees	`atan2d(1, 0) → 90`
cos (double precision) → double precision	Cosine, argument in radians	`cos(0) → 1`
cosd (double precision) → double precision	Cosine, argument in degrees	`cosd(60) → 0.5`

cot (double precision) → double precision	Cotangent, argument in radians	`cot(0.5) → 1.830487721712452`
cotd (double precision) → double precision	Cotangent, argument in degrees	`cotd(45) → 1`
sin (double precision) → double precision	Sine, argument in radians	`sin(1) → 0.8414709848078965`
sind (double precision) → double precision	Sine, argument in degrees	`sind(30) → 0.5`
tan (double precision) → double precision	Tangent, argument in radians	`tan(1) → 1.5574077246549023`

Function	Description	Example(s)

sinh (double precision) → double precision	Hyperbolic sine	`sinh(1) → 1.1752011936438014`
cosh (double precision) → double precision	Hyperbolic cosine	`cosh(0) → 1`
tanh (double precision) → double precision	Hyperbolic tangent	`tanh(1) → 0.7615941559557649`
asinh (double precision) → double precision	Inverse hyperbolic sine	`asinh(1) → 0.881373587019543`
acosh (double precision) → double precision	Inverse hyperbolic cosine	`acosh(1) → 0`

atanh (double precision) → double precision	Inverse hyperbolic tangent	`atanh(0.5) → 0.5493061443340548`

Function/Operator	Description	Example(s)
text || text → text	Concatenates the two strings	`'Post' || 'greSQL' → PostgreSQL`
text || anynonarray → text anynonarray || text → text	Converts the non-string input to text, then concatenates the two strings. (The non-string input cannot be of an array type, because that would create ambiguity with the array || operators. If you want to concatenate an array's text equivalent, cast it to text explicitly.) (UNSUPPORTED)	`#'Value: ' || 42 → Value: 42`
text IS [NOT] [form] NORMALIZED → boolean	Checks whether the string is in the specified Unicode normalization form. The optional form key word specifies the form: NFC (the default), NFD, NFKC, or NFKD. This expression can only be used when the server encoding is UTF8. Note that checking for normalization using this expression is often faster than normalizing possibly already normalized strings.	`U&'\0061\0308bc' IS NFD NORMALIZED → true`
bit_length (text) → integer	Returns number of bits in the string (8 times the octet_length)	`bit_length('jose') → 32`

char_length (text) → integer character_length (text) → integer	Returns number of characters in the string.	`char_length('josé') → 4`
lower (text) → text	Converts the string to all lower case, according to the rules of the database's locale.	`lower('TOM') → tom`
normalize (text [, form]) → text	Converts the string to the specified Unicode normalization form. The optional form key word specifies the form: NFC (the default), NFD, NFKC, or NFKD. This function can only be used when the server encoding is UTF8.	`normalize(U&'\0061\0308bc', NFC) → äbc`
octet_length (text) → integer	Returns number of bytes in the string.	`octet_length('josé') → 5`
octet_length (character) → integer	Returns number of bytes in the string. Since this version of the function accepts type character directly, it will not strip trailing spaces.	`octet_length('abc '::character(4)) → 4`

overlay (string text PLACING newsubstring text FROM start integer [FOR count integer]) → text	Replaces the substring of string that starts at the start'th character and extends for count characters with newsubstring. If count is omitted, it defaults to the length of newsubstring.	`overlay('Txxxxas' placing 'hom' from 2 for 4) → Thomas`
position (substring text IN string text) → integer	Returns first starting index of the specified substring within string, or zero if it's not present.	`position('om' in 'Thomas') → 3`

substring (string text [FROM start integer] [FOR count integer]) → text	Extracts the substring of string starting at the start'th character if that is specified, and stopping after count characters if that is specified. Provide at least one of start and count.	`substring('Thomas' from 2 for 3) → hom`
		`substring('Thomas' from 3) → omas`
		`substring('Thomas' for 2) → Th`

substring (string text FROM pattern text) → text	Extracts the first substring matching POSIX regular expression; see Section 9.7.3.	`substring('Thomas' from '...$') → mas`

substring (string text SIMILAR pattern text ESCAPE escape text) → text substring (string text FROM pattern text FOR escape text) → text	Extracts the first substring matching SQL regular expression; see Section 9.7.2. The first form has been specified since SQL:2003; the second form was only in SQL:1999 and should be considered obsolete.	`substring('Thomas' similar '%#"o_a#"_' escape '#') → oma`
trim ([LEADING | TRAILING | BOTH] [characters text] FROM	Removes the longest string containing only characters in characters (a space by default) from the start, end, or both ends (BOTH is the default) of string.	

y to implement the SQL-

Function	Description	Example(s)
ascii (text) → integer	Returns the numeric code of the first character of the argument. In UTF8 encoding, returns the Unicode code point of the character. In other multibyte encodings, the argument must be an ASCII character.	`ascii('x') → 120`
btrim (string text [, characters text]) → text	Removes the longest string containing only characters in characters (a space by default) from the start and end of string.	`btrim('xyxtrimyyx', 'xyz') → trim`
chr (integer) → text	Returns the character with the given code. In UTF8 encoding the argument is treated as a Unicode code point. In other multibyte encodings the argument must designate an ASCII character. chr(0) is disallowed because text data types cannot store that character.	`chr(65) → A`

concat (val1 "any" [, val2 "any" [, ...]]) → text	Concatenates the text representations of all the arguments. NULL arguments are ignored. (NOT SUPPORTED)	`concat('abcde', 2, NULL, 22) → abcde222`
concat_ws (sep text, val1 "any" [, val2 "any" [, ...]]) → text	Concatenates all but the first argument, with separators. The first argument is used as the separator string, and should not be NULL. Other NULL arguments are ignored. (NOT SUPPORTED)	`concat_ws(',', 'abcde', 2, NULL, 22) → abcde,2,22`
format (formatstr text [, formatarg "any" [, ...]]) → text	Formats arguments according to a format string; see Section 9.4.1. This function is similar to the C function sprintf. (NOT SUPPORTED)	`format('Hello %s, %1$s', 'World') → Hello World, World`
initcap (text) → text	Converts the first letter of each word to upper case and the rest to lower case. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.	`initcap('hi THOMAS') → Hi Thomas`
left (string text, n integer) → text	Returns first n characters in the string, or when n is negative, returns all but last |n| characters.	`left('abcde', 2) → ab`

length (text) → integer	Returns the number of characters in the string.	`length('jose') → 4`
lpad (string text, length integer [, fill text]) → text	Extends the string to length length by prepending the characters fill (a space by default). If the string is already longer than length then it is truncated (on the right).	`lpad('hi', 5, 'xy') → xyxhi`
ltrim (string text [, characters text]) → text	Removes the longest string containing only characters in characters (a space by default) from the start of string.	`ltrim('zzzytest', 'xyz') → test`
md5 (text) → text	Computes the MD5 hash of the argument, with the result written in hexadecimal.	`md5('abc') → 900150983cd24fb0d6963f7d28e17f72`
parse_ident (qualified_identifier text [, strict_mode boolean DEFAULT true]) → text[]	Splits qualified_identifier into an array of identifiers, removing any quoting of individual identifiers. By default, extra characters after the last identifier are considered an error; but if the second parameter is false, then such extra characters are ignored. (This behavior is useful for parsing names for objects like functions.) Note that this function does not truncate over-length identifiers. If you want truncation you can cast the result to name[].	`parse_ident('"SomeSchema".someTable') → {SomeSchema,someTable}`

pg_client_encoding ( ) → name	Returns current client encoding name.	`pg_client_encoding() → UTF8`
quote_ident (text) → text	Returns the given string suitably quoted to be used as an identifier in an SQL statement string. Quotes are added only if necessary (i.e., if the string contains non-identifier characters or would be case-folded). Embedded quotes are properly doubled. See also Example 43.1.	`quote_ident('Foo bar') → "Foo bar"`
quote_literal (text) → text	Returns the given string suitably quoted to be used as a string literal in an SQL statement string. Embedded single-quotes and backslashes are properly doubled. Note that quote_literal returns null on null input; if the argument might be null, quote_nullable is often more suitable. See also Example 43.1.	`quote_literal(E'O\'Reilly') → ''O''Reilly''`
quote_literal (anyelement) → text	Converts the given value to text and then quotes it as a literal. Embedded single-quotes and backslashes are properly doubled. (NOT SUPPORTED)	`#quote_literal(42.5) → '42.5'`
quote_nullable (text) → text	Returns the given string suitably quoted to be used as a string literal in an SQL statement string; or, if the argument is null, returns NULL. Embedded single-quotes and backslashes are properly doubled. See also Example 43.1.	`quote_nullable(NULL) → NULL`

quote_nullable (anyelement) → text	Converts the given value to text and then quotes it as a literal; or, if the argument is null, returns NULL. Embedded single-quotes and backslashes are properly doubled. (NOT SUPPORTED)	`#quote_nullable(42.5) → '42.5'`
regexp_match (string text, pattern text [, flags text]) → text[]	Returns captured substrings resulting from the first match of a POSIX regular expression to the string; see Section 9.7.3.	`regexp_match('foobarbequebaz', '(bar)(beque)') → {bar,beque}`
regexp_matches (string text, pattern text [, flags text]) → setof text[]	Returns captured substrings resulting from the first match of a POSIX regular expression to the string, or multiple matches if the g flag is used; see Section 9.7.3. (NOT SUPPORTED)	`#regexp_matches('foobarbequebaz', 'ba.', 'g') → {bar},{baz}`
regexp_replace (string text, pattern text, replacement text [, flags text]) → text	Replaces substrings resulting from the first match of a POSIX regular expression, or multiple substring matches if the g flag is used; see Section 9.7.3.	`regexp_replace('Thomas', '.[mN]a.', 'M') → ThM`
regexp_split_to_array (string text, pattern text [, flags text]) → text[]	Splits string using a POSIX regular expression as the delimiter, producing an array of results; see Section 9.7.3.	`regexp_split_to_array('hello world', '\s+') → {hello,world}`

regexp_split_to_table (string text, pattern text [, flags text]) → setof text	Splits string using a POSIX regular expression as the delimiter, producing a set of results; see Section 9.7.3. (NOT SUPPORTED)	`#regexp_split_to_table('hello world', '\s+') → hello,world`
repeat (string text, number integer) → text	Repeats string the specified number of times.	`repeat('Pg', 4) → PgPgPgPg`
replace (string text, from text, to text) → text	Replaces all occurrences in string of substring from with substring to.	`replace('abcdefabcdef', 'cd', 'XX') → abXXefabXXef`
reverse (text) → text	Reverses the order of the characters in the string.	`reverse('abcde') → edcba`
right (string text, n integer) → text	Returns last n characters in the string, or when n is negative, returns all but first |n| characters.	`right('abcde', 2) → de`

rpad (string text, length integer [, fill text]) → text	Extends the string to length length by appending the characters fill (a space by default). If the string is already longer than length then it is truncated.	`rpad('hi', 5, 'xy') → hixyx`
rtrim (string text [, characters text]) → text	Removes the longest string containing only characters in characters (a space by default) from the end of string.	`rtrim('testxxzx', 'xyz') → test`
split_part (string text, delimiter text, n integer) → text	Splits string at occurrences of delimiter and returns the n'th field (counting from one), or when n is negative, returns the |n|'th-from-last field.	`split_part('abc~@~def~@~ghi', '~@~', 2) → def` `split_part('abc,def,ghi,jkl', ',', -2) → ghi`
strpos (string text, substring text) → integer	Returns first starting index of the specified substring within string, or zero if it's not present. (Same as position(substring in string), but note the reversed argument order.)	`strpos('high', 'ig') → 2`
substr (string text, start integer [, count integer]) → text	Extracts the substring of string starting at the start'th character, and extending for count characters if that is specified. (Same as substring(string from start for count).)	`substr('alphabet', 3) → phabet` `substr('alphabet', 3, 2) → ph`

starts_with (string text, prefix text) → boolean	Returns true if string starts with prefix.	`starts_with('alphabet', 'alph') → true`

ues of type bytea. Many of these

ails are in Table 9.11. PostgreSQL
.

Function/Operator	Description	Example(s)
bytea || bytea → bytea	Concatenates the two binary strings.	`'\x123456'::bytea || '\x789a00bcde'::bytea → \x123456789a00bcde`
bit_length (bytea) → integer	Returns number of bits in the binary string (8 times the octet_length).	`bit_length('\x123456'::bytea) → 24`
octet_length (bytea) → integer	Returns number of bytes in the binary string.	`octet_length('\x123456'::bytea) → 3`
overlay (bytes bytea PLACING newsubstring bytea FROM start integer [FOR count integer]) → bytea	Replaces the substring of bytes that starts at the start'th byte and extends for count bytes with newsubstring. If count is omitted, it defaults to the length of newsubstring.	`overlay('\x1234567890'::bytea placing '\002\003'::bytea from 2 for 3) → \x12020390`

position (substring bytea IN bytes bytea) → integer	Returns first starting index of the specified substring within bytes, or zero if it's not present.	`position('\x5678'::bytea in '\x1234567890'::bytea) → 3`
substring (bytes bytea [FROM start integer] [FOR count integer]) → bytea	Extracts the substring of bytes starting at the start'th byte if that is specified, and stopping after count bytes if that is specified. Provide at least one of start and count.	`substring('\x1234567890'::bytea from 3 for 2) → \x5678`

) → a2x5

ıta
data

re used internally to implement the

Function	Description	Example(s)
bit_count (bytes bytea) → bigint	Returns the number of bits set in the binary string (also known as "popcount").	`bit_count('\x1234567899'::bytea) → 15`
btrim (bytes bytea, bytesremoved bytea) → bytea	Removes the longest string containing only bytes appearing in bytesremoved from the start and end of bytes.	`btrim('\x1234567890'::bytea, '\x9012'::bytea) → \x345678`
get_bit (bytes bytea, n bigint) → integer	Extracts n'th bit from binary string.	`get_bit('\x1234567890'::bytea, 30) → 1`

get_byte (bytes bytea, n integer) → integer	Extracts n'th byte from binary string.	`get_byte('\x1234567890'::bytea, 4) → 144`
length (bytea) → integer	Returns the number of bytes in the binary string.	`length('\x1234567890'::bytea) → 5`
length (bytes bytea, encoding name) → integer	Returns the number of characters in the binary string, assuming that it is text in the given encoding.	`length('jose'::bytea, 'UTF8') → 4`
ltrim (bytes bytea, bytesremoved bytea) → bytea	Removes the longest string containing only bytes appearing in bytesremoved from the start of bytes.	`ltrim('\x1234567890'::bytea, '\x9012'::bytea) → \x34567890`
md5 (bytea) → text	Computes the MD5 hash of the binary string, with the result written in hexadecimal.	`md5('Th\000omas'::bytea) → 8ab2d3c9689aaf18b4958c334c82d8b1`

Functions get_byte and set_byte number the first byte of a binary string as byte 0. Functions get_bit and set_bit number bits from the right within each byte; for example bit 0 is the least significant bit of the first byte, and bit 15 is the most significant bit of the second byte.

For historical reasons, the function md5 returns a hex-encoded value of type text whereas the SHA-2 functions return type bytea. Use the functions encode and decode to convert between the two. For example write encode(sha256('abc'), 'hex') to get a hex-encoded text representation, or decode(md5('abc'), 'hex') to get a bytea value.

Functions for converting strings between different character sets (encodings), and for representing arbitrary binary data in textual form, are shown in Table 9.13. For these functions, an argument or result of type text is expressed in the database's default encoding, while arguments or results of type bytea are in an encoding named by another argument.

Table 9.13. Text/Binary String Conversion Functions

Function	Description	Example(s)
convert (bytes bytea, src_encoding name, dest_encoding name) → bytea	Converts a binary string representing text in encoding src_encoding to a binary string in encoding dest_encoding (see Section 24.3.4 for available conversions). (NOT SUPPORTED)	`#convert('text_in_utf8', 'UTF8', 'LATIN1') → \x746578745f696e5f75746638`

convert_from (bytes bytea, src_encoding name) → text	Converts a binary string representing text in encoding src_encoding to text in the database encoding (see Section 24.3.4 for available conversions).	`convert_from('text_in_utf8', 'UTF8') → text_in_utf8`
convert_to (string text, dest_encoding name) → bytea	Converts a text string (in the database encoding) to a binary string encoded in encoding dest_encoding (see Section 24.3.4 for available conversions).	`convert_to('some_text', 'UTF8') → \x736f6d655f74657874`
encode (bytes bytea, format text) → text	Encodes binary data into a textual representation; supported format values are: base64, escape, hex.	`encode('123\000\001', 'base64') → MTIzAAE=`
decode (string text, format text) → bytea	Decodes binary data from a textual representation; supported format values are the same as for encode.	`decode('MTIzAAE=', 'base64') → \x3132330001`

s of the types bit and bit varying.
y.) Bit strings support the usual comparison operators shown in Table 9.1, as well as the operators shown in Table 9.14.

Table 9.14. Bit String Operators

Operator	Description	Example(s)
Concatenation		
Bitwise AND (inputs must be of equal length)		
Bitwise OR (inputs must be of equal length)		
Bitwise exclusive OR (inputs must be of equal length)		
Bitwise NOT		
Bitwise shift left (string length is preserved)		

Bitwise shift right (string length is preserved)		

wn in Table 9.15.

sha512 (bytea) →	Computes the SHA-512 hash	

Function	Description	Example(s)
bit_count (bit) → bigint	Returns the number of bits set in the bit string (also known as “popcount”).	`bit_count(B'10111') → 4`
bit_length (bit) → integer	Returns number of bits in the bit string.	`bit_length(B'10111') → 5`
length (bit) → integer	Returns number of bits in the bit string.	`length(B'10111') → 5`
octet_length (bit) → integer	Returns number of bytes in the bit string.	`octet_length(B'1011111011') → 2`

overlay (bits bit PLACING newsubstring bit FROM start integer [FOR count integer]) → bit	Replaces the substring of bits that starts at the start'th bit and extends for count bits with newsubstring. If count is omitted, it defaults to the length of newsubstring.	`overlay(B'010101010101010' placing B'11111' from 2 for 3) → 0111110101010101010`
position (substring bit IN bits bit) → integer	Returns first starting index of the specified substring within bits, or zero if it's not present.	`position(B'010' in B'000001101011') → 8`
substring (bits bit [FROM start integer] [FOR count integer]) → bit	Extracts the substring of bits starting at the start'th bit if that is specified, and stopping after count bits if that is specified. Provide at least one of start and count.	`substring(B'110010111111' from 3 for 2) → 00`
get_bit (bits bit, n integer) → integer	Extracts n'th bit from bit string; the first (leftmost) bit is bit 0.	`get_bit(B'101010101010101010', 6) → 1`
set_bit (bits bit, n integer, newvalue integer) → bit	Sets n'th bit in bit string to newvalue; the first (leftmost) bit is bit 0.	`set_bit(B'101010101010101010', 6, 0) → 101010001010101010`

opies the rightmost n bits. Casting an
les:

bit of the integer.

OT LIKE expression returns false if LIKE

ring itself; in that case LIKE acts like the
nt sign (%) matches any sequence of zero

Some examples:

```
'abc' LIKE 'abc'    → true
'abc' LIKE 'a%'     → true
'abc' LIKE '_b_'    → true
'abc' LIKE 'c'      → false
```

LIKE pattern matching always covers the entire string. Therefore, if it's desired to match a sequence anywhere within a string, the pattern must start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, the respective character in pattern must be preceded by the escape character. The default escape character is the backslash but a different one can be selected by using the ESCAPE clause. To match the escape character itself, write two escape characters.

Note
If you have standard_conforming_strings turned off, any backslashes you write in literal string constants will need to be doubled. See Section 4.1.2.1 for more information.

It's also possible to select no escape character by writing ESCAPE ''. This effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

According to the SQL standard, omitting ESCAPE means there is no escape character (rather than defaulting to a backslash), and a zero-length ESCAPE value is disallowed. PostgreSQL's behavior in this regard is therefore slightly nonstandard.

The key word ILIKE can be used instead of LIKE to make the match case-insensitive according to the active locale. This is not in the SQL standard but is a PostgreSQL extension.

The operator `~~` is equivalent to LIKE, and `~~*` corresponds to ILIKE. There are also `!~~` and `!~~*` operators that represent NOT LIKE and NOT ILIKE, respectively. All of these operators are PostgreSQL-specific. You may see these operator names in EXPLAIN output and similar places, since the parser actually translates LIKE et al. to these operators.

The phrases LIKE, ILIKE, NOT LIKE, and NOT ILIKE are generally treated as operators in PostgreSQL syntax; for example they can be used in expression operator ANY (subquery) constructs, although an ESCAPE clause cannot be included there. In some obscure cases it may be necessary to use the underlying operator names instead.

Also see the prefix operator ^@ and corresponding starts_with function, which are useful in cases where simply matching the beginning of a string is needed.

9.7.2. SIMILAR TO Regular Expressions

```
string SIMILAR TO pattern [ESCAPE escape-character] (NOT SUPPORTED)
string NOT SIMILAR TO pattern [ESCAPE escape-character] (NOT SUPPORTED)
```

The SIMILAR TO operator returns true or false depending on whether its pattern matches the given string. It is similar to LIKE, except that it interprets the pattern using the SQL standard's definition of a regular expression. SQL regular expressions are a curious cross between LIKE notation and common (POSIX) regular expression notation.

Like LIKE, the SIMILAR TO operator succeeds only if its pattern matches the entire string; this is unlike common regular expression behavior where the pattern can match any part of the string. Also like LIKE, SIMILAR TO uses _ and % as wildcard characters denoting any single character and any string, respectively (these are comparable to . and .* in POSIX regular expressions).

In addition to these facilities borrowed from LIKE, SIMILAR TO supports these pattern-matching metacharacters borrowed from POSIX regular expressions:

| denotes alternation (either of two alternatives).

- * denotes repetition of the previous item zero or more times.
- + denotes repetition of the previous item one or more times.

? denotes repetition of the previous item zero or one time.

{m} denotes repetition of the previous item exactly m times.

{m,} denotes repetition of the previous item m or more times.

{m,n} denotes repetition of the previous item at least m and not more than n times.

Parentheses () can be used to group items into a single logical item.

A bracket expression [...] specifies a character class, just as in POSIX regular expressions.

Notice that the period (.) is not a metacharacter for SIMILAR TO.

As with LIKE, a backslash disables the special meaning of any of these metacharacters. A different escape character can be specified with ESCAPE, or the escape capability can be disabled by writing ESCAPE ''.

According to the SQL standard, omitting ESCAPE means there is no escape character (rather than defaulting to a backslash), and a zero-length ESCAPE value is disallowed. PostgreSQL's behavior in this regard is therefore slightly nonstandard.

Another nonstandard extension is that following the escape character with a letter or digit provides access to the escape sequences defined for POSIX regular expressions; see Table 9.20, Table 9.21, and Table 9.22 below.

Some examples:

```
#'abc' SIMILAR TO 'abc'           → true
#'abc' SIMILAR TO 'a'             → false
#'abc' SIMILAR TO '%(b|d)%'       → true
#'abc' SIMILAR TO '(b|c)%'        → false
#'-abc-' SIMILAR TO '%\mabc\M%'   → true
#'xabcy' SIMILAR TO '%\mabc\M%'   → false
```

The substring function with three parameters provides extraction of a substring that matches an SQL regular expression pattern. The function can be written according to standard SQL syntax:

```
substring(string similar pattern escape escape-character)
```

or using the now obsolete SQL:1999 syntax:

```
substring(string from pattern for escape-character)
```

or as a plain three-argument function:

```
substring(string, pattern, escape-character)
```

As with SIMILAR TO, the specified pattern must match the entire data string, or else the function fails and returns null. To indicate the part of the pattern for which the matching data sub-string is of interest, the pattern should contain two occurrences of the escape character followed by a double quote ("). The text matching the portion of the pattern between these separators is returned when the match is successful.

The escape-double-quote separators actually divide substring's pattern into three independent regular expressions; for example, a vertical bar (|) in any of the three sections affects only that section. Also, the first and third of these regular expressions are defined to match the smallest possible amount of text, not the largest, when there is any ambiguity about how much of the data string matches which pattern. (In POSIX parlance, the first and third regular expressions are forced to be non-greedy.)

As an extension to the SQL standard, PostgreSQL allows there to be just one escape-double-quote separator, in which case the third regular expression is taken as empty; or no separators, in which case the first and third regular expressions are taken as empty.

Some examples, with #" delimiting the return string:

```
substring('foobar' similar '%#"o_b#"%' escape '#')    → oob
substring('foobar' similar '#"o_b#"%' escape '#')     → NULL
```

9.7.3. POSIX Regular Expressions
Table 9.16 lists the available operators for pattern matching using POSIX regular expressions.

Table 9.16. Regular Expression Match Operators

Operator	Description	Example(s)
ext ~ text → boolean	String matches regular expression, case sensitively	`'thomas' ~ 't.*ma' → true`
text ~* text → boolean	String matches regular expression, case insensitively	`'thomas' ~* 'T.*ma' → true`
text !~ text → boolean	String does not match regular expression, case sensitively	`'thomas' !~ 't.*max' → true`

text !~* text → boolean	String does not match regular expression, case insensitively	`'thomas' !~* 'T.*ma' → false`

characters exactly unless they are special characters in the regular expression language — but regular expressions use different special characters than LIKE does. Unlike LIKE patterns, a regular expression is allowed to match anywhere within a string, unless the regular expression is explicitly anchored to the beginning or end of the string.

Some examples:

```
'abcd' ~ 'bc'     → true
'abcd' ~ 'a.c'    → true /* dot matches any character */
'abcd' ~ 'a.*d'   → true /* * repeats the preceding pattern item */
'abcd' ~ '(b|x)'  → true /* | means OR, parentheses group */
'abcd' ~ '^a'     → true /* ^ anchors to start of string */
'abcd' ~ '^(b|c)' → false /* would match except for anchoring */
```

The POSIX pattern language is described in much greater detail below.

The substring function with two parameters, substring(string from pattern), provides extraction of a substring that matches a POSIX regular expression pattern. It returns null if there is no match, otherwise the first portion of the text that matched the pattern. But if the pattern contains any parentheses, the portion of the text that matched the first parenthesized subexpression (the one whose left parenthesis comes first) is returned. You can put parentheses around the whole expression if you want to use parentheses within it without triggering this exception. If you need parentheses in the pattern before the subexpression you want to extract, see the non-capturing parentheses described below.

Some examples:

```
substring('foobar' from 'o.b')      → oob
substring('foobar' from 'o(.)b')    → o
```

The regexp_replace function provides substitution of new text for substrings that match POSIX regular expression patterns. It has the syntax regexp_replace(source, pattern, replacement [, flags]). The source string is returned unchanged if there is no match to the pattern. If there is a match, the source string is returned with the replacement string substituted for the matching substring. The replacement string can contain \n, where n is 1 through 9, to indicate that the source substring matching the n'th parenthesized subexpression of the pattern should be inserted, and it can contain & to indicate that the substring matching the entire pattern should be inserted. Write \ if you need to put a literal backslash in the replacement text. The flags parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Flag i specifies case-insensitive matching, while flag g specifies replacement of each matching substring rather than only the first one. Supported flags (though not g) are described in Table 9.24.

Some examples:

```
regexp_replace('foobarbaz', 'b..', 'X') → fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g') → fooXX
regexp_replace('foobarbaz', 'b(..)', 'X\1Y', 'g') → fooXarYXazY
```

The regexp_match function returns a text array of captured substring(s) resulting from the first match of a POSIX regular expression pattern to a string. It has the syntax regexp_match(string, pattern [, flags]). If there is no match, the result is NULL. If a match is found, and the pattern contains no parenthesized subexpressions, then the result is a single-element text array containing the substring matching the whole pattern. If a match is found, and the pattern contains parenthesized subexpressions, then the result is a text array whose n'th element is the substring matching the n'th parenthesized subexpression of the pattern (not counting “non-capturing” parentheses; see below for details). The flags parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Supported flags are described in Table 9.24.

Some examples:

```
regexp_match('foobarbequebaz', 'bar.*que') → {barbeque}
regexp_match('foobarbequebaz', '(bar)(beque)') → {bar,beque}
```

In the common case where you just want the whole matching substring or NULL for no match, write something like

```
#(regexp_match('foobarbequebaz', 'bar.*que'))[1] → barbeque
```

The regexp_matches function returns a set of text arrays of captured substring(s) resulting from matching a POSIX regular expression pattern to a string. It has the same syntax as regexp_match. This function returns no rows if there is no match, one row if there is a match and the g flag is not given, or N rows if there are N matches and the g flag is given. Each returned row is a text array containing the whole matched substring or the substrings matching parenthesized subexpressions of the pattern, just as described above for regexp_match. regexp_matches accepts all the flags shown in Table 9.24, plus the g flag which commands it to return all matches, not just the first one.

Some examples:

```
SELECT * FROM regexp_matches('foo', 'not there') a → [
]
```

```
SELECT * FROM regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g') a → [
{bar,beque}
{bazil,barf}
]
```

Tip
In most cases regexp_matches() should be used with the g flag, since if you only want the first match, it's easier and more efficient to use regexp_match(). However, regexp_match() only exists in PostgreSQL version 10 and up. When working in older versions, a common trick is to place a regexp_matches() call in a sub-select, for example:

```
SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)')) FROM tab;
```

This produces a text array if there's a match, or NULL if not, the same as regexp_match() would do. Without the sub-select, this query would produce no output at all for table rows without a match, which is typically not the desired behavior.

The regexp_split_to_table function splits a string using a POSIX regular expression pattern as a delimiter. It has the syntax regexp_split_to_table(string, pattern [, flags]). If there is no match to the pattern, the function returns the string. If there is at least one match, for each match it returns the text from the end of the last match (or the beginning of the string) to the beginning of the match. When there are no more matches, it returns the text from the end of the last match to the end of the string. The flags parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. regexp_split_to_table supports the flags described in Table 9.24.

The regexp_split_to_array function behaves the same as regexp_split_to_table, except that regexp_split_to_array returns its result as an array of text. It has the syntax regexp_split_to_array(string, pattern [, flags]). The parameters are the same as for regexp_split_to_table.

Some examples:

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumps over the lazy dog', '\s+') AS foo → [
the
quick
brown
fox
jumps
over
the
lazy
dog
]
```

```
SELECT regexp_split_to_array('the quick brown fox jumps over the lazy dog', '\s+') →
{the,quick,brown,fox,jumps,over,the,lazy,dog}
```

```
SELECT foo FROM regexp_split_to_table('the quick brown fox', '\s*') AS foo → [
t
h
e
q
u
i
c
k
b
r
o
w
n
f
o
x
]
```

As the last example demonstrates, the regexp split functions ignore zero-length matches that occur at the start or end of the string or immediately after a previous match. This is contrary to the strict definition of regexp matching that is implemented by regexp_match and regexp_matches, but is usually the most convenient behavior in practice. Other software systems such as Perl use similar definitions.

9.7.3.5. Regular Expression Matching Rules
In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, either the longest possible match or the shortest possible match will be taken, depending on whether the RE is greedy or non-greedy.

Whether an RE is greedy or not is determined by the following rules:

Most atoms, and all constraints, have no greediness attribute (because they cannot match variable amounts of text anyway).

Adding parentheses around an RE does not change its greediness.

A quantified atom with a fixed-repetition quantifier ({m} or {m}?) has the same greediness (possibly none) as the atom itself.

A quantified atom with other normal quantifiers (including {m,n} with m equal to n) is greedy (prefers longest match).

A quantified atom with a non-greedy quantifier (including {m,n}? with m equal to n) is non-greedy (prefers shortest match).

A branch — that is, an RE that has no top-level | operator — has the same greediness as the first quantified atom in it that has a greediness attribute.

An RE consisting of two or more branches connected by the | operator is always greedy.

The above rules associate greediness attributes not only with individual quantified atoms, but with branches and entire REs that contain quantified atoms. What that means is that the matching is done in such a way that the branch, or whole RE, matches the longest or shortest possible substring as a whole. Once the length of the entire match is determined, the part of it that matches any particular subexpression is determined on the basis of the greediness attribute of that subexpression, with subexpressions starting earlier in the RE taking priority over ones starting later.

An example of what this means:

```
SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})')  → 123
SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})') → 1
```

In the first case, the RE as a whole is greedy because `Y*` is greedy. It can match beginning at the Y, and it matches the longest possible string starting there, i.e., Y123. The output is the parenthesized part of that, or 123. In the second case, the RE as a whole is non-greedy because `Y*?` is non-greedy. It can match beginning at the Y, and it matches the shortest possible string starting there, i.e., Y1. The subexpression [0-9]{1,3} is greedy but it cannot change the decision as to the overall match length; so it is forced to match just 1.

In short, when an RE contains both greedy and non-greedy subexpressions, the total match length is either as long as possible or as short as possible, according to the attribute assigned to the whole RE. The attributes assigned to the subexpressions only affect how much of that match they are allowed to “eat” relative to each other.

The quantifiers {1,1} and {1,1}? can be used to force greediness or non-greediness, respectively, on a subexpression or a whole RE. This is useful when you need the whole RE to have a greediness attribute different from what's deduced from its elements. As an example, suppose that we are trying to separate a string containing some digits into the digits and the parts before and after them. We might try to do that like this:

```
regexp_match('abc01234xyz', '(.*)(\d+)(.*)') → {abc0123,4,xyz}
```

That didn't work: the first .* is greedy so it “eats” as much as it can, leaving the \d+ to match at the last possible place, the last digit. We might try to fix that by making it non-greedy:

```
regexp_match('abc01234xyz', '(.*?)(\d+)(.*)') → {abc,0,""}
```

That didn't work either, because now the RE as a whole is non-greedy and so it ends the overall match as soon as possible. We can get what we want by forcing the RE as a whole to be greedy:

```
regexp_match('abc01234xyz', '(?:(.*?)(\d+)(.*)){1,1}') → {abc,01234,xyz}
```

Controlling the RE's overall greediness separately from its components' greediness allows great flexibility in handling variable-length patterns.

When deciding what is a longer or shorter match, match lengths are measured in characters, not collating elements. An empty string is considered longer than no match at all. For example: `bb*` matches the three middle characters of abbbc; (week|wee)(night|knights) matches all ten characters of weeknights; when `(.*).*` is matched against abc the parenthesized subexpression matches all three characters; and when (a*)* is matched against bc both the whole RE and the parenthesized subexpression match an empty string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, e.g., x becomes [xX]. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, e.g., [x] becomes [xX] and [^x] becomes [^xX].

If newline-sensitive matching is specified, . and bracket expressions using ^ will never match the newline character (so that matches will not cross lines unless the RE explicitly includes a newline) and ^ and $ will match the empty string after and before a newline respectively, in addition to matching at beginning and end of string respectively. But the ARE escapes \A and \Z continue to match beginning or end of string only. Also, the character class shorthands \D and \W will match a newline regardless of this mode. (Before PostgreSQL 14, they did not match newlines when in newline-sensitive mode. Write [^[:digit:]] or [^[:word:]] to get the old behavior.)

If partial newline-sensitive matching is specified, this affects . and bracket expressions as with newline-sensitive matching, but not ^ and $.

If inverse partial newline-sensitive matching is specified, this affects ^ and $ as with newline-sensitive matching, but not . and bracket expressions. This isn't very useful but is provided for symmetry

9.8. Data Type Formatting Functions

The PostgreSQL formatting functions provide a powerful set of tools for converting various data types (date/time, integer, floating point, numeric) to formatted strings and for converting from formatted strings to specific data types. Table 9.25 lists them. These functions all follow a common calling convention: the first argument is the value to be formatted and the second argument is a template that defines the output or input format.

Table 9.25. Formatting Functions

Function	Description	Example(s)
to_char (timestamp with time zone, text) → text	Converts time stamp to string according to the given format.	`to_char(timestamp '2002-04-20 17:31:12.66', 'HH12:MI:SS') → 05:31:12`
to_char (interval, text) → text	Converts interval to string according to the given format.	`to_char(interval '15h 2m 12s', 'HH24:MI:SS') → 15:02:12`

to_char (numeric_type, text) → text	Converts number to string according to the given format; available for integer, bigint, numeric, real, double precision.	`to_char(125, '999') → ' 125'`
		`to_char(125.8::real, '999D9') → ' 125.8'`
		`to_char(-125.8, '999D99S') → '125.80-'`

to_date (text, text) → date	Converts string to date according to the given format.	`to_date('05 Dec 2000', 'DD Mon YYYY') → '2000-12-05'`
to_number (text, text) → numeric	Converts string to numeric according to the given format.	`to_number('12,454.8-', '99G999D9S') → '-12454.8'`
to_timestamp (text, text) → timestamp with time zone	Converts string to time stamp according to the given format. (See also to_timestamp(double precision) in Table 9.32.)	`cast(to_timestamp('05 Dec 2000', 'DD Mon YYYY') as timestamp) → '2000-12-05 00:00:00'`

```
ay  , 06  05:39:18'
uesday, 6  05:39:18'
```

```
to_char(-485, '9SG99') → '4-85'
to_char(-485, '999PR') → '<485>'
to_char(485, 'L999') → '  485'
to_char(485, 'RN') → '        CDLXXXV'
to_char(485, 'FMRN') → 'CDLXXXV'
to_char(5.2, 'FMRN') → 'V'
to_char(482, '999th') → ' 482nd'
to_char(485, '"Good number:"999') → 'Good number: 485'
to_char(485.8, '"Pre:"999" Post:" .999') → 'Pre: 485 Post: .800'
to_char(12, '99V999') → ' 12000'
to_char(12.4, '99V999') → ' 12400'
to_char(12.45, '99V9') → ' 125'
to_char(0.0004859, '9.99EEEE') → ' 4.86e-04'
```

9.9. Date/Time Functions and Operators

Table 9.32 shows the available functions for date/time value processing, with details appearing in the following subsections. Table 9.31 illustrates the behaviors of the basic arithmetic operators (+, *, etc.). For formatting functions, refer to Section 9.8. You should be familiar with the background information on date/time data types from Section 8.5.

In addition, the usual comparison operators shown in Table 9.1 are available for the date/time types. Dates and timestamps (with or without time zone) are all comparable, while times (with or without time zone) and intervals can only be compared to other values of the same data type. When comparing a timestamp without time zone to a timestamp with time zone, the former value is assumed to be given in the time zone specified by the TimeZone configuration parameter, and is rotated to UTC for comparison to the latter value (which is already in UTC internally). Similarly, a date value is assumed to represent midnight in the TimeZone zone when comparing it to a timestamp.

All the functions and operators described below that take time or timestamp inputs actually come in two variants: one that takes time with time zone or timestamp with time zone, and one that takes time without time zone or timestamp without time zone. For brevity, these variants are not shown separately. Also, the + and * operators come in commutative pairs (for example both date + integer and integer + date); we show only one of each such pair.

Table 9.31. Date/Time Operators

Operator	Description	Example(s)

date + integer → date	Add a number of days to a date	`date '2001-09-28' + 7 → 2001-10-05`
date + interval → timestamp	Add an interval to a date	`date '2001-09-28' + interval '1 hour' → 2001-09-28 01:00:00`
date + time → timestamp	Add a time-of-day to a date	`date '2001-09-28' + time '03:00' → 2001-09-28 03:00:00`
interval + interval → interval	Add intervals	`interval '1 day' + interval '1 hour' → 1 day 01:00:00`
timestamp + interval → timestamp	Add an interval to a timestamp	`timestamp '2001-09-28 01:00' + interval '23 hours' → 2001-09-29 00:00:00`

time + interval → time	Add an interval to a time	`time '01:00' + interval '3 hours' → 04:00:00`
• interval → interval	Negate an interval	`- interval '23 hours' → -23:00:00`
date - date → integer	Subtract dates, producing the number of days elapsed	`date '2001-10-01' - date '2001-09-28' → 3`
date - integer → date	Subtract a number of days from a date	`date '2001-10-01' - 7 → 2001-09-24`
date - interval → timestamp	Subtract an interval from a date	`date '2001-09-28' - interval '1 hour' → 2001-09-27 23:00:00`

time - time → interval	Subtract times	time '05:00' - time '03:00' → 02:00:00

Function	Description	Example(s)
age (timestamp, timestamp) → interval	Subtract arguments, producing a "symbolic" result that uses years and months, rather than just days	age(timestamp '2001-04-10', timestamp '1957-06-13') → 43 years 9 mons 27 days
age (timestamp) → interval	Subtract argument from current_date (at midnight)	age(timestamp '1957-06-13') ⌐→ 62 years 6 mons 10 days
clock_timestamp ( ) → timestamp with time zone	Current date and time (changes during statement execution); see Section 9.9.5	clock_timestamp() ⌐→ 2019-12-23 14:39:53.662522-05
current_date → date	Current date; see Section 9.9.5	current_date ⌐→ 2019-12-23

- interval '2 hours' →

01-09-28 23:00' - interval
2001-09-28 00:00:00

ay' - interval '1 hour' →
00

		timestamp '2001-09-29 03:00' - timestamp '2001-07-27 12:00' → 63 days 15:00:00

current_time → time with time zone	Current time of day; see Section 9.9.5	`current_time -→ 14:39:53.662522-05`
current_time (integer) → time with time zone	Current time of day, with limited precision; see Section 9.9.5	`current_time(2) -→ 14:39:53.66-05`
current_timestamp → timestamp with time zone	Current date and time (start of current transaction); see Section 9.9.5	`current_timestamp -→ 2019-12-23 14:39:53.662522-05`
current_timestamp (integer) → timestamp with time zone	Current date and time (start of current transaction), with limited precision; see Section 9.9.5	`current_timestamp(0) -→ 2019-12-23 14:39:53-05`
date_bin (interval, timestamp, timestamp) → timestamp	Bin input into specified interval aligned with specified origin; see Section 9.9.3	`date_bin('15 minutes', timestamp '2001-02-16 20:38:40', timestamp '2001-02-16 20:05:00') → 2001-02-16 20:35:00`

```
econd' * 900 → 00:15:00
ay' * 21 → 21 days
our' * 3.5 → 03:30:00
```

```
our' / 1.5 → 00:40:00
```

date_part (text, timestamp) → double precision	Get timestamp subfield (equivalent to extract); see Section 9.9.1	`date_part('hour', timestamp '2001-02-16 20:38:40') → 20`
date_part (text, interval) → double precision	Get interval subfield (equivalent to extract); see Section 9.9.1	`date_part('month', interval '2 years 3 months') → 3`
date_trunc (text, timestamp) → timestamp	Truncate to specified precision; see Section 9.9.2	`date_trunc('hour', timestamp '2001-02-16 20:38:40') → 2001-02-16 20:00:00`
date_trunc (text, timestamp with time zone, text) → timestamp with time zone	Truncate to specified precision in the specified time zone; see Section 9.9.2 (NOT SUPPORTED)	`#date_trunc('day', timestamptz '2001-02-16 20:38:40+00', 'Australia/Sydney') → 2001-02-16 13:00:00+00`

date_trunc (text, interval) → interval	Truncate to specified precision; see Section 9.9.2	`date_trunc('hour', interval '2 days 3 hours 40 minutes') → 2 days 03:00:00`
extract (field from timestamp) → numeric	Get timestamp subfield; see Section 9.9.1	`extract(hour from timestamp '2001-02-16 20:38:40') → 20`
extract (field from interval) → numeric	Get interval subfield; see Section 9.9.1	`extract(month from interval '2 years 3 months') → 3`
isfinite (date) → boolean	Test for finite date (not +/-infinity)	`isfinite(date '2001-02-16') → true`
isfinite (timestamp) → boolean	Test for finite timestamp (not +/-infinity)	`isfinite(timestamp 'infinity') → false`

In addition to these functions, the SQL OVERLAPS operator is supported: (NOT SUPPORTED)

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

This expression yields true when two time periods (defined by their endpoints) overlap, false when they do not overlap. The endpoints can be specified as pairs of dates, times, or time stamps; or as a date, time, or time stamp followed by an interval. When a pair of values is provided, either the start or the end can be written first; OVERLAPS automatically takes the earlier value of the pair as the start. Each time period is considered to represent the half-open interval start <= time < end, unless start and end are equal in which case it represents that single time instant. This means for instance that two time periods with only an endpoint in common do not overlap.

```
(DATE '2001-02-16', DATE '2001-12-21') OVERLAPS (DATE '2001-10-30', DATE '2002-10-30') → true
(DATE '2001-02-16', INTERVAL '100 days') OVERLAPS (DATE '2001-10-30', DATE '2002-10-30') → false
(DATE '2001-10-29', DATE '2001-10-30') OVERLAPS (DATE '2001-10-30', DATE '2001-10-31') → false
(DATE '2001-10-30', DATE '2001-10-30') OVERLAPS (DATE '2001-10-30', DATE '2001-10-31') → true
```

When adding an interval value to (or subtracting an interval value from) a timestamp with time zone value, the days component advances or decrements the date of the timestamp with time zone by the indicated number of days, keeping the time of day the same. Across daylight saving time changes (when the session time zone is set to a time zone that recognizes DST), this means interval '1 day' does not necessarily equal interval '24 hours'. For example, with the session time zone set to America/Denver:

isfinite (interval) → boolean	Test for finite interval (currently always true)	`isfinite(interval '4 hours') → true`
justify_days (interval) → interval	Adjust interval so 30-day time periods are represented as months	`justify_days(interval '35 days') → 1 mon 5 days`
justify_hours (interval) → interval	Adjust interval so 24-hour time periods are represented as days	`justify_hours(interval '27 hours') → 1 day 03:00:00`
justify_interval (interval) → interval	Adjust interval using justify_days and justify_hours, with additional sign adjustments	`justify_interval(interval '1 mon -1 hour') → 29 days 23:00:00`
localtime → time	Current time of day; see Section 9.9.5 (NOT SUPPORTED)	`#localtime ~→ 14:39:53.662522`

```
'1 day' ~→ 2005-04-03 12:00:00-06
'24 hours' ~→ 2005-04-03 13:00:00-06
```

saving time at 2005-04-03 02:00:00 in time zone

different months have different numbers of days. PostgreSQL's ng partial months. For example, age('2004-06-01', '2004-04-30') because May has 31 days, while April has only 30.

ally simple way to perform subtraction is to convert each value to e results; this produces the number of seconds between the two hanges, and daylight saving time adjustments. Subtraction of date -hours) and hours/minutes/seconds between the values, making l hours/minutes/seconds, performing field-by-field subtraction te the differences in these approaches. The sample results were ange between the two dates used:

```
ACT(EPOCH FROM timestamptz '2013-03-01 12:00:00') ~→

RACT(EPOCH FROM timestamptz '2013-03-01 12:00:00')) /

12:00:00' ~→ 121 days 23:00:00
01 12:00:00') → 4 mons
```

me values. source must be a value expression of type timestamp, therefore be used as well.) field is an identifier or string that eturns values of type numeric. The following are valid field

```
EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40') → 21
```

The first century starts at 0001-01-01 00:00:00 AD, although they did not know it at the time. This definition applies to all Gregorian calendar countries. There is no century number 0, you go from -1 century to 1 century. If you disagree with this, please write your complaint to: Pope, Cathedral Saint-Peter of Roma, Vatican.

day
For timestamp values, the day (of the month) field (1–31) ; for interval values, the number of days

```
EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40') → 16
 EXTRACT(DAY FROM INTERVAL '40 days 1 minute') → 40
```

decade
The year field divided by 10

```
EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40') → 200
```

dow
The day of the week as Sunday (0) to Saturday (6)

```
EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40') → 5
```

Note that extract's day of the week numbering differs from that of the to_char(..., 'D') function.

doy
The day of the year (1–365/366)

```
EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40') → 47
```

epoch
For timestamp with time zone values, the number of seconds since 1970-01-01 00:00:00 UTC (negative for timestamps before that); for

localtime (integer) → time	Current time of day, with limited precision; see Section 9.9.5 (NOT SUPPORTED)	`#localtime(0) -→ 14:39:53`
localtimestamp → timestamp	Current date and time (start of current transaction); see Section 9.9.5 (NOT SUPPORTED)	`#localtimestamp -→ 2019-12-23 14:39:53.662522`
localtimestamp (integer) → timestamp	Current date and time (start of current transaction), with limited precision; see Section 9.9.5 (NOT SUPPORTED)	`#localtimestamp(2) -→ 2019-12-23 14:39:53.66`
make_date (year int, month int, day int) → date	Create date from year, month and day fields (negative years signify BC)	`make_date(2013, 7, 15) → 2013-07-15`
make_interval ([years int [, months int [, weeks int [, days int [, hours int [, mins int [, secs double precision]]]]]]]) → interval	Create interval from years, months, weeks, days, hours, minutes and seconds fields, each of which can default to zero (NOT SUPPORTED)	`#make_interval(days => 10) → 10 days`

1-01 00:00:00, without regard to timezone or daylight-savings

```
8:40.12-08') → 982384720.120000
2355920.120000
00
```

to_timestamp:

timestamp value could produce a misleading result: the result will
might not be the case.

of the week numbering.

o intervals)

k containing the 4th of January, so in early January or late
e week field for more information.

julian
The Julian Date corresponding to the date or timestamp (not applicable to intervals). Timestamps that are not local midnight result in a fractional value. See Section B.7 for more information.

```
EXTRACT(JULIAN FROM DATE '2006-01-01') → 2453737
EXTRACT(JULIAN FROM TIMESTAMP '2006-01-01 12:00') → 2453737.50000000000000000000
```

microseconds
The seconds field, including fractional parts, multiplied by 1 000 000; note that this includes full seconds

```
EXTRACT(MICROSECONDS FROM TIME '17:12:28.5') → 28500000
```

millennium
The millennium

```
EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40') → 3
```

Years in the 1900s are in the second millennium. The third millennium started January 1, 2001.

milliseconds
The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds.

```
EXTRACT(MILLISECONDS FROM TIME '17:12:28.5') → 28500.000
```

minute
The minutes field (0–59)

```
EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40') → 38
```

month
For timestamp values, the number of the month within the year (1–12) ; for interval values, the number of months, modulo 12 (0–11)

make_time (hour int, min int, sec double precision) → time	Create time from hour, minute and seconds fields	`make_time(8, 15, 23.5) → 08:15:23.5`
make_timestamp (year int, month int, day int, hour int, min int, sec double precision) → timestamp	Create timestamp from year, month, day, hour, minute and seconds fields (negative years signify BC)	`make_timestamp(2013, 7, 15, 8, 15, 23.5) → 2013-07-15 08:15:23.5`
make_timestamptz (year int, month int, day int, hour int, min int, sec double precision [, timezone text]) → timestamp with time zone	Create timestamp with time zone from year, month, day, hour, minute and seconds fields (negative years signify BC). If timezone is not specified, the current time zone is used; the examples assume the session time zone is Europe/London	`make_timestamptz(2013, 7, 15, 8, 15, 23.5) -→ 2013-07-15 08:15:23.5+01` `#make_timestamptz(2013, 7, 15, 8, 15, 23.5, 'America/New_York') -→ 2013-07-15 13:15:23.5+01`
now ( ) → timestamp with time zone	Current date and time (start of current transaction); see Section 9.9.5	`now() -→ 2019-12-23 14:39:53.662522-05`

2

→ 1

→ 40.000000

rrespond to time zones east of UTC, negative values to zones
econds are not handled.)

tion, ISO weeks start on Mondays and the first week of a year
ar is in week 1 of that year.

to be part of the 52nd or 53rd week of the previous year, and for
mple, 2005-01-01 is part of the 53rd week of year 2004, and
rt of the first week of 2013. It's recommended to use the isoyear

The year field. Keep in mind there is no 0 AD, so subtracting BC years from AD years should be done with care.

```
EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40') → 2001
```

Note
When the input value is +/-Infinity, extract returns +/-Infinity for monotonically-increasing fields (epoch, julian, year, isoyear, decade, century, and millennium). For other fields, NULL is returned. PostgreSQL versions before 9.6 returned zero for all cases of infinite input.

The extract function is primarily intended for computational processing. For formatting date/time values for display, see Section 9.8.

The date_part function is modeled on the traditional Ingres equivalent to the SQL-standard function extract:

```
date_part('field', source)
```

Note that here the field parameter needs to be a string value, not a name. The valid field names for date_part are the same as for extract. For historical reasons, the date_part function returns values of type double precision. This can result in a loss of precision in certain uses. Using extract is recommended instead.

```
date_part('day', TIMESTAMP '2001-02-16 20:38:40') → 16
date_part('hour', INTERVAL '4 hours 3 minutes') → 4
```

9.9.2. date_trunc

The function date_trunc is conceptually similar to the trunc function for numbers.

```
date_trunc(field, source [, time_zone ])
```

source is a value expression of type timestamp, timestamp with time zone, or interval. (Values of type date and time are cast automatically to timestamp or interval, respectively.) field selects to which precision to truncate the input value. The return value is likewise of type timestamp, timestamp with time zone, or interval, and it has all fields that are less significant than the selected one set to zero (or one, for day and month).

Valid values for field are:

statement_timestamp ( ) → timestamp with time zone	Current date and time (start of current statement); see Section 9.9.5	`statement_timestamp() → 2019-12-23 14:39:53.662522-05`
timeofday ( ) → text	Current date and time (like clock_timestamp, but as a text string); see Section 9.9.5	`timeofday() → Mon Dec 23 14:39:53.662522 2019 EST`
transaction_timestamp ( ) → timestamp with time zone	Current date and time (start of current transaction); see Section 9.9.5	`transaction_timestamp() → 2019-12-23 14:39:53.662522-05`
to_timestamp (double precision) → timestamp with time zone	Convert Unix epoch (seconds since 1970-01-01 00:00:00+00) to timestamp with time zone	`to_timestamp(1284352323) → 2010-09-13 04:32:03+00`

is performed with respect to a particular time zone; for example,
lt, truncation is done with respect to the current TimeZone
a different time zone. The time zone name can be specified in any

e zone or interval inputs. These are always taken at face value.

```
-02-16 20:00:00
-01-01 00:00:00
:40+00') ~→ 2001-02-16 00:00:00-05
8:40+00', 'Australia/Sydney') ~→ 2001-02-16 08:00:00-

00:00
```

The function date_bin “bins” the input timestamp into the specified interval (the stride) aligned with a specified origin.

```
date_bin(stride, source, origin)
```

source is a value expression of type timestamp or timestamp with time zone. (Values of type date are cast automatically to timestamp.) stride is a value expression of type interval. The return value is likewise of type timestamp or timestamp with time zone, and it marks the beginning of the bin into which the source is placed.

Examples:

```
date_bin('15 minutes', TIMESTAMP '2020-02-11 15:44:17', TIMESTAMP '2001-01-01') → 2020-02-11 15:30:00
date_bin('15 minutes', TIMESTAMP '2020-02-11 15:44:17', TIMESTAMP '2001-01-01 00:02:30') → 2020-02-11 15:32:30
```

In the case of full units (1 minute, 1 hour, etc.), it gives the same result as the analogous date_trunc call, but the difference is that date_bin can truncate to an arbitrary interval.

The stride interval must be greater than zero and cannot contain units of month or larger.

9.9.4. AT TIME ZONE

The AT TIME ZONE operator converts time stamp without time zone to/from time stamp with time zone, and time with time zone values to different time zones. Table 9.33 shows its variants. (NOT SUPPORTED)

Table 9.33. AT TIME ZONE Variants

Operator	Description	Example(s)	
Converts given time stamp without time zone to time stamp with time zone, assuming the given value is in the named time zone.			
Converts given time stamp with time zone to time stamp without time zone, as the time would appear in that zone.			
Converts given time with time zone to a new time zone. Since no date is supplied, this uses the currently active UTC offset for the named destination zone.			

In these expressions, the desired time zone zone can be specified either as a text value (e.g., 'America/Los_Angeles') or as an interval (e.g., INTERVAL '-08:00'). In the text case, a time zone name can be specified in any of the ways described in Section 8.5.3. The interval case is only useful for zones that have fixed offsets from UTC, so it is not very common in practice.

Examples (assuming the current TimeZone setting is America/Los_Angeles):

```
#TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver' ~→ 2001-02-16 19:38:40-08
#TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'America/Denver' ~→ 2001-02-16 18:38:40
#TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'Asia/Tokyo' AT TIME ZONE 'America/Chicago' ~→ 2001-02-16 05:38:40
```

The first example adds a time zone to a value that lacks it, and displays the value using the current TimeZone setting. The second example shifts the time stamp with time zone value to the specified time zone, and returns the value without a time zone. This allows storage and display of values different from the current TimeZone setting. The third example converts Tokyo time to Chicago time.

The function timezone(zone, timestamp) is equivalent to the SQL-conforming construct timestamp AT TIME ZONE zone.

9.9.5. Current Date/Time

PostgreSQL provides a number of functions that return values related to the current date and time. These SQL-standard functions all return values based on the start time of the current transaction:

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(precision)
LOCALTIMESTAMP(precision)
```

CURRENT_TIME and CURRENT_TIMESTAMP deliver values with time zone; LOCALTIME and LOCALTIMESTAMP deliver values without time zone.

CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME, and LOCALTIMESTAMP can optionally take a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field. Without a precision parameter, the result is given to the full available precision.

Some examples:

```
CURRENT_TIME ~→ 14:39:53.662522-05
CURRENT_DATE ~→ 2019-12-23
CURRENT_TIMESTAMP ~→ 2019-12-23 14:39:53.662522-05
CURRENT_TIMESTAMP(2) ~→ 2019-12-23 14:39:53.66-05
#LOCALTIMESTAMP ~→ 2019-12-23 14:39:53.662522
```

Since these functions return the start time of the current transaction, their values do not change during the transaction. This is considered a feature: the intent is to allow a single transaction to have a consistent notion of the “current” time, so that multiple modifications within the same transaction bear the same time stamp.

Note
Other database systems might advance these values more frequently.

PostgreSQL also provides functions that return the start time of the current statement, as well as the actual current time at the instant the function is called. The complete list of non-SQL-standard time functions is:

```
transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()
now()
```

transaction_timestamp() is equivalent to CURRENT_TIMESTAMP, but is named to clearly reflect what it returns. statement_timestamp() returns the start time of the current statement (more specifically, the time of receipt of the latest command message from the client). statement_timestamp() and transaction_timestamp() return the same value during the first command of a transaction, but might differ during subsequent commands. clock_timestamp() returns the actual current time, and therefore its value changes even within a single SQL command. timeofday() is a historical PostgreSQL function. Like clock_timestamp(), it returns the actual current time, but as a formatted text string rather than a timestamp with time zone value. now() is a traditional PostgreSQL equivalent to transaction_timestamp().

All the date/time data types also accept the special literal value now to specify the current date and time (again, interpreted as the transaction start time). Thus, the following three all return the same result:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';  /* but see tip below */
```

Tip
Do not use the third form when specifying a value to be evaluated later, for example in a DEFAULT clause for a table column. The system will convert now to a timestamp as soon as the constant is parsed, so that when the default value is needed, the time of the table creation would be used! The first two forms will not be evaluated until the default value is used, because they are function calls. Thus they will give the desired behavior of defaulting to the time of row insertion. (See also Section 8.5.1.4.)

9.9.6. Delaying Execution

The following functions are available to delay execution of the server process:

```
pg_sleep ( double precision )
pg_sleep_for ( interval )
pg_sleep_until ( timestamp with time zone )
```

pg_sleep makes the current session's process sleep until the given number of seconds have elapsed. Fractional-second delays can be specified. pg_sleep_for is a convenience function to allow the sleep time to be specified as an interval. pg_sleep_until is a convenience function for when a specific wake-up time is desired. For example:

```
SELECT pg_sleep(1.5);
SELECT pg_sleep_for('5 minutes');
SELECT pg_sleep_until('tomorrow 03:00');
```

Note
The effective resolution of the sleep interval is platform-specific; 0.01 seconds is a common value. The sleep delay will be at least as long as specified. It might be longer depending on factors such as server load. In particular, pg_sleep_until is not guaranteed to wake up exactly at the specified time, but it will not wake up any earlier.

Warning
Make sure that your session does not hold more locks than necessary when calling pg_sleep or its variants. Otherwise other sessions might have to wait for your sleeping process, slowing down the entire system.

9.10. Enum Support Functions (NOT SUPPORTED)

9.11. Geometric Functions and Operators

The geometric types point, box, lseg, line, path, polygon, and circle have a large set of native support functions and operators, shown in Table 9.35, Table 9.36, and Table 9.37.

Table 9.35. Geometric Operators

Operator	Description	Example(s)
geometric_type + point → geometric_type	Adds the coordinates of the second point to those of each point of the first argument, thus performing translation. Available for point, box, path, circle.	`box '(1,1),(0,0)' + point '(2,0)' → (3,1),(2,0)`

path + path → path	Concatenates two open paths (returns NULL if either path is closed).	`path '[(0,0),(1,1)]' + path '[(2,2),(3,3),(4,4)]' → [(0,0),(1,1),(2,2),(3,3),(4,4)]`
geometric_type - point → geometric_type	Subtracts the coordinates of the second point from those of each point of the first argument, thus performing translation. Available for point, box, path, circle.	`box '(1,1),(0,0)' - point '(2,0)' → (-1,1),(-2,0)`
geometric_type * point → geometric_type	Multiplies each point of the first argument by the second point (treating a point as being a complex number represented by real and imaginary parts, and performing standard complex multiplication). If one interprets the second point as a vector, this is equivalent to scaling the object's size and distance from the origin by the length of the vector, and rotating it counterclockwise around the origin by the vector's angle from the x axis. Available for point, box,[a] path, circle.	`path '((0,0),(1,0),(1,1))' * point '(3.0,0)' → ((0,0),(3,0),(3,3))` `path '((0,0),(1,0),(1,1))' * point(cosd(45), sind(45)) → ((0,0),(0.7071067811865475,0.7071067811865475),(0,1.414213562373095))`
geometric_type / point → geometric_type	Divides each point of the first argument by the second point (treating a point as being a complex number represented by real and imaginary parts, and performing standard complex division). If one interprets the second point as a vector, this is equivalent to scaling the object's size and distance from the origin down by the length of the vector, and rotating it clockwise around the origin by the vector's angle from the x axis. Available for point, box,[a] path, circle.	`path '((0,0),(1,0),(1,1))' / point '(2.0,0)' → ((0,0),(0.5,0),(0.5,0.5))` `path '((0,0),(1,0),(1,1))' / point(cosd(45), sind(45)) → ((0,0),(0.7071067811865476,-0.7071067811865476),(1.4142135623730951,0))`

@-@ geometric_type → double precision	Computes the total length. Available for lseg, path.	`@-@ path '[(0,0),(1,0),(1,1)]' → 2`
@@ geometric_type → point	Computes the center point. Available for box, lseg, polygon, circle.	`@@ box '(2,2),(0,0)' → (1,1)`
# geometric_type → integer	Returns the number of points. Available for path, polygon.	`# path '((1,0),(0,1),(-1,0))' → 3`
geometric_type # geometric_type → point	Computes the point of intersection, or NULL if there is none. Available for lseg, line.	`lseg '[(0,0),(1,1)]' # lseg '[(1,0),(0,1)]' → (0.5,0.5)`
box # box → box	Computes the intersection of two boxes, or NULL if there is none.	`box '(2,2),(-1,-1)' # box '(1,1),(-2,-2)' → (1,1),(-1,-1)`

geometric_type ## geometric_type → point	Computes the closest point to the first object on the second object. Available for these pairs of types: (point, box), (point, lseg), (point, line), (lseg, box), (lseg, lseg), (line, lseg).	`point '(0,0)' ## lseg '[(2,0),(0,2)]' → (1,1)`
geometric_type <-> geometric_type → double precision	Computes the distance between the objects. Available for all geometric types except polygon, for all combinations of point with another geometric type, and for these additional pairs of types: (box, lseg), (lseg, line), (polygon, circle) (and the commutator cases).	`circle '<(0,0),1>' <-> circle '<(5,0),1>' → 3`
geometric_type @> geometric_type → boolean	Does first object contain second? Available for these pairs of types: (box, point), (box, box), (path, point), (polygon, point), (polygon, polygon), (circle, point), (circle, circle).	`circle '<(0,0),2>' @> point '(1,1)' → true`
geometric_type <@ geometric_type → boolean	Is first object contained in or on second? Available for these pairs of types: (point, box), (point, lseg), (point, line), (point, path), (point, polygon), (point, circle), (box, box), (lseg, box), (lseg, line), (polygon, polygon), (circle, circle).	`point '(1,1)' <@ circle '<(0,0),2>' → true`
geometric_type && geometric_type → boolean	Do these objects overlap? (One point in common makes this true.) Available for box, polygon, circle.	`box '(1,1),(0,0)' && box '(2,2),(0,0)' → true`

[a] “Rotating” a box with these operators only moves its corner points: the box is still considered to have sides parallel to the axes. Hence the box's size is not preserved, as a true rotation would do.

geometric_type << geometric_type → boolean	Is first object strictly left of second? Available for point, box, polygon, circle.	`circle '<(0,0),1>' << circle '<(5,0),1>' → true`
geometric_type >> geometric_type → boolean	Is first object strictly right of second? Available for point, box, polygon, circle.	

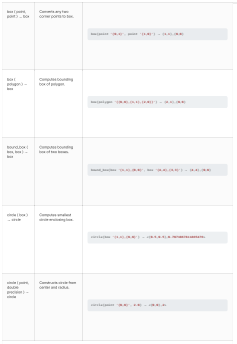
equality for the point, box, polygon, and circle types. Some of the areas only. The other scalar comparison operators (<= and so on),

operators point <<| point and point |>> point were respectively called ill eventually be removed.

Function	Description	Example(s)
area (geometric_type) → double precision	Computes area. Available for box, path, circle. A path input must be closed, else NULL is returned. Also, if the path is self-intersecting, the result may be meaningless.	`area(box '(2,2),(0,0)') → 4`
center (geometric_type) → point	Computes center point. Available for box, circle.	`center(box '(1,2),(0,0)') → (0.5,1)`
diagonal (box) → lseg	Extracts box's diagonal as a line segment (same as lseg(box)).	`diagonal(box '(1,2),(0,0)') → [(1,2),(0,0)]`
diameter (circle) → double precision	Computes diameter of circle.	`diameter(circle '<(0,0),2>') → 4`

height (box) → double precision	Computes vertical size of box.	`height(box '(1,2),(0,0)') → 2`
isclosed (path) → boolean	Is path closed?	`isclosed(path '((0,0),(1,1),(2,0))') → true`
isopen (path) → boolean	Is path open?	`isopen(path '[(0,0),(1,1),(2,0)]') → true`
length (geometric_type) → double precision	Computes the total length. Available for lseg, path.	`length(path '((-1,0),(1,0))') → 4`
npoints (geometric_type) → integer	Returns the number of points. Available for path, polygon.	`npoints(path '[(0,0),(1,1),(2,0)]') → 3`

pclose (path) → path	Converts path to closed form.	`pclose(path '[(0,0),(1,1),(2,0)]') → ((0,0),(1,1),(2,0))`
popen (path) → path	Converts path to open form.	`popen(path '((0,0),(1,1),(2,0))') → [(0,0),(1,1),(2,0)]`
radius (circle) → double precision	Computes radius of circle.	`radius(circle '<(0,0),2>') → 2`
 double precision	Computes slope of a line drawn through the two points.	`slope(point '(0,0)', point '(2,1)') → 0.5`



Computes horizontal size of box.

```
width(box '(1,2),(0,0)') →
1
```

geometric_type ~= geometric_type → boolean	Are these objects the same? Available for point, box, polygon, circle.	`polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))' → true`

arison operators shown in Table 9.1 as well as the specialized

and functions shown below as operating on inet also work on cidr cause the behavior should be different for the two cases.) Also, it is the right of the netmask are silently zeroed to create a valid cidr value.

Operator	Description	Example(s)

inet << inet → boolean	Is subnet strictly contained by subnet? This operator, and the next four, test for subnet inclusion. They consider only the network parts of the two addresses (ignoring any bits to the right of the netmasks) and determine whether one network is identical to or a subnet of the other.	`inet '192.168.1.5' << inet '192.168.1/24' → true`
		`inet '192.168.0.5' << inet '192.168.1/24' → false`
		`inet '192.168.1/24' << inet '192.168.1/24' → false`

inet <<= inet → boolean	Is subnet contained by or equal to subnet?	`inet '192.168.1/24' <<= inet '192.168.1/24' → true`
inet >> inet → boolean	Does subnet strictly contain subnet?	`inet '192.168.1/24' >> inet '192.168.1.5' → true`

inet >>= inet → boolean	Does subnet contain or equal subnet?	`inet '192.168.1/24' >>= inet '192.168.1/24' → true`
inet && inet → boolean	Does either subnet contain or equal the other?	`inet '192.168.1/24' && inet '192.168.1.80/28' → true` `inet '192.168.1/24' && inet '192.168.2.0/28' → false`
~ inet → inet	Computes bitwise NOT.	`~ inet '192.168.1.6' → 63.87.254.249`
inet & inet → inet	Computes bitwise AND.	`inet '192.168.1.6' & inet '0.0.0.255' → 0.0.0.6`

inet | inet → inet	Computes bitwise OR.	`inet '192.168.1.6' | inet '0.0.0.255' → 192.168.1.255`
inet + bigint → inet	Adds an offset to an address.	`inet '192.168.1.6' + 25 → 192.168.1.31`
bigint + inet → inet	Adds an offset to an address. (NOT SUPPORTED)	`#200 + inet '::ffff:fff0:1' → ::ffff:255.240.0.201`
inet - bigint → inet	Subtracts an offset from an address.	`inet '192.168.1.43' - 36 → 192.168.1.7`

Function	Description	Example(s)
abbrev (inet) → text	Creates an abbreviated display format as text. (The result is the same as the inet output function produces; it is "abbreviated" only in comparison to the result of an explicit cast to text, which for historical reasons will never suppress the netmask part.)	`abbrev(inet '10.1.0.0/32') → 10.1.0.0`

abbrev (cidr) → text	Creates an abbreviated display format as text. (The abbreviation consists of dropping all-zero octets to the right of the netmask; more examples are in Table 8.22.)	`abbrev(cidr '10.1.0.0/16') → 10.1/16`
broadcast (inet) → inet	Computes the broadcast address for the address's network.	`broadcast(inet '192.168.1.5/24') → 192.168.1.255/24`
family (inet) → integer	Returns the address's family: 4 for IPv4, 6 for IPv6.	`family(inet '::1') → 6`
host (inet) → text	Returns the IP address as text, ignoring the netmask.	`host(inet '192.168.1.0/24') → 192.168.1.0`
hostmask (inet) → inet	Computes the host mask for the address's network.	`hostmask(inet '192.168.23.20/30') → 0.0.0.3`

```
'192.168.1.43' - inet
.168.1.19' → 24
'::1' - inet
fff:1' → -4294901760
```

inet_merge (inet, inet) → cidr	Computes the smallest network that includes both of the given networks.	`inet_merge(inet '192.168.1.5/24', inet '192.168.2.5/24') → 192.168.0.0/22`
inet_same_family (inet, inet) → boolean	Tests whether the addresses belong to the same IP family.	`inet_same_family(inet '192.168.1.5/24', inet '::1') → false`
masklen (inet) → integer	Returns the netmask length in bits.	`masklen(inet '192.168.1.5/24') → 24`
netmask (inet) → inet	Computes the network mask for the address's network.	`netmask(inet '192.168.1.5/24') → 255.255.255.0`

addresses.

The MAC address types, macaddr and macaddr8, support the usual comparison operators shown in Table 9.1 as well as the specialized functions shown in Table 9.40. In addition, they support the bitwise logical operators ~, & and | (NOT, AND and OR), just as shown above for IP addresses.

Table 9.40. MAC Address Functions

Function	Description	Example(s)

trunc (macaddr) → macaddr	Sets the last 3 bytes of the address to zero. The remaining prefix can be associated with a particular manufacturer (using data not included in PostgreSQL).	`trunc(macaddr '12:34:56:78:90:ab') → 12:34:56:00:00:00`
trunc (macaddr8) → macaddr8	Sets the last 5 bytes of the address to zero. The remaining prefix can be associated with a particular manufacturer (using data not included in PostgreSQL).	`trunc(macaddr8 '12:34:56:78:90:ab:cd:ef') → 12:34:56:00:00:00:00:00`
macaddr8_set7bit (macaddr8) → macaddr8	Sets the 7th bit of the address to one, creating what is known as modified EUI-64, for inclusion in an IPv6 address.	`macaddr8_set7bit(macaddr8 '00:34:56:ab:cd:ef') → 02:34:56:ff:fe:ab:cd:ef`

s appropriate for most applications.

text (inet) → text	Returns the unabbreviated IP address and netmask length as text. (This has the same result as an explicit cast to text.)	`text(inet '192.168.1.5') → 192.168.1.5/32`

See Section 8.13 for information about
e xml are documented there, not in

ta. As such, they are particularly

9.15.1.1. Xmlcomment

xmlcomment (text) → xml

The function xmlcomment creates an XML value containing an XML comment with the specified text as content. The text cannot contain “--” or end with a “-”, otherwise the resulting construct would not be a valid XML comment. If the argument is null, the result is null.

Example:

```
xmlcomment('hello') → <!--hello-->
```

9.15.1.2. Xmlconcat

xmlconcat (xml [, ...]) → xml (NOT SUPPORTED)

The function xmlconcat concatenates a list of individual XML values to create a single value containing an XML content fragment. Null values are omitted; the result is only null if there are no nonnull arguments.

Example:

```
#xmlconcat('<abc/>', '<bar>foo</bar>') → <abc/><bar>foo</bar>
```

XML declarations, if present, are combined as follows. If all argument values have the same XML version declaration, that version is used in the result, else no version is used. If all argument values have the standalone declaration value “yes”, then that value is used in the result. If all argument values have a standalone declaration value and at least one is “no”, then that is used in the result. Else the result will have no standalone declaration. If the result is determined to require a standalone declaration but no version declaration, a version declaration with version 1.0 will be used because XML requires an XML declaration to contain a version declaration. Encoding declarations are ignored and removed in all cases.

Example:

```
#xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1" standalone="no"?><bar/>') → <?xml version="1.1"?>
<foo/><bar/>
```

9.15.1.3. Xmlelement

xmlelement (NAME name [, XMLATTRIBUTES (attvalue [AS attname] [, ...])] [, content [, ...]]) → xml (NOT SUPPORTED)

```
#xmlelement(name foo) → <foo/>
#xmlelement(name foo, xmlattributes('xyz' as bar)) → <foo bar="xyz"/>
#xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent') ~→ <foo bar="2007-01-26">content</foo>
```

Element and attribute names that are not valid XML names are escaped by replacing the offending characters by the sequence *xHHHH*, where HHHH is the character's Unicode codepoint in hexadecimal notation. For example:

```
#xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b")) → <foo_x0024_bar a_x0026_b="xyz"/>
```

An explicit attribute name need not be specified if the attribute value is a column reference, in which case the column's name will be used as the attribute name by default. In other cases, the attribute must be given an explicit name. So this example is valid:

Element content, if specified, will be formatted according to its data type. If the content is itself of type xml, complex XML documents can be constructed. For example:

```
#xmlelement(name foo, xmlattributes('xyz' as bar), xmlelement(name abc), xmlcomment('test'), xmlelement(name xyz))
→ <foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

Content of other types will be formatted into valid XML character data. This means in particular that the characters <, >, and & will be converted to entities. Binary data (data type bytea) will be represented in base64 or hex encoding, depending on the setting of the configuration parameter xmlbinary. The particular behavior for individual data types is expected to evolve in order to align the PostgreSQL mappings with those specified in SQL:2006 and later, as discussed in Section D.3.1.3.

9.15.1.4. Xmlforest

xmlforest (content [AS name] [, ...]) → xml (NOT SUPPORTED)

9.15.1.5. Xmlpi

xmlpi (NAME name [, content]) → xml (NOT SUPPORTED)

9.15.1.6. Xmlroot

xmlroot (xml, VERSION {text|NO VALUE} [, STANDALONE {YES|NO|NO VALUE}]) → xml (NOT SUPPORTED)

9.15.1.7. Xmlagg

xmlagg (xml) → xml

The function xmlagg is, unlike the other functions described here, an aggregate function. It concatenates the input values to the aggregate function call, much like xmlconcat does, except that concatenation occurs across rows rather than across expressions in a single row. See Section 9.21 for additional information about aggregate functions.

Example:

```
SELECT xmlagg(x::xml) from (values ('<a/>'),('<b/>')) a(x) → <a/><b/>
```

9.15.2. XML Predicates
The expressions described in this section check properties of xml values.

9.15.2.1. IS DOCUMENT

xml IS DOCUMENT → boolean (NOT SUPPORTED)

The expression IS DOCUMENT returns true if the argument XML value is a proper XML document, false if it is not (that is, it is a content fragment), or null if the argument is null. See Section 8.13 about the difference between documents and content fragments.

9.15.2.2. IS NOT DOCUMENT

xml IS NOT DOCUMENT → boolean (NOT SUPPORTED)

The expression IS NOT DOCUMENT returns false if the argument XML value is a proper XML document, true if it is not (that is, it is a content fragment), or null if the argument is null.

9.15.2.3. XMLEXISTS

XMLEXISTS (text PASSING [BY {REF|VALUE}] xml [BY {REF|VALUE}]) → boolean

The function xmlexists evaluates an XPath 1.0 expression (the first argument), with the passed XML value as its context item. The function returns false if the result of that evaluation yields an empty node-set, true if it yields any other value. The function returns null if any

argument is null. A nonnull value passed as the context item must be an XML document, not a content fragment or any non-XML value.

Example:

```
xmlexists('//town[text() = ''Toronto'']' PASSING BY VALUE '<towns><town>Toronto</town><town>Ottawa</town>
</towns>') → true
```

9.15.2.4. Xml_is_well_formed (NOT SUPPORTED)

xml_is_well_formed (text) → boolean
xml_is_well_formed_document (text) → boolean
xml_is_well_formed_content (text) → boolean

9.15.3. Processing XML
To process values of data type xml, PostgreSQL offers the functions xpath and xpath_exists, which evaluate XPath 1.0 expressions, and the XMLTABLE table function.

9.15.3.1. Xpath

xpath (xpath text, xml xml [, nsarray text[]]) → xml[] (NOT SUPPORTED)

9.15.3.2. Xpath_exists

xpath_exists (xpath text, xml xml [, nsarray text[]]) → boolean

The function xpath_exists is a specialized form of the xpath function. Instead of returning the individual XML values that satisfy the XPath 1.0 expression, this function returns a Boolean indicating whether the query was satisfied or not (specifically, whether it produced any value other than an empty node-set). This function is equivalent to the XMLEXISTS predicate, except that it also offers support for a namespace mapping argument.

Example:

```
xpath_exists('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',ARRAY[ARRAY['my',
'http://example.com']]) → true
```

9.15.3.3. Xmltable

XMLTABLE (
[XMLNAMESPACES (namespace_uri AS namespace_name [, ...]),]
row_expression PASSING [BY {REF|VALUE}] document_expression [BY {REF|VALUE}]
COLUMNS name { type [PATH column_expression] [DEFAULT default_expression] [NOT NULL | NULL]
| FOR ORDINALITY }
[, ...]
) → setof record (NOT SUPPORTED)

9.15.4. Mapping Tables to XML (NOT SUPPORTED)

The following functions map the contents of relational tables to XML values. They can be thought of as XML export functionality:

table_to_xml (table regclass, nulls boolean,
tableforest boolean, targetns text) → xml
query_to_xml (query text, nulls boolean,
tableforest boolean, targetns text) → xml
cursor_to_xml (cursor refcursor, count integer, nulls boolean,
tableforest boolean, targetns text) → xml

9.16. JSON Functions and Operators

This section describes:

functions and operators for processing and creating JSON data

the SQL/JSON path language

To learn more about the SQL/JSON standard, see [sqltr-19075-6]. For details on JSON types supported in PostgreSQL, see Section 8.14.

9.16.1. Processing and Creating JSON Data

Table 9.44 shows the operators that are available for use with JSON data types (see Section 8.14). In addition, the usual comparison operators shown in Table 9.1 are available for jsonb, though not for json. The comparison operators follow the ordering rules for B-tree operations outlined in Section 8.14.4. See also Section 9.21 for the aggregate function json_agg which aggregates record values as JSON, the aggregate function json_object_agg which aggregates pairs of values into a JSON object, and their jsonb equivalents, jsonb_agg and jsonb_object_agg.

Table 9.44. json and jsonb Operators

Operator	Description	Example(s)

json → integer → json jsonb → integer → jsonb	Extracts n'th element of JSON array (array elements are indexed from zero, but negative integers count from the end).	`'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json -> 2 → {"c":"baz"}` `'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json -> -3 → {"a":"foo"}`
json → text → json jsonb → text → jsonb	Extracts JSON object field with the given key.	`'{"a": {"b":"foo"}}'::json -> 'a' → {"b":"foo"}`
json →> integer → text jsonb →> integer → text	Extracts n'th element of JSON array, as text.	`'[1,2,3]'::json ->> 2 → 3`
json →> text → text jsonb →> text → text	Extracts JSON object field with the given key, as text.	`'{"a":1,"b":2}'::json ->> 'b' → 2`

Note
The field/element/path extraction operators return NULL, rather than failing, if the JSON input does not have the right structure to match the request; for example if no such key or array element exists.

an be used to

json #> text[] → json jsonb #> text[]	Extracts JSON sub-object at the specified path, where path elements can be either field keys or array indexes. (NOT SUPPORTED)	

Operator	Description	Example(s)
jsonb @> jsonb → boolean	Does the first JSON value contain the second? (See Section 8.14.3 for details about containment.)	`'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb → true`
jsonb <@ jsonb → boolean	Is the first JSON value contained in the second?	`'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb → true`
jsonb ? text → boolean	Does the text string exist as a top-level key or array element within the JSON value?	`'{"a":1, "b":2}'::jsonb ? 'b' → true` `'["a", "b", "c"]'::jsonb ? 'b' → true`
jsonb ?| text[] → boolean	Do any of the strings in the text array exist as top-level keys or array elements?	`'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'd'] → true`

jsonb ?& text[] → boolean	Do all of the strings in the text array exist as top-level keys or array elements?	`'["a", "b", "c"]'::jsonb ?& array['a', 'b'] → true`

jsonb || jsonb → jsonb	Concatenates two jsonb values. Concatenating two arrays generates an array containing all the elements of each input. Concatenating two objects generates an object containing the union of their keys, taking the second object's value when there are duplicate keys. All other cases are treated by converting a non-array input into a single-element array, and then proceeding as for two arrays. Does not operate recursively: only the top-level array or object structure is merged.	`'["a", "b"]'::jsonb || '["a", "d"]'::jsonb → ["a", "b", "a", "d"]`
		`'{"a": "b"}'::jsonb || '{"c": "d"}'::jsonb → {"a": "b", "c": "d"}`
		`'[1, 2]'::jsonb || '3'::jsonb → [1, 2, 3]`
		`'{"a": "b"}'::jsonb || '42'::jsonb → [{"a": "b"}, 42]`
		
		To append an array to another array as a single entry, wrap it in an additional layer of array, for example:
		
		`'[1, 2]'::jsonb || jsonb_build_array('[3, 4]'::jsonb) → [1, 2, [3, 4]]`

jsonb - text → jsonb	Deletes a key (and its value) from a JSON object, or matching string value(s) from a JSON array.	`'{"a": "b", "c": "d"}'::jsonb - 'a' → {"c": "d"}` `'["a", "b", "c", "b"]'::jsonb - 'b' → ["a", "c"]`

jsonb - text[] → jsonb	Deletes all matching keys or array elements from the left operand. (NOT SUPPORTED)	`#'{"a": "b", "c": "d"}'::jsonb - '{a,c}'::text[] → {}`
jsonb - integer → jsonb	Deletes the array element with specified index (negative integers count from the end). Throws an error if JSON value is not an array.	`'["a", "b"]'::jsonb - 1 → ["a"]`

xpected JSON item type,
ese types of errors. This

Function	Description	Example(s)
to_json (anyelement) → json to_jsonb (anyelement) → jsonb	Converts any SQL value to json or jsonb. Arrays and composites are converted recursively to arrays and objects (multidimensional arrays become arrays of arrays in JSON). Otherwise, if there is a cast from the SQL data type to json, the cast function will be used to perform the conversion;[a] otherwise, a scalar JSON value is produced. For any scalar other than a number, a Boolean, or a null value, the text representation will be used, with escaping as necessary to make it a valid JSON string value. (NOT SUPPORTED)	`#to_json('Fred said "Hi."'::text) → "Fred said \"Hi.\""` `#to_jsonb(row(42, 'Fred said "Hi."'::text)) → {"f1": 42, "f2": "Fred said \"Hi.\""}`
array_to_json (anyarray [, boolean]) → json	Converts an SQL array to a JSON array. The behavior is the same as to_json except that line feeds will be added between top-level array elements if the optional boolean parameter is true.	`array_to_json('{{1,5}, {99,100}}'::int[]) → [[1,5], [99,100]]`

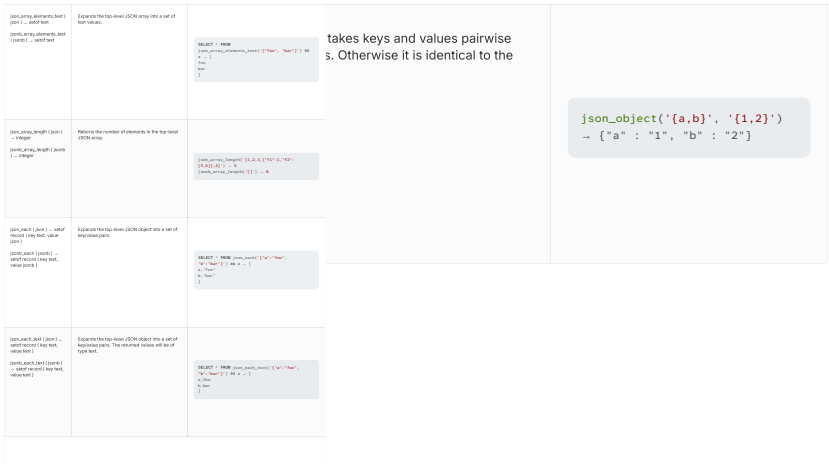
`':1}]'::jsonb #- ["a", {}]`

`3,4,5]}'::jsonb @? @ > 2)' → true`

`3,4,5]}'::jsonb @@ ' → true`

row_to_json (record [, boolean]) → json	Converts an SQL composite value to a JSON object. The behavior is the same as to_json except that line feeds will be added between top-level elements if the optional boolean parameter is true. (NOT SUPPORTED)	`#row_to_json(row(1,'foo')) → {"f1":1,"f2":"foo"}`
json_build_array (VARIADIC "any") → json jsonb_build_array (VARIADIC "any") → jsonb	Builds a possibly-heterogeneously-typed JSON array out of a variadic argument list. Each argument is converted as per to_json or to_jsonb. (NOT SUPPORTED)	`json_build_array(1, 2, 'foo', 4, 5) → [1, 2, "foo", 4, 5]`
json_build_object (VARIADIC "any") → json jsonb_build_object (VARIADIC "any") → jsonb	Builds a JSON object out of a variadic argument list. By convention, the argument list consists of alternating keys and values. Key arguments are coerced to text; value arguments are converted as per to_json or to_jsonb. (NOT SUPPORTED)	`#json_build_object('foo', 1, 2, row(3,'bar')) → {"foo" : 1, "2" : {"f1":3,"f2":"bar"}}`
json_object (text[]) → json jsonb_object (text[]) → jsonb	Builds a JSON object out of a text array. The array must have either exactly one dimension with an even number of members, in which case they are taken as alternating key/value pairs, or two dimensions such that each inner array has exactly two elements, which are taken as a key/value pair. All values are converted to JSON strings.	`json_object('{a, 1, b, "def", c, 3.5}') → {"a" : "1", "b" : "def", "c" : "3.5"}` `json_object('{{a, 1}, {b, "def"}, {c, 3.5}}') → {"a" : "1", "b" : "def", "c" : "3.5"}`



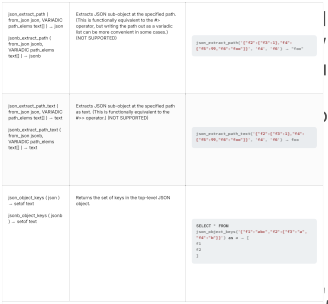


9.16.2. The SQL/JSON Path Language

SQL/JSON path expressions specify the items to be retrieved from the JSON data, similar to XPath expressions used for SQL access to XML. In PostgreSQL, path expressions are implemented as the jsonpath data type and can use any elements described in Section 8.14.7.

JSON query functions and operators pass the provided path expression to the path engine for evaluation. If the expression matches the queried JSON data, the corresponding JSON item, or set of items, is returned. Path expressions are written in the SQL/JSON path language and can include arithmetic expressions and functions.

A path expression consists of a sequence of elements allowed by the jsonpath data type. The path expression is normally evaluated from left to right, but you can use parentheses to change the order of operations. If the evaluation is successful, a sequence of JSON items is produced, and the evaluation result is returned to the JSON query function that completes the specified computation.



queried (the context item), use the $ variable in the path expression. It can be followed by one or more
n the JSON structure level by level to retrieve sub-items of the context item. Each operator that follows
s evaluation step.

me JSON data from a GPS tracker that you would like to parse, such as:

}
]
}
}
To retrieve the available track segments, you need to use the .key accessor operator to descend through surrounding JSON objects:

$.track.segments
To retrieve the contents of an array, you typically use the [*] operator. For example, the following path will return the location coordinates for all the available track segments:

$.track.segments[*].location
To return the coordinates of the first segment only, you can specify the corresponding subscript in the [] accessor operator. Recall that JSON array indexes are 0-relative:

$.track.segments[0].location
The result of each path evaluation step can be processed by one or more jsonpath operators and methods listed in Section 9.16.2.2. Each method name must be preceded by a dot. For example, you can get the size of an array:

$.track.segments.size()
More examples of using jsonpath operators and methods within path expressions appear below in Section 9.16.2.2.

When defining a path, you can also use one or more filter expressions that work similarly to the WHERE clause in SQL. A filter expression begins with a question mark and provides a condition in parentheses:

? (condition)
Filter expressions must be written just after the path evaluation step to which they should apply. The result of that step is filtered to include only those items that satisfy the provided condition. SQL/JSON defines three-valued logic, so the condition can be true, false, or unknown. The unknown value plays the same role as SQL NULL and can be tested for with the is unknown predicate. Further path evaluation steps use only those items for which the filter expression returned true.

The functions and operators that can be used in filter expressions are listed in Table 9.49. Within a filter expression, the @ variable denotes the value being filtered (i.e., one result of the preceding path step). You can write accessor operators after @ to retrieve component items.

For example, suppose you would like to retrieve all heart rate values higher than 130. You can achieve this using the following expression:

$.track.segments[*].HR ? (@ > 130)
To get the start times of segments with such values, you have to filter out irrelevant segments before returning the start times, so the filter expression is applied to the previous step, and the path used in the condition is different:

$.track.segments[*] ? (@.HR > 130)."start time"
You can use several filter expressions in sequence, if required. For example, the following expression selects start times of all segments that contain locations with relevant coordinates and high heart rate values:

$.track.segments[*] ? (@.location[1] < 13.4) ? (@.HR > 130)."start time"
Using filter expressions at different nesting levels is also allowed. The following example first filters all segments by location, and then returns high heart rate values for these segments, if available:

$.track.segments[*] ? (@.location[1] < 13.4).HR ? (@ > 130)
You can also nest filter expressions within each other:

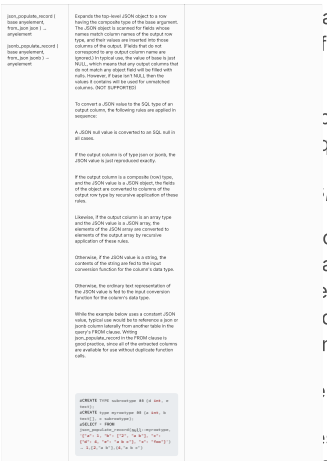
$.track ? (exists(@.segments[*] ? (@.HR > 130))).segments.size()
This expression returns the size of the track if it contains any segments with high heart rate values, or an empty sequence otherwise.

PostgreSQL's implementation of the SQL/JSON path language has the following deviations from the SQL/JSON standard:

A path expression can be a Boolean predicate, although the SQL/JSON standard allows predicates only in filters. This is necessary for implementation of the @@ operator. For example, the following jsonpath expression is valid in PostgreSQL:

$.track.segments[*].HR < 70
There are minor differences in the interpretation of regular expression patterns used in like_regex filters, as described in Section 9.16.2.3.

9.16.2.1. Strict And Lax Modes



ath expression may not match the actual JSON data structure. An attempt to access a non-existent f an array results in a structural error. SQL/JSON path expressions have two modes of handling structural

olicitly adapts the queried data to the specified path. Any remaining structural errors are suppressed and quences.

, an error is raised.

of a JSON document structure and path expression if the JSON data does not conform to the expected atch the requirements of a particular operation, it can be automatically wrapped as an SQL/JSON array ements into an SQL/JSON sequence before performing this operation. Besides, comparison operators ds in the lax mode, so you can compare SQL/JSON arrays out-of-the-box. An array of size 1 is nt. Automatic unwrapping is not performed only when:

() or size() methods that return the type and the number of elements in the array, respectively.

sted arrays. In this case, only the outermost array is unwrapped, while all the inner arrays remain unchanged. Thus, implicit unwrapping can only go one level down within each path evaluation step.

For example, when querying the GPS data listed above, you can abstract from the fact that it stores an array of segments when using the lax mode:

lax $.track.segments.location
In the strict mode, the specified path must exactly match the structure of the queried JSON document to return an SQL/JSON item, so using this path expression will cause an error. To get the same result as in the lax mode, you have to explicitly unwrap the segments array:

strict $.track.segments[*].location
The .** accessor can lead to surprising results when using the lax mode. For instance, the following query selects every HR value twice:

lax $.**.HR
This happens because the .** accessor selects both the segments array and each of its elements, while the .HR accessor automatically unwraps arrays when using the lax mode. To avoid surprising results, we recommend using the .** accessor only in the strict mode. The following query selects each HR value just once:

strict $.**.HR

9.16.2.2. SQL/JSON Path Operators And Methods

Table 9.48 shows the operators and methods available in jsonpath. Note that while the unary operators and methods can be applied to multiple values resulting from a preceding path step, the binary operators (addition etc.) can only be applied to single values.

Table 9.48. jsonpath Operators and Methods

Operator/Method	Description	Example(s)
number + number → number	Addition	`jsonb_path_query_array('[2]', '$[0] + 3', '{}', false)` → `[5]`
+ number → number	Unary plus (no operation); unlike addition, this can iterate over multiple values	`jsonb_path_query_array('{"x": [2,3,4]}', '+ $.x', '{}', false)` → `[2, 3, 4]`

number - number → number	Subtraction	`jsonb_path_query_array('[2]', '7 - $[0]', '{}', false)` → `[5]`
- number → number	Negation; unlike subtraction, this can iterate over multiple values	`jsonb_path_query_array('{"x": [2,3,4]}', '-$.x', '{}', false)` → `[-2, -3, -4]`
number * number → number	Multiplication	`jsonb_path_query_array('[4]', '2 * $[0]', '{}', false)` → `[8]`
number / number → number	Division	`jsonb_path_query_array('[8.5]', '$[0] / 2', '{}', false)` → `[4.250000000000000]`
number % number → number	Modulo (remainder)	`jsonb_path_query_array('[32]', '$[0] % 10', '{}', false)` → `[2]`

Note
The result type of the datetime() and datetime(template) methods can be date, timetz, time, timestamptz, or timestamp. Both methods determine their result type dynamically.

The datetime() method sequentially tries to match its input string to the ISO formats for date, timetz, time, timestamptz, and timestamp. It stops on the first matching format and emits the corresponding data type.

The datetime(template) method determines the result type according to the fields used in the provided template string.

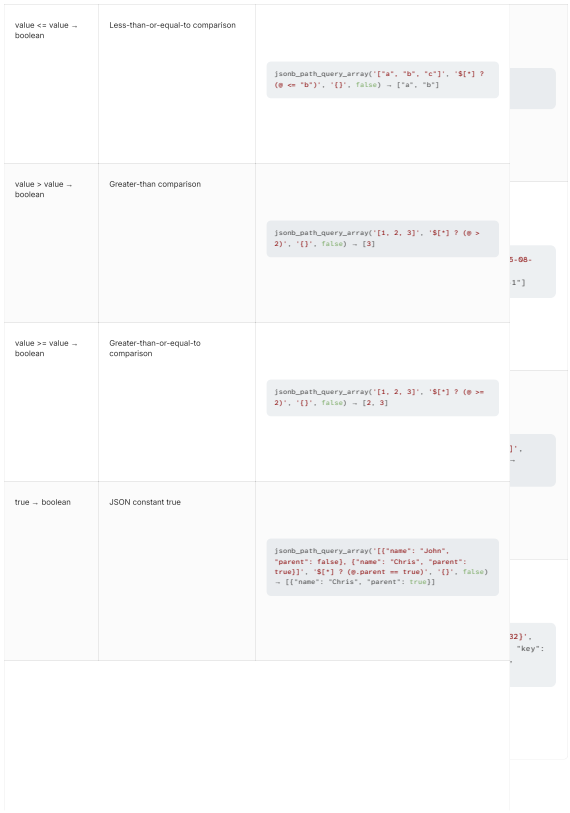
The datetime() and datetime(template) methods use the same parsing rules as the to_timestamp SQL function does (see Section 9.8), with three exceptions. First, these methods don't allow unmatched template patterns. Second, only the following separators are allowed in the

apostrophe, semicolon, colon and space. Third, separators in the template

cast is applied. A date value can be cast to timestamp or timestamptz, owever, all but the first of these conversions depend on the current TimeZone are jsonpath functions.

value . type() → string	Type of the JSON item (see json_typeof)	`jsonb_path_query_array('[1, "2", {}]', '$[*].type()', '{}', false) → ["number", "string", "object"]`
value . size() → number	Size of the JSON item (number of array elements, or 1 if not an array)	

Predicate/Value	Description	Example(s)
value == value → boolean	Equality comparison (this, and the other comparison operators, work on all JSON scalar values)	`jsonb_path_query_array('[1, "a", 1, 3]', '$[*] ? (@ == 1)', '{}', false) → [1, 1]` `jsonb_path_query_array('[1, "a", 1, 3]', '$[*] ? (@ == "a")', '{}', false) → ["a"]`
value != value → boolean value <> value → boolean	Non-equality comparison	`jsonb_path_query_array('[1, 2, 1, 3]', '$[*] ? (@ != 1)', '{}', false) → [2, 3]` `jsonb_path_query_array('["a", "b", "c"]', '$[*] ? (@ <> "b")', '{}', false) → ["a", "c"]`
value < value → boolean	Less-than comparison	`jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ < 2)', '{}', false) → [1]`



9.16.2.3. SQL/JSON Regular Expressions
SQL/JSON path expressions allow matching text to a regular expression with the like_regex filter. For example, the following SQL/JSON path query would case-insensitively match all strings in an array that start with an English vowel:

$[*] ? (@ like_regex "^[aeiou]" flag "i")

The optional flag string may include one or more of the characters i for case-insensitive match, m to allow ^ and $ to match at newlines, s to allow . to match a newline, and q to quote the whole pattern (reducing the behavior to a simple substring match).

The SQL/JSON standard borrows its definition for regular expressions from the LIKE_REGEX operator, which in turn uses the XQuery standard. PostgreSQL does not currently support the LIKE_REGEX operator. Therefore, the like_regex filter is implemented using the POSIX regular expression engine described in Section 9.7.3. This leads to various minor discrepancies from standard SQL/JSON behavior, which are cataloged in Section 9.7.3.8. Note, however, that the flag-letter incompatibilities described there do not apply to SQL/JSON, as it translates the XQuery flag letters to match what the POSIX engine expects.

Keep in mind that the pattern argument of like_regex is a JSON path string literal, written according to the rules given in Section 8.14.7. This means in particular that any backslashes you want to use in the regular expression must be doubled. For example, to match string values of the root document that contain only digits:

```
$.* ? (@ like_regex "^\\d+$")
```

9.17. Sequence Manipulation Functions (NOT SUPPORTED)

9.18. Conditional Expressions

9.18.1. CASE
The SQL CASE expression is a generic conditional expression, similar to if/else statements in other programming languages:

CASE WHEN condition THEN result
[WHEN ...]
[ELSE result]
END
CASE clauses can be used wherever an expression is valid. Each condition is an expression that returns a boolean result. If the condition's result is true, the value of the CASE expression is the result that follows the condition, and the remainder of the CASE expression is not processed. If the condition's result is not true, any subsequent WHEN clauses are examined in the same manner. If no WHEN condition yields true, the value of the CASE expression is the result of the ELSE clause. If the ELSE clause is omitted and no condition is true, the result is null.

An example:

```
SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END as case
    FROM (VALUES (1),(2),(3)) as x(a)

 a | case
```

false → boolean	JSON constant false	`jsonb_path_query_array('[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]', '$[*] ? (@.parent == false)', '{}', false) → [{"name": "John", "parent": false}]`
null → value	JSON constant null (note that, unlike in SQL, comparison to null works normally)	`jsonb_path_query_array('[{"name": "Mary", "job": null}, {"name": "Michael", "job": "driver"}]', '$[*] ? (@.job == null) .name', '{}', false) → ["Mary"]`
boolean && boolean → boolean	Boolean AND	`jsonb_path_query_array('[1, 3, 7]', '$[*] ? (@ > 1 && @ < 5)', '{}', false) → [3]`
boolean || boolean → boolean	Boolean OR	`jsonb_path_query_array('[1, 3, 7]', '$[*] ? (@ < 1 || @ > 5)', '{}', false) → [7]`

onvertible to a single output type. See Section 10.5 for more details.

ariant of the general form above:

ach of the value expressions in the WHEN clauses until one is found that is equal to
or a null value) is returned. This is similar to the switch statement in C.

ASE syntax:

```
  3 | other
```

A CASE expression does not evaluate any subexpressions that are not needed to determine the result. For example, this is a possible way of avoiding a division-by-zero failure:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

Note
As described in Section 4.2.14, there are various situations in which subexpressions of an expression are evaluated at different times, so that the principle that "CASE evaluates only necessary subexpressions" is not ironclad. For example a constant 1/0 subexpression will usually result in a division-by-zero failure at planning time, even if it's within a CASE arm that would never be entered at run time.

9.18.2. COALESCE

COALESCE(value [, ...])
The COALESCE function returns the first of its arguments that is not null. Null is returned only if all arguments are null. It is often used to substitute a default value for null values when data is retrieved for display, for example:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

This returns description if it is not null, otherwise short_description if it is not null, otherwise (none).

The arguments must all be convertible to a common data type, which will be the type of the result (see Section 10.5 for details).

Like a CASE expression, COALESCE only evaluates the arguments that are needed to determine the result; that is, arguments to the right of the first non-null argument are not evaluated. This SQL-standard function provides capabilities similar to NVL and IFNULL, which are used in some other database systems.

9.18.3. NULLIF

NULLIF(value1, value2) (NOT SUPPORTED)
The NULLIF function returns a null value if value1 equals value2; otherwise it returns value1. This can be used to perform the inverse operation of the COALESCE example given above:

```
SELECT NULLIF(value, '(none)') ...
```

In this example, if value is (none), null is returned, otherwise the value of value is returned.

The two arguments must be of comparable types. To be specific, they are compared exactly as if you had written value1 = value2, so there must be a suitable = operator available.

The result has the same type as the first argument — but there is a subtlety. What is actually returned is the first argument of the implied = operator, and in some cases that will have been promoted to match the second argument's type. For example, NULLIF(1, 2.2) yields numeric, because there is no integer = numeric operator, only numeric = numeric.

9.18.4. GREATEST and LEAST

! boolean → boolean	Boolean NOT	`jsonb_path_query_array('[1, 3, 7]', '$[*] ? (!(@ < 5))', '{}', false) → [7]`
boolean is unknown → boolean	Tests whether a Boolean condition is unknown.	`jsonb_path_query_array('[-1, 2, 7, "foo"]', '$[*] ? ((@ > 0) is unknown)', '{}', false) → ["foo"]`
string like_regex string [flag string] → boolean	Tests whether the first operand matches the regular expression given by the second operand, optionally with modifications described by a string of flag characters (see Section 9.16.2.3).	`jsonb_path_query_array('["abc", "abd", "aBdC", "abdacb", "babc"]', '$[*] ? (@ like_regex "^ab.*c")', '{}', false) → ["abc", "abdacb"]` `jsonb_path_query_array('["abc", "abd", "aBdC", "abdacb", "babc"]', '$[*] ? (@ like_regex "^ab.*c" flag "i")', '{}', false) → ["abc", "aBdC", "abdacb"]`

t or smallest value from a list of any number of expressions. The expressions must all
ne type of the result (see Section 10.5 for details). NULL values in the list are ignored.
uate to NULL. (NOT SUPPORTED)

andard, but are a common extension. Some other databases make them return NULL
NULL.

or array types. In addition to those, the usual comparison operators shown in Table
compare the array contents element-by-element, using the default B-tree
ort based on the first difference. In multidimensional arrays the elements are visited in
the contents of two arrays are equal but the dimensionality is different, the first
s the sort order.

Operator	Description	Example(s)
anyarray @> anyarray → boolean	Does the first array contain the second, that is, does each element appearing in the second array equal some element of the first array? (Duplicates are not treated specially, thus ARRAY[1] and ARRAY[1,1] are each considered to contain the other.)	`ARRAY[1,4,3] @> ARRAY[3,1,3] → true`
anyarray <@ anyarray → boolean	Is the first array contained by the second?	`ARRAY[2,2,7] <@ ARRAY[1,7,4,2,6] → true`
anyarray && anyarray → boolean	Do the arrays overlap, that is, have any elements in common?	`ARRAY[1,4,3] && ARRAY[2,1] → true`

anycompatiblearray || anycompatiblearray → anycompatiblearray	Concatenates the two arrays. Concatenating a null or empty array is a no-op; otherwise the arrays must have the same number of dimensions (as illustrated by the first example) or differ in number of dimensions by one (as illustrated by the second). If the arrays are not of identical element types, they will be coerced to a common type (see Section 10.5). (NOT SUPPORTED)	`#ARRAY[1,2,3] || ARRAY[4,5,6,7] → {1,2,3,4,5,6,7}` `#ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9.9]] → {{1,2,3},{4,5,6},{7,8,9.9}}`
anycompatible || anycompatiblearray → anycompatiblearray	Concatenates an element onto the front of an array (which must be empty or one-dimensional). (NOT SUPPORTED)	`#3 || ARRAY[4,5,6] → {3,4,5,6}`
anycompatiblearray || anycompatible → anycompatiblearray	Concatenates an element onto the end of an array (which must be empty or one-dimensional). (NOT SUPPORTED)	`#ARRAY[4,5,6] || 7 → {4,5,6,7}`

out which operators support indexed

tion and examples of the use of these

Function	Description	Example(s)
array_append (anycompatiblearray, anycompatible) → anycompatiblearray	Appends an element to the end of an array (same as the anycompatiblearray || anycompatible operator). (NOT SUPPORTED)	`#array_append(ARRAY[1,2], 3) → {1,2,3}`
array_cat (anycompatiblearray, anycompatiblearray) → anycompatiblearray	Concatenates two arrays (same as the anycompatiblearray || anycompatiblearray operator). (NOT SUPPORTED)	`#array_cat(ARRAY[1,2,3], ARRAY[4,5]) → {1,2,3,4,5}`

array_dims (anyarray) → text	Returns a text representation of the array's dimensions.	`array_dims(ARRAY[[1,2,3], [4,5,6]]) → [1:2][1:3]`
array_fill (anyelement, integer[] [, integer[]]) → anyarray	Returns an array filled with copies of the given value, having dimensions of the lengths specified by the second argument. The optional third argument supplies lower-bound values for each dimension (which default to all 1). (NOT SUPPORTED)	`#array_fill(11, ARRAY[2,3]) → {{11,11,11}, {11,11,11}}` `#array_fill(7, ARRAY[3], ARRAY[2]) → [2:4]={7,7,7}`

array_length (anyarray, integer) → integer	Returns the length of the requested array dimension. (Produces NULL instead of 0 for empty or missing array dimensions.)	`array_length(array[1,2,3], 1) → 3`
		`#array_length(array[]::int[], 1) → NULL`
		`array_length(array['text'], 2) → NULL`

array_lower (anyarray, integer) → integer	Returns the lower bound of the requested array dimension.	`array_lower('[0:2]={1,2,3}'::integer[], 1) → 0`

array_ndims (anyarray) → integer	Returns the number of dimensions of the array.	`array_ndims(ARRAY[[1,2,3], [4,5,6]]) → 2`
array_position (anycompatiblearray, anycompatible [, integer]) → integer	Returns the subscript of the first occurrence of the second argument in the array, or NULL if it's not present. If the third argument is given, the search begins at that subscript. The array must be one-dimensional. Comparisons are done using IS NOT DISTINCT FROM semantics, so it is possible to search for NULL. (NOT SUPPORTED)	`#array_position(ARRAY['sun', 'mon', 'tue', 'wed', 'thu', 'fri', 'sat'], 'mon') → 2`
array_positions (anycompatiblearray, anycompatible) → integer[]	Returns an array of the subscripts of all occurrences of the second argument in the array given as first argument. The array must be one-dimensional. Comparisons are done using IS NOT DISTINCT FROM semantics, so it is possible to search for NULL. NULL is returned only if the array is NULL; if the value is not found in the array, an empty array is returned. (NOT SUPPORTED)	`#array_positions(ARRAY['A','A','B','A'], 'A') → {1,2,4}`
array_prepend (anycompatible, anycompatiblearray) → anycompatiblearray	Prepends an element to the beginning of an array (same as the anycompatible || anycompatiblearray operator). (NOT SUPPORTED)	`#array_prepend(1, ARRAY[2,3]) → {1,2,3}`
array_remove (anycompatiblearray, anycompatible) → anycompatiblearray	Removes all elements equal to the given value from the array. The array must be one-dimensional. Comparisons are done using IS NOT DISTINCT FROM semantics, so it is possible to remove NULLs. (NOT SUPPORTED)	`#array_remove(ARRAY[1,2,3,2], 2) → {1,3}`

Note
There are two differences in the behavior of string_to_array from pre-9.1 versions of PostgreSQL. First, it will return an empty (zero-element) array rather than NULL when the input string is of zero length. Second, if the delimiter string is NULL, the function splits the input into individual characters, rather than returning NULL as before.

array_replace (anycompatiblearray, anycompatible, anycompatible) → anycompatiblearray	Replaces each array element equal to the second argument with the third argument. (NOT SUPPORTED)	`#array_replace(ARRAY[1,2,5,4], 5, 3) → {1,2,3,4}`
array_to_string (array anyarray, delimiter text [, null_string text]) → text	Converts each array element to its text representation, and concatenates those separated by the delimiter string. If null_string is given and is not NULL, then NULL array entries are represented by that string; otherwise, they are omitted.	`array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*') → 1,2,3,*,5`

Function	Description	Partial Mode	Example
array_agg (anynonarray) → anyarray	Collects all the input values, including nulls, into an array.	No	`SELECT array_agg(x) FROM (VALUES (1),(2)) a(x) → {1,2}`
array_agg (anyarray) → anyarray	Concatenates all the input arrays into an array of one higher dimension. (The inputs must all have the same dimensionality, and cannot be empty or null.)	No	`SELECT array_agg(x) FROM (VALUES (ARRAY[1,2]),(ARRAY[3,4])) a(x) → {{1,2},{3,4}}`

`array_upper(ARRAY[1,8,3,7], 1) → 4`

`cardinality(ARRAY[[1,2],[3,4]]) → 4`

trim_array (array anyarray, n integer) → anyarray	Trims an array by removing the last n elements. If the array is multidimensional, only the first dimension is trimmed.	`trim_array(ARRAY[1,2,3,4,5,6], 2) → {1,2,3,4}`

avg (smallint) → numeric
avg (integer) → numeric
avg (bigint) → numeric
avg (numeric) → numeric
avg (real) → double precision
avg (double precision) → double precision
avg (interval) → interval
	Computes the average (arithmetic mean) of all the non-null input values.	Yes	`SELECT avg(x::smallint) FROM (VALUES (1),(2),(3)) a(x) → 2.0000000000000000`
			`SELECT avg(x::integer) FROM (VALUES (1),(2),(3)) a(x) → 2.0000000000000000`
			`SELECT avg(x::bigint) FROM (VALUES (1),(2),(3)) a(x) → 2.0000000000000000`
			`SELECT avg(x::numeric) FROM (VALUES (1),(2),(3)) a(x) → 2.0000000000000000`
			`SELECT avg(x::real) FROM (VALUES (1),(2),(3)) a(x) → 2`
			`SELECT avg(x::double precision) FROM (VALUES (1),(2),(3)) a(x) → 2`
			`SELECT avg(cast(x as interval day)) FROM (VALUES ('1'),('2'),('3')) a(x) → 2 days`

bit_and (smallint) → smallint
bit_and (integer) → integer
bit_and (bigint) → bigint
bit_and (bit) → bit
	Computes the bitwise AND of all non-null input values.	Yes	`SELECT bit_and(x::smallint) FROM (VALUES (5),(6),(7)) a(x) → 4`
			`SELECT bit_and(x::integer) FROM (VALUES (5),(6),(7)) a(x) → 4`
			`SELECT bit_and(x::bigint) FROM (VALUES (5),(6),(7)) a(x) → 4`
			`SELECT bit_and(x::bit(3)) FROM (VALUES ('101'),('110'),('111')) a(x) → 100`

bit_or (smallint) → smallint
bit_or (integer) → integer
bit_or (bigint) → bigint
bit_or (bit) → bit
	Computes the bitwise OR of all non-null input values.	Yes	`SELECT bit_or(x::smallint) FROM (VALUES (4),(5),(6)) a(x) → 7`
			`SELECT bit_or(x::integer) FROM (VALUES (4),(5),(6)) a(x) → 7`
			`SELECT bit_or(x::bigint) FROM (VALUES (4),(5),(6)) a(x) → 7`
			`SELECT bit_or(x::bit(3)) FROM (VALUES ('100'),('101'),('110')) a(x) → 111`

```
SELECT * FROM unnest(ARRAY[1,2]) as a → [
1
2
]

SELECT * FROM unnest(ARRAY[['foo','bar'],
['baz','quux']]) as a → [
foo
bar
baz
quux
]
```

```
#SELECT * FROM unnest(ARRAY[1,2],
ARRAY['foo','bar','baz']) as x(a,b) → [
1,foo
2,bar
,baz
]
```

It should be noted that except for count, these functions return a null value when no rows are selected. In particular, sum of no rows returns null, not zero as one might expect, and array_agg returns null rather than an empty array when there are no input rows. The coalesce function can be used to substitute zero or an empty array for null when necessary.

bit_xor (smallint) → smallint
 	
bit_xor (integer) → integer
 	
bit_xor (bigint) → bigint
 	
bit_xor (bit) → bit
	Computes the bitwise exclusive OR of all non-null input values. Can be useful as a checksum for an unordered set of values.	Yes	`SELECT bit_xor(x::smallint) FROM (VALUES (5),(6),(6)) a(x) → 5`
			`SELECT bit_xor(x::integer) FROM (VALUES (5),(6),(6)) a(x) → 5`
			`SELECT bit_xor(x::bigint) FROM (VALUES (5),(6),(6)) a(x) → 5`
			`SELECT bit_xor(x::bit(3)) FROM (VALUES ('101'),('110'),('119')) a(x) → 101`

bool_and (boolean) → boolean	Returns true if all non-null input values are true, otherwise false.	Yes	`SELECT bool_and(x) FROM (VALUES (null),(false),(true)) a(x) → false`
			`SELECT bool_and(x) FROM (VALUES (null),(true),(true)) a(x) → true`
			`SELECT bool_and(x) FROM (VALUES (null::bool),(null::bool),(null::bool)) a(x) → NULL`

bool_or (boolean) → boolean	Returns true if any non-null input value is true, otherwise false.	Yes	`SELECT bool_or(x) FROM (VALUES (null),(false),(false)) a(x) → false`
			`SELECT bool_or(x) FROM (VALUES (null),(false),(true)) a(x) → true`
			`SELECT bool_or(x) FROM (VALUES (null::bool),(null::bool),(null::bool)) a(x) → NULL`

count (*) → bigint	Computes the number of input rows.	Yes	`SELECT count(*) FROM (VALUES (4),(5),(6)) a(x) → 3`

)_agg, json_object_agg, jsonb_object_agg, string_agg, and xmlagg, as well as similar gfully different result values depending on the order of the input values. This ordering is iting an ORDER BY clause within the aggregate call, as shown in Section 4.2.7. ed subquery will usually work. For example:

```
ORDER BY y DESC) AS tab;
```

y level contains additional processing, such as a join, because that might cause the egate is computed.

espond to the standard SQL aggregates every and any or some. PostgreSQL supports biguity built into the standard syntax:

```
FROM t1 ...;
```

a subquery, or as being an aggregate function, if the subquery returns one row with a given to these aggregates.

base management systems might be disappointed by the performance of the count uery like:

e: PostgreSQL will need to scan either the entire table or the entirety of an index that

Table 9.58 shows aggregate functions typically used in statistical analysis. (These are separated out merely to avoid cluttering the listing of more-commonly-used aggregates.) Functions shown as accepting numeric_type are available for all the types smallint, integer, bigint, numeric, real, and double precision. Where the description mentions N, it means the number of input rows for which all the input expressions are non-null. In all cases, null is returned if the computation is meaningless, for example when N is zero.

Table 9.58. Aggregate Functions for Statistics

Function	Description	Partial Mode	Examples
corr (Y double precision, X double precision) → double precision	Computes the correlation coefficient.	Yes	`SELECT corr(x,y) FROM (VALUES (1,2),(2,3),(3,1)) a(x,y) → -0.5`
covar_pop (Y double precision, X double precision) → double precision	Computes the population covariance.	Yes	`SELECT covar_pop(x,y) FROM (VALUES (1,2),(2,3),(3,1)) a(x,y) → -0.3333333333333333`

covar_samp (Y double precision, X double precision) → double precision	Computes the sample covariance.	Yes	`SELECT covar_samp(x,y) FROM (VALUES (1,2),(2,3),(3,1)) a(x,y) → -0.5`
regr_avgx (Y double precision, X double precision) → double precision	Computes the average of the independent variable, sum(X)/N.	Yes	`SELECT regr_avgx(x,y) FROM (VALUES (1,2),(2,3),(3,1)) a(x,y) → 2`
regr_avgy (Y double precision, X double precision) → double precision	Computes the average of the dependent variable, sum(Y)/N.	Yes	`SELECT regr_avgy(x,y) FROM (VALUES (1,2),(2,3),(3,1)) a(x,y) → 2`
regr_count (Y double precision, X double precision) → bigint	Computes the number of rows in which both inputs are non-null.	Yes	`SELECT regr_count(x,y) FROM (VALUES (1,2),(2,3),(3,1)) a(x,y) → 3`

regr_intercept (Y double precision, X double precision) → double precision	Computes the y-intercept of the least-squares-fit linear equation determined by the (X, Y) pairs.	Yes	`SELECT regr_intercept(x,y) FROM (VALUES (1,2),(2,3),(3,1)) a(x,y) → 3`
regr_r2 (Y double precision, X double precision) → double precision	Computes the square of the correlation coefficient.	Yes	`SELECT regr_r2(x,y) FROM (VALUES (1,2),(2,3),(3,1)) a(x,y) → 0.25`
regr_slope (Y double precision, X double precision) → double precision	Computes the slope of the least-squares-fit linear equation determined by the (X, Y) pairs.	Yes	`SELECT regr_slope(x,y) FROM (VALUES (1,2),(2,3),(3,1)) a(x,y) → -0.5`
regr_sxx (Y double precision, X double precision) → double precision	Computes the “sum of squares” of the independent variable, sum(X^2) - sum(X)^2/N.	Yes	`SELECT regr_sxx(x,y) FROM (VALUES (1,2),(2,3),(3,1)) a(x,y) → 2`

These functions are sometimes referred to as they may also take a direct argument that is not aggregated, but is computed only once. All these functions ignore null values in their aggregated input. For those that take a fraction parameter, the fraction value must be between 0 and 1; an error is thrown if not. However, a null fraction value simply produces a null result.

Table 9.59. Ordered-Set Aggregate Functions (NOT SUPPORTED)

Function	Description	Partial Mode
mode () WITHIN GROUP (ORDER BY anyelement) → anyelement	Computes the mode, the most frequent value of the aggregated argument (arbitrarily choosing the first one if there are multiple equally-frequent values). The aggregated argument must be of a sortable type.	No
percentile_cont (fraction double precision) WITHIN GROUP (ORDER BY double precision) → double precision percentile_cont (fraction double precision) WITHIN GROUP (ORDER BY interval) → interval	Computes the continuous percentile, a value corresponding to the specified fraction within the ordered set of aggregated argument values. This will interpolate between adjacent input items if needed.	No

percentile_cont (fractions double precision[]) WITHIN GROUP (ORDER BY double precision) → double precision[] percentile_cont (fractions double precision[]) WITHIN GROUP (ORDER BY interval) → interval[]	Computes multiple continuous percentiles. The result is an array of the same dimensions as the fractions parameter, with each non-null element replaced by the (possibly interpolated) value corresponding to that percentile.	No
percentile_disc (fraction double precision) WITHIN GROUP (ORDER BY anyelement) → anyelement	Computes the discrete percentile, the first value within the ordered set of aggregated argument values whose position in the ordering equals or exceeds the specified fraction. The aggregated argument must be of a sortable type.	No
percentile_disc (fractions double precision[]) WITHIN GROUP (ORDER BY anyelement) → anyarray	Computes multiple discrete percentiles. The result is an array of the same dimensions as the fractions parameter, with each non-null element replaced by the input value corresponding to that percentile. The aggregated argument must be of a sortable type.	No

stddev_pop (numeric_type) → double precision for real or double precision, otherwise numeric	Computes the population standard deviation of the input values.	Yes	`SELECT stddev_pop(x) FROM (VALUES (1),(2),(3)) a(x) → 0.8164965809277260327`

w function of the same name defined in Section
on would have returned for the “hypothetical” row
ented by the sorted_args. For each of these
the aggregated arguments given in sorted_args.
nput rows containing nulls. Null values sort

Function	Description	Partial Mode
rank (args) WITHIN GROUP (ORDER BY sorted_args) → bigint	Computes the rank of the hypothetical row, with gaps; that is, the row number of the first row in its peer group.	No
dense_rank (args) WITHIN GROUP (ORDER BY sorted_args) → bigint	Computes the rank of the hypothetical row, without gaps; this function effectively counts peer groups.	No
percent_rank (args) WITHIN GROUP (ORDER BY sorted_args) → double precision	Computes the relative rank of the hypothetical row, that is (rank - 1) / (total rows - 1). The value thus ranges from 0 to 1 inclusive.	No
cume_dist (args) WITHIN GROUP (ORDER BY sorted_args) → double precision	Computes the cumulative distribution, that is (number of rows preceding or peers with hypothetical row) / (total rows). The value thus ranges from 1/N to 1.	No

Table 9.61. Grouping Operations

Function	Description
GROUPING (group_by_expression(s)) → integer	Returns a bit mask indicating which GROUP BY expressions are not included in the current grouping set. Bits are assigned with the rightmost argument corresponding to the least-significant bit; each bit is 0 if the corresponding expression is included in the grouping criteria of the grouping set generating the current result row, and 1 if it is not included. (NOT SUPPORTED)
variance (numeric_type) → double precision for real or double precision, otherwise numeric	This is a historical alias for var_samp.	Yes	`SELECT variance(x) FROM (VALUES (1),(2),(3)) a(x) → 1.0000000000000000000`
var_pop (numeric_type) → double precision for real or double precision, otherwise numeric	Computes the population variance of the input values (square of the population standard deviation).	Yes	`SELECT var_pop(x) FROM (VALUES (1),(2),(3)) a(x) → 0.6666666666666666667`
var_samp (numeric_type) → double precision for real or double precision, otherwise numeric	Computes the sample variance of the input values (square of the sample standard deviation).	Yes	`SELECT var_samp(x) FROM (VALUES (1),(2),(3)) a(x) → 1.0000000000000000000`

(see Section 7.2.4) to distinguish result rows. The
natch expressions given in the GROUP BY clause

```
ROUP BY ROLLUP(make,model);
```

Here, the grouping value 0 in the first four rows shows that those have been grouped normally, over both the grouping columns. The value 1 indicates that model was not grouped by in the next-to-last two rows, and the value 3 indicates that neither make nor model was grouped by in the last row (which therefore is an aggregate over all the input rows).

9.22. Window Functions

Window functions provide the ability to perform calculations across sets of rows that are related to the current query row. See Section 3.5 for an introduction to this feature, and Section 4.2.8 for syntax details.

The built-in window functions are listed in Table 9.62. Note that these functions must be invoked using window function syntax, i.e., an OVER clause is required.

In addition to these functions, any built-in or user-defined ordinary aggregate (i.e., not ordered-set or hypothetical-set aggregates) can be used as a window function; see Section 9.21 for a list of the built-in aggregates. Aggregate functions act as window functions only when an OVER clause follows the call; otherwise they act as plain aggregates and return a single row for the entire set.

Table 9.62. General-Purpose Window Functions

Function	Description	Examples

row_number () → bigint	Returns the number of the current row within its partition, counting from 1.	`SELECT row_number() OVER (ORDER BY x) FROM (VALUES (4),(5),(6)) a(x) → [ 1 2 3 ]`
rank () → bigint	Returns the rank of the current row, with gaps; that is, the row_number of the first row in its peer group.	`SELECT rank() OVER (ORDER BY x) FROM (VALUES (4),(5),(5),(6)) a(x) → [ 1 2 2 4 ]`
dense_rank () → bigint	Returns the rank of the current row, without gaps; this function effectively counts peer groups.	`SELECT dense_rank() OVER (ORDER BY x) FROM (VALUES (4),(5),(5),(6)) a(x) → [ 1 2 2 3 ]`

percent_rank () → double precision	Returns the relative rank of the current row, that is (rank - 1) / (total partition rows - 1). The value thus ranges from 0 to 1 inclusive.	`SELECT percent_rank() OVER (ORDER BY x) FROM (VALUES (4),(5),(5),(6)) a(x) → [ 0 0.3333333333333333 0.3333333333333333 1 ]`
cume_dist () → double precision	Returns the cumulative distribution, that is (number of partition rows preceding or peers with current row) / (total partition rows). The value thus ranges from 1/N to 1.	`SELECT cume_dist() OVER (ORDER BY x) FROM (VALUES (4),(5),(5),(6)) a(x) → [ 0.25 0.75 0.75 1 ]`
ntile (num_buckets integer) → integer	Returns an integer ranging from 1 to the argument value, dividing the partition as equally as possible.	`SELECT ntile(2) OVER (ORDER BY x) FROM (VALUES (4),(5),(5),(6)) a(x) → [ 1 1 2 2 ]`

lag (value anycompatible [, offset integer [, default anycompatible]]) → anycompatible	Returns value evaluated at the row that is offset rows before the current row within the partition; if there is no such row, instead returns default (which must be of a type compatible with value). Both offset and default are evaluated with respect to the current row. If omitted, offset defaults to 1 and default to NULL.	`SELECT lag(x) OVER (ORDER BY x) FROM (VALUES (4),(5),(5),(6)) a(x) → [ NULL 4 5 5 ]`
lead (value anycompatible [, offset integer [, default anycompatible]]) → anycompatible	Returns value evaluated at the row that is offset rows after the current row within the partition; if there is no such row, instead returns default (which must be of a type compatible with value). Both offset and default are evaluated with respect to the current row. If omitted, offset defaults to 1 and default to NULL.	`SELECT lead(x) OVER (ORDER BY x) FROM (VALUES (4),(5),(5),(6)) a(x) → [ 5 5 6 NULL ]`
first_value (value anyelement) → anyelement	Returns value evaluated at the row that is the first row of the window frame.	`SELECT first_value(x) OVER (ORDER BY x) FROM (VALUES (4),(5),(5),(6)) a(x) → [ 4 4 4 4 ]`

se of the associated window definition.
our ranking functions (including

nich by default contains the rows from
lts for last_value and sometimes also
ROUPS) to the OVER clause. See

urrent row's window frame. An
type of behavior, which may or may not
TWEEN UNBOUNDED PRECEDING AND

t_value, and nth_value. This is not
PECT NULLS. Likewise, the standard's
behavior is supported. (You can achieve

expression forms documented in this

to determine whether it returns any
e result of EXISTS is “false”.

any one evaluation of the subquery.

returned, not all the way to completion.
the side effects occur might be

ows, the output list of the subquery is
SELECT 1 WHERE ...). There are

o1 row, even if there are several

```
),(5)) b(y) WHERE x=y) → [
```

expression IN (subquery)

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of IN is “true” if any equal subquery row is found. The result is “false” if no equal row is found (including the case where the subquery returns no rows).

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the IN construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

row_constructor IN (subquery) (NOT SUPPORTED)

The left-hand side of this form of IN is a row constructor, as described in Section 4.2.13. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of IN is “true” if any equal subquery row is found. The result is “false” if no equal row is found (including the case where the subquery returns no rows).

As usual, null values in the rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (null). If all the per-row results are either unequal or null, with at least one null, then the result of IN is null.

Example

last_value (value anyelement) → anyelement	Returns value evaluated at the row that is the last row of the window frame.	`SELECT last_value(x) OVER (ORDER BY x) FROM (VALUES (4),(5),(5),(6)) a(x) → [ 4 5 5 6 ]`
nth_value (value anyelement, n integer) → anyelement	Returns value evaluated at the row that is the n'th row of the window frame (counting from 1); returns NULL if there is no such row.	`SELECT nth_value(x,2) OVER (ORDER BY x) FROM (VALUES (4),(5),(5),(6)) a(x) → [ NULL 5 5 5 ]`

```
SELECT x FROM (VALUES (1),(2),(3)) a(x) WHERE x NOT IN (SELECT y FROM (VALUES (3),(4),(5)) b(y)) ORDER BY x → [
1
2
]
```

9.23.4. ANY/SOME

expression operator ANY (subquery)
expression operator SOME (subquery)

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of ANY is “true” if any true result is obtained. The result is “false” if no true result is found (including the case where the subquery returns no rows).

SOME is a synonym for ANY. IN is equivalent to = ANY.

Note that if there are no successes and at least one right-hand row yields null for the operator's result, the result of the ANY construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

row_constructor operator ANY (subquery) (NOT SUPPORTED)
row_constructor operator SOME (subquery) (NOT SUPPORTED)

The left-hand side of this form of ANY is a row constructor, as described in Section 4.2.13. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given operator. The result of ANY is “true” if the comparison returns true for any subquery row. The result is “false” if the comparison returns false for every subquery row (including the case where the subquery returns no rows). The result is NULL if no comparison with a subquery row returns true, and at least one comparison returns NULL.

See Section 9.24.5 for details about the meaning of a row constructor comparison.

Example

```
SELECT x FROM (VALUES (1),(2),(3)) a(x) WHERE x = ANY (SELECT y FROM (VALUES (3),(4),(5)) b(y)) → [
3
]
```

9.23.5. ALL

expression operator ALL (subquery)

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of ALL is “true” if all rows yield true (including the case where the subquery returns no rows). The result is “false” if any false result is found. The result is NULL if no comparison with a subquery row returns false, and at least one comparison returns NULL.

NOT IN is equivalent to <> ALL.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

row_constructor operator ALL (subquery) (NOT SUPPORTED)

The left-hand side of this form of ALL is a row constructor, as described in Section 4.2.13. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given operator. The result of ALL is “true” if the comparison returns true for all subquery rows (including the case where the subquery returns no rows). The result is “false” if the comparison returns false for any subquery row. The result is NULL if no comparison with a subquery row returns false, and at least one comparison returns NULL.

See Section 9.24.5 for details about the meaning of a row constructor comparison.

Example

```
SELECT x FROM (VALUES (1),(2),(3)) a(x) WHERE x <> ALL (SELECT y FROM (VALUES (3),(4),(5)) b(y)) ORDER BY x → [
1
2
]
```

9.23.6. Single-Row Comparison

row_constructor operator (subquery) (NOT SUPPORTED)

The left-hand side is a row constructor, as described in Section 4.2.13. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. Furthermore, the subquery cannot return more than one row. (If it returns zero rows, the result is taken to be null.) The left-hand side is evaluated and compared row-wise to the single subquery result row.

See Section 9.24.5 for details about the meaning of a row constructor comparison.

9.24. Row and Array Comparisons

This section describes several specialized constructs for making multiple comparisons between groups of values. These forms are syntactically related to the subquery forms of the previous section, but do not involve subqueries. The forms involving array subexpressions are PostgreSQL extensions; the rest are SQL-compliant. All of the expression forms documented in this section return Boolean (true/false) results.

9.24.1. IN

expression IN (value [, ...])
The right-hand side is a parenthesized list of expressions. The result is “true” if the left-hand expression's result is equal to any of the right-hand expressions. This is a shorthand notation for

expression = value1
OR
expression = value2
OR
...
Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand expression yields null, the result of the IN construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

Example

```
SELECT x FROM (VALUES (1),(2),(3)) a(x) WHERE x IN (1,2) ORDER BY x → [
1
2
]

SELECT x IN (y, z) FROM (VALUES (1,1,2),(2,3,null),(3,4,5),(4,null,null)) a(x,y,z) ORDER BY x → [
true
NULL
false
NULL
]
```

9.24.2. NOT IN

expression NOT IN (value [, ...])
The right-hand side is a parenthesized list of expressions. The result is “true” if the left-hand expression's result is unequal to all of the right-hand expressions. This is a shorthand notation for

expression <> value1
AND
expression <> value2
AND
...
Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand expression yields null,

the result of the NOT IN construct will be null, not true as one might naively expect. This is in accordance with SQL's normal rules for Boolean combinations of null values.

Tip
x NOT IN y is equivalent to NOT (x IN y) in all cases. However, null values are much more likely to trip up the novice when working with NOT IN than when working with IN. It is best to express your condition positively if possible.

Example

```
SELECT x FROM (VALUES (1),(2),(3)) a(x) WHERE x NOT IN (1,2) ORDER BY x → [
3
]

SELECT x NOT IN (y, z) FROM (VALUES (1,1,2),(2,3,null),(3,4,5),(4,null,null)) a(x,y,z) ORDER BY x → [
false
NULL
true
NULL
]
```

9.24.3. ANY/SOME (array)

expression operator ANY (array expression)
expression operator SOME (array expression)

The right-hand side is a parenthesized expression, which must yield an array value. The left-hand expression is evaluated and compared to each element of the array using the given operator, which must yield a Boolean result. The result of ANY is “true” if any true result is obtained. The result is “false” if no true result is found (including the case where the array has zero elements).

If the array expression yields a null array, the result of ANY will be null. If the left-hand expression yields null, the result of ANY is ordinarily null (though a non-strict comparison operator could possibly yield a different result). Also, if the right-hand array contains any null elements and no true comparison result is obtained, the result of ANY will be null, not false (again, assuming a strict comparison operator). This is in accordance with SQL's normal rules for Boolean combinations of null values.

SOME is a synonym for ANY.

Example

```
SELECT x FROM (VALUES (1),(2),(3)) a(x) WHERE x = ANY (array[1,2]) ORDER BY x → [
1
2
]
]
```

9.24.4. ALL (array)

expression operator ALL (array expression)

The right-hand side is a parenthesized expression, which must yield an array value. The left-hand expression is evaluated and compared to each element of the array using the given operator, which must yield a Boolean result. The result of ALL is “true” if all comparisons yield true (including the case where the array has zero elements). The result is “false” if any false result is found.

If the array expression yields a null array, the result of ALL will be null. If the left-hand expression yields null, the result of ALL is ordinarily null (though a non-strict comparison operator could possibly yield a different result). Also, if the right-hand array contains any null elements and no false comparison result is obtained, the result of ALL will be null, not true (again, assuming a strict comparison operator). This is in accordance with SQL's normal rules for Boolean combinations of null values.

Example

```
SELECT x FROM (VALUES (1),(2),(3)) a(x) WHERE x <> ALL (array[1,2]) ORDER BY x → [
3
]
]
```

9.24.5. Row Constructor Comparison (NOT SUPPORTED)

row_constructor operator row_constructor

Each side is a row constructor, as described in Section 4.2.13. The two row constructors must have the same number of fields. The given operator is applied to each pair of corresponding fields. (Since the fields could be of different types, this means that a different specific operator could be selected for each pair.) All the selected operators must be members of some B-tree operator class, or be the negator of an = member of a B-tree operator class, meaning that row constructor comparison is only possible when the operator is =, <>, <, <=, >, or >=, or has semantics similar to one of these.

The = and <> cases work slightly differently from the others. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of the row comparison is unknown (null).

For the <, <=, > and >= cases, the row elements are compared left-to-right, stopping as soon as an unequal or null pair of elements is found. If either of this pair of elements is null, the result of the row comparison is unknown (null); otherwise comparison of this pair of elements determines the result. For example, ROW(1,2,NULL) < ROW(1,3,0) yields true, not null, because the third pair of elements are not considered.

Note
Prior to PostgreSQL 8.2, the <, <=, > and >= cases were not handled per SQL specification. A comparison like ROW(a,b) < ROW(c,d) was implemented as a < c AND b < d whereas the correct behavior is equivalent to a < c OR (a = c AND b < d).

row_constructor IS DISTINCT FROM row_constructor

This construct is similar to a <> row comparison, but it does not yield null for null inputs. Instead, any null value is considered unequal to (distinct from) any non-null value, and any two nulls are considered equal (not distinct). Thus the result will either be true or false, never null.

row_constructor IS NOT DISTINCT FROM row_constructor

This construct is similar to a = row comparison, but it does not yield null for null inputs. Instead, any null value is considered unequal to (distinct from) any non-null value, and any two nulls are considered equal (not distinct). Thus the result will always be either true or false, never null.

9.24.6. Composite Type Comparison (NOT SUPPORTED)

record operator record

The SQL specification requires row-wise comparison to return NULL if the result depends on comparing two NULL values or a NULL and a non-NULL. PostgreSQL does this only when comparing the results of two row constructors (as in Section 9.24.5) or comparing a row constructor to the output of a subquery (as in Section 9.23). In other contexts where two composite-type values are compared, two NULL field values are considered equal, and a NULL is considered larger than a non-NULL. This is necessary in order to have consistent sorting and indexing behavior for composite types.

Each side is evaluated and they are compared row-wise. Composite type comparisons are allowed when the operator is =, <>, <, <=, > or >=, or has semantics similar to one of these. (To be specific, an operator can be a row comparison operator if it is a member of a B-tree operator class, or is the negator of the = member of a B-tree operator class.) The default behavior of the above operators is the same as for IS [NOT] DISTINCT FROM for row constructors (see Section 9.24.5).

To support matching of rows which include elements without a default B-tree operator class, the following operators are defined for composite type comparison: `*=`, `*<>`, `*<`, `*<=`, `*>`, and `*>=`. These operators compare the internal binary representation of the two rows. Two rows might have a different binary representation even though comparisons of the two rows with the equality operator is true. The ordering of rows under these comparison operators is deterministic but not otherwise meaningful. These operators are used internally for materialized views and might be useful for other specialized purposes such as replication and B-Tree deduplication (see Section 64.4.3). They are not intended to be generally useful for writing queries, though.

9.25. Set Returning Functions

This section describes functions that possibly return more than one row. The most widely used functions in this class are series generating functions, as detailed in Table 9.63 and Table 9.64. Other, more specialized set-returning functions are described elsewhere in this manual. See Section 7.2.1.4 for ways to combine multiple set-returning functions.

Table 9.63. Series Generating Functions

Function	Description
generate_series (start integer, stop integer [, step integer]) → setof integer generate_series (start bigint, stop bigint [, step bigint]) → setof bigint generate_series (start numeric, stop numeric [, step numeric]) → setof numeric	Generates a series of values from start to stop, with a step size of step. step defaults to 1.
generate_series (start timestamp, stop timestamp, step interval) → setof timestamp generate_series (start timestamp with time zone, stop timestamp with time zone, step interval) → setof timestamp with time zone	Generates a series of values from start to stop, with a step size of step.

When step is positive, zero rows are returned if start is greater than stop. Conversely, when step is negative, zero rows are returned if start is less than stop. Zero rows are also returned if any input is NULL. It is an error for step to be zero. Some examples follow:

```
SELECT * FROM generate_series(2,4) a → [
2
3
4
]

SELECT * FROM generate_series(5,1,-2) a → [
5
3
1
]

SELECT * FROM generate_series(4,3) a → [
]

SELECT * FROM generate_series(1.1, 4, 1.3) a → [
1.1
2.4
3.7
]

-- this example relies on the date-plus-integer operator:
SELECT date '2004-02-05' + s.a AS dates FROM generate_series(0,14,7) AS s(a) → [
2004-02-05
2004-02-12
2004-02-19
]

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp, '2008-03-04 12:00', '10 hours') a → [
2008-03-01 00:00:00
2008-03-01 10:00:00
2008-03-01 20:00:00
2008-03-02 06:00:00
2008-03-02 16:00:00
2008-03-03 02:00:00
2008-03-03 12:00:00
2008-03-03 22:00:00
2008-03-04 08:00:00
]
```

Table 9.64. Subscript Generating Functions (NOT SUPPORTED)

Function	Description
generate_subscripts (array anyarray, dim integer) → setof integer	Generates a series comprising the valid subscripts of the dim'th dimension of the given array.
generate_subscripts (array anyarray, dim integer, reverse boolean) → setof integer	Generates a series comprising the valid subscripts of the dim'th dimension of the given array. When reverse is true, returns the series in reverse order.

```
-- basic usage:
#SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s → [
1
2
3
4
]

SELECT a AS array, s AS subscript, a[s] AS value FROM (SELECT generate_subscripts(a, 1) AS s, a FROM (VALUES
(array[-1,-2]),(array[100,200,300])) s(a)) foo;
```

9.26. System Information Functions and Operators (NOT SUPPORTED)

9.27. System Administration Functions (NOT SUPPORTED)

9.28. Trigger Functions (NOT SUPPORTED)

9.29. Event Trigger Functions (NOT SUPPORTED)

9.30. Statistics Information Functions (NOT SUPPORTED)

Dump data form PostgreSQL

> **Warning**
>
> At the moment, YDB's compatibility with PostgreSQL **is under development**, so not all PostgreSQL constructs and functions are supported yet. PostgreSQL compatibility is available for testing in the form of a Docker container, which can be deployed by following these instructions.

Data from PostgreSQL can be migrated to YDB using utilities such as pg_dump, psql, and YDB CLI. The pg_dump and psql utilities are installed with PostgreSQL. YDB CLI is YDB's command-line client, which is installed separately.

To do this, you need to:

1. Create a data dump using pg_dump with the following parameters:
   - `--inserts` — to add data using the INSERT command, instead of the COPY protocol.
   - `--column-inserts` — to add data using the INSERT command with column names.
   - `--rows-per-insert=1000` — to insert data in batches to speed up the process.
   - `--encoding=utf_8` — YDB only supports string data in UTF-8.
2. Convert the dump to a format supported by YDB using the `ydb tools pg-convert` command from YDB CLI.
3. Load the result into YDB in PostgreSQL compatibility mode.

pg-convert command

The `ydb tools pg-convert` command reads a dump file or standard input created by the pg_dump utility, performs transformations, and outputs to standard output a dump that can be sent to YDB's PostgreSQL-compatible middleware.

`ydb tools pg-convert` performs the following transformations:

- Moving the creation of the primary key into the body of the CREATE TABLE command.
- Removing the `public` schema from table names.
- Deleting the `WITH (...)` section in `CREATE TABLE`.
- Commenting out unsupported constructs (optionally):
  - `SELECT pg_catalog.set_config.*`
  - `ALTER TABLE`

If the CLI cannot find a table's primary key, it will automatically create a BIGSERIAL column named `__ydb_stub_id` as the primary key.

The general form of the command:

```
ydb [global options...] tools pg-convert [options...]
```

- `global options` — global parameters.
- `options` — subcommand parameters.

subcommand parameters

Name	Description
`-i`	The name of the file containing the original dump. If the option is not specified, the dump is read from standard input.
`--ignore-unsupported`	When this option is specified, unsupported constructs will be commented out in the resulting dump and duplicated in standard error. By default, if unsupported constructs are detected, the command returns an error. This does not apply to `ALTER TABLE` expressions that define a table's primary key, as they are commented out in any case.

> **Warning**
>
> When loading large dumps, reading from standard input is not recommended because the entire dump will be stored in RAM. It is advised to use the file option, in which case the CLI will only keep a small portion of the dump in memory.

Example of importing a dump into YDB

As an example, data generated by pgbench will be loaded.

1. Start Docker containers with PostgreSQL and YDB:

```
docker run --name postgres_container \
    -e POSTGRES_USER=pgroot -e POSTGRES_PASSWORD=1234 \
    -e POSTGRES_DB=local \
    -p 5433:5433 -d postgres:14 -c 'port=5433'
docker run --name ydb-postgres -d --pull always -p 5432:5432 -p 8765:8765 \
    -e POSTGRES_USER=ydbroot -e POSTGRES_PASSWORD=4321 \
    -e YDB_FEATURE_FLAGS=enable_temp_tables,enable_table_pg_types \
    -e YDB_USE_IN_MEMORY_PDISKS=true \
    ghcr.io/ydb-platform/local-ydb:latest
```

Information about the started Docker containers:

Database	PostgreSQL	YDB
Container name	postgres_container	ydb-postgres
Address	postgres://pgroot:1234@localhost:5433/local	postgresql://ydbroot:4321@localhost:5432/local
Port	5433	5432
User name	pgroot	ydbroot
Password	1234	4321

2. Generate data through pgbench:

```
docker exec postgres_container pgbench postgres://pgroot:1234@localhost:5433/local -i
```

3. Create a dump of the database using pg_dump:

```
docker exec postgres_container pg_dump postgres://pgroot:1234@localhost:5433/local --inserts \
    --column-inserts --encoding=utf_8 --rows-per-insert=1000 > dump.sql
```

4. Load the dump into YDB:

```
ydb tools pg-convert --ignore-unsupported -i dump.sql | psql postgresql://ydbroot:4321@localhost:5432/local
```

This command uses YDB CLI to convert the dump.sql file to the format readable by the YDB PostgreSQL compatibility layer. The converted dump file is then redirected to the `psql` utility for loading the data into YDB via PostgreSQL protocol.

CREATE TABLE

> **Warning**
>
> At the moment, YDB's compatibility with PostgreSQL **is under development**, so not all PostgreSQL constructs and functions are supported yet. PostgreSQL compatibility is available for testing in the form of a Docker container, which can be deployed by following these instructions.

The `CREATE TABLE` statement is used to create an empty table in the current database. The syntax of the command is:

```
CREATE [TEMPORARY | TEMP] TABLE <table name> (

<column name> <column data type> [COLLATE][PRIMARY KEY]

[CONSTRAINT  <constraint name> [PRIMARY KEY <column name>],
...]

);
```

When creating a table, you can specify:

1. **Table Type**: `TEMPORARY` / `TEMP` – a temporary table that is automatically deleted at the end of the session. If this parameter is not set (left empty), a permanent table is created. Any indexes created on a temporary table will also be deleted at the end of the session, which means that they are temporary as well. A temporary table and a permanent table with the same name are allowed, in which case a temporary table will be selected.
2. **Table Name**: `<table name>` – you can use English letters in lowercase, numbers, underscores and dollar signs ($). For example, the table name "People" will be stored as "people". For more information, see Identifiers and Key Words.
3. **Column Name**: `<column name>` – the same naming rules apply as for table names.
4. **Data Type**: `<column data type>` – standard PostgreSQL data types are specified.
5. **Collation Rule**: `COLLATE` – collation rules allow setting sorting order and character classification features in individual columns or even when performing individual operations. Sortable types include: `text`, `varchar`, and `char`. You can specify the locale (e.g., `en_US`, `ru_RU`) used to determine the sorting and string comparison rules in the specified columns.
6. Table's Primary Key: `PRIMARY KEY` – a mandatory condition when creating a table in YDB's PostgreSQL compatibility mode.
7. Table-level Constraints (there can be multiple, delimited by commas): `CONSTRAINT` – this type of constraint is used as an alternative syntax to column constraints, or when the same constraint conditions need to be applied to multiple columns. To specify a constraint, you need to state:
   - The keyword `CONSTRAINT`.
   - The constraint name `<constraint name>`. The rules for creating an identifier for the constraint are the same as for table names and column names.
   - The constraint. For example, a primary key constraint can be defined for a single column as `PRIMARY KEY (<column name>)` or for multiple columns as a composite key: `PRIMARY KEY (<column name1>, <column name2>, ...)`.

Creating two tables with primary key autoincrement

Table `people`	Table `social_card`

o create an auto-increment that

```
CREATE TABLE people (
    id                Serial PRIMARY KEY,
    name              Text,
    lastname          Text,
    age               Int,
    country           Text,
    state             Text,
    city              Text,
    birthday          Date,
    sex               Text,
    social_card_number Int
);
```

```
CREATE TABLE social_card (
    id                  Serial PRIMARY KEY,
    social_card_number  Int,
    card_holder_name    Text,
    card_holder_lastname Text,
    issue               Date,
    expiry              Date,
    issuing_authority   Text,
    category            Text
);
```

In this example, we created the "people" table with a constraint block (`CONSTRAINT`), where we defined a primary key (`PRIMARY KEY`) consisting of the "id" column. An alternative notation could look like this: `PRIMARY KEY(id)` without mentioning the `CONSTRAINT` keyword.

Creating a temporary table

```
CREATE TEMPORARY TABLE people (
    id serial PRIMARY KEY,
    name TEXT NOT NULL
);
```

The temporary table is defined using the `TEMPORARY` or `TEMP` keywords.

Creating a table with sorting conditions

```
CREATE TABLE people (
    id                  Serial PRIMARY KEY,
    name                Text COLLATE "en_US",
    lastname            Text COLLATE "en_US",
    age                 Int,
    country             Text,
    state               Text,
    city                Text,
    birthday            Date,
    sex                 Text,
    social_card_number  Int
);
```

In this example, the "name" and "lastname" columns use sorting with `en_US` localization.

Note

Unlike PostgreSQL, YDB uses optimistic locking. This means that transactions check the conditions for the necessary locks at the end of their operation, not at the beginning. If the lock has been violated during the transaction's execution, such a transaction will end with a `Transaction locks invalidated` error. In this case, you can try to execute a similar transaction again.

DROP TABLE

Warning

At the moment, YDB's compatibility with PostgreSQL **is under development**, so not all PostgreSQL constructs and functions are supported yet. PostgreSQL compatibility is available for testing in the form of a Docker container, which can be deployed by following these instructions.

Syntax of the `DROP TABLE` statement:

```
DROP TABLE [IF EXISTS] <table name>;
```

The `DROP TABLE <table name>;` statement is used to delete a table. For example: `DROP TABLE people;`. If the table being deleted does not exist – an error message will be displayed:

```
Error: Cannot find table '...' because it does not exist or you do not have access permissions.
Please check correctness of table path and user permissions., code: 2003.
```

In a number of scenarios, such behavior is not required. For example, if we want to ensure the creation of a new table by deleting the previous one within a single SQL script or a sequence of SQL commands. In such cases, the instruction `DROP TABLE IF EXISTS <table name>` is used. If the table does not exist, the instruction will return a `DROP TABLE` message, not an error.

Note

Unlike PostgreSQL, YDB uses optimistic locking. This means that transactions check the conditions for the necessary locks at the end of their operation, not at the beginning. If the lock has been violated during the transaction's execution, such a transaction will end with a `Transaction locks invalidated` error. In this case, you can try to execute a similar transaction again.

DELETE FROM

Warning

At the moment, YDB's compatibility with PostgreSQL **is under development**, so not all PostgreSQL constructs and functions are supported yet. PostgreSQL compatibility is available for testing in the form of a Docker container, which can be deployed by following these instructions.

Syntax of the `DELETE FROM` statement:

```
DELETE FROM <table name>
WHERE <column name><condition><value/range>;
```

To delete a row from a table based on a specific column value, the construction `DELETE FROM <table name> WHERE <column name> <condition><value/range>` is used.

Warning

Note that the use of the `WHERE ...` clause is optional, so when working with `DELETE FROM` it is very important to avoid accidentally executing the command before specifying the `WHERE ...` clause.

Note

Unlike PostgreSQL, YDB uses optimistic locking. This means that transactions check the conditions for the necessary locks at the end of their operation, not at the beginning. If the lock has been violated during the transaction's execution, such a transaction will end with a `Transaction locks invalidated` error. In this case, you can try to execute a similar transaction again.

INSERT INTO

> **Warning**
>
> At the moment, YDB's compatibility with PostgreSQL **is under development**, so not all PostgreSQL constructs and functions are supported yet. PostgreSQL compatibility is available for testing in the form of a Docker container, which can be deployed by following these instructions.

Syntax of the `INSERT INTO` statement:

```
INSERT INTO <table name> (<column name>, ...)
VALUES (<value>);
```

The `INSERT INTO` statement is adds rows to a table. It can insert one or several rows in a single execution. Example of inserting a single row into the "people" table:

```
INSERT INTO people (name, lastname, age, country, state, city, birthday, sex)
VALUES ('John', 'Doe', 30, 'USA', 'California', 'Los Angeles', CAST('1992-01-15' AS Date), 'Male');
```

In this query, we did not specify the `id` column and did not assign a value to it. This is intentional, as the "id" column in the "people" table is set to the `Serial` data type. When executing the `INSERT INTO` statement, the value of the "id" column will be assigned automatically, taking into account previous values, the current "id" value will be incremented.

For inserting multiple rows into a table, the same construction is used with the enumeration of groups of data to be inserted, separated by commas:

```
INSERT INTO people (name, lastname, age, country, state, city, birthday, sex)
VALUES
    ('Jane', 'Smith', 25, 'Canada', 'Ontario', 'Toronto', CAST('1997-08-23' AS Date), 'Female'),
    ('Alice', 'Johnson', 28, 'UK', 'England', 'London', CAST('1994-05-05' AS Date), 'Female'),
    ('Bob', 'Brown', 40, 'USA', 'Texas', 'Dallas', CAST('1982-12-10' AS Date), 'Male'),
    ('Charlie', 'Davis', 35, 'Canada', 'Quebec', 'Montreal', CAST('1987-02-17' AS Date), 'Male'),
    ('Eve', 'Martin', 29, 'UK', 'Scotland', 'Edinburgh', CAST('1993-11-21' AS Date), 'Female'),
    ('Frank', 'White', 45, 'USA', 'Florida', 'Miami', CAST('1977-03-14' AS Date), 'Male'),
    ('Grace', 'Clark', 50, 'Canada', 'British Columbia', 'Vancouver', CAST('1972-04-26' AS Date), 'Female'),
    ('Hank', 'Miller', 33, 'UK', 'Wales', 'Cardiff', CAST('1989-07-30' AS Date), 'Male'),
    ('Ivy', 'Garcia', 31, 'USA', 'Arizona', 'Phoenix', CAST('1991-09-05' AS Date), 'Female'),
    ('Jack', 'Anderson', 22, 'Canada', 'Manitoba', 'Winnipeg', CAST('2000-06-13' AS Date), 'Male'),
    ('Kara', 'Thompson', 19, 'UK', 'Northern Ireland', 'Belfast', CAST('2003-10-18' AS Date), 'Female'),
    ('Liam', 'Martinez', 55, 'USA', 'New York', 'New York City', CAST('1967-01-29' AS Date), 'Male'),
    ('Molly', 'Robinson', 40, 'Canada', 'Alberta', 'Calgary', CAST('1982-12-01' AS Date), 'Female'),
    ('Noah', 'Lee', 47, 'UK', 'England', 'Liverpool', CAST('1975-05-20' AS Date), 'Male'),
    ('Olivia', 'Gonzalez', 38, 'USA', 'Illinois', 'Chicago', CAST('1984-03-22' AS Date), 'Female'),
    ('Paul', 'Harris', 23, 'Canada', 'Saskatchewan', 'Saskatoon', CAST('1999-08-19' AS Date), 'Male'),
    ('Quinn', 'Lewis', 34, 'UK', 'England', 'Manchester', CAST('1988-07-25' AS DATE), 'Female'),
    ('Rachel', 'Young', 42, 'USA', 'Ohio', 'Cleveland', CAST('1980-02-03' AS Date), 'Female');
```

In both examples, to specify the release date of the movie, we used the `CAST()` function, which is used to convert one data type to another. In this case, using the keyword `AS` and the data type `Date`, we explicitly indicated that we want to convert the string representation of the date in ISO8601 format.

You can specify the required data type, for example, `DATE`, by using the type cast operator `::`, which is a PostgreSQL-specific syntax for explicit conversion of a value from one data type to another. This is in contrast to the `CAST` function, which is used more broadly across different SQL databases for the same purpose. An example of using the `::` operator might look like this: `'2023-01-01'::DATE`, which explicitly converts the string to a `DATE` type, ensuring the database treats the value as a date. This explicit casting with `::` is particularly useful when you want to override implicit type conversion rules of the database.

An example of using the `::` operator might look like this:

```
INSERT INTO people (name, lastname, age, country, state, city, birthday, sex)
VALUES ('Sam', 'Walker', 60, 'Canada', 'Nova Scotia', 'Halifax', '1962-04-15'::Date, 'Male');
```

Note

Unlike PostgreSQL, YDB uses optimistic locking. This means that transactions check the conditions for the necessary locks at the end of their operation, not at the beginning. If the lock has been violated during the transaction's execution, such a transaction will end with a `Transaction locks invalidated` error. In this case, you can try to execute a similar transaction again.

SELECT

> **Warning**
>
> At the moment, YDB's compatibility with PostgreSQL **is under development**, so not all PostgreSQL constructs and functions are supported yet. PostgreSQL compatibility is available for testing in the form of a Docker container, which can be deployed by following these instructions.

Syntax of the `SELECT` statement:

```
SELECT [<table column>, ... | *]
FROM [<table name> | <sub query>] AS <table name alias>
LEFT | RIGHT | CROSS | INNER JOIN <another table> AS <table name alias> ON <join condition>
WHERE <condition>
GROUP BY <table column>
HAVING <condition>
UNION | UNION ALL | EXCEPT | INTERSECT
ORDER BY <table column> [ASC | DESC]
LIMIT [<limit value>]
OFFSET <offset number>
```

Calling SELECT without specifying a target table

`SELECT` is used to return computations to the client side and can be used even without specifying a table, as seen in constructs like `SELECT NOW()`, or for performing operations such as working with dates, converting numbers, or calculating string lengths. However, `SELECT` is also used in conjunction with `FROM ...` to retrieve data from a specified table. When used with `INSERT INTO ...`, `SELECT` serves to select data that will be inserted into another table. In subqueries, `SELECT` is utilized within a larger query, not necessarily with `FROM ...`, to contribute to the overall computation or condition.

For example, `SELECT` can be used for working with dates, converting numbers, or calculating string length:

```
SELECT CURRENT_DATE + INTERVAL '1 day';  -- Returns tomorrow's date
SELECT LENGTH('Hello');  -- Returns the length of the string 'Hello'
SELECT CAST('123' AS INTEGER);  -- Converts the string '123' to an integer
```

Such use of `SELECT` can be useful for testing, debugging expressions, or SQL functions without accessing a real table, but more often `SELECT` is used to retrieve rows from one or more tables.

Retrieving values from one or multiple columns

To return values from one or several columns of a table, `SELECT` is used in the following form:

```
SELECT <column name> , <column name>
FROM <table name>;
```

To read all data from a table, for example, the `people` table, you need to execute the command `SELECT * FROM people;`, where `*` is the special symbol indicating all columns. With this statement, all rows from the table will be returned with data from all columns.

To display the "id", "name", and "lastname" columns for all rows of the `people` table, you can do it as follows:

```
SELECT id, name, lastname
FROM people;
```

Limiting the results obtained from a query using WHERE

To select only a subset of rows, the `WHERE` clause with filtering criteria is used: `WHERE <column name> <condition> <column value>;`:

```
SELECT id, name, lastname
FROM people
WHERE age > 30;
```

`WHERE` allows the use of multiple conditional selection operators (`AND`, `OR`) to create complex selection conditions, such as ranges:

```
SELECT id, name, lastname
FROM people
WHERE age > 30 AND age < 45;
```

Retrieving a subset of rows using LIMIT and OFFSET conditions

To limit the number of rows in the result set, `LIMIT` is used with the specified number of rows:

```
SELECT id, name, lastname
FROM people
WHERE age > 30 AND age < 45
LIMIT 5;
```

Thus, the first 5 rows from the query will be printed out. With `OFFSET`, you can specify how many rows to skip before starting to print out rows:

```
SELECT id, name, lastname
FROM people
WHERE age > 30 AND age < 45
OFFSET 3
LIMIT 5;
```

When specifying `OFFSET 3`, the first 3 rows of the resulting selection from the `people` table will be skipped.

Sorting the results of a query using ORDER BY

By default, the database does not guarantee the order of returned rows, and it may vary from query to query. If a specific order of rows is required, the `ORDER BY` clause is used with the designation of the column for sorting and the direction of the sort:

```
SELECT id, name, lastname, age
FROM people
WHERE age > 30 AND age < 45
ORDER BY age DESC;
```

Sorting is applied to the results returned by the `SELECT` clause, not to the original columns of the table specified in the `FROM` clause. Sorting can be done in ascending order – `ASC` (from smallest to largest - this is the default option and does not need to be specified) or in descending order – `DESC` (from largest to smallest). How sorting is executed depends on the data type of the column. For example, strings are stored in utf-8 and are compared according to "unicode collate" (based on character codes).

Grouping the results of a query from one or more tables using GROUP BY

`GROUP BY` is used to aggregate data across multiple records and group the results by one or several columns. The syntax for using `GROUP BY` is as follows:

```
SELECT <column name>, <column name>, ...
FROM <table name>
[WHERE <column name> = <value>]
GROUP BY <column name>, <column name>, ...;
[HAVING <column name> = <limit column value>]
[LIMIT <value>]
[OFFSET <value>]
```

Example of grouping data from the "people" table by gender ("sex") and age ("age") with a selection limit (`WHERE`) based on age:

```
SELECT sex, age
FROM people
WHERE age > 40
GROUP BY sex, age;
```

In the previous example, we used `WHERE` – an optional parameter for filtering the result, which filters individual rows before applying `GROUP BY`. In the next example, we use `HAVING` to exclude from the result the rows of groups that do not meet the condition. `HAVING` filters the rows of groups created by `GROUP BY`. When using `HAVING`, the query becomes grouped, even if `GROUP BY` is absent. All selected rows are considered to form one group, and in the `SELECT` list and `HAVING` clause, one can refer to the table columns only from aggregate functions. Such a query will yield a single row if the result of the `HAVING` condition is true, and zero rows otherwise.

Examples for HAVING

`HAVING` + `GROUP BY`	`HAVING` + `WHERE` + `GROUP BY`

```sql
SELECT sex, country, age
FROM people
GROUP BY sex, country, age
HAVING sex = 'Female';
```

```sql
SELECT sex, name,age
FROM people
WHERE age > 40
GROUP BY sex, name, age
HAVING sex = 'Female';
```

`SELECT` can be applied to multiple tables with the specification of the type of table join. The joining of tables is set through the `JOIN` clause, which comes in five types: `LEFT JOIN`, `RIGHT JOIN`, `INNER JOIN`, `CROSS JOIN`, `FULL JOIN`. When a `JOIN` is performed on a specific condition, such as a key, and one of the tables has several rows with the same value of this key, a Cartesian product occurs. This means that each row from one table will be joined with every corresponding row from the other table.

Joining tables using LEFT JOIN, RIGHT JOIN, or INNER JOIN

The syntax for `SELECT` using `LEFT JOIN`, `RIGHT JOIN`, `INNER JOIN`, `FULL JOIN` is the same:

```sql
SELECT <table name left>.<column name>, ... ,
FROM <table name left>
LEFT | RIGHT | INNER | FULL JOIN <table name right> AS <table name right alias>
ON <table name left>.<column name> = <table name right>.<column name>;
```

All `JOIN` modes, except `CROSS JOIN`, use the keyword `ON` for joining tables. In the case of `CROSS JOIN`, its usage syntax will be as follows: `CROSS JOIN <table name> AS <table name alias>;`. Let's consider an example of using each `JOIN` mode separately.

LEFT JOIN

Returns all rows from the left table and the matching rows from the right table. If there are no matches, it returns `NULL` (the output will be empty) for all columns of the right table. Example of using `LEFT JOIN`:

```sql
SELECT people.name, people.lastname, card.social_card_number
FROM people
LEFT JOIN social_card AS card
ON people.name = card.card_holder_name AND people.lastname = card.card_holder_lastname;
```

The result of executing an SQL query using `LEFT JOIN` without one record in the right table `social_card`:

```
 name    | lastname | social_card_number
---------+----------+--------------------
 John    | Doe      |          123456789
 Jane    | Smith    |          223456789
 Alice   | Johnson  |          323456789
 Bob     | Brown    |          423456789
 Charlie | Davis    |          523456789
 Eve     | Martin   |          623456789
 Frank   | White    |
```

RIGHT JOIN

Returns all rows from the right table and the matching rows from the left table. If there are no matches, it returns `NULL` for all columns of the left table. This type of `JOIN` is rarely used, as its functionality can be replaced by `LEFT JOIN`, and swapping the tables. Example of using `RIGHT JOIN`:

```sql
SELECT people.name, people.lastname, card.social_card_number
FROM people
RIGHT JOIN social_card AS card
ON people.name = card.card_holder_name AND people.lastname = card.card_holder_lastname;
```

The result of executing an SQL query using `RIGHT JOIN` without one record in the left table `people`:

```
 name    | lastname | social_card_number
---------+----------+--------------------
John     | Doe      |          123456789
Jane     | Smith    |          223456789
Alice    | Johnson  |          323456789
Bob      | Brown    |          423456789
Charlie  | Davis    |          523456789
Eve      | Martin   |          623456789
         |          |          723456789
```

INNER JOIN

Returns rows when there are matching values in both tables. Excludes from the results those rows for which there are no matches in the joined tables. Example of using `INNER JOIN`:

```
SELECT people.name, people.lastname, card.social_card_number
FROM people
RIGHT JOIN social_card AS card
ON people.name = card.card_holder_name AND people.lastname = card.card_holder_lastname;
```

Such an SQL query will return only those rows for which there are matches in both tables:

```
 name    | lastname | social_card_number
---------+----------+--------------------
John     | Doe      |          123456789
Jane     | Smith    |          223456789
Alice    | Johnson  |          323456789
Bob      | Brown    |          423456789
Charlie  | Davis    |          523456789
Eve      | Martin   |          623456789
```

CROSS JOIN

Returns the combined result of every row from the left table with every row from the right table. `CROSS JOIN` is usually used when all possible combinations of rows from two tables are needed. `CROSS JOIN` simply combines each row of one table with every row of another without any condition, which is why its syntax lacks the `ON` keyword: `CROSS JOIN <table name> AS <table name alias>;`.

Example of using `CROSS JOIN` with the result output limited by `LIMIT 5`:

```
SELECT people.name, people.lastname, card.social_card_number
FROM people
CROSS JOIN social_card AS card
LIMIT 5;
```

The example above will return all possible combinations of columns participating in the selection from the two tables:

```
name | lastname | social_card_number
------+----------+--------------------
 John | Doe      |          123456789
 John | Doe      |          223456789
 John | Doe      |          323456789
 John | Doe      |          423456789
 John | Doe      |          523456789
```

FULL JOIN

Returns both matched and unmatched rows in both tables, filling in `NULL` for columns from the table for which there is no match. Example of executing an SQL query using `FULL JOIN`:

```
SELECT people.name, people.lastname, card.social_card_number
FROM people
FULL JOIN social_card AS card
ON people.name = card.card_holder_name AND people.lastname = card.card_holder_lastname;
```

As a result of executing the SQL query, the following output will be returned:

```
 name    | lastname | social_card_number
---------+----------+--------------------
 Liam    | Martinez |         1323456789
```

```
 Eve     | Martin   |          623456789
 Hank    | Miller   |          923456789
 Molly   | Robinson |         1423456789
 Sam     | Walker   |
 Paul    | Harris   |         1723456789
 Kara    | Thompson |         1223456789
         |          |         1923456789
...
```

UPDATE

> **Warning**
>
> At the moment, YDB's compatibility with PostgreSQL **is under development**, so not all PostgreSQL constructs and functions are supported yet. PostgreSQL compatibility is available for testing in the form of a Docker container, which can be deployed by following these instructions.

The syntax of the `UPDATE` statement:

```
UPDATE <table name>
SET <column name> = [<new value>, CASE ... END]
WHERE <search column name> = [<search value>, IN]
```

The `UPDATE ... SET ... WHERE` statements works as follows:

1. **Table name is specified** – `UPDATE <table name>`, where the data will be updated;
2. **Column name is indicated** – `SET <column name>`, where the data will be updated;
3. **New value is set** – `<new value>`;
4. **Search criteria are specified** – `WHERE` with the indication of the search column `<search column name>` and the value that the search criterion should match `<search value>`. If `CASE` is used, then the `IN` operator is specified with a list of values `<column name>`.

Updating a single row in a table with conditions

Update without conditions	Update with conditions

`UPDATE people`
`SET name = 'Alexander'`
`WHERE lastname = 'Doe';`
	`UPDATE people`
	`SET age = 31`
	`WHERE country = 'USA' AND city = 'Los Angeles';`

In the example "Update with conditions", the condition combining operator `AND` is used – the condition will only be satisfied when both parts meet the truth conditions. The operator `OR` can also be used – the condition will be satisfied if at least one part meets the truth conditions. There can be multiple condition operators:

```
UPDATE people
SET age = 31
WHERE country = 'USA' AND city = 'Los Angeles' OR city = 'Florida';
```

Updating a single record in a table using expressions or functions:

Frequently during updates, it is necessary to perform mathematical actions on the data or to modify it using functions.

Update with the use of expressions	Update with the use of functions

```
UPDATE people
SET age = age + 1
WHERE country = 'Canada';
```

```
UPDATE people
SET name = UPPER(name)
WHERE country = 'USA';
```

`umn name> = <column new value>` is made after the

```
WHERE lastname = 'Smith';
```

Updating multiple rows in a table using the **CASE ... END**construction

For simultaneous updating of different values in different rows, the `CASE ... END` instruction can be used with nested data selection conditions `WHEN <column name> <condition> (=, >, <) THEN <new value>`. This is followed by the `WHERE <column name> IN (<column value>, ...)` construct, which allows setting a list of values for which the condition will be executed.

Example of changing the age (`age`) of people (`people`) depending on their names:

```
UPDATE people
SET age = CASE
            WHEN name = 'John' THEN 32
            WHEN name = 'Jane' THEN 26
          END
WHERE name IN ('John', 'Jane');
```

Note

Unlike PostgreSQL, YDB uses optimistic locking. This means that transactions check the conditions for the necessary locks at the end of their operation, not at the beginning. If the lock has been violated during the transaction's execution, such a transaction will end with a `Transaction locks invalidated` error. In this case, you can try to execute a similar transaction again.

BEGIN, COMMIT, ROLLBACK (working with Transactions)

Warning

At the moment, YDB's compatibility with PostgreSQL **is under development**, so not all PostgreSQL constructs and functions are supported yet. PostgreSQL compatibility is available for testing in the form of a Docker container, which can be deployed by following these instructions.

Transactions are a method of grouping one or more database operation into a single unit of work. A transaction may consist of one or several SQL statements and is used to ensure the consistency of data. A transaction guarantees that either all or none of the included SQL statements will be executed. Transactions are managed by the commands `BEGIN`, `COMMIT`, `ROLLBACK`.

Transactions are executed within sessions. A **session** is a single connection to the database, which begins when a client connects to the database and ends upon disconnection. A transaction starts with the `BEGIN` command and ends with the `COMMIT` command (for successful completion) or `ROLLBACK` (to revert). It is not obligatory to specify `BEGIN`, `COMMIT`, and `ROLLBACK` explicitly, they are implied if not specified. If a session is unexpectedly interrupted, then all uncommitted transaction that were initiated in the current session are automatically rolled back.

Let's review each of the commands:

- `BEGIN` initiates a new transaction. After this command is executed, all subsequent database operations are performed within the context of this transaction.
- `COMMIT` completes the current transaction by applying all of its operations. If all operations within the transaction are successful, the results of these operations are made permanent. The changes become visible to subsequent transactions.
- `ROLLBACK` reverts the current transaction, canceling all of its operations, if errors occurred during the transaction's execution or if the transaction is being aborted by the application based on its internal logic. When `ROLLBACK` is called, only the changes made within the current transaction are canceled. Changes made by other transactions (even if they were initiated and completed during the execution of the current transaction) remain unaffected. If an error occurs during the execution of a transaction, further operations within that transaction become impossible – a `ROLLBACK` must be done since performing a `COMMIT` would return an error. If a session is disconnected during an active transaction – a `ROLLBACK` will automatically be executed. For more detailed information about concurrency control (MVCC), refer to this article.

Suppose you need to make changes to different rows in a table for different columns so that the transaction is combined into a single unit of work and has the guarantees of ACID. Such a record may look like this:

```
-- Start of the transaction
BEGIN;

-- Data update instruction
UPDATE people
SET name = 'Ivan'
WHERE id = 1;

-- Another data update instruction
UPDATE people
SET lastname = 'Gray'
WHERE id = 10;

-- If all the data is correct, then you need to execute the transaction confirmation instruction:
COMMIT;
```

Note

Unlike PostgreSQL, YDB uses optimistic locking. This means that transactions check the conditions for the necessary locks at the end of their operation, not at the beginning. If the lock has been violated during the transaction's execution, such a transaction will end with a `Transaction locks invalidated` error. In this case, you can try to execute a similar transaction again.

The page has been moved to a new location. Please update the links.

YDB Monitoring

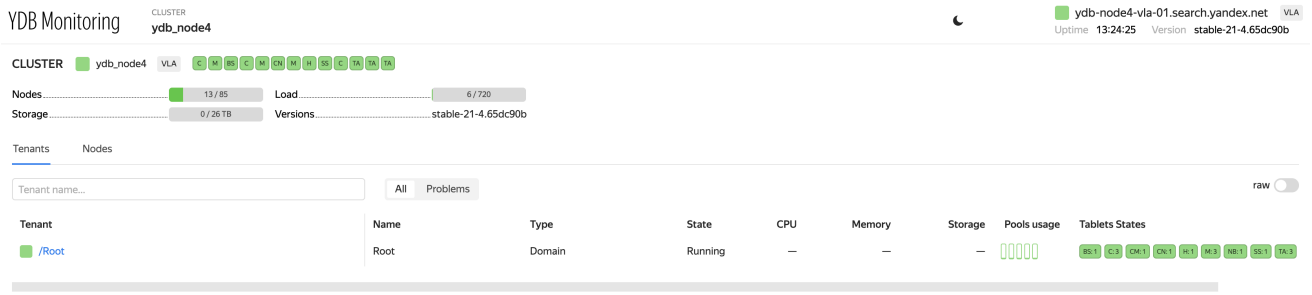
YDB Monitoring is a set of web pages that display the system health according to a range of different aspects. The pages contain lists of components and their current parameters. Many components have a health color indicator before the name.

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**: 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`.
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.

Graphical user interfaces

Environment	Instruction	Compatibility level
Embedded 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	PostgreSQL wire protocol	Instruction
DataLens	Full	Instruction
FineBI	PostgreSQL wire protocol	Instruction
Grafana	Full	Instruction

Orchestration

System	Instruction
Apache Airflow™	Instruction

Data ingestion

Delivery System	Instruction
FluentBit	Instruction
LogStash	Instruction
Kafka Connect Sink	Instruction
Arbitrary JDBC data sources	Instruction

Streaming data ingestion

Delivery System	Instruction
Apache Kafka API	Instruction

Data migrations

Environment	Instruction
goose	Instruction
Liquibase	Instruction
Flyway	Instruction

Object–relational mapping

Delivery System	Instruction
Hibernate	Instruction
Spring Data JDBC	Instruction
JOOQ	Instruction
Dapper	Instruction

Vector search

System	Instruction
LangChain	Instruction

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.

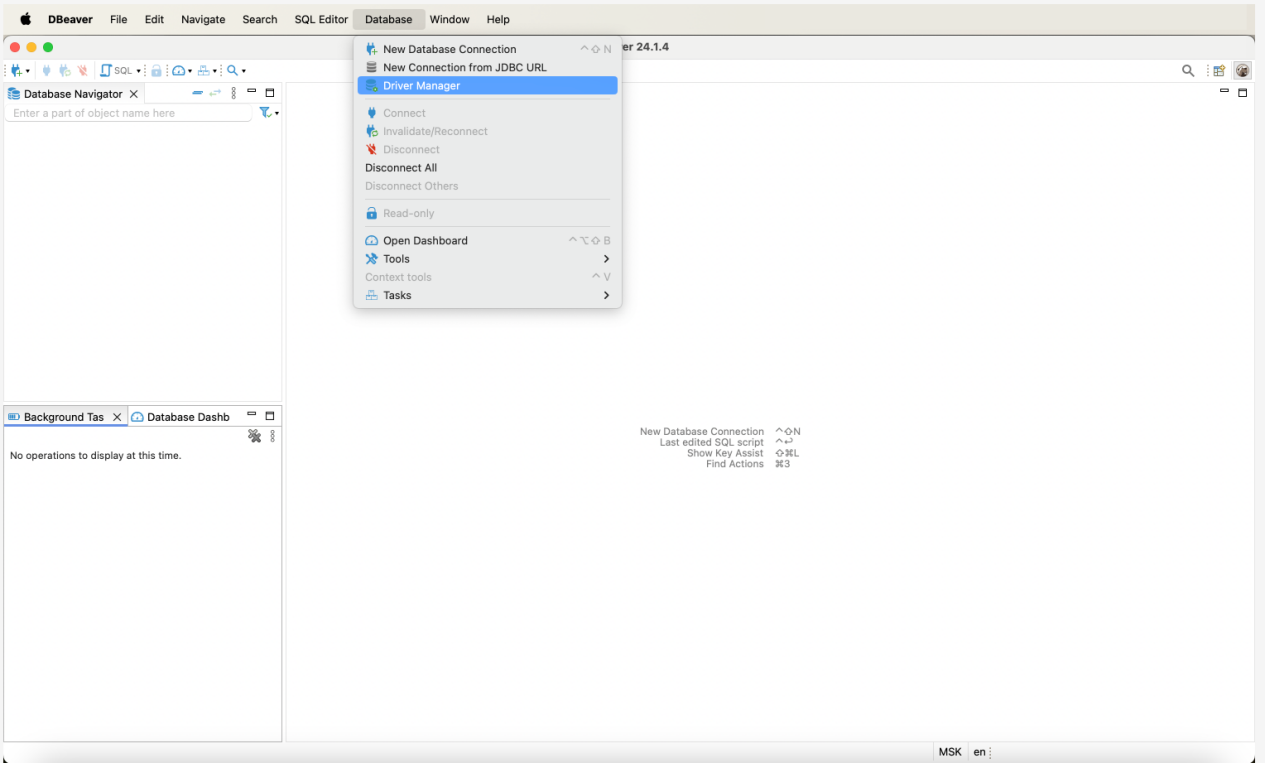
Connecting the YDB JDBC Driver to DBeaver

To connect to YDB from DBeaver, you will need the JDBC driver. Follow these steps to download the JDBC driver:

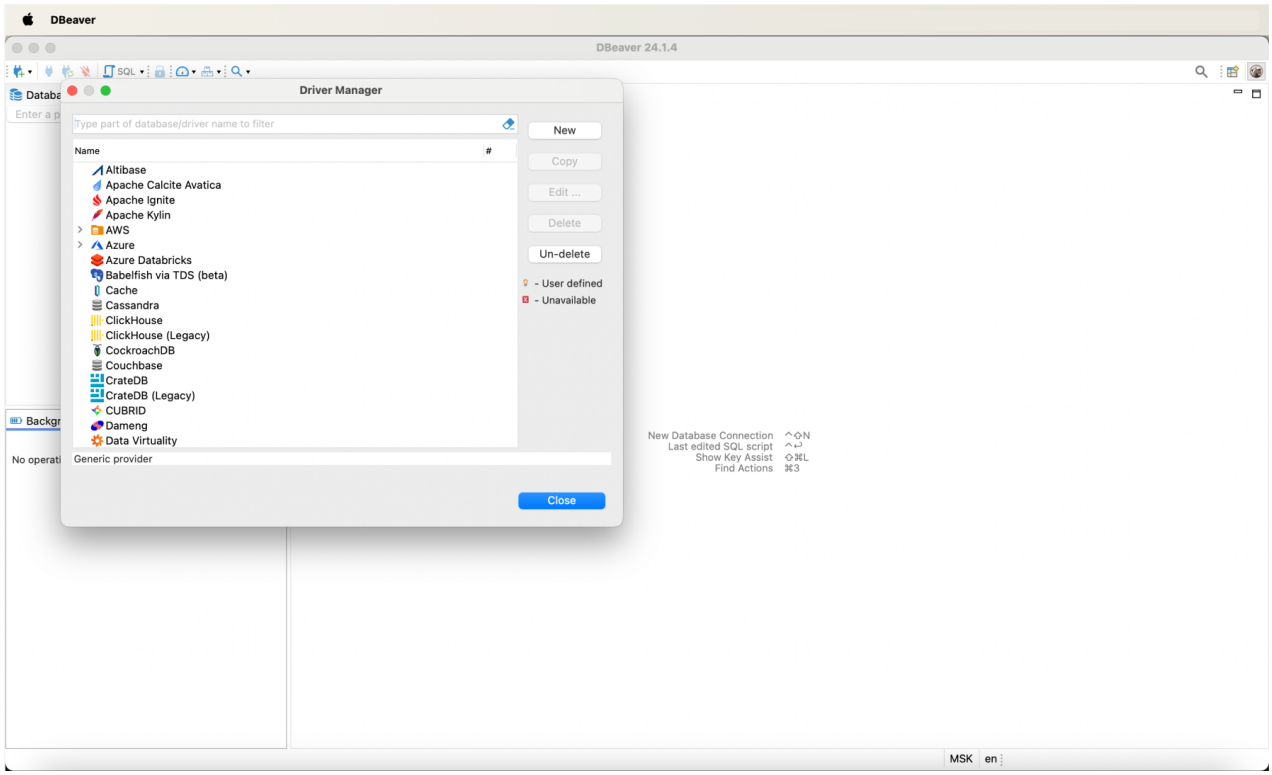
1. Go to the ydb-jdbc-driver repository.
2. Select the latest release (tagged as `Latest`) and save the `ydb-jdbc-driver-shaded-<driver-version>.jar` file.

Follow these steps to connect the downloaded JDBC driver:

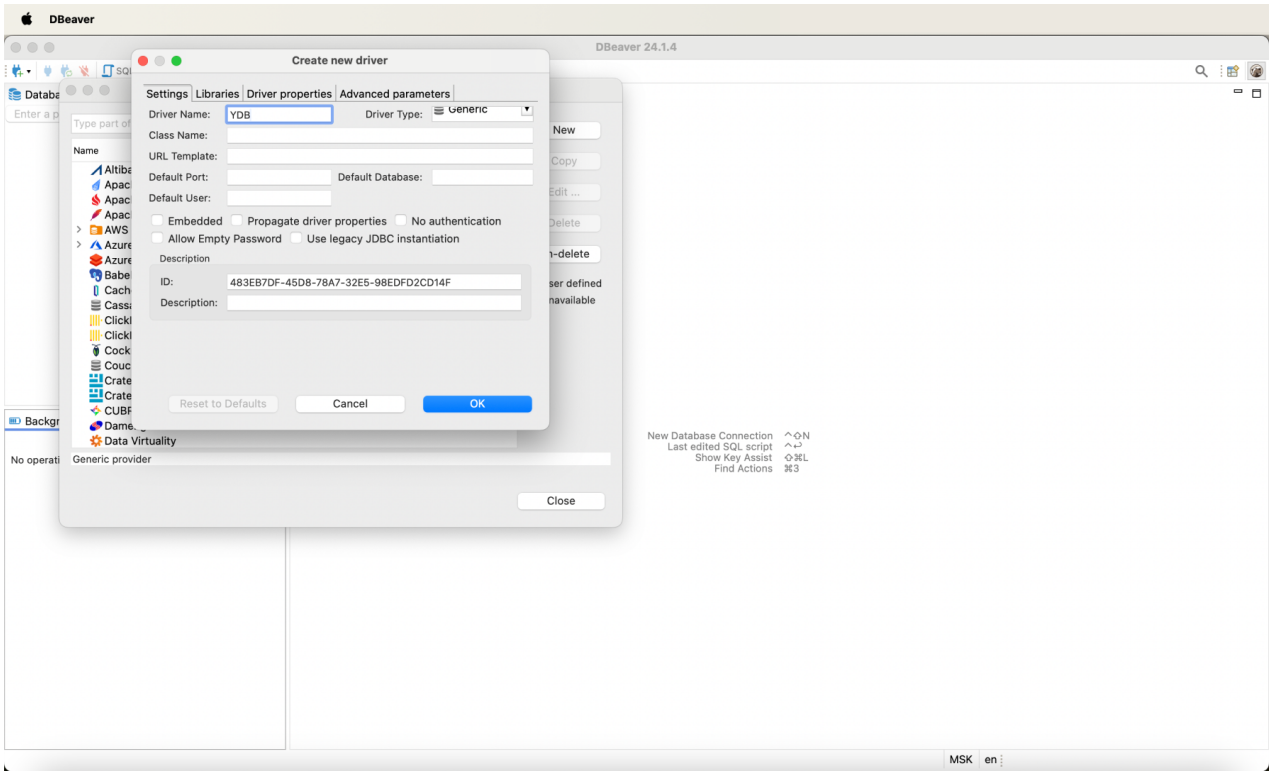
1. In the top menu of DBeaver, select the **Database** option, then select **Driver Manager**:



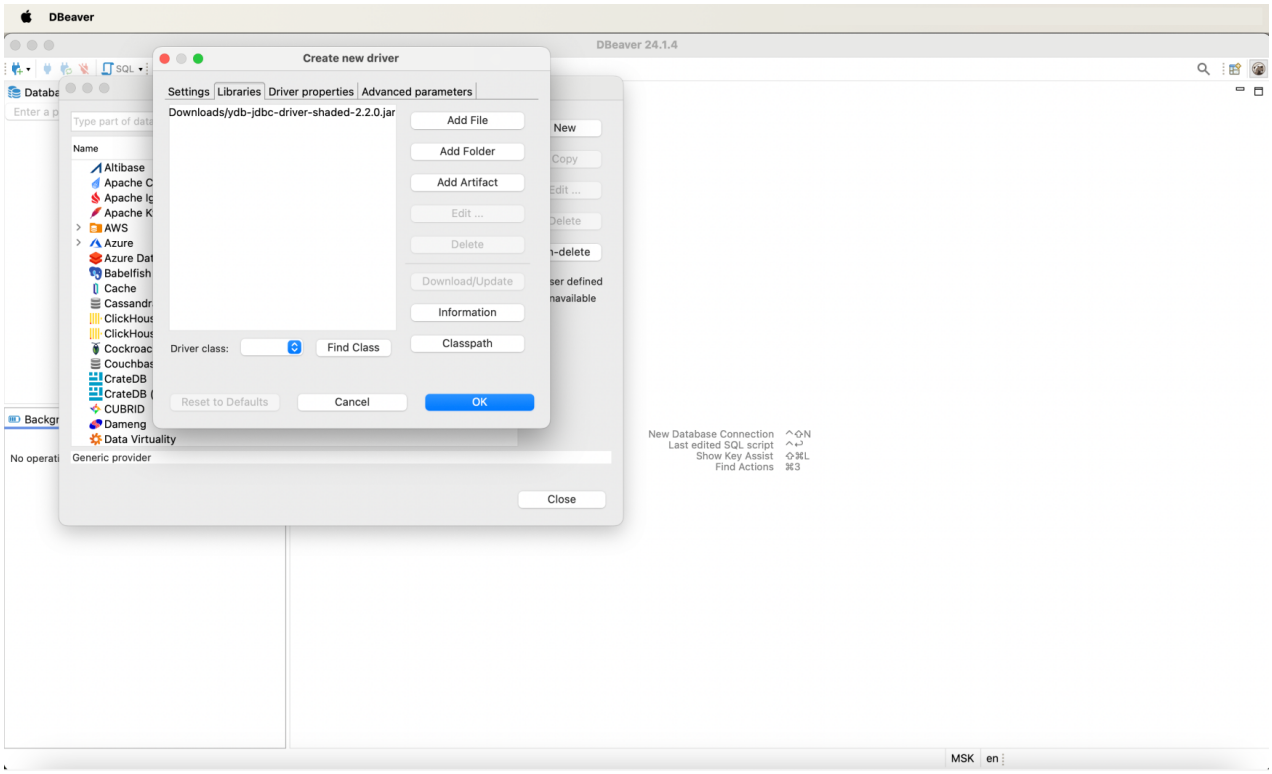
2. To create a new driver, click the **New** button in the **Driver Manager** window that opens



3. In the **Create Driver** window that opens, specify `YDB` in the **Driver Name** field:

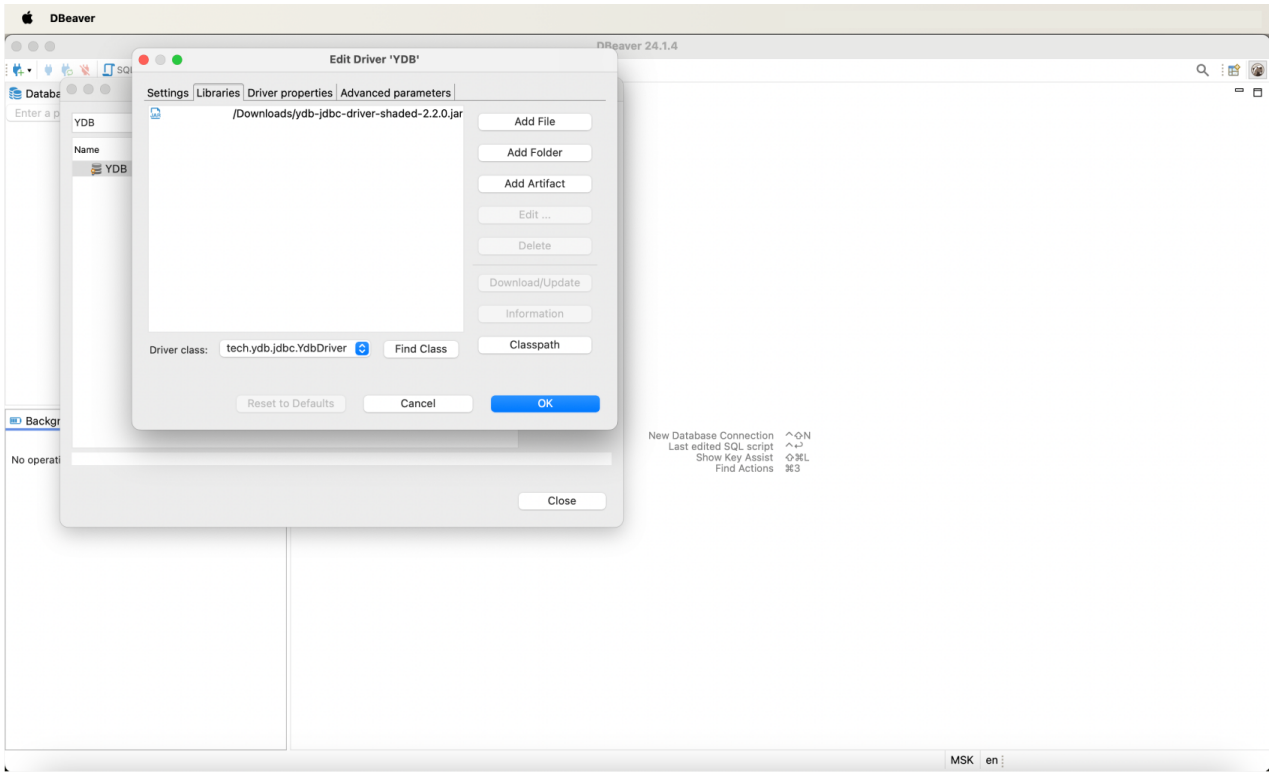


4. Go to the **Libraries** section, click **Add File**, specify the path to the previously downloaded YDB JDBC driver (the `ydb-jdbc-driver-shaded-<driver-version>.jar` file), and click **OK**:



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

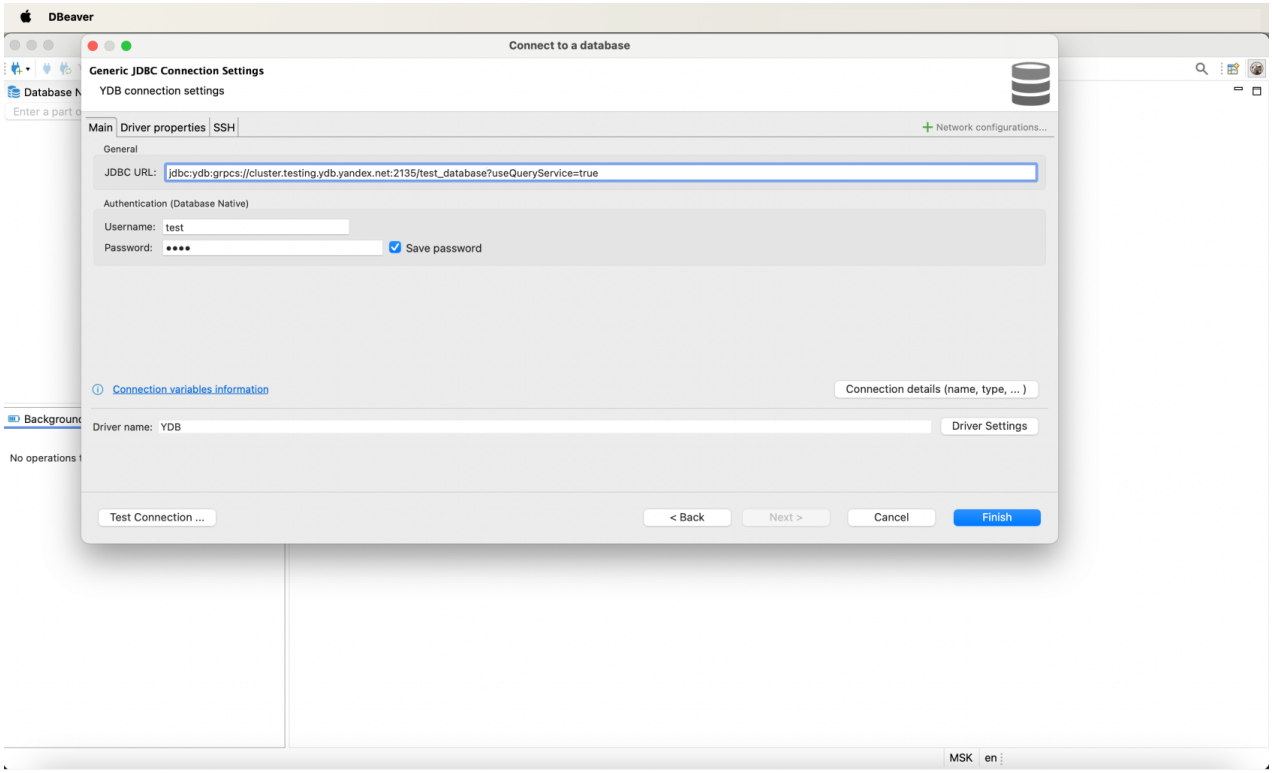
Perform the following steps to establish a connection:

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
```

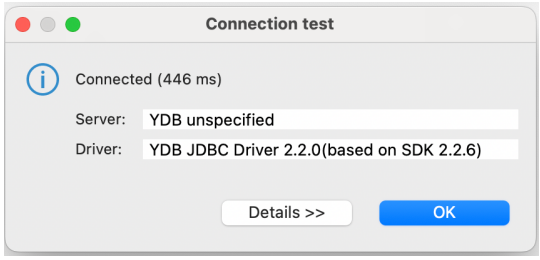
Where:

- `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.
5. Click **Test Connection...** to verify the settings.

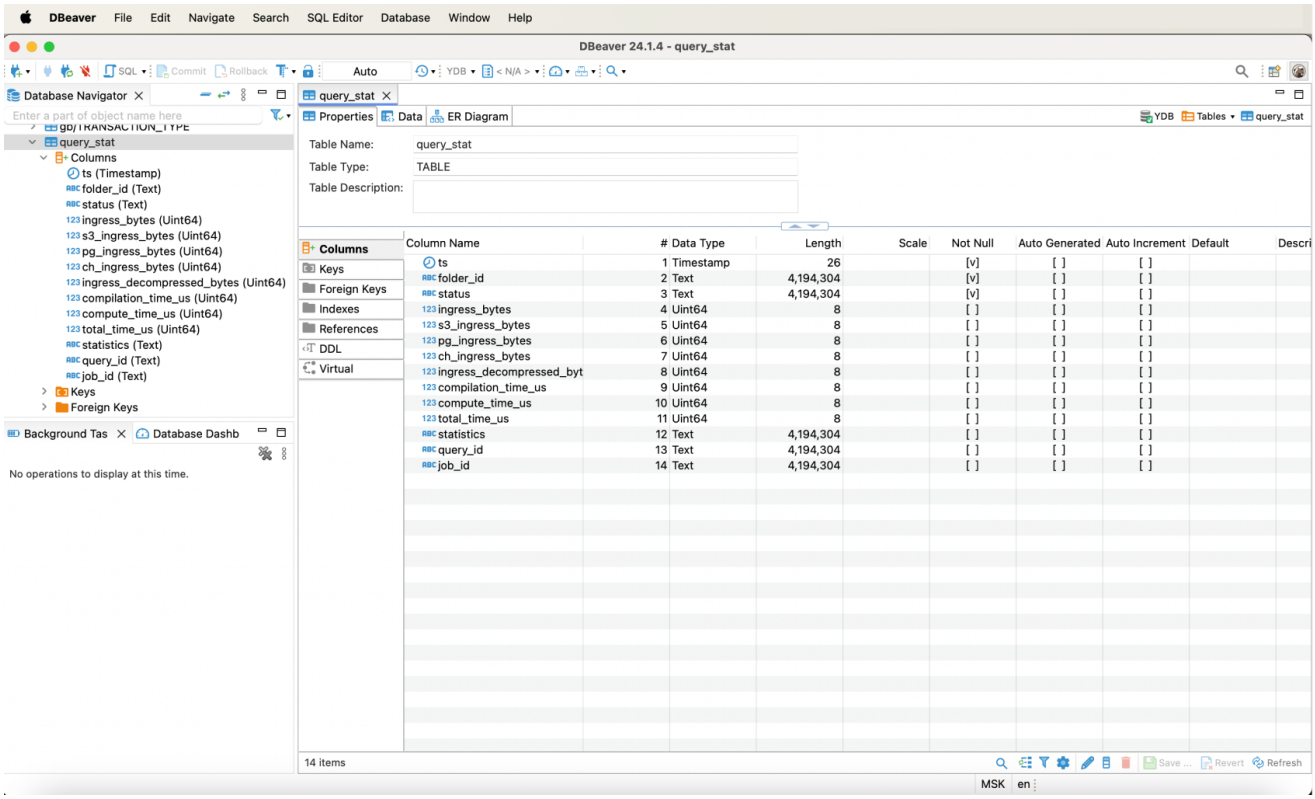
   If all settings are correct, a message indicating successful connection testing will appear:



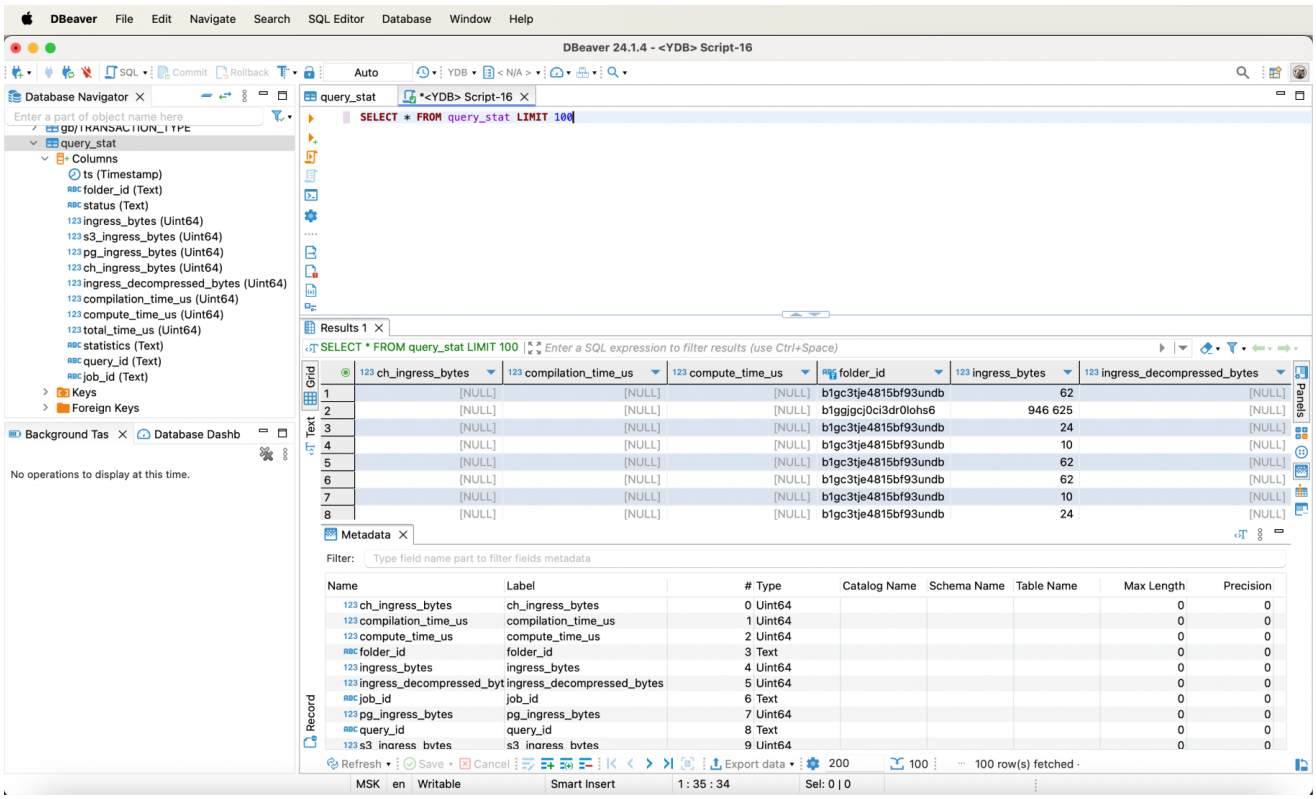
6. Click **Finish** to save the connection.

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 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.

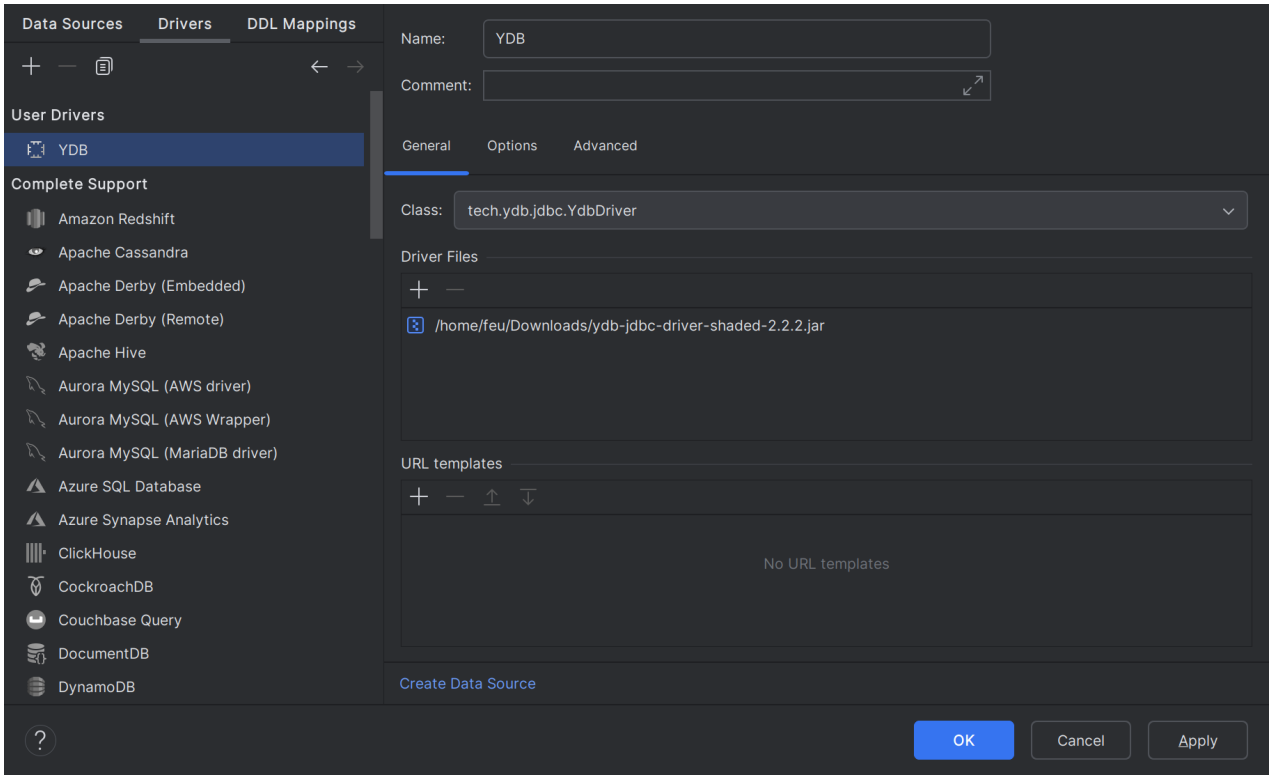
To download the YDB JDBC driver, follow these steps:

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:

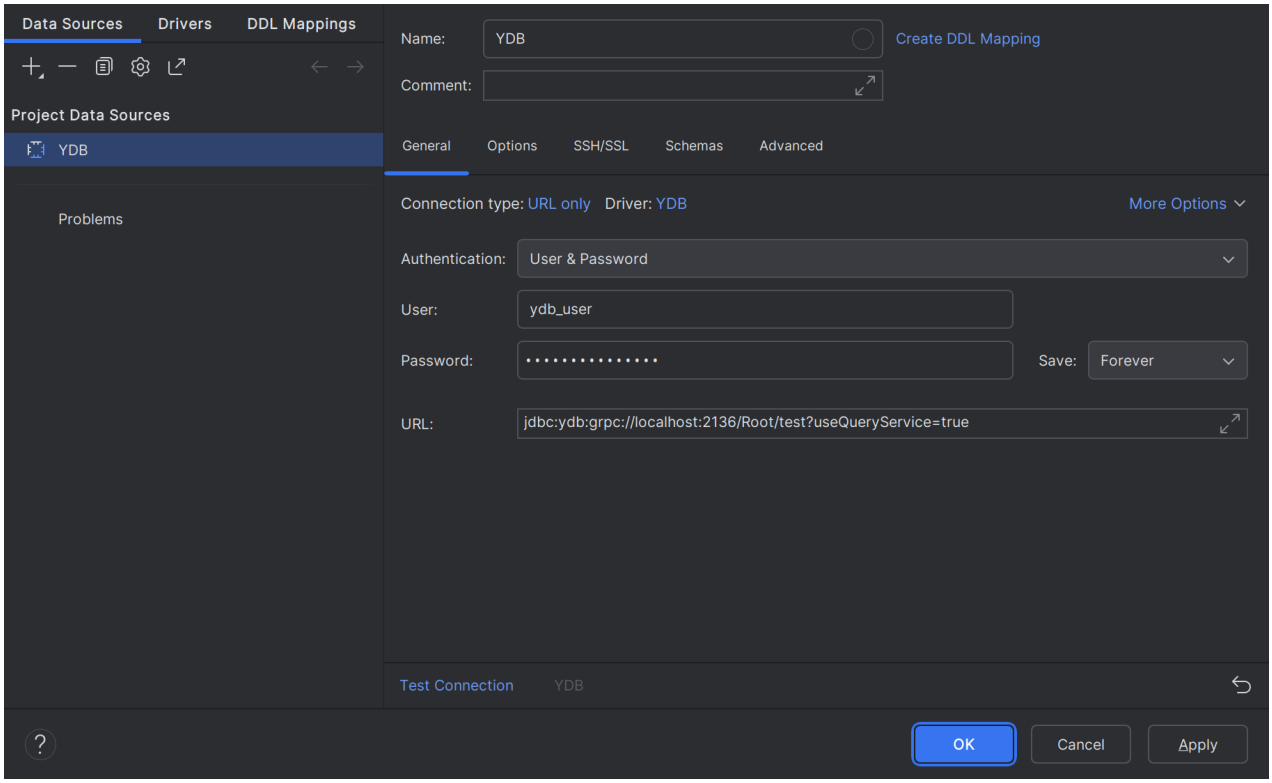
   ```
   jdbc:ydb:<ydb_endpoint>/<ydb_database>?useQueryService=true
   ```

   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.

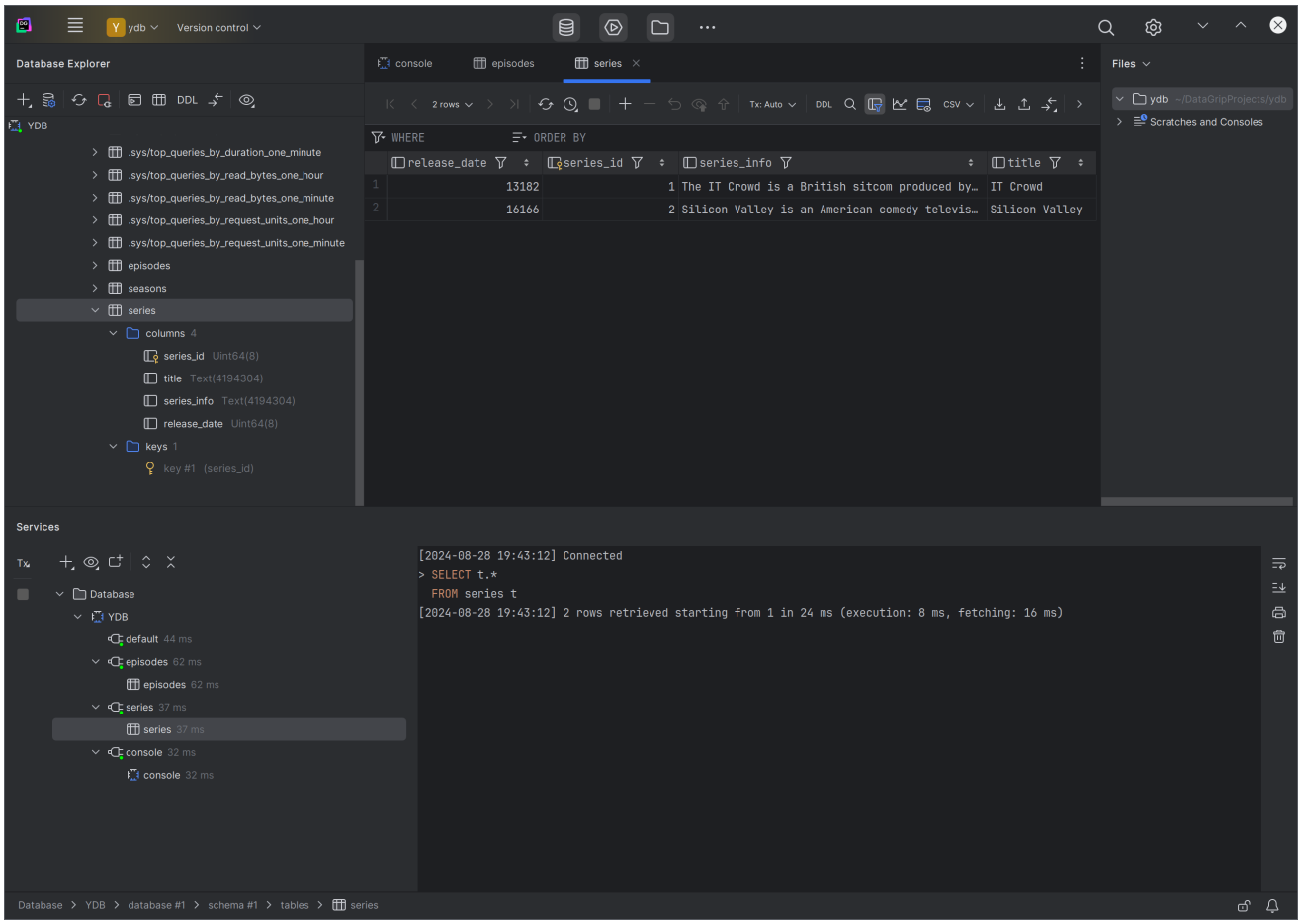


6. Click **Test Connection** to verify the settings.

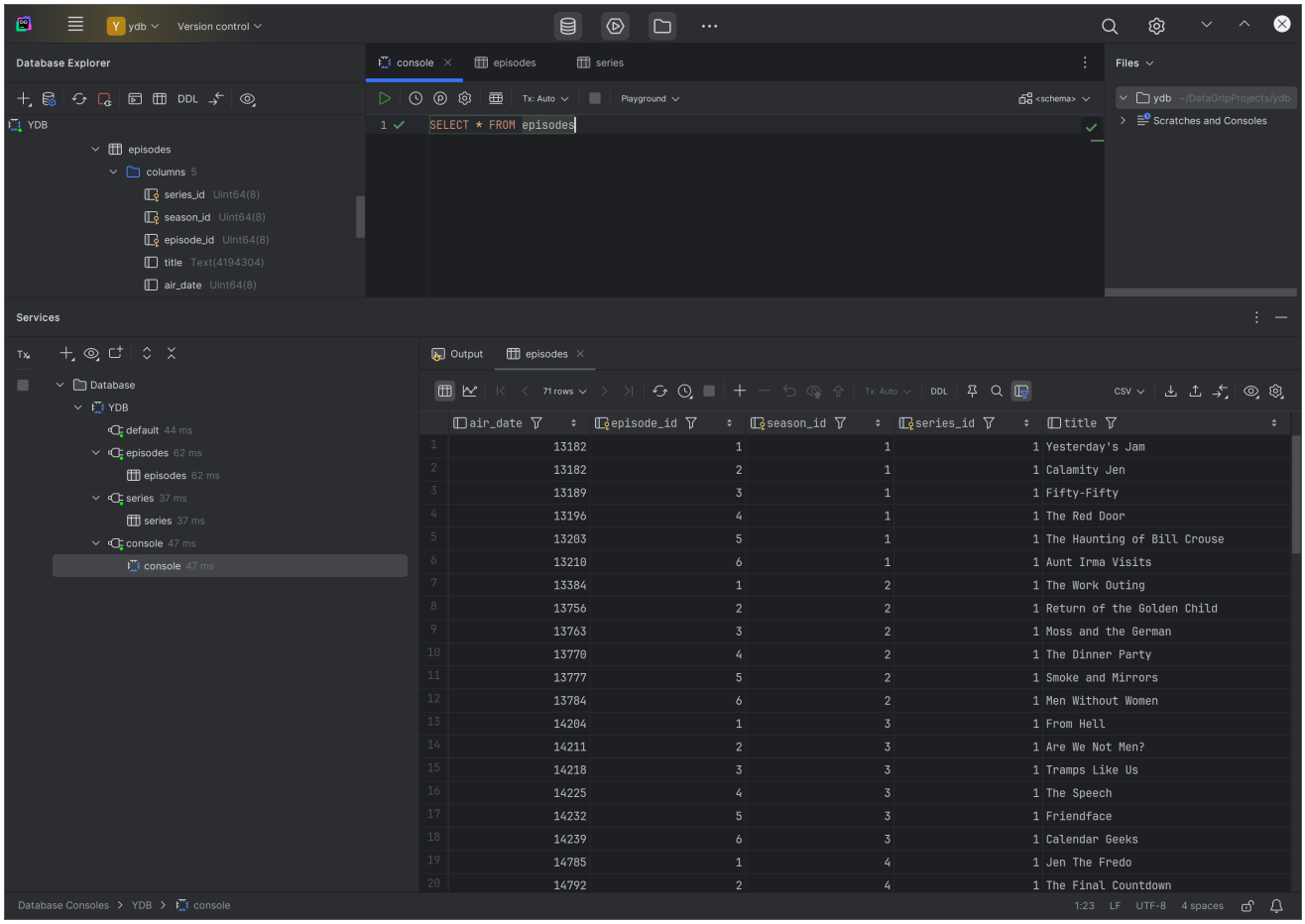
   If all the settings are correct, a message appears indicating a successful connection test.
7. Click **OK** to save the connection.

Working with YDB

With DataGrip you can view the list and structure of tables.



You can also execute queries on the data.


ydb
Version control
Database Explorer
DDL
YDB
episodes
columns 5
series_id Uint64(8)
season_id Uint64(8)
episode_id Uint64(8)
title Text(4194304)
air_date Uint64(8)
console
episodes
series
Tx: Auto
Playground
<schema>
1
SELECT * FROM episodes
Files
ydb ~/DataGripProjects/ydb
Scratches and Consoles
Services
Database
YDB
default 44 ms
episodes 62 ms
episodes 62 ms
series 37 ms
series 37 ms
console 47 ms
console 47 ms
Output
episodes
71 rows
Tx: Auto
DDL
CSV
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
Database Consoles > YDB > console
1:23 LF UTF-8 4 spaces

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 {% file src="https://raw.githubusercontent.com/ydb-platform/ydb-sqlalchemy/refs/heads/main/examples/jupyter_notebook/YDB%20SQLAlchemy%20%2B%20Jupyter%20Notebook%20Example.ipynb" name="YDB SQLAlchemy - Jupyter Notebook Example.ipynb" %}, open it in Jupyter, and follow each cell sequentially, executing code as necessary.

Apache Superset

Apache Superset is a modern data exploration and data visualization platform.

PostgreSQL compatibility mode in YDB enables the use of Apache Superset to query and visualize data from YDB. In this case Apache Superset works with YDB just like with PostgreSQL.

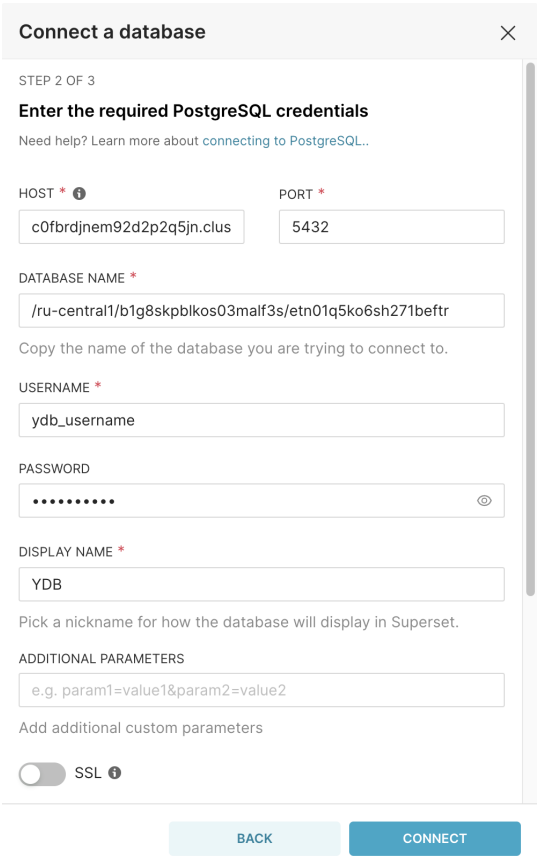
> **Warning**
>
> At the moment, YDB's compatibility with PostgreSQL **is under development**, so not all PostgreSQL constructs and functions are supported yet. PostgreSQL compatibility is available for testing in the form of a Docker container, which can be deployed by following these instructions.

Adding a database connection to YDB

To connect to YDB from Apache Superset using the PostgreSQL wire protocol, 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, click the **PostgreSQL** button.
4. In **Step 2** of the wizard, enter the YDB credentials in the corresponding fields:
   - **HOST**. The endpoint of the YDB cluster to which the connection will be made.
   - **PORT**. The port of the YDB endpoint.
   - **DATABASE NAME**. The path to the database in the YDB cluster where queries will be executed.
   - **USERNAME**. The login for connecting to the YDB database.
   - **PASSWORD**. The password for connecting to the YDB database.
   - **DISPLAY NAME**. The YDB connection name in Apache Superset.



5. Click **CONNECT**.
6. To save the database connection, click **FINISH**.

Creating a dataset

To create a dataset for a YDB table, follow these steps:

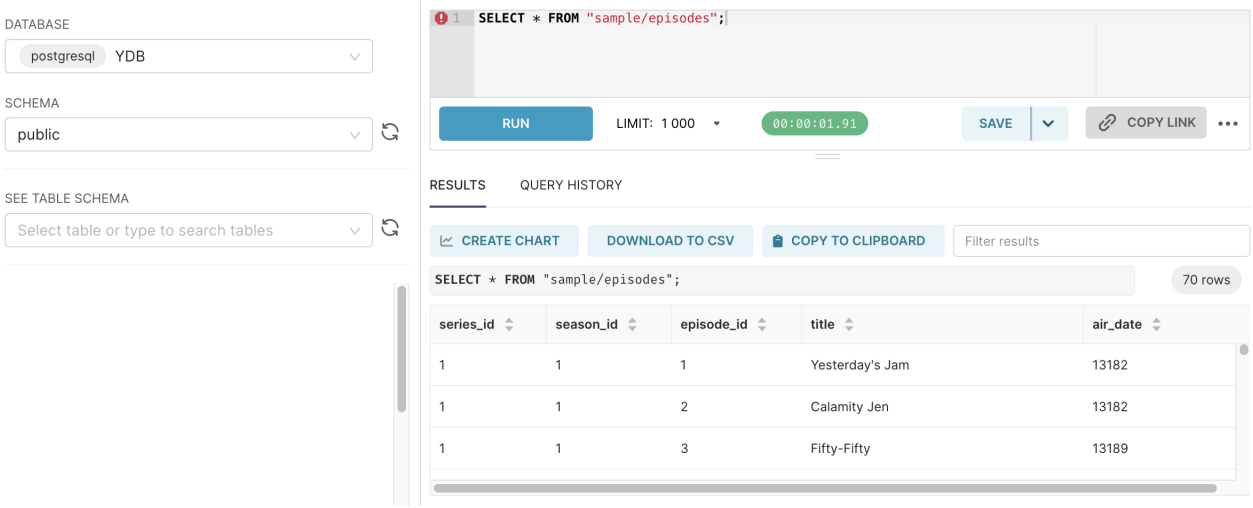
1. In the Apache Superset toolbar, hover over the **+** button and select **SQL query**.
2. In the **DATABASE** drop-down list, select the YDB database connection.
3. In the **SCHEMA** drop-down list, select `public`.

> **Alert**
>
> YDB currently does not provide table schema information via the PostgreSQL protocol. You can skip selecting a table in the **SEE TABLE SCHEMA** drop-down list.

4. 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:

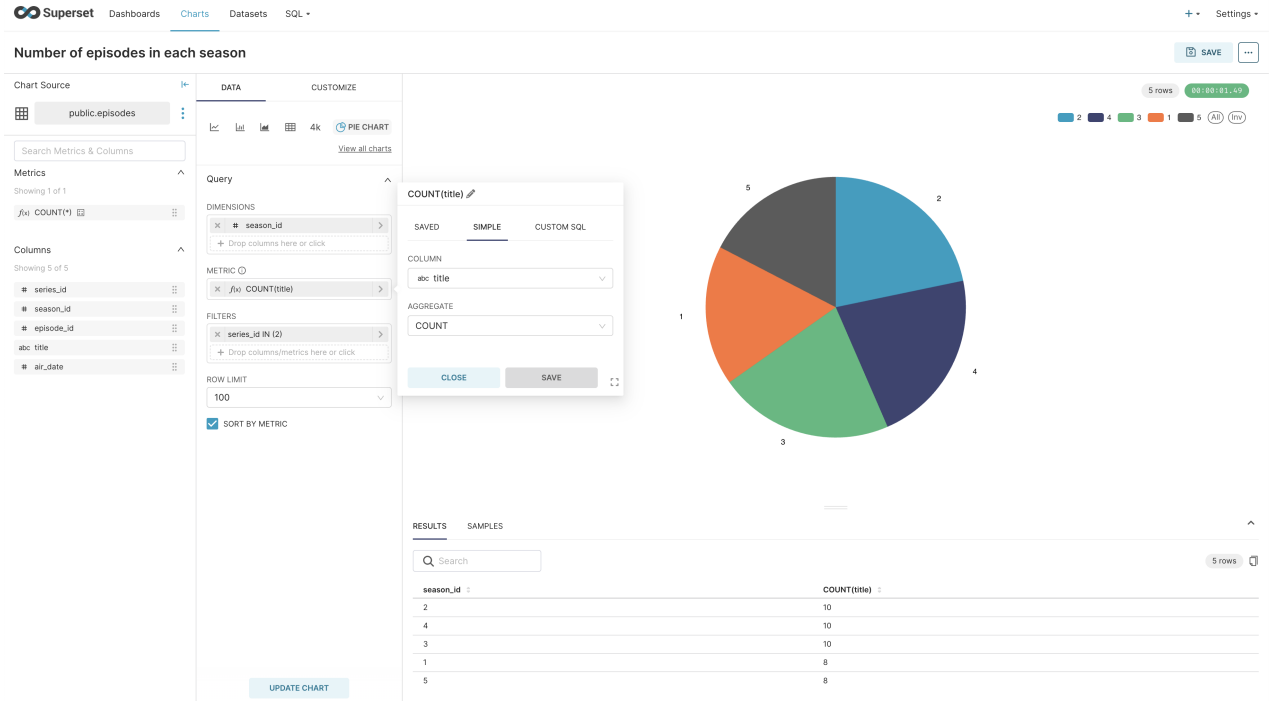
- series_id
- season_id
- episode_id
- title
- air_date

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.

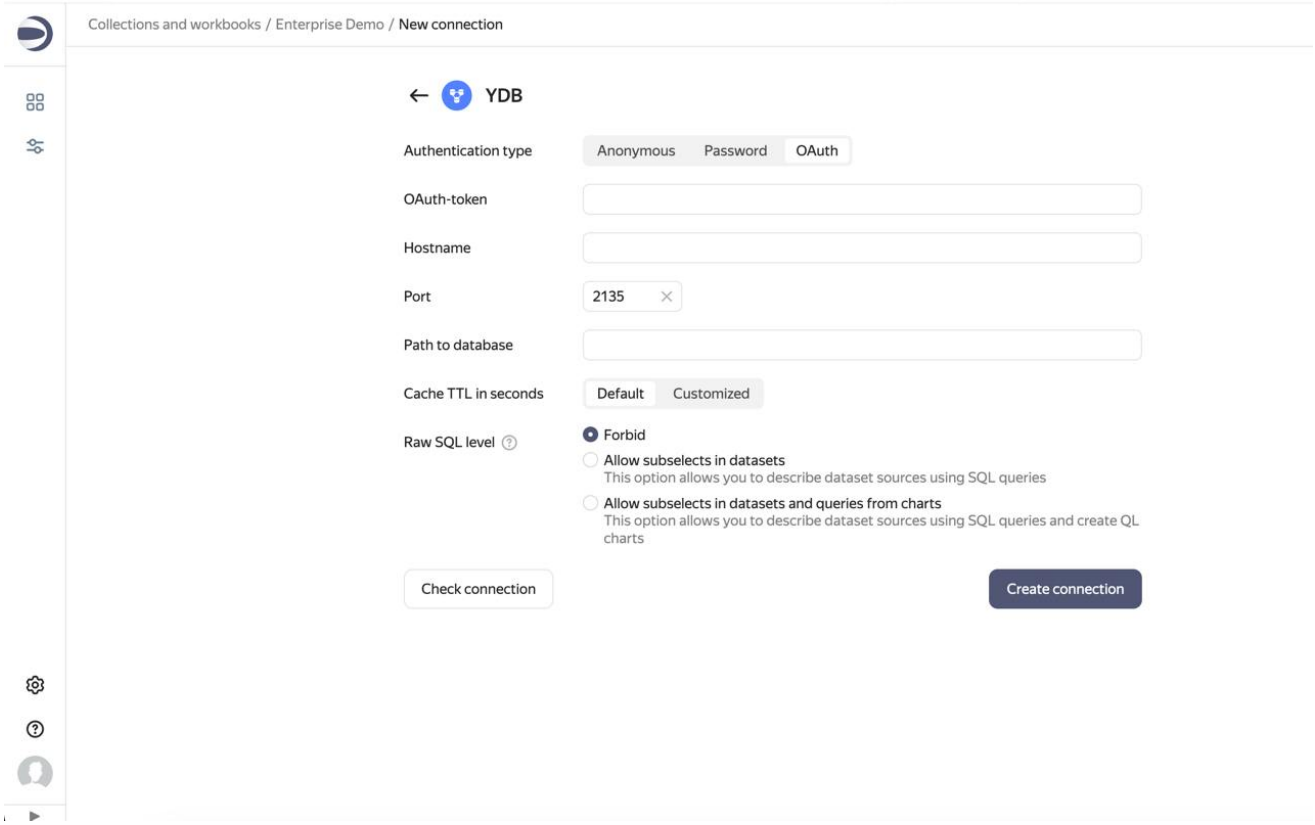
   Password

   - **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


Collections and workbooks / Enterprise Demo / New connection
YDB
Authentication type
Anonymous
Password
OAuth
OAuth-token
Hostname
Port
2135
Path to database
Cache TTL in seconds
Default
Customized
Raw SQL level
Forbid
Allow subselects in datasets
This option allows you to describe dataset sources using SQL queries
Allow subselects in datasets and queries from charts
This option allows you to describe dataset sources using SQL queries and create QL charts
Check connection
Create connection

FineBI

FineBI is a powerful big data analytics tool. FineBI allows organizations to analyze and share data for informed decision-making. It helps transform raw data into insightful visualizations, track KPIs, identify trends, and predict future outcomes.

PostgreSQL compatibility mode in YDB enables the use of FineBI to query and visualize data from YDB. In this case FineBI works with YDB just like with PostgreSQL.

> **Warning**
>
> At the moment, YDB's compatibility with PostgreSQL **is under development**, so not all PostgreSQL constructs and functions are supported yet. PostgreSQL compatibility is available for testing in the form of a Docker container, which can be deployed by following these instructions.

Prerequisites

Before you begin, make sure that the following software is installed:

- FineBI.
- PostgreSQL JDBC driver uploaded to FineBI.

> **Note**
>
> You can download the latest PostgreSQL JDBC driver from the Download page of the PostgreSQL web site. For information on how to upload the PostgreSQL JDBC driver to FineBI, refer to the FineBI documentation.

Adding a database connection to YDB

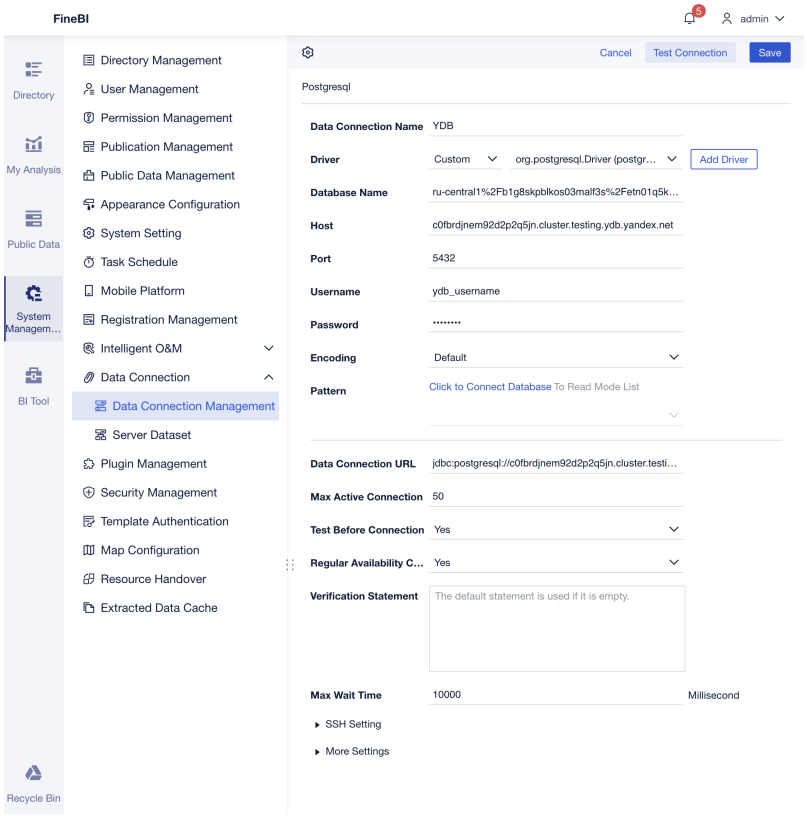
To connect to YDB from FineBI using the PostgreSQL wire protocol, follow these steps:

1. Log in to FineBI as the `admin` user.
2. Navigate to **System Management** > **Data Connection** > **Data Connection Management**.
3. Click the **New Data Connection** button.
4. In the **Search** field, type `postgresql` to find the PostgreSQL icon.
5. Click the PostgreSQL icon.
6. Enter the YDB credentials in the corresponding fields:
   - **Data Connection Name**. The YDB connection name in FineBI.
   - **Driver**. The driver that FineBI uses to connect to YDB.

     Select `Custom` and the installed JDBC driver `org.postgresql.Driver`.
   - **Database Name**. The path to the database in the YDB cluster where queries will be executed.

     > **Alert**
     >
     > Special characters in the path string must be URL encoded. For example, ensure that you replace slash (`/`) characters with `%2F`.

   - **Host**. The endpoint of the YDB cluster to which the connection will be made.
   - **Port**. The port of the YDB endpoint.
   - **Username**. The login for connecting to the YDB database.
   - **Password**. The password for connecting to the YDB database.



7. Click **Test Connection**.

   If the connection details are correct, a message confirming a successful connection will appear.
8. To save the database connection, click **Save**.

   A new database connection will appear in the **Data Connection** list.

Adding an SQL Dataset

To create a dataset for a YDB table, follow these steps:

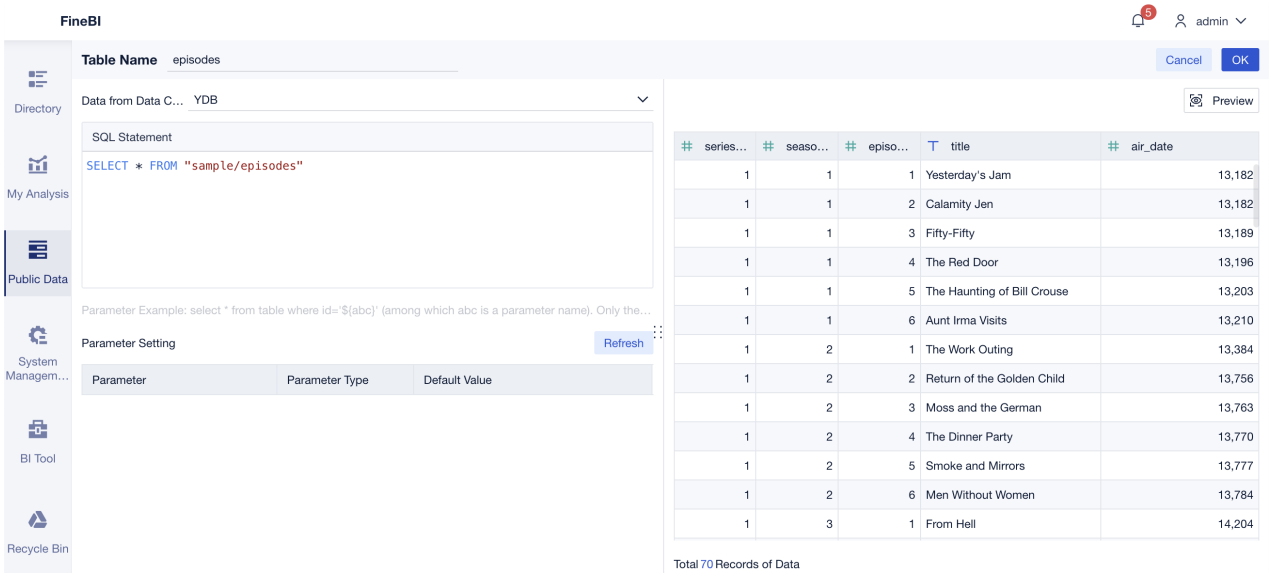
1. In FineBI, open the **Public Data** tab.
2. Select a folder, to which you want to add a dataset.

> **Warning**
>
> You must have the Public Data Management permission for the selected folder in FineBI.

3. Click **Add Dataset** and select **SQL Dataset** from the drop-down list.
4. In the **Table Name** field, enter a name for the dataset.
5. In the **Data from Data Connection** drop-down list, select the YDB connection you created.
6. In the **SQL Statement** field, enter the SQL query to retrieve the necessary columns from a YDB table. For example, `SELECT * FROM <ydb_table_name>` for all columns.

> **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>";
> ```

7. To test the SQL query, click **Preview**. If the query is correct, the table data will appear in the preview pane.



8. To save the dataset, click **OK**.

After creating datasets, you can use data from YDB to create charts in FineBI. For more information, refer to the FineBI documentation.

Creating a chart

Let's create a sample chart using the dataset from the `episodes` table, as described in the YQL tutorial. Among other things, this tutorial covers how to create this table and populate it with data. It will be a pie chart that demonstrates how many episodes each season of a given series contains.

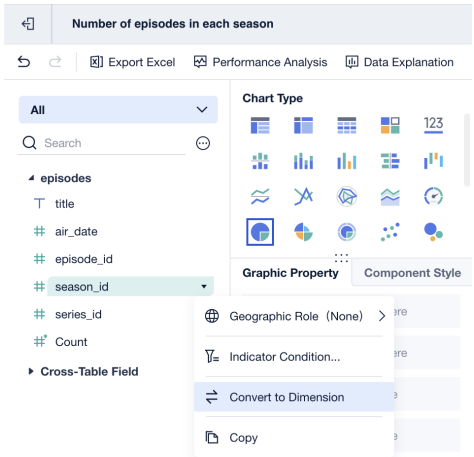
The table contains the following columns:

- `series_id`
- `season_id`
- `episode_id`
- `title`
- `air_date`

To create a chart, follow these steps:

1. In FineBI, open the **My Analysis** tab.
2. Click **New Subject**.

   The **Select Data** dialog box will appear.
3. In the **Select Data** dialog box, navigate to the dataset for the `episodes` table and click **OK**.
4. Click the **Component** tab at the bottom of the page.
5. In the **Chart Type** pane, click the **Pie Chart** icon.
6. In the list of columns in the `episodes` dataset, click the arrow next to the `episode_id` column and select **Convert to Dimension** from the drop-down list.



7. Drag the `season_id` column to the **Color** field.
8. Drag the `title` column to the **Label** field.
9. Drag the `series_id` column to the **Filter** field.

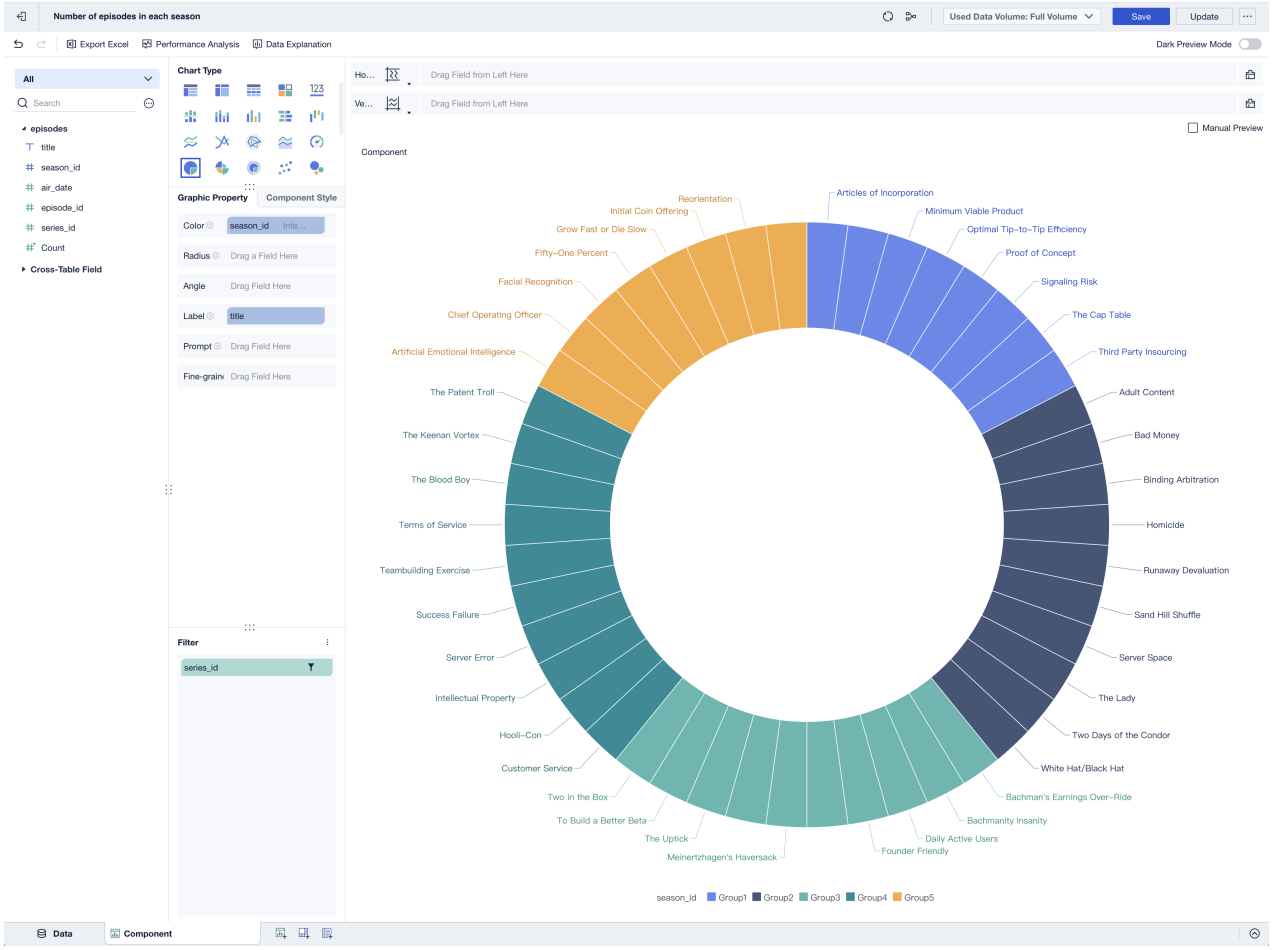
   The **Add Filter to episodes.series_id** dialog box will appear.
10. In the **Add Filter to episodes.series_id** dialog box, select `Detailed Value` and click **Next Step**.

11. Specify the following condition:

    `series_id` `Equal To` `Fixed Value` `2`

12. Click **OK**.

    The diagram will display data only for the series that has the ID of `2`.



13. Click **Save**.

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)

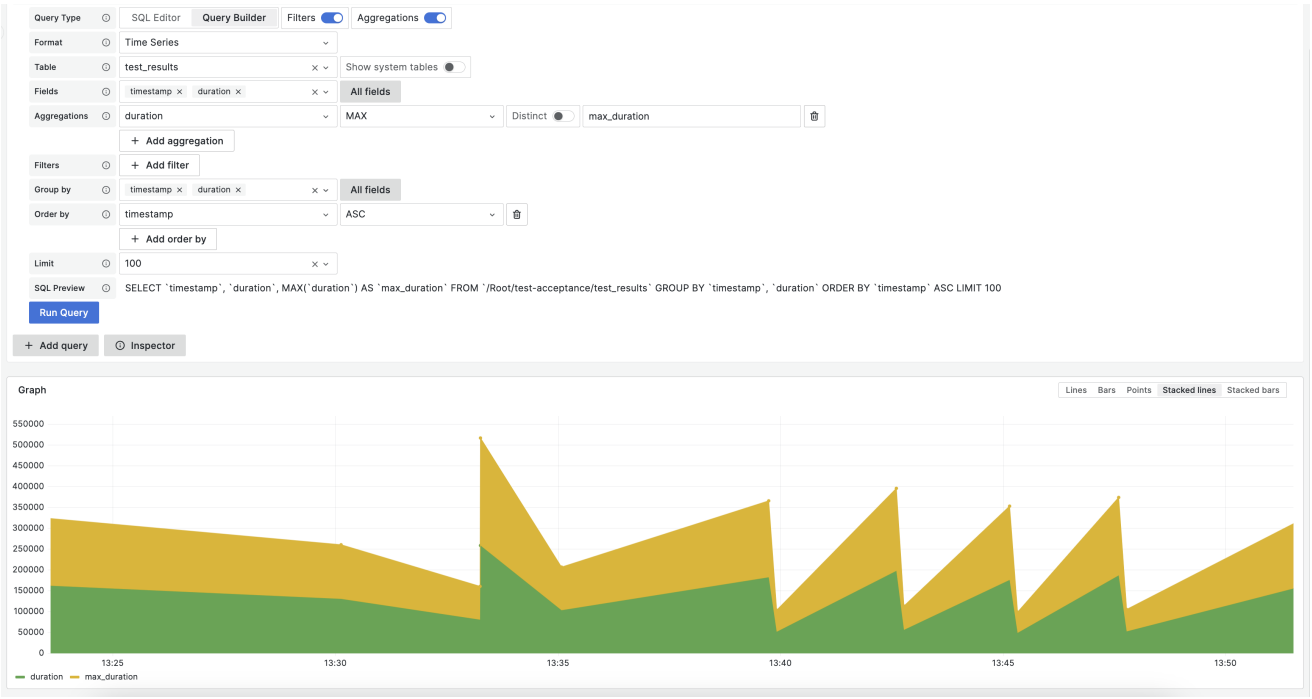
accessToken	Access token	`string` (secured)
password	User password	`string` (secured)
certificate	If self-signed certificates are used on your YDB cluster nodes, specify the Certificate Authority certificate used to issue them	`string` (secured)

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.

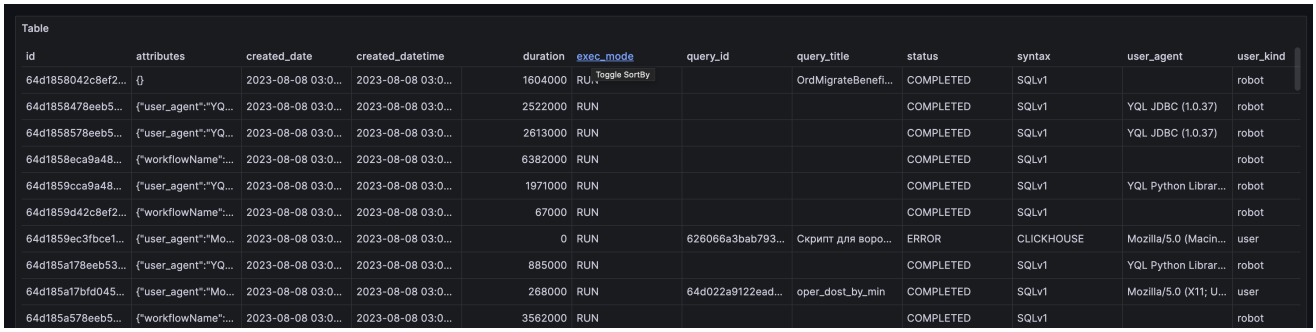
For example:

```
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.



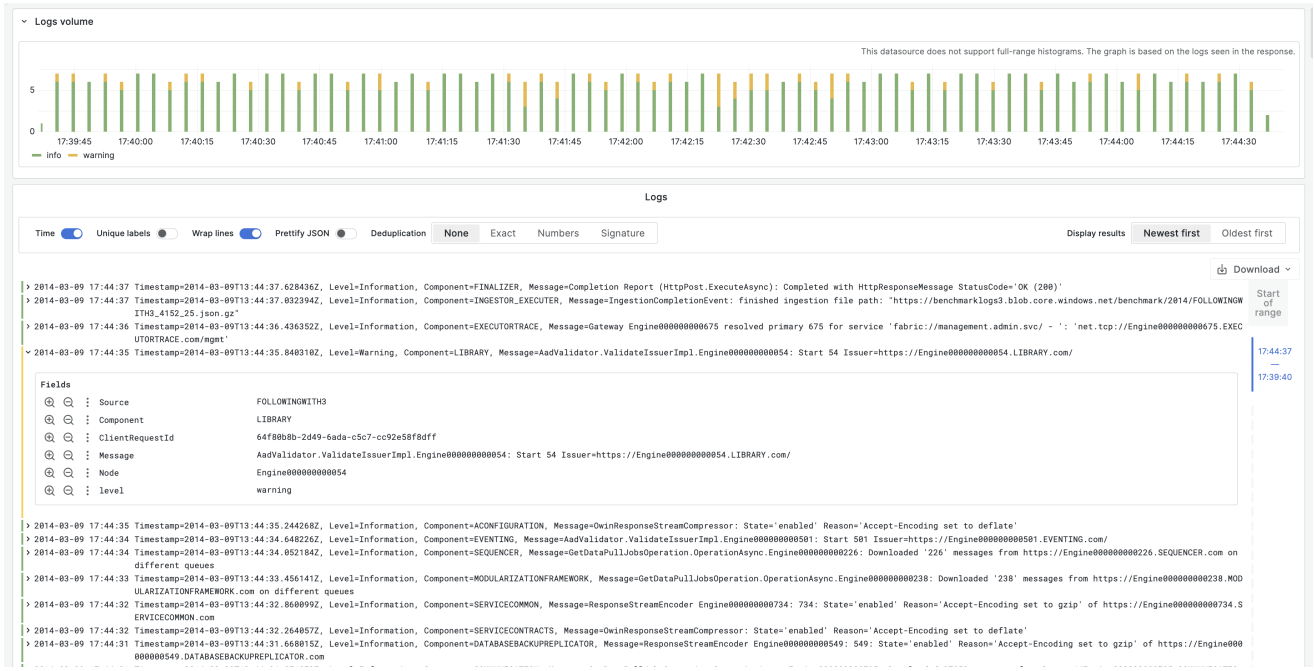
Table

id	attributes	created_date	created_datetime	duration	exec_mode	query_id	query_title	status	syntax	user_agent	user_kind
64d1858042c8ef2...	{}	2023-08-08 03:0...	2023-08-08 03:0...	1604000	RU...		OrdMigrateBenefi...	COMPLETED	SQLv1		robot
64d1858478eeb5...	{"user_agent":"YQ...	2023-08-08 03:0...	2023-08-08 03:0...	2522000	RUN			COMPLETED	SQLv1	YQL JDBC (1.0.37)	robot
64d1858578eeb5...	{"user_agent":"YQ...	2023-08-08 03:0...	2023-08-08 03:0...	2613000	RUN			COMPLETED	SQLv1	YQL JDBC (1.0.37)	robot
64d1858eca9a48...	{"workflowName":...	2023-08-08 03:0...	2023-08-08 03:0...	6382000	RUN			COMPLETED	SQLv1		robot
64d1859cca9a48...	{"user_agent":"YQ...	2023-08-08 03:0...	2023-08-08 03:0...	1971000	RUN			COMPLETED	SQLv1	YQL Python Librar...	robot
64d1859d42c8ef2...	{"workflowName":...	2023-08-08 03:0...	2023-08-08 03:0...	67000	RUN			COMPLETED	SQLv1		robot
64d1859ec3fbce1...	{"user_agent":"Mo...	2023-08-08 03:0...	2023-08-08 03:0...	0	RUN	626066a3bab793...	Скрипт для воро...	ERROR	CLICKHOUSE	Mozilla/5.0 (Macin...	user
64d185a178eeb53...	{"user_agent":"YQ...	2023-08-08 03:0...	2023-08-08 03:0...	885000	RUN			COMPLETED	SQLv1	YQL Python Librar...	robot
64d185a17bfd045...	{"user_agent":"Mo...	2023-08-08 03:0...	2023-08-08 03:0...	268000	RUN	64d022a9122ead...	oper_dost_by_min	COMPLETED	SQLv1	Mozilla/5.0 (X11; U...	user
64d185a578eeb5...	{"workflowName":...	2023-08-08 03:0...	2023-08-08 03:0...	3562000	RUN			COMPLETED	SQLv1		robot

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, $templateVar)`	Replaced by the first parameter when the template variable in the second parameter is not provided.	`$__varFallback('foo', $bar)` `foo` if variable `bar` is not provided, or `$bar`'s value

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.

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;")

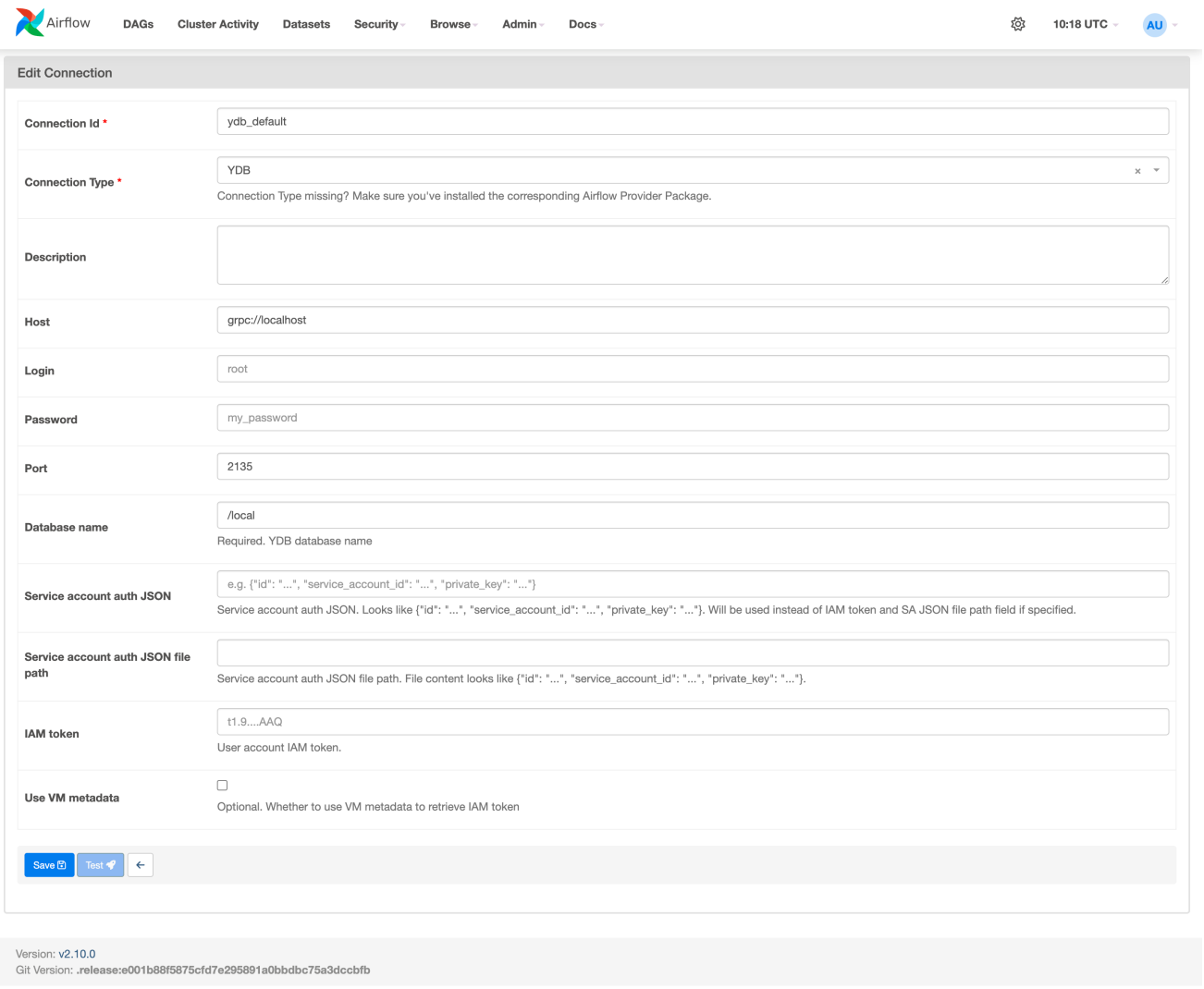
# Extract the result and column names
result = cursor.fetchall()
columns = [desc[0] for desc in cursor.description]

# 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="test_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
    )
```

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@`
`CredentialsYcServiceAccountKeyFile`	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):

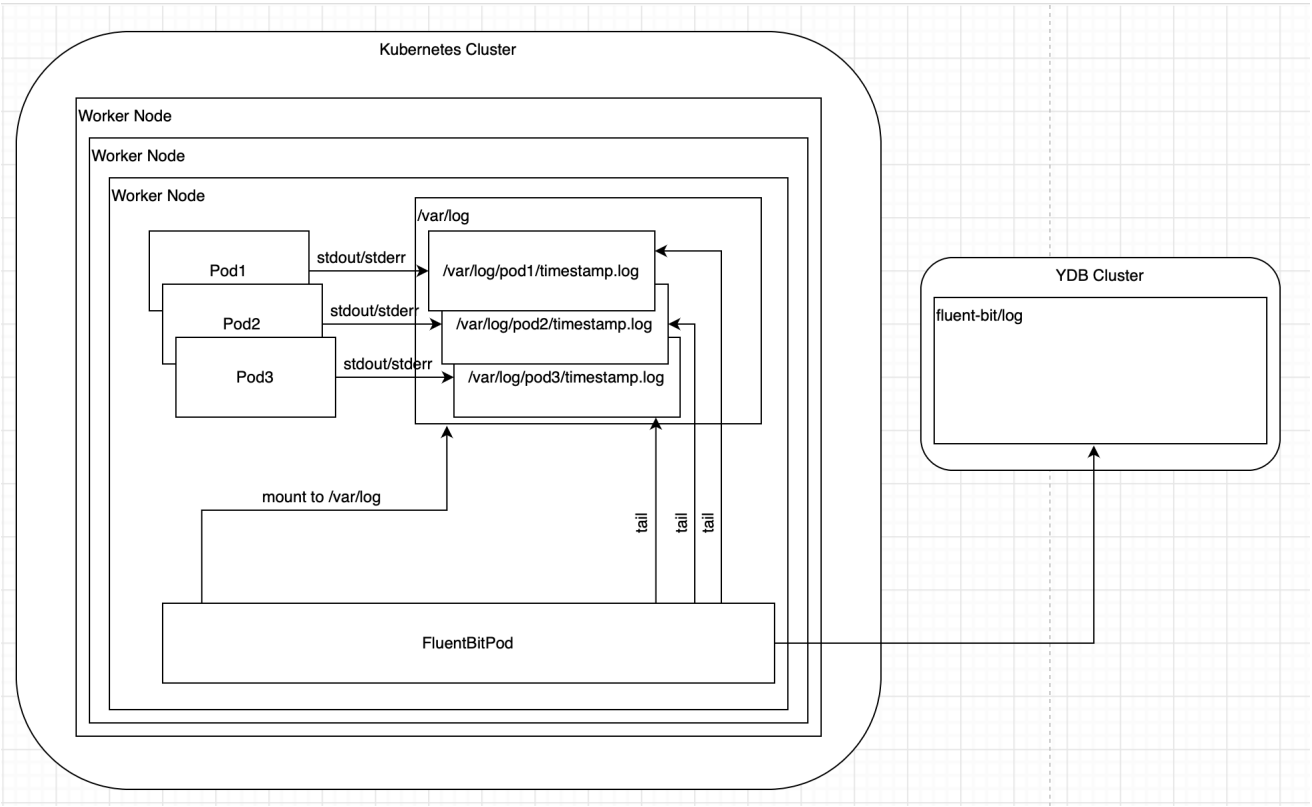
- `.timestamp` - log record timestamp (mandatory)
- `.input` - log input stream name (mandatory)
- `.hash` - uint64 hash code, computed over the log record fields (optional)
- `.other` - JSON document containing all log record fields that were not explicitly mapped to any table column (optional)

Example of `Columns` parameter value:

```
{".timestamp": "timestamp", ".input": "input", ".hash": "datahash", "log": "message", "level": "level", "host":
"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":"message_structured","strea
        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", "level":
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, execute the following query using ScanQuery mode:

```
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        │ "74cd4048-0b61-4fb9-9385-308714e21881" │
│  1    │ null         │ "2024-05-22T13:15:56.591000Z" │ "test1"     │ "1df27d0a-9aa0-42c7-9ea2-ab69bc1f5d87" │
│  4    │ "demo message" │ "2024-05-22T13:15:38.760000Z" │ "demo_user" │ "b7468cb1-e1e3-46fa-965d-83e604e80a31" │
└─────────┴──────────────┴──────────────────────────────┴─────────────┴──────────────────────────────────────┘
```

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_topic
```

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_topic
ydb -e grpc://localhost:2136 -d /local topic read /local/logstash_demo_topic --consumer logstash-consumer --commit
true
```

The latter command will return the message content:

```
{"level":4,"message":"demo message","timestamp":1716470292640,"user":"demo_user"}
```

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=AdventureWorks2022;
            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/etn63999hrinbapmef6g
            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>
```

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

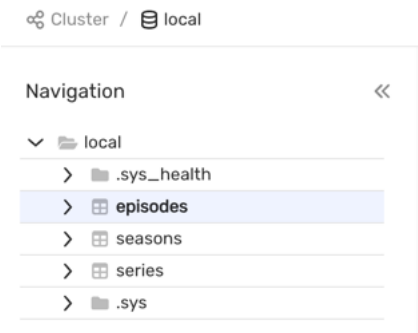
Flyway Desktop is currently not supported.

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

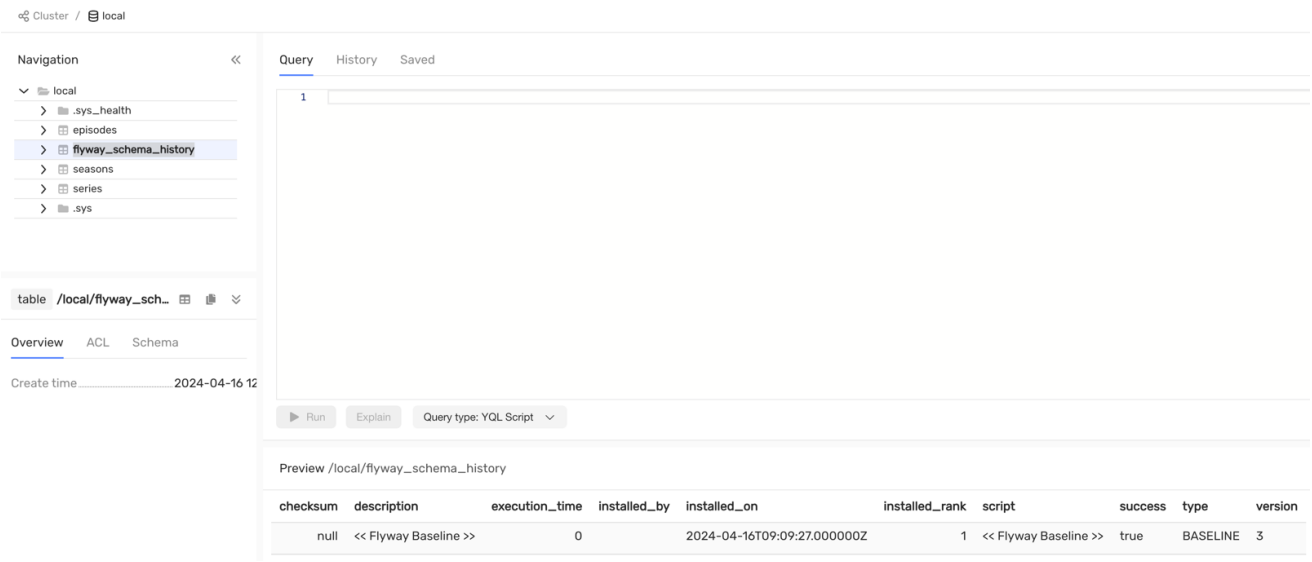
```
CREATE TABLE episodes
(
    series_id Uint64,
    season_id Uint64,
    episode_id Uint64,
    title Utf8,
    air_date Uint64,
    PRIMARY KEY (series_id, season_id, episode_id)
);
```

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 Dave
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)),
```

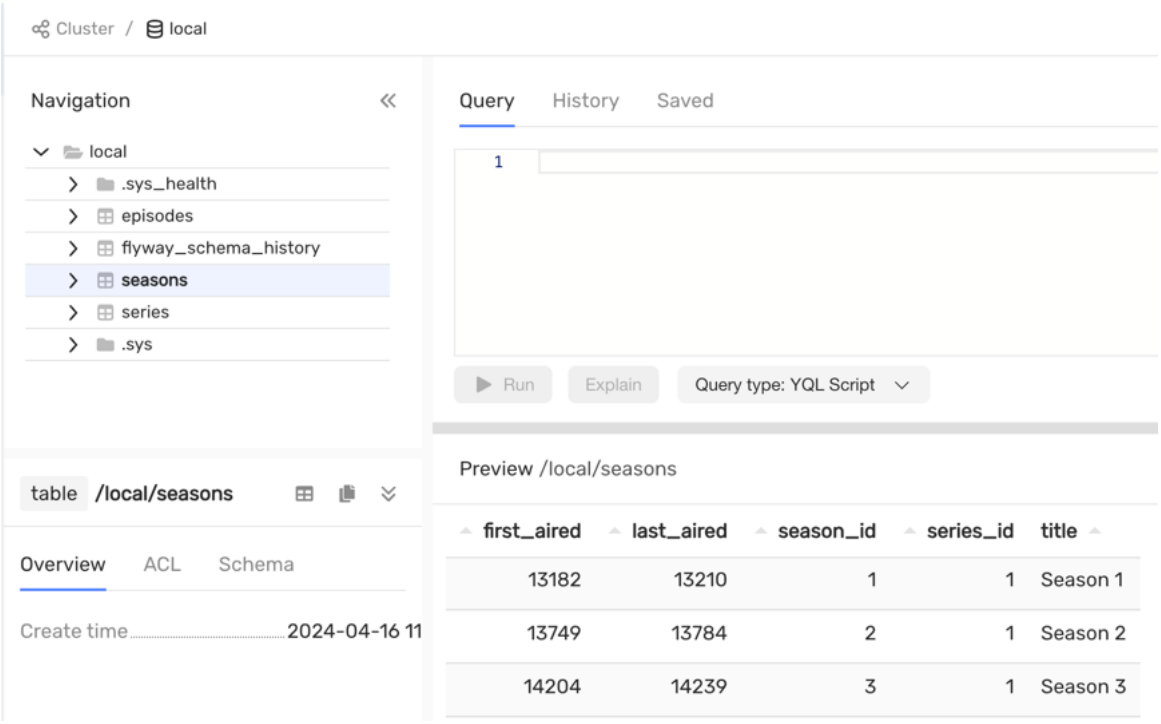
```
       (2, 2, "Season 2", CAST(Date ("2015-04-12") AS Uint64), CAST(Date ("2015-06-14") AS Uint64)),
       (2, 3, "Season 3", CAST(Date ("2016-04-24") AS Uint64), CAST(Date ("2016-06-26") AS Uint64)),
       (2, 4, "Season 4", CAST(Date ("2017-04-23") AS Uint64), CAST(Date ("2017-06-25") AS Uint64)),
       (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
```

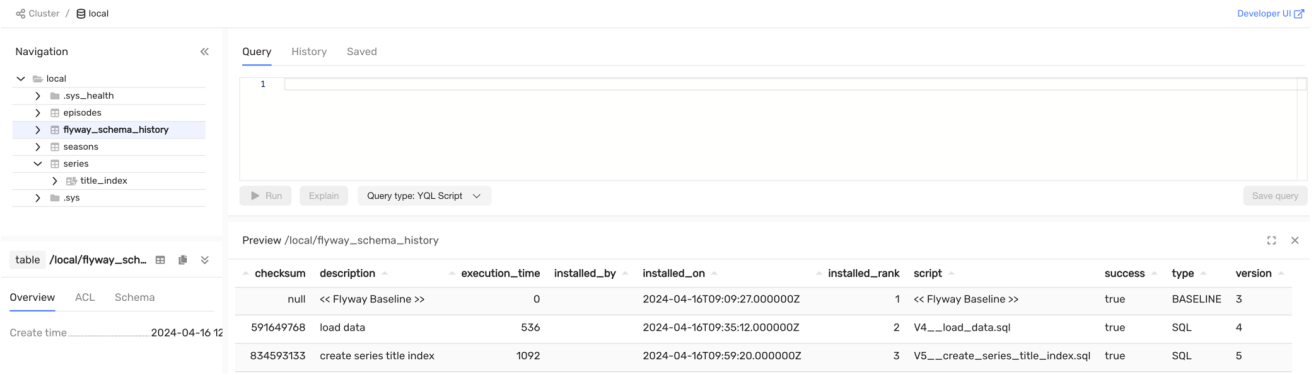
The contents of `V5__create_series_title_index.sql`

```
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              | Undoable
|
+-----------+---------+---------------------------+----------+---------------------+--------------------+---------
-+
| 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
```

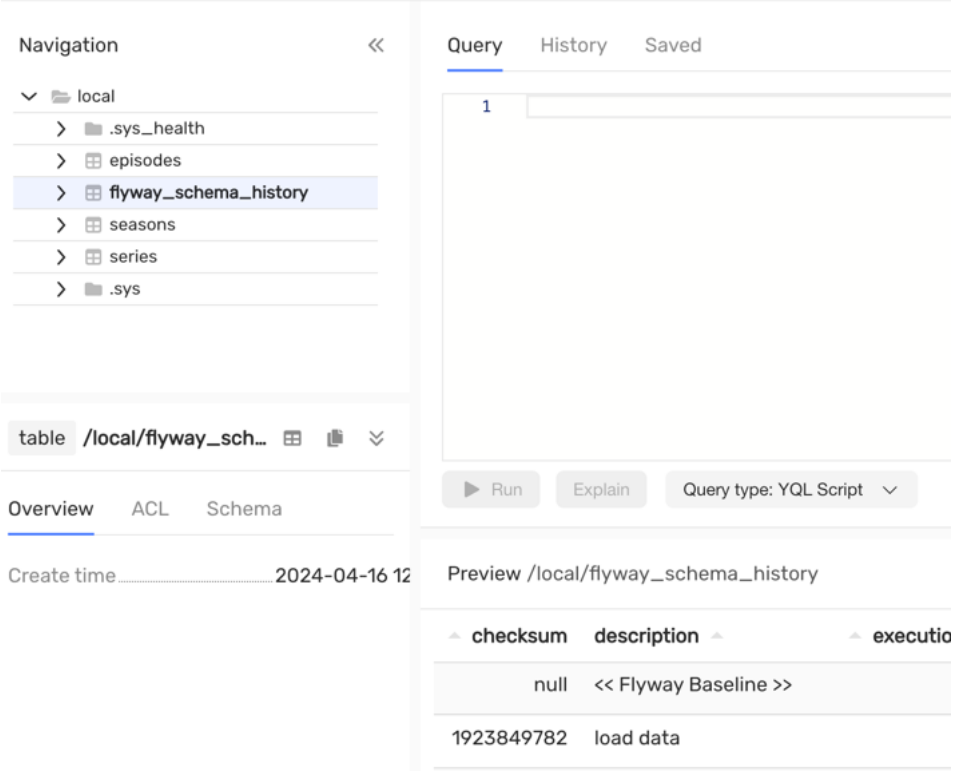
repair

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=declare,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_query_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
```

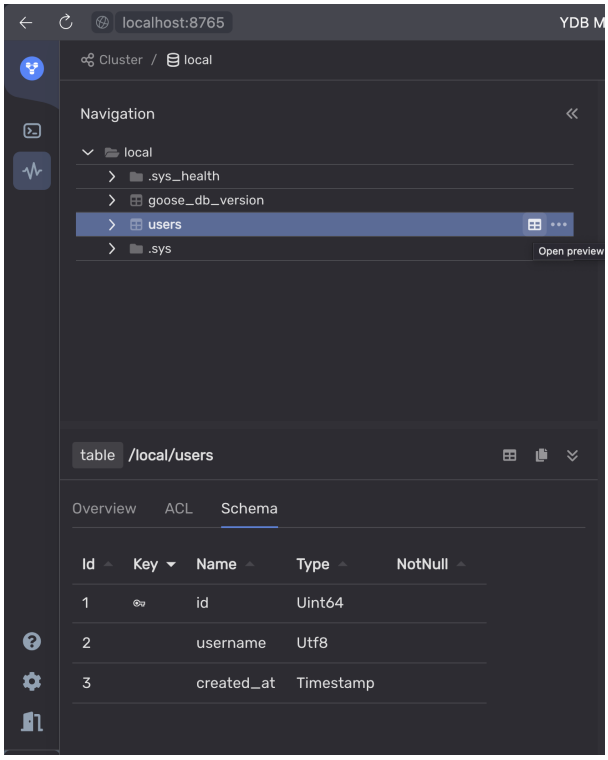
Let's check the status of migration through `goose status`:

```
$ 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)
```

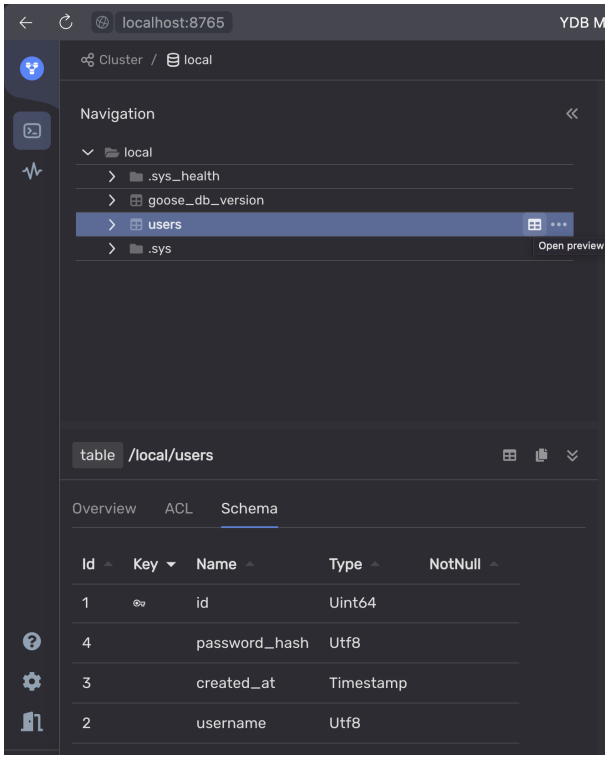
Let's check the migration status through `goose status`:

```
$ 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`:

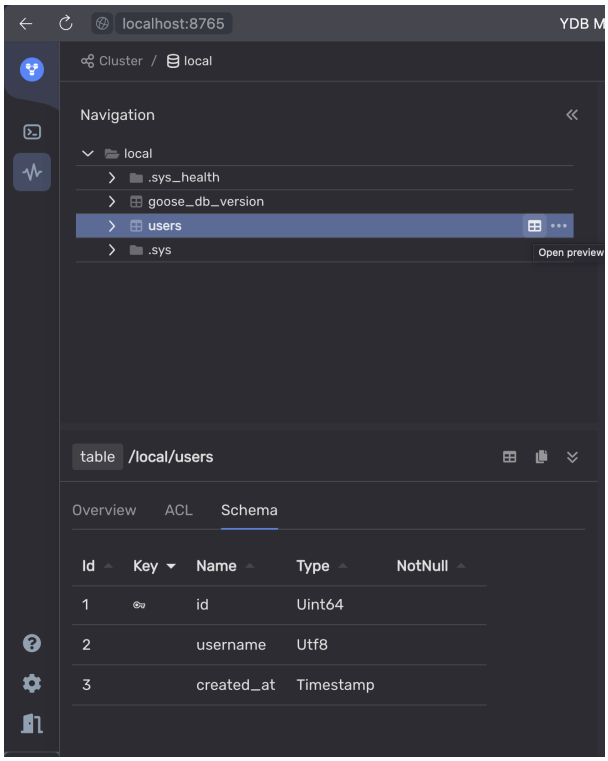
```
$ goose ydb $YDB_CONNECTION_STRING status
2024/01/12 13:07:36     Applied At                  Migration
2024/01/12 13:07:36     =======================================
2024/01/12 13:07:36     Fri Jan 12 11:55:18 2024 -- 20240112115229_00001_create_first_table.sql
```

```
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" unique="false">
    <column name="title"/>
</createIndex>
```

json

```
"changes": [
    {
      "createIndex": {
        "tableName": "episodes",
        "indexName": "episodes_index",
        "unique": "false",
        "columns": {
          "column": {
            "name": "title"
```

```
          }
        }
      }
    }
```

yaml

```
changes:
- createIndex:
    tableName: episodes
    indexName: episodes_index
    unique: false
    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,jsondocument,date,da
2,true,123123,13000,112,1.123,1.123123,1.123123,12,13,14,15,kurdyukov-kir,binary,{"asd": "asd"},{"asd":
"asd"},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-
dialect/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
<?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>

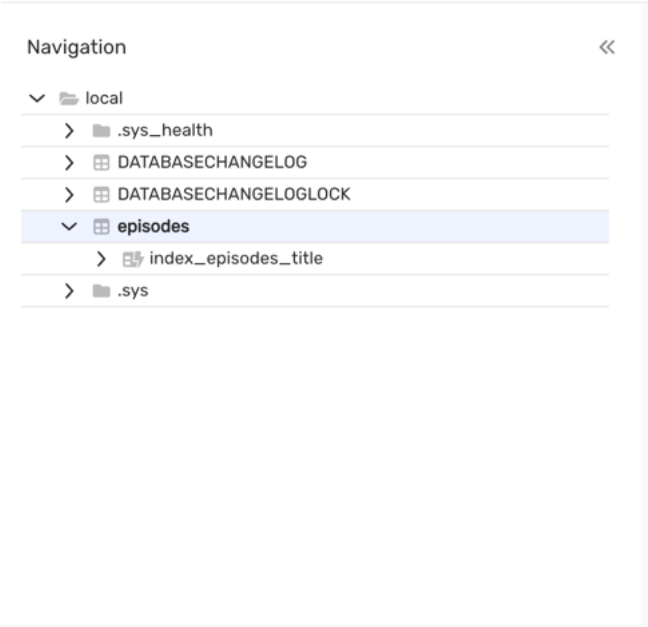
            <column name="title" type="text"/>
            <column name="air_date" type="timestamp"/>
        </createTable>
        <rollback>
            <dropTable tableName="episodes"/>
        </rollback>
    </changeSet>
    <changeSet id="index_episodes_title" author="kurdyukov-kir">
        <createIndex tableName="episodes" indexName="index_episodes_title" unique="false">
            <column name="title"/>
        </createIndex>
    </changeSet>
</databaseChangeLog>
```

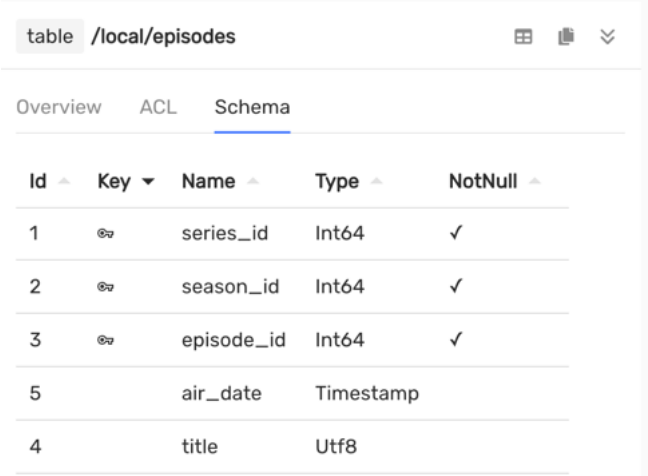
After executing the `liquibase update` command, Liquibase will print the following log:

```
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	ID	LABELS	LIQUIBASE	MD5SUM	ORDEREXECUTED	TAG
kurdyukov-kir	Table episodes.		12:53:27	1544007500	createTable tableName=episodes	EXECUTED	migration/episodes.xml	episodes		4.25.1	9:4067056a5ab61db09b379a936258 70ca	1	
kurdyukov-kir	""		12:53:28	1544007500	createIndex indexName=index_episodes_title, tableName=episodes	EXECUTED	migration/episodes.xml	index_episodes_title		4.25.1	9:49b8b0b22df18c7fd90a3d6b2c561455d	2	

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"/>
```

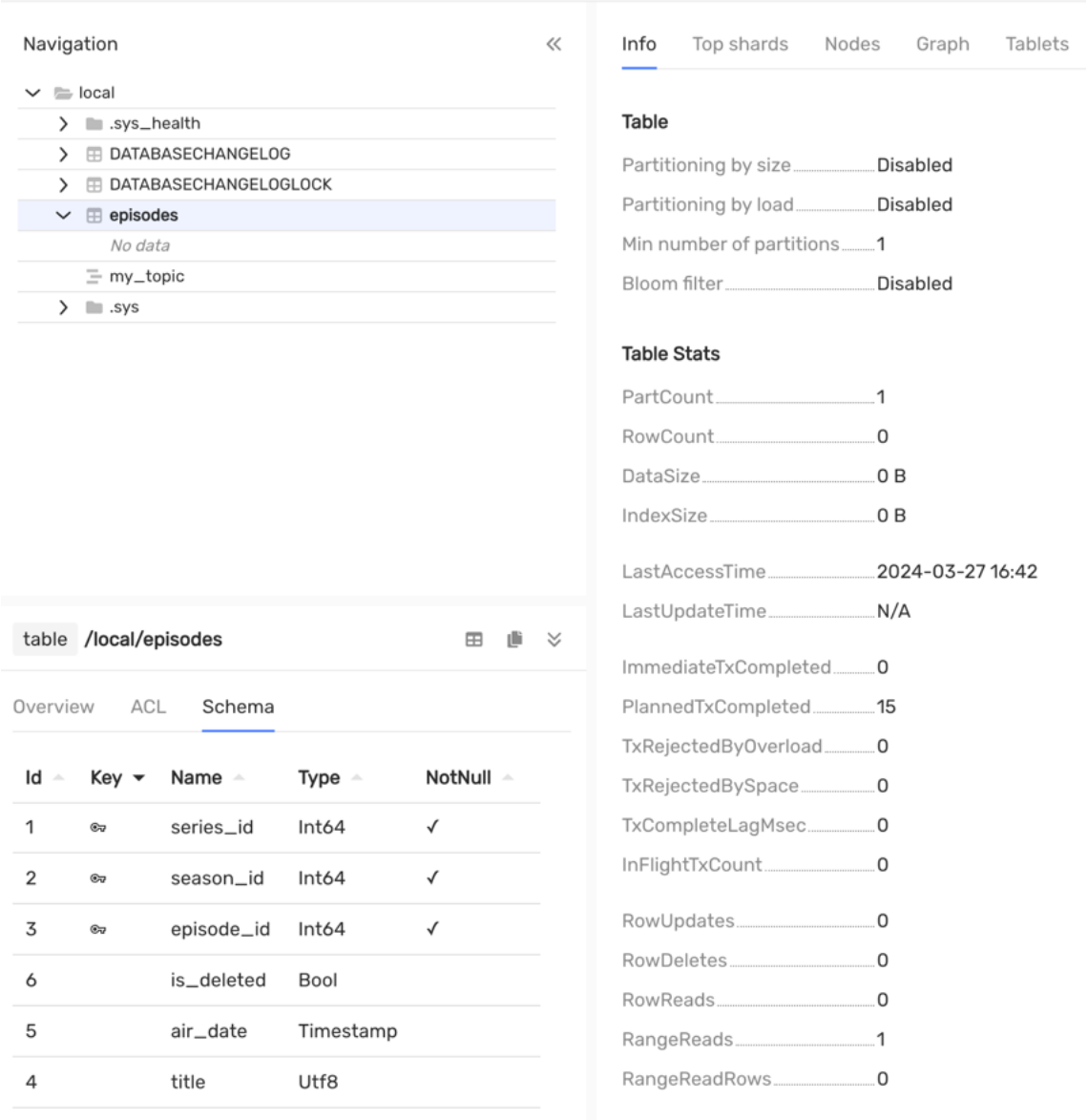
```
    <addColumn tableName="episodes">
        <column name="is_deleted" type="bool"/>
    </addColumn>
</changeSet>
<include file="/migration/sql/yql.sql" relativeToChangelogFile="true"/>
```

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:

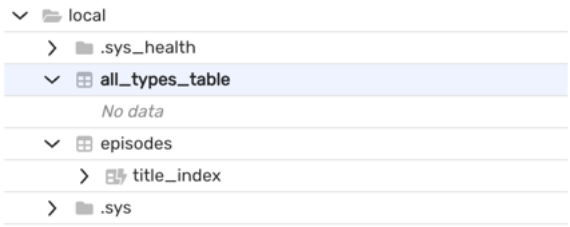


table /local/all_types_table

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:

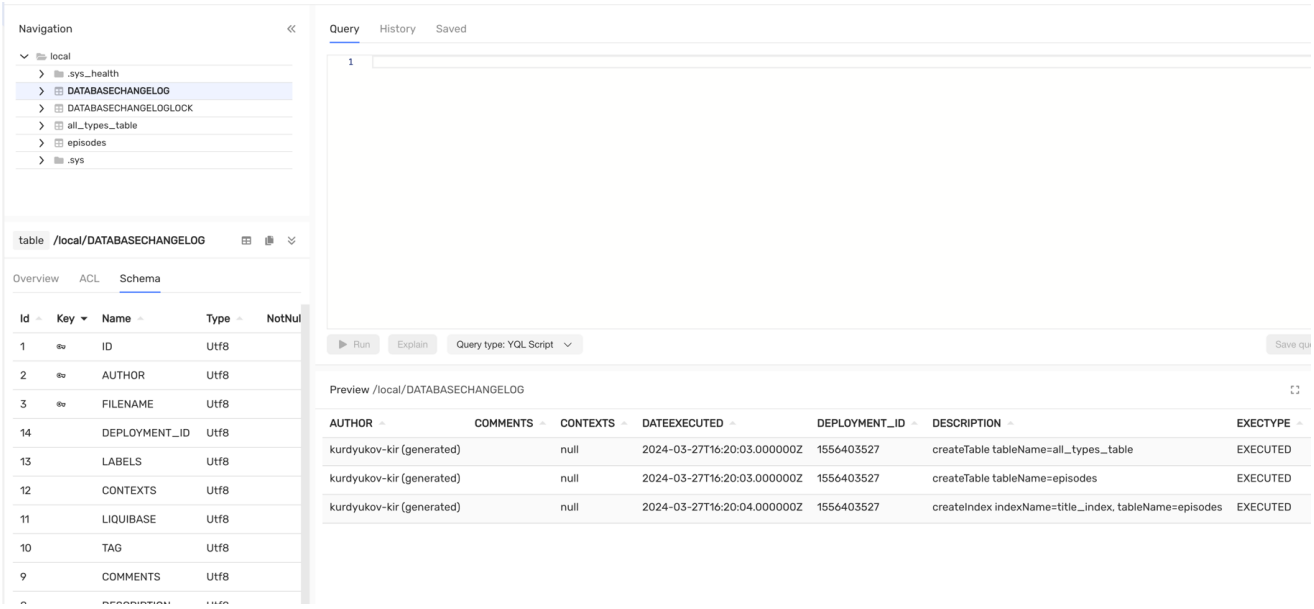
```
<changeSet author="kurdyukov-kir (generated)" id="1711556283305-1">
    <createTable tableName="all_types_table">
        <column name="id" type="INT32">
            <constraints nullable="false" primaryKey="true"/>
        </column>
        <column name="bool_column" type="BOOL"/>
        <column 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.

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.

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

```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

```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.

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

```xml
<!-- 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`.

LangChain

Integration of YDB with langchain enables the use of YDB as a vector store for RAG applications.

This integration allows developers to efficiently manage, query, and retrieve vectorized data, which is fundamental for modern applications involving natural language processing, search, and data analysis. By leveraging embedding models, users can create sophisticated systems that understand and retrieve information based on semantic similarity.

Setup

To use this integration, install the following software:

- `langchain-ydb`

  To install `langchain-ydb`, run the following command:

  ```
  pip install -qU langchain-ydb
  ```

- embedding model

  This tutorial uses `HuggingFaceEmbeddings`. To install this package, run the following command:

  ```
  pip install -qU langchain-huggingface
  ```

- Local YDB

  For more information, see Install and start YDB.

Initialization

Creating a YDB vector store requires specifying an embedding model. In this instance, `HuggingFaceEmbeddings` is used:

```
from langchain_huggingface import HuggingFaceEmbeddings

embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-mpnet-base-v2")
```

Once the embedding model is created, the YDB vector store can be initiated:

```
from langchain_ydb.vectorstores import YDB, YDBSearchStrategy, YDBSettings

settings = YDBSettings(
    host="localhost",
    port=2136,
    database="/local",
    table="ydb_example",
    strategy=YDBSearchStrategy.COSINE_SIMILARITY,
)
vector_store = YDB(embeddings, config=settings)
```

Manage Vector Store

After the vector store has been established, you can start adding and removing items from the store.

Add items to vector store

The following code prepares the documents:

```
from uuid import uuid4

from langchain_core.documents import Document

document_1 = Document(
    page_content="I had chocalate chip pancakes and scrambled eggs for breakfast this morning.",
    metadata={"source": "tweet"},
)

document_2 = Document(
    page_content="The weather forecast for tomorrow is cloudy and overcast, with a high of 62 degrees.",
    metadata={"source": "news"},
)

document_3 = Document(
```

```
    page_content="Building an exciting new project with LangChain - come check it out!",
    metadata={"source": "tweet"},
)

document_4 = Document(
    page_content="Robbers broke into the city bank and stole $1 million in cash.",
    metadata={"source": "news"},
)

document_5 = Document(
    page_content="Wow! That was an amazing movie. I can't wait to see it again.",
    metadata={"source": "tweet"},
)

document_6 = Document(
    page_content="Is the new iPhone worth the price? Read this review to find out.",
    metadata={"source": "website"},
)

document_7 = Document(
    page_content="The top 10 soccer players in the world right now.",
    metadata={"source": "website"},
)

document_8 = Document(
    page_content="LangGraph is the best framework for building stateful, agentic applications!",
    metadata={"source": "tweet"},
)

document_9 = Document(
    page_content="The stock market is down 500 points today due to fears of a recession.",
    metadata={"source": "news"},
)

document_10 = Document(
    page_content="I have a bad feeling I am going to get deleted :(",
    metadata={"source": "tweet"},
)

documents = [
    document_1,
    document_2,
    document_3,
    document_4,
    document_5,
    document_6,
    document_7,
    document_8,
    document_9,
    document_10,
]
uuids = [str(uuid4()) for _ in range(len(documents))]
```

Items are added to the vector store using the `add_documents` function.

```
vector_store.add_documents(documents=documents, ids=uuids)
```

Output:

```
Inserting data...: 100%|██████████| 10/10 [00:00<00:00, 14.67it/s]
['947be6aa-d489-44c5-910e-62e4d58d2ffb',
 '7a62904d-9db3-412b-83b6-f01b34dd7de3',
 'e5a49c64-c985-4ed7-ac58-5ffa31ade699',
 '99cf4104-36ab-4bd5-b0da-e210d260e512',
 '5810bcd0-b46e-443e-a663-e888c9e028d1',
 '190c193d-844e-4dbb-9a4b-b8f5f16cfae6',
 'f8912944-f80a-4178-954e-4595bf59e341',
 '34fc7b09-6000-42c9-95f7-7d49f430b904',
 '0f6b6783-f300-4a4d-bb04-8025c4dfd409',
 '46c37ba9-7cf2-4ac8-9bd1-d84e2cb1155c']
```

Delete items from vector store

To delete items from the vector store by ID, use the `delete` function:

```
vector_store.delete(ids=[uuids[-1]])
```

Output:

```
True
```

Query Vector Store

After establishing the vector store and adding relevant documents, you can query the store during chain or agent execution.

Query directly

Similarity search

A simple similarity search can be performed as follows:

```
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}]")
```

Output:

```
* 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

To perform a similarity search with score, use the following code:

```
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}]")
```

Output:

```
* [SIM=0.595] The weather forecast for tomorrow is cloudy and overcast, with a high of 62 degrees. [{'source':
'news'}]
* [SIM=0.212] I had chocalate chip pancakes and scrambled eggs for breakfast this morning. [{'source': 'tweet'}]
* [SIM=0.118] Wow! That was an amazing movie. I can't wait to see it again. [{'source': 'tweet'}]
```

Filtering

Searching with filters is performed as described below:

```
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}]")
```

Output:

```
* I had chocalate 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 by turning into retriever

The vector store can also be transformed into a retriever for easier use in chains.

Here's how to transform the vector store into a retriever and invoke it with a simple query and filter.

```
retriever = vector_store.as_retriever(
    search_kwargs={"k": 2},
)
results = retriever.invoke(
    "Stealing from the bank is a crime", filter={"source": "news"}
)
for res in results:
    print(f"* {res.page_content} [{res.metadata}]")
```

Output:

```
* 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'}]
```

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.

> **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.

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.

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.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]
```

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.

The authentication mode and parameters are selected by setting one of the following options:

< path="options_cloud.md" keyword="undefined">

- `--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.

< path="env_static.md" keyword="undefined">

- 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 YDBCLI 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
config info	Displaying connection parameters
config profile activate	Activating a profile
config profile create	Creating a profile
config profile delete	Deleting a profile
config profile get	Getting parameters of a profile
config profile list	List of profiles
config profile set	Activating a profile
discovery list	List of endpoints
discovery whoami	Authentication
export s3	Exporting data to S3 storage
import file csv	Importing data from a CSV file
import file tsv	Importing data from a TSV file
import s3	Importing data from S3 storage
init	Initializing the CLI, creating a profile
monitoring healthcheck	Health check
operation cancel	Aborting long-running operations
operation forget	Deleting long-running operations from the list
operation get	Status of long-running operations
operation list	List of long-running operations
scheme describe	Description of a data schema object
scheme ls	List of data schema objects
scheme mkdir	Creating a directory
scheme permissions chown	Change object owner
scheme permissions clear	Clear permissions
scheme permissions grant	Grant permission

scheme permissions revoke	Revoke permission
scheme permissions set	Set permissions
scheme permissions list	View permissions
scheme permissions clear-inheritance	Disable permission inheritance
scheme permissions set-inheritance	Enable permission inheritance
scheme rmdir	Deleting a directory
scripting yql	Executing a YQL script (deprecated, use `ydb sql`)
sql	Execute any query
table attribute add	Adding a table attribute
table attribute drop	Deleting a table attribute
table drop	Deleting a table
table index add global-async	Adding an asynchronous index
table index add global-sync	Adding a synchronous index
table index drop	Deleting an index
table query execute	Executing a YQL query (deprecated, use `ydb sql`)
table query explain	YQL query execution plan (deprecated, use `ydb sql --explain`)
table read	Streaming table reads
table ttl set	Setting TTL parameters
table ttl reset	Resetting TTL parameters
tools copy	Copying tables
tools dump	Dumping a directory or table to the file system
tools rename	Renaming tables
tools restore	Restoring data from the file system
topic create	Creating a topic
topic alter	Updating topic parameters and consumers
topic drop	Deleting a topic
topic consumer add	Adding a consumer to a topic
topic consumer drop	Deleting a consumer from a topic
topic consumer offset commit	Saving a consumer offset
topic read	Reading messages from a topic
topic write	Writing messages to a topic
update	Update the YDB CLI
version	Output details about the YDB CLI version
workload	Generate the workload
yql	Execute a YQL script (deprecated, use `ydb sql`)

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.

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 cluster
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
```

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.granular.remove_sche
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.granular.remove_sche
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.

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/seasons-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 default database settings do not allow an interval of less than 15 minutes (900 seconds). The default value is `3600`.

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
```

Getting a query plan

Get a query plan:

```
ydb 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 main section of the query plan, `tables`, contains information about querying tables. Reads are described in the `reads` section and writes in the `writes` section. The key characteristic of any table query is its type.

Types of reads:

- `FullScan`: Full table scan. All entries on all shards are read.
- `Scan`: A read of a certain range of entries.
- `Lookup`: A read by key or key prefix.
- `MultiLookup`: Multiple reads by key or key prefix. Supported, for example, in JOINs.

Types of writes:

- `Upsert`: Add a single entry.
- `MultiUpsert`: Add multiple entries.
- `Erase`: A single delete by key.
- `MultiErase`: Multiple deletes.

Let's take the query plan from the example above.
The `lookup_by` parameter shows what columns (key or key prefix) reads are made by.
The `scan_by` parameter shows what columns a read of all entries in a certain range of values is made by.
The `columns` parameter lists the columns whose values will be read from the table.

Example of query modification

Adjust the query so that you get only the first seasons of all the series:

```
ydb table query explain \
  -q "SELECT sa.title AS season_title, sr.title AS series_title, sr.series_id, sa.season_id
  FROM seasons AS sa
  INNER JOIN series AS sr ON sa.series_id = sr.series_id
  WHERE sa.season_id = 1"
```

This query plan implies that a `FullScan` is made for the `seasons` table and multiple reads are made for the `series` table (the `MultiLookup` type) by the key `series_id` (lookup_by). The `MultiLookup` read type and the `lookup_by` section indicate that the `series` table is subject to multiple reads by the `series_id` key.

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
```

Performing scan queries

> **Warning**
>
> This command is deprecated.
> The preferred way to run queries in YDB CLI is to use the `ydb sql` command.

You can run a query using Scan Queries via the YDB CLI by adding the `-t scan` flag to the `ydb table query execute` command.

Run the data query:

```
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"                |
└───────────┴────────────┴──────────────────────────────────────┘
```

Exporting and importing data

The YDB CLI contains a set of commands designed to export and import data and descriptions of data schema objects. Data can be exported to create backups for subsequent recovery and for other purposes.

- The export file structure is used for exporting data both to the file system and S3-compatible object storage.
- Exporting data to the file system using `tools dump`
- Importing data from the file system using `tools restore`
- Connecting to and authenticating with S3-compatible object storage
- Exporting data to S3-compatible object storage using `export s3`
- Importing data from S3-compatible object storage using `import s3`

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.

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

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

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:

```
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:

```
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:

```
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:

```
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.
- `--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.

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`).

Exporting data to S3-compatible storage

The `export s3` command starts exporting data and information on the server side about data schema objects to S3-compatible storage, in the format described under File structure:

```
ydb [connection options] export s3 [options]
```

Warning

The export feature is available only for objects of the following types:

- Directory
- Row-oriented table
- Secondary index
- Vector index

where [connection options] are database connection options

Command line parameters

`[options]`: Command parameters:

S3 connection parameters

To run the command to export data to S3 storage, specify the S3 connection parameters. Since data is exported by the YDB server asynchronously, the specified endpoint must be available to establish a connection on the server side.

List of exported items

`--item STRING`: Description of the item to export. You can specify the `--item` parameter multiple times if you need to export multiple items. `STRING` is set in `<property>=<value>,...` format with the following mandatory properties:

- `source`, `src`, or `s`: Path to the exported directory or table, `.` indicates the DB root directory. If you specify a directory, all of its items whose names do not start with a dot and, recursively, all subdirectories whose names do not start with a dot are exported.
- `destination`, `dst`, or `d`: Path (key prefix) in S3 storage to store exported items.

`--exclude STRING`: Template (PCRE) to exclude paths from export. Specify this parameter multiple times for different templates.

Additional parameters

Parameter	Description
`--description STRING`	Operation text description saved to the history of operations.
`--retries NUM`	Number of export retries to be made by the server. Defaults to `10`.
`--compression STRING`	Compress exported data. If the default compression level is used for the Zstandard algorithm, data can be compressed by 5-10 times. Compressing data uses the CPU and may affect the speed of performing other DB operations. Possible values: • `zstd`: Compression using the Zstandard algorithm with the default compression level (`3`). • `zstd-N`: Compression using the Zstandard algorithm, where `N` stands for the compression level (`1` — `22`).
`--format STRING`	Result format. Possible values: • `pretty`: Human-readable format (default). • `proto-json-base64`: Protocol Buffers in JSON format, binary strings are Base64-encoded.

Running the export command

Export result

If successful, the `export s3` command prints summary information about the enqueued operation to export data to S3, in the format specified in the `--format` option. The export 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://export/6?id=281474976788395&kind=s3 | true  | SUCC...
├--------------------------------------------┴-------┴----...
| StorageClass: NOT_SET
| Items:
...
```

- In the proto-json-base64 mode, the operation ID is in the "id" attribute:

```
{"id":"ydb://export/6?id=281474976788395&kind=s3","ready":true, ... }
```

Export status

Data is exported in the background. To find out the export status and progress, 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://export/6?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 export progress 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 export operation

When running the export operation, a directory named `export_*` is created in the root directory, where `*` is the numeric part of the export ID. This directory stores tables with a consistent snapshot of exported data as of the export start time.

Once the export is done, use the `operation forget` command to make sure the export is completed: the operation is removed from the list of operations and all files created for it are deleted:

```
ydb -p quickstart operation forget "ydb://export/6?id=281474976788395&kind=s3"
```

List of export operations

To get a list of export operations, run the `operation list export/s3` command:

```
ydb -p quickstart operation list export/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.

Exporting a database

Exporting all DB objects whose names do not start with a dot and that are not stored in directories whose names start with a dot to the `export1` directory in `mybucket` using the S3 authentication parameters from environment variables or the `~/.aws/credentials` file:

```
ydb -p quickstart export s3 \
  --s3-endpoint storage.yandexcloud.net --bucket mybucket \
  --item src=.,dst=export1
```

Exporting multiple directories

Exporting items from DB directories named dir1 and dir2 to the `export1` directory in `mybucket` using the explicitly set S3 authentication parameters:

```
ydb -p quickstart export s3 \
  --s3-endpoint storage.yandexcloud.net --bucket mybucket \
  --access-key VJGSOScgs-5kDGeo2hO9 --secret-key fZ_VB1Wi5-fdKSqH6074a7w0J4X0 \
  --item src=dir1,dst=export1/dir1 --item src=dir2,dst=export1/dir2
```

Getting operation IDs

To get a list of export operation IDs in a format suitable for handling in bash scripts, use the jq utility:

```
ydb -p quickstart operation list export/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://export/6?id=281474976789577&kind=s3
ydb://export/6?id=281474976789526&kind=s3
ydb://export/6?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 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 object 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 import some more data to your existing S3 tables (for example, using S3cmd), you can copy the S3 contents to the file system and use the `tools restore` command.

Command line parameters

`[options]`: Command parameters:

S3 connection 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.

List of imported objects

`--item STRING`: Description of the item to import. You can specify the `--item` parameter multiple times if you need to import multiple items. `STRING` is set in `<property>=<value>,...` format with the following mandatory properties:

- `source`, `src` or `s` is the path (key prefix) in S3 that hosts 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.

Additional parameters

`--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.
`--format STRING`: The format of the results.

- `pretty`: Human-readable format (default).
- `proto-json-base64`: Protobuf in JSON format, binary strings are Base64-encoded.

Importing

Export 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=2814749767883 95&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=2814749767883 95&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 \
  --item src=export1,dst=.
```

Importing multiple directories

Importing items from the dir1 and dir2 directories 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 VJGSOScgs-5kDGeo2hO9 --secret-key fZ_VB1Wi5-fdKSqH6074a7w0J4X0 \
  --item src=export/dir1,dst=dir1 --item src=export/dir2,dst=dir2
```

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
```

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.

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"           |
└──────────────┴───────────┴──────────────────────────────────────────┴──────────────────────┘
```

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-hours`	Data retention time in a topic, set in hours. The default value is `18`.
`--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 default value is `0` (no limit).
`--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-hours 2 \
  my-topic
```

View parameters of the created topic:

```
ydb -p quickstart scheme describe my-topic
```

Result:

```
RetentionPeriod: 2 hours
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-hours`	The retention period for topic data, in hours.
`--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 default value is `0` (no limit).
`--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: 2 hours
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: 2 hours
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 this command by calling it with `--help` option:

```
ydb sql --help
```

Parameters of the subcommand

Name	Description
`-h` , `--help`	Print general usage help.
`-hh`	Print complete usage help, including specific options 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 Running parameterized queries.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

A script to create a table, populate it with data, and select data from the table:

```
ydb -p quickstart sql -s '
    CREATE TABLE series (series_id Uint64, title Utf8, series_info Utf8, release_date Date, PRIMARY KEY
(series_id));
    COMMIT;
    UPSERT INTO series (series_id, title, series_info, release_date) values (1, "Title1", "Info1", Cast("2023-04-
20" as Date));
    COMMIT;
    SELECT * from series;
  '
```

Command output:

```
┌──────────────┬─────────────┬──────────────┬─────────────┐
| release_date | series_id | series_info | title    |
├──────────────┼─────────────┼──────────────┼─────────────┤
| "2023-04-20" | 1         | "Info1"     | "Title1" |
└──────────────┴─────────────┴──────────────┴─────────────┘
```

Running a script from the example above saved as the `script1.yql` file, with results output in `JSON` format:

```
ydb -p quickstart sql -f script1.yql --format json
```

Command output:

```
{"release_date":"2023-04-20","series_id":1,"series_info":"Info1","title":"Title1"}
```

You can find examples of passing parameters to queries in the article on how to pass parameters to `ydb sql`.

Running parameterized queries

Overview

YDB CLI can execute parameterized queries. To use parameters, you need to declare them using the YQL `DECLARE` command in your query text.

The preferred way to run parameterized queries in YDB CLI is to use the `ydb sql` command.

Parameter values can be set via the command-line arguments, uploaded from JSON files, and read from `stdin` in binary or JSON format. Binary data can be encoded as base64 or UTF-8. While reading from `stdin` or a file, you can stream multiple parameter values, triggering multiple query executions with batching options.

Why use parameterized queries?

Using parameterized queries offers several key advantages:

- **Enhanced Performance:** Parameterized queries significantly boost performance when executing multiple similar queries that differ only in input parameters. This is achieved through the use of prepared statements. The query is compiled once and then cached on the server. Subsequent requests with the same query text bypass the compilation phase, allowing for immediate execution.
- **Protection Against SQL Injection:** Another critical benefit of using parameterized queries is the protection they offer against SQL injection attacks. This security feature ensures that the input parameters are appropriately handled, mitigating the risk of malicious code execution.

Executing a single query

To provide parameters for a single query execution, you can use the command-line arguments, JSON files, or `stdin`, using the following YDB CLI options:

Name	Description
`-p, --param`	The value of a single query parameter in the `name=value` or `$name=value` format, where `name` is the parameter name and `value` is its value (a valid JSON value). This option can be specified multiple times. All specified parameters must be declared in the query using the DECLARE operator. Otherwise, you will receive the "Query does not contain parameter" error. If you specify the same parameter multiple times, you will receive the "Parameter value found in more than one source" error. Depending on your operating system, you might need to escape the `$` character or enclose your expression in single quotes (`'`).
`--input-file`	The name of a file in JSON format and UTF-8 encoding that contains parameter values matched against the query parameters by key names. Only one input file can be used. If values for the same parameter are found in multiple files or set by the `--param` command-line option, you will receive the "Parameter value found in more than one source" error. Keys that are present in the file but not declared in the query will be ignored without an error message.
`--input-format`	The format of parameter values applied to all sources of parameters (command line, file, or `stdin`). Available options: • `json` (default): JSON format. • `csv`: CSV format. • `tsv`: TSV format. • `raw`: Input is read as parameter values with no transformation or parsing. The parameter name should be set with the `--input-param-name` option.
`--input-binary-strings`	The input binary string encoding format. Defines how binary strings in the input should be interpreted. Available options: • `unicode`: Every byte in binary strings that is not a printable ASCII symbol (codes 32-126) should be encoded as UTF-8. • `base64`: Binary strings should be fully encoded with base64.

If values are specified for all non-optional (i.e., NOT NULL) parameters in the `DECLARE` clause, the query will be executed on the server. If a value is absent for even one such parameter, the command fails with the error message "Missing value for parameter".

More specific options for input parameters

The following options are not described in the `--help` output. To see their descriptions, use the `-hh` option instead.

Name	Description
`--input-framing`	The input framing format. Defines how parameter sets are delimited in the input. Available options: • `no-framing` (default): Data from the input is taken as a single set of parameters. • `newline-delimited`: A newline character delimits parameter sets in the input and triggers processing according to the `--input-batch` option.
`--input-param-name`	The parameter name in the input stream, required when the input format contains only values (that is, when `--input-format raw` is used).
`--input-columns`	A string with column names that replaces the CSV/TSV header. Relevant only when passing parameters in CSV/TSV format. It is assumed that the file does not contain a header.
`--input-skip-rows`	The number of CSV/TSV header rows to skip in the input data (excluding the row of column names if the `--header` option is used). Relevant only when passing parameters in CSV/TSV format.
`--input-batch`	The batch mode applied to parameter sets from `stdin` or `--input-file`. Available options: • `iterative` (default): Executes the query for each parameter set (exactly one execution when `no-framing` is specified for `--input-framing`). • `full`: A simplified batch mode where the query runs only once and all the parameter sets received from the input (`stdin` or `--input-file`) are wrapped into a `List<...>`. • `adaptive`: Executes the query with a JSON list of parameter sets when either the number of sets reaches `--input-batch-max-rows` or the waiting time reaches `--input-batch-max-delay`.
`--input-batch-max-rows`	The maximum size of the list for the input adaptive batching mode (default: 1000).
`--input-batch-max-delay`	The maximum delay before submitting a received parameter set for processing in the `adaptive` batch mode. The value is specified as a number with a time unit: `s` (seconds), `ms` (milliseconds), `m` (minutes), etc. Default value: `1s` (1 second). The YDB CLI starts a timer when it receives the first set of parameters for the batch from the input and sends the accumulated batch for execution once the timer expires. This parameter enables efficient batching when the arrival rate of new parameter sets is unpredictable.

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Passing the value of a single parameter

From the command line using `--param` option:

```
ydb -p quickstart sql -s 'DECLARE $a AS Int64; SELECT $a' --param '$a=10'
```

Using 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 set of one parameter:

```
echo '{"a":10}' | ydb -p quickstart sql -s 'DECLARE $a AS Int64; SELECT $a'
```

Via `stdin` passing only a parameter value and setting a 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 the values of parameters 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 Base64-encoded binary strings

```
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" |
└──────────────────────┘
```

Passing raw binary content directly

```
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 (the exact number of bytes may vary):

```
┌──────────┐
| column0 |
├──────────┤
| 66426   |
└──────────┘
```

Passing CSV data

```
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 executing a query multiple times with different sets of parameter values provided via `stdin` **or** an input file (but not both). In this case, the database connection is established once, and the query execution plan is cached. This approach significantly improves performance compared to making separate CLI calls.

To use this feature, stream different sets of values for the same parameters to the command input (`stdin` or `--input-file`) one after another, specifying a rule for the YDB CLI to separate the sets.

The query is executed as many times as there are parameter value sets received from the input. Each set is combined with the parameter values defined using the `--param` options. The command completes once the input stream is closed. Each query is executed within a dedicated transaction.

A rule for separating parameter sets (framing) complements the `--input-format` option:

Name	Description
`--input-framing`	Input framing format. Defines how parameter sets are delimited on the input. Available options: • `no-framing` (default): Data from the input is taken as a single set of parameters. • `newline-delimited`: A newline character delimits parameter sets in the input and triggers processing according to the `--input-batch` option.

Warning

When using a newline character as a separator between parameter sets, ensure that newline characters are not used inside the parameter sets. Quoting a text value does not allow newlines within the text. Multiline JSON documents are also not allowed.

Example

Streaming processing of multiple parameter sets

JSON

Suppose you need to run your query three times with the following sets of values for the `a` and `b` parameters:

1. `a` = 10, `b` = 20
2. `a` = 15, `b` = 25
3. `a` = 35, `b` = 48

Let's create a file that contains lines with JSON representations 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}
```

Let's execute the query by passing the content of this file to `stdin`, formatting the output as 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 just by passing the input 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}
```

This output can be passed as input to the next query command if it has a `column0` parameter.

CSV

Suppose you need to run your query three times with the following sets of values for the `a` and `b` parameters:

1. `a` = 10, `b` = 20
2. `a` = 15, `b` = 25
3. `a` = 35, `b` = 48

Let's create a file that contains lines with CSV representations of these sets:

```
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 content of this file to `stdin`, formatting the output as 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 just by passing the input 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
```

This output can be passed as input to another command running a different parameterized query.

TSV

Suppose you need to run your query three times, with the following sets of values for the `a` and `b` parameters:

1. `a` = 10, `b` = 20
2. `a` = 15, `b` = 25
3. `a` = 35, `b` = 48

Let's create a file that includes lines with TSV representations of these sets:

```
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 content of this file to `stdin`, formatting the output as TSV:

```
cat par1.txt | \
ydb -p quickstart sql \
  -s 'DECLARE $a AS Int64;
    DECLARE $b AS Int64;
    SELECT $a + $b' \
  --input-format tsv \
  --input-framing newline-delimited \
  --format tsv
```

Command output:

```
30
40
83
```

Or just by passing the input 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
```

This output can be passed as input to the next query command.

Streaming processing with joining parameter values from different sources

For example, you need to run your query three times with the following sets of values for the `a` and `b` parameters:

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}
```

Batched streaming processing

The YDB CLI supports automatic conversion of multiple consecutive parameter sets to a `List<...>`, enabling you to process them in a single request and transaction. As a result, you can achieve a substantial performance gain compared to one-by-one query processing.

Two batch modes are supported:

- Full
- Adaptive

Full batch mode

The `full` mode is a simplified batch mode where the query runs only once, and all the parameter sets received from the input (`stdin` or `--input-file`) are wrapped into a `List<...>`. If the request is too large, you will receive an error.

Use this batch mode when you want to ensure transaction atomicity by applying all the parameters within a single transaction.

Adaptive batch mode

In the `adaptive` mode, the input stream is split into multiple transactions, with the batch size automatically determined for each of them.

In this mode, you can process a broad range of dynamic workloads with unpredictable or infinite amounts of data, as well as workloads with an unpredictable or significantly varying rate of new sets appearing in the input. For example, this scenario is common when sending the output of another command to `stdin` using the `|` operator.

The adaptive mode solves two key issues of dynamic stream processing:

1. Limiting the maximum batch size.
2. Limiting the maximum data processing delay.

Syntax

To use the batching capabilities, define the `List<...>` or `List<Struct<...>>` parameter in the query's `DECLARE` clause, and use the following options:

Name	Description
`--input-batch`	The batch mode applied to parameter sets on `stdin` or `--input-file`. Available options: • `iterative` (default): Executes the query for each parameter set (exactly one execution when `no-framing` is specified for `--input-framing`). • `full`: A simplified batch mode where the query runs only once and all the parameter sets received from the input (`stdin` or `--input-file`) are wrapped into a `List<...>`. • `adaptive`: Executes the query with a JSON list of parameter sets whenever the number of sets reaches `--input-batch-max-rows` or the waiting time reaches `--input-batch-max-delay`.

In the adaptive batch mode, you can use the following additional parameters:

Name	Description
`--input-batch-max-rows`	The maximum number of parameter sets per batch in the `adaptive` batch mode. The next batch will be sent with the query if the number of parameter sets reaches the specified limit. When set to `0`, there is no limit. Default value: `1000`. Parameter values are sent to each query execution without streaming, so the total size per gRPC request that includes the parameter values has an upper limit of about 5 MB.
`--input-batch-max-delay`	The maximum delay before submitting a received parameter set for processing in the `adaptive` batch mode. The value is specified as a number with a time unit: `s` (seconds), `ms` (milliseconds), `m` (minutes), etc. Default value: `1s` (1 second). The YDB CLI starts a timer when it receives the first set of parameters for the batch from the input and sends the accumulated batch for execution once the timer expires. This parameter enables efficient batching when the arrival rate of new parameter sets is unpredictable.

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 data processing delay

This example demonstrates adaptive batching triggered by a processing delay. In the first line of the command below, we generate 1,000 rows with a delay of 0.2 seconds on `stdout` and pipe them to `stdin` for the `ydb sql` query execution command. The query execution command displays the 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 (the actual values may differ):

```
┌──────────┬──────────────
```

Deleting multiple records from a YDB table based on primary keys

> **Warning**
>
> Supported only for row-oriented tables. Support for column-oriented tables is currently under development.

If you attempt to delete a large number of rows from a substantial table using a simple `DELETE FROM large_table WHERE id > 10;` statement, you may encounter an error due to exceeding the transaction record limit. This example shows how to delete an unlimited number of records from YDB tables without breaching this limitation.

Let's create a test table:

```
ydb -p quickstart sql -s 'CREATE TABLE test_delete_1(id UInt64 NOT NULL, PRIMARY KEY (id))'
```

Add 100,000 records to it:

```
for i in $(seq 1 100000); do echo "$i"; done | \
ydb -p quickstart import file csv -p test_delete_1
```

Delete all records with `id` 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 provided in 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

After executing `ydb` command without subcommands, the interactive query execution mode will be launched. The console or terminal will be switched to interactive mode. After that, you can enter queries directly into the console or terminal, and when you enter a newline character, the query is considered complete and it starts to execute. The query text can be either a YQL query or a special command. Also, query history is saved between runs, command autocomplete is available, and there is a search function for query history.

General format of the command:

```
ydb [global options...]
```

- `global options` — global parameters.

Hotkeys

Hotkey	Description
`CTRL + D`	Allows you to exit interactive mode.
`Up arrow`	Scrolls through query history toward older queries.
`Down arrow`	Scrolls through query history toward newer queries.
`TAB`	Autocompletes the entered text to a suitable YQL command.
`CTRL + R`	Allows searching for a query in history containing a specified substring.

Special commands

Special commands are CLI-specific commands and are not part of the YQL syntax. They are intended for performing various functions that cannot be accomplished through a YQL query.

Command	Description
`SET param = value`	The `SET` command 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.

Examples

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
```

Running a script (with streaming support)

> **Warning**
>
> This command is deprecated.
> The preferred way to run queries in YDB CLI is to use the `ydb sql` command.

You can use the `yql` subcommand to run a YQL script. The script can include queries of different types. Unlike `scripting yql`, the `yql` subcommand establishes a streaming connection and retrieves data through it. With the in-stream query execution, no limit is imposed on the amount of data read.

General format of the command:

```
ydb [global options...] yql [options...]
```

- `global options`: Global parameters.
- `options`: Parameters of the subcommand.

View the description of the YQL script command:

```
ydb yql --help
```

Parameters of the subcommand

Name	Description
`--timeout`	The time within which the operation should be completed on the server.
`--stats`	Statistics mode. Acceptable values: - `none` (default): Do not collect. - `basic`: Collect statistics for basic events. - `full`: Collect statistics for all events.
`-s`, `--script`	Text of the YQL query to be executed.
`-f`, `--file`	Path to the text of the YQL query to be executed.

`--format`	Result format. Possible values: • `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.

Name	Description
`-p, --param`	The value of a single parameter of a YQL query, in the format: `$name=value` , where `$name` is the parameter name and `value` is its value (a valid JSON value).
`--param-file`	Name of the file in JSON format and in UTF-8 encoding that specifies values of the parameters matched against the YQL query parameters by key names.
`--input-format`	Format of parameter values. Applies to all the methods of parameter transmission (among command parameters, in a file or using `stdin`). Acceptable values: • `json-unicode` (default):JSON. • `json-base64` : JSON format in which values of binary string parameters (`DECLARE $par AS String`) are Base64-encoded.
`--stdin-format`	The parameter format and framing for `stdin` . To set both values, specify the parameter twice. **Format of parameter encoding for `stdin`** Acceptable values: • `json-unicode` : JSON. • `json-base64` : JSON format in which values of binary string parameters (`DECLARE $par AS String`) are Base64-encoded. • `raw` is binary data; the parameter name is set in `--stdin-par` . If the format of parameter encoding for `stdin` isn't specified, the format set in `--input-format` is used. **Classification of parameter sets for `stdin` (framing)** Acceptable values: • `no-framing` (default): Framing isn't used • `newline-delimited` : The newline character is used in `stdin` to end a given parameter set, separating it from the next one.
`--stdin-par`	The name of the parameter whose value will be sent over `stdin` is specified without a `$` .
`--batch`	The batch mode of transmitting parameter sets received via `stdin` . Acceptable values: • `iterative` (default): Batch mode is disabled • `full` : Full-scale batch mode is enabled • `adaptive` : Adaptive batching is enabled

`--batch-limit`	A maximum number of sets of parameters per batch in the adaptive batch mode. The setting of `0` removes the limit. The default value is `1000`.
`--batch-max-delay`	The maximum delay related to processing the resulting parameter set in the adaptive batch mode. It's set as a number of `s`, `ms`, `m`. Default value: `1s` (1 second).

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

A script to create a table, populate it with data, and select data from the table:

```
ydb -p quickstart yql -s '
    CREATE TABLE series (series_id Uint64, title Utf8, series_info Utf8, release_date Date, PRIMARY KEY
(series_id));
    COMMIT;
    UPSERT INTO series (series_id, title, series_info, release_date) values (1, "Title1", "Info1", Cast("2023-04-
20" as Date));
    COMMIT;
    SELECT * from series;
  '
```

Command output:

```
┌──────────────┬─────────────┬──────────────┬──────────────┐
| release_date | series_id | series_info | title    |
├──────────────┼─────────────┼──────────────┼──────────────┤
| "2023-04-20" | 1         | "Info1"     | "Title1" |
└──────────────┴─────────────┴──────────────┴──────────────┘
```

Running a script from the example above saved as the `script1.yql` file, with results output in `JSON` format:

```
ydb -p quickstart yql -f script1.yql --format json-unicode
```

Command output:

```
{"release_date":"2023-04-20","series_id":1,"series_info":"Info1","title":"Title1"}
```

You can find examples of passing parameters to scripts in the article on how to pass parameters to YQL execution commands.

Running a script

> **Warning**
>
> This command is deprecated.
> The preferred way to run queries in YDB CLI is to use the `ydb sql` command.

You can use the `scripting yql` subcommand to run a YQL script. The script can include queries of different types. Unlike `yql`, the `scripting yql` command has a limit on the number of returned rows and accessed data.

General format of the command:

```
ydb [global options...] scripting yql [options...]
```

- `global options`: Global parameters.
- `options`: Parameters of the subcommand.

View the description of the YQL script command:

```
ydb scripting yql --help
```

Parameters of the subcommand

Name	Description
`--timeout`	The time within which the operation should be completed on the server.
`--stats`	Statistics mode. Acceptable values: • `none`: Do not collect statistics. • `basic`: Collect statistics for basic events. • `full`: Collect statistics for all events. Defaults to `none`.
`-s`, `--script`	Text of the YQL query to be executed.
`-f`, `--file`	Path to the text of the YQL query to be executed.
`--explain`	Show the query execution plan.

`--show-response-metadata`	Show the response metadata.
`--format`	Result format. Default value: `pretty`. Acceptable values: • `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.

Working with parameterized queries

A brief help is provided below. For a detailed description with examples, see Running parametrized YQL queries and scripts.

Name	Description
`-p, --param`	The value of a single parameter of a YQL query, in the format: `$name=value`, where `$name` is the parameter name and `value` is its value (a valid JSON value).
`--param-file`	Name of the file in JSON format and in UTF-8 encoding that specifies values of the parameters matched against the YQL query parameters by key names.
`--input-format`	Format of parameter values. Applies to all the methods of parameter transmission (among command parameters, in a file or using `stdin`). Acceptable values: • `json-unicode` (default):JSON. • `json-base64`: JSON format in which values of binary string parameters (`DECLARE $par AS String`) are Base64-encoded.
`--stdin-format`	The parameter format and framing for `stdin`. To set both values, specify the parameter twice. **Format of parameter encoding for `stdin`** Acceptable values: • `json-unicode`: JSON. • `json-base64`: JSON format in which values of binary string parameters (`DECLARE $par AS String`) are Base64-encoded. • `raw` is binary data; the parameter name is set in `--stdin-par`. If the format of parameter encoding for `stdin` isn't specified, the format set in `--input-format` is used. **Classification of parameter sets for `stdin` (framing)** Acceptable values: • `no-framing` (default): Framing isn't used • `newline-delimited`: The newline character is used in `stdin` to end a given parameter set, separating it from the next one.

`--stdin-par`	The name of the parameter whose value will be sent over `stdin` is specified without a `$`.
`--batch`	The batch mode of transmitting parameter sets received via `stdin`. Acceptable values: • `iterative` (default): Batch mode is disabled • `full`: Full-scale batch mode is enabled • `adaptive`: Adaptive batching is enabled
`--batch-limit`	A maximum number of sets of parameters per batch in the adaptive batch mode. The setting of `0` removes the limit. The default value is `1000`.
`--batch-max-delay`	The maximum delay related to processing the resulting parameter set in the adaptive batch mode. It's set as a number of `s`, `ms`, `m`. Default value: `1s` (1 second).

Examples

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

A script to create a table, populate it with data, and select data from the table:

```
ydb -p quickstart scripting yql -s '
    CREATE TABLE series (series_id Uint64, title Utf8, series_info Utf8, release_date Date, PRIMARY KEY
(series_id));
    COMMIT;
    UPSERT INTO series (series_id, title, series_info, release_date) values (1, "Title1", "Info1", Cast("2023-04-
20" as Date));
    COMMIT;
    SELECT * from series;
  '
```

Command output:

```
┌──────────────┬───────────┬─────────────┬──────────┐
| release_date | series_id | series_info | title    |
├──────────────┼───────────┼─────────────┼──────────┤
| "2023-04-20" | 1         | "Info1"     | "Title1" |
└──────────────┴───────────┴─────────────┴──────────┘
```

Running a script from the example above saved as the `script1.yql` file, with results output in `JSON` format:

```
ydb -p quickstart scripting yql -f script1.yql --format json-unicode
```

Command output:

```
{"release_date":"2023-04-20","series_id":1,"series_info":"Info1","title":"Title1"}
```

You can find examples of passing parameters to scripts in the article on how to pass parameters to YQL execution commands.

Running a query

> **Warning**
>
> This command is deprecated.
> The preferred way to run queries in YDB CLI is to use the `ydb sql` command.

The `table query execute` subcommand is designed for reliable execution of YQL queries. With this sub-command, you can successfully execute your query when certain table partitions are unavailable for a short time (for example, due to being split or merged) by using built-in retry policies.

General format of the command:

```
ydb [global options...] table query execute [options...]
```

- `global options` : Global parameters.
- `options` : Parameters of the subcommand.

View the description of the YQL query command:

```
ydb table query execute --help
```

Parameters of the subcommand

Name	Description
`--timeout`	The time within which the operation should be completed on the server.
`-t`, `--type`	Query type. Acceptable values: • `data` : A YQL query that includes DML operations; it can be used both to update data in the database and fetch several selections limited to 1,000 rows per selection. • `scan` : A YQL query of the scan type. It can only be used to read data from the database. It returns a single selection, but without a limit on the number of records in it. The algorithm of executing a `scan` query on the server is more sophisticated compared to a `data` query. Hence, if you don't need to return more than 1,000 rows, `data` queries are more effective. • `scheme` : A YQL query that includes DDL operations. The default value is `data` .

`--stats`	Statistics mode. Acceptable values: • `none`: Do not collect statistics. • `basic`: Collect statistics for basic events. • `full`: Collect statistics for all events. Defaults to `none`.
`-s`	Enable statistics collection in the `basic` mode.
`--tx-mode`	Transaction mode (for `data` queries). Acceptable values: • `serializable-rw`: The result of parallel transactions is equivalent to their serial execution. • `online-ro`: Each of the reads in the transaction reads data that is most recent at the time of its execution. • `stale-ro`: Data reads in a transaction return results with a possible delay (fractions of a second).Default value: `serializable-rw`.
`-q`, `--query`	Text of the YQL query to be executed.
`-f`, `--file`	Path to the text of the YQL query to be executed.

Name	Description
`-p, --param`	The value of a single parameter of a YQL query, in the format: `$name=value`, where `$name` is the parameter name and `value` is its value (a valid JSON value).
`--param-file`	Name of the file in JSON format and in UTF-8 encoding that specifies values of the parameters matched against the YQL query parameters by key names.
`--input-format`	Format of parameter values. Applies to all the methods of parameter transmission (among command parameters, in a file or using `stdin`). Acceptable values: • `json-unicode` (default):JSON. • `json-base64`: JSON format in which values of binary string parameters (`DECLARE $par AS String`) are Base64-encoded.
`--stdin-format`	The parameter format and framing for `stdin`. To set both values, specify the parameter twice. **Format of parameter encoding for `stdin`** Acceptable values: • `json-unicode`: JSON. • `json-base64`: JSON format in which values of binary string parameters (`DECLARE $par AS String`) are Base64-encoded. • `raw` is binary data; the parameter name is set in `--stdin-par`. If the format of parameter encoding for `stdin` isn't specified, the format set in `--input-format` is used. **Classification of parameter sets for `stdin` (framing)** Acceptable values: • `no-framing` (default): Framing isn't used • `newline-delimited`: The newline character is used in `stdin` to end a given parameter set, separating it from the next one.
`--stdin-par`	The name of the parameter whose value will be sent over `stdin` is specified without a `$`.
`--batch`	The batch mode of transmitting parameter sets received via `stdin`. Acceptable values: • `iterative` (default): Batch mode is disabled • `full`: Full-scale batch mode is enabled • `adaptive`: Adaptive batching is enabled
`--batch-limit`	A maximum number of sets of parameters per batch in the adaptive batch mode. The setting of `0` removes the limit. The default value is `1000`.
`--batch-max-delay`	The maximum delay related to processing the resulting parameter set in the adaptive batch mode. It's set as a number of `s`, `ms`, `m`. Default value: `1s` (1 second).

Examples

Note

The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Creating tables

```
ydb -p quickstart table query execute \
  --type scheme \
  -q '
  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));
  '
```

Populating the table with data

```
ydb -p quickstart table query execute \
  -q '
UPSERT INTO series (series_id, title, release_date, series_info) VALUES
  (1, "IT Crowd", Date("2006-02-03"), "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", Date("2014-04-06"), "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
in Silicon Valley.");

UPSERT INTO seasons (series_id, season_id, title, first_aired, last_aired) VALUES
    (1, 1, "Season 1", Date("2006-02-03"), Date("2006-03-03")),
    (1, 2, "Season 2", Date("2007-08-24"), Date("2007-09-28")),
    (2, 1, "Season 1", Date("2014-04-06"), Date("2014-06-01")),
    (2, 2, "Season 2", Date("2015-04-12"), Date("2015-06-14"));

UPSERT INTO episodes (series_id, season_id, episode_id, title, air_date) VALUES
    (1, 1, 1, "Yesterday'"'"'s Jam", Date("2006-02-03")),
    (1, 1, 2, "Calamity Jen", Date("2006-02-03")),
    (2, 1, 1, "Minimum Viable Product", Date("2014-04-06")),
    (2, 1, 2, "The Cap Table", Date("2014-04-13"));
'
```

Simple data selection

```
ydb -p quickstart table query execute -q '
  SELECT season_id, episode_id, title
  FROM episodes
  WHERE series_id = 1
'
```

Result:

```
┌─────────────┬────────────┬──────────────────────┐
| season_id | episode_id | title              |
├─────────────┼────────────┼──────────────────────┤
| 1         | 1          | "Yesterday's Jam" |
├─────────────┼────────────┼──────────────────────┤
| 1         | 2          | "Calamity Jen"    |
└─────────────┴────────────┴──────────────────────┘
```

Unlimited selection for automated processing

Selecting data by a query whose text is saved to a file, without a limit on the number of rows in the selection and data output in the format: Newline-delimited JSON stream.

Let's write the query text to the `request1.yql` file.

```
echo 'SELECT season_id, episode_id, title FROM episodes' > request1.yql
```

Now, run the query:

```
ydb -p quickstart table query execute -f request1.yql --type scan --format json-unicode
```

Result:

```
{"season_id":1,"episode_id":1,"title":"Yesterday's Jam"}
{"season_id":1,"episode_id":2,"title":"Calamity Jen"}
{"season_id":1,"episode_id":1,"title":"Minimum Viable Product"}
{"season_id":1,"episode_id":2,"title":"The Cap Table"}
```

Passing parameters

You can find examples of executing parameterized queries, including streamed processing, in the Passing parameters to YQL execution commands article.

Warning

This page is outdated. Please refer to Running parameterized queries for up-to-date information.

Running parametrized YQL queries and scripts

Overview

YDB CLI can execute parameterized YQL queries. To use parameters you need to declare them using the YQL `DECLARE` command in your YQL query text.

To run parameterized YQL queries you can use the following YDB CLI commands:

- ydb yql.
- ydb scripting yql.
- ydb table query execute.

These commands support the same query parametrization options. Parameter values can be set on the command line, uploaded from JSON files, and read from `stdin` in binary or JSON format. On `stdin` you can stream multiple parameter values triggering multiple YQL query executions with batching options.

> **Warning**
>
> Among the above commands, only the `table query execute` applies retry policies. Such policies ensure reliable query execution and continuity when certain data ranges are unavailable for a short time because of partition changes or other regular processes in a distributed database.

Executing a single YQL query

To provide parameters for a YQL query execution, you can use command line, JSON files, and `stdin`, using the following YDB CLI options:

Name	Description
`-p, --param`	An expression in the format `$name=value`, where `$name` is the name of the YQL query parameter and `value` is its value (a correct JSON value). The option can be specified repeatedly. All the specified parameters must be declared in the YQL query by the DECLARE operator; otherwise, you will get an error "Query does not contain parameter". If you specify the same parameter several times, you will get an error "Parameter value found in more than one source". Depending on your operating system, you might need to escape the `$` character or enclose your expression in single quotes (`'`).
`--param-file`	Name of a file in JSON format in UTF-8 encoding that contains parameter values matched against the YQL query parameters by key names. The option can be specified repeatedly. If values of the same parameter are found in multiple files or set by the `--param` command line option, you'll get an error "Parameter value found in more than one source". Names of keys in the JSON file are expected without the leading `$` sign. Keys that are present in the file but aren't declared in the YQL query will be ignored without an error message.
`--input-format`	Format of parameter values, applied to all sources of parameters (command line, file, or `stdin`). Available options: • `json-unicode` (default):JSON. • `json-base64` : JSON with values of binary string parameters (`DECLARE $par AS String`) are Base64-encoded. This feature enables you to process binary data, being decoded from Base64 by the YDB CLI.
`--stdin-format`	Format of parameter values for `stdin`. The YDB CLI automatically detects that a file or an output of another shell command has been redirected to the standard input device `stdin`. In this case, the CLI interprets the incoming data based on the following available options: • `json-unicode` : JSON. • `json-base64` : JSON with values of binary string parameters (`DECLARE $par AS String`) are Base64-encoded. • `raw` : Binary data. If format of parameter values for `stdin` isn't specified, the `--input-format` is used.

`--stdin-par`	Name of a parameter whose value is provided on `stdin`, without a `$` sign. This name is required when you use the `raw` format in `--stdin-format`. When used with JSON formats, `stdin` is interpreted not as a JSON document but as a JSON value passed to the parameter with the specified name.

The query will be executed on the server once, provided that values are specified for all the parameters in the `DECLARE` clause. If a value is absent for at least one parameter, the command fails with the "Missing value for parameter" message.

Examples

In our examples, we use the `table query execute` command, but you can also run them using the `yql` and `scripting yql` commands.

> **Note**
>
> The examples use the `quickstart` profile. To learn more, see Creating a profile to connect to a test database.

Passing the value of a single parameter

From the command line:

```
ydb -p quickstart table query execute -q 'declare $a as Int64;select $a' --param '$a=10'
```

Using a file:

```
echo '{"a":10}' > p1.json
ydb -p quickstart table query execute -q 'declare $a as Int64;select $a' --param-file p1.json
```

Via `stdin`:

```
echo '{"a":10}' | ydb -p quickstart table query execute -q 'declare $a as Int64;select $a'
```

```
echo '10' | ydb -p quickstart table query execute -q 'declare $a as Int64;select $a' --stdin-par a
```

Passing the values of parameters of different types from multiple sources

```
echo '{ "a":10, "b":"Some text", "x":"Ignore me" }' > p1.json
echo '{ "c":"2012-04-23T18:25:43.511Z" }' | ydb -p quickstart table query execute \
  -q 'declare $a as Int64;
      declare $b as Utf8;
      declare $c as DateTime;
      declare $d as Int64;

      select $a, $b, $c, $d' \
  --param-file p1.json \
  --param '$d=30'
```

Command output:

```
┌─────────┬─────────────┬──────────────────────┬─────────┐
| column0 | column1     | column2              | column3 |
├─────────┼─────────────┼──────────────────────┼─────────┤
| 10      | "Some text" | "2012-04-23T18:25:43Z" | 30      |
└─────────┴─────────────┴──────────────────────┴─────────┘
```

Passing Base64-encoded binary strings

```
ydb -p quickstart table query execute \
  -q 'DECLARE $a AS String;
      SELECT $a' \
  --input-format json-base64 \
  --param '$a="SGVsbG8sIHdvcmxkCg=="'
```

Command output:

```
┌──────────────────────┐
| column0              |
├──────────────────────┤
| "Hello, world\n" |
└──────────────────────┘
```

Passing binary content directly

```
curl -Ls http://ydb.tech/docs | ydb -p quickstart table query execute \
  -q 'DECLARE $a AS String;
      SELECT LEN($a)' \
  --stdin-format raw \
  --stdin-par a
```

Command output (exact number of bytes may vary):

```
┌────────────┐
| column0 |
├────────────┤
| 66426   |
└────────────┘
```

Iterative streaming processing

YDB CLI supports execution of a YQL query multiple times with different sets of parameter values provided on `stdin`. In this case, the database connection is established once and the query execution plan is cached. This substantially increases the performance of such an approach compared to separate CLI calls.

To use this feature, you need to stream different sets of the same parameters to `stdin` one after another, specifying a rule for the YDB CLI on how to separate the sets from each other.

The YQL query runs as many times as many parameter value sets received on `stdin`. Each set received on `stdin` is joined with the parameter values defined on other sources (`--param`, `--param-file`). The command will complete once the `stdin` stream is closed. Each query is executed within a dedicated transaction.

A rule for separating parameter sets from one another (framing) complements the `stdin` format specified by the `--stdin-format` option:

Name	Description
`--stdin-format`	Defines the `stdin` framing. Available options: • `no-framing` (default): No framing, `stdin` expects a single set of parameters, and the YQL query is executed only once when the `stdin` stream is closed. • `newline-delimited`: A newline character marks the end of one set of parameter values on `stdin`, separating it from the next one. The YQL query is executed each time a newline character is read from `stdin`.

Warning

When using a newline character as a separator between the parameter sets, make sure that it isn't used inside the parameter sets. Putting some text value in quotes does not enable newlines within the text. Multiline JSON documents are not allowed.

Example

Streaming processing of multiple parameter sets

Suppose you need to run your query thrice, with the following sets of values for the `a` and `b` parameters:

1. `a` = 10, `b` = 20
2. `a` = 15, `b` = 25
3. `a` = 35, `b` = 48

Let's create a file that includes lines with JSON representations of these sets:

```
echo -e '{"a":10,"b":20}\n{"a":15,"b":25}\n{"a":35,"b":48}' > par1.txt
cat par1.txt
```

Command output:

```
{"a":10,"b":20}
{"a":15,"b":25}
{"a":35,"b":48}
```

Let's execute the query by passing the content of this file to `stdin`, formatting the output as JSON:

```
cat par1.txt | \
ydb -p quickstart table query execute \
  -q 'DECLARE $a AS Int64;
      DECLARE $b AS Int64;
      SELECT $a + $b' \
  --stdin-format newline-delimited \
  --format json-unicode
```

Command output:

```
{"column0":30}
{"column0":40}
{"column0":83}
```

This output can be passed as input to the next YQL query command.

Streaming processing with joining parameter values from different sources

For example, you need to run your query thrice, with the following sets of values for the `a` and `b` parameters:

1. `a` = 10, `b` = 100
2. `a` = 15, `b` = 100
3. `a` = 35, `b` = 100

```
echo -e '10\n15\n35' | \
ydb -p quickstart table query execute \
  -q 'DECLARE $a AS Int64;
      DECLARE $b AS Int64;
      SELECT $a + $b AS sum1' \
  --param '$b=100' \
  --stdin-format newline-delimited \
  --stdin-par a \
  --format json-unicode
```

Command output:

```
{"sum1":110}
{"sum1":115}
{"sum1":135}
```

Batched streaming processing

The YDB CLI supports automatic conversion of multiple consecutive parameter sets to a `List<>`, enabling you to process them in a single request and transaction. As a result, you can have a substantial performance gain compared to one-by-one query processing.

Two batch modes are supported:

- Full
- Adaptive

Full batch mode

The `full` mode is a simplified batch mode where the query runs only once, and all the parameter sets received through `stdin` are wrapped into a `List<>`. If the request is too large, you will get an error.

Use this batch mode when you want to ensure transaction atomicity by applying all the parameters within a single transaction.

Adaptive batch mode

In the `adaptive` mode, the input stream is split into multiple transactions, with the batch size automatically determined for each of them.

In this mode, you can process a broad range of dynamic workloads with unpredictable or infinite amounts of data, as well as with unpredictable or significantly varying rate of new sets appearance at the input. For example, such a profile is typical when sending the output of another command to `stdin` using the `|` operator.

The adaptive mode solves two basic issues of dynamic stream processing:

1. Limiting the maximum batch size.
2. Limiting the maximum data processing delay.

Syntax

To use the batching capbilities, define the `List<...>` or `List<Struct<...>>` parameter in the YQL query's DECLARE clause, and use the following options:

Name	Description
`--batch`	The batch mode applied to parameter sets on `stdin`. Available options: • `iterative` (default): Batching is disabled. • `full`: Full batch mode. The YQL query runs only once when `stdin` is closed, with all the received sets of parameters wrapped into `List<>`, the parameter name is set by the `--stdin-par` option. • `adaptive`: Adaptive batch mode. The YQL query runs every time when limits are exceeded either on the number of parameter sets per query (`--batch-limit`) or on the batch processing delay (`--batch-max-delay`). All the sets of parameters received by that moment are wrapped into a `List<>`, the parameter name is set by the `--stdin-par` option.

In the adaptive batch mode, you can use the following additional parameters:

Name	Description
`--batch-limit`	The maximum number of sets of parameters per batch in the adaptive batch mode. The next batch will be sent to the YQL query if the number of parameter sets in it reaches the specified limit. When it's `0`, there's no limit. Default value: `1000`. Parameter values are sent to each YQL execution without streaming, so the total size per GRPC request that includes the parameter values has the upper limit of about 5 MB.
`--batch-max-delay`	The maximum delay to submit a received parameter set for processing in the adaptive batch mode. It's set as a number with a time unit - `s`, `ms`, `m`. Default value: `1s` (1 second). The YDB CLI starts a timer when it receives a first set of parameters for the batch on `stdin`, and sends the whole accumulated batch for execution once the timer expires. With this parameter, you can batch efficiently when new parameter sets arrival rate on `stdin` is unpredictable.

Examples: Full batch processing

```
echo -e '{"a":10,"b":20}\n{"a":15,"b":25}\n{"a":35,"b":48}' | \
ydb -p quickstart table query execute \
  -q 'DECLARE $x AS List<Struct<a:Int64,b:Int64>>;
```

```
    SELECT ListLength($x), $x' \
  --stdin-format newline-delimited \
  --stdin-par x \
  --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 data processing delay

This example demonstrates the adaptive batching triggered by a processing delay. In the first line of the command below, we generate 1,000 rows at a delay of 0.2 seconds on `stdout` and pipe them to `stdin` to the YQL query execution command. The YQL query execution command shows the parameter batches in each subsequent YQL query call.

```
for i in $(seq 1 1000); do echo "Line$i"; sleep 0.2; done | \
ydb -p quickstart table query execute \
  -q 'DECLARE $x AS List<Utf8>;
      SELECT ListLength($x), $x' \
  --stdin-format newline-delimited \
  --stdin-format raw \
  --stdin-par x \
  --batch adaptive
```

Command output (actual values may differ):

```
┌──────────┬──────────────────────────────────────────────────────────
```

```
  --batch-limit 20 \
  --param '$p2=10' \
  --format json-unicode
```

Command output:

```
{"count":20,"p2":10,"items":
["Line1","Line2","Line3","Line4","Line5","Line6","Line7","Line8","Line9","Line10","Line11","Line12","Line13","Line14"
{"count":20,"p2":10,"items":
["Line21","Line22","Line23","Line24","Line25","Line26","Line27","Line28","Line29","Line30","Line31","Line32","Line33"
...
{"count":20,"p2":10,"items":
["Line161","Line162","Line163","Line164","Line165","Line166","Line167","Line168","Line169","Line170","Line171","Line1
{"count":20,"p2":10,"items":
["Line181","Line182","Line183","Line184","Line185","Line186","Line187","Line188","Line189","Line190","Line191","Line1
```

Deleting multiple records from a YDB table based on primary keys

This example shows how you can delete an unlimited number of records from YDB tables without risking exceeding the limit on the number of records per transaction.

Let's create a test table:

```
ydb -p quickstart yql -s 'create table test_delete_1( id UInt64 not null, primary key (id))'
```

Add 100,000 records to it:

```
for i in $(seq 1 100000); do echo "$i"; done | \
ydb -p quickstart import file csv -p test_delete_1
```

Delete all records with ID > 10:

```
ydb -p quickstart table query execute -t scan \
  -q 'SELECT t.id FROM test_delete_1 AS t WHERE t.id > 10' \
  --format json-unicode | \
ydb -p quickstart table query execute \
  -q 'DECLARE $lines AS List<Struct<id:UInt64>>;
      DELETE FROM test_delete_1 WHERE id IN (SELECT tl.id FROM AS_TABLE($lines) AS tl)' \
  --stdin-format newline-delimited \
  --stdin-par lines \
  --batch adaptive \
  --batch-limit 10000
```

Processing of messages read from a topic

Examples of processing messages read from a topic are given in Running an SQL query with the transmission of messages from the topic as parameters.

See also

- Parameterized YQL queries in YDB SDK

Getting a list of long-running operations

Use the `ydb operation list` subcommand to get a list of long-running operations of the specified type.

General format of the command:

```
ydb [global options...] operation list [options...] <kind>
```

- `global options` : Global parameters.
- `options` : Parameters of the subcommand.
- `kind` : The type of operation. Possible values:
  - `buildindex` : The build index operations.
  - `export/s3` : The export operations.
  - `import/s3` : The import operations.

View a description of the command to get a list of long-running operations:

```
ydb operation list --help
```

Parameters of the subcommand

Name	Description
`-s`, `--page-size`	Number of operations on one page. If the list of operations contains more strings than specified in the `--page-size` parameter, the result will be split into several pages. To get the next page, specify the `--page-token` parameter.
`-t`, `--page-token`	Page token.
`--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.

Get a list of long-running build index 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_release |
└──────────────────────────────────────────┴───────┴─────────┴───────┴──────────┴──────────────────────┴──────────────┘

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_release |
└──────────────────────────────────────────┴───────┴─────────┴───────┴──────────┴──────────────────────┴─────────────┘
```

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.
   yandex.cloud/docs/iam/concepts/authorization/oauth-token
     [4] Use metadata service on a virtual machine (use-metadata-credentials)
   yandex.cloud/docs/compute/operations/vm-connect/auth-inside-vm
     [5] Use service account key file (sa-key-file) yandex.cloud/docs/iam/operations/iam-token/create-for-sa
     [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 profiles
```

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]
```

where [connection options] are database connection options

The response includes the account name (User SID) and, if the `-g` option is specified, the information whether the account belongs to groups.

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.

Example

```
$ ydb -p quickstart discovery whoami -g
User SID: aje5kkjdgs0puc18976co@as

User has no groups
```

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`.

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.
- ClickBench: ClickBench analytical benchmark.
- TPC-H: TPC-H benchmark.
- TPC-DS: TPC-DS benchmark.
- Topic: Topic load.
- Transfer: Transfer load.

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 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 =
ENABLED, 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>,
AUTO_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_PARTITIONS = <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`: 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 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 transactions 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.

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`: The 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`: The 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`: The 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`: The 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`: The 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`.	`row`.
`--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`
`--clear`	If the table at the specified path has already been created, it will be deleted.	

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.	

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

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
`--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

`--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`.	Not saved by default
`--query-settings <setting>`	Query execution settings. Each setting is added as a separate line at the beginning of each query. Use multiple times for multiple settings.	Not specified by default
`--include`	Query numbers or segments to be executed as part of the load.	All queries executed
`--exclude`	Query numbers or segments to be excluded from the load.	None excluded by default
`--executer`	Query execution engine. Available values: `scan`, `generic`.	`generic`
`--verbose` or `-v`	Print additional information to the screen during query execution.	

ClickBench-specific options

Name	Description	Default value
`--ext-queries <queries>` or `-q <queries>`	External queries to execute during the load, separated by semicolons.	
`--ext-queries-file <name>`	Name of the file containing external queries to execute during the load, separated by semicolons.	
`--ext-query-dir <name>`	Directory containing external queries for the load. Queries should be in files named `q[0-42].sql`.	
`--ext-results-dir <name>`	Directory containing external query results for comparison. Results should be in files named `q[0-42].sql`.	
`--check-canonical` or `-c`	Use special deterministic internal queries and compare the results against canonical ones.	

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`.

```
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
percentile,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     Upsert
time     Commit time
#       msg/s   MB/s    percentile,ms   percentile,msg  msg/s   MB/s    percentile,ms   percentile,ms
percentile,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-H workload

The workload is based on the TPC-H documentation, 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`.	`row`
`--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`
`--float-mode <value>`	Specifies the data type to use for fractional fields. Possible values are `float`, `decimal`, and `decimal_ydb`. `float` uses the `Float` type, `decimal` uses `Decimal` with dimensions specified by the test standard, and `decimal_ydb` uses `Decimal(22,9)` — the only type currently supported by YDB.	`float`
`--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.	
`--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.	

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

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
`--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
`--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`.	Not saved by default
`--query-settings <setting>`	Query execution settings. Each setting is added as a separate line at the beginning of each query. Use multiple times for multiple settings.	Not specified by default
`--include`	Query numbers or segments to be executed as part of the load.	All queries executed
`--exclude`	Query numbers or segments to be excluded from the load.	None excluded by default
`--executer`	Query execution engine. Available values: `scan`, `generic`.	`generic`
`--verbose` or `-v`	Print additional information to the screen during query execution.	

TPC-H-specific options

Name	Description	Default value
`--ext-query-dir <name>`	Directory with external queries for load execution. Queries should be in files named `q[1-23].sql`.	

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 documentation, 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`.	`row`
`--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`
`--float-mode <value>`	Specifies the data type to use for fractional fields. Possible values are `float`, `decimal`, and `decimal_ydb`. `float` uses the `Float` type, `decimal` uses `Decimal` with dimensions specified by the test standard, and `decimal_ydb` uses `Decimal(22,9)` — the only type currently supported by YDB.	`float`
`--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.	
`--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.	

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

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
`--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
`--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`.	Not saved by default
`--query-settings <setting>`	Query execution settings. Each setting is added as a separate line at the beginning of each query. Use multiple times for multiple settings.	Not specified by default
`--include`	Query numbers or segments to be executed as part of the load.	All queries executed
`--exclude`	Query numbers or segments to be excluded from the load.	None excluded by default
`--executer`	Query execution engine. Available values: `scan`, `generic`.	`generic`
`--verbose` or `-v`	Print additional information to the screen during query execution.	

TPC-DS-specific options

Name	Description	Default value
`--ext-query-dir <name>`	Directory with external queries for load execution. Queries should be in files named `q[1-99].sql`.	

Test data cleanup

Run cleanup:

```
ydb workload tpcds --path tpcds/s1 clean
```

The command has no parameters.

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.

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

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 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
```

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.

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.

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.

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.

C# (.NET)

```
dotnet add package Ydb.Sdk
```

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.

PHP

```
composer require ydb-platform/ydb-php-sdk
```

Node.JS

```
npm install ydb-sdk
```

Rust

Add to Cargo.toml last version of ydb crate.

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();

Node.js

Mode	Method
Anonymous	AnonymousAuthService()
Access Token	TokenAuthService(accessToken, database)
Metadata	MetadataAuthService(database)
Service Account Key	getSACredentialsFromJson(saKeyFile)
Static Credentials	StaticCredentialsAuthService(user, password, endpoint)
Determined by environment variables	getCredentialsFromEnv(entryPoint, database, logger)

Rust

Mode	Method
Anonymous	ydb::StaticToken("")
Access Token	ydb::StaticToken(token)
Metadata	ydb::GCEMetadata, ydb::YandexMetadata
Static Credentials	ydb::StaticCredentialsAuth
Service Account Key	not supported
Determined by environment variables	not supported
Execution of an external command	ydb.CommandLineYcToken (for example, for authentication using a Yandex.Cloud IAM token from the developer's computer `ydb::CommandLineYcToken.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-exchang`
`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.	

Parameterized queries

YDB supports and recommends the use of so-called parameterized queries. In such queries, the data is transmitted separately from the request body itself, and in the SQL query, special parameters are used to indicate the location of the data.

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 has the ability to cache the query plan for parameterized requests. This radically reduces CPU consumption and increases system throughput.
- The use of parameterized queries saves from vulnerabilities like SQL Injection.

YDB SDK automatically caches parameterized query plans by default, the setting `KeepInCache = true` is usually used for this.

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

Initializing a connection

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 = TDriverConfig()
    .SetEndpoint(opts.Endpoint)
    .SetDatabase(opts.Database)
    .SetAuthToken(GetEnv("YDB_TOKEN"));

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:

```
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();
```

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:

```
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
    }
});
```

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:

```
driver.topic_client.create_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());
```

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

```
info = 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();
```

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

```
driver.topic_client.drop_topic(topic_path)
```

Java

```
topicClient.dropTopic(topicPath);
```

C#

```
await topicClient.DropTopic(topicName);
```

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:

```
TString producerAndGroupID = "group-id";
auto settings = 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

```
writer = driver.topic_client.writer(topic_path)
```

Java (sync)

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);
```

Writer should be initialized after it is created. There are two methods to do that:

- `init()`: non-blocking, launches initialization in background and doesn't wait for it to finish.

  ```
  writer.init();
  ```

- `initAndWait()`: blocking, launches initialization and waits for it to finish.
  If an error occurs during this process, exception will be thrown.

  ```
  try {
      writer.initAndWait();
      logger.info("Init finished succsessfully");
  } catch (Exception exception) {
      logger.error("Exception while initializing writer: ", exception);
      return;
  }
  ```

Java (async)

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();
```

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
    TMaybe<TWriteSessionEvent::TEvent> event = session->GetEvent(/*block=*/true);

    if (auto* readyEvent = std::get_if<TWriteSessionEvent::TReadyToAcceptEvent>(&*event)) {
        session->Write(std::move(event.ContinuationToken), "This is yet another message.");

    } else if (auto* ackEvent = std::get_if<TWriteSessionEvent::TAcksEvent>(&*event)) {
        std::cout << ackEvent->DebugString() << std::endl;

    } else if (auto* closeSessionEvent = std::get_if<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.

```
# 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 send
                                # binary data.
writer.write(["mess-1", "mess-2"])  # This line 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 properties.
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(),
  ])
```

Java (sync)

Method `send` blocks until a message is put into writers sending queue.
Putting a message into this queue means that the writer will do its best to deliver it.
For example, if a writing session will be accidentally closed, the writer will reconnect and try to resend this message on a new session.
But putting a message into message queue has no guarantees that this message will be written.
For example, there could be errors that will lead to writer shutdown before messages from the queue are sent.
If you have to be sure for each message that it is written, use async writer and check status returned by `send` method.

```
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", timeoutSeconds);
} catch (InterruptedException | ExecutionException exception) {
    logger.error("Couldn't put the message into sending queue due to exception: ", exception);
}
```

Java (async)

Method `send` puts a message into writer's sending queue.
Method returns `CompletableFuture<WriteAck>` which allows checking if the message was really written.
In case if the queue is full, `QueueOverflowException` exception will be thrown.
It is a way to signal a user that writing speed should be slowed down.
In this case a message write should be skipped or retried with exponential backoff.
Client buffer size can be also increased (`setMaxSendBufferMemorySize`) to be able to store more messages in memory before this exception is thrown.

```
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>
```

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 = TWriteSessionSettings()
  // other settings are set here
  .EventHandlers(
    TWriteSessionSettings::TEventHandlers()
      .AcksHandler(
        [&](TWriteSessionEvent::TAcksEvent& event) {
          for (const auto& ack : event.Acks) {
            if (ack.State == 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:

- `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 (sync)

Blocking method `flush()` waits until all the messages previously written to the internal buffer are acknowledged:

```
for (byte[] message : messages) {
    writer.send(Message.of(message));
}
writer.flush();
```

Java (async)

`send` method returns `CompletableFuture<WriteAck>`.
Its successful completion means that the fact that this message is written is confirmed by server.
`WriteAck` struct 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);
```

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 = 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();
```

Writing messages in no-deduplication mode

C++

Using message metadata feature

You can provide some metadata for any particular message when writing. This metadata can be a list of up to 100 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 = TWriteSessionSettings()
    .Path(myTopicPath)
//set all oter settings;
;

auto session = topicClient.CreateWriteSession(settings);

TMaybe<TWriteSessionEvent::TEvent> event = session->GetEvent(/*block=*/true);
TWriteMessage message("This is yet another message").MessageMeta({
    {"meta-key", "meta-value"},
    {"another-key", "value"}
});

if (auto* readyEvent = std::get_if<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()) } }
);
```

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.

```
  auto tableSession = tableClient.GetSession().GetValueSync().GetSession();
  auto transaction = tableSession.BeginTransaction().GetValueSync().GetTransaction();
  NYdb::NTopic::TWriteMessage writeMessage("message");

  topicSession->Write(std::move(writeMessage), transaction);
  transaction.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 (sync)

Example on GitHub

Transaction can be set in the `SendSettings` argument of the `send` method while sending a message.
Such a message will be written on the transaction 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
writer.flush();
```

```
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.

Java (async)

Example on GitHub

Transaction can be set in the `SendSettings` argument of the `send` method while sending a message.
Such a message will be written on the transaction 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.

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 = 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 (sync)

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, it has to be initialized. Sync reader has two methods for this:

- `init()`: non-blocking, launches initialization in background and does not wait for it to finish.

```
reader.init();
```

- `initAndWait()`: blocking, launches initialization and waits for it to finish.
  If an error occurs during this process, exception will be thrown.

```
try {
    reader.initAndWait();
    logger.info("Init finished succsessfully");
} catch (Exception exception) {
    logger.error("Exception while initializing reader: ", exception);
    return;
}
```

Java (async)

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 async reader, `ReadEventHandlersSettings` also have to be provided with an implementation of `ReadEventHandler`.
It describes how events should be handled during reading.

```
ReadEventHandlersSettings handlerSettings = ReadEventHandlersSettings.newBuilder()
    .setEventHandler(new Handler())
    .build();
```

Optionally, an executor for message handling can be also provided in `ReadEventHandlersSettings`.
To implement a Handler, default abstract class `AbstractReadEventHandler` can be used.
It is enough to override the `onMessages` method that describes message handling. Implementation 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();
```

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 = TReadSessionSettings()
    .ConsumerName("my-consumer")
    .AppendTopics("my-topic")
    .AppendTopics(
        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();
```

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.

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

```
while True:
    message = reader.receive_message()
    process(message)
```

Java (sync)

To read messages one-by-one without commit just do not call the `commit` method on messages:

```
while(true) {
    Message message = reader.receive();
    process(message);
}
```

Java (async)

Reading messages one-by-one is not supported in async Reader.

C#

```
try
{
    while (!readerCts.IsCancellationRequested)
    {
        var message = await reader.ReadAsync(readerCts.Token);

        logger.LogInformation("Received message: [{MessageData}]", message.Data);
    }
}
catch (OperationCanceledException)
{
}
```

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 = TReadSessionSettings()
    .EventHandlers_.SimpleDataHandlers(
        [](TReadSessionEvent::TDataReceivedEvent& event) {
            std::cout << "Get data event " << DebugString(event);
        }
    );

auto session = topicClient.CreateReadSession(settings);

// Wait SessionClosed event.
ReadSession->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

```
while True:
  batch = reader.receive_batch()
  process(batch)
```

Java (sync)

Reading messages in batches is not supported in sync Reader.

Java (async)

To read messages without commit just do not call the `commit` method:

```
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)
{
}
```

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

```
while True:
    message = reader.receive_message()
    process(message)
    reader.commit(message)
```

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

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)
{
}
```

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 = TReadSessionSettings()
    .EventHandlers_.SimpleDataHandlers(
        [](TReadSessionEvent::TDataReceivedEvent& event) {
            std::cout << "Get data event " << DebugString(event);
        }
        , /* commitDataAfterProcessing = */true
    );

auto session = topicClient.CreateReadSession(settings);

// Wait SessionClosed event.
ReadSession->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

```
while True:
  batch = 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 (sync)

Not relevant due to sync reader only reading messages one by one.

Java (async)

In `onMessage` handler whole message batch in `DataReceivedEvent` can be committed:

```
@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)
{
}
```

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(
    [](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.Connection)
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.

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`.

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(),
)
```

Reading in a transaction

C++

Before reading messages, the client code must pass a transaction object reference to the reading session settings.

```
ReadSession->WaitEvent().Wait(TDuration::Seconds(1));

auto tableSettings = NYdb::NTable::TTxSettings::SerializableRW();
auto transactionResult = TableSession->BeginTransaction(tableSettings).GetValueSync();
auto Transaction = transactionResult.GetTransaction();

NYdb::NTopic::TReadSessionGetEventSettings topicSettings;
topicSettings.Block(false);
topicSettings.Tx(Transaction);

auto events = ReadSession->GetEvents(topicSettings);

for (auto& event : events) {
    // process the event and write results to a table
}
```

```
NYdb::NTable::TCommitTxSettings commitSettings;
auto commitResult = Transaction.Commit(commitSettings).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<TStopPartitionSessionEvent> stopPartitionSession;

auto events = ReadSession->GetEvents(topicSettings);

for (auto& event : events) {
    if (auto* e = std::get_if<TStopPartitionSessionEvent>(&event) {
        stopPartitionSessionEvent = std::move(*e);
    } else {
        // process the event and write results to a table
    }
}

NYdb::NTable::TCommitTxSettings commitSettings;
auto commitResult = Transaction.Commit(commitSettings).GetValueSync();

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 transaction
    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.

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)
```

Python (asyncio)

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.

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 (sync)

Example on GitHub

A transaction can be set in `ReceiveSettings` for the `receive` method:

```
Message message = reader.receive(ReceiveSettings.newBuilder()
        .setTransaction(transaction)
        .build());
```

A message received this way will be automatically committed with the provided transaction and shouldn't be committed directly.
The `receive` method sends the `sendUpdateOffsetsInTransaction` request on the server to link the message offset with this transaction and blocks until a response is received.

> **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.

Java (async)

Example on GitHub

In the `onMessages` callback, one or more messages can be linked with a transaction.
To do that request `reader.updateOffsetsInTransaction` should be called. And transaction should not be committed until a response is received.
This method needs a partition offsets list as a parameter.
Such a list can be constructed manually or using the helper method `getPartitionOffsets()` that `Message` and `DataReceivedEvent` both provide.

```
@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.

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<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.

```
while True:
  batch = reader.receive_batch()
  process(batch)
  reader.commit(batch)
```

Java (sync)

Not relevant due to not being possible to change the way of handling such events.
Client will automatically respond to server that it is ready to stop.

Java (async)

`onStopPartitionSession(StopPartitionSessionEvent event)` handler should be overridden to handle this event:

```
@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();
}
```

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<TReadSessionEvent::TPartitionSessionClosedEvent>(&*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.

```
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)
```

Java (sync)

Not relevant due to not being possible to change the way of handling such events.

Java (async)

```
@Override
public void onPartitionSessionClosed(PartitionSessionClosedEvent event) {
    logger.info("Partition session {} is closed.", event.getPartitionSession().getPartitionId());
}
```

Topic autoscaling

Python

Autoscaling of a topic can be enabled during its creation using the `auto_partitioning_settings` argument of `create_topic`:

```
    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.

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.

Python

Commit outside the reader is done using the `topic_client.commit_offset` method:

```
driver.topic_client.commit_offset(
    topic_path,
```

```
    consumer_name,
    partition_id,
    offset,
)
```

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`.

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.

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.

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();
```

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>()`.)

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();
```

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`.

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();
```

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

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'	–	+	+	+	+	–	–	–
Load balancer initialization through Discovery/ListEndpoints	+	+	+	+	+	+	+	+
Anonymous (default)	+	+	+	+	+	+	+	+
Int/Uint(8,16,32,64)	+	+	+	+	+	+	+	+
Int128, UInt128	–	–	–	–	–	–	–	–
ListEndpoints	+	–	+	–	+	–	–	–
Session pool	+	+	+	+	–	–	–	–
MakeDirectory	+	+	+	+	+	–	–	+
Session pool	+	+	+	+	+	+	+	+
Consumed Units from metadata	+	+	+	+	+	–	–	–
ExecuteYql	+	+	+	–	–	–	–	+
CreateNode	+	–	+	+	–	–	–	–
CreateTopic	+	+	+	+	+	+	–	–
CreateResource	+	–	+	–	–	–	–	–

Feature	C++	Python	Go	Java	NodeJS	C#	Rust	PHP
SDK event logging	–	–	+	+	+	+	+	+
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	–	+	+	+	–	–	–	–
DescribeTableOptions		+	+	+	–	–	–	–
StreamExecuteScanQuery		+	+	+	+	+	+	+

StreamReadTable	+	+	+	+	+	+	–	+

Code	Status	Retryability	Backoff strategy	Recreate session
400000	SUCCESS	–	–	–
400010	BAD_REQUEST	non-retryable	–	no
400020	UNAUTHORIZED	non-retryable	–	no
400030	INTERNAL_ERROR	non-retryable	–	no
400040	ABORTED	retryable	fast	no
400050	UNAVAILABLE	retryable	fast	no
400060	OVERLOADED	retryable	slow	no
400070	SCHEME_ERROR	non-retryable	–	no
400080	GENERIC_ERROR	non-retryable	–	no
400090	TIMEOUT	non-retryable	–	no
400100	BAD_SESSION	retryable	instant	yes
400120	PRECONDITION_FAILED	non-retryable	–	no
400130	ALREADY_EXISTS	non-retryable	–	no
400140	NOT_FOUND	non-retryable	–	no
400150	SESSION_EXPIRED	retryable	instant	yes
400160	CANCELLED	non-retryable	–	no
400170	UNDETERMINED	conditionally-retryable	fast	no

400180	UNSUPPORTED	non-retryable	–	no
400190	SESSION_BUSY	retryable	fast	yes
400200	EXTERNAL_ERROR	non-retryable	–	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 Embedded 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

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	fast	yes
2	UNKNOWN	non-retryable	–	yes
3	INVALID_ARGUMENT	non-retryable	–	yes
4	DEADLINE_EXCEEDED	conditionally-retryable	fast	yes
5	NOT_FOUND	non-retryable	–	yes
6	ALREADY_EXISTS	non-retryable	–	yes
7	PERMISSION_DENIED	non-retryable	–	yes
8	RESOURCE_EXHAUSTED	retryable	slow	no
9	FAILED_PRECONDITION	non-retryable	–	yes
10	ABORTED	retryable	instant	yes
11	OUT_OF_RANGE	non-retryable	–	no
12	UNIMPLEMENTED	non-retryable	–	yes
13	INTERNAL	conditionally-retryable	fast	yes
14	UNAVAILABLE	conditionally-retryable	fast	yes
15	DATA_LOSS	non-retryable	–	yes

16	UNAUTHENTICATED	non-retryable	–	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.

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();
```

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);
};
```

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:

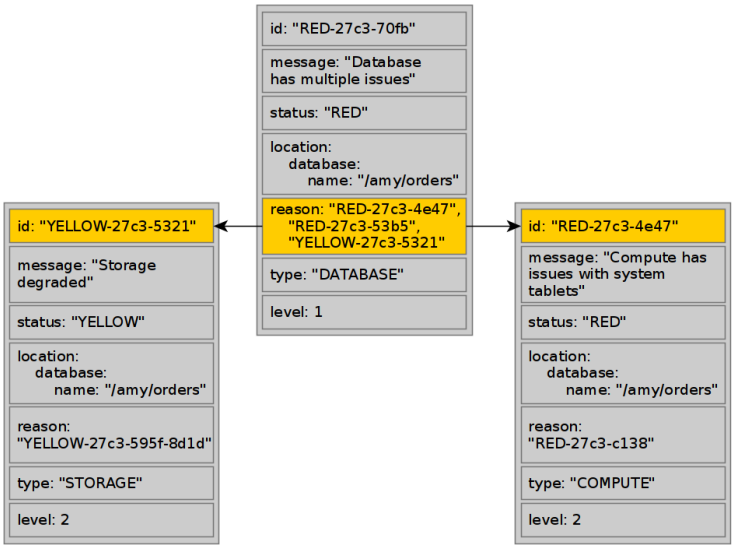
```
message IssueLog {
    string id = 1;
    StatusFlag.Status status = 2;
    string message = 3;
    Location location = 4;
    repeated string reason = 5;
    string type = 6;
    uint32 level = 7;
}
```

Description of fields in the response

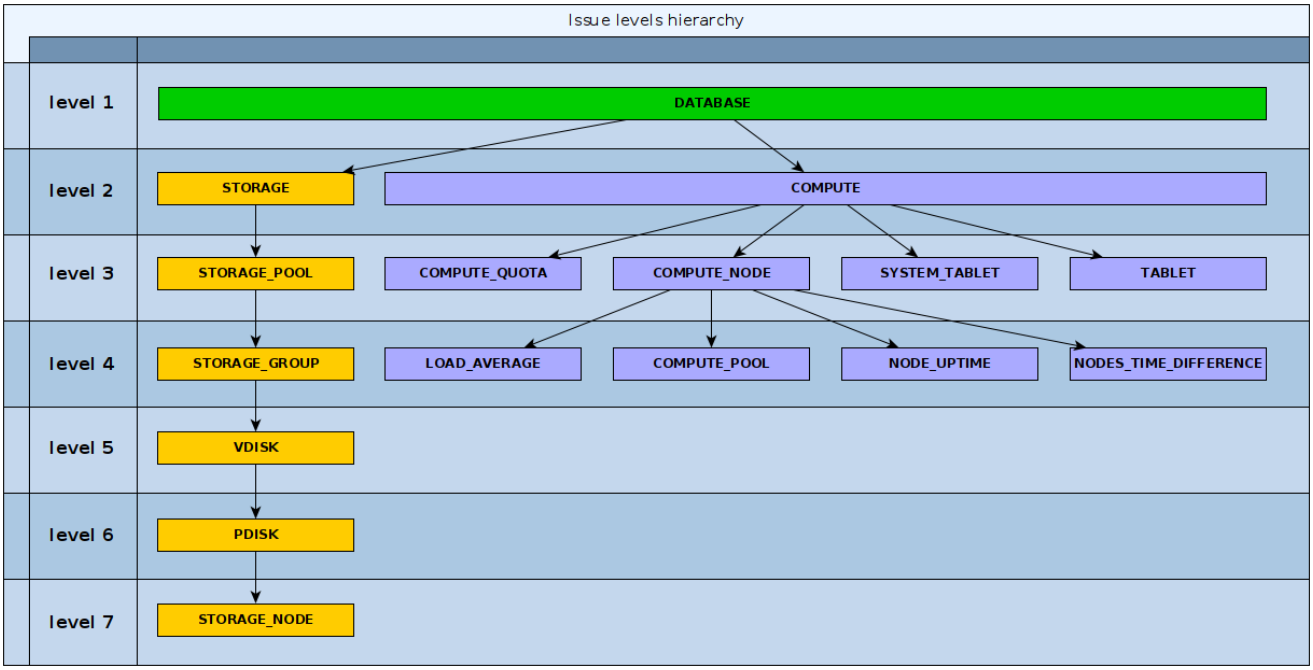
Field	Description
`self_check_result`	enum field which contains the database check result
`issue_log.id`	A unique issue ID within this response.
`issue_log.status`	enum field which contains the issue status
`issue_log.message`	Text that describes the issue.
`issue_log.location`	Location of the issue. 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 issuehs 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. It 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:

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 Embedded 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 Embedded 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 Embedded 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 Embedded 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 Embedded 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 Embedded 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 Embedded 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 Embedded 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 Embedded 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 Embedded 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 Embedded 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 Embedded 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 Embedded 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
  }
}
```

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

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

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>"
      ]
    }
  }
}
```

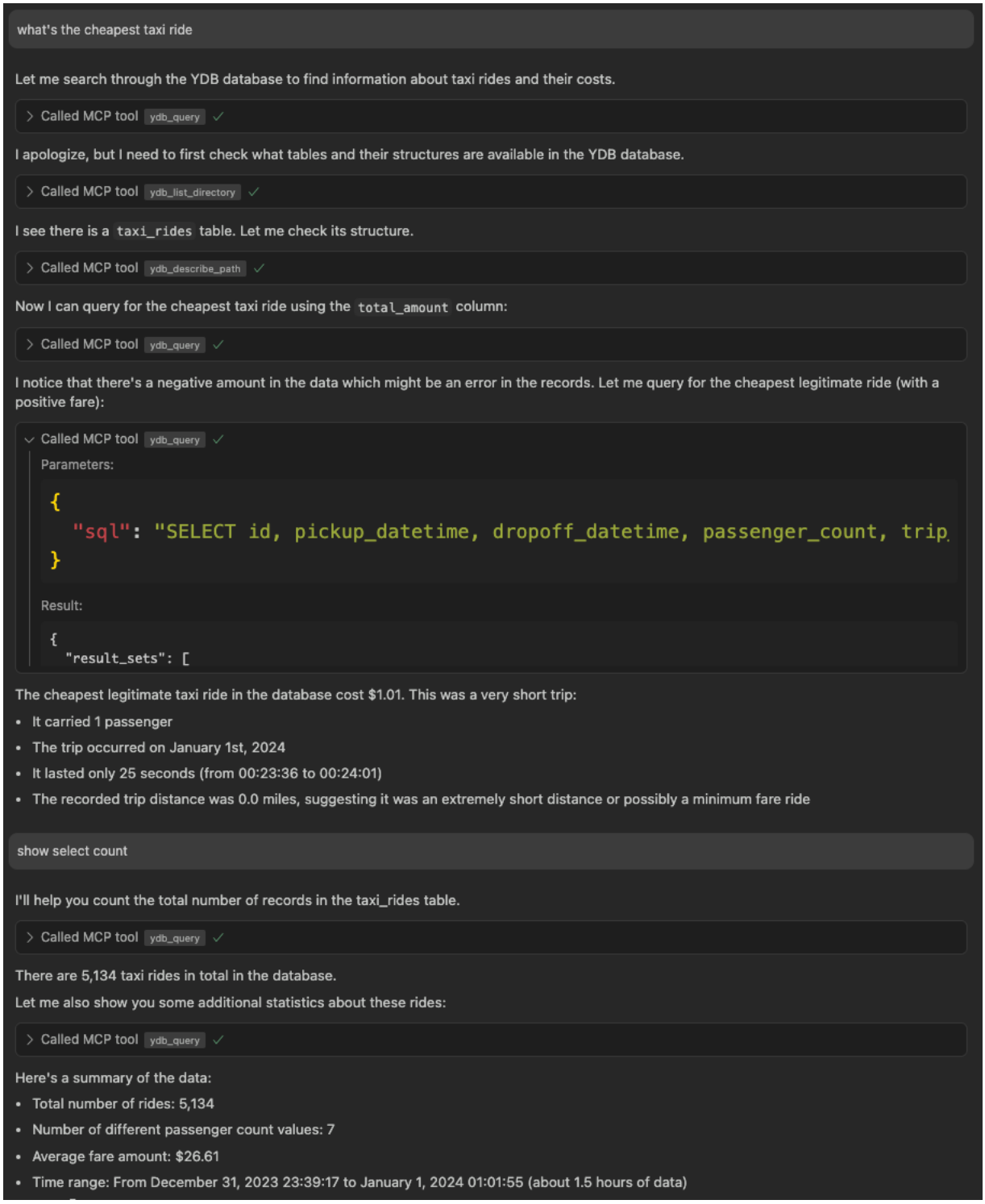
pip

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.

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 connection = new YdbConnection("Host=localhost;Port=2136;Database=/local;MaxSessionPool=50");
await connection.OpenAsync();

var ydbCommand = connection.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}, title:
{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.

Connections

A connection to YDB is established using `YdbConnection`.

1. **Using an empty connection**:

   The following code creates a connection with the default settings:

   ```
   await using var ydbConnection = new YdbConnection("");
   await ydbConnection.OpenAsync();
   ```

   This option creates a connection to the database at the URL `grpc://localhost:2136/local` with anonymous authentication.

2. **Using the constructor with a connection string**:

   In the following example, a connection is created using a connection string in ADO.NET:

   ```
   await using var ydbConnection = new YdbConnection(
       "Host=database-sample-grpc;Port=2135;Database=/root/database-sample");
   await ydbConnection.OpenAsync();
   ```

   In this case, the connection is established at the URL `grpc://database-sample-grpc:2135/root/database-sample`. The supported set of settings is explained on the connection parameters page.

3. **Using the constructor with a `YdbConnectionStringBuilder` argument**:

   The example using `YdbConnectionStringBuilder` is demonstrated in the code below:

   ```
   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();
   ```

   `YdbConnectionStringBuilder` supports additional configuration beyond the connection string, such as logging, advanced authentication options.

Pooling

Opening and closing a logical connection to YDB is an expensive and time-consuming process. Therefore, connections to YDB are pooled. Closing or disposing of a connection does not close the underlying logical connection; rather, it returns it to a pool managed by `Ydb.Sdk.Ado`. When a connection is needed again, a pooled connection is returned. This makes opening and closing operations extremely fast. Do not hesitate to open and close connections often if necessary, rather than keeping a connection open unnecessarily for a long period of time.

ClearPool

Closes idle connections immediately. Active connections close when returned.

```
YdbConnection.ClearPool(ydbConnection)
```

ClearAllPools

Closes all idle connections across all pools. Active connections close on return.

```
YdbConnection.ClearAllPools()
```

Data Source

Starting with .NET 7.0, the starting point for any database operation is DbDataSource.

The simplest way to create a data source is the following:

```
await using var dataSource = new YdbDataSource("Host=localhost;Port=2136;Database=/local");
```

Or

```
var ydbConnectionBuilder = new YdbConnectionStringBuilder
{
    Host = "localhost",
    Port = 2136,
    Database = "/local",
    UseTls = false
};

await using var dataSource = new YdbDataSource(ydbConnectionBuilder);
```

Basic SQL Execution

Once you have a `YdbConnection`, an `YdbCommand` can be used to execute SQL against it:

```
await using var command = dataSource.CreateCommand("SELECT some_field FROM some_table")
await using var reader = await command.ExecuteReaderAsync();

while (await reader.ReadAsync())
{
    Console.WriteLine(reader.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 command = dataSource.CreateCommand("INSERT INTO some_table (some_field) VALUES ('Hello YDB!'u)");
await command.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 connection = new YdbConnection(_cmdOptions.SimpleConnectionString);
await connection.OpenAsync();

var ydbCommand = connection.CreateCommand();
ydbCommand.CommandText = """
                         DECLARE $series_id AS Uint64;
                         DECLARE $season_id AS Uint64;
                         DECLARE $limit_size AS Uint64;

                         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 will be determined according to the DbType or the .NET type of the value itself.

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

To create a client transaction, use the standard ADO.NET `ydbConnection.BeginTransaction()` method.

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`.

Consider the following example of using a transaction:

```
await using var connection = await dataSource.OpenConnectionAsync();
await using var transaction = await connection.BeginTransactionAsync();

await using var command1 = new YdbCommand(connection) { CommandText = "...", Transaction = transaction };
await command1.ExecuteNonQueryAsync();

await using var command2 = new YdbCommand(connection) { CommandText = "...", Transaction = transaction };
await command2.ExecuteNonQueryAsync();

await transaction.CommitAsync();
```

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.

Error Handling

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`	Specifies the YDB server host	`localhost`
`Port`	Specifies the YDB server port	`2136`
`Database`	Specifies the database name	`/local`
`User`	Specifies the username	Not defined
`Password`	Specifies the user password	Not defined

Security and Encryption

Parameter	Description	Default value
`UseTls`	Indicates whether to use the TLS protocol (`grpcs` or `grpc`)	`false`
`RootCertificate`	Specifies the path to the trusted server TLS certificate. If this parameter is set, the `UseTls` parameter will be forced to be `true`	Not defined

Pooling

Parameter	Description	Default value
`MaxSessionPool`	Specifies the maximum session pool size	`100`

Keepalive

Parameter	Description	Default value
`KeepAlivePingDelay`	The client sends a keep-alive ping to the server if it doesn't receive any frames on a connection for this period of time. This property is used together with `KeepAlivePingTimeout` to check whether the connection is broken. The delay value must be greater than or equal to `1` second. Set to `0` to disable the keep-alive ping	`10` seconds
`KeepAlivePingTimeout`	Keep-alive pings are sent when a period of inactivity exceeds the configured `KeepAlivePingDelay` value. The client closes the connection if it doesn't receive any frames within the timeout. The timeout must be greater than or equal to `1` second. Set to `0` to disable the keep-alive ping timeout	`10` seconds

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.

YDB type	.NET type
`Bool`	`bool`
`Text` (synonym of `Utf8`)	`string`
`Bytes` (synonym of `String`)	`byte[]`
`Uint8`	`byte`
`Uint16`	`ushort`
`Uint32`	`uint`
`Uint64`	`ulong`
`Int8`	`sbyte`
`Int16`	`short`
`Int32`	`int`
`Int64`	`long`
`Float`	`float`
`Double`	`double`
`Date`	`DateTime`
`Datetime`	`DateTime`
`Timestamp`	`DateTime`
`Decimal(22,9)`	`Decimal`
`Json`	`string`
`JsonDocument`	`string`
`Yson`	`byte[]`

Type Mapping Table for Writing

YDB type	DbType	.NET type
`Bool`	`Boolean`	`bool`
`Text` (synonym of `Utf8`)	`String`, `AnsiString`, `AnsiStringFixedLength`, `StringFixedLength`	`string`
`Bytes` (synonym of `String`)	`Binary`	`byte[]`
`Uint8`	`Byte`	`byte`
`Uint16`	`UInt16`	`ushort`
`Uint32`	`UInt32`	`uint`

`Uint64`	`UInt64`	`ulong`
`Int8`	`SByte`	`sbyte`
`Int16`	`Int16`	`short`
`Int32`	`Int32`	`int`
`Int64`	`Int64`	`long`
`Float`	`Single`	`float`
`Double`	`Double`	`double`
`Date`	`Date`	`DateTime`
`Datetime`	`DateTime`	`DateTime`
`Timestamp`	`DateTime2` (for .NET type `DateTime`), `DateTimeOffset` (for .NET type `DateTimeOffset`)	`DateTime`, `DateTimeOffset`
`Decimal(22,9)`	`Decimal`, `Currency`	`decimal`

Note

It's important to understand that if the `DbType` is not specified, the parameter will be inferred from the `System.Type`.

You can also specify any YDB type using the constructors from `Ydb.Sdk.Value.YdbValue`. For example:

```
var parameter = new YdbParameter("$parameter", YdbValue.MakeJsonDocument("{\"type\": \"jsondoc\"}"));
```

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

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=file:~/<myca>.cer`
   - A cloud database instance with a token:
     `jdbc:ydb:grpcs://<host>:2135/<path/to/database>?token=file:~/my_token`
   - A cloud database instance with a service account:
     `jdbc:ydb:grpcs://<host>:2135/<path/to/database>?saFile=file:~/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` 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:

- `saFile` — service account key for authentication. The valid value is either the content of the JSON file or a file reference.
- `iamEndpoint` — custom IAM endpoint for authentication using a service account key.
- `token` — token value for authentication. The valid value is either the token content or a token file reference.
- `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.
- `secureConnection` — indicates whether to use TLS. Valid values are:
  - `true` — enforce TLS.
  - `false` — do not enforce TLS.

  The primary way to indicate whether a connection is secure or not is by using the `grpcs://` scheme for secure connections and `grpc://` for insecure connections in the JDBC URL. This property allows overriding it.
- `secureConnectionCertificate` — custom CA certificate for TLS connections. The valid value is either the certificate content or a certificate file reference.

> **Note**
>
> File references for `saFile`, `token`, or `secureConnectionCertificate` must be prefixed with the `file:` URL scheme, for example:
>
> - `saFile=file:~/mysakey1.json`
> - `token=file:/opt/secret/token-file`
> - `secureConnectionCertificate=file:/etc/ssl/cacert.cer`

Using the QueryService mode

By default, the JDBC driver currently uses a legacy API for running queries to be compatible with a broader range of YDB versions. However, that API has some extra limitations. To turn off this behavior and use a modern API called "Query Service", add the `useQueryService=true` property to the JDBC URL.

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=file:~/<myca>.cer`
- A cloud database instance with a token:
  `jdbc:ydb:grpcs://<host>:2135/<path/to/database>?token=file:~/my_token`
- A cloud database instance with a service account:
  `jdbc:ydb:grpcs://<host>:2135/<path/to/database>?saFile=file:~/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`

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` or `SASL_SSL/PLAIN` authentication mechanism.

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.

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

Consider the following limitations of using the Kafka API for reading:

- No support for the check.crcs option.
- Only one partition assignment strategy - `roundrobin`.
- No reading without a pre-created consumer group.

Therefore, in the consumer configuration, you must always specify the **consumer group name** and the parameters:

- `check.crcs=false`
- `partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRobinAssignor`

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

When working with Kafka, Apache Spark does not use any of the Kafka API features that are currently not supported in YDB Topics. Therefore, all features of Spark-Kafka integrations should work through the YDB Topics Kafka API.

```
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

> **Note**
>
> Currently, not all functionality of Flink is supported for reading and writing. The following limitations exist:
>
> - Exactly-once functionality via the Kafka API is not supported at the moment because transaction support in the Kafka API is still under development.
> - Subscription to topics using a pattern is currently unavailable.
> - Using message `CreateTime` as a watermark is not available at the moment because the current read time is used instead of `CreateTime` (this will be fixed in future versions).

```
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.

Frequent problems and solutions

Unexpected error in join group response

Full text of an exception:

```
Unexpected error in join group response: This most likely occurs because of a request being malformed by the
client library or the message was sent to an incompatible broker. See the broker logs for more details.
```

Most likely it means that a consumer group is not specified or, if specified, it does not exist in the YDB cluster.

Solution: create a consumer group in YDB using CLI or SDK.

Writing

> **Note**
>
> Using Kafka transactions when writing via Kafka API is currently not supported. Transactions are only available when using the YDB Topic API.
>
> Otherwise, writing to Apache Kafka and YDB Topics through Kafka API is no different.

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

When working with Kafka, Apache Spark does not use any of the Kafka API features that are currently not supported in YDB Topics. Therefore, all features of Spark-Kafka integrations should work through the YDB Topics Kafka API.

```
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

Note

Currently, not all functionality of Flink is supported for reading and writing. The following limitations exist:

- Exactly-once functionality via the Kafka API is not supported at the moment because transaction support in the Kafka API is still under development.
- Subscription to topics using a pattern is currently unavailable.
- Using message `CreateTime` as a watermark is not available at the moment because the current read time is used instead of `CreateTime` (this will be fixed in future versions).

```
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 9092
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 username=\"
<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 required
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());
  }
}
```

Kafka API constraints

YDB supports Apache Kafka protocol version 3.4.0 with the following constraints:

1. Only SASL/PLAIN authentication is supported.
2. Compacted topics are not supported. Consequently, Kafka Connect, Schema Registry, and Kafka Streams do not work over the Kafka API in YDB Topics.
3. Message compression is not supported.
4. Transactions are not supported.
5. DDL operations are not supported. For DDL operations, use YDB SDK or YDB CLI.
6. CRC checks are not supported.
7. Kafka Connect works only in a standalone mode (single-worker mode).
8. If auto-partitioning is enabled on a topic, you cannot write to or read from such a topic using the Kafka API.

`auth_config` configuration section

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 an 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 an 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"
    use_tls:
      enable: true
      ca_cert_file: "/path/to/ca.pem"
      cert_require: DEMAND
  ldap_authentication_domain: "ldap"
  scheme: "ldap"
  requested_group_attribute: "memberOf"
  extended_settings:
      enable_nested_groups_search: true

  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`

gure IAM

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`

`client_certificate_authorization` configuration section

Database node authentication within the YDB cluster ensures that service connections between cluster nodes are assigned the correct security identifiers, or SIDs. The process of database node authentication applies to connections that use the gRPC protocol and provide functions for registering nodes in the cluster, as well as for accessing configuration information. SIDs assigned to connections are considered when checking the authorization rules that apply to the corresponding gRPC service calls.

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"]
```

`security_config` configuration section

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
  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>
  register_dynamic_node_allowed_sids: <list of SIDs that are allowed to register database nodes in the cluster>

  # 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:

Parameter	Description
`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 Embedded UI). No changes are allowed.

`monitoring_allowed_sids`	The list of SIDs with the operator 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 Embedded UI.
`administration_allowed_sids`	The list of SIDs with the administrator access level. This level grants privileges to administer the YDB cluster and its databases.
`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 grants its access level to any user, including anonymous users.

If all three lists are empty, any user has the administrative access level.

For a secure YDB deployment, plan the access model beforehand and define the group lists before starting the cluster for the first time.

The access level lists can include the SIDs of users or user groups. A user belongs to an access level list if the list includes the SID of the user or the SID of a group to which the user or its subgroup (recursively) belongs.

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.

Note

Access level lists are 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).
- Each of the `viewer_allowed_sids`, `monitoring_allowed_sids`, and `administration_allowed_sids` lists adds privileges to the access subject. For the maximum level of privileges, an access subject must be added to all three access level lists.
- Adding an access subject to the `monitoring_allowed_sids` or `administration_allowed_sids` list without adding it to `viewer_allowed_sids` has no effect.

For example:

- An operator (the SID of the user or the group to which the user belongs) must be added to `viewer_allowed_sids` and `monitoring_allowed_sids`.
- An administrator must be added to `viewer_allowed_sids`, `monitoring_allowed_sids`, and `administration_allowed_sids`.

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`

TLS configuration

YDB supports data-in-transit encryption, and each network protocol can have different TLS settings. This documentation section provides a reference for configuring TLS in YDB.

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"
```

PostgreSQL wire protocol

YDB exposes a separate network port for the PostgreSQL wire protocol. This protocol is used for external communication with client applications initially designed to work with PostgreSQL.

Example of enabling TLS for the PostgreSQL wire protocol:

```
local_pg_wire_config:
    ssl_certificate: "/opt/ydb/certs/node.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 Embedded 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.

Metrics reference

Resource usage metrics

Metric name Type, units of measurement	Description Labels
`resources.storage.used_bytes` `IGAUGE`, bytes	The size of user and service data stored in distributed network storage. `resources.storage.used_bytes` = `resources.storage.table.used_bytes` + `resources.storage.topic.used_bytes`.
`resources.storage.table.used_bytes` `IGAUGE`, bytes	The size of user and service data stored by tables in distributed network storage. Service data includes the data of the primary, secondary indexes and vector indexes.
`resources.storage.topic.used_bytes` `IGAUGE`, bytes	The size of storage used by topics. This metric sums the `topic.storage_bytes` values of all topics.
`resources.storage.limit_bytes` `IGAUGE`, bytes	A limit on the size of user and service data that a database can store in distributed network storage.

API metrics

Metric name Type, units of measurement	Description Labels
`api.grpc.request.bytes` `RATE`, bytes	The size of queries received by the database in a certain period of time. Labels: - *api_service*: The name of the gRPC API service, such as `table`. - *method*: The name of a gRPC API service method, such as `ExecuteDataQuery`.
`api.grpc.request.dropped_count` `RATE`, pieces	The number of requests dropped at the transport (gRPC) layer due to an error. Labels: - *api_service*: The name of the gRPC API service, such as `table`. - *method*: The name of a gRPC API service method, such as `ExecuteDataQuery`.
`api.grpc.request.inflight_count` `IGAUGE`, pieces	The number of requests that a database is simultaneously handling in a certain period of time. Labels: - *api_service*: The name of the gRPC API service, such as `table`. - *method*: The name of a gRPC API service method, such as `ExecuteDataQuery`.
`api.grpc.request.inflight_bytes` `IGAUGE`, bytes	The size of requests that a database is simultaneously handling in a certain period of time. Labels: - *api_service*: The name of the gRPC API service, such as `table`. - *method*: The name of a gRPC API service method, such as `ExecuteDataQuery`.
`api.grpc.response.bytes` `RATE`, bytes	The size of responses sent by the database in a certain period of time. Labels: - *api_service*: The name of the gRPC API service, such as `table`. - *method*: The name of a gRPC API service method, such as `ExecuteDataQuery`.
`api.grpc.response.count` `RATE`, pieces	The number of responses sent by the database in a certain period of time. Labels: - *api_service*: The name of the gRPC API service, such as `table`. - *method*: The name of a gRPC API service method, such as `ExecuteDataQuery`. - *status* is the request execution status. See a more detailed description of statuses under Error Handling.
`api.grpc.response.dropped_count` `RATE`, pieces	The number of responses dropped at the transport (gRPC) layer due to an error. Labels: - *api_service*: The name of the gRPC API service, such as `table`. - *method*: The name of a gRPC API service method, such as `ExecuteDataQuery`.

`api.grpc.response.issues` `RATE`, pieces	The number of errors of a certain type arising in the execution of a request over a certain period of time. Tags: - *issue_type* is the error type wth the only value being `optimistic_locks_invalidation`. For more on lock invalidation, review Transactions and requests to YDB.

Session metrics

Metric name Type, units of measurement	Description Labels
`table.session.active_count` `IGAUGE`, pieces	The number of sessions started by clients and running at a given time.
`table.session.closed_by_idle_count` `RATE`, pieces	The number of sessions closed by the DB server in a certain period of time due to exceeding the lifetime allowed for an idle session.

Transaction processing metrics

You can analyze a transaction's execution time using a histogram counter. The intervals are set in milliseconds. The chart shows the number of transactions whose duration falls within a certain time interval.

Metric name Type, units of measurement	Description Labels
`table.transaction.total_duration_milliseconds` `HIST_RATE`, pieces	The number of transactions with a certain duration on the server and client. The duration of a transaction is counted from the point of its explicit or implicit start to committing changes or its rollback. Includes the transaction processing time on the server and the time on the client between sending different requests within the same transaction. Labels: - *tx_kind*: The transaction type, possible values are `read_only`, `read_write`, `write_only`, and `pure`.
`table.transaction.server_duration_milliseconds` `HIST_RATE`, pieces	The number of transactions with a certain duration on the server. The duration is the time of executing requests within a transaction on the server. Does not include the waiting time on the client between sending separate requests within a single transaction. Labels: -*tx_kind*: The transaction type, possible values are `read_only`, `read_write`, `write_only`, and `pure`.
`table.transaction.client_duration_milliseconds` `HIST_RATE`, pieces	The number of transactions with a certain duration on the client. The duration is the waiting time on the client between sending individual requests within a single transaction. Does not include the time of executing requests on the server. Labels: - *tx_kind*: The transaction type, possible values are `read_only`, `read_write`, `write_only`, and `pure`.

Query processing metrics

Metric name Type, units of measurement	Description Labels
`table.query.request.bytes` `RATE`, bytes	The size of YQL query text and parameter values to queries received by the database in a certain period of time.
`table.query.request.parameters_bytes` `RATE`, bytes	The parameter size to the queries received by the database in a certain period of time.
`table.query.response.bytes` `RATE`, bytes	The size of responses sent by the database in a certain period of time.
`table.query.compilation.latency_milliseconds` `HIST_RATE`, pieces	Histogram counter. The intervals are set in milliseconds. Shows the number of successfully executed compilation queries whose duration falls within a certain time interval.
`table.query.compilation.active_count` `IGAUGE`, pieces	The number of active compilations at a given time.
`table.query.compilation.count` `RATE`, pieces	The number of compilations that completed successfully in a certain time period.
`table.query.compilation.errors` `RATE`, pieces	The number of compilations that failed in a certain period of time.
`table.query.compilation.cache_hits` `RATE`, pieces	The number of queries in a certain period of time, which didn't require any compilation, because there was an existing plan in the cache of prepared queries.
`table.query.compilation.cache_misses` `RATE`, pieces	The number of queries in a certain period of time that required query compilation.
`table.query.execution.latency_milliseconds` `HIST_RATE`, pieces	Histogram counter. The intervals are set in milliseconds. Shows the number of queries whose execution time falls within a certain interval.

Table partition metrics

Metric name Type, units of measurement	Description Labels
`table.datashard.row_count` `GAUGE`, pieces	The number of rows in DB tables.
`table.datashard.size_bytes` `GAUGE`, bytes	The size of data in DB tables.
`table.datashard.used_core_percents` `HIST_GAUGE`, %	Histogram counter. The intervals are set as a percentage. Shows the number of table partitions using computing resources in the ratio that falls within a certain interval.
`table.datashard.read.rows` `RATE`, pieces	The number of rows that are read by all partitions of all DB tables in a certain period of time.
`table.datashard.read.bytes` `RATE`, bytes	The size of data that is read by all partitions of all DB tables in a certain period of time.
`table.datashard.write.rows` `RATE`, pieces	The number of rows that are written by all partitions of all DB tables in a certain period of time.
`table.datashard.write.bytes` `RATE`, bytes	The size of data that is written by all partitions of all DB tables in a certain period of time.
`table.datashard.scan.rows` `RATE`, pieces	The number of rows that are read through `StreamExecuteScanQuery` or `StreamReadTable` gRPC API calls by all partitions of all DB tables in a certain period of time.
`table.datashard.scan.bytes` `RATE`, bytes	The size of data that is read through `StreamExecuteScanQuery` or `StreamReadTable` gRPC API calls by all partitions of all DB tables in a certain period of time.
`table.datashard.bulk_upsert.rows` `RATE`, pieces	The number of rows that are added through a `BulkUpsert` gRPC API call to all partitions of all DB tables in a certain period of time.
`table.datashard.bulk_upsert.bytes` `RATE`, bytes	The size of data that is added through a `BulkUpsert` gRPC API call to all partitions of all DB tables in a certain period of time.
`table.datashard.erase.rows` `RATE`, pieces	The number of rows deleted from the database in a certain period of time.
`table.datashard.erase.bytes` `RATE`, bytes	The size of data deleted from the database in a certain period of time.
`table.datashard.cache_hit.bytes` `RATE`, bytes	The total amount of data successfully retrieved from memory (cache), indicating efficient cache utilization in serving frequently accessed data without accessing distributed storage.
`table.datashard.cache_miss.bytes` `RATE`, bytes	The total amount of data that was requested but not found in memory (cache) and was read from distributed storage, highlighting potential areas for cache optimization.

Resource usage metrics (for Dedicated mode only)

Metric name Type units of measurement	Description Labels

`resources.cpu.used_core_percents` `RATE`, %	CPU usage. If the value is `100`, one of the cores is being used for 100%. The value may be greater than `100` for multi-core configurations. Labels: - *pool*: The computing pool, possible values are `user`, `system`, `batch`, `io`, and `ic`.
`resources.cpu.limit_core_percents` `IGAUGE`, %	The percentage of CPU available to a database. For example, for a database that has three nodes with four cores in `pool=user` per node, the value of this metric will be `1200`. Labels: - *pool*: The computing pool, possible values are `user`, `system`, `batch`, `io`, and `ic`.
`resources.memory.used_bytes` `IGAUGE`, bytes	The amount of RAM used by the database nodes.
`resources.memory.limit_bytes` `IGAUGE`, bytes	RAM available to the database nodes.

Query processing metrics (for Dedicated mode only)

Metric name Type units of measurement	Description Labels
`table.query.compilation.cache_evictions` `RATE`, pieces	The number of queries evicted from the cache of prepared queries in a certain period of time.
`table.query.compilation.cache_size_bytes` `IGAUGE`, bytes	The size of the cache of prepared queries.
`table.query.compilation.cached_query_count` `IGAUGE`, pieces	The size of the cache of prepared queries.

Topic metrics

Metric name Type units of measurement	Description Labels
`topic.producers_count` `GAUGE`, pieces	The number of unique topic producers. Labels: - *topic* – the name of the topic.
`topic.storage_bytes` `GAUGE`, bytes	The size of the topic in bytes. Labels: - *topic* - the name of the topic.
`topic.read.bytes` `RATE`, bytes	The number of bytes read by the consumer from the topic. Labels: - *topic* – the name of the topic. - *consumer* – the name of the consumer.
`topic.read.messages` `RATE`, pieces	The number of messages read by the consumer from the topic. Labels: - *topic* – the name of the topic. - *consumer* – the name of the consumer.
`topic.read.lag_messages` `RATE`, pieces	The number of unread messages by the consumer in the topic. Labels: - *topic* – the name of the topic. - *consumer* – the name of the consumer.
`topic.read.lag_milliseconds` `HIST_RATE`, pieces	A histogram counter. The intervals are specified in milliseconds. It shows the number of messages where the difference between the reading time and the message creation time falls within the specified interval. Labels: - *topic* – the name of the topic. - *consumer* – the name of the consumer.

`topic.write.bytes` `RATE`, bytes	The size of the written data. Labels: - *topic* – the name of the topic.
`topic.write.uncommited_bytes` `RATE`, bytes	The size of data written as part of ongoing transactions. Labels: - *topic* — the name of the topic.
`topic.write.uncompressed_bytes` `RATE`, bytes	The size of uncompressed written data. Метки: - *topic* – the name of the topic.
`topic.write.messages` `RATE`, pieces	The number of written messages. Labels: - *topic* – the name of the topic.
`topic.write.uncommitted_messages` `RATE`, pieces	The number of messages written as part of ongoing transactions. Labels: - *topic* — the name of the topic.
`topic.write.message_size_bytes` `HIST_RATE`, pieces	A histogram counter. The intervals are specified in bytes. It shows the number of messages which size falls within the boundaries of the interval. Labels: - *topic* – the name of the topic.
`topic.write.lag_milliseconds` `HIST_RATE`, pieces	A histogram counter. The intervals are specified in milliseconds. It shows the number of messages where the difference between the write time and the message creation time falls within the specified interval. Labels: - *topic* – the name of the topic.

Grafana dashboards for YDB

This page describes Grafana dashboards for YDB. For information about how to install dashboards, see Setting up monitoring with Prometheus and Grafana.

DB status

General database dashboard.

Download the dbstatus.json file with the **DB status** dashboard.

DB overview

General database dashboard by categories:

- Health
- API
- API details
- CPU
- CPU pools
- Memory
- Storage
- DataShard
- DataShard details
- Latency

Download the dboverview.json file with the **DB overview** dashboard.

Actors

CPU utilization in an actor system.

Name	Description
CPU by execution pool (us)	CPU utilization in different execution pools across all nodes, microseconds per second (one million indicates utilization of a single core)
Actor count	Number of actors (by actor type)
CPU	CPU utilization in different execution pools (by actor type)
Events	Actor system event handling metrics

Download the actors.json file with the **Actors** dashboard.

CPU

CPU utilization in execution pools.

Name	Description
CPU by execution pool	CPU utilization in different execution pools across all nodes, microseconds per second (one million indicates utilization of a single core)
Actor count	Number of actors (by actor type)
CPU	CPU utilization in each execution pool
Events	Event handling metrics in each execution pool

Download the cpu.json file with the **CPU** dashboard.

gRPC

gRPC layer metrics.

Name	Description
Requests	Number of requests received by a database per second (by gRPC method type)

Request bytes	Size of database requests, bytes per second (by gRPC method type)
Response bytes	Size of database responses, bytes per second (by gRPC method type)
Dropped requests	Number of requests per second with processing terminated at the transport layer due to an error (by gRPC method type)
Dropped responses	Number of responses per second with sending terminated at the transport layer due to an error (by gRPC method type)
Requests in flight	Number of requests that a database is simultaneously handling (by gRPC method type)
Request bytes in flight	Size of requests that a database is simultaneously handling (by gRPC method type)

Download the grpc.json file with the **gRPC API** dashboard.

Query engine

Information about the query engine.

Name	Description
Requests	Number of incoming requests per second (by request 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 size, bytes per second (total, query result)
Sessions	Information about running sessions
Latencies	Request execution time histograms for different types of requests

Download the queryengine.json file with the **Query engine** dashboard.

TxProxy

Information about transactions from the DataShard transaction proxy layer.

Name	Description
Transactions	Datashard transaction metrics
Latencies	Execution time histograms for different stages of datashard transactions

Download the txproxy.json file with the **TxProxy** dashboard.

DataShard

DataShard tablet metrics.

Name	Description
Operations	Datashard operation statistics for different types of operations
Transactions	Information about datashard tablet transactions (by transaction type)
Latencies	Execution time histograms for different stages of custom transactions
Tablet latencies	Tablet transaction execution time histograms
Compactions	Information about LSM compaction operations performed
ReadSets	Information about ReadSets that are sent when executing a customer transaction
Other	Other metrics

Download the datashard.json file with the **DataShard** dashboard.

Database Hive

Hive metrics for the selected database.

The dashboard includes the following filters:

- database – selects the database for which metrics are displayed;
- ds – selects the Prometheus data source the dashboard will use;
- Tx type – determines the transaction type for which "`{Tx type}` average time" panel is displayed.

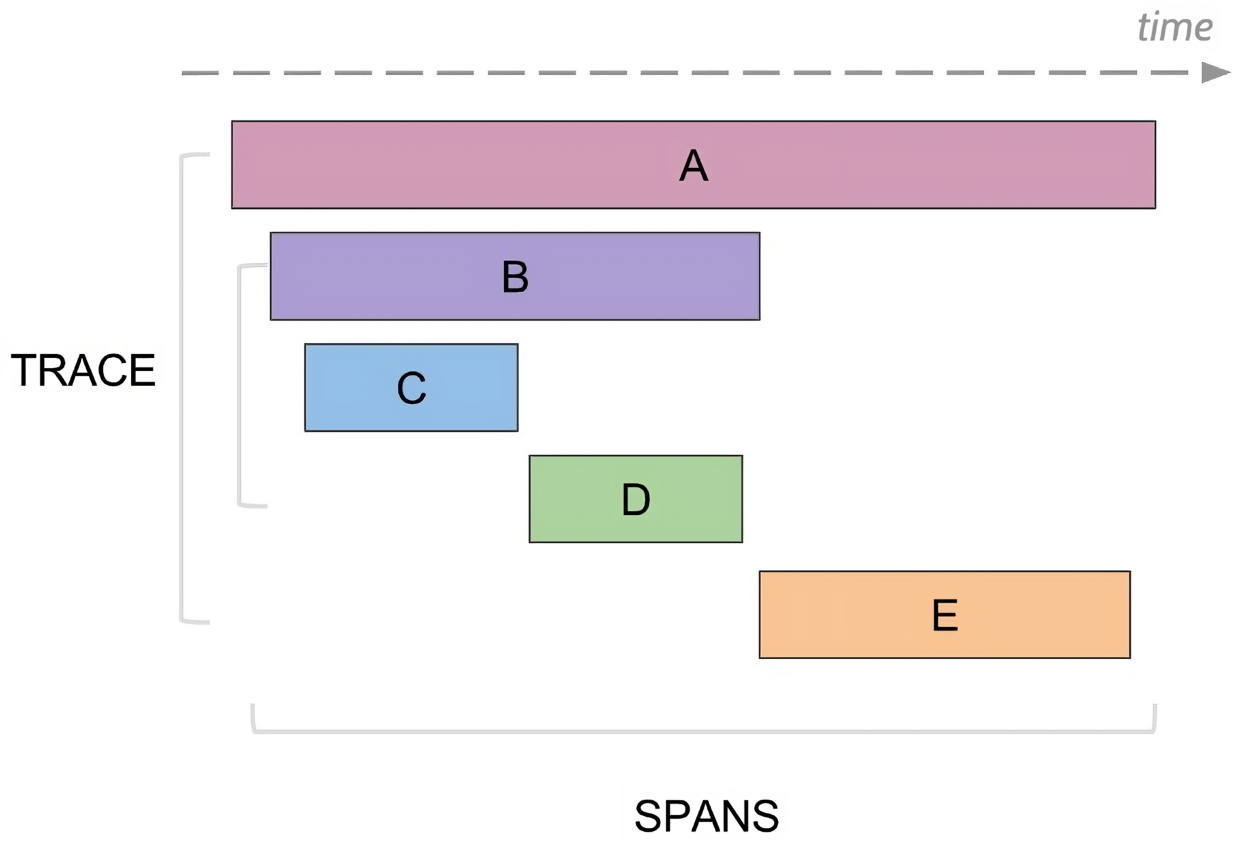
Name	Description
CPU usage by HIVE_ACTOR, HIVE_BALANCER_ACTOR	CPU time utilized by `HIVE_ACTOR` and `HIVE_BALANCER_ACTOR`, two of the most important actors of the Hive tablet.
Self-ping time	Time it takes Hive to respond to itself. High values indicate heavy load (and low responsiveness) of the Hive.
Local transaction times	CPU time utilized by various local transaction types in Hive. Shows the structure of Hive load based on different activities.
Tablet count	Total number of tablets in the database.
Event queue size	Size of the incoming event queue in Hive. Consistently high values indicate Hive cannot process events fast enough.
`{Tx type}` average time	Average execution time of a single local transaction of the type specified in the `Tx type` selector on the dashboard.
Versions	Versions of YDB running on cluster nodes.
Hive node	Node where the database Hive is running.

Download the database-hive-detailed.json file with the **Database Hive** dashboard.

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 in 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 in YDB

gRPC API

YDB supports the transmission of external trace-ids to construct a comprehensive operation trace. The transmission of trace-ids is carried out according to the trace context specification. The `traceparent` header of the gRPC request should contain a string of the form `00-0af7651916cd43dd8448eb211c80319c-b7ad6b7169203331-01`. It consists of four 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 working with the data transmitted in the header.

Only version 00 is supported, and flags are ignored. If the header does not comply with the specification, it is ignored. All traces obtained in this way have a tracing level of 13 (all components are traced at the `Detailed` level).

> **Warning**
>
> If the `external_throttling` section is present and the request flow exceeds the established 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 the transmission of trace-ids. You can find their list and usage examples on the Enabling tracing in Jaeger page.

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 Embedded 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 Embedded 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.Net.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 Embedded 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]       |
...
```

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 {#T}.

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: `$HOME/ydb/config/config.yaml`
- default configuration file for `ydbops` CLI is in same folder, `ydbops` subdirectory: `$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.

Scenarios

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'
```

ydbops reference sheet

ydbops options

DB 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.

ydbops 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.

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 run \
    -d \ # run container in background and print container ID
    --rm \ # automatically remove the container
    --name ydb-local \ # assign a name to the container
    -h localhost \ # hostname
    -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 5432:5432 \ # publish a container port to the host that provides PostgreSQL compatibility
    -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 image name and tag
```

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.

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, Embedded UI will be available at http://localhost:8765.

For more information about stopping and deleting a Docker container with YDB, see {#T}.

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
`POSTGRES_USER`	`string`	`postgres`	User name for access via compatibility with PostgreSQL
`POSTGRES_PASSWORD`	`string`		User password for access via compatibility with PostgreSQL
`YDB_GRPC_ENABLE_TLS`	`0` or `1`	`1`	
`YDB_GRPC_TLS_DATA_PATH`	`string`	`/ydb_data`	Data path with TLS certificates for the `grpcs://` connection
`MON_PORT`	`int`	`8765`	HTTP port of Embedded UI
`GRPC_PORT`	`int`	`2135`	gRPC port
`IC_PORT`	`int`	`19001`	Interconnect port
`GRPC_TLS_PORT`	`int`	`2137`	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 Queries
`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_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`	
`YDB_ENABLE_COLUMN_TABLES`	`0` or `1`	`0`	Enables column-oriented tables

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.

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

YDB SDK code recipes

This section contains code recipes in different programming languages for a variety of tasks that are common when working with the YDB SDK.

Table of contents:

- Initializing the driver
- Authentication
  - Using a token
  - Anonymous
  - Service account file
  - Metadata service
  - Using environment variables
  - Username and password based
- Balancing
  - Random choice
  - Prefer the nearest data center
  - Prefer the availability zone
- Running repeat queries
- Setting the session pool size
- Inserting data
- Bulk upsert of data
- Setting up the transaction execution mode
- Configuring time to live (TTL)
- Coordination
  - Distributed lock
  - Service discovery
  - Configuration publication
  - Leader election
- Troubleshooting
  - Enable logging
  - Enable metrics in Prometheus
  - Enable tracing in Jaeger

See also:

- YDB for Application Developers / Software Engineers
- Example applications working with YDB
- YDB SDK reference

Initialize the driver

To connect to YDB, you need to specify the required and additional parameters that define the driver's behavior (learn more in Connecting to the YDB server).

Below are examples of the code for connecting to YDB (driver creation) in different YDB SDKs.

Go (native)

```go
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()

  // ...
}
```

Go (database/sql)

Using a connector (recommended)

```go
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 importing the package of a specific driver separated by an underscore:

```go
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

```java
public void work() {
    GrpcTransport transport = GrpcTransport.forConnectionString("grpc://localhost:2136/local")
            .build());
    // Do work with the transport
    doWork(transport);
    transport.close();
}
```

JDBC Driver

```java
public void work() {
    // JDBC Driver must be in the classpath for automatic detection
    Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local");
    // Do work with the connection
    doWork(connection);
    connection.close();
}
```

C# (.NET)

```csharp
using Ydb.Sdk;

var config = new DriverConfig(
    endpoint: "grpc://localhost:2136",
    database: "/local"
);

await using var driver = await Driver.CreateInitialized(config);
```

PHP

```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('AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA') // use from
reference/ydb-sdk/auth
];

$ydb = new Ydb($config);
```

Rust

```
let client = ClientBuilder::new_from_connection_string("grpc://localhost:2136?database=local")?
      .with_credentials(AccessTokenCredentials::from("..."))
      .client()?
```

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 arising from YDB operation.
There are different response scenarios depending on the error type.
YDB
To ensure high database availability, SDKs provide built-in tools for retries,
accounting for error types and responses to them.

Below are code examples showing the YDB SDK built-in tools for retries:

Go (native)

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)
    },
    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.

Go (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

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.utf8())
    .addNullableColumn("name", PrimitiveType.utf8())
    .addNullableColumn("color", PrimitiveType.utf8())
    .addNullableColumn("price", PrimitiveType.float32())
    .setPrimaryKeys("species", "name")
    .build();

  String tablePath = database + "/" + tableName;
  retryCtx.supplyStatus(session -> session.createTable(tablePath, pets))
    .join().expect("ok");
}
```

Sample code using SessionRetryContext.supplyResult:

```
private void selectData(TableClient tableClient, String tableName) {
  SessionRetryContext retryCtx = SessionRetryContext.create(tableClient).build();
  String selectQuery
    = "DECLARE $species AS Utf8;"
    + "DECLARE $name AS Utf8;"
```

```
    + "SELECT * FROM " + tableName + " "
    + "WHERE species = $species AND name = $name;";

  Params params = Params.of(
    "$species", PrimitiveValue.utf8("cat"),
    "$name", PrimitiveValue.utf8("Tom")
  );

  DataQueryResult data = retryCtx
    .supplyResult(session -> session.executeDataQuery(selectQuery, TxControl.onlineRo(), params))
    .join().expect("ok");

  ResultSetReader rsReader = data.getResultSet(0);
  logger.info("Result of select query:");
  while (rsReader.next()) {
    logger.info("  species: {}, name: {}, color: {}, price: {}",
      rsReader.getColumn("species").getUtf8(),
      rsReader.getColumn("name").getUtf8(),
      rsReader.getColumn("color").getUtf8(),
      rsReader.getColumn("price").getFloat32()
    );
  }
}
```

Setting the session pool size

YDB creates an actor for each session. Consequently, the session pool size of a client affects resource consumption (RAM, CPU) on the server side of YDB.

For example, if 1000 clients of the same database have 1000 sessions each, then 1000000 actors are created on the server side, consuming a significant amount of memory and CPU. If you do not limit the number of sessions on the client side, this may result in a slow cluster that is at risk of failure.

By default, the YDB SDK sets a limit of 50 sessions when using native drivers. However, no session limit is enforced for third-party libraries, such as Go `database/sql`.

It's recommended to set the client session limit to the minimum required for the normal operation of the client application. Keep in mind that sessions are single-threaded on both the server and client sides. For instance, if the application needs to handle 1000 simultaneous (in-flight) requests to YDB based on its estimated load, the session limit should be set to 1000.

It is important to distinguish between estimated RPS (requests per second) and in-flight requests. RPS refers to the total number of requests completed by YDB within one second. For example, if RPS = 10000 and the average latency is 100 ms, a session limit of 1000 is sufficient. This configuration allows each session to perform an average of 10 consecutive requests within the estimated second.

Below are examples of the code for setting the session pool limit in different YDB SDKs.

Go (native)

```
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)
  ...
}
```

Go (database/sql)

The `database/sql` library has its own connection pool. Each `database/sql` connection corresponds to a specific YDB session. A `database/sql` connection pool is managed by the `sql.DB.SetMaxOpenConns` and `sql.DB.SetMaxIdleConns` functions. Learn more in the `database/sql` documentation.

Example of the code that uses the size of `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

```
this.queryClient = QueryClient.newClient(transport)
        // 10 - minimum number of active sessions to keep in the pool during the cleanup
        // 500 - maximum number of sessions in the pool
        .sessionPoolMinSize(10)
        .sessionPoolMaxSize(500)
        .build();
```

JDBC Driver

Usually working with JDBC applications use the different connections pools, such as HikariCP or C3p0. By default the YDB JDBC driver detects current count of opened connections and tunes the session pool size itself. So if the application has correct configured `HikariCP` или `C3p0`, it may not configure the session pool.

Example of HikariCP configuration in `String` application.properties:

```
  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 # maximum size of JDBC connections
```

Inserting data

Below are code examples showing the YDB SDK built-in tools for batch insert:

Go (native)

```
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)
  }
}
```

Go (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 and "+
            "Dave Krinsky. The series focuses on five young men who founded a startup company in Silicon Valley.",
          )),
          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)
  }
}
```

Bulk upsert of data

YDB supports bulk upsert of many records without atomicity guarantees. The upsert process is split into multiple independent parallel transactions, each covering a single partition. For that reason, this approach is more effective than using YQL. If successful, the `BulkUpsert` method guarantees inserting all the data transmitted by the query.

Below are code examples showing the YDB SDK built-in tools for bulk upsert:

Go (native)

```
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)
  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)
```

```
    }
}
```

Go (database/sql)

The implementation of YDB `database/sql` doesn't support bulk nontransactional upsert of data.

For bulk upsert, use transactional upsert.

Java

```
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();
    }
}
```

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*.

Serializable

Go (native)

```go
package main

import (
  "context"
  "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)
  txControl := table.TxControl(
    table.BeginTx(table.WithSerializableReadWrite()),
    table.CommitTx(),
  )
  err := driver.Table().Do(scope.Ctx, func(ctx context.Context, s table.Session) error {
    _, _, err := s.Execute(ctx, txControl, "SELECT 1", nil)
    return err
  })
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
}
```

PHP

```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 only)
    ],

    'credentials' => new AccessTokenAuthentication('AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA') // use from
reference/ydb-sdk/auth
];

$ydb = new Ydb($config);
$ydb->table()->retryTransaction(function(Session $session){
  $session->query('SELECT 1;');
})
```

Online Read-Only

Go (native)

```
package main

import (
  "context"
  "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)
  txControl := table.TxControl(
    table.BeginTx(table.WithOnlineReadOnly(table.WithInconsistentReads())),
    table.CommitTx(),
  )
  err := driver.Table().Do(scope.Ctx, func(ctx context.Context, s table.Session) error {
    _, _, err := s.Execute(ctx, txControl, "SELECT 1", nil)
    return err
  })
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
}
```

Stale Read-Only

Go (native)

```
package main

import (
  "context"
  "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)
  txControl := table.TxControl(
    table.BeginTx(table.WithStaleReadOnly()),
    table.CommitTx(),
  )
  err := driver.Table().Do(scope.Ctx, func(ctx context.Context, s table.Session) error {
    _, _, err := s.Execute(ctx, txControl, "SELECT 1", nil)
    return err
  })
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
}
```

Snapshot Read-Only

Go (native)

```go
package main

import (
  "context"
  "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)
  txControl := table.TxControl(
    table.BeginTx(table.WithSnapshotReadOnly()),
    table.CommitTx(),
  )
  err := driver.Table().Do(scope.Ctx, func(ctx context.Context, s table.Session) error {
    _, _, err := s.Execute(ctx, txControl, "SELECT 1", nil)
    return err
  })
  if err != nil {
    fmt.Printf("unexpected error: %v", err)
  }
}
```

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))
```

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))
```

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_data")),
                    TTtlTierSettings(TDuration::Hours(24), TTtlDeleteAction("/Root/s3_cold_data"))
                })
        .EndAlterTtlSettings()
);
```

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'))
)
```

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)
```

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
```

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 the code for authentication using a token in different YDB SDKs.

Go (native)

```
  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)
  ...
}
```

Go (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://localohost:2135/local?token="+os.Getenv("YDB_TOKEN"))
  if err != nil {
    panic(err)
  }
  defer db.Close()
  ...
}
```

Java

```
public void work(String accessToken) {
    AuthProvider authProvider = new TokenAuthProvider(accessToken);

    GrpcTransport transport = GrpcTransport.forConnectionString("grpcs://localohost:2135/local")
            .withAuthProvider(authProvider)
            .build());

    QueryClient queryClient = QueryClient.newClient(transport).build();

    doWork(queryClient);

    queryClient.close();
    transport.close();
}
```

JDBC

```
public void work() {
    // Connect with the specified 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);
    }

    // Connect with the token value read from the specified file path
    Properties props2 = new Properties();
    props2.setProperty("tokenFile", "~/.ydb_token");
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local", props2)) {
      doWork(connection);
    }
}
```

Node.js

```
import { Driver, TokenAuthService } from 'ydb-sdk';

export async function connect(endpoint: string, database: string, accessToken: string) {
    const authService = new TokenAuthService(accessToken);
    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

```
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)
    ...
```

Python (asyncio)

```python
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())
```

C# (.NET)

```csharp
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
<?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 the code for anonymous authentication in different YDB SDKs.

Go (native)

By default, anonymous authentication is used.
You can explicitly enable anonymous authentication as follows:

```
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,
    os.Getenv("YDB_CONNECTION_STRING"),
    ydb.WithAnonymousCredentials(),
  )
  if err != nil {
    panic(err)
  }
  defer db.Close(ctx)
  ...
}
```

Go (database/sql)

By default, anonymous authentication is used.
You can explicitly enable anonymous authentication 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

```
public void work(String connectionString) {
    AuthProvider authProvider = NopAuthProvider.INSTANCE;

    GrpcTransport transport = GrpcTransport.forConnectionString(connectionString)
            .withAuthProvider(authProvider)
            .build());
```

```
    QueryClient queryClient = QueryClient.newClient(transport).build();

    doWork(queryClient);

    queryClient.close();
    transport.close();
}
```

JDBC

```
public void work() {
    // Without additional properties, the driver uses anonymous authentication
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local")) {
      doWork(connection);
    }
}
```

Node.js

```
import { Driver, AnonymousAuthService } from 'ydb-sdk';

export async function connect(endpoint: string, database: string) {
    const authService = new AnonymousAuthService();
    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

```
import os
import ydb

with ydb.Driver(
    connection_string=os.environ["YDB_CONNECTION_STRING"],
    credentials=ydb.credentials.AnonymousCredentials(),
) as driver:
    driver.wait(timeout=5)
    ...
```

Python (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())
```

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
<?php

use YdbPlatform\Ydb\Ydb;
use YdbPlatform\Ydb\Auth\Implement\AnonymousAuthentication;

$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 the code for authentication using a service account file in different YDB SDKs.

Go (native)

```
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)
  ...
}
```

Go (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

```
public void work(String connectionString, String saKeyPath) {
    AuthProvider authProvider = CloudAuthHelper.getServiceAccountFileAuthProvider(saKeyPath);
```

```
    GrpcTransport transport = GrpcTransport.forConnectionString(connectionString)
            .withAuthProvider(authProvider)
            .build());

    QueryClient queryClient = QueryClient.newClient(transport).build();

    doWork(queryClient);

    queryClient.close();
    transport.close();
}
```

JDBC

```
public void work() {
    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);
    }

    // Option saKeyFile can be added to a JDBC URL directly
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local?
saKeyFile=~/keys/sa_key.json")) {
      doWork(connection);
    }
}
```

Node.js

Loading service account data from a file:

```
import { Driver, getSACredentialsFromJson, IamAuthService } from 'ydb-sdk';

export async function connect(endpoint: string, database: string, serviceAccountFilename: string) {
    const saCredentials = getSACredentialsFromJson(serviceAccountFilename);
    const authService = new IamAuthService(saCredentials);
    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
}
```

Loading service account data from a third-party source (for example, a secret storage):

```
import { Driver, IamAuthService } from 'ydb-sdk';
import { IIamCredentials } from 'ydb-sdk/build/cjs/src/credentials';

export async function connect(endpoint: string, database: string) {
    const saCredentials: IIamCredentials = {
        serviceAccountId: 'serviceAccountId',
        accessKeyId: 'accessKeyId',
        privateKey: Buffer.from('-----BEGIN PRIVATE KEY-----\nyJ1yFwJq...'),
        iamEndpoint: 'iam.api.cloud.yandex.net:443',
    };
    const authService = new IamAuthService(saCredentials);
    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

```
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)
    ...
```

Python (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())
```

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
<?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 the code for authentication using environment variables in different YDB SDKs.

Go (native)

```
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)
  ...
}
```

Go (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

```
public void work(String connectionString) {
    AuthProvider authProvider = CloudAuthHelper.getMetadataAuthProvider();

    GrpcTransport transport = GrpcTransport.forConnectionString(connectionString)
            .withAuthProvider(authProvider)
            .build());
```

```
    QueryClient queryClient = QueryClient.newClient(transport).build();

    doWork(queryClient);

    queryClient.close();
    transport.close();
}
```

JDBC

```
public void work() {
    Properties props = new Properties();
    props.setProperty("useMetadata", "true");
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local", props)) {
      doWork(connection);
    }

    // Option useMetadata can be added to a JDBC URL directly
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local?
useMetadata=true")) {
      doWork(connection);
    }
}
```

Node.js

```
import { Driver, MetadataAuthService } from 'ydb-sdk';

export async function connect(endpoint: string, database: string) {
    const authService = new MetadataAuthService();
    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

```
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)
    ...
```

Python (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())
```

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

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 file in Yandex Cloud.
- `YDB_ANONYMOUS_CREDENTIALS="1"`: Use anonymous authentication. Relevant for testing against a Docker container with YDB.
- `YDB_METADATA_CREDENTIALS="1"`: Use the metadata service inside Yandex Cloud (a Yandex function or a VM).
- `YDB_ACCESS_TOKEN_CREDENTIALS=<access_token>`: Use token-based authentication.

Below are examples of the code for authentication using environment variables in different YDB SDKs.

Go (native)

```
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)
  ...
}
```

Go (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

```
public void work(String connectionString) {
    AuthProvider authProvider = new EnvironAuthProvider();

    GrpcTransport transport = GrpcTransport.forConnectionString(connectionString)
            .withAuthProvider(authProvider)
            .build());

    QueryClient queryClient = QueryClient.newClient(transport).build();

    doWork(queryClient);

    queryClient.close();
    transport.close();
}
```

Node.js

```
  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

```
  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)
      ...
```

Python (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())
```

PHP

```
<?php

use YdbPlatform\Ydb\Ydb;
use YdbPlatform\Ydb\Auth\EnvironCredentials;

$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 the code for authentication based on a username and token 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)

You can pass the username and password in the connection string. For example:

```
"grpcs://login:password@localohost:2135/local"
```

You can also explicitly pass them using the `ydb.WithStaticCredentials` parameter:

```
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)
  ...
}
```

Go (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 explicitly pass the username and password at driver initialization via a connector using the `ydb.WithStaticCredentials` parameter:

```go
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

```java
public void work(String connectionString, String username, String password) {
    AuthProvider authProvider = new StaticCredentials(username, password);

    GrpcTransport transport = GrpcTransport.forConnectionString(connectionString)
            .withAuthProvider(authProvider)
            .build());

    QueryClient queryClient = QueryClient.newClient(transport).build();

    doWork(queryClient);

    queryClient.close();
    transport.close();
}
```

JDBC

```java
public void work(String username, String password) {
    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 via the special method of DriverManager
    try (Connection connection = DriverManager.getConnection("jdbc:ydb:grpc://localhost:2136/local", username,
password)) {
      doWork(connection);
    }
}
```

Node.js

```javascript
import { Driver, StaticCredentialsAuthService } from 'ydb-sdk';

export async function connect(endpoint: string, database: string, user: string, password: string) {
    const authService = new StaticCredentialsAuthService(user, password, endpoint, {
        tokenExpirationTimeout: 20000,
    })
    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

```
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)
    ...
```

Python (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())
```

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;
```

```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 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

Random choice

The YDB SDK uses the `random_choice` algorithm by default.

Below are examples of the code for forced setting of the "random choice" balancing algorithm in different YDB SDKs.

Go (native)

```
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)
  ...
}
```

Go (database/sql)

Client load balancing in the YDB `database/sql` driver is performed only when establishing a new connection (in `database/sql` terms), which is a 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.

Example of the code for setting the "random choice" balancing algorithm:

```
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()
```

```
  ...
}
```

Prefer the nearest data center

Below are examples of the code for setting the "prefer the nearest data center" balancing algorithm option in different YDB SDKs.

Go (native)

```
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)
  ...
}
```

Go (database/sql)

Client load balancing in the YDB `database/sql` driver is performed only when establishing a new connection (in `database/sql` terms), which is a 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.

Example of the code for setting the "prefer the nearest data center" balancing algorithm option:

```
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()
```

```
  ...
}
```

Prefer the specific availability zone

Below are examples of the code for setting the "prefer the availability zone" balancing algorithm option in different YDB SDKs.

Go (native)

```
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)
  ...
}
```

Go (database/sql)

Client load balancing in the YDB `database/sql` driver is performed only when establishing a new connection (in `database/sql` terms), which is a 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.

Example of the code for setting the "prefer the availability zone":

```
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()
  ...
}
```

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
}
```

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.

Troubleshooting

When troubleshooting issues with YDB, diagnostics tools such as logging, metrics, OpenTracing/Jaeger tracing are helpful. We strongly recommend that you enable them in advance before any problems occur. This will help see changes in the overall picture before, during, and after an issue when troubleshooting it. This greatly speeds up our investigation into incidents and lets us provide assistance much faster.

This section contains code recipes for enabling diagnostics tools in different YDB SDKs.

Table of contents:

- Enable logging
- Enable metrics in Prometheus
- Enable tracing in Jaeger

Enabling logging

Below are examples of code that enables logging in different YDB SDKs.

Go (native)

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.

Go (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

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 `com.yandex.ydb.core.grpc` logger provides information about the internal implementation of the gRPC protocol
- The `debug` level logs all operations run over gRPC, so we recommend using it only for debugging purposes.
- The `info` level is recommended by default
- On the `debug` level, the `com.yandex.ydb.table.impl` logger enables you to track the internal status of the YDB driver, including session pool health.
- On the `debug` level, the `com.yandex.ydb.table.SessionRetryContext` logger will inform you of the number of retries, results of executed queries, execution time of specific retries, and the total operation execution time
- On the `debug` level, the `com.yandex.ydb.table.Session` logger gives you details about the query text, response status, and execution time of specific operations within the session

Enabling and configuration the Java SDK loggers depends on the `slf4j-api` implementation used.
Here's an example of a `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="com.yandex.ydb.core.grpc" level="info" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
    <Logger name="com.yandex.ydb.table.impl" level="info" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
    <Logger name="com.yandex.ydb.table.SessionRetryContext" level="debug" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
    <Logger name="com.yandex.ydb.table.Session" level="debug" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>

    <Root level="debug" >
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

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:

```
$config = [
  'logger' => new \YdbPlatform\Ydb\Logger\SimpleStdLogger(\YdbPlatform\Ydb\Logger\SimpleStdLogger::INFO)
]
$ydb = new \YdbPlatform\Ydb\Ydb($config);
```

Enabling metrics in Prometheus

Below are examples of the code for enabling metrics in Prometheus in different YDB SDKs.

Go (native)

```
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)
    ...
}
```

Go (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.OpnDB(connector)
    defer db.Close()
    ...
}
```

Enabling tracing in Jaeger

Below are examples of the code enabling Jaeger tracing in different YDB SDKs.

Go (native)

```
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)
    ...
}
```

Go (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.OpnDB(connector)
    defer db.Close()
    ...
}
```

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 several standard benchmarks:

Benchmark	Reference
TPC-H	tpch
TPC-DS	tpcds
ClickBench	clickbench

They all function similarly. For a detailed description of each, refer to the relevant reference via the links 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 clickbench --path path/in/database ...
ydb workload tpch --path path/in/database ...
ydb workload tpcds --path path/in/database ...
```

Load testing can be divided into 3 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 clickbench --path clickbench/hits init --store=column
ydb workload tpch --path tpch/s1 init --store=column
ydb workload tpcds --path tpcds/s1 init --store=column
```

At this stage, 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 description of the commands for each benchmark:

- clickbench init
- tpch init
- tpcds init

Data filling

Filling with data is performed using the `import` command. This command is specific to each benchmark, and its behavior depends on the subcommands. However, there are also parameters common to all benchmarks.

For a detailed description, see the relevant reference sections:

- clickbench import
- tpch import
- tpcds import

Examples:

```
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
```

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 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
```

The command allows you to select queries for execution, generate various types of reports, collect execution statistics, and more.

For a detailed description, see the relevant reference sections:

- clickbench run
- tpch run
- tpcds 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 clickbench --path clickbench/hits clean
ydb workload tpch --path tpch/s1 clean
ydb workload tpcds --path tpcds/s1 clean
```

For a detailed description, see the corresponding sections:

- clickbench clean
- tpch clean
- tpcds 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 mytable
```

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 OBJECT access_key (TYPE SECRET) WITH (value="...");
CREATE OBJECT secret_key (TYPE SECRET) 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_NAME="access_key",
    AWS_SECRET_ACCESS_KEY_SECRET_NAME="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("PT0S") ON expire_at
);
```

Disabling TTL

```
ALTER TABLE `mytable` RESET (TTL);
```
```

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

  Diagnistic steps for most performance issues involve analyzing Grafana dashboards that use YDB metrics collected by Prometheus For information on installing Grafana and Prometheus, see Setting up monitoring for a YDB cluster.
- YDB logs
- Tracing
- YDB CLI
- Embedded UI
- Query plans
- 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

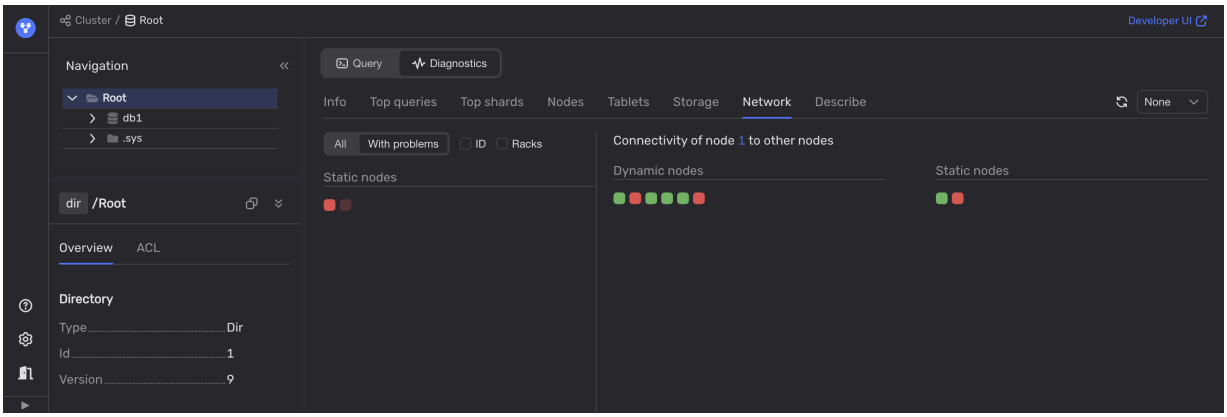
Network issues

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 Embedded UI:

1. Open the Embedded 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.

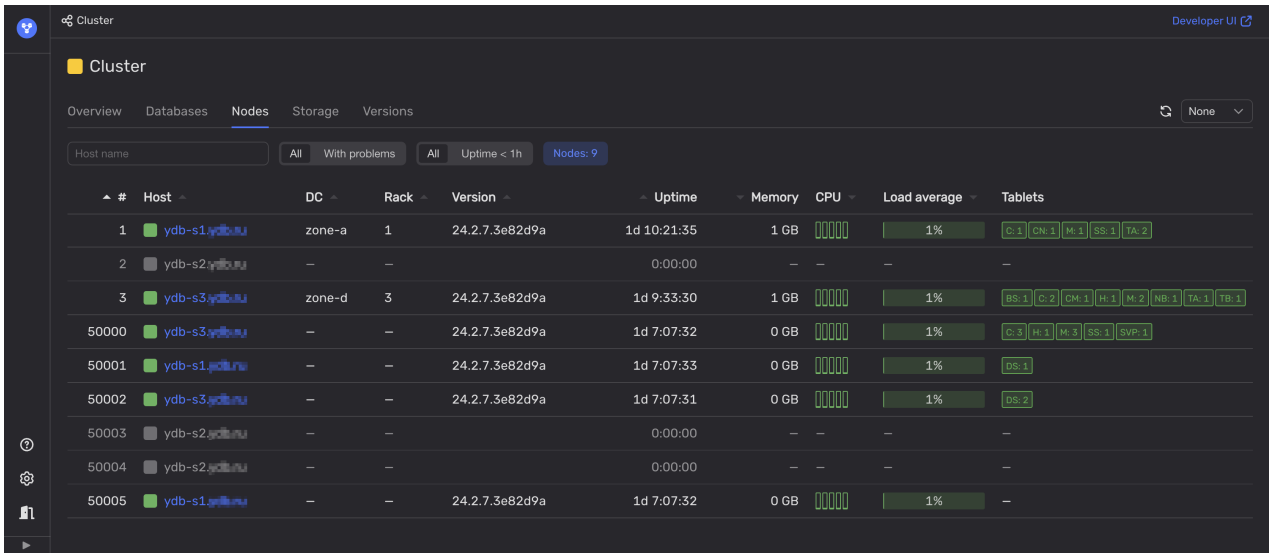
Data center outages

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 Embedded 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 Embedded 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.

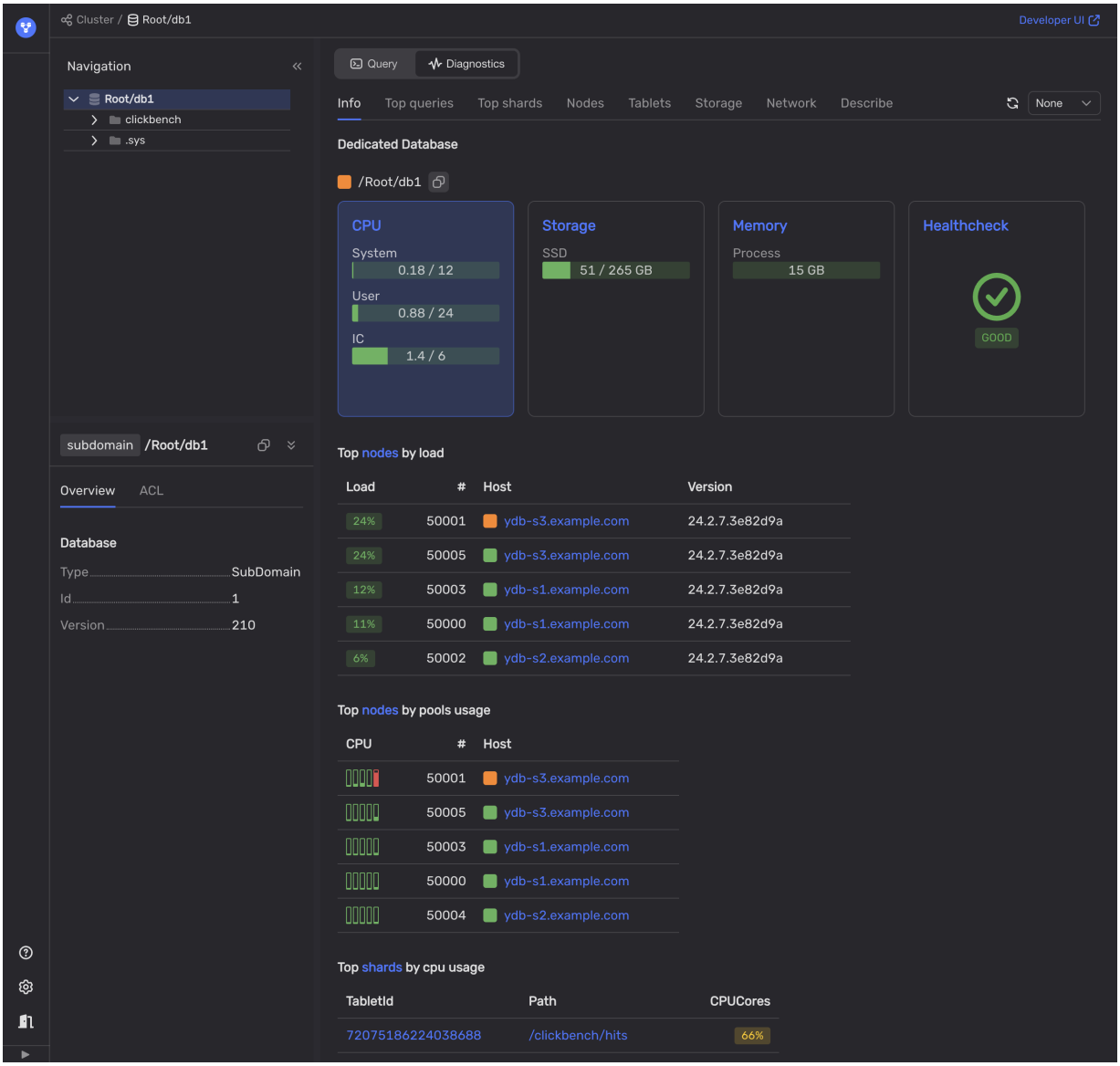
CPU bottleneck

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.

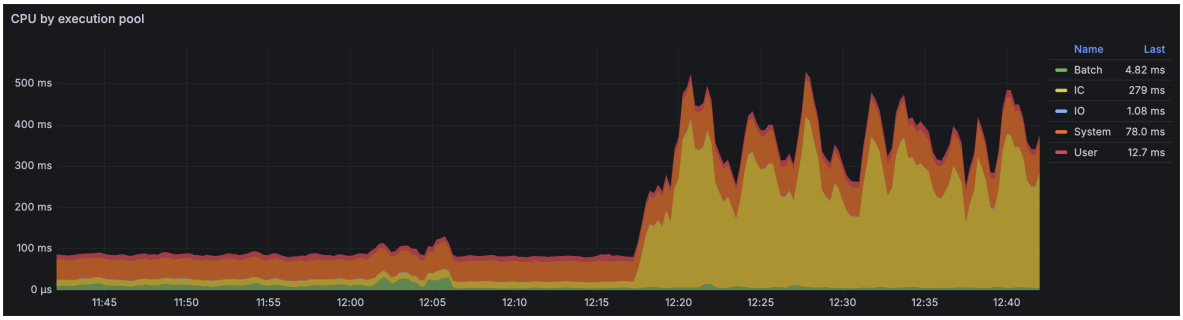
YDB nodes primarily consume CPU resources for running actors. On each node, actors are executed using multiple actor system pools. The resource consumption of each pool is measured separately which allows to identify what kind of activity changed its behavior.

Diagnostics

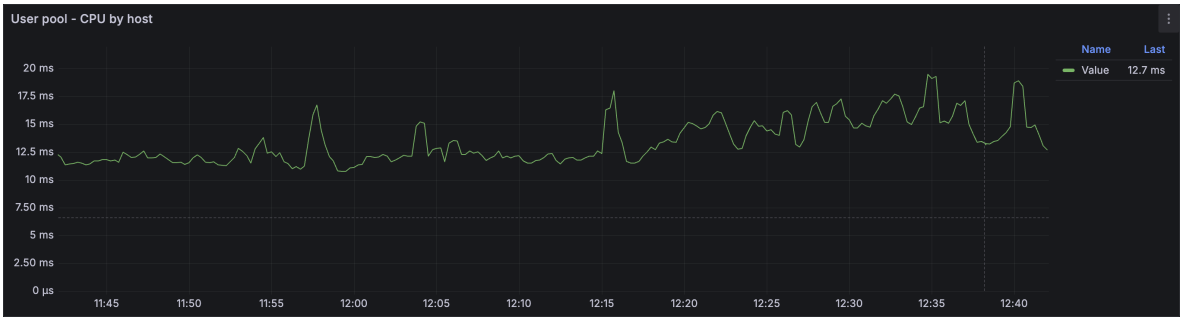
1. Use **Diagnostics** in the Embedded UI to analyze CPU utilization in all pools:
   1.1. In the Embedded UI, go to the **Databases** tab and click on the database.
   1.2. On the **Navigation** tab, ensure the required database is selected.
   1.3. Open the **Diagnostics** tab.
   1.4. On the **Info** 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



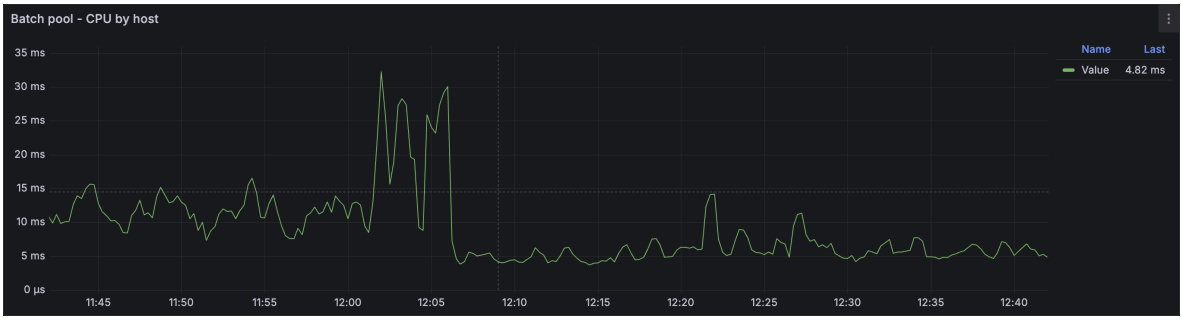
- **User pool - CPU by host** chart



- **System pool - CPU by host** chart



- **Batch pool - CPU by host** chart



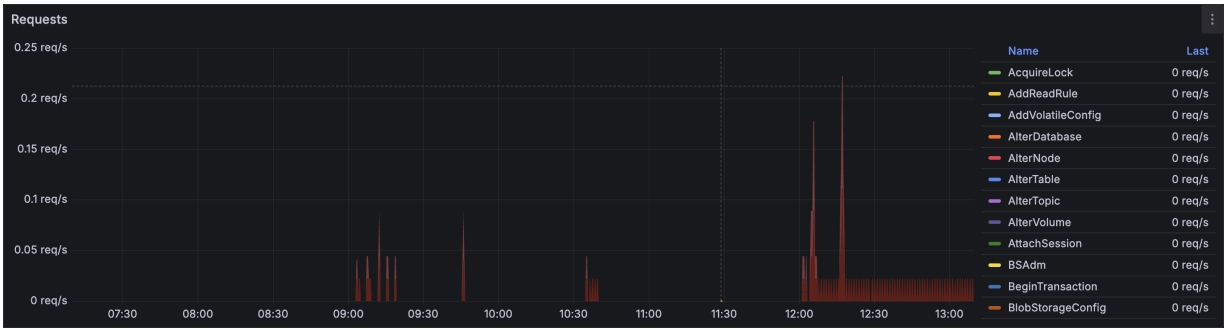
- **IC pool - CPU by host** chart



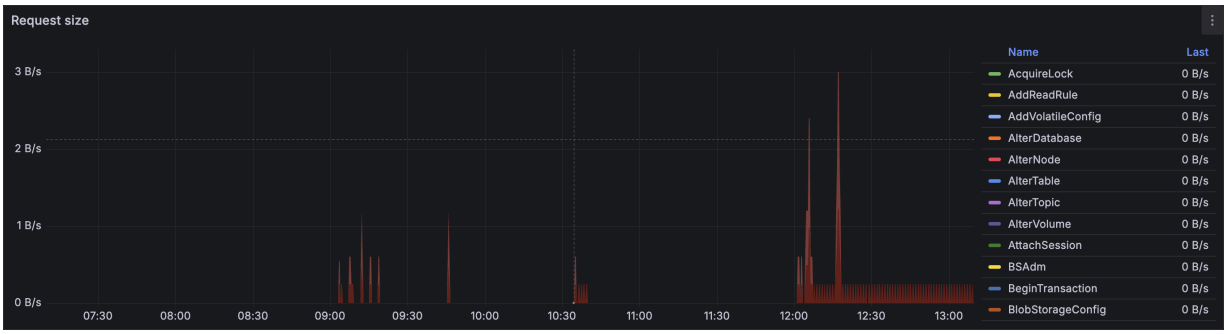
- **IO pool - CPU by host** chart



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 Embedded 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, file-
   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:

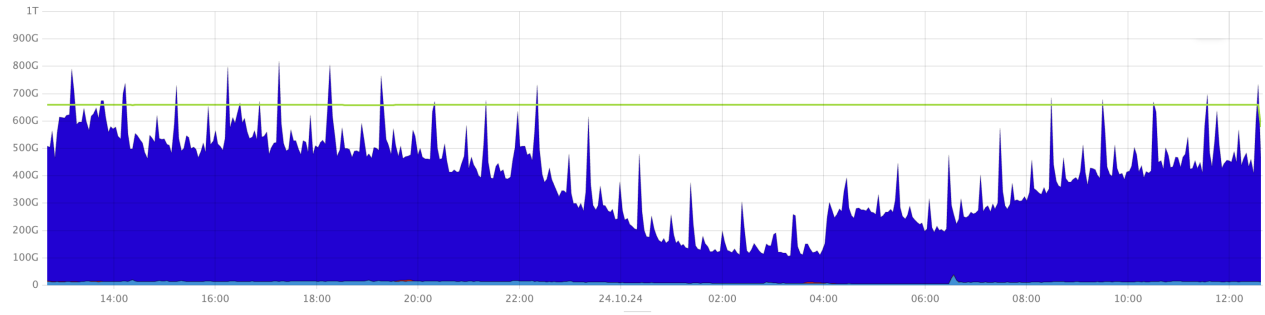
- If the load on YDB has increased due to new usage patterns or increased query rate, try optimizing the application to reduce the load on YDB or add more YDB nodes.
- If the load on YDB has not changed but nodes are still restarting, consider adding more YDB nodes or raising the hard memory limit for the nodes. For more information about memory management in YDB, see Memory controller.

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.

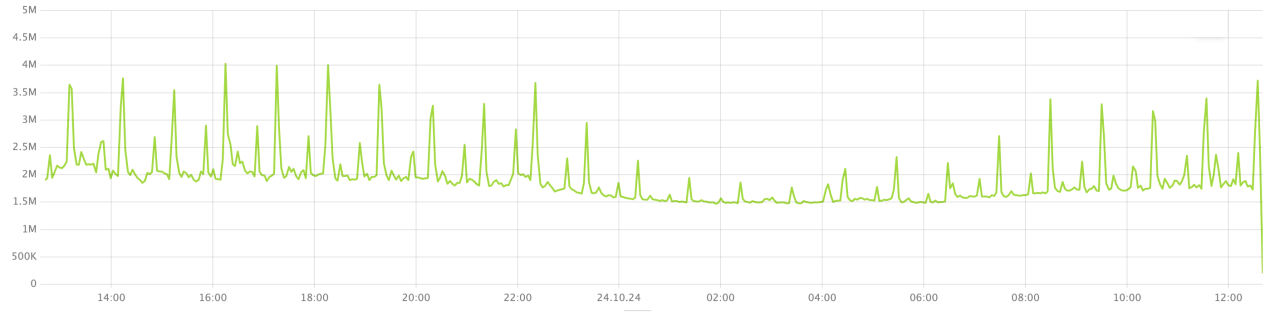
Diagnostics

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.

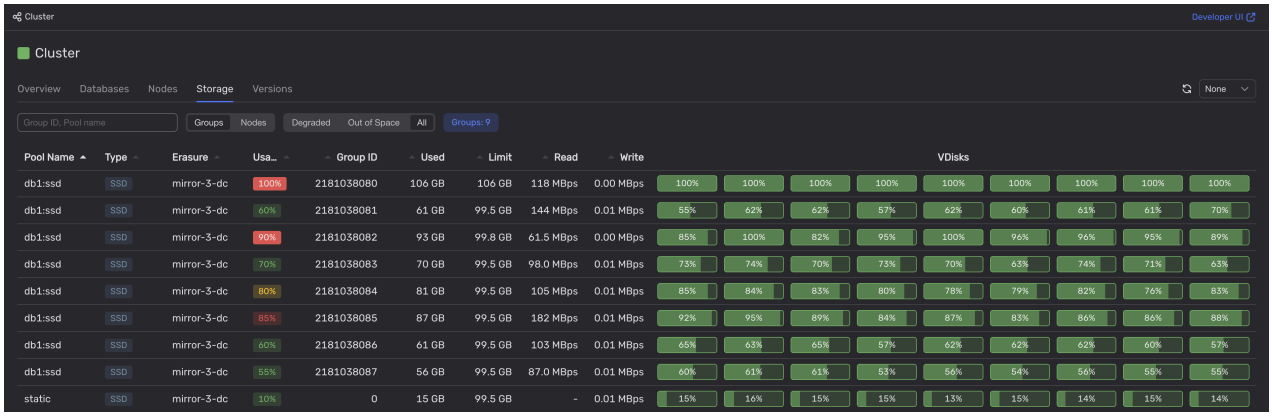
Disk space

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 Embedded 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.

If the system clocks of the nodes running the coordinator tablets differ, 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 coordinator with the slowest clock reaches the same time.

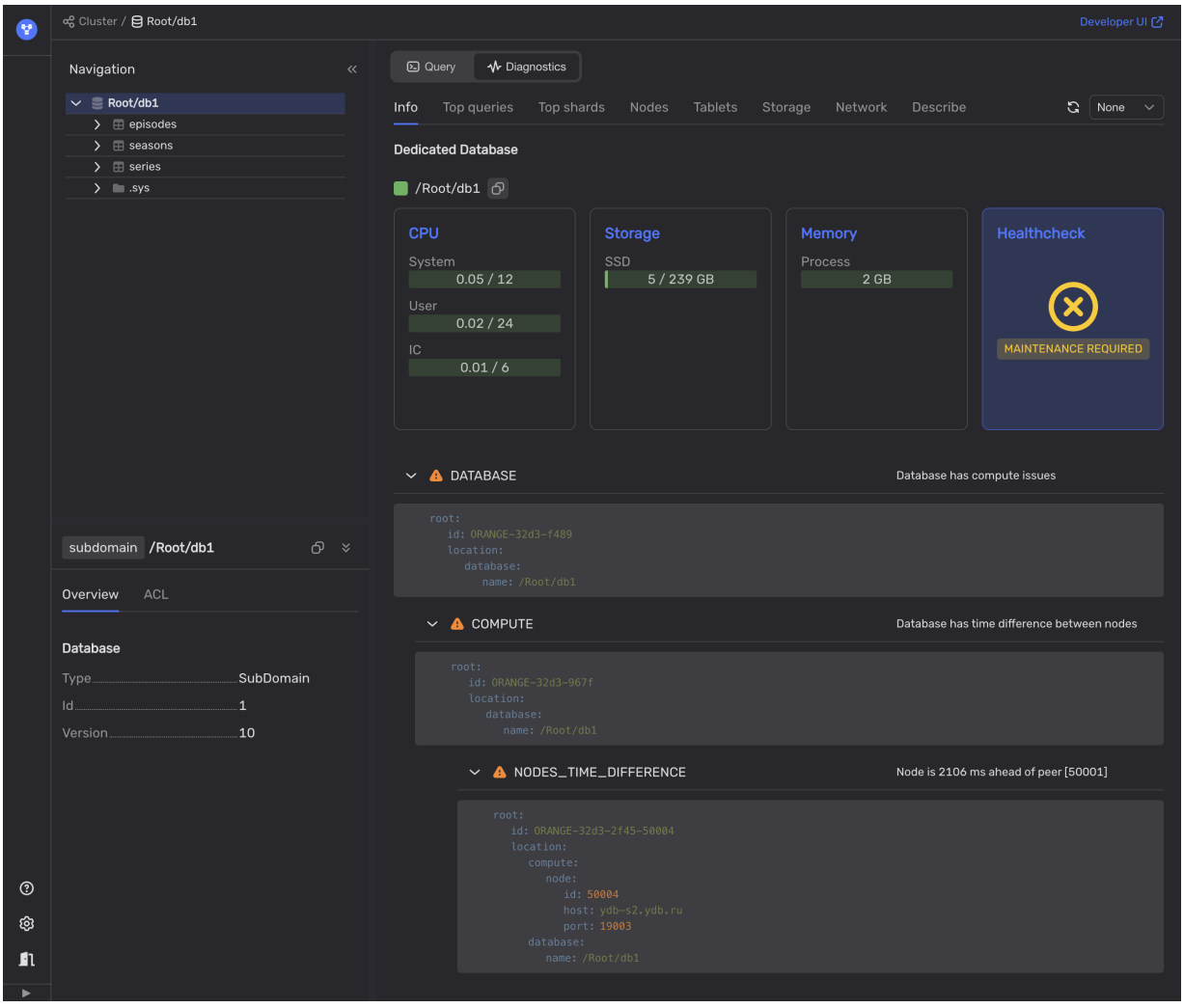
Furthermore, if the system clock drift exceeds 30 seconds, YDB will refuse to process distributed transactions. Before coordinators start planning a transaction, affected Data shards determine an acceptable range of timestamps for the transaction. The start of this range is the current time of the mediator tablet's clock, while the 30-second planning timeout determines the end. If the coordinator's system clock exceeds this time range, it cannot plan a distributed transaction, resulting in errors for such queries.

Diagnostics

To diagnose the system clock drift, use the following methods:

1. Use **Healthcheck** in the Embedded UI:
   1.1. In the Embedded 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.



The system clock drift problems will be listed under `NODES_TIME_DIFFERENCE`.

> **Note**
>
> For more information, see Health Check API

2. Open the Interconnect overview page of the Embedded UI.
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, `timedatectl show-timesync --all`.

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 `timesyncd`, `ntpd`, `chrony`, or a similar tool. It's recommended to use the same time source for all servers in the YDB cluster.

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.

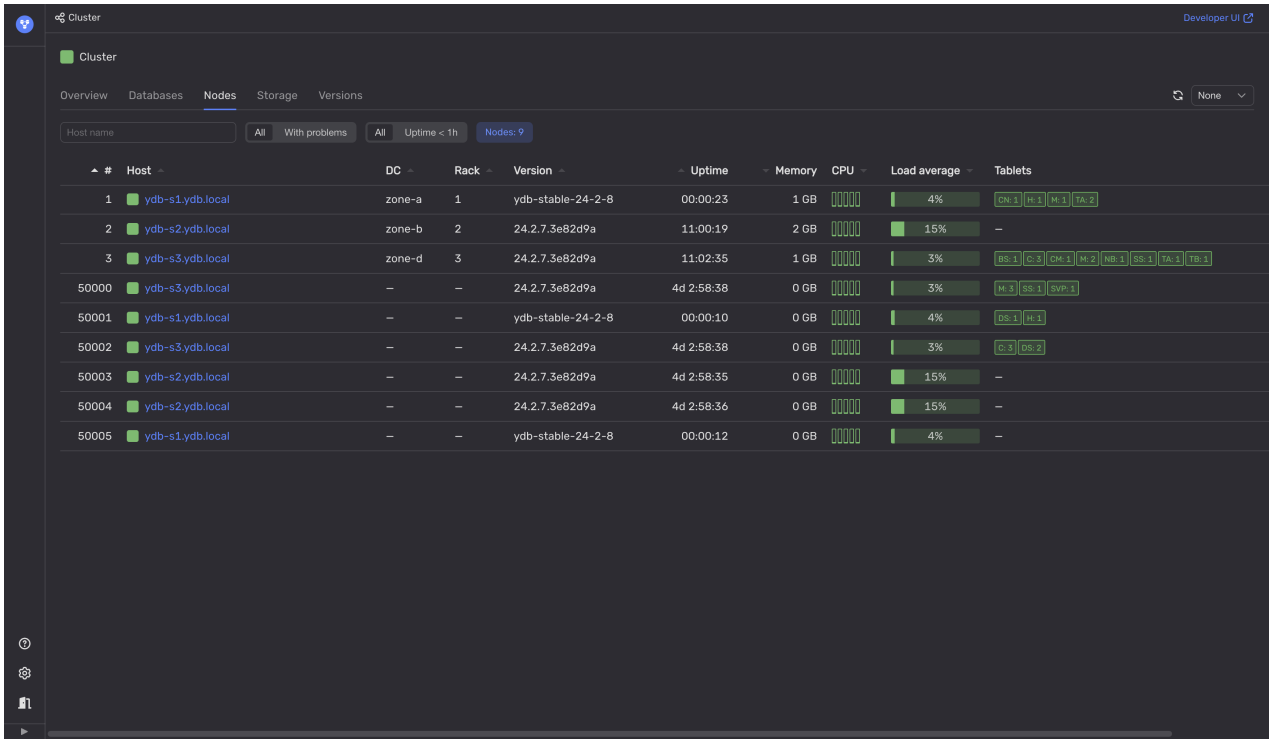
Diagnostics

> **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 Embedded 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
- Count

Autobalancing occurs in the following cases:

- **Disbalance in hardware resource usage**

  YDB uses the **scatter** metric to evaluate the balance in hardware resource usage. This metric is calculated for each resource using the following formula:

  where:

  - is the maximum hardware resource usage among all of the nodes.
  - is the minimum hardware resource usage among all of the nodes.

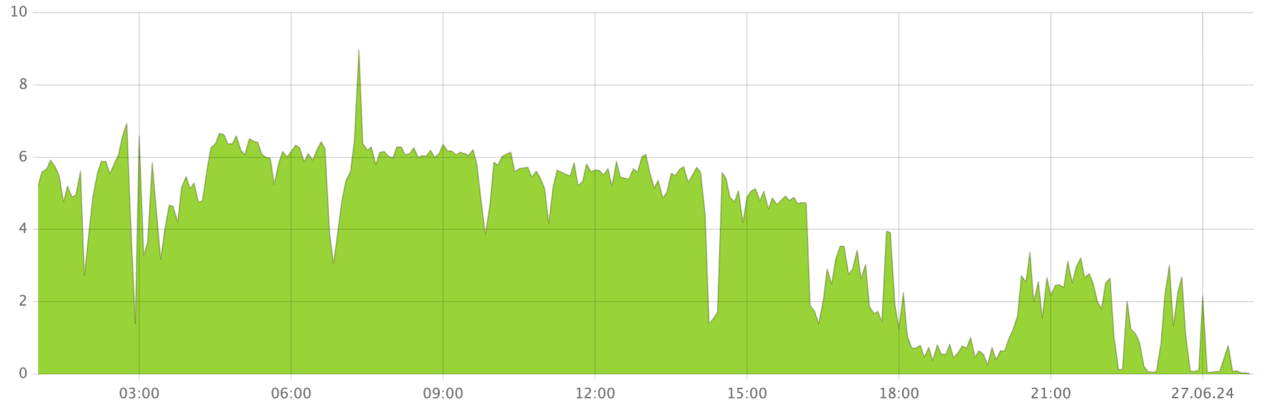
  To distribute the load, YDB considers the hardware resources available to each node. Under low loads, the scatter value may vary significantly across nodes; however, the minimum value for this formula is set to never fall below 30%.
- **Overloaded nodes (CPU and memory usage)**

  Hive starts the autobalancing procesure when the highest load on a node exceeds 90%, while the lowest load on a node is below 70%.
- **Uneven distribution of database objects**

  YDB uses the **ObjectImbalance** metric to monitor the distribution of tablets utilizing the **count* resource across YDB nodes. When YDB nodes restart, these tablets may not distribute evenly, prompting Hive to initiate the autobalancing procedure.

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 Embedded UI.
   2.2. Click **Developer UI** in the upper right corner of the Embedded 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			

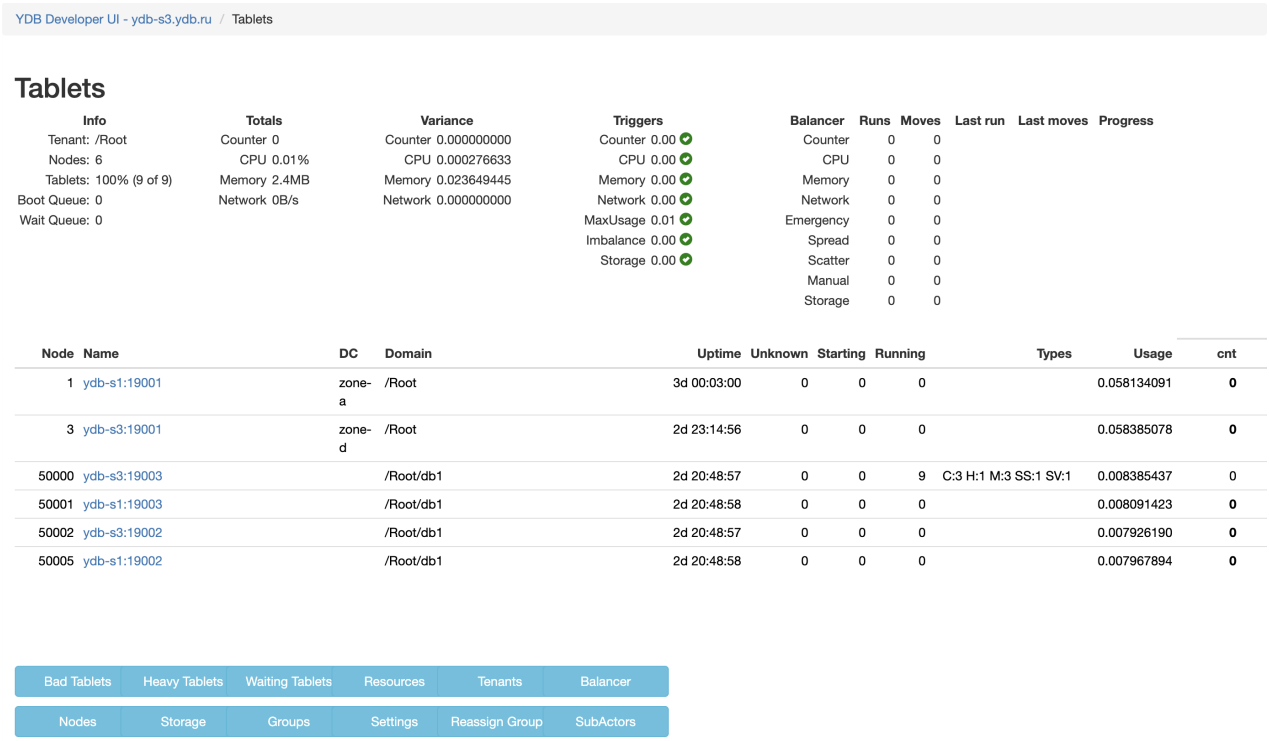
2.4. Additionally, to see the recently moved tablets, click the **Balancer** button.

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 Embedded UI.
2. Click **Developer UI** in the upper right corner of the Embedded 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.

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.

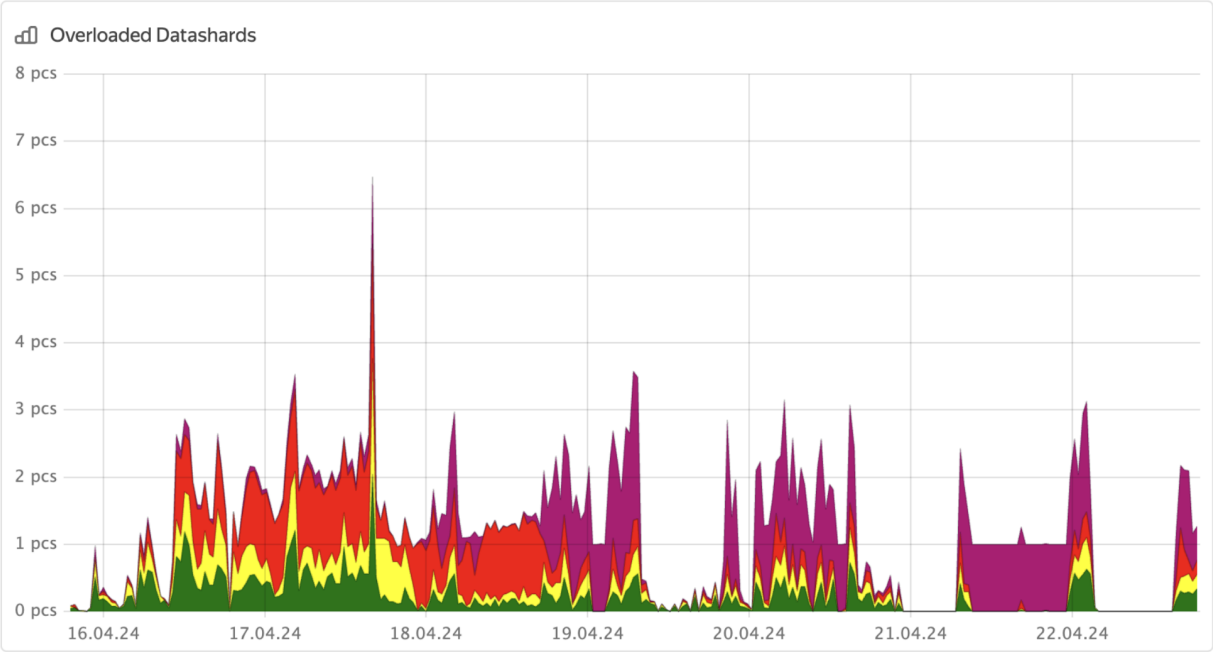
  In this case, YDB does not split overloaded shards.

  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

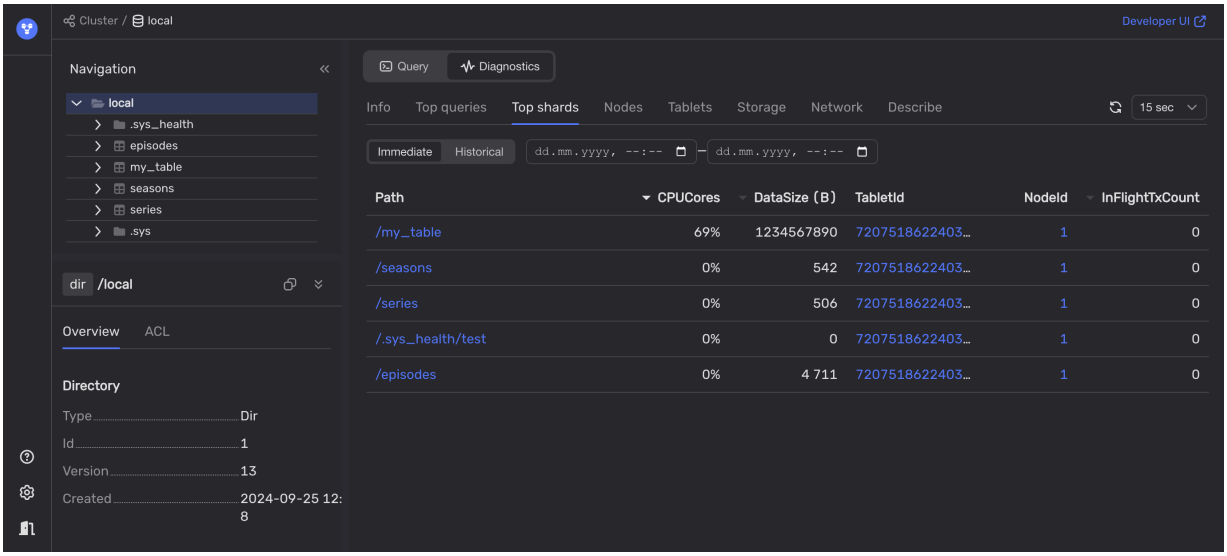
1. Use the Embedded UI or Grafana to see if the YDB nodes are overloaded:
   - In the **DB overview** Grafana dashboard, analyze the **Overloaded shard count** chart.



The chart indicates whether the YDB cluster has overloaded shards, but it does not specify which table's shards are overloaded.

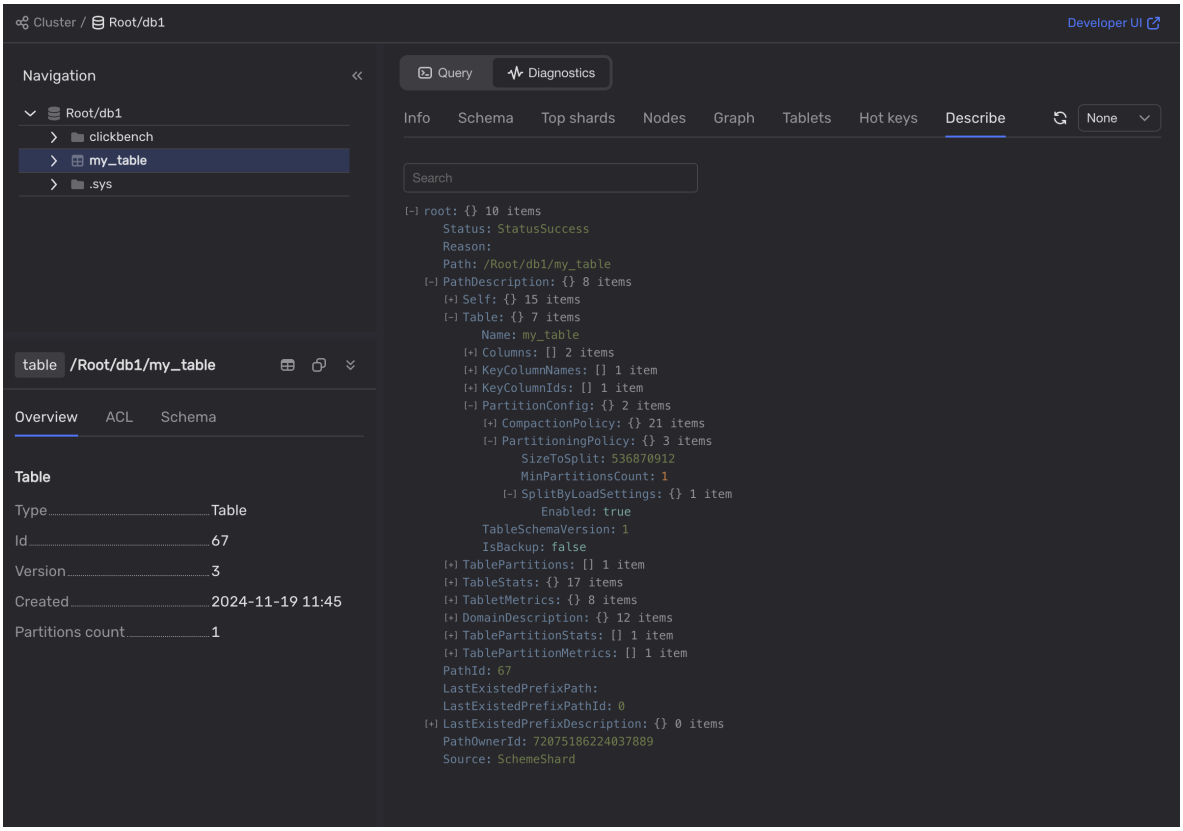
> **Tip**
>
> Use Grafana to set up alert notifications when YDB data shards get overloaded.

   - In the Embedded 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 Embedded UI or YDB CLI:
   - In the Embedded 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 **Embedded 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 **Embedded 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%.

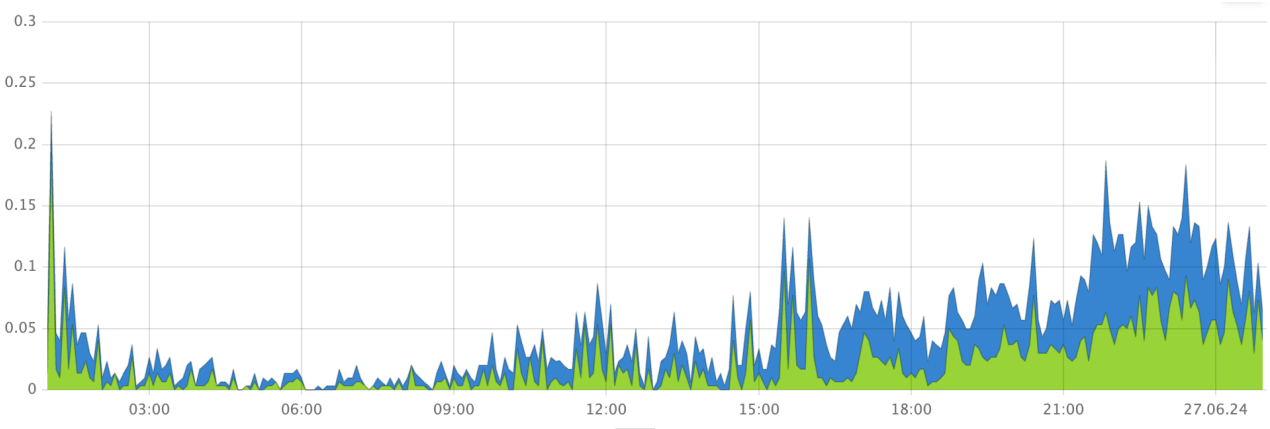
When YDB merges adjacent partitions in a row-oriented table, they are replaced with a single partition that covers their range of primary keys. TThe corresponding data shards are also consolidated into a single data shard to manage the new partition.

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

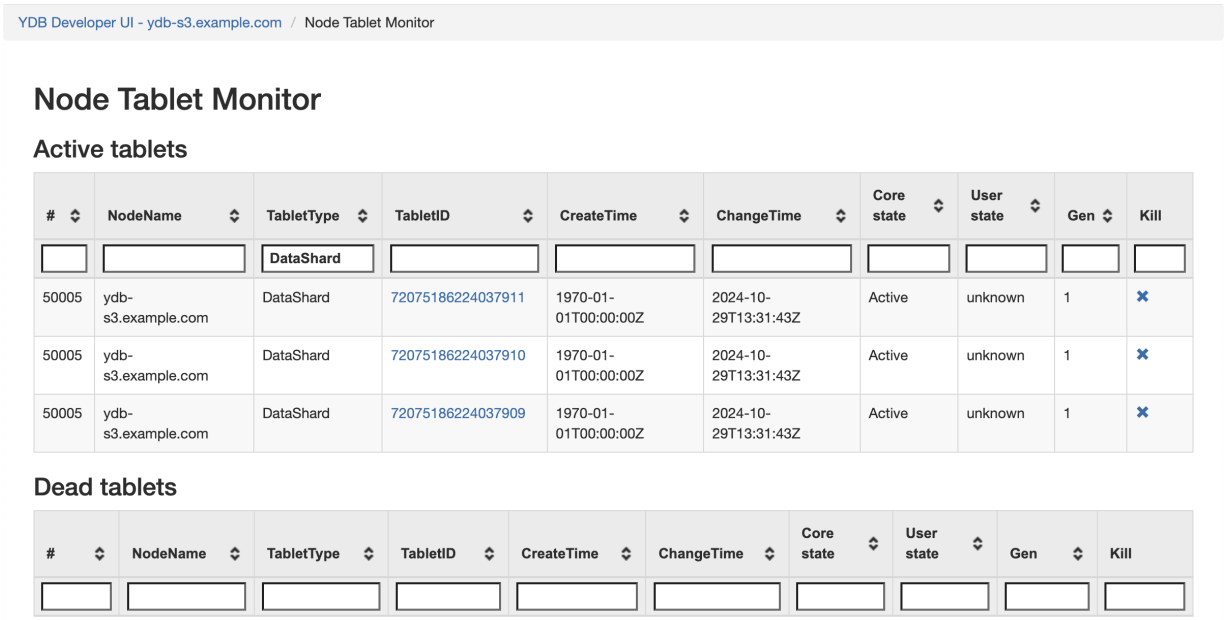
1. See if the **Split / Merge partitions** chart in the **DB status** Grafana dashboard shows any spikes.




```
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 Embedded 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.

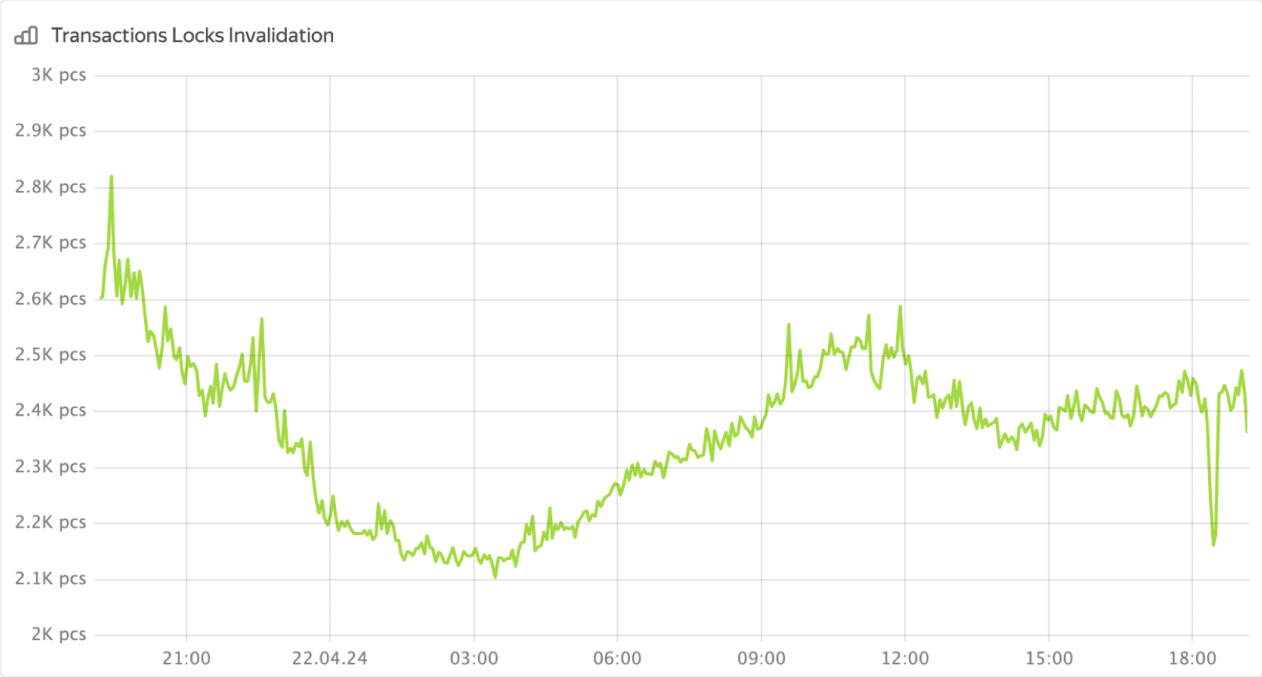
Transaction lock invalidation

YDB uses optimistic locking to find conflicts with other transactions being executed. If the locks check during the commit phase reveals conflicting modifications, the committing transaction rolls back and must be restarted. In this case, YDB returns a **transaction locks invalidated** error. Restarting a significant share of transactions can degrade your application's performance.

> **Note**
>
> The YDB SDK provides a built-in mechanism for handling temporary failures. For more information, see Handling errors.

Diagnostics

1. Open the **DB overview** Grafana dashboard.
2. See if the **Transaction Locks Invalidation** chart shows any spikes.



This chart shows the number of queries that returned the transaction locks invalidation error per second.

Recommendations

Consider the following recommendations:

- The longer a transaction lasts, the higher the likelihood of encountering a **transaction locks invalidated** error.

  If possible, avoid interactive transactions. A better approach is to use a single YQL query with `begin;` and `commit;` to select data, update data, and commit the transaction.

  If you do need interactive transactions, perform `commit` in the last query in the transaction.
- Analyze the range of primary keys where conflicting modifications occur, and try to change the application logic to reduce the number of conflicts.

  For example, if a single row with a total balance value is frequently updated, split this row into a hundred rows and calculate the total balance as a sum of these rows. This will drastically reduce the number of **transaction locks invalidated** errors.

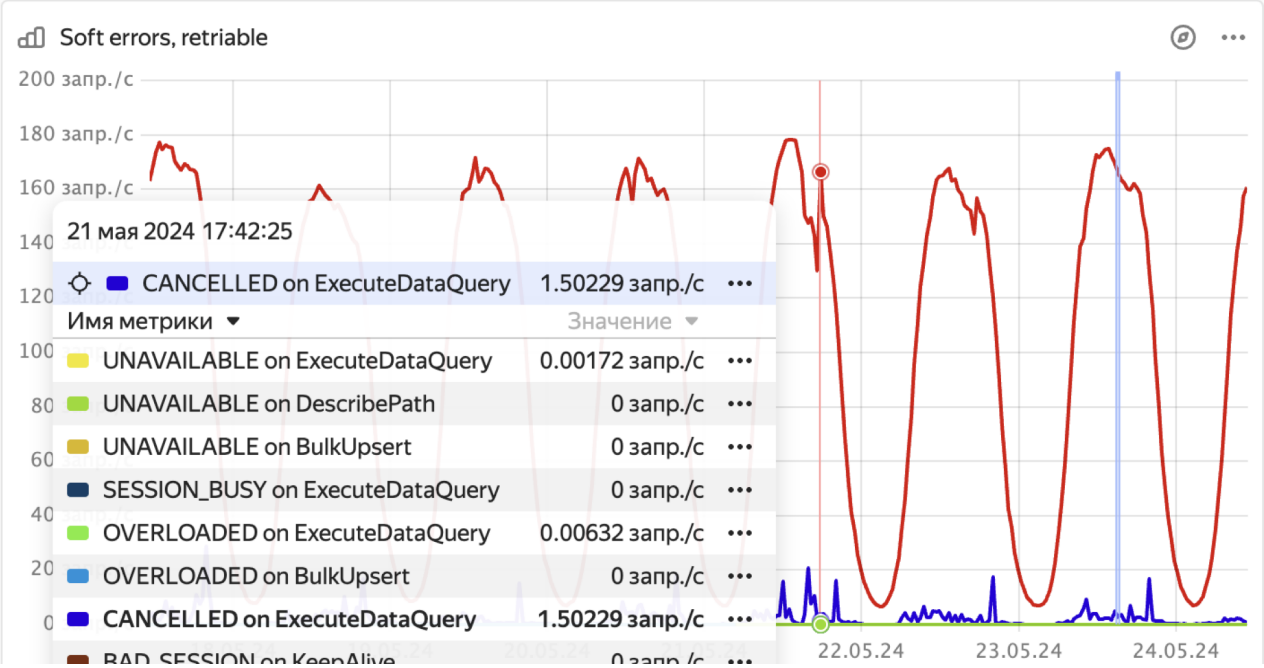
Overloaded errors

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 Embedded 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 back-off 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.

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 Embedded UI to solve the problem and observe the solution in action.

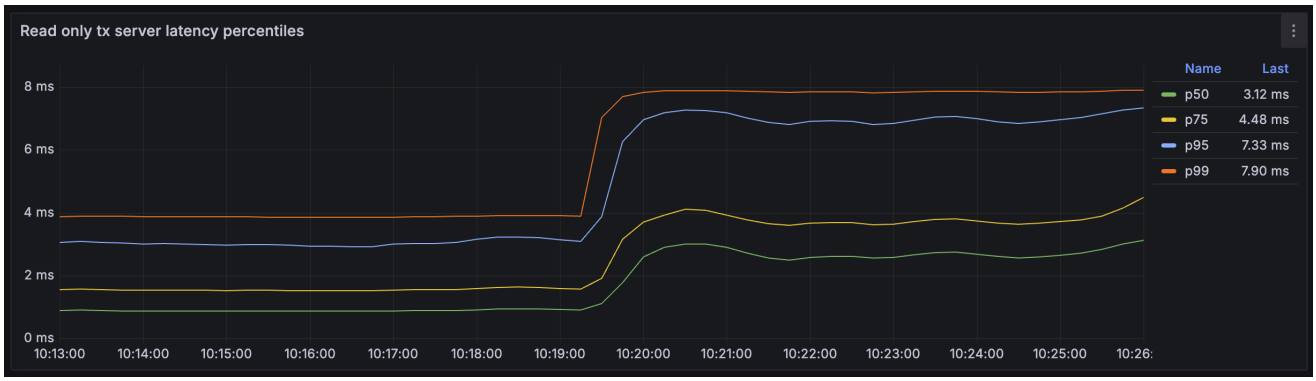
At the end of the article, you can find the steps to reproduce the situation.

Initial issue

You were notified that your system has started taking too long to process user requests.

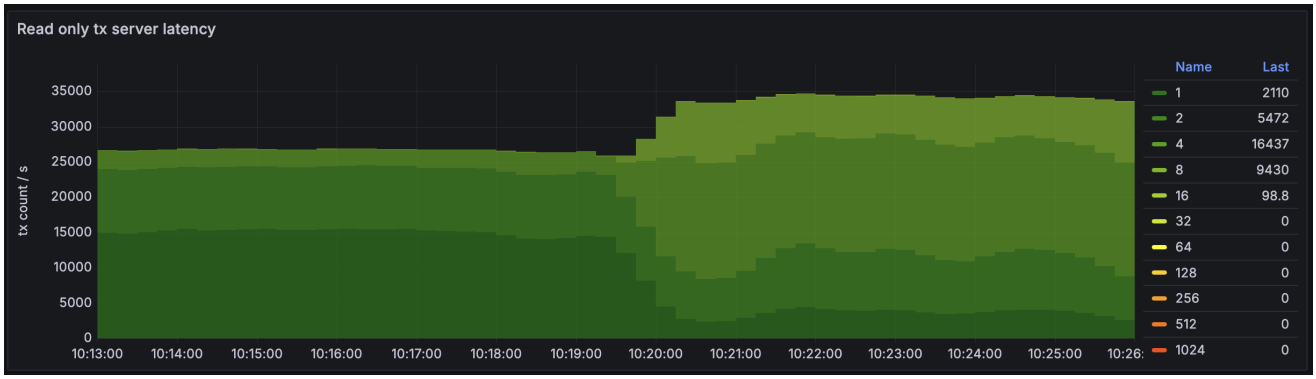
> **Note**
>
> These requests access a row-oriented table, which is managed by data shards.

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

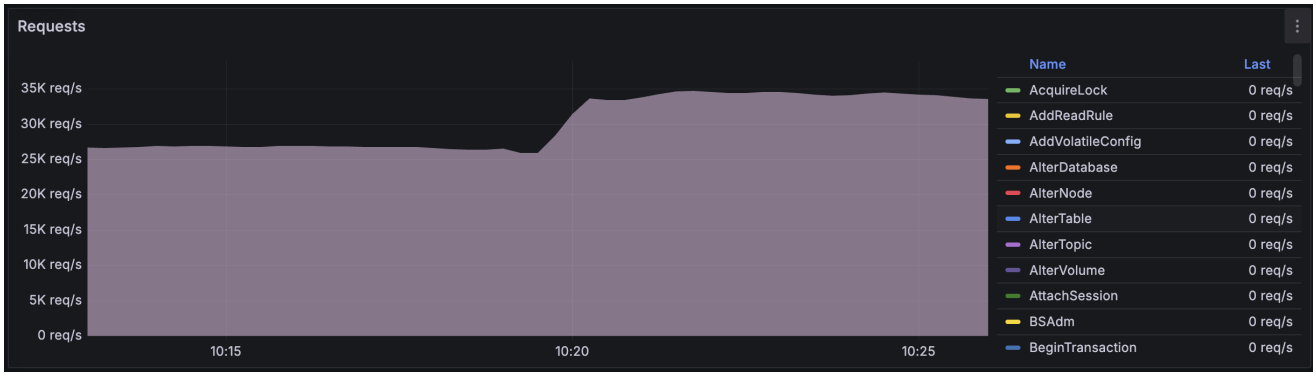
The diagram shows a heatmap of transaction latencies. Transactions are grouped into buckets based on their latency, with each bucket represented by a different color. This diagram displays both the number of transactions processed by YDB per second (on the vertical axis) and the latency distribution 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.

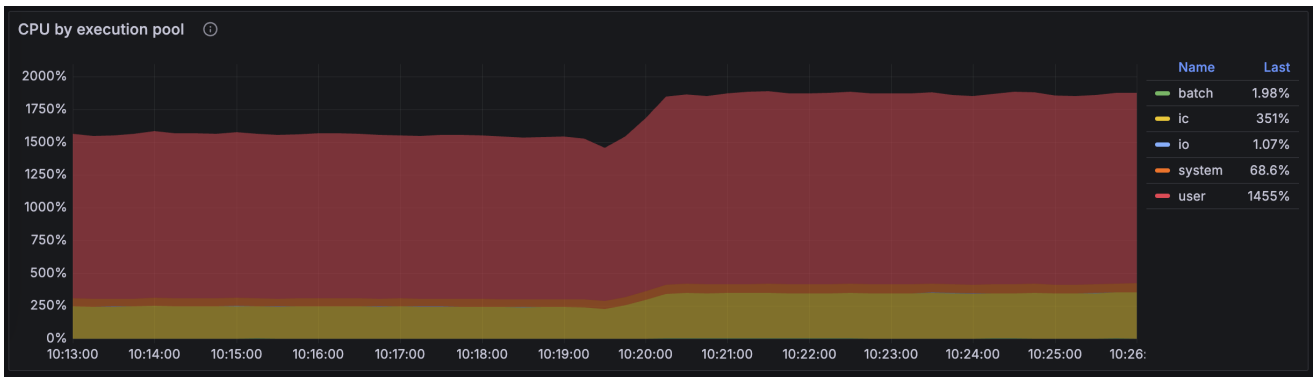
Diagnostics

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:



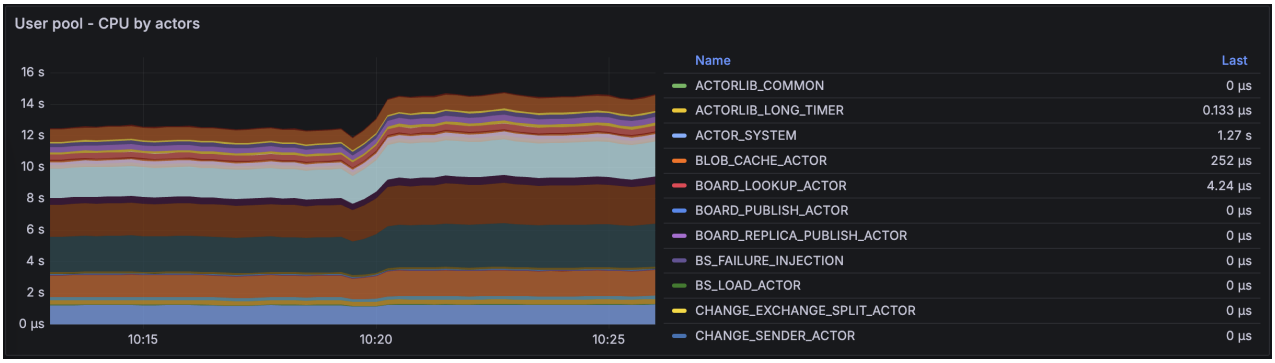
The number of user requests increased from approximately 27,000 to 35,000 at around `10:20:00`. But can YDB handle the increased load without additional hardware resources?

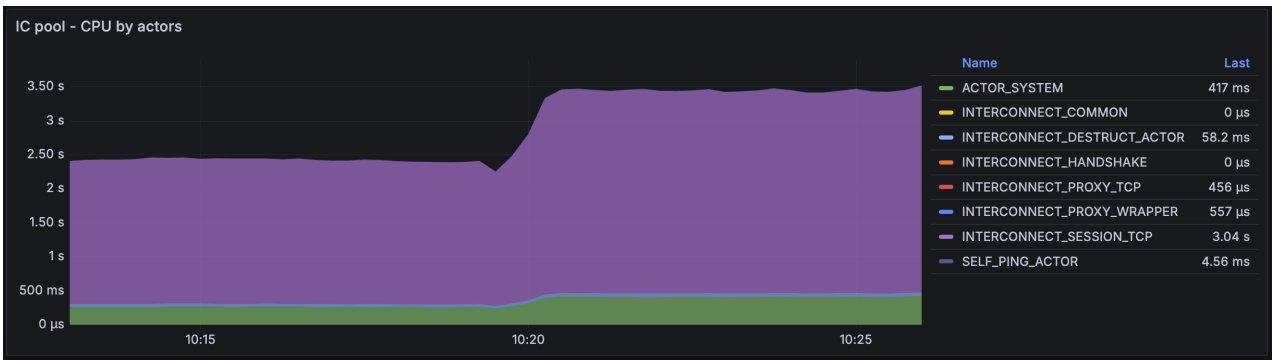
The CPU load has increased, as shown in the **CPU by execution pool** diagram.

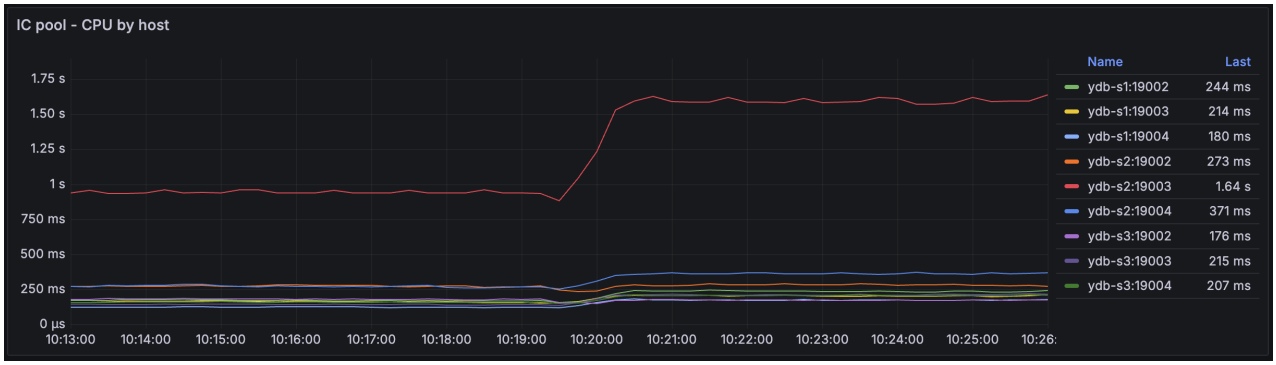


See the details on the CPU Grafana dashboard

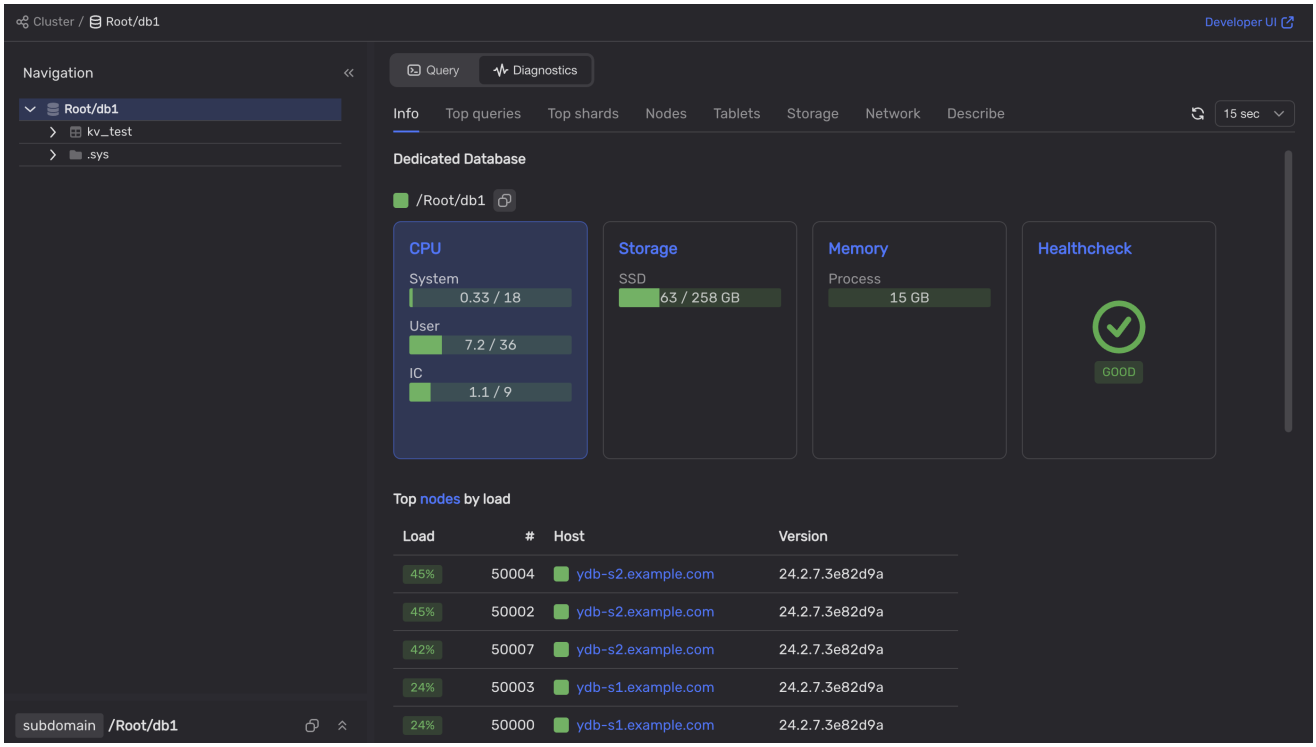
Examining the **CPU** Grafana dashboard reveals that CPU usage increased in the user pool and the interconnect pool:





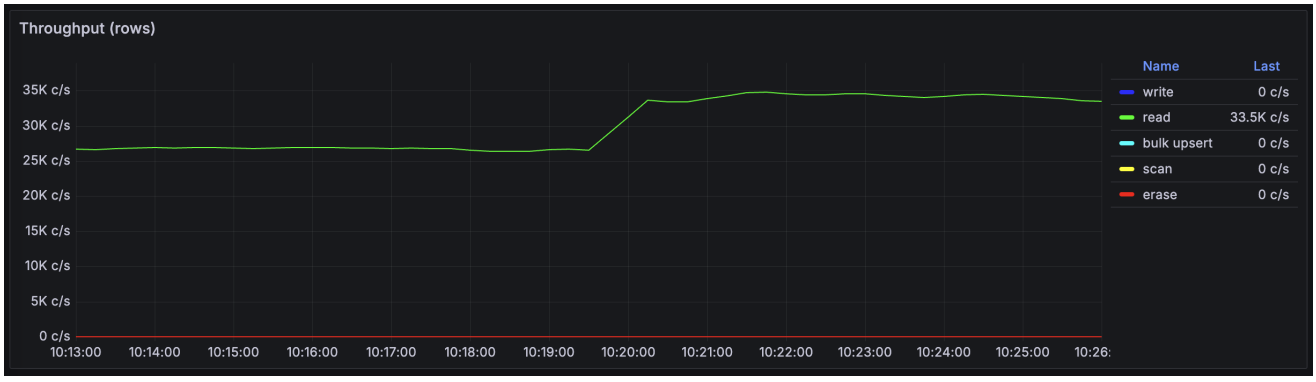


We can also observe overall CPU usage on the **Diagnostics** tab of the Embedded UI:



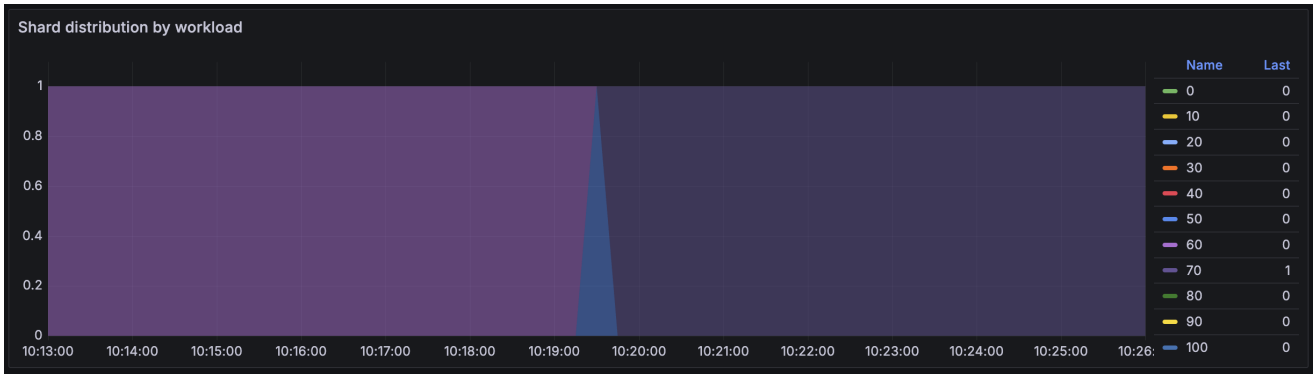
The YDB cluster appears not to utilize all of its CPU capacity.

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

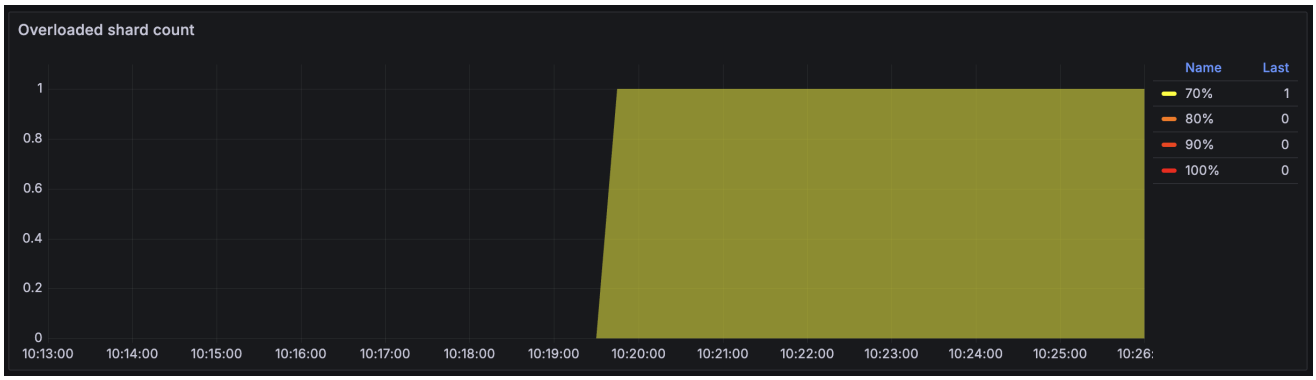
> This diagram shows that the number of rows read per second in the YDB database increased from approximately 26,000 to 33,500 rows per second.



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.

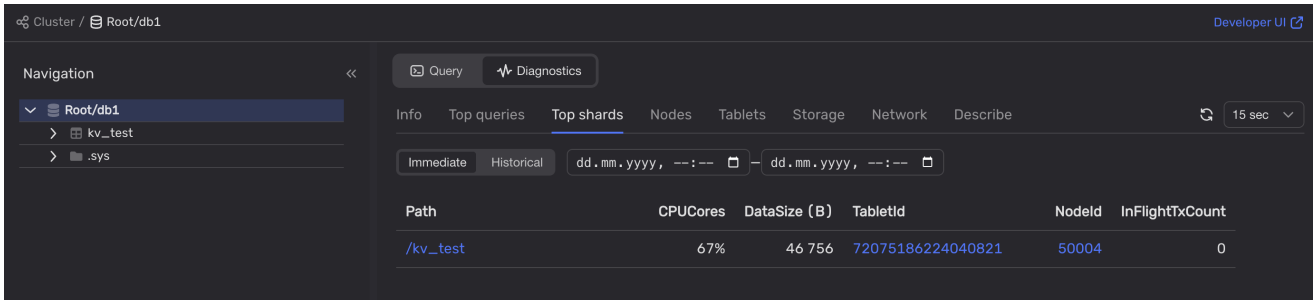


See the diagram description

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%.

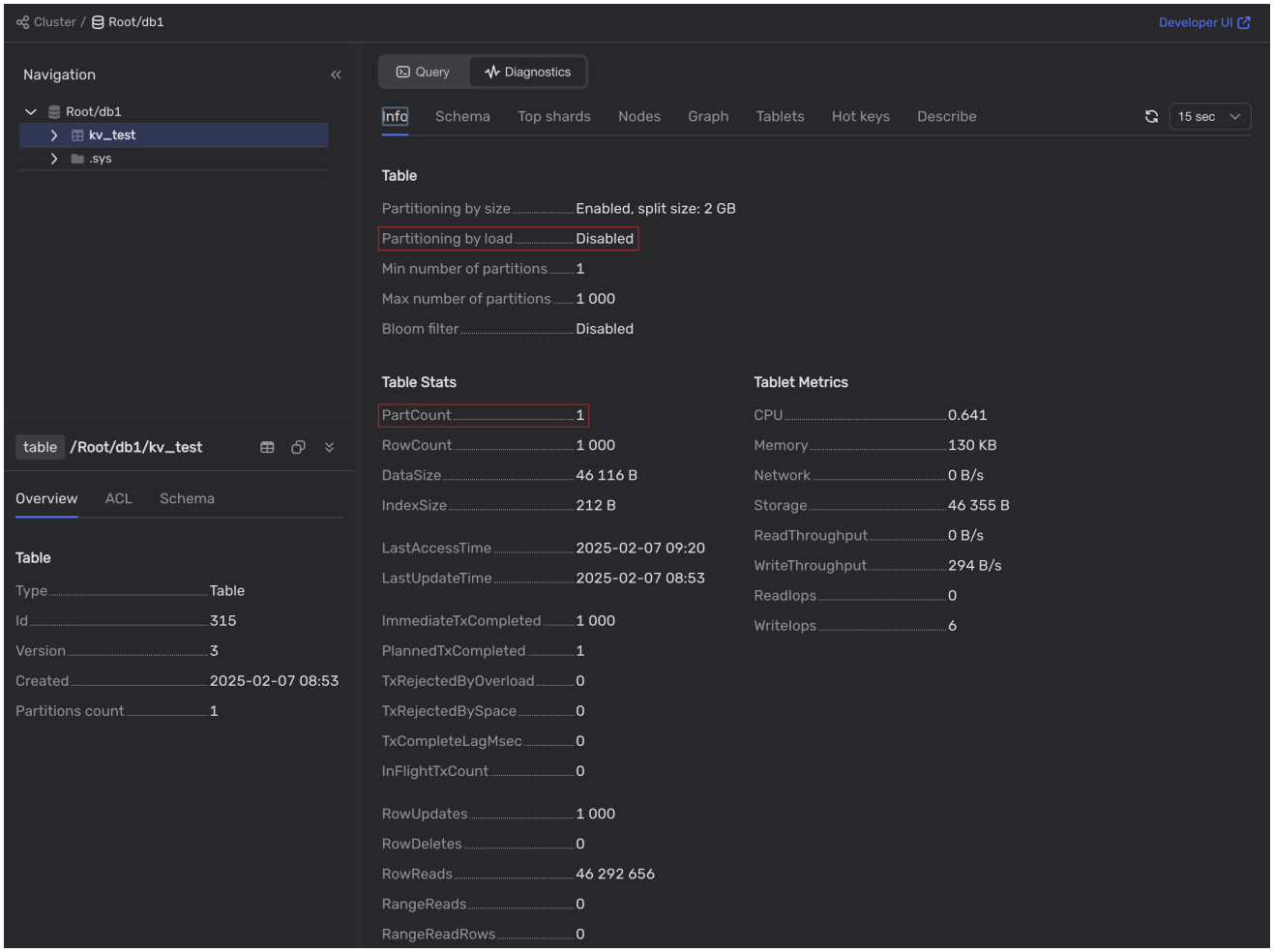
This diagram shows that the workload on one data shard increased to 70% at approximately `10:19:30`.

To determine which table the overloaded data shard is processing, let's open the **Diagnostics > Top shards** tab in the Embedded 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

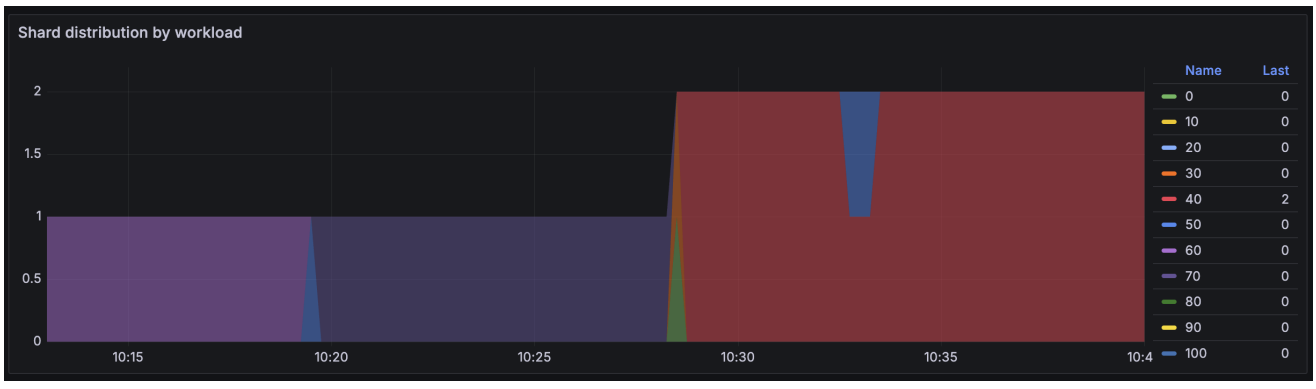
We should enable partitioning by load for the `kv_test` table:

1. In the Embedded UI, select the database.
2. Open the **Query** tab.
3. Run the following query:

```
ALTER TABLE kv_test SET (
    AUTO_PARTITIONING_BY_LOAD = ENABLED
);
```

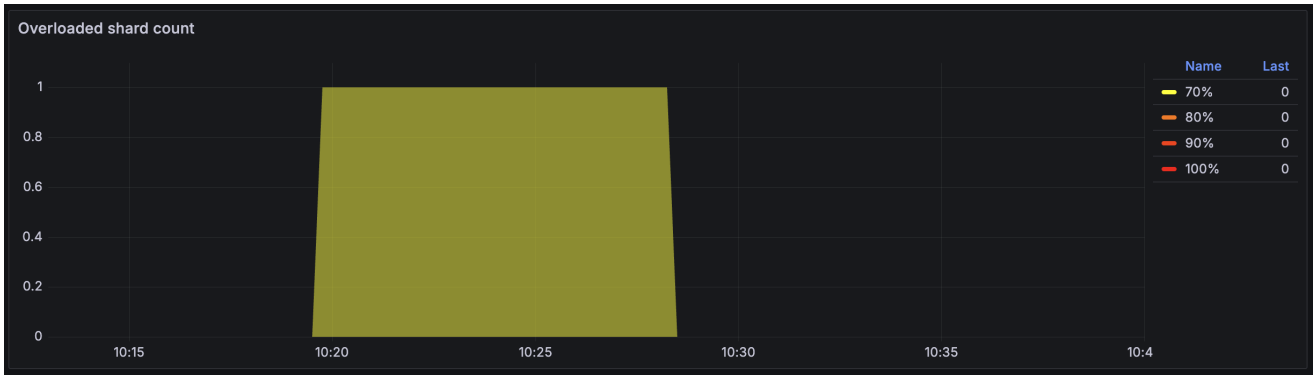
Aftermath

When we enable automatic partitioning for the `kv_test` table, the overloaded data shard splits into two.



See the diagram description

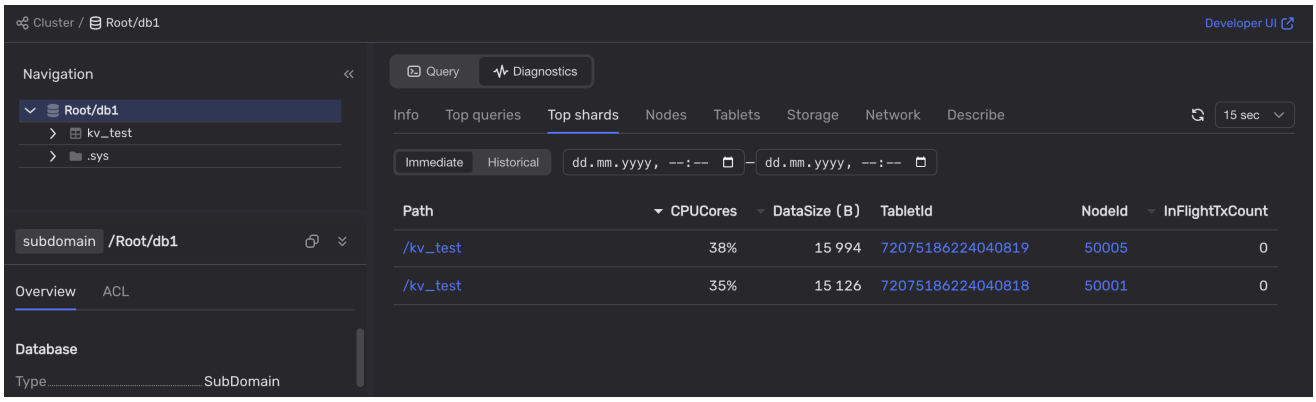
The diagram shows that the number of data shards increased at about `10:28:00`. Based on the bucket color, their workload does not exceed 40%.



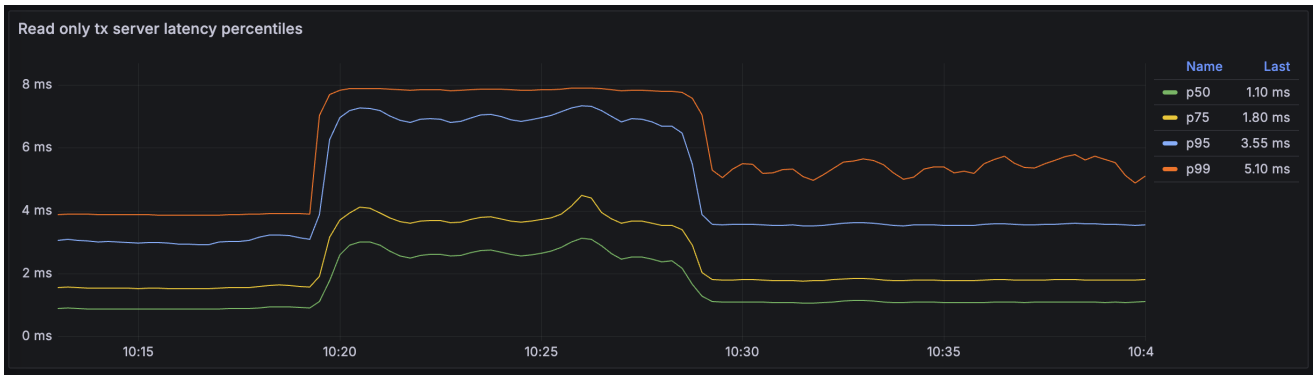
See the diagram description

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:

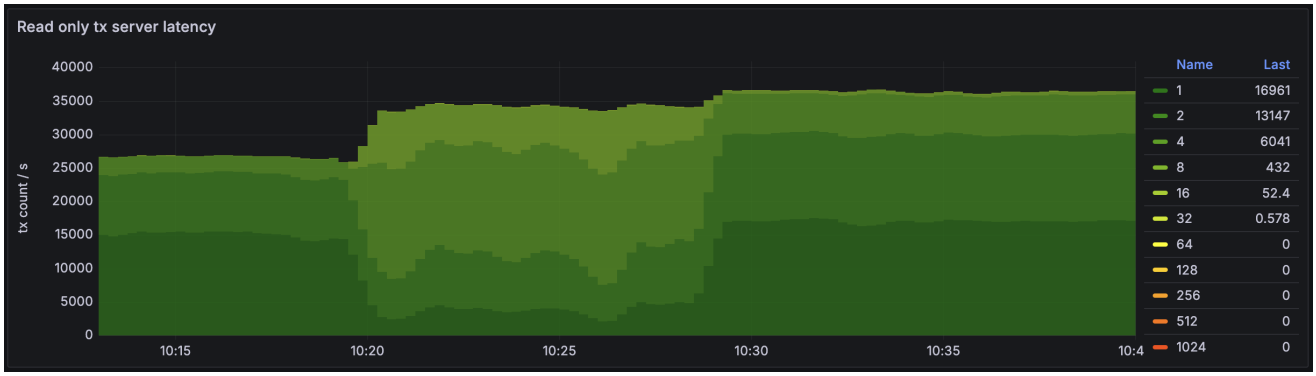


Let's confirm that latencies have returned to normal:



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 × 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

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 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), ...)?

It's better to write it using a JOIN with a constant table:

```
$keys = AsList(
    AsStruct(1 AS Key1, "One" AS Key2),
    AsStruct(2 AS Key1, "Three" AS Key2),
    AsStruct(4 AS Key1, "One" AS Key2)
);

SELECT t.* FROM AS_TABLE($keys) AS k
INNER JOIN table1 AS t
ON t.Key1 = k.Key1 AND t.Key2 = k.Key2;
```

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.

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 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), ...)?

It's better to write it using a JOIN with a constant table:

```
$keys = AsList(
    AsStruct(1 AS Key1, "One" AS Key2),
    AsStruct(2 AS Key1, "Three" AS Key2),
    AsStruct(4 AS Key1, "One" AS Key2)
);

SELECT t.* FROM AS_TABLE($keys) AS k
INNER JOIN table1 AS t
ON t.Key1 = k.Key1 AND t.Key2 = k.Key2;
```

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.

Videos

Video recordings from conferences and webinars. 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.

2025

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 streaming and vector search, 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

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

2024

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 | Kubernetes——YDB

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

Kubernetes

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.

The presentation is suitable for people interested in trade-offs during implementation of PostgreSQL-compatible DBMS.

Slides

YDB: dealing with Big Data and moving towards AI

General

YDB is 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 neighbor searches 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

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

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

2023

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 — 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

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, we describe the reasons why distributed SQL databases were created. Illustrate a brief history of Distributed SQL DBMS development, which products have appeared first.

Slides

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

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.

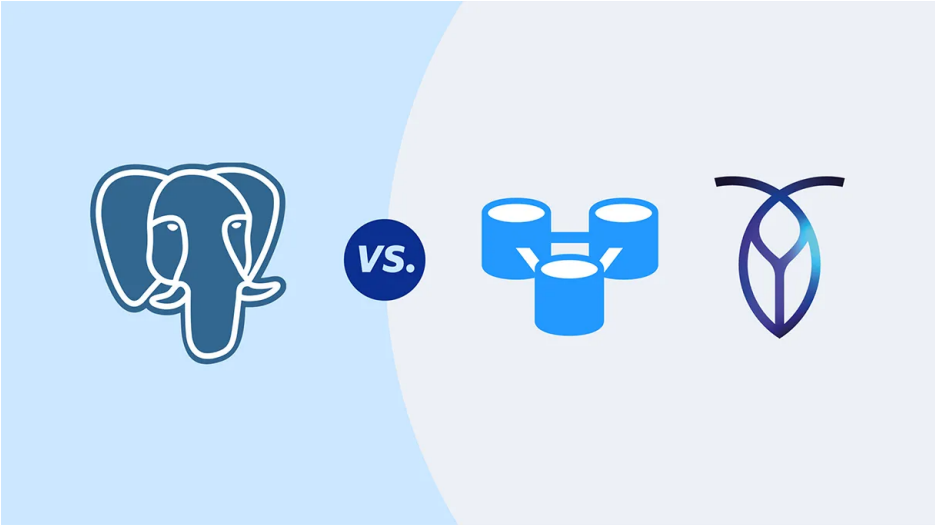
General

– generic materials.

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

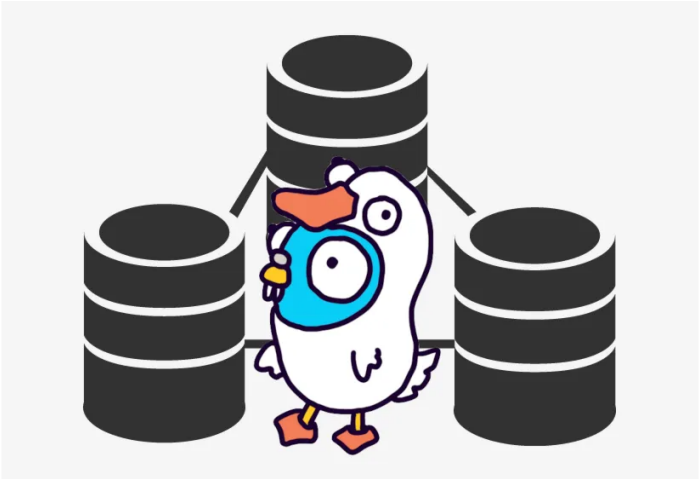


In this post, we present a case study on how we encountered a deadlock with virtual threads in TPC-C for PostgreSQL, even without the dining philosophers problem. This post might be helpful for Java developers who are considering switching to virtual threads.

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

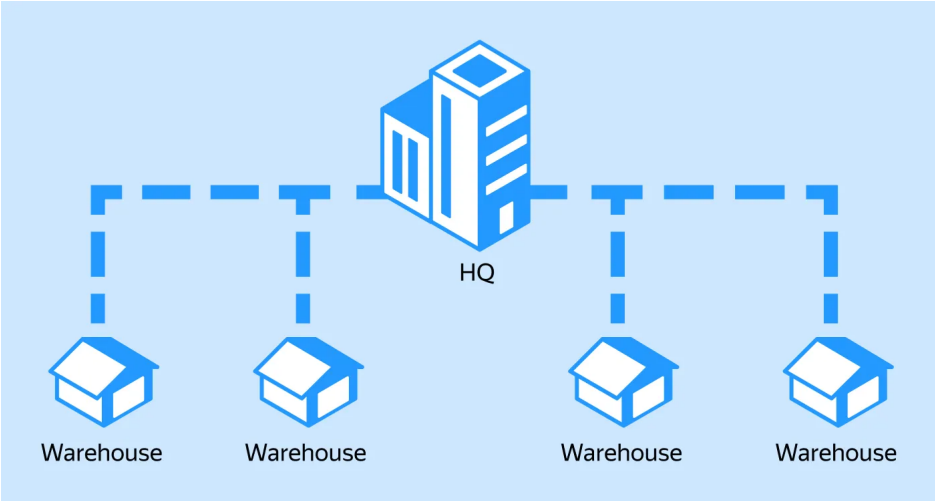
Database internals



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

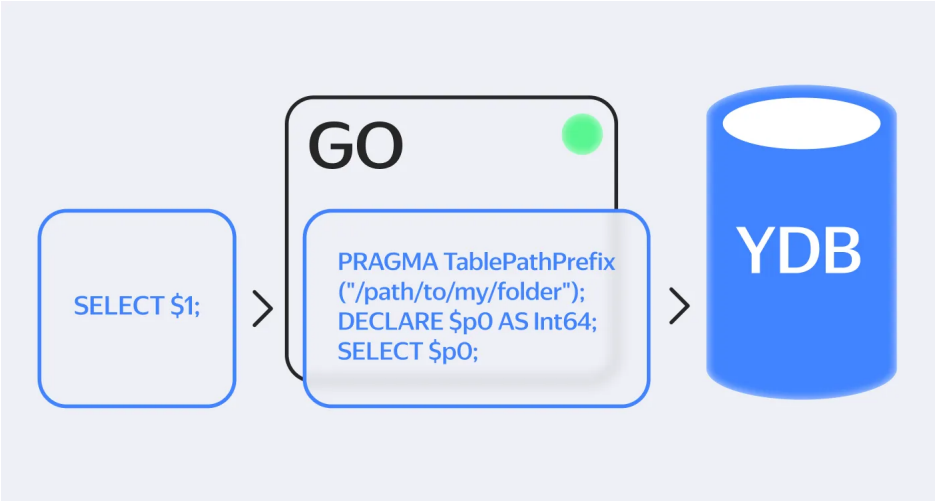
Database internals



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

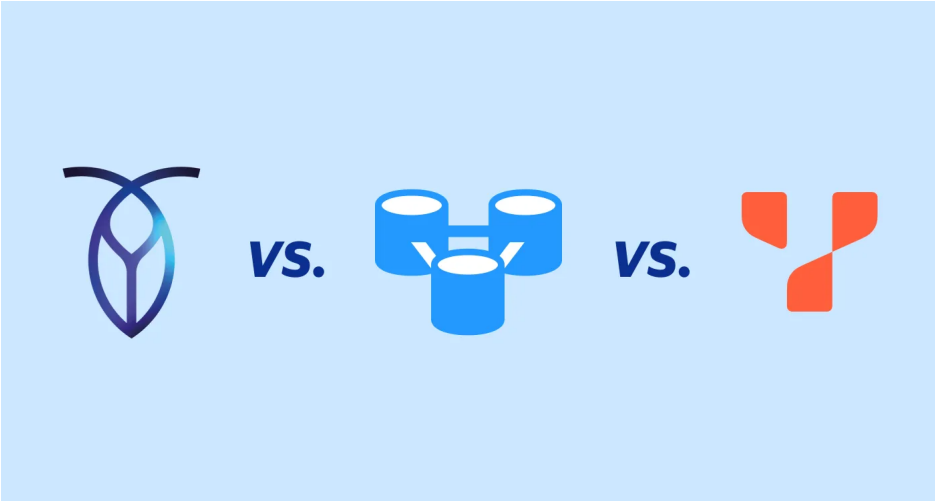
Database internals



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.

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
v24.4			
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	Download	Changelog
v24.4			
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

Version	Release date	Link	Changelog
v24.4			
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
v24.3				
v.24.3.13.11	06.03.2024	• Distribution • Debug symbols	• For distribution • For debug symbols	-
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	
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	

Download YDB CLI

YDB CLI (`ydb`) is a command-line utility for working with YDB databases.

Linux (amd64)

Version	Release date	Download	Changelog
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.2.20.0	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.2.20.0	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.2.20.0	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.2.20.0	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.

Download YDB Ops

YDB Ops (`ydbops`) is a command-line utility for managing YDB clusters.

Linux

Version	Release date	Download	Changelog
v0.0.14	09/12/2024	Binary file	

macOS (Intel)

Version	Release date	Download	Changelog
v0.0.14	09/12/2024	Binary file	

macOS (Apple Silicon)

Version	Release date	Download	Changelog
v0.0.14	09/12/2024	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	Checksums
v0.14	30.11.2024	ydb-ansible-0.14.zip	ydb-ansible-0.14.txt
v0.10	01.08.2024	ydb-ansible-0.10.zip	ydb-ansible-0.10.txt
v0.9	20.07.2024	ydb-ansible-0.9.zip	ydb-ansible-0.9.txt
v0.8	10.07.2024	ydb-ansible-0.8.zip	ydb-ansible-0.7.txt
v0.7	03.05.2024	ydb-ansible-0.7.zip	ydb-ansible-0.7.txt

YDB Server changelog

Version 24.4

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 `KQP` 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.

YDB CLI changelog

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, allowing to perform DDL and DML operations, return with arbitrarily-sized results and support for 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.