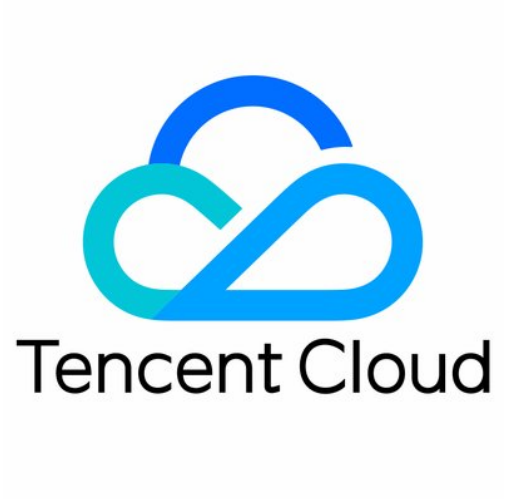
Cloud Data Warehouse

Development Guide

Product Documentation



Contents

Development Guide
Database Engine

Last updated：2024-01-19 16:45:30

By default, ClickHouse uses its own database engine, which provides configurable table engines and all the supported SQL syntax. You can also use the MySQL engine.

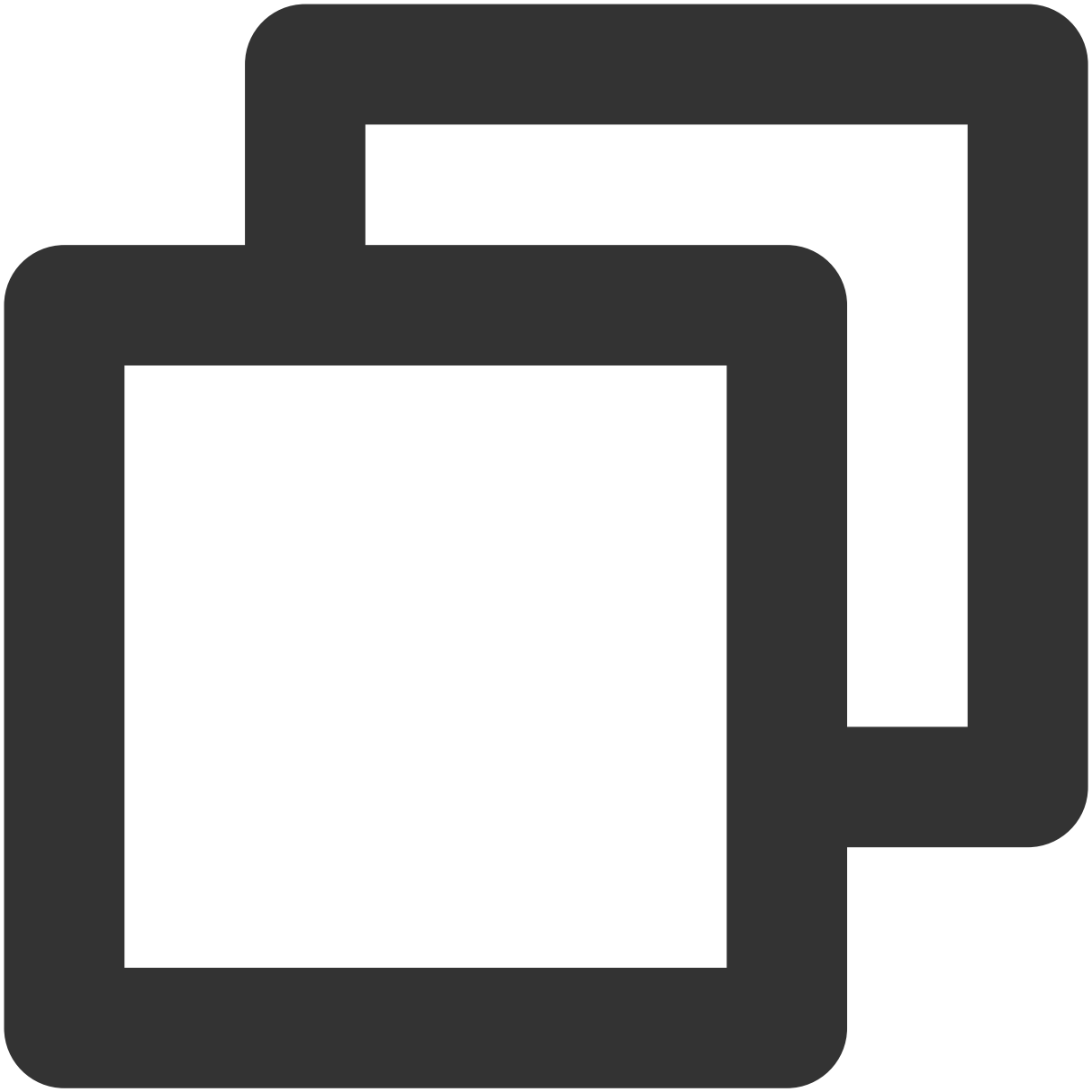
Lazy Engine

It stores a table in memory since the last `expiration_time_in_seconds` (for `\\*Log` engine tables only). Due to the long interval to access this kind of table, it is optimized to store a large number of `\\*Log` engine tables.

MySQL Engine

The MySQL engine is used to map tables from remote MySQL servers to ClickHouse and allow INSERT and SELECT queries on tables to exchange data between ClickHouse and MySQL.
The MySQL engine converts queries to MySQL statements and sends them to the MySQL server, so that you can perform SHOW TABLES and SHOW CREATE TABLE operations. **You cannot perform RENAME, CREATE TABLE, and ALTER operations on them**.

CREATE DATABASE



```
CREATE DATABASE [IF NOT EXISTS] db_name [ON CLUSTER cluster]ENGINE = MySQL('host:p
```

MySQL engine parameters are as described below:

Parameter	Description
host:port	Connected MySQL address
database	Connected MySQL database
user	Connected MySQL user

password	Connected MySQL user password

The types supported by MySQL and ClickHouse are as described below:

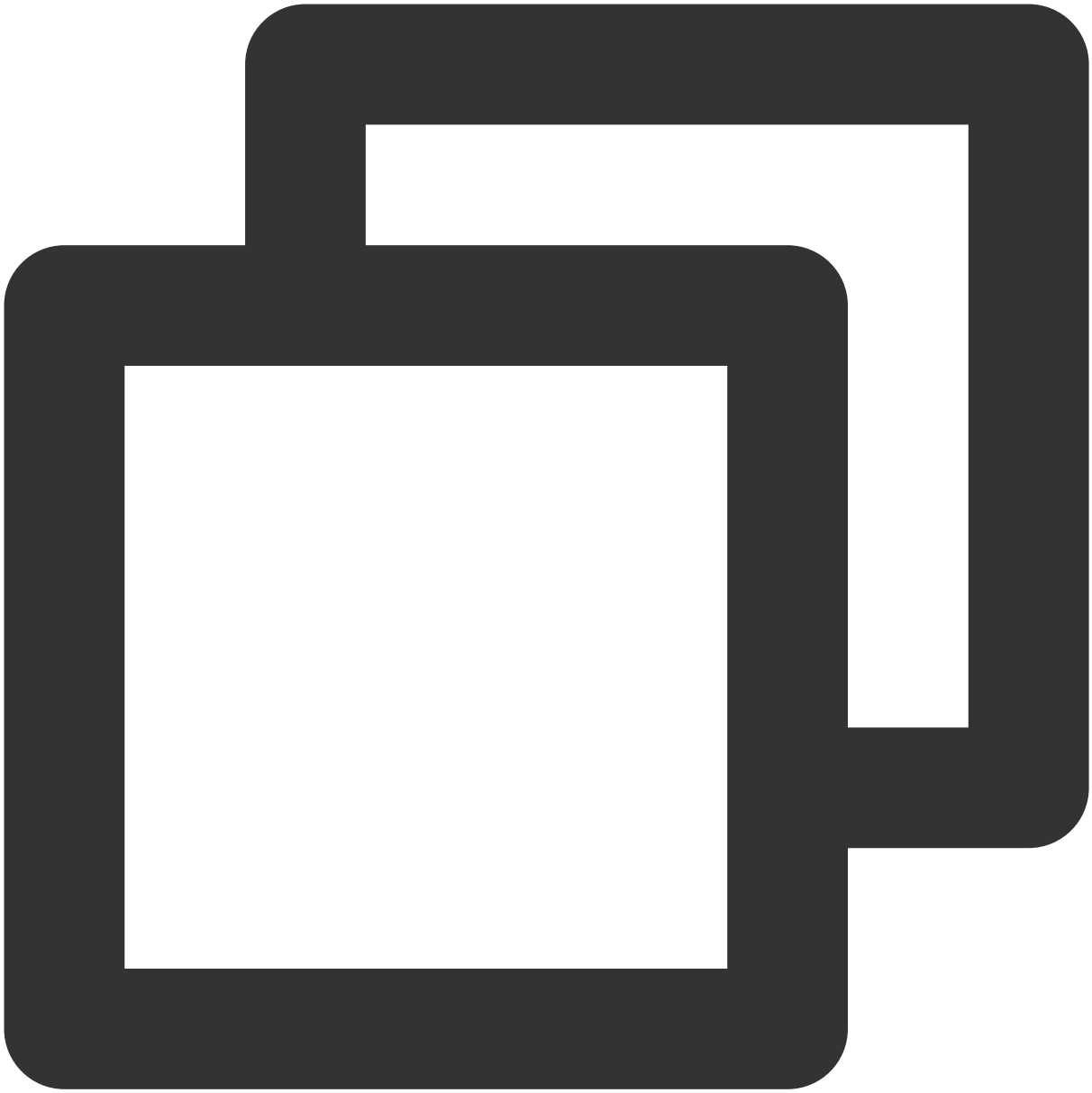
MySQL	ClickHouse
UNSIGNED TINYINT	UInt8
TINYINT	Int8
UNSIGNED SMALLINT	UInt16
SMALLINT	Int16
UNSIGNED INT, UNSIGNED MEDIUMINT	UInt32
INT, MEDIUMINT	Int32
UNSIGNED BIGINT	UInt64
BIGINT	Int64
FLOAT	Float32
DOUBLE	Float64
DATE	Date
DATETIME, TIMESTAMP	Datetime
BINARY	Fixedstring

All other MySQL data types will be converted to string.

All the above types can be nullable.

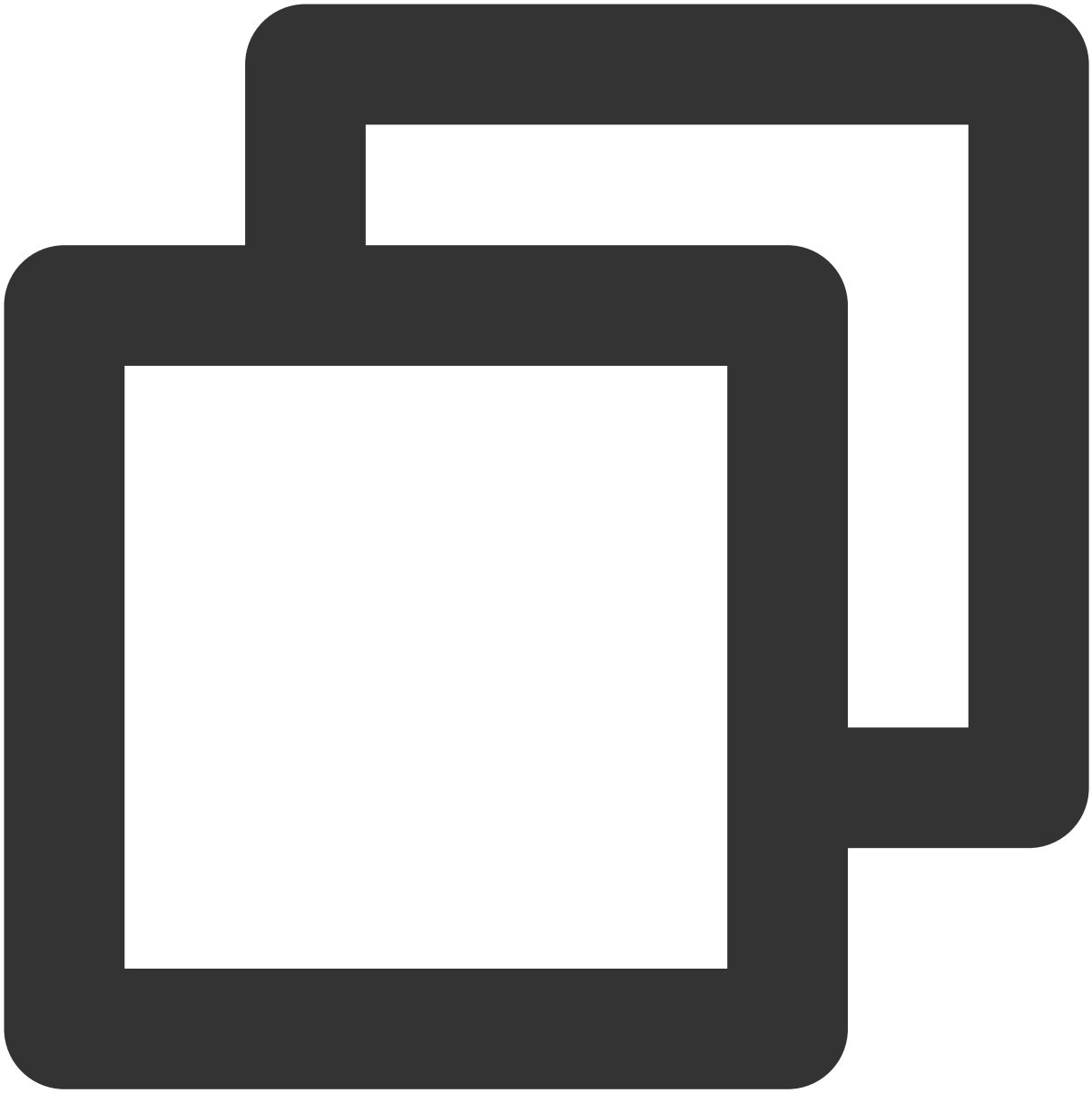
Samples

Create a table in MySQL:



```
mysql> USE test;Database changed
mysql> CREATE TABLE `mysql_table` (
    ->   `int_id` INT NOT NULL AUTO_INCREMENT,
    ->   `float` FLOAT NOT NULL,
    ->   PRIMARY KEY (`int_id`));Query OK, 0 rows affected (0,09 sec)
mysql> insert into mysql_table (`int_id`, `float`) VALUES (1,2);Query OK, 1 row aff
mysql> select * from mysql_table;+--------+-------+| int_id | value |+--------+----
```

Create a database of the MySQL type in ClickHouse and exchange data with the MySQL server:



```
CREATE DATABASE mysql_db ENGINE = MySQL('localhost:3306', 'test', 'my_user', 'user_
 SHOW DATABASES
 ┌─name─────┐
│ default  │
│ mysql_db │
│ system   │
└──────────┘
 SHOW TABLES FROM mysql_db
 ┌─name──────────┐
│  mysql_table │
└───────────────┘
```

```
 SELECT * FROM mysql_db.mysql_table
 ┌─int_id─┬─value─┐
│      1 │     2 │
└────────┴───────┘
 INSERT INTO mysql_db.mysql_table VALUES (3,4)
 SELECT * FROM mysql_db.mysql_table
 ┌─int_id─┬─value─┐
│      1 │     2 │
│      3 │     4 │
└────────┴───────┘
```

Table Engines
Overview

Last updated：2024-01-19 16:45:30

The table engine (i.e., table type) determines the following:
Data storage method and location, and where to write the data to or read the data from
Supported queries and how they are supported
Concurrent data access
Use of indexes (if any)
Whether multi-threaded requests can be executed
Data replication parameters

Engine Types

MergeTree

`MergeTree` engines are the most universal and powerful table engines for high-load tasks. A common feature among them is quick data insertion with subsequent backend data processing. They support data replication (with `Replicated*` versions of engines), partitioning, and some features not supported by other engines. For more information, see MergeTree.
Engines in this family:
MergeTree
ReplacingMergeTree
SummingMergeTree
AggregatingMergeTree
CollapsingMergeTree
VersionedCollapsingMergeTree
GraphiteMergeTree

Log

Log engines are lightweight engines with minimum functionality. They are most effective when you need to quickly write many small tables (up to around 1 million rows) and read them later as a whole.
Engines in this family:
TinyLog
StripeLog
Log

Integration engines

Integration engines are for integration with other data storage and processing systems.
Engines in this family:
Kafka
MySQL
ODBC
JDBC
HDFS

Special engines

Engines in this family:
Distributed
MaterializedView
Dictionary
Merge
File
Null
Set
Join
URL
View
Memory
Buffer

Virtual Columns

Virtual column is an integral table engine attribute that is defined in the engine source code.
You should not specify virtual columns in the `CREATE TABLE` query and you cannot see them in `SHOW CREATE TABLE` and `DESCRIBE TABLE` query results. Virtual columns are also read-only, so you can't insert data into them.
To select data from a virtual column, you must specify its name in the `SELECT` query. `SELECT *` does not return data from virtual columns.
If you create a table with a column that has the same name as a table virtual column, the virtual column becomes inaccessible. To avoid such conflicts, virtual column names are usually prefixed with an underscore.

MergeTree

Last updated：2024-01-19 16:45:30

The `MergeTree` table engine is used to analyze a very large amount of data. It supports data partitioning, primary key indexing, sparse indexing, data TTL, and other features.
Columnar storage: Only reads the required columns, reducing I/O and CPU usage.
Data partitioning: Cuts off data into multiple parts by date or other conditions to simplify data management and query.
Primary key indexing: Sorts data by primary key or sorting key to speed up range queries.
Secondary data-skipping indexes: Skips data that does not meet the conditions based on statistical information such as the minimum and maximum values of a column to further improve query efficiency.
Data merge: Periodically merges small data parts in the background to reduce data redundancy and fragmentation.
The `MergeTree` engine also has some variants, such as `ReplicatedMergeTree`, `AggregatingMergeTree`, and `SummingMergeTree`, which add data replication, data aggregation, data summation, and other features to the basic `MergeTree` features. The `MergeTree` engine supports all SQL syntax in ClickHouse, but there are differences in some features compared with standard SQL.
The following table describes the usage of `MergeTree` and its variants:

Family	Table Engine	Description	Reference
MergeTree	MergeTree	Used to insert a very large amount of data into a table. The data is quickly written to the table part by part, and the parts are merged according to rules.	MergeTree
	ReplacingMergeTree	Used to remove duplicate entries with the same primary key.	ReplacingMergeTree
	CollapsingMergeTree	Used to eliminate the feature limitations of the `ReplacingMergeTree` table engine. It greatly reduces the volume of storage and increases the efficiency of SELECT query as a consequence.	CollapsingMergeTree
	VersionedCollapsingMergeTree	Serves the same purpose as `CollapsingMergeTree`	VersionedCollapsingMergeTr

		but allows retention of data with the latest version.	
	SummingMergeTree	Used to summarize data with the same primary key.	SummingMergeTree
	AggregatingMergeTree	Used to aggregate data with the same primary key.	AggregatingMergeTree

Note

In a production environment, a `Replicated` prefix needs to be added to the table engine name to represent multiple replicas. For more information, see Data Replication.

ReplicatedSummingMergeTree

ReplicatedReplacingMergeTree

ReplicatedAggregatingMergeTree

ReplicatedCollapsingMergeTree

ReplicatedVersionedCollapsingMergeTree

ReplicatedGraphiteMergeTree

ClickHouse SQL Syntax Reference

Last updated：2024-01-19 16:45:30

Data Type

ClickHouse supports multiple data types such as integer, floating point, character, date, enumeration, and array.

Type list

Type	Name	Type ID	Data Range or Description
Integer	1-byte integer	Int8	[-128, 127]
	2-byte integer	Int16	[-32768, 32767]
	4-byte integer	Int32	[-2147483648, 2147483647]
	8-byte integer	Int64	[-9223372036854775808, 9223372036854775807]
	1-byte unsigned integer	UInt8	[0, 255]
	2-byte unsigned integer	UInt16	[0, 65535]
	4-byte unsigned integer	UInt32	[0, 4294967295]
	8-byte unsigned integer	UInt64	[0, 18446744073709551615]
Floating point	Single-precision floating point	Float32	6–7 significant digits
	Double-precision floating point	Float64	15–16 significant digits
	Custom-precision floating point	Decimal32(S)	1–9 significant digits (specified by `S`)
		Decimal64(S)	10–18 significant digits (specified by `S`)
			19–38 significant digits (specified by `S`)

		Decimal128(S)	
Character	Varchar	String	The string length is unlimited
	Char	FixedString(N)	The string length is fixed
	UUID	UUID	The `UUID` is generated by the built-in function `generateUUIDv4`
Time	Date	Date	It stores the year, month, and day in the format of `yyyy-MM-dd`
	Timestamp (s)	DateTime(timezone)	Unix timestamp accurate down to the second
	Timestamp (custom precision)	DateTime(precision, timezone)	You can specify the time precision
Enumeration	1-byte enumeration	Enum8	[-128, 127], 256 values in total
	2-byte enumeration	Enum16	[-32768, 32767], 65536 values in total
Array	Array	Array(T)	It indicates an array consisting of data in `T` type. You are not advised to use nested arrays

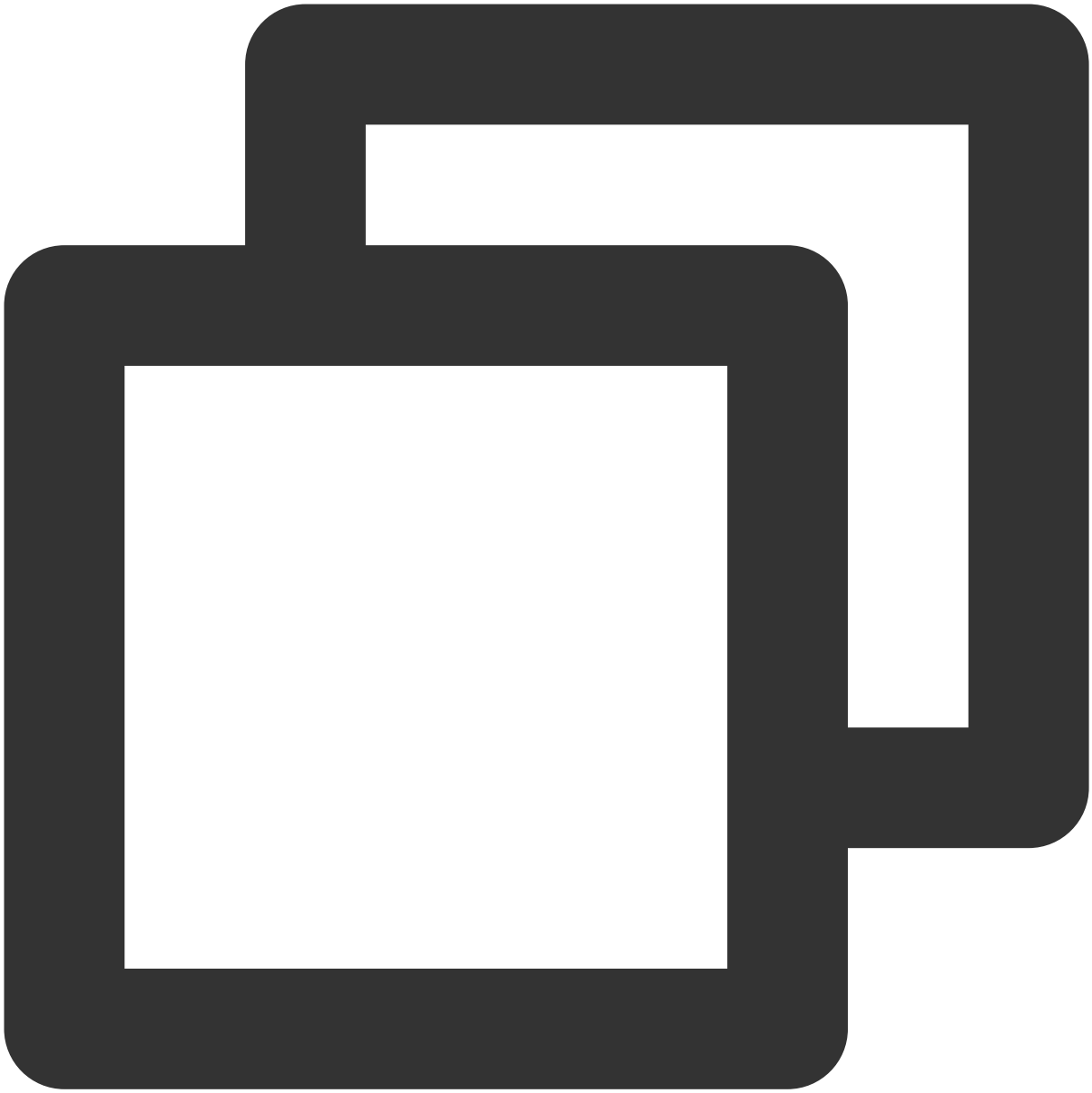
Note:

You can use UInt8 to store boolean values and limit the value to 0 or 1.
For more information on other data types, see the official documentation.

Use cases

Enumeration

The following sample code is used to store the gender information of users in a site:

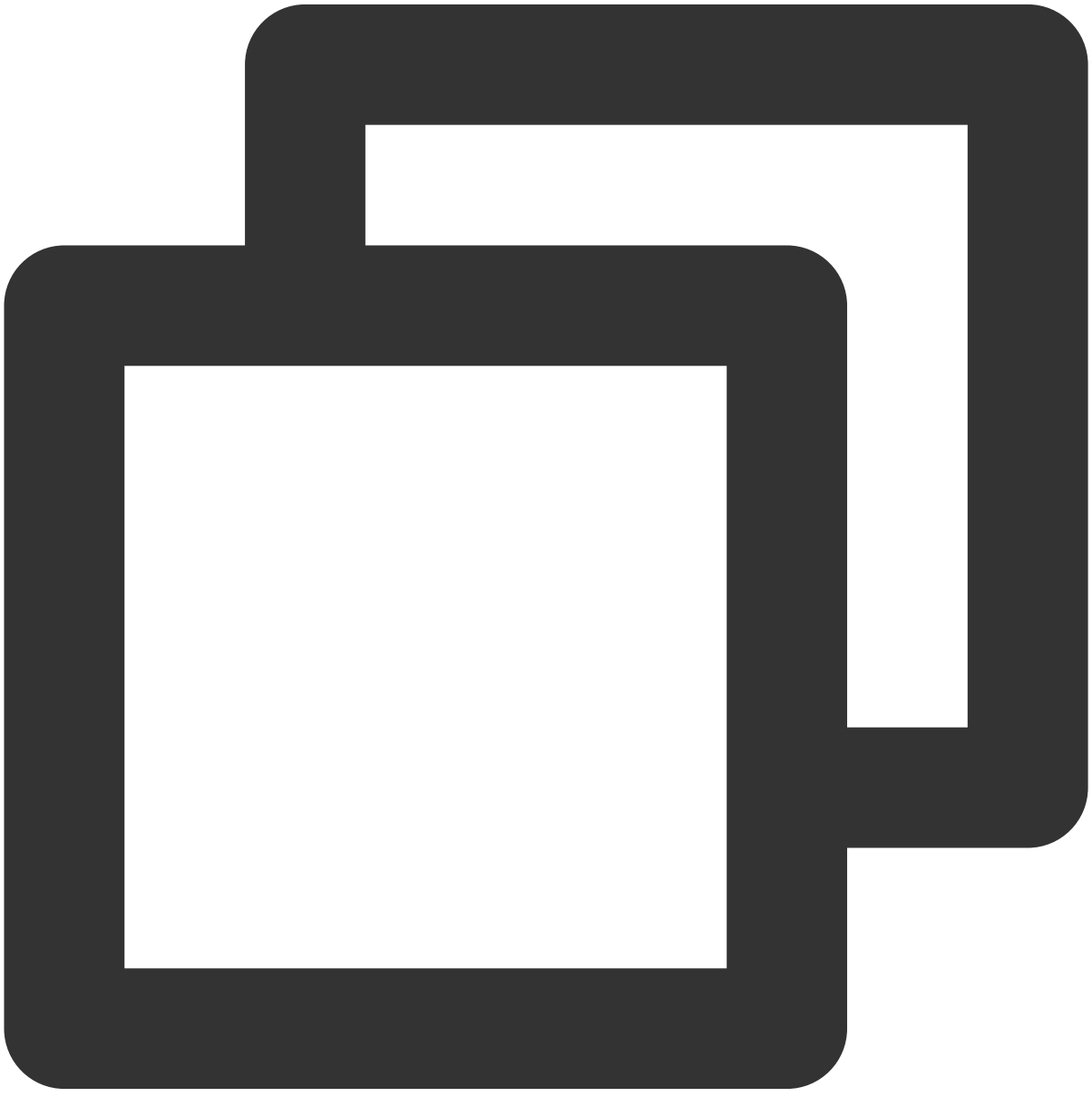


```
CREATE TABLE user (uid Int16, name String, gender Enum('male'=1, 'female'=2)) ENGIN
INSERT INTO user VALUES (1, 'Gary', 'male'), (2, 'Jaco', 'female');
# Query data: SELECT * FROM user;

┌─uid─┬─name─┬─gender─┐
│   1 │ Gary │ male   │
│   2 │ Jaco │ female │
└─────┴──────┴────────┘
# Use the CAST function to query the enumerated integers: SELECT uid, name, CAST(ge

┌─uid─┬─name─┬─CAST(gender, 'Int8')─┐
```

```
|   1 | Gary |                    1 |
|   2 | Jaco |                    2 |
└─────┴──────┴──────────────────────┘
```

Array

The following sample code is used to record the IDs of users who log in to the site every day so as to analyze active users.

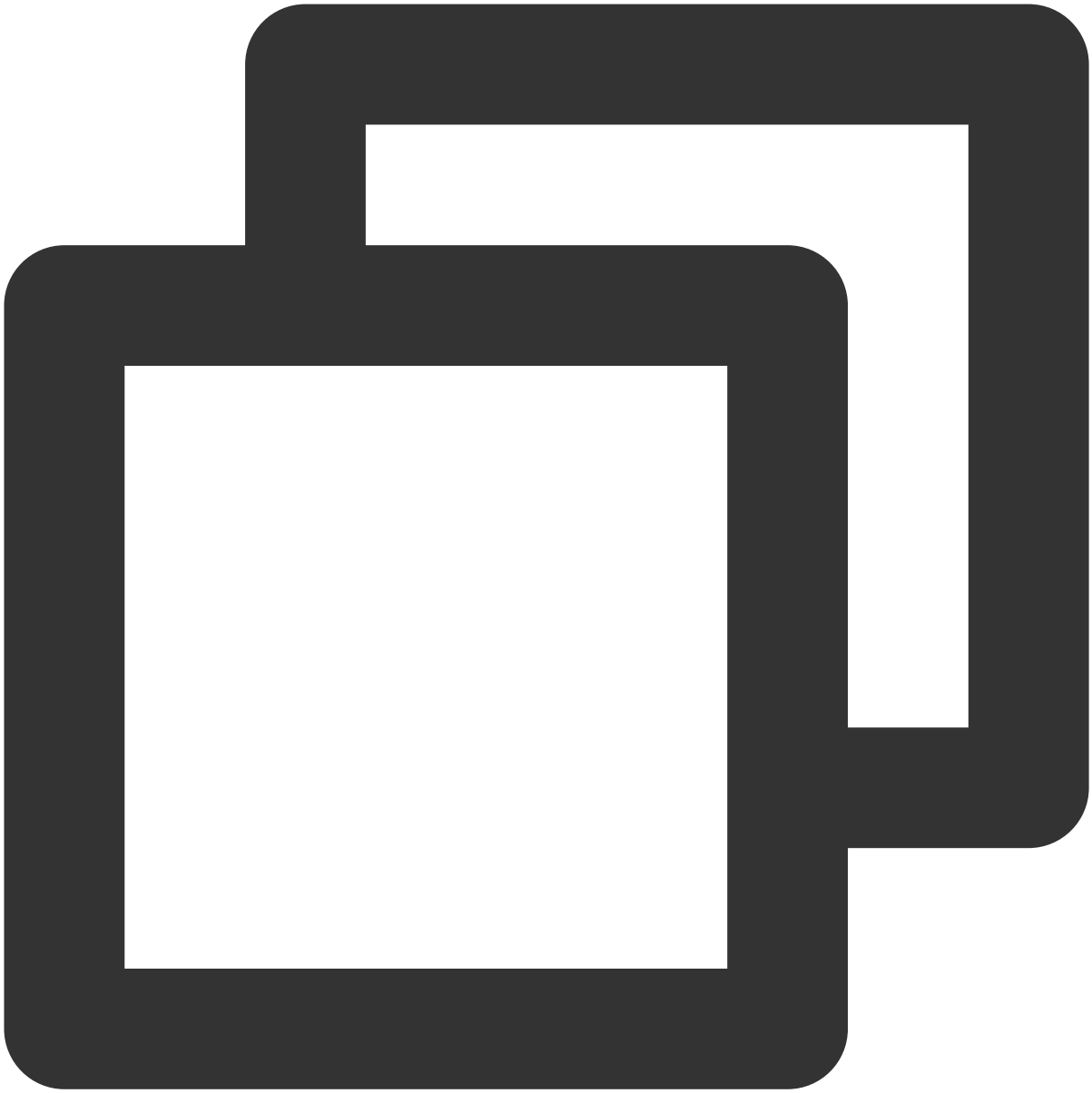


```
CREATE TABLE userloginlog (logindate Date, uids Array(String)) ENGINE=TinyLog;
INSERT INTO userloginlog VALUES ('2020-01-02', ['Gary', 'Jaco']), ('2020-02-03', ['
```

```
# Query result: SELECT * FROM userloginlog;

┌─logindate─┬─uids──────────────┐
│ 2020-01-02 │ ['Gary','Jaco']   │
│ 2020-02-03 │ ['Jaco','Sammie'] │
└────────────┴───────────────────┘
```

Creating Database/Table

ClickHouse uses the `CREATE` statement to create a database or table.



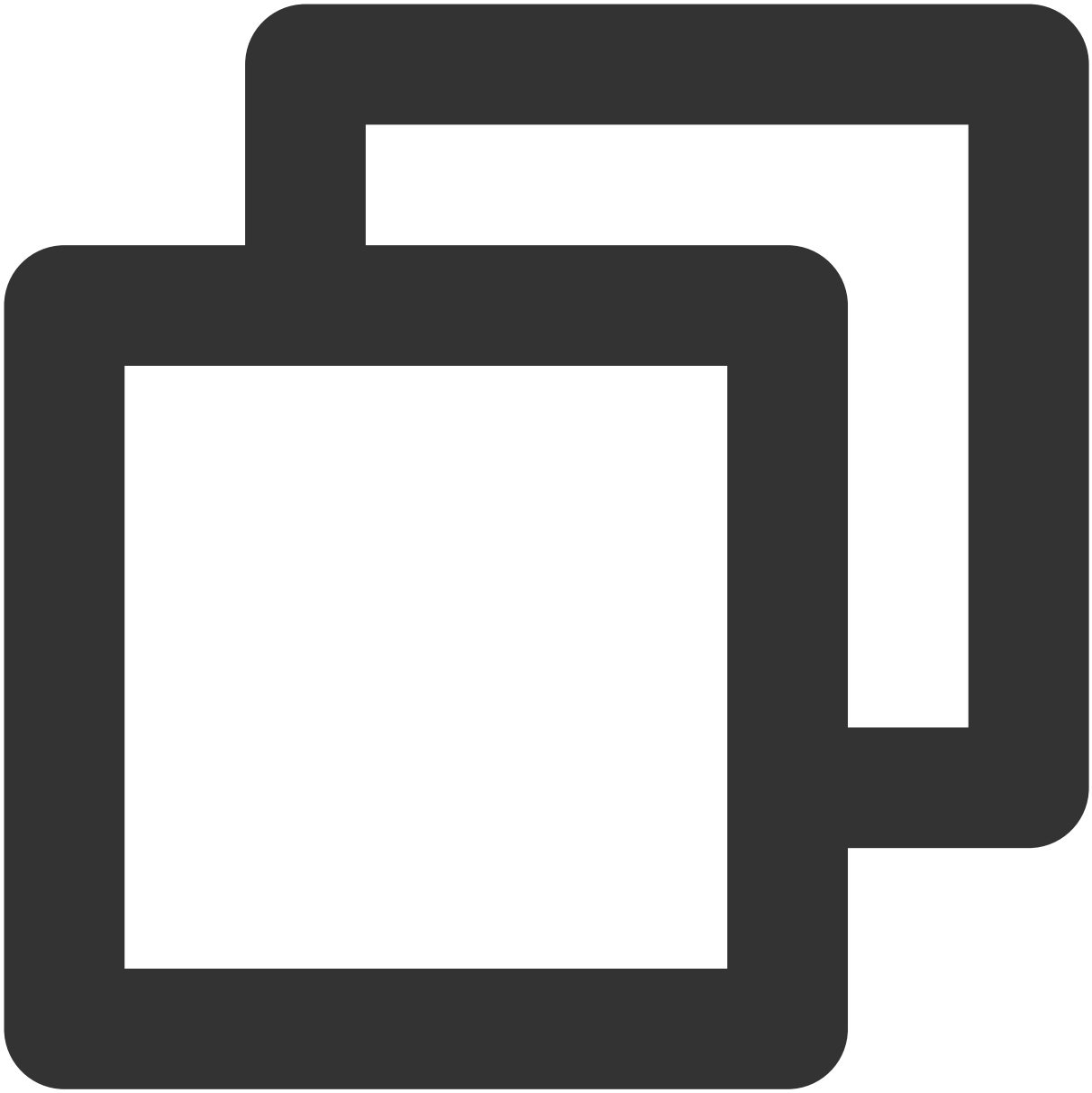
```
CREATE DATABASE [IF NOT EXISTS] db_name [ON CLUSTER cluster] [ENGINE = engine(...)]
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1] [compression_codec] [TTL expr1
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2] [compression_codec] [TTL expr2
    ...
) ENGINE = engine
```

Databases and tables can be created on local disks or in a distributed manner. Distributed creation can be implemented in the following two methods:

Run the `CREATE` statement on all servers where clickhouse-server resides.
Use the ON CLUSTER command to create a database or table on any server in the cluster. When the command is executed successfully, the database or table will be created successfully on each node of the current V-cluster.
If you use clickhouse-client to query a local table of server B on server A, the error "Table xxx doesn't exist." will be reported. If you want that all servers in the cluster can query a table, you are advised to use distributed tables.
For more information, see CREATE Queries.

Query

ClickHouse uses the `SELECT` statement to query data.



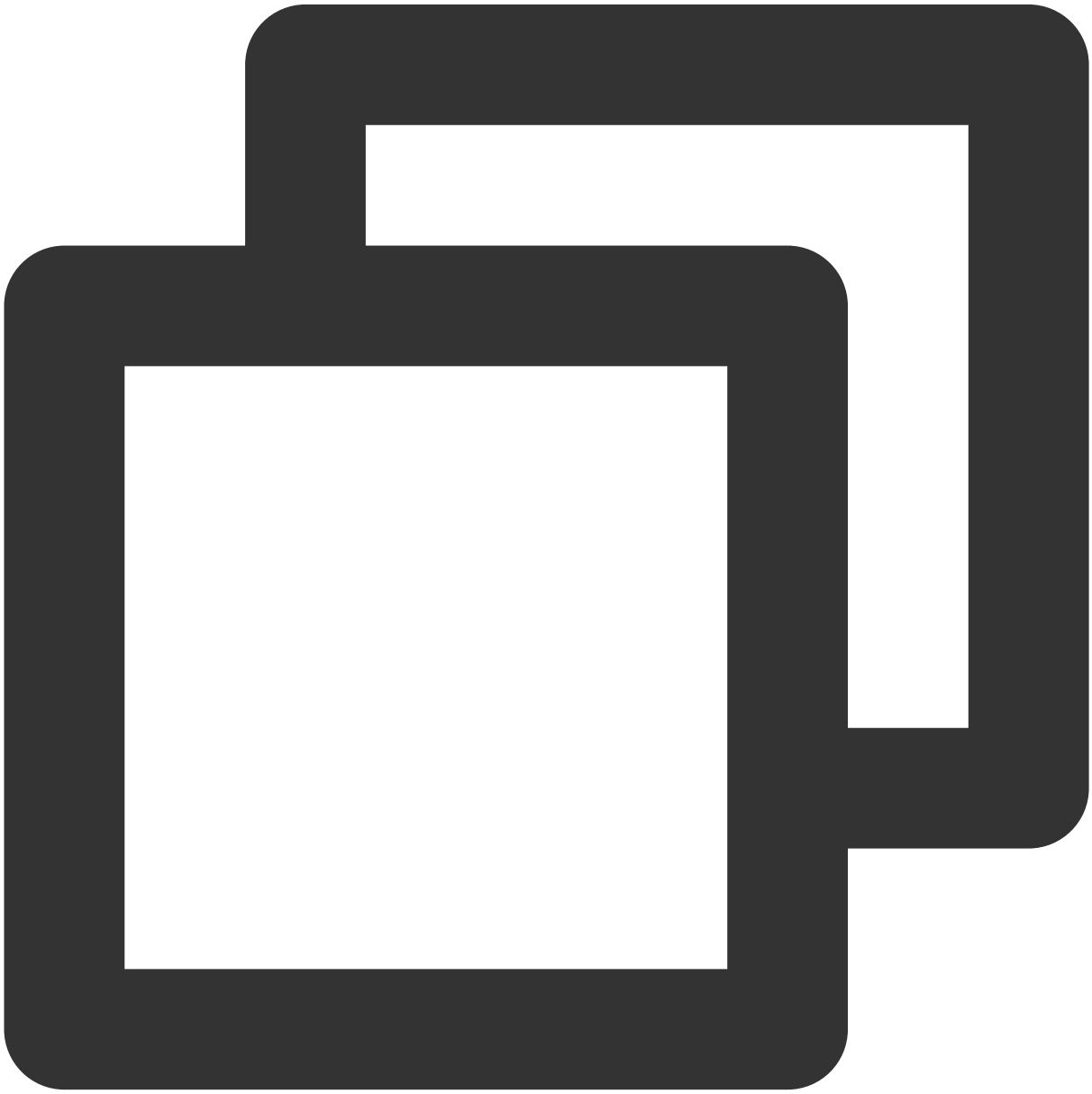
```
SELECT [DISTINCT] expr_list
[FROM [db.]table | (subquery) | table_function] [FINAL]
[SAMPLE sample_coeff]
[GLOBAL] [ANY|ALL] [INNER|LEFT|RIGHT|FULL|CROSS] [OUTER] JOIN (subquery)|table USIN
[PREWHERE expr]
[WHERE expr]
[GROUP BY expr_list] [WITH TOTALS]
[HAVING expr]
[ORDER BY expr_list]
[LIMIT [offset_value, ]n BY columns]
[LIMIT [n, ]m]
```

```
[UNION ALL ...]
[INTO OUTFILE filename]
[FORMAT format]
```

For more information, see SELECT Queries Syntax.

Batch Write

ClickHouse uses the `INSERT INTO` statement to write data.



```
INSERT INTO [db.]table [(c1, c2, c3)] VALUES (v11, v12, v13), (v21, v22, v23), ...
INSERT INTO [db.]table [(c1, c2, c3)] SELECT ...
```

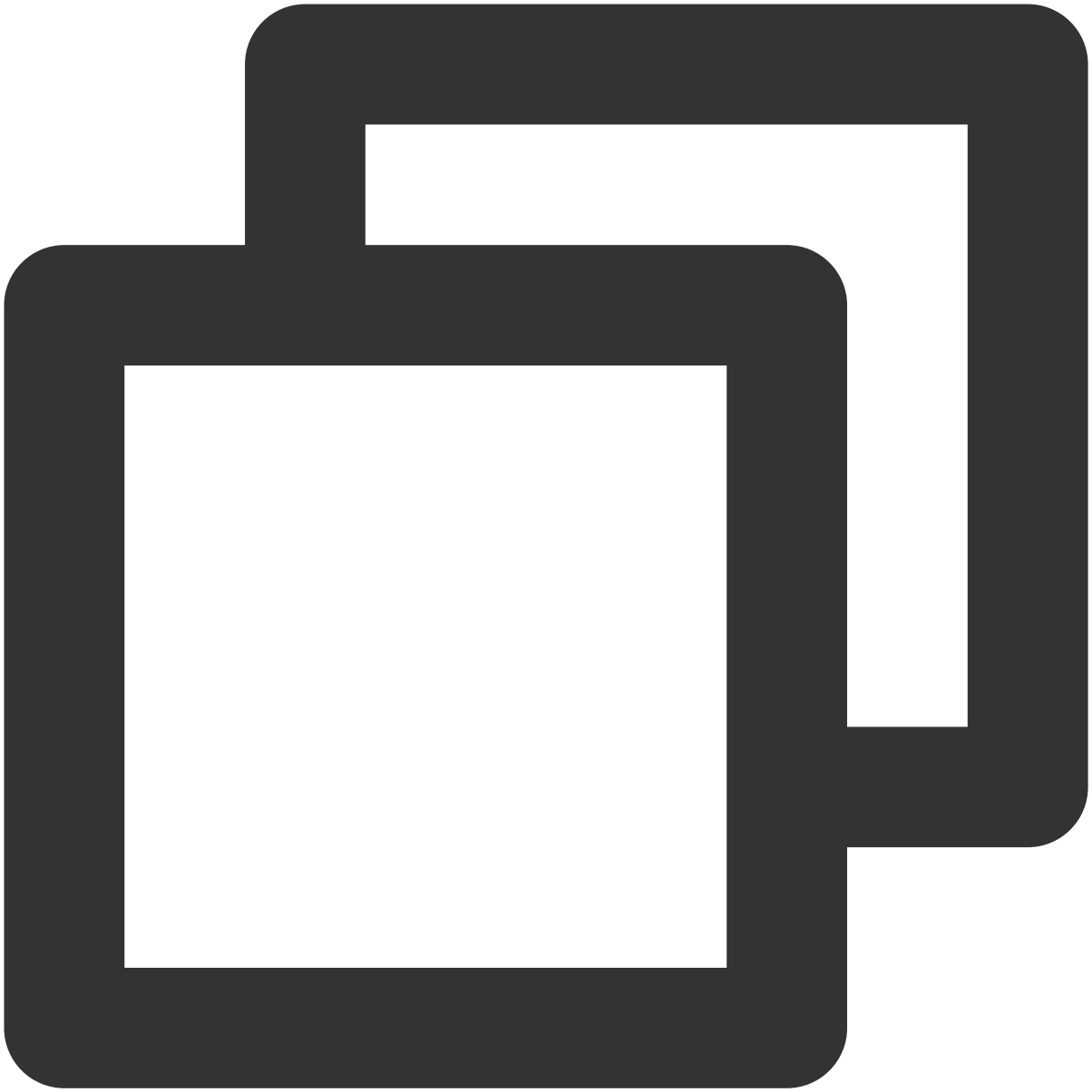
For more information, see INSERT.

Data Deletion

ClickHouse uses the `DROP` or `TRUNCATE` statement to delete data.

Note:

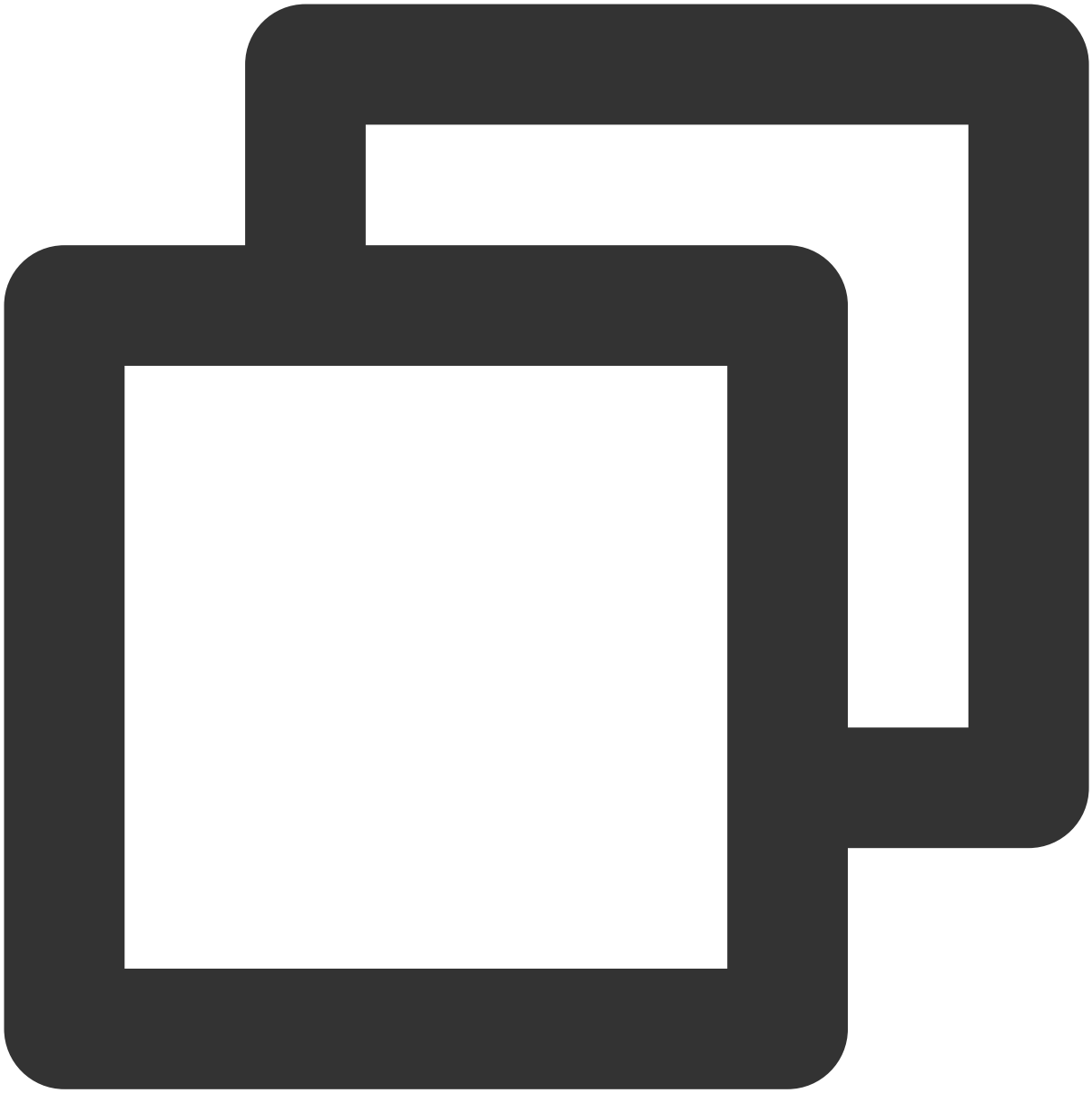
Use `DROP` to delete metadata and data, or use `TRUNCATE` to delete data only.



```
DROP DATABASE [IF EXISTS] db [ON CLUSTER cluster]DROP [TEMPORARY] TABLE [IF EXISTS]
TRUNCATE TABLE [IF EXISTS] [db.]name [ON CLUSTER cluster]
```

Table Structure Modification

ClickHouse uses the `ALTER` statement to modify the table structure.



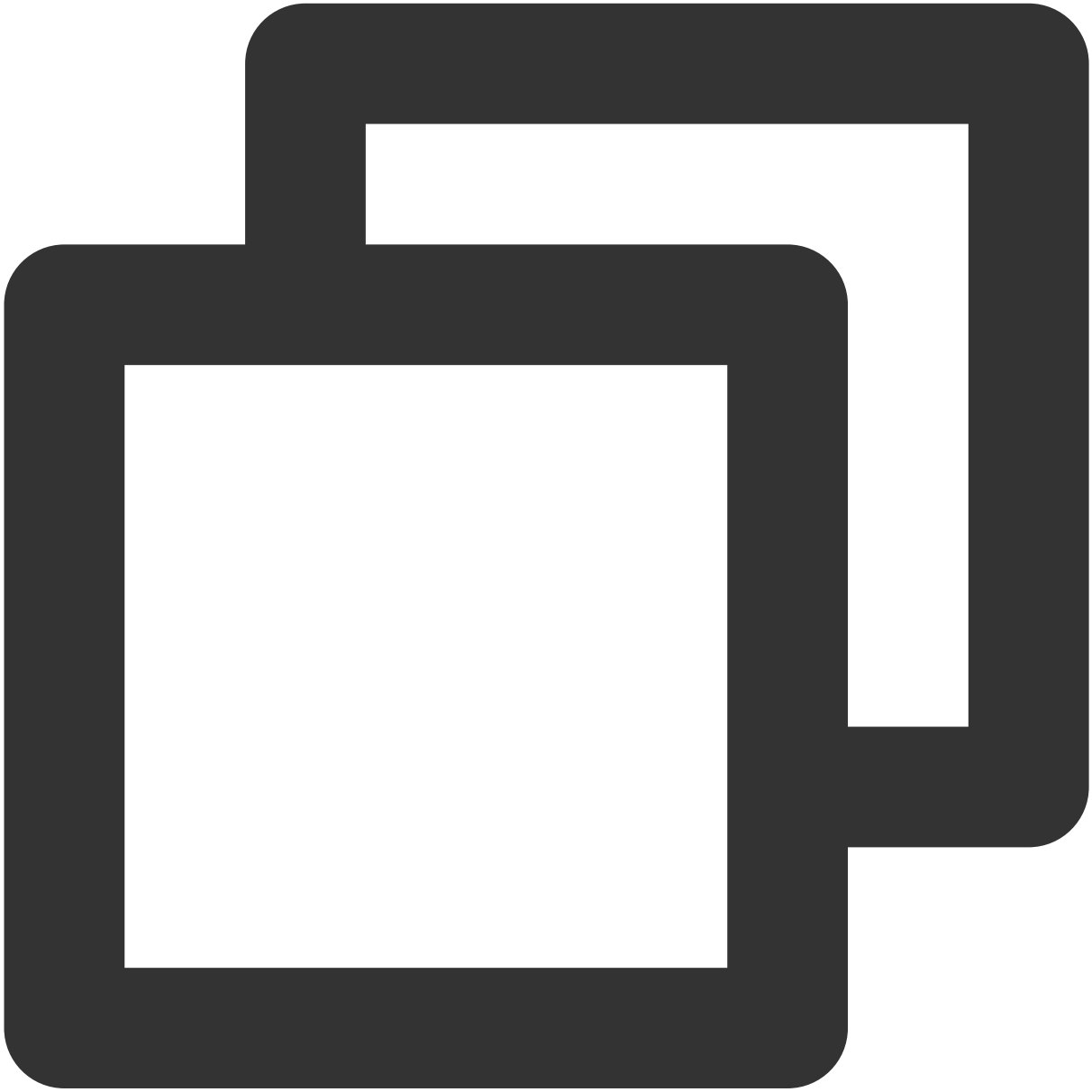
```
# Column operations on table
ALTER TABLE [db].name [ON CLUSTER cluster] ADD COLUMN [IF NOT EXISTS] name [type] [

# Partition operations on table
ALTER TABLE table_name DETACH PARTITION partition_exprALTER TABLE table_name DROP P

# Attribute operations on table
ALTER TABLE table-name MODIFY TTL ttl-expression
```

For more information, see ALTER.

Information Viewing

SHOW statement

It is used to display information such as databases, processing lists, tables, and dictionaries.

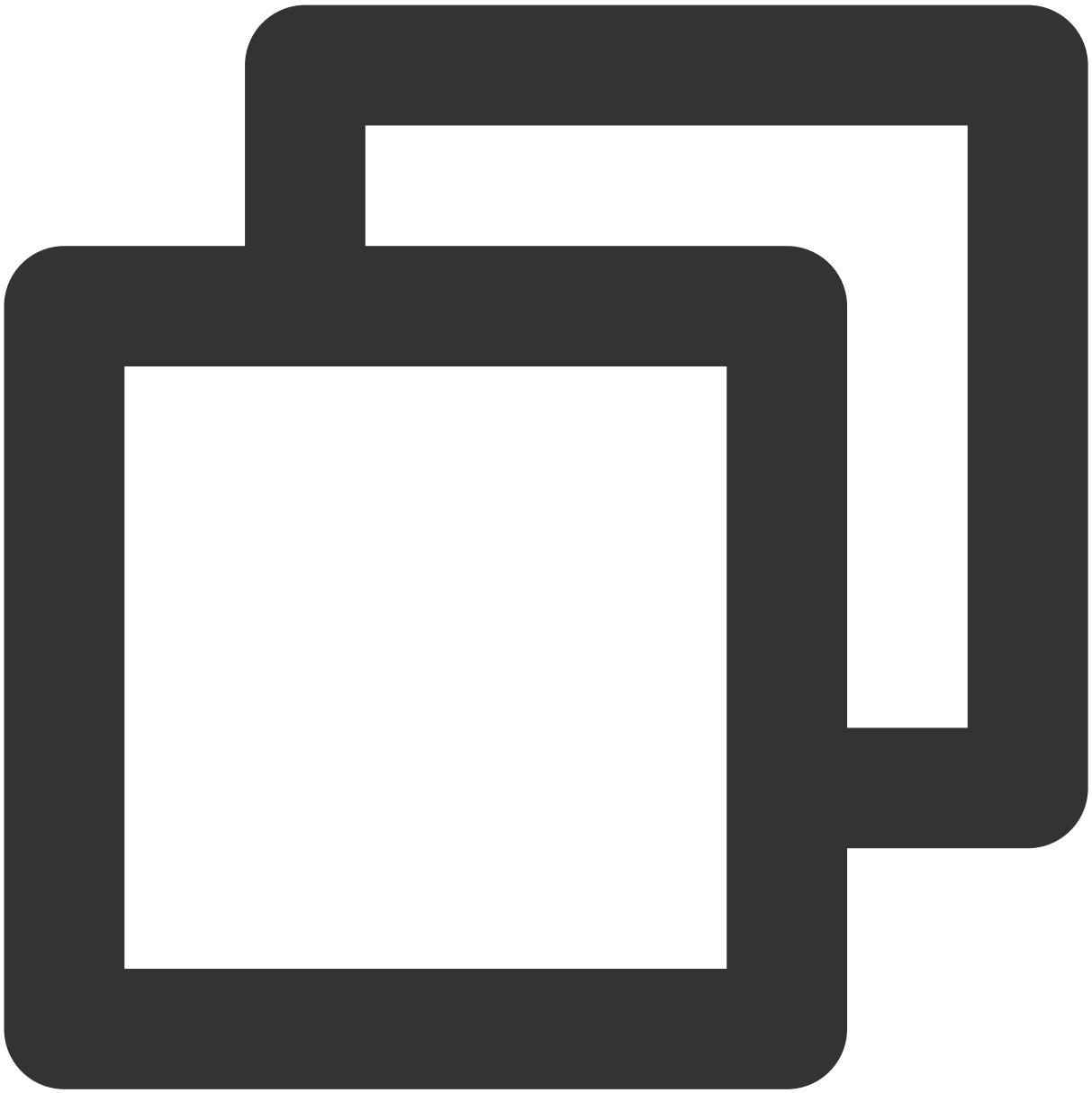


```
SHOW DATABASES [INTO OUTFILE filename] [FORMAT format]SHOW PROCESSLIST [INTO OUTFIL
```

For more information, see SHOW Queries.

DESCRIBE statement

It is used to view table metadata.



```
DESC|DESCRIBE TABLE [db.]table [INTO OUTFILE filename] [FORMAT format]
```

Functions

There are two types of ClickHouse functions: regular functions and aggregate functions. The difference is that a regular function can generate the result for each row, while an aggregate function needs a set of data to generate the

result.

Regular functions

Arithmetic functions

In this type of functions, all fields in the table engage in the arithmetic calculation.

Function Name	Purpose	Use Case
plus(a, b), a + b	Calculates the sum of two fields	plus(table.field1, table.field2)
minus(a, b), a - b	Calculates the difference between two fields	minus(table.field1, table.field2)
multiply(a, b), a * b	Calculates the product of two fields	multiply(table.field1, table.field2)
divide(a, b), a / b	Calculates the quotient of two fields	divide(table.field1, table.field2)
modulo(a, b), a % b	Calculates the remainder between two fields	modulo(table.field1, table.field2)
abs(a)	Calculates absolute value	abs(table.field1)
negate(a)	Calculates opposite	negate(table.field1)

Comparison functions

Function Name	Purpose	Use Case
=, ==	Determines whether the values are the same	table.field1 = value
!=, <>	Determines whether the values are different	table.field1 != value
>	Determines whether the former value is greater than the latter value	table.field1 > value
>=	Determines whether the former value is greater than or equal to the latter value	table.field1 >= value
<	Determines whether the former value is smaller than the latter value	table.field1 < value
<=	Determines whether the former value is smaller than or equal to the latter value	table.field1 <= value

Logical operation functions

Function Name	Purpose	Use Case
AND	Returns result if both two conditions are met	-
OR	Returns result if either condition is met	-
NOT	Returns result if no condition is met	-

Type conversion functions

Overflow may occur when you use a type conversion function. The data types of overflowed values are the same as those in C.

Function Name	Purpose	Use Case
toInt(8|16|32|64)	Converts `String` value to `Int` value	The result of `toInt8('128')` is -127
toUInt(8|16|32|64)	Converts `String` value to `UInt` value	The result of `toUInt8('128')` is 128
toInt(8|16|32|64)OrZero	Converts `String` value to `Int` value and returns 0 if failed	The result of `toInt8OrZero('a')` is 0
toUInt(8|16|32|64)OrZero	Converts `String` value to `UInt` value and returns 0 if failed	The result of `toUInt8OrZero('a')` is 0
toInt(8|16|32|64)OrNull	Converts `String` value to `Int` value and returns `NULL` if failed	The result of `toInt8OrNull('a')` is `NULL`
toUInt(8|16|32|64)OrNull	Converts `String` value to `UInt` value and returns `NULL` if failed	The result of `toUInt8OrNull('a')` is `NULL`

Functions similar to those above are also provided for the floating point and date types.

For more information, see Type Conversion Functions.

Date functions

For more information, see Functions for Working with Dates and Times.

String functions

For more information, see Functions for Working with Strings.

UUID

For more information, see Functions for Working with UUID.

JSON processing functions

For more information, see Functions for Working with JSON.

Aggregate functions

Function Name	Purpose	Use Case
count	Counts the number of rows or non-NULL values	count(expr), COUNT(DISTINCT expr), count(), count(*)
any(x)	Returns the first encountered value. The result is indeterminate	any(column)
anyHeavy(x)	Returns a frequently occurring value using the heavy hitters algorithm. The result is generally indeterminate	anyHeavy(column)
anyLast(x)	Returns the last encountered value. The result is indeterminate	anyLast(column)
groupBitAnd	Applies bitwise `AND`	groupBitAnd(expr)
groupBitOr	Applies bitwise `OR`	groupBitOr(expr)
groupBitXor	Applies bitwise `XOR`	groupBitXor(expr)
groupBitmap	Returns cardinality	groupBitmap(expr)
min(x)	Calculates the minimum value	min(column)
max(x)	Calculates the maximum value	max(x)
argMin(arg, val)	Returns the `arg` value for a minimal `val` value	argMin(c1, c2)
argMax(arg, val)	Returns the `arg` value for a maximum `val` value	argMax(c1, c2)
sum(x)	Calculates the sum	sum(x)
sumWithOverflow(x)	Calculates the sum. If the sum exceeds the maximum value for this data type, the function will return an error	sumWithOverflow(x)

sumMap(key, value)	Sums the `value` array of the same `key` and returns the tuples of `value` and `key` arrays: `keys` in sorted order, and `value` sum of corresponding `keys`	-
skewPop	Calculates skewness	skewPop(expr)
skewSamp	Calculates sample skewness	skewSamp(expr)
kurtPop	Calculates kurtosis	kurtPop(expr)
kurtSamp	Calculates sample kurtosis	kurtSamp(expr)
timeSeriesGroupSum(uid, timestamp, value)	Sums the timestamps in time-series grouped by `uid` . It uses linear interpolation to add missing sample timestamps before summing the values	-
timeSeriesGroupRateSum(uid, ts, val)	Calculates the rate of time-series grouped by `uid` and then sum rates together	-
avg(x)	Calculates the average value	-
uniq	Calculates the approximate number of different values	uniq(x[, ...])
uniqCombined	Calculates the approximate number of different values. Compared with `uniq` , it consumes less memory and is more accurate but has slightly lower performance	uniqCombined(HLL_precision)(x[, ...]), uniqCombined(x[, ...])
uniqCombined64	Functions in the same way as `uniqCombined` but uses 64-bit values to reduce the probability of result value overflow	-
uniqHLL12	Calculates the approximate number of different values. It is not recommended. Please use `uniq` and `uniqCombined` instead	-
uniqExact	Calculates the exact number of different values	uniqExact(x[, ...])
groupArray(x),	Returns an array of `x` values. The array	-

groupArray(max_size)(x)	size can be specified by `max_size`	
groupArrayInsertAt(value, position)	Inserts value into array at specified position	-
groupArrayMovingSum	-	-
groupArrayMovingAvg	-	-
groupUniqArray(x), groupUniqArray(max_size)(x)	-	-
quantile	-	-
quantileDeterministic	-	-
quantileExact	-	-
quantileExactWeighted	-	-
quantileTiming	-	-
quantileTimingWeighted	-	-
quantileTDigest	-	-
quantileTDigestWeighted	-	-
median	-	-
quantiles(level1, level2, ...)(x)	-	-
varSamp(x)	-	-
varPop(x)	-	-
stddevSamp(x)	-	-
stddevPop(x)	-	-
topK(N)(x)	-	-
topKWeighted	-	-
covarSamp(x, y)	-	-
covarPop(x, y)	-	-
corr(x, y)	-	-

categoricalInformationValue	-	-
simpleLinearRegression	-	-
stochasticLinearRegression	-	-
stochasticLogisticRegression	-	-
groupBitmapAnd	-	-
groupBitmapOr	-	-
groupBitmapXor	-	-

Dictionary

A dictionary is a mapping between a key and attributes and can be used as a function for query, which is simpler and more efficient than the method of combining referencing tables with a `JOIN` clause.
There are two types of data dictionaries: internal and external dictionaries.

Internal dictionary

ClickHouse supports one type of internal dictionaries, i.e., geobase. For more information on the supported functions, see Functions for Working with Yandex.Metrica Dictionaries.

External dictionary

ClickHouse allows you to add external dictionaries from multiple data sources. For more information on the supported data sources, see Sources of External Dictionaries.

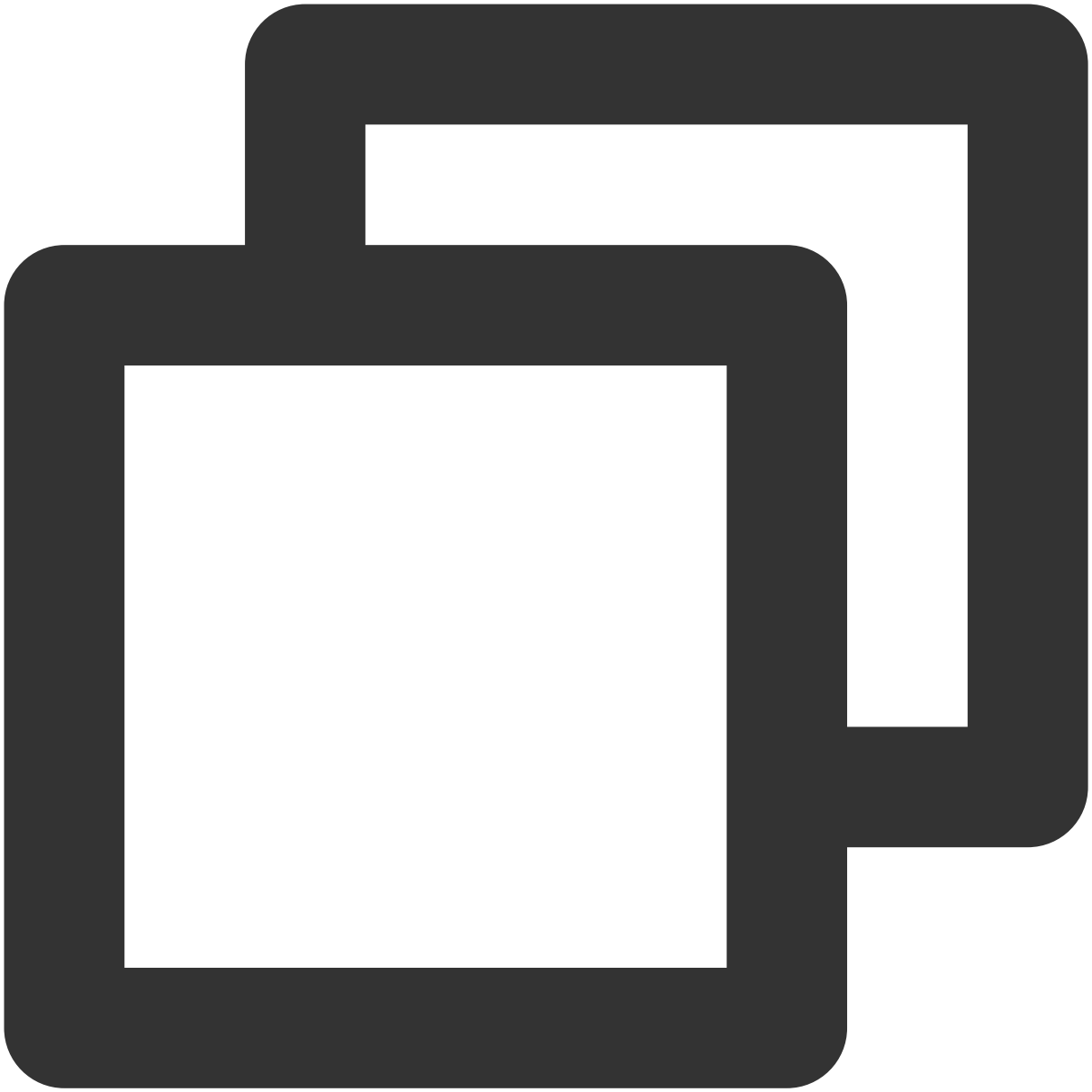
ClickHouse Client Overview

Last updated：2024-01-19 16:45:30

Cloud Data Warehouse provides two types of client APIs over HTTP and TCP protocols respectively.

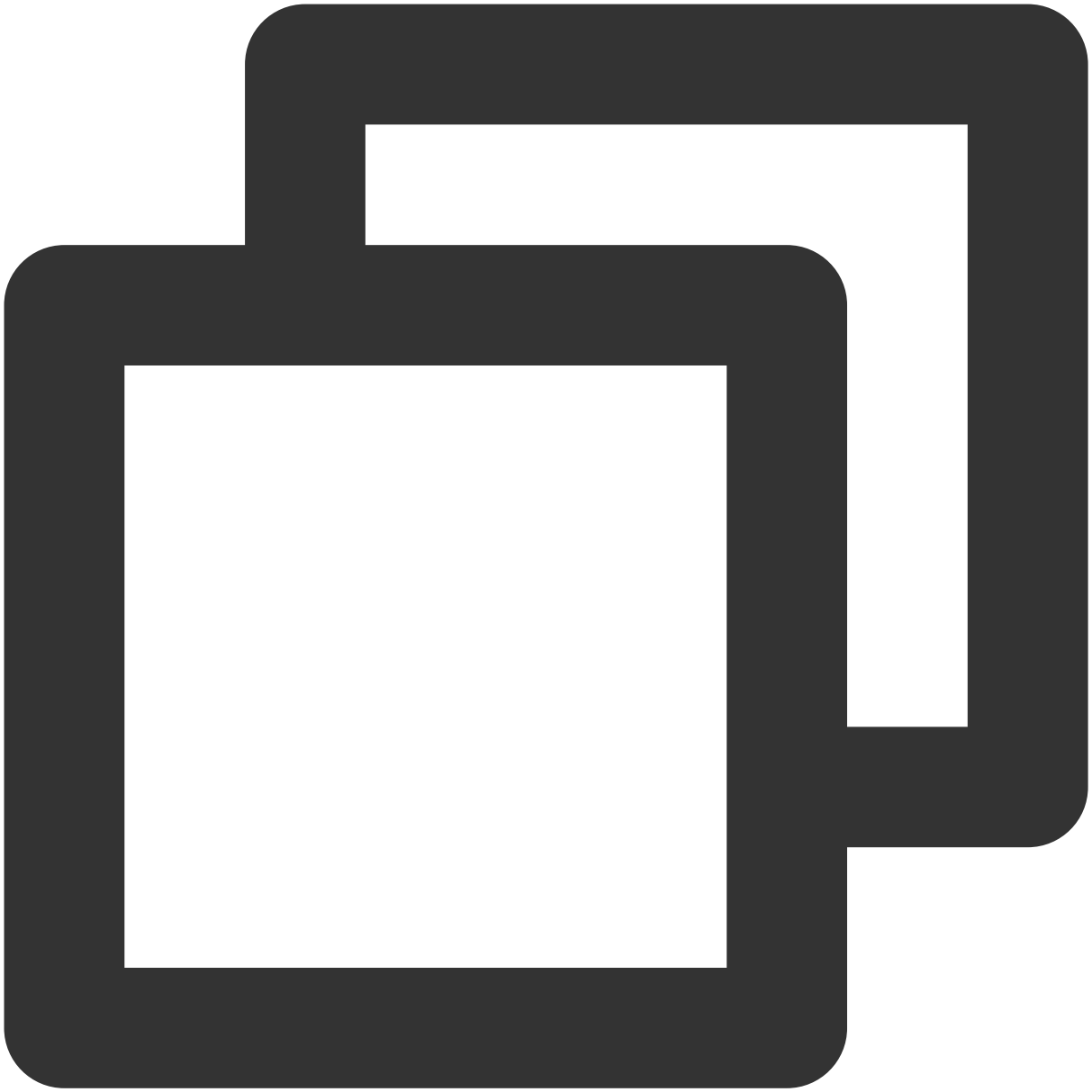
Over HTTP

HTTP is mainly used to support simple lightweight operations and suitable in cross-platform and cross-programming language scenarios. The `clickhouse-server` process on the EMR cluster can start the HTTP service at port 8123 to send simple `GET` requests in order to check whether the service is normal.



```
$ curl http://127.0.0.1:8123Ok.
```

You can also send requests through `query` parameters. For example, the following sample code is to query data in the `account` table in `testdb` :



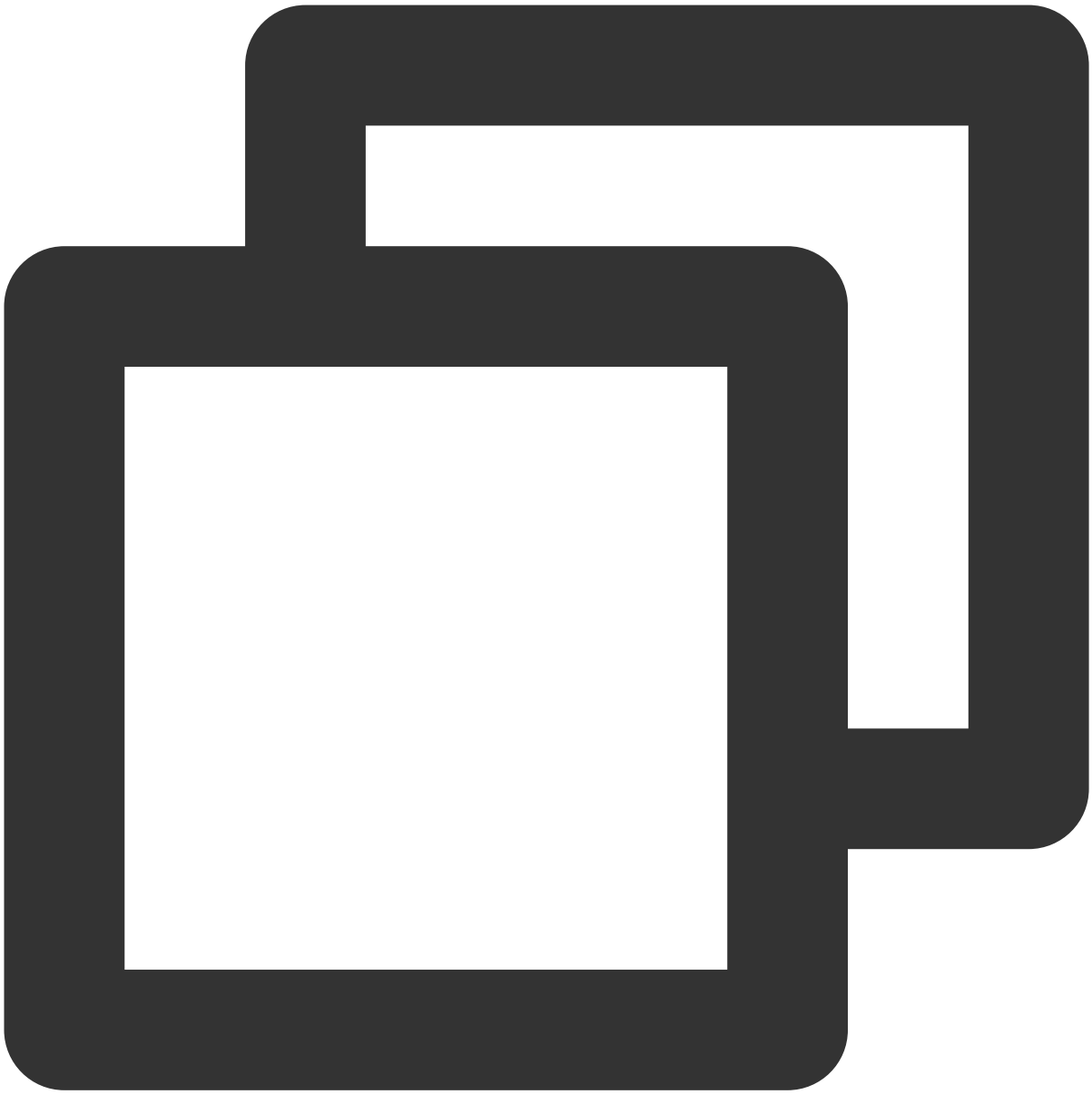
```
$ wget -q -O- 'http://127.0.0.1:8123/?query=SELECT * from testdb.account'1       GH
```

For other methods, see HTTP Interface.

Over TCP

TCP is used mainly on `clickhouse-client` . You can enter the `clickhouse-client` command in the Cloud Data Warehouse cluster to get information such as the version information, address of the connected

`clickhouse-server` , and database used by default. **You can run `quit` , `exit` , or `q` to exit.**



```
$ clickhouse-client
ClickHouse client version 19.16.12.49.
Connecting to localhost:9000 as user default.
Connected to ClickHouse server version 19.16.12 revision 54427.
```

Main parameters used by ClickHouse client:

Parameter	Description

-C --config-file	Specifies the configuration file used by the client
-h --host	Specifies the ClickHouse server IP address
--port	Specifies the ClickHouse server port address
-u --user	Username
--password	Password
-d --database	Database name
-V --version	Displays the client version
-E --vertical	Displays query results in vertical format
-q --query	Passes in SQL statements in non-interactive mode
-t --time	Displays the execution time in non-interactive mode
--log-level	Client log level
--send_logs_level	Specifies the level of log data returned by the server
--server_logs_file	Specifies the server-side log storage path

For more parameters, see Command-line Client.

Self-Built ClickHouse Migration Solution

Last updated：2024-01-19 16:45:30

You can use clickhouse-copier to migrate a CDW cluster.

Migrating a Cluster with clickhouse-copier

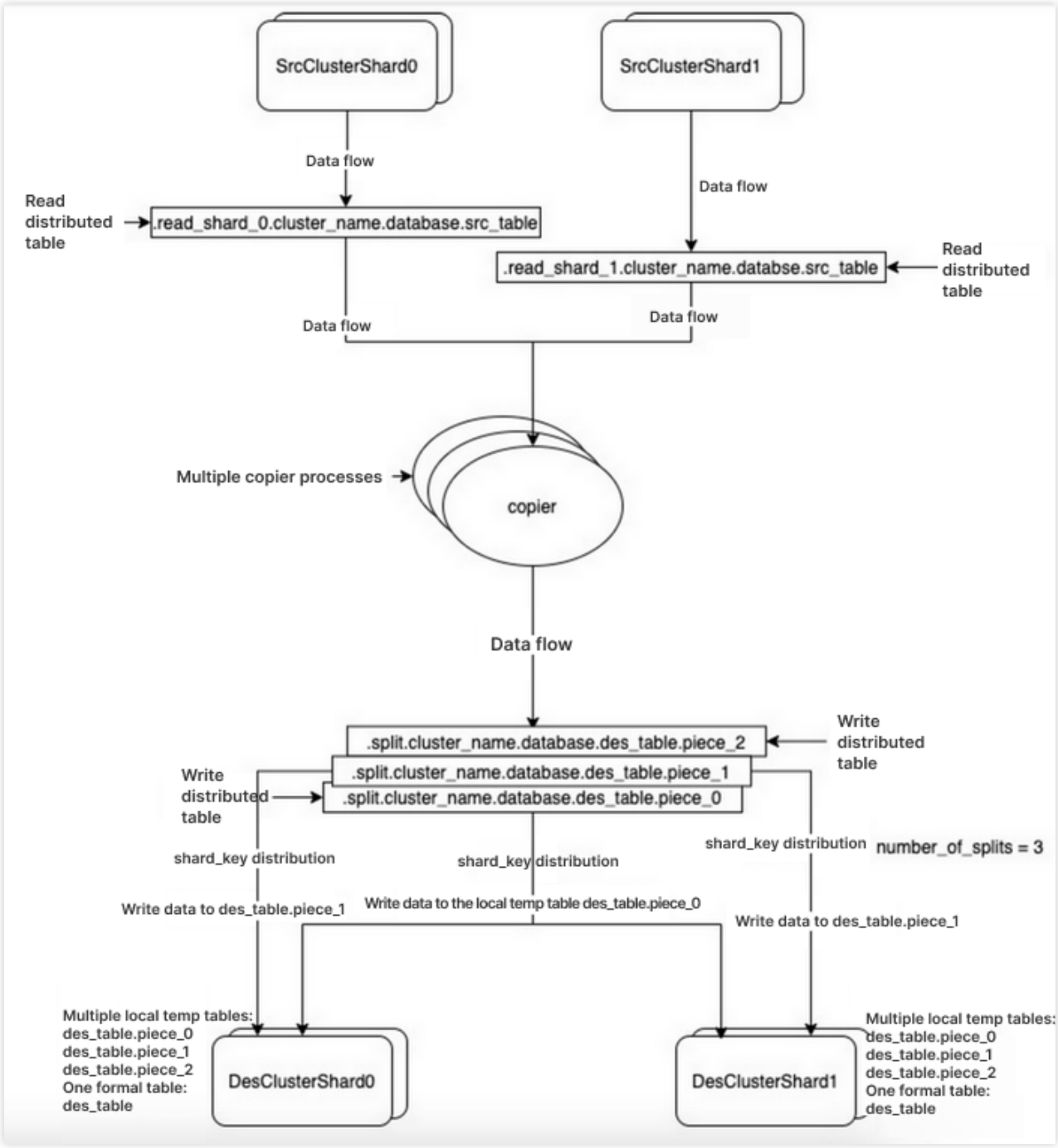
How clickhouse-copier works

clickhouse-copier is ClickHouse's official data migration tool, which migrates a cluster or redistributes data based on INSERT operations in distributed tables.

You can run a clickhouse-copier instance to move data between clusters. It can run simple INSERT...SELECT queries to replicate data between tables with different engine parameters and clusters with different numbers of shards. In the task configuration file, you need to describe the layouts of the source and target clusters and list the tables to be replicated. You can replicate an entire table or certain partitions. clickhouse-copier uses temp distributed tables to select data from the source cluster and insert it to the target cluster.

clickhouse-copier configures the source and target cluster information as well as the distribution logic of the migrated tables to construct a migration task in each shard and partition. It reads the data from each partition according to certain rules, inserts the read data to temp tables in the target cluster, and runs ATTACH PARTITION to attach the data to the final tables. The task execution status is saved to ZooKeeper to implement task restart, checkpoint restart, and co-execution of the same task by multiple clickhouse-copier processes.

As the official migration tool, clickhouse-copier also supports tables with MergeTree engines, including replicated and non-replicated ones.



Strengths of clickhouse-copier

1. clickhouse-copier is ClickHouse's official tool maintained and improved by the community in terms of functionality, performance, and other aspects.

2. Although it is implemented based on the SELECT and INSERT logic and each copier process is executed in a single thread, multiple processes can be used together to process the same task. If one or more processes run on each shard for each copier to handle the local shard data, the overall performance will be improved. The larger the cluster and the more the processes, the better the overall performance.

3. It is compatible with various sharding rules of distributed tables. It reads and writes data completely based on the distributed table logic to control the data write logic (such as retaining the original logic or adjusting the sharing rule) during migration based on the task configuration.
4. It is adaptive to the cluster shard size before and after migration, eliminating your need to map shards.
5. It is not affected by business data writes and MERGE operations on the backend.
6. It can record the status of the migration task in ZooKeeper to support restart upon error and checkpoint restart. This guarantees the migration stability based on the logic of multiple processes.
7. It doesn't depend on replicated tables, so it supports all tables with MergeTree engines.

Notes on clickhouse-copier

1. It is not very easy to use. Many users report in the community that the configuration is complex and difficult, the official documents are insufficient, and best practices need to be summarized.
2. It has a low standalone performance. It can achieve a high performance only when there are many processes and shards. ClickHouse also recommends a large cluster size for a high performance.
3. It cannot migrate materialized views at one stop and can hardly retain the logical relationships of underlying tables and materialized views. You need to carefully design a migration plan based on the table characteristics to accelerate migration and minimize the migration's impact on the business.
4. Although it supports business data writes and merges during migration, it cannot migrate incremental data if the business doesn't stop writing data. It doesn't support DDL operations either.
5. It doesn't support non-MergeTree table engines.