

What is ClickHouse?

ClickHouse is a column-oriented database management system (DBMS) for online analytical processing of queries (OLAP).

In a "normal" row-oriented DBMS, data is stored in this order:

Row	WatchID	JavaEnable	Title	GoodEvent	EventTime
#0	89354350662	1	Investor Relations	1	2016-05-18 05:19:20
#1	90329509958	0	Contact us	1	2016-05-18 08:10:20
#2	89953706054	1	Mission	1	2016-05-18 07:38:00
#N

In other words, all the values related to a row are physically stored next to each other.

Examples of a row-oriented DBMS are MySQL, Postgres, and MS SQL Server.

In a column-oriented DBMS, data is stored like this:

Row:	#0	#1	#2	#N
WatchID:	89354350662	90329509958	89953706054	...
JavaEnable:	1	0	1	...
Title:	Investor Relations	Contact us	Mission	...
GoodEvent:	1	1	1	...
EventTime:	2016-05-18 05:19:20	2016-05-18 08:10:20	2016-05-18 07:38:00	...

These examples only show the order that data is arranged in.

The values from different columns are stored separately, and data from the same column is stored together.

Examples of a column-oriented DBMS: Vertica, Paracel (Actian Matrix and Amazon Redshift), Sybase IQ, Exasol, Infobright, InfiniDB, MonetDB (VectorWise and Actian Vector), LucidDB, SAP HANA, Google Dremel, Google PowerDrill, Druid, and kdb+.

Different orders for storing data are better suited to different scenarios.

The data access scenario refers to what queries are made, how often, and in what proportion; how much data is read for each type of query – rows, columns, and bytes; the relationship between reading and updating data; the working size of the data and how locally it is used; whether transactions are used, and how isolated they are; requirements for data replication and logical integrity; requirements for latency and throughput for each type of query, and so on.

The higher the load on the system, the more important it is to customize the system set up to match the requirements of the usage scenario, and the more fine grained this customization becomes. There is no system that is equally well-suited to significantly different scenarios. If a system is adaptable to a wide set of scenarios, under a high load, the system will handle all the scenarios equally poorly, or will work well for just one or few of possible scenarios.

Key Properties of the OLAP scenario

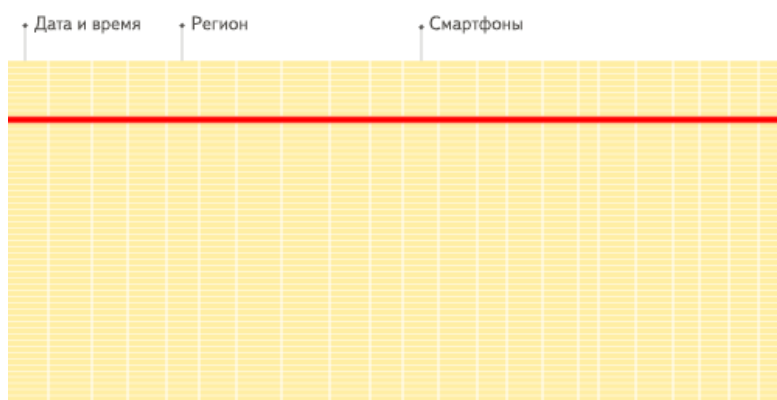
- The vast majority of requests are for read access.
- Data is updated in fairly large batches (> 1000 rows), not by single rows; or it is not updated at all.
- Data is added to the DB but is not modified.
- For reads, quite a large number of rows are extracted from the DB, but only a small subset of columns.
- Tables are "wide," meaning they contain a large number of columns.
- Queries are relatively rare (usually hundreds of queries per server or less per second).
- For simple queries, latencies around 50 ms are allowed.
- Column values are fairly small: numbers and short strings (for example, 60 bytes per URL).
- Requires high throughput when processing a single query (up to billions of rows per second per server).
- Transactions are not necessary.
- Low requirements for data consistency.
- There is one large table per query. All tables are small, except for one.
- A query result is significantly smaller than the source data. In other words, data is filtered or aggregated, so the result fits in a single server's RAM.

It is easy to see that the OLAP scenario is very different from other popular scenarios (such as OLTP or Key-Value access). So it doesn't make sense to try to use OLTP or a Key-Value DB for processing analytical queries if you want to get decent performance. For example, if you try to use MongoDB or Redis for analytics, you will get very poor performance compared to OLAP databases.

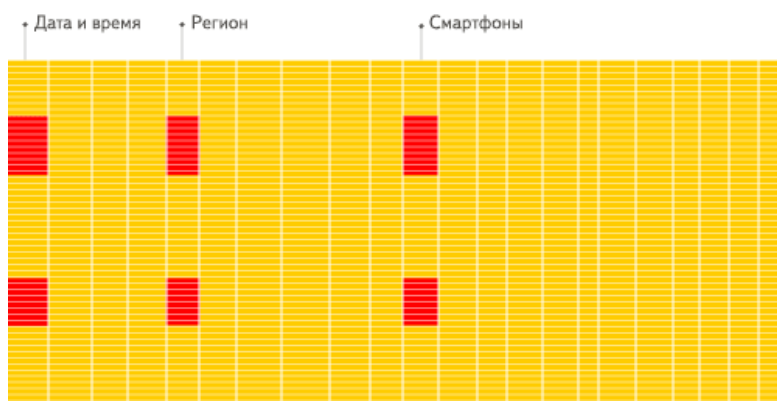
Why Column-Oriented Databases Work Better in the OLAP Scenario

Column-oriented databases are better suited to OLAP scenarios: they are at least 100 times faster in processing most queries. The reasons are explained in detail below, but the fact is easier to demonstrate visually:

Row-oriented DBMS



Column-oriented DBMS



See the difference?

Input/output

1. For an analytical query, only a small number of table columns need to be read. In a column-oriented database, you can read just the data you need. For example, if you need 5 columns out of 100, you can

database, you can read just the data you need. For example, if you need 5 columns out of 100, you can expect a 20-fold reduction in I/O.

2. Since data is read in packets, it is easier to compress. Data in columns is also easier to compress. This further reduces the I/O volume.
3. Due to the reduced I/O, more data fits in the system cache.

For example, the query "count the number of records for each advertising platform" requires reading one "advertising platform ID" column, which takes up 1 byte uncompressed. If most of the traffic was not from advertising platforms, you can expect at least 10-fold compression of this column. When using a quick compression algorithm, data decompression is possible at a speed of at least several gigabytes of uncompressed data per second. In other words, this query can be processed at a speed of approximately several billion rows per second on a single server. This speed is actually achieved in practice.

▼ Example

```
$ clickhouse-client
ClickHouse client version 0.0.52053.
Connecting to localhost:9000.
Connected to ClickHouse server version 0.0.52053.
```

```
:) SELECT CounterID, count() FROM hits GROUP BY CounterID ORDER BY count() DESC LIMIT 20
```

```
SELECT
CounterID,
count()
FROM hits
GROUP BY CounterID
ORDER BY count() DESC
LIMIT 20
```

CounterID	count()
114208	56057344
115080	51619590
3228	44658301
38230	42045932
145263	42042158
91244	38297270
154139	26647572
150748	24112755
242232	21302571
338158	13507087
62180	12229491
82264	12187441
232261	12148031
146272	11438516
168777	11403636
4120072	11227824
10938808	10519739
74088	9047015
115079	8837972
337234	8205961

```
20 rows in set. Elapsed: 0.153 sec. Processed 1.00 billion rows, 4.00 GB (6.53 billion rows/s., 26.10 GB/s.)
```

```
:)
```

CPU

Since executing a query requires processing a large number of rows, it helps to dispatch all operations for entire vectors instead of for separate rows, or to implement the query engine so that there is almost no dispatching cost. If you don't do this, with any half-decent disk subsystem, the query interpreter inevitably stalls the CPU.

It makes sense to both store data in columns and process it, when possible, by columns.

There are two ways to do this:

1. A vector engine. All operations are written for vectors, instead of for separate values. This means you don't need to call operations very often, and dispatching costs are negligible. Operation code contains an optimized internal cycle.
2. Code generation. The code generated for the query has all the indirect calls in it.

This is not done in "normal" databases, because it doesn't make sense when running simple queries. However, there are exceptions. For example, MemSQL uses code generation to reduce latency when processing SQL queries. (For comparison, analytical DBMSs require optimization of throughput, not latency.)

Note that for CPU efficiency, the query language must be declarative (SQL or MDX), or at least a vector (J, K). The query should only contain implicit loops, allowing for optimization.

Distinctive Features of ClickHouse

True Column-Oriented DBMS

In a true column-oriented DBMS, no extra data is stored with the values. Among other things, this means that constant-length values must be supported, to avoid storing their length "number" next to the values. As an example, a billion UInt8-type values should actually consume around 1 GB uncompressed, or this will strongly affect the CPU use. It is very important to store data compactly (without any "garbage") even when uncompressed, since the speed of decompression (CPU usage) depends mainly on the volume of uncompressed data.

This is worth noting because there are systems that can store values of different columns separately, but that can't effectively process analytical queries due to their optimization for other scenarios. Examples are HBase, BigTable, Cassandra, and HyperTable. In these systems, you will get throughput around a hundred thousand rows per second, but not hundreds of millions of rows per second.

It's also worth noting that ClickHouse is a database management system, not a single database. ClickHouse allows creating tables and databases in runtime, loading data, and running queries without reconfiguring and restarting the server.

Data Compression

Some column-oriented DBMSs (InfiniDB CE and MonetDB) do not use data compression. However, data compression does play a key role in achieving excellent performance.

Disk Storage of Data

Keeping data physically sorted by primary key makes it possible to extract data for its specific values or value ranges with low latency, less than few dozen milliseconds. Some column-oriented DBMSs (such as SAP HANA and Google PowerDrill) can only work in RAM. This approach encourages the allocation of a larger hardware budget than is actually necessary for real-time analysis. ClickHouse is designed to work on regular hard drives, which means the cost per GB of data storage is low, but SSD and additional RAM are also fully used if available.

Parallel Processing on Multiple Cores

Large queries are parallelized in a natural way, taking all the necessary resources that are available on the current server.

Distributed Processing on Multiple Servers

Almost none of the columnar DBMSs mentioned above have support for distributed query processing.

In ClickHouse, data can reside on different shards. Each shard can be a group of replicas that are used for fault tolerance. The query is processed on all the shards in parallel. This is transparent for the user.

SQL Support

ClickHouse supports a declarative query language based on SQL that is identical to the SQL standard in many cases.

Supported queries include GROUP BY, ORDER BY, subqueries in FROM, IN, and JOIN clauses, and scalar subqueries.

Dependent subqueries and window functions are not supported.

Vector Engine

Data is not only stored by columns, but is processed by vectors (parts of columns). This allows us to achieve high CPU efficiency.

Real-time Data Updates

ClickHouse supports tables with a primary key. In order to quickly perform queries on the range of the primary key, the data is sorted incrementally using the merge tree. Due to this, data can continually be added to the table. No locks are taken when new data is ingested.

Index

Having a data physically sorted by primary key makes it possible to extract data for its specific values or value ranges with low latency, less than few dozen milliseconds.

Suitable for Online Queries

Low latency means that queries can be processed without delay and without trying to prepare answer in advance, right at the same moment while user interface page is loading. In other words, online.

Support for Approximated Calculations

ClickHouse provides various ways to trade accuracy for performance:

1. Aggregate functions for approximated calculation of the number of distinct values, medians, and quantiles.
2. Running a query based on a part (sample) of data and getting an approximated result. In this case, proportionally less data is retrieved from the disk.
3. Running an aggregation for a limited number of random keys, instead of for all keys. Under certain conditions for key distribution in the data, this provides a reasonably accurate result while using fewer resources.

Data replication and data integrity support

Uses asynchronous multimaster replication. After being written to any available replica, data is distributed to all the remaining replicas in the background. The system maintains identical data on different replicas. Recovery after most failures is performed automatically, and in complex cases — semi-automatically.

For more information, see the section [Data replication](#).

ClickHouse Features that Can be Considered Disadvantages

1. No full-fledged transactions.
2. Lack of ability to modify or delete already inserted data with high rate and low latency. There are batch deletes and updates available to clean up or modify data, for example to comply with [GDPR](#).
3. The sparse index makes ClickHouse not really suitable for point queries retrieving single rows by their keys.

Performance

According to internal testing results at Yandex, ClickHouse shows the best performance (both the highest throughput for long queries and the lowest latency on short queries) for comparable operating scenarios among systems of its class that were available for testing. You can view the test results on a [separate page](#).

This has also been confirmed by numerous independent benchmarks. They are not difficult to find using an

internet search, or you can see [our small collection of related links](#).

Throughput for a Single Large Query

Throughput can be measured in rows per second or in megabytes per second. If the data is placed in the page cache, a query that is not too complex is processed on modern hardware at a speed of approximately 2-10 GB/s of uncompressed data on a single server (for the simplest cases, the speed may reach 30 GB/s). If data is not placed in the page cache, the speed depends on the disk subsystem and the data compression rate. For example, if the disk subsystem allows reading data at 400 MB/s, and the data compression rate is 3, the speed will be around 1.2 GB/s. To get the speed in rows per second, divide the speed in bytes per second by the total size of the columns used in the query. For example, if 10 bytes of columns are extracted, the speed will be around 100-200 million rows per second.

The processing speed increases almost linearly for distributed processing, but only if the number of rows resulting from aggregation or sorting is not too large.

Latency When Processing Short Queries

If a query uses a primary key and does not select too many rows to process (hundreds of thousands), and does not use too many columns, we can expect less than 50 milliseconds of latency (single digits of milliseconds in the best case) if data is placed in the page cache. Otherwise, latency is calculated from the number of seeks. If you use rotating drives, for a system that is not overloaded, the latency is calculated by this formula: seek time (10 ms) * number of columns queried * number of data parts.

Throughput When Processing a Large Quantity of Short Queries

Under the same conditions, ClickHouse can handle several hundred queries per second on a single server (up to several thousand in the best case). Since this scenario is not typical for analytical DBMSs, we recommend expecting a maximum of 100 queries per second.

Performance When Inserting Data

We recommend inserting data in packets of at least 1000 rows, or no more than a single request per second. When inserting to a MergeTree table from a tab-separated dump, the insertion speed will be from 50 to 200 MB/s. If the inserted rows are around 1 Kb in size, the speed will be from 50,000 to 200,000 rows per second. If the rows are small, the performance will be higher in rows per second (on Banner System data -> 500,000 rows per second; on Graphite data -> 1,000,000 rows per second). To improve performance, you can make multiple INSERT queries in parallel, and performance will increase linearly.

Yandex.Metrica Use Case

ClickHouse was originally developed to power [Yandex.Metrica](#), the second largest web analytics platform in the world, and continues to be the core component of this system. With more than 13 trillion records in the database and more than 20 billion events daily, ClickHouse allows generating custom reports on the fly directly from non-aggregated data. This article briefly covers the goals of ClickHouse in the early stages of its development.

Yandex.Metrica builds customized reports on the fly based on hits and sessions, with arbitrary segments defined by the user. This often requires building complex aggregates, such as the number of unique users. New data for building a report is received in real time.

As of April 2014, Yandex.Metrica was tracking about 12 billion events (page views and clicks) daily. All these events must be stored in order to build custom reports. A single query may require scanning millions of rows within a few hundred milliseconds, or hundreds of millions of rows in just a few seconds.

Usage in Yandex.Metrica and Other Yandex Services

ClickHouse is used for multiple purposes in Yandex.Metrica.

Its main task is to build reports in online mode using non-aggregated data. It uses a cluster of 374 servers, which store over 20.3 trillion rows in the database. The volume of compressed data, without counting duplication and replication, is about 2 PB. The volume of uncompressed data (in TSV format) would be approximately 17 PB.

approximately 17 TB.

ClickHouse is also used for:

- Storing data for Session Replay from Yandex.Metrica.
- Processing intermediate data.
- Building global reports with Analytics.
- Running queries for debugging the Yandex.Metrica engine.
- Analyzing logs from the API and the user interface.

ClickHouse has at least a dozen installations in other Yandex services: in search verticals, Market, Direct, business analytics, mobile development, AdFox, personal services, and others.

Aggregated and Non-aggregated Data

There is a popular opinion that in order to effectively calculate statistics, you must aggregate data, since this reduces the volume of data.

But data aggregation is a very limited solution, for the following reasons:

- You must have a pre-defined list of reports the user will need.
- The user can't make custom reports.
- When aggregating a large quantity of keys, the volume of data is not reduced, and aggregation is useless.
- For a large number of reports, there are too many aggregation variations (combinatorial explosion).
- When aggregating keys with high cardinality (such as URLs), the volume of data is not reduced by much (less than twofold).
- For this reason, the volume of data with aggregation might grow instead of shrink.
- Users do not view all the reports we generate for them. A large portion of calculations are useless.
- The logical integrity of data may be violated for various aggregations.

If we do not aggregate anything and work with non-aggregated data, this might actually reduce the volume of calculations.

However, with aggregation, a significant part of the work is taken offline and completed relatively calmly. In contrast, online calculations require calculating as fast as possible, since the user is waiting for the result.

Yandex.Metrica has a specialized system for aggregating data called Metrage, which is used for the majority of reports.

Starting in 2009, Yandex.Metrica also used a specialized OLAP database for non-aggregated data called OLAPServer, which was previously used for the report builder.

OLAPServer worked well for non-aggregated data, but it had many restrictions that did not allow it to be used for all reports as desired. These included the lack of support for data types (only numbers), and the inability to incrementally update data in real-time (it could only be done by rewriting data daily). OLAPServer is not a DBMS, but a specialized DB.

To remove the limitations of OLAPServer and solve the problem of working with non-aggregated data for all reports, we developed the ClickHouse DBMS.

Getting Started

System Requirements

ClickHouse can run on any Linux, FreeBSD or Mac OS X with x86_64 CPU architecture.

Though pre-built binaries are typically compiled to leverage SSE 4.2 instruction set, so unless otherwise stated usage of CPU that supports it becomes an additional system requirement. Here's the command to check if current CPU has support for SSE 4.2:

```
$ grep -q sse4_2 /proc/cpuinfo && echo "SSE 4.2 supported" || echo "SSE 4.2 not supported"
```

Installation

Installation

From DEB Packages

Yandex ClickHouse team recommends using official pre-compiled `deb` packages for Debian or Ubuntu.

To install official packages add the Yandex repository in `/etc/apt/sources.list` or in a separate `/etc/apt/sources.list.d/clickhouse.list` file:

```
deb http://repo.yandex.ru/clickhouse/deb/stable/ main/
```

If you want to use the most recent version, replace `stable` with `testing` (this is not recommended for production environments).

Then run these commands to actually install packages:

```
sudo apt-get install dirmngr # optional
sudo apt-key adv --keyserver hkps://keyserver.ubuntu.com:80 --recv E0C56BD4 # optional
sudo apt-get update
sudo apt-get install clickhouse-client clickhouse-server
```

You can also download and install packages manually from here:

<https://repo.yandex.ru/clickhouse/deb/stable/main/>.

From RPM Packages

Yandex does not run ClickHouse on `rpm` based Linux distributions and `rpm` packages are not as thoroughly tested. So use them at your own risk, but there are many other companies that do successfully run them in production without any major issues.

For CentOS, RHEL or Fedora there are the following options:

- Packages from https://repo.yandex.ru/clickhouse/rpm/stable/x86_64/ are generated from official `deb` packages by Yandex and have byte-identical binaries.
- Packages from <https://github.com/Altinity/clickhouse-rpm-install> are built by independent company Altinity, but are used widely without any complaints.
- Or you can use Docker (see below).

From Docker Image

To run ClickHouse inside Docker follow the guide on [Docker Hub](#). Those images use official `deb` packages inside.

From Sources

To manually compile ClickHouse, follow the instructions for [Linux](#) or [Mac OS X](#).

You can compile packages and install them or use programs without installing packages. Also by building manually you can disable SSE 4.2 requirement or build for AArch64 CPUs.

```
Client: dbms/programs/clickhouse-client
Server: dbms/programs/clickhouse-server
```

You'll need to create a data and metadata folders and `chown` them for the desired user. Their paths can be changed in server config (`src/dbms/programs/server/config.xml`), by default they are:

```
/opt/clickhouse/data/default/
/opt/clickhouse/metadata/default/
```

On Gentoo you can just use `emerge clickhouse` to install ClickHouse from sources.

Launch

To start the server as a daemon, run:

```
$ sudo service clickhouse-server start
```

See the logs in the `/var/log/clickhouse-server/` directory.

If the server doesn't start, check the configurations in the file `/etc/clickhouse-server/config.xml`.

You can also manually launch the server from the console:

```
$ clickhouse-server --config-file=/etc/clickhouse-server/config.xml
```

In this case, the log will be printed to the console, which is convenient during development.

If the configuration file is in the current directory, you don't need to specify the `--config-file` parameter. By default, it uses `./config.xml`.

ClickHouse supports access restriction settings. They are located in the `users.xml` file (next to `config.xml`).

By default, access is allowed from anywhere for the `default` user, without a password. See `user/default/networks`. For more information, see the section ["Configuration Files"](#).

After launching server, you can use the command-line client to connect to it:

```
$ clickhouse-client
```

By default it connects to `localhost:9000` on behalf of the user `default` without a password. It can also be used to connect to a remote server using `--host` argument.

The terminal must use UTF-8 encoding.

For more information, see the section ["Command-line client"](#).

Example:

```
$ ./clickhouse-client
ClickHouse client version 0.0.18749.
Connecting to localhost:9000.
Connected to ClickHouse server version 0.0.18749.

:) SELECT 1

SELECT 1

┌1┐
│1│
└─┘

1 rows in set. Elapsed: 0.003 sec.

:)
```

Congratulations, the system works!

To continue experimenting, you can download one of test data sets or go through [tutorial](#).

OnTime

This dataset can be obtained in two ways:

```
1. import from raw data
```

- import from raw data
- download of prepared partitions

Import From Raw Data

Downloading data:

```
for s in `seq 1987 2018`  
do  
for m in `seq 1 12`  
do  
wget https://transtats.bts.gov/PREZIP/On_Time_Reporting_Carrier_On_Time_Performance_1987_present_${s}_${m}.zip  
done  
done
```

(from <https://github.com/Percona-Lab/ontime-airline-performance/blob/master/download.sh>)

Creating a table:

```
CREATE TABLE `ontime` (  
  `Year` UInt16,  
  `Quarter` UInt8,  
  `Month` UInt8,  
  `DayOfMonth` UInt8,  
  `DayOfWeek` UInt8,  
  `FlightDate` Date,  
  `UniqueCarrier` FixedString(7),  
  `AirlineID` Int32,  
  `Carrier` FixedString(2),  
  `TailNum` String,  
  `FlightNum` String,  
  `OriginAirportID` Int32,  
  `OriginAirportSeqID` Int32,  
  `OriginCityMarketID` Int32,  
  `Origin` FixedString(5),  
  `OriginCityName` String,  
  `OriginState` FixedString(2),  
  `OriginStateFips` String,  
  `OriginStateName` String,  
  `OriginWac` Int32,  
  `DestAirportID` Int32,  
  `DestAirportSeqID` Int32,  
  `DestCityMarketID` Int32,  
  `Dest` FixedString(5),  
  `DestCityName` String,  
  `DestState` FixedString(2),  
  `DestStateFips` String,  
  `DestStateName` String,  
  `DestWac` Int32,  
  `CRSDepTime` Int32,  
  `DepTime` Int32,  
  `DepDelay` Int32,  
  `DepDelayMinutes` Int32,  
  `DepDel15` Int32,  
  `DepartureDelayGroups` String,  
  `DepTimeBlk` String,  
  `TaxiOut` Int32,  
  `WheelsOff` Int32,  
  `WheelsOn` Int32,  
  `TaxiIn` Int32,  
  `CRSArrTime` Int32,  
  `ArrTime` Int32,  
  `ArrDelay` Int32,
```

`ArrDelayMinutes` Int32,
`ArrDel15` Int32,
`ArrivalDelayGroups` Int32,
`ArrTimeBlk` String,
`Cancelled` UInt8,
`CancellationCode` FixedString(1),
`Diverted` UInt8,
`CRSElapsedTime` Int32,
`ActualElapsedTime` Int32,
`AirTime` Int32,
`Flights` Int32,
`Distance` Int32,
`DistanceGroup` UInt8,
`CarrierDelay` Int32,
`WeatherDelay` Int32,
`NASDelay` Int32,
`SecurityDelay` Int32,
`LateAircraftDelay` Int32,
`FirstDepTime` String,
`TotalAddGTime` String,
`LongestAddGTime` String,
`DivAirportLandings` String,
`DivReachedDest` String,
`DivActualElapsedTime` String,
`DivArrDelay` String,
`DivDistance` String,
`Div1Airport` String,
`Div1AirportID` Int32,
`Div1AirportSeqID` Int32,
`Div1WheelsOn` String,
`Div1TotalGTime` String,
`Div1LongestGTime` String,
`Div1WheelsOff` String,
`Div1TailNum` String,
`Div2Airport` String,
`Div2AirportID` Int32,
`Div2AirportSeqID` Int32,
`Div2WheelsOn` String,
`Div2TotalGTime` String,
`Div2LongestGTime` String,
`Div2WheelsOff` String,
`Div2TailNum` String,
`Div3Airport` String,
`Div3AirportID` Int32,
`Div3AirportSeqID` Int32,
`Div3WheelsOn` String,
`Div3TotalGTime` String,
`Div3LongestGTime` String,
`Div3WheelsOff` String,
`Div3TailNum` String,
`Div4Airport` String,
`Div4AirportID` Int32,
`Div4AirportSeqID` Int32,
`Div4WheelsOn` String,
`Div4TotalGTime` String,
`Div4LongestGTime` String,
`Div4WheelsOff` String,
`Div4TailNum` String,
`Div5Airport` String,
`Div5AirportID` Int32,
`Div5AirportSeqID` Int32,
`Div5WheelsOn` String,
`Div5TotalGTime` String,
`Div5LongestGTime` String,

```
`Div5WheelsOff` String,  
`Div5TailNum` String  
) ENGINE = MergeTree(FlightDate, (Year, FlightDate), 8192)
```

Loading data:

```
for i in *.zip; do echo $i; unzip -cq $i '*.csv' | sed 's/\./00/g' | clickhouse-client --host=example-perftest01j --query="INSERT  
INTO ontime FORMAT CSVWithNames"; done
```

Download of Prepared Partitions

```
curl -O https://clickhouse-datasets.s3.yandex.net/ontime/partitions/ontime.tar  
tar xvf onttime.tar -C /var/lib/clickhouse # path to ClickHouse data directory  
## check permissions of unpacked data, fix if required  
sudo service clickhouse-server restart  
clickhouse-client --query "select count(*) from datasets.ontime"
```

Info

If you will run queries described below, you have to use full table name,
`datasets.ontime`.

Queries

Q0.

```
select avg(c1) from (select Year, Month, count(*) as c1 from ontime group by Year, Month);
```

Q1. The number of flights per day from the year 2000 to 2008

```
SELECT DayOfWeek, count(*) AS c FROM ontime WHERE Year >= 2000 AND Year <= 2008 GROUP BY DayOfWeek ORDER BY  
c DESC;
```

Q2. The number of flights delayed by more than 10 minutes, grouped by the day of the week, for 2000-2008

```
SELECT DayOfWeek, count(*) AS c FROM ontime WHERE DepDelay>10 AND Year >= 2000 AND Year <= 2008 GROUP BY  
DayOfWeek ORDER BY c DESC
```

Q3. The number of delays by airport for 2000-2008

```
SELECT Origin, count(*) AS c FROM ontime WHERE DepDelay>10 AND Year >= 2000 AND Year <= 2008 GROUP BY Origin  
ORDER BY c DESC LIMIT 10
```

Q4. The number of delays by carrier for 2007

```
SELECT Carrier, count(*) FROM ontime WHERE DepDelay>10 AND Year = 2007 GROUP BY Carrier ORDER BY count(*) DESC
```

Q5. The percentage of delays by carrier for 2007

```
SELECT Carrier, c, c2, c*100/c2 as c3  
FROM  
(  
  SELECT  
    Carrier,
```

```

        count(*) AS c
    FROM ontime
    WHERE DepDelay>10
        AND Year=2007
    GROUP BY Carrier
)
ANY INNER JOIN
(
    SELECT
        Carrier,
        count(*) AS c2
    FROM ontime
    WHERE Year=2007
    GROUP BY Carrier
) USING Carrier
ORDER BY c3 DESC;

```

Better version of the same query:

```

SELECT Carrier, avg(DepDelay > 10) * 100 AS c3 FROM ontime WHERE Year = 2007 GROUP BY Carrier ORDER BY Carrier

```

Q6. The previous request for a broader range of years, 2000-2008

```

SELECT Carrier, c, c2, c*100/c2 as c3
FROM
(
    SELECT
        Carrier,
        count(*) AS c
    FROM ontime
    WHERE DepDelay>10
        AND Year >= 2000 AND Year <= 2008
    GROUP BY Carrier
)
ANY INNER JOIN
(
    SELECT
        Carrier,
        count(*) AS c2
    FROM ontime
    WHERE Year >= 2000 AND Year <= 2008
    GROUP BY Carrier
) USING Carrier
ORDER BY c3 DESC;

```

Better version of the same query:

```

SELECT Carrier, avg(DepDelay > 10) * 100 AS c3 FROM ontime WHERE Year >= 2000 AND Year <= 2008 GROUP BY Carrier
ORDER BY Carrier

```

Q7. Percentage of flights delayed for more than 10 minutes, by year

```

SELECT Year, c1/c2
FROM
(
    select
        Year,
        count(*)*100 as c1
    from ontime

```

```

WHERE DepDelay>10
GROUP BY Year
)
ANY INNER JOIN
(
select
Year,
count(*) as c2
from ontime
GROUP BY Year
) USING (Year)
ORDER BY Year

```

Better version of the same query:

```

SELECT Year, avg(DepDelay > 10) FROM ontime GROUP BY Year ORDER BY Year

```

Q8. The most popular destinations by the number of directly connected cities for various year ranges

```

SELECT DestCityName, uniqExact(OriginCityName) AS u FROM ontime WHERE Year >= 2000 and Year <= 2010 GROUP BY
DestCityName ORDER BY u DESC LIMIT 10;

```

Q9.

```

select Year, count(*) as c1 from ontime group by Year;

```

Q10.

```

select
min(Year), max(Year), Carrier, count(*) as cnt,
sum(ArrDelayMinutes>30) as flights_delayed,
round(sum(ArrDelayMinutes>30)/count(*),2) as rate
FROM ontime
WHERE
DayOfWeek not in (6,7) and OriginState not in ('AK', 'HI', 'PR', 'VI')
and DestState not in ('AK', 'HI', 'PR', 'VI')
and FlightDate < '2010-01-01'
GROUP by Carrier
HAVING cnt > 100000 and max(Year) > 1990
ORDER by rate DESC
LIMIT 1000;

```

Bonus:

```

SELECT avg(cnt) FROM (SELECT Year,Month,count(*) AS cnt FROM ontime WHERE DepDel15=1 GROUP BY Year,Month)

select avg(c1) from (select Year,Month,count(*) as c1 from ontime group by Year,Month)

SELECT DestCityName, uniqExact(OriginCityName) AS u FROM ontime GROUP BY DestCityName ORDER BY u DESC LIMIT 10;

SELECT OriginCityName, DestCityName, count() AS c FROM ontime GROUP BY OriginCityName, DestCityName ORDER BY c
DESC LIMIT 10;

SELECT OriginCityName, count() AS c FROM ontime GROUP BY OriginCityName ORDER BY c DESC LIMIT 10;

```

This performance test was created by Vadim Tkachenko. See:

• <https://www.nersone.com/blog/2008/10/02/analyzing-air-traffic-performance-with-infobright-and-monthdb/>

- <https://www.percona.com/blog/2009/10/02/analyzing-air-traffic-performance-with-mongodb-and-monetdb/>
- <https://www.percona.com/blog/2009/10/26/air-traffic-queries-in-luciddb/>
- <https://www.percona.com/blog/2009/11/02/air-traffic-queries-in-infinidb-early-alpha/>
- <https://www.percona.com/blog/2014/04/21/using-apache-hadoop-and-impala-together-with-mysql-for-data-analysis/>
- <https://www.percona.com/blog/2016/01/07/apache-spark-with-air-ontime-performance-data/>
- <http://nickmakos.blogspot.ru/2012/08/analyzing-air-traffic-performance-with.html>

New York Taxi Data

This dataset can be obtained in two ways:

- import from raw data
- download of prepared partitions

How to Import The Raw Data

See <https://github.com/toddwschneider/nyc-taxi-data> and <http://tech.marksblogg.com/billion-nyc-taxi-rides-redshift.html> for the description of a dataset and instructions for downloading.

Downloading will result in about 227 GB of uncompressed data in CSV files. The download takes about an hour over a 1 Gbit connection (parallel downloading from s3.amazonaws.com recovers at least half of a 1 Gbit channel).

Some of the files might not download fully. Check the file sizes and re-download any that seem doubtful.

Some of the files might contain invalid rows. You can fix them as follows:

```
sed -E '!(.*){18,}/d' data/yellow_tripdata_2010-02.csv > data/yellow_tripdata_2010-02.csv_
sed -E '!(.*){18,}/d' data/yellow_tripdata_2010-03.csv > data/yellow_tripdata_2010-03.csv_
mv data/yellow_tripdata_2010-02.csv_ data/yellow_tripdata_2010-02.csv
mv data/yellow_tripdata_2010-03.csv_ data/yellow_tripdata_2010-03.csv
```

Then the data must be pre-processed in PostgreSQL. This will create selections of points in the polygons (to match points on the map with the boroughs of New York City) and combine all the data into a single denormalized flat table by using a JOIN. To do this, you will need to install PostgreSQL with PostGIS support.

Be careful when running `initialize_database.sh` and manually re-check that all the tables were created correctly.

It takes about 20-30 minutes to process each month's worth of data in PostgreSQL, for a total of about 48 hours.

You can check the number of downloaded rows as follows:

```
time psql nyc-taxi-data -c "SELECT count(*) FROM trips;"
### Count
1298979494
(1 row)

real    7m9.164s
```

(This is slightly more than 1.1 billion rows reported by Mark Litwintschik in a series of blog posts.)

The data in PostgreSQL uses 370 GB of space.

Exporting the data from PostgreSQL:

```
COPY
(
  SELECT trips.id,
         trips.vendor_id,
         trips.pickup_datetime,
```


reads from the disk at a speed of about 28 MB per second.
This takes about 5 hours. The resulting TSV file is 590612904969 bytes.

Create a temporary table in ClickHouse:

```
CREATE TABLE trips
(
trip_id            UInt32,
vendor_id          String,
pickup_datetime    DateTime,
dropoff_datetime   Nullable(DateTime),
store_and_fwd_flag Nullable(FixedString(1)),
rate_code_id       Nullable(UInt8),
pickup_longitude   Nullable(Float64),
pickup_latitude    Nullable(Float64),
dropoff_longitude   Nullable(Float64),
dropoff_latitude   Nullable(Float64),
passenger_count    Nullable(UInt8),
trip_distance      Nullable(Float64),
fare_amount        Nullable(Float32),
extra              Nullable(Float32),
mta_tax            Nullable(Float32),
tip_amount         Nullable(Float32),
tolls_amount       Nullable(Float32),
ehail_fee          Nullable(Float32),
improvement_surcharge Nullable(Float32),
total_amount       Nullable(Float32),
payment_type       Nullable(String),
trip_type          Nullable(UInt8),
pickup             Nullable(String),
dropoff            Nullable(String),
cab_type           Nullable(String),
precipitation       Nullable(UInt8),
snow_depth         Nullable(UInt8),
snowfall           Nullable(UInt8),
max_temperature    Nullable(UInt8),
min_temperature    Nullable(UInt8),
average_wind_speed Nullable(UInt8),
pickup_nyct2010_gid Nullable(UInt8),
pickup_ctlabel     Nullable(String),
pickup_borocode    Nullable(UInt8),
pickup_boroname    Nullable(String),
pickup_ct2010      Nullable(String),
pickup_boroc2010   Nullable(String),
pickup_cdeligibil  Nullable(FixedString(1)),
pickup_ntacode     Nullable(String),
pickup_ntaname     Nullable(String),
pickup_puma        Nullable(String),
dropoff_nyct2010_gid Nullable(UInt8),
dropoff_ctlabel    Nullable(String),
dropoff_borocode   Nullable(UInt8),
dropoff_boroname   Nullable(String),
dropoff_ct2010     Nullable(String),
dropoff_boroc2010  Nullable(String),
dropoff_cdeligibil Nullable(String),
dropoff_ntacode    Nullable(String),
dropoff_ntaname    Nullable(String),
dropoff_puma       Nullable(String)
) ENGINE = Log;
```

It is needed for converting fields to more correct data types and, if possible, to eliminate NULLs.

```
time clickhouse-client --query="INSERT INTO trips FORMAT TabSeparated" < trips.tsv
```

```
real 75m56.214s
```

Data is read at a speed of 112-140 Mb/second.

Loading data into a Log type table in one stream took 76 minutes.

The data in this table uses 142 GB.

(Importing data directly from Postgres is also possible using `COPY ... TO PROGRAM`.)

Unfortunately, all the fields associated with the weather (precipitation...average_wind_speed) were filled with NULL. Because of this, we will remove them from the final data set.

To start, we'll create a table on a single server. Later we will make the table distributed.

Create and populate a summary table:

```
CREATE TABLE trips_mergetree
ENGINE = MergeTree(pickup_date, pickup_datetime, 8192)
AS SELECT

trip_id,
CAST(vendor_id AS Enum8('1' = 1, '2' = 2, 'CMT' = 3, 'VTS' = 4, 'DDS' = 5, 'B02512' = 10, 'B02598' = 11, 'B02617' = 12,
'B02682' = 13, 'B02764' = 14)) AS vendor_id,
toDate(pickup_datetime) AS pickup_date,
ifNull(pickup_datetime, toDateTime(0)) AS pickup_datetime,
toDate(dropoff_datetime) AS dropoff_date,
ifNull(dropoff_datetime, toDateTime(0)) AS dropoff_datetime,
assumeNotNull(store_and_fwd_flag) IN ('Y', '1', '2') AS store_and_fwd_flag,
assumeNotNull(rate_code_id) AS rate_code_id,
assumeNotNull(pickup_longitude) AS pickup_longitude,
assumeNotNull(pickup_latitude) AS pickup_latitude,
assumeNotNull(dropoff_longitude) AS dropoff_longitude,
assumeNotNull(dropoff_latitude) AS dropoff_latitude,
assumeNotNull(passenger_count) AS passenger_count,
assumeNotNull(trip_distance) AS trip_distance,
assumeNotNull(fare_amount) AS fare_amount,
assumeNotNull(extra) AS extra,
assumeNotNull(mta_tax) AS mta_tax,
assumeNotNull(tip_amount) AS tip_amount,
assumeNotNull(tolls_amount) AS tolls_amount,
assumeNotNull(ehail_fee) AS ehail_fee,
assumeNotNull(improvement_surcharge) AS improvement_surcharge,
assumeNotNull(total_amount) AS total_amount,
CAST((assumeNotNull(payment_type) AS pt) IN ('CSH', 'CASH', 'Cash', 'CAS', 'Cas', '1') ? 'CSH' : (pt IN ('CRD', 'Credit', 'Cre',
'CRE', 'CREDIT', '2') ? 'CRE' : (pt IN ('NOC', 'No Charge', 'No', '3') ? 'NOC' : (pt IN ('DIS', 'Dispute', 'Dis', '4') ? 'DIS' : 'UNK')))) AS
Enum8('CSH' = 1, 'CRE' = 2, 'UNK' = 0, 'NOC' = 3, 'DIS' = 4)) AS payment_type_,
assumeNotNull(trip_type) AS trip_type,
ifNull(toFixedString(unhex(pickup), 25), toFixedString('', 25)) AS pickup,
ifNull(toFixedString(unhex(dropoff), 25), toFixedString('', 25)) AS dropoff,
CAST(assumeNotNull(cab_type) AS Enum8('yellow' = 1, 'green' = 2, 'uber' = 3)) AS cab_type,

assumeNotNull(pickup_nyct2010_gid) AS pickup_nyct2010_gid,
toFloat32(ifNull(pickup_ctlabel, '0')) AS pickup_ctlabel,
assumeNotNull(pickup_borocode) AS pickup_borocode,
CAST(assumeNotNull(pickup_boroname) AS Enum8('Manhattan' = 1, 'Queens' = 4, 'Brooklyn' = 3, '' = 0, 'Bronx' = 2, 'Staten
Island' = 5)) AS pickup_boroname,
toFixedString(ifNull(pickup_ct2010, '000000'), 6) AS pickup_ct2010,
toFixedString(ifNull(pickup_boroct2010, '0000000'), 7) AS pickup_boroct2010,
CAST(assumeNotNull(ifNull(pickup_cdeligibil, '')) AS Enum8(' ' = 0, 'E' = 1, 'I' = 2)) AS pickup_cdeligibil,
toFixedString(ifNull(pickup_ntacode, '0000'), 4) AS pickup_ntacode,

CAST(assumeNotNull(pickup_ntaname) AS Enum16(' ' = 0, 'Airport' = 1, 'Allerton-Pelham Gardens' = 2, 'Annadale-Huguenot-
```

Prince's Bay-Eltingville' = 3, 'Arden Heights' = 4, 'Astoria' = 5, 'Auburndale' = 6, 'Baisley Park' = 7, 'Bath Beach' = 8, 'Battery Park City-Lower Manhattan' = 9, 'Bay Ridge' = 10, 'Bayside-Bayside Hills' = 11, 'Bedford' = 12, 'Bedford Park-Fordham North' = 13, 'Bellerose' = 14, 'Belmont' = 15, 'Bensonhurst East' = 16, 'Bensonhurst West' = 17, 'Borough Park' = 18, 'Breezy Point-Belle Harbor-Rockaway Park-Broad Channel' = 19, 'Briarwood-Jamaica Hills' = 20, 'Brighton Beach' = 21, 'Bronxdale' = 22, 'Brooklyn Heights-Cobble Hill' = 23, 'Brownsville' = 24, 'Bushwick North' = 25, 'Bushwick South' = 26, 'Cambria Heights' = 27, 'Canarsie' = 28, 'Carroll Gardens-Columbia Street-Red Hook' = 29, 'Central Harlem North-Polo Grounds' = 30, 'Central Harlem South' = 31, 'Charleston-Richmond Valley-Tottenville' = 32, 'Chinatown' = 33, 'Claremont-Bathgate' = 34, 'Clinton' = 35, 'Clinton Hill' = 36, 'Co-op City' = 37, 'College Point' = 38, 'Corona' = 39, 'Crotona Park East' = 40, 'Crown Heights North' = 41, 'Crown Heights South' = 42, 'Cypress Hills-City Line' = 43, 'DUMBO-Vinegar Hill-Downtown Brooklyn-Boerum Hill' = 44, 'Douglas Manor-Douglaston-Little Neck' = 45, 'Dyker Heights' = 46, 'East Concourse-Concourse Village' = 47, 'East Elmhurst' = 48, 'East Flatbush-Farragut' = 49, 'East Flushing' = 50, 'East Harlem North' = 51, 'East Harlem South' = 52, 'East New York' = 53, 'East New York (Pennsylvania Ave)' = 54, 'East Tremont' = 55, 'East Village' = 56, 'East Williamsburg' = 57, 'Eastchester-Edenwald-Baychester' = 58, 'Elmhurst' = 59, 'Elmhurst-Maspeth' = 60, 'Erasmus' = 61, 'Far Rockaway-Bayswater' = 62, 'Flatbush' = 63, 'Flatlands' = 64, 'Flushing' = 65, 'Fordham South' = 66, 'Forest Hills' = 67, 'Fort Greene' = 68, 'Fresh Meadows-Utopia' = 69, 'Ft. Totten-Bay Terrace-Clearview' = 70, 'Georgetown-Marine Park-Bergen Beach-Mill Basin' = 71, 'Glen Oaks-Floral Park-New Hyde Park' = 72, 'Glendale' = 73, 'Gramercy' = 74, 'Grasmere-Arrochar-Ft. Wadsworth' = 75, 'Gravesend' = 76, 'Great Kills' = 77, 'Greenpoint' = 78, 'Grymes Hill-Clifton-Fox Hills' = 79, 'Hamilton Heights' = 80, 'Hammels-Arverne-Edgemere' = 81, 'Highbridge' = 82, 'Holлис' = 83, 'Homecrest' = 84, 'Hudson Yards-Chelsea-Flatiron-Union Square' = 85, 'Hunters Point-Sunnyside-West Maspeth' = 86, 'Hunts Point' = 87, 'Jackson Heights' = 88, 'Jamaica' = 89, 'Jamaica Estates-Holliswood' = 90, 'Kensington-Ocean Parkway' = 91, 'Kew Gardens' = 92, 'Kew Gardens Hills' = 93, 'Kingsbridge Heights' = 94, 'Laurelton' = 95, 'Lenox Hill-Roosevelt Island' = 96, 'Lincoln Square' = 97, 'Lindenwood-Howard Beach' = 98, 'Longwood' = 99, 'Lower East Side' = 100, 'Madison' = 101, 'Manhattanville' = 102, 'Marble Hill-Inwood' = 103, 'Mariner's Harbor-Arlington-Port Ivory-Graniteville' = 104, 'Maspeth' = 105, 'Melrose South-Mott Haven North' = 106, 'Middle Village' = 107, 'Midtown-Midtown South' = 108, 'Midwood' = 109, 'Morningside Heights' = 110, 'Morrisania-Melrose' = 111, 'Mott Haven-Port Morris' = 112, 'Mount Hope' = 113, 'Murray Hill' = 114, 'Murray Hill-Kips Bay' = 115, 'New Brighton-Silver Lake' = 116, 'New Dorp-Midland Beach' = 117, 'New Springville-Bloomfield-Travis' = 118, 'North Corona' = 119, 'North Riverdale-Fieldston-Riverdale' = 120, 'North Side-South Side' = 121, 'Norwood' = 122, 'Oakland Gardens' = 123, 'Oakwood-Oakwood Beach' = 124, 'Ocean Hill' = 125, 'Ocean Parkway South' = 126, 'Old Astoria' = 127, 'Old Town-Dongan Hills-South Beach' = 128, 'Ozone Park' = 129, 'Park Slope-Gowanus' = 130, 'Parkchester' = 131, 'Pelham Bay-Country Club-City Island' = 132, 'Pelham Parkway' = 133, 'Pomonok-Flushing Heights-Hillcrest' = 134, 'Port Richmond' = 135, 'Prospect Heights' = 136, 'Prospect Lefferts Gardens-Wingate' = 137, 'Queens Village' = 138, 'Queensboro Hill' = 139, 'Queensbridge-Ravenswood-Long Island City' = 140, 'Rego Park' = 141, 'Richmond Hill' = 142, 'Ridgewood' = 143, 'Rikers Island' = 144, 'Rosedale' = 145, 'Rossville-Woodrow' = 146, 'Rugby-Remsen Village' = 147, 'Schuylerville-Throgs Neck-Edgewater Park' = 148, 'Seagate-Coney Island' = 149, 'Sheepshead Bay-Gerritsen Beach-Manhattan Beach' = 150, 'SoHo-TriBeCa-Civic Center-Little Italy' = 151, 'Soundview-Bruckner' = 152, 'Soundview-Castle Hill-Clason Point-Harding Park' = 153, 'South Jamaica' = 154, 'South Ozone Park' = 155, 'Springfield Gardens North' = 156, 'Springfield Gardens South-Brookville' = 157, 'Spuyten Duyvil-Kingsbridge' = 158, 'St. Albans' = 159, 'Stapleton-Rosebank' = 160, 'Starrett City' = 161, 'Steinway' = 162, 'Stuyvesant Heights' = 163, 'Stuyvesant Town-Cooper Village' = 164, 'Sunset Park East' = 165, 'Sunset Park West' = 166, 'Todt Hill-Emerson Hill-Heartland Village-Lighthouse Hill' = 167, 'Turtle Bay-East Midtown' = 168, 'University Heights-Morris Heights' = 169, 'Upper East Side-Carnegie Hill' = 170, 'Upper West Side' = 171, 'Van Cortlandt Village' = 172, 'Van Nest-Morris Park-Westchester Square' = 173, 'Washington Heights North' = 174, 'Washington Heights South' = 175, 'West Brighton' = 176, 'West Concourse' = 177, 'West Farms-Bronx River' = 178, 'West New Brighton-New Brighton-St. George' = 179, 'West Village' = 180, 'Westchester-Unionport' = 181, 'Westerleigh' = 182, 'Whitestone' = 183, 'Williamsbridge-Olinville' = 184, 'Williamsburg' = 185, 'Windsor Terrace' = 186, 'Woodhaven' = 187, 'Woodlawn-Wakefield' = 188, 'Woodside' = 189, 'Yorkville' = 190, 'park-cemetery-etc-Bronx' = 191, 'park-cemetery-etc-Brooklyn' = 192, 'park-cemetery-etc-Manhattan' = 193, 'park-cemetery-etc-Queens' = 194, 'park-cemetery-etc-Staten Island' = 195)) AS pickup_ntaname,

toUInt16(ifNull(pickup_puma, '0')) AS pickup_puma,

assumeNotNull(dropoff_nyct2010_gid) AS dropoff_nyct2010_gid,

toFloat32(ifNull(dropoff_ctlabel, '0')) AS dropoff_ctlabel,

assumeNotNull(dropoff_borocode) AS dropoff_borocode,

CAST(assumeNotNull(dropoff_borocode) AS Enum8('Manhattan' = 1, 'Queens' = 4, 'Brooklyn' = 3, '' = 0, 'Bronx' = 2, 'Staten Island' = 5)) AS dropoff_borocode,

toFixedString(ifNull(dropoff_ct2010, '000000'), 6) AS dropoff_ct2010,

toFixedString(ifNull(dropoff_boroc2010, '0000000'), 7) AS dropoff_boroc2010,

CAST(assumeNotNull(ifNull(dropoff_cdeligibil, '')) AS Enum8('' = 0, 'E' = 1, 'I' = 2)) AS dropoff_cdeligibil,

toFixedString(ifNull(dropoff_ntacode, '0000'), 4) AS dropoff_ntacode,

CAST(assumeNotNull(dropoff_ntaname) AS Enum16('' = 0, 'Airport' = 1, 'Allerton-Pelham Gardens' = 2, 'Annadale-Huguenot-Prince's Bay-Eltingville' = 3, 'Arden Heights' = 4, 'Astoria' = 5, 'Auburndale' = 6, 'Baisley Park' = 7, 'Bath Beach' = 8, 'Battery Park City-Lower Manhattan' = 9, 'Bay Ridge' = 10, 'Bayside-Bayside Hills' = 11, 'Bedford' = 12, 'Bedford Park-Fordham North' = 13, 'Bellerose' = 14, 'Belmont' = 15, 'Bensonhurst East' = 16, 'Bensonhurst West' = 17, 'Borough

Park' = 18, 'Breezy Point-Belle Harbor-Rockaway Park-Broad Channel' = 19, 'Briarwood-Jamaica Hills' = 20, 'Brighton Beach' = 21, 'Bronxdale' = 22, 'Brooklyn Heights-Cobble Hill' = 23, 'Brownsville' = 24, 'Bushwick North' = 25, 'Bushwick South' = 26, 'Cambria Heights' = 27, 'Canarsie' = 28, 'Carroll Gardens-Columbia Street-Red Hook' = 29, 'Central Harlem North-Polo Grounds' = 30, 'Central Harlem South' = 31, 'Charleston-Richmond Valley-Tottenville' = 32, 'Chinatown' = 33, 'Claremont-Bathgate' = 34, 'Clinton' = 35, 'Clinton Hill' = 36, 'Co-op City' = 37, 'College Point' = 38, 'Corona' = 39, 'Crotona Park East' = 40, 'Crown Heights North' = 41, 'Crown Heights South' = 42, 'Cypress Hills-City Line' = 43, 'DUMBO-Vinegar Hill-Downtown Brooklyn-Boerum Hill' = 44, 'Douglas Manor-Douglaston-Little Neck' = 45, 'Dyker Heights' = 46, 'East Concourse-Concourse Village' = 47, 'East Elmhurst' = 48, 'East Flatbush-Farragut' = 49, 'East Flushing' = 50, 'East Harlem North' = 51, 'East Harlem South' = 52, 'East New York' = 53, 'East New York (Pennsylvania Ave)' = 54, 'East Tremont' = 55, 'East Village' = 56, 'East Williamsburg' = 57, 'Eastchester-Edenwald-Baychester' = 58, 'Elmhurst' = 59, 'Elmhurst-Maspeth' = 60, 'Erasmus' = 61, 'Far Rockaway-Bayswater' = 62, 'Flatbush' = 63, 'Flatlands' = 64, 'Flushing' = 65, 'Fordham South' = 66, 'Forest Hills' = 67, 'Fort Greene' = 68, 'Fresh Meadows-Utopia' = 69, 'Ft. Totten-Bay Terrace-Clearview' = 70, 'Georgetown-Marine Park-Bergen Beach-Mill Basin' = 71, 'Glen Oaks-Floral Park-New Hyde Park' = 72, 'Glendale' = 73, 'Gramercy' = 74, 'Grasmere-Arrochar-Ft. Wadsworth' = 75, 'Gravesend' = 76, 'Great Kills' = 77, 'Greenpoint' = 78, 'Grymes Hill-Clifton-Fox Hills' = 79, 'Hamilton Heights' = 80, 'Hammels-Arverne-Edgemere' = 81, 'Highbridge' = 82, 'Hollis' = 83, 'Homecrest' = 84, 'Hudson Yards-Chelsea-Flatiron-Union Square' = 85, 'Hunters Point-Sunnyside-West Maspeth' = 86, 'Hunts Point' = 87, 'Jackson Heights' = 88, 'Jamaica' = 89, 'Jamaica Estates-Holliswood' = 90, 'Kensington-Ocean Parkway' = 91, 'Kew Gardens' = 92, 'Kew Gardens Hills' = 93, 'Kingsbridge Heights' = 94, 'Laurelton' = 95, 'Lenox Hill-Roosevelt Island' = 96, 'Lincoln Square' = 97, 'Lindenwood-Howard Beach' = 98, 'Longwood' = 99, 'Lower East Side' = 100, 'Madison' = 101, 'Manhattanville' = 102, 'Marble Hill-Inwood' = 103, 'Mariner's Harbor-Arlington-Port Ivory-Graniteville' = 104, 'Maspeth' = 105, 'Melrose South-Mott Haven North' = 106, 'Middle Village' = 107, 'Midtown-Midtown South' = 108, 'Midwood' = 109, 'Morningside Heights' = 110, 'Morrisania-Melrose' = 111, 'Mott Haven-Port Morris' = 112, 'Mount Hope' = 113, 'Murray Hill' = 114, 'Murray Hill-Kips Bay' = 115, 'New Brighton-Silver Lake' = 116, 'New Dorp-Midland Beach' = 117, 'New Springville-Bloomfield-Travis' = 118, 'North Corona' = 119, 'North Riverdale-Fieldston-Riverdale' = 120, 'North Side-South Side' = 121, 'Norwood' = 122, 'Oakland Gardens' = 123, 'Oakwood-Oakwood Beach' = 124, 'Ocean Hill' = 125, 'Ocean Parkway South' = 126, 'Old Astoria' = 127, 'Old Town-Dongan Hills-South Beach' = 128, 'Ozone Park' = 129, 'Park Slope-Gowanus' = 130, 'Parkchester' = 131, 'Pelham Bay-Country Club-City Island' = 132, 'Pelham Parkway' = 133, 'Pomonok-Flushing Heights-Hillcrest' = 134, 'Port Richmond' = 135, 'Prospect Heights' = 136, 'Prospect Lefferts Gardens-Wingate' = 137, 'Queens Village' = 138, 'Queensboro Hill' = 139, 'Queensbridge-Ravenswood-Long Island City' = 140, 'Rego Park' = 141, 'Richmond Hill' = 142, 'Ridgewood' = 143, 'Rikers Island' = 144, 'Rosedale' = 145, 'Rossville-Woodrow' = 146, 'Rugby-Remsen Village' = 147, 'Schuylerville-Throgs Neck-Edgewater Park' = 148, 'Seagate-Coney Island' = 149, 'Sheepshead Bay-Gerritsen Beach-Manhattan Beach' = 150, 'SoHo-TriBeCa-Civic Center-Little Italy' = 151, 'Soundview-Bruckner' = 152, 'Soundview-Castle Hill-Clason Point-Harding Park' = 153, 'South Jamaica' = 154, 'South Ozone Park' = 155, 'Springfield Gardens North' = 156, 'Springfield Gardens South-Brookville' = 157, 'Spuyten Duyvil-Kingsbridge' = 158, 'St. Albans' = 159, 'Stapleton-Rosebank' = 160, 'Starrett City' = 161, 'Steinway' = 162, 'Stuyvesant Heights' = 163, 'Stuyvesant Town-Cooper Village' = 164, 'Sunset Park East' = 165, 'Sunset Park West' = 166, 'Todt Hill-Emerson Hill-Heartland Village-Lighthouse Hill' = 167, 'Turtle Bay-East Midtown' = 168, 'University Heights-Morris Heights' = 169, 'Upper East Side-Carnegie Hill' = 170, 'Upper West Side' = 171, 'Van Cortlandt Village' = 172, 'Van Nest-Morris Park-Westchester Square' = 173, 'Washington Heights North' = 174, 'Washington Heights South' = 175, 'West Brighton' = 176, 'West Concourse' = 177, 'West Farms-Bronx River' = 178, 'West New Brighton-New Brighton-St. George' = 179, 'West Village' = 180, 'Westchester-Unionport' = 181, 'Westerleigh' = 182, 'Whitestone' = 183, 'Williamsbridge-Olinville' = 184, 'Williamsburg' = 185, 'Windsor Terrace' = 186, 'Woodhaven' = 187, 'Woodlawn-Wakefield' = 188, 'Woodside' = 189, 'Yorkville' = 190, 'park-cemetery-etc-Bronx' = 191, 'park-cemetery-etc-Brooklyn' = 192, 'park-cemetery-etc-Manhattan' = 193, 'park-cemetery-etc-Queens' = 194, 'park-cemetery-etc-Staten Island' = 195)) AS dropoff_ntaname,

```
toUInt16(ifNull(dropoff_puma, '0')) AS dropoff_puma
```

```
FROM trips
```

This takes 3030 seconds at a speed of about 428,000 rows per second.

To load it faster, you can create the table with the `Log` engine instead of `MergeTree`. In this case, the download works faster than 200 seconds.

The table uses 126 GB of disk space.

```
:) SELECT formatReadableSize(sum(bytes)) FROM system.parts WHERE table = 'trips_mergetree' AND active
```

```
SELECT formatReadableSize(sum(bytes))
```

```
FROM system.parts
```

```
WHERE (table = 'trips_mergetree') AND active
```

```
└─formatReadableSize(sum(bytes))┐
```

126.18 GiB

Among other things, you can run the OPTIMIZE query on MergeTree. But it's not required, since everything will be fine without it.

Download of Prepared Partitions

```
curl -O https://clickhouse-datasets.s3.yandex.net/trips_mergetree/partitions/trips_mergetree.tar
tar xvf trips_mergetree.tar -C /var/lib/clickhouse # path to ClickHouse data directory
## check permissions of unpacked data, fix if required
sudo service clickhouse-server restart
clickhouse-client --query "select count(*) from datasets.trips_mergetree"
```

Info

If you will run queries described below, you have to use full table name, `datasets.trips_mergetree`.

Results on Single Server

Q1:

```
SELECT cab_type, count(*) FROM trips_mergetree GROUP BY cab_type
```

0.490 seconds.

Q2:

```
SELECT passenger_count, avg(total_amount) FROM trips_mergetree GROUP BY passenger_count
```

1.224 seconds.

Q3:

```
SELECT passenger_count, toYear(pickup_date) AS year, count(*) FROM trips_mergetree GROUP BY passenger_count, year
```

2.104 seconds.

Q4:

```
SELECT passenger_count, toYear(pickup_date) AS year, round(trip_distance) AS distance, count(*)
FROM trips_mergetree
GROUP BY passenger_count, year, distance
ORDER BY year, count(*) DESC
```

3.593 seconds.

The following server was used:

Two Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz, 16 physical kernels total, 128 GiB RAM, 8x6 TB HD on hardware RAID-5

Execution time is the best of three runs. But starting from the second run, queries read data from the file system cache. No further caching occurs: the data is read out and processed in each run.

Creating a table on three servers:

On each server:

```
CREATE TABLE default.trips_mergetree_third ( trip_id UInt32, vendor_id Enum8('1' = 1, '2' = 2, 'CMT' = 3, 'VTS' = 4, 'DDS' = 5, 'B02512' = 10, 'B02598' = 11, 'B02617' = 12, 'B02682' = 13, 'B02764' = 14), pickup_date Date, pickup_datetime DateTime, dropoff_date Date, dropoff_datetime DateTime, store_and_fwd_flag UInt8, rate_code_id UInt8, pickup_longitude Float64, pickup_latitude Float64, dropoff_longitude Float64, dropoff_latitude Float64, passenger_count UInt8, trip_distance Float64, fare_amount Float32, extra Float32, mta_tax Float32, tip_amount Float32, tolls_amount Float32, ehail_fee Float32, improvement_surcharge Float32, total_amount Float32, payment_type_Enum8('UNK' = 0, 'CSH' = 1, 'CRE' = 2, 'NOC' = 3, 'DIS' = 4), trip_type UInt8, pickup FixedString(25), dropoff FixedString(25), cab_type Enum8('yellow' = 1, 'green' = 2, 'uber' = 3), pickup_nyct2010_gid UInt8, pickup_ctlabel Float32, pickup_borocode UInt8, pickup_boroname Enum8('' = 0, 'Manhattan' = 1, 'Bronx' = 2, 'Brooklyn' = 3, 'Queens' = 4, 'Staten Island' = 5), pickup_ct2010 FixedString(6), pickup_boroc2010 FixedString(7), pickup_cdeligibil Enum8('' = 0, 'E' = 1, 'I' = 2), pickup_ntacode FixedString(4), pickup_ntaname Enum16('' = 0, 'Airport' = 1, 'Allerton-Pelham Gardens' = 2, 'Annadale-Huguenot-Prince's Bay-Eltingville' = 3, 'Arden Heights' = 4, 'Astoria' = 5, 'Auburndale' = 6, 'Baisley Park' = 7, 'Bath Beach' = 8, 'Battery Park City-Lower Manhattan' = 9, 'Bay Ridge' = 10, 'Bayside-Bayside Hills' = 11, 'Bedford' = 12, 'Bedford Park-Fordham North' = 13, 'Bellerose' = 14, 'Belmont' = 15, 'Bensonhurst East' = 16, 'Bensonhurst West' = 17, 'Borough Park' = 18, 'Breezy Point-Belle Harbor-Rockaway Park-Broad Channel' = 19, 'Briarwood-Jamaica Hills' = 20, 'Brighton Beach' = 21, 'Bronxdale' = 22, 'Brooklyn Heights-Cobble Hill' = 23, 'Brownsville' = 24, 'Bushwick North' = 25, 'Bushwick South' = 26, 'Cambria Heights' = 27, 'Canarsie' = 28, 'Carroll Gardens-Columbia Street-Red Hook' = 29, 'Central Harlem North-Polo Grounds' = 30, 'Central Harlem South' = 31, 'Charleston-Richmond Valley-Tottenville' = 32, 'Chinatown' = 33, 'Claremont-Bathgate' = 34, 'Clinton' = 35, 'Clinton Hill' = 36, 'Co-op City' = 37, 'College Point' = 38, 'Corona' = 39, 'Crotona Park East' = 40, 'Crown Heights North' = 41, 'Crown Heights South' = 42, 'Cypress Hills-City Line' = 43, 'DUMBO-Vinegar Hill-Downtown Brooklyn-Boerum Hill' = 44, 'Douglas Manor-Douglaston-Little Neck' = 45, 'Dyker Heights' = 46, 'East Concourse-Concourse Village' = 47, 'East Elmhurst' = 48, 'East Flatbush-Farragut' = 49, 'East Flushing' = 50, 'East Harlem North' = 51, 'East Harlem South' = 52, 'East New York' = 53, 'East New York (Pennsylvania Ave)' = 54, 'East Tremont' = 55, 'East Village' = 56, 'East Williamsburg' = 57, 'Eastchester-Edenwald-Baychester' = 58, 'Elmhurst' = 59, 'Elmhurst-Maspeth' = 60, 'Erasmus' = 61, 'Far Rockaway-Bayswater' = 62, 'Flatbush' = 63, 'Flatlands' = 64, 'Flushing' = 65, 'Fordham South' = 66, 'Forest Hills' = 67, 'Fort Greene' = 68, 'Fresh Meadows-Utopia' = 69, 'Ft. Totten-Bay Terrace-Clearview' = 70, 'Georgetown-Marine Park-Bergen Beach-Mill Basin' = 71, 'Glen Oaks-Floral Park-New Hyde Park' = 72, 'Glendale' = 73, 'Gramercy' = 74, 'Grasmere-Arrochar-Ft. Wadsworth' = 75, 'Gravesend' = 76, 'Great Kills' = 77, 'Greenpoint' = 78, 'Grymes Hill-Clifton-Fox Hills' = 79, 'Hamilton Heights' = 80, 'Hammels-Arverne-Edgemere' = 81, 'Highbridge' = 82, 'Hollis' = 83, 'Homecrest' = 84, 'Hudson Yards-Chelsea-Flatiron-Union Square' = 85, 'Hunters Point-Sunnyside-West Maspeth' = 86, 'Hunts Point' = 87, 'Jackson Heights' = 88, 'Jamaica' = 89, 'Jamaica Estates-Holliswood' = 90, 'Kensington-Ocean Parkway' = 91, 'Kew Gardens' = 92, 'Kew Gardens Hills' = 93, 'Kingsbridge Heights' = 94, 'Laurelton' = 95, 'Lenox Hill-Roosevelt Island' = 96, 'Lincoln Square' = 97, 'Lindenwood-Howard Beach' = 98, 'Longwood' = 99, 'Lower East Side' = 100, 'Madison' = 101, 'Manhattanville' = 102, 'Marble Hill-Inwood' = 103, 'Mariner's Harbor-Arlington-Port Ivory-Graniteville' = 104, 'Maspeth' = 105, 'Melrose South-Mott Haven North' = 106, 'Middle Village' = 107, 'Midtown-Midtown South' = 108, 'Midwood' = 109, 'Morningside Heights' = 110, 'Morrisania-Melrose' = 111, 'Mott Haven-Port Morris' = 112, 'Mount Hope' = 113, 'Murray Hill' = 114, 'Murray Hill-Kips Bay' = 115, 'New Brighton-Silver Lake' = 116, 'New Dorp-Midland Beach' = 117, 'New Springville-Bloomfield-Travis' = 118, 'North Corona' = 119, 'North Riverdale-Fieldston-Riverdale' = 120, 'North Side-South Side' = 121, 'Norwood' = 122, 'Oakland Gardens' = 123, 'Oakwood-Oakwood Beach' = 124, 'Ocean Hill' = 125, 'Ocean Parkway South' = 126, 'Old Astoria' = 127, 'Old Town-Dongan Hills-South Beach' = 128, 'Ozone Park' = 129, 'Park Slope-Gowanus' = 130, 'Parkchester' = 131, 'Pelham Bay-Country Club-City Island' = 132, 'Pelham Parkway' = 133, 'Pomonok-Flushing Heights-Hillcrest' = 134, 'Port Richmond' = 135, 'Prospect Heights' = 136, 'Prospect Lefferts Gardens-Wingate' = 137, 'Queens Village' = 138, 'Queensboro Hill' = 139, 'Queensbridge-Ravenswood-Long Island City' = 140, 'Rego Park' = 141, 'Richmond Hill' = 142, 'Ridgewood' = 143, 'Rikers Island' = 144, 'Rosedale' = 145, 'Rossville-Woodrow' = 146, 'Rugby-Renssen Village' = 147, 'Schuylerville-Throgs Neck-Edgewater Park' = 148, 'Seagate-Coney Island' = 149, 'Sheepshead Bay-Gerritsen Beach-Manhattan Beach' = 150, 'SoHo-TriBeCa-Civic Center-Little Italy' = 151, 'Soundview-Bruckner' = 152, 'Soundview-Castle Hill-Clason Point-Harding Park' = 153, 'South Jamaica' = 154, 'South Ozone Park' = 155, 'Springfield Gardens North' = 156, 'Springfield Gardens South-Brookville' = 157, 'Spuyten Duyvil-Kingsbridge' = 158, 'St. Albans' = 159, 'Stapleton-Rosebank' = 160, 'Starrett City' = 161, 'Steinway' = 162, 'Stuyvesant Heights' = 163, 'Stuyvesant Town-Cooper Village' = 164, 'Sunset Park East' = 165, 'Sunset Park West' = 166, 'Todt Hill-Emerson Hill-Heartland Village-Lighthouse Hill' = 167, 'Turtle Bay-East Midtown' = 168, 'University Heights-Morris Heights' = 169, 'Upper East Side-Carnegie Hill' = 170, 'Upper West Side' = 171, 'Van Cortlandt Village' = 172, 'Van Nest-Morris Park-Westchester Square' = 173, 'Washington Heights North' = 174, 'Washington Heights South' = 175, 'West Brighton' = 176, 'West Concourse' = 177, 'West Farms-Bronx River' = 178, 'West New Brighton-New Brighton-St. George' = 179, 'West Village' = 180, 'Westchester-Unionport' = 181, 'Westerleigh' = 182, 'Whitestone' = 183, 'Williamsbridge-Olinville' = 184, 'Williamsburg' = 185, 'Windsor Terrace' = 186, 'Woodhaven' = 187, 'Woodlawn-Wakefield' = 188, 'Woodside' = 189, 'Yorkville' = 190, 'park-cemetery-etc-Bronx' = 191, 'park-cemetery-etc-Brooklyn' = 192, 'park-cemetery-etc-Manhattan' = 193, 'park-cemetery-etc-Queens' = 194, 'park-cemetery-etc-Staten Island' = 195), pickup_puma UInt16, dropoff_nyct2010_gid UInt8, dropoff_ctlabel Float32, dropoff_borocode UInt8, dropoff_boroname Enum8('' = 0, 'Manhattan' = 1, 'Bronx' = 2, 'Brooklyn' = 3, 'Queens' = 4, 'Staten Island' = 5), dropoff_ct2010 FixedString(6), dropoff_boroc2010 FixedString(7), dropoff_cdeligibil Enum8('' = 0, 'E' = 1, 'I' = 2), dropoff_ntacode FixedString(4), dropoff_ntaname Enum16('' = 0, 'Airport' = 1, 'Allerton-Pelham Gardens' = 2, 'Annadale-Huguenot-Prince's Bay-Eltingville' = 3, 'Arden Heights' = 4, 'Astoria' = 5, 'Auburndale' = 6, 'Baisley Park' = 7, 'Bath Beach'
```



```

huguenot-france's Bay-Eltingville' = 3, 'Arden Heights' = 4, 'Astoria' = 5, 'Auburndale' = 6, 'Baisley Park' = 7, 'Bath Beach'
= 8, 'Battery Park City-Lower Manhattan' = 9, 'Bay Ridge' = 10, 'Bayside-Bayside Hills' = 11, 'Bedford' = 12, 'Bedford Park-
Fordham North' = 13, 'Bellerose' = 14, 'Belmont' = 15, 'Bensonhurst East' = 16, 'Bensonhurst West' = 17, 'Borough Park' =
18, 'Breezy Point-Belle Harbor-Rockaway Park-Broad Channel' = 19, 'Briarwood-Jamaica Hills' = 20, 'Brighton Beach' = 21,
'Bronxdale' = 22, 'Brooklyn Heights-Cobble Hill' = 23, 'Brownsville' = 24, 'Bushwick North' = 25, 'Bushwick South' = 26,
'Cambria Heights' = 27, 'Canarsie' = 28, 'Carroll Gardens-Columbia Street-Red Hook' = 29, 'Central Harlem North-Polo
Grounds' = 30, 'Central Harlem South' = 31, 'Charleston-Richmond Valley-Tottenville' = 32, 'Chinatown' = 33, 'Claremont-
Bathgate' = 34, 'Clinton' = 35, 'Clinton Hill' = 36, 'Co-op City' = 37, 'College Point' = 38, 'Corona' = 39, 'Crotona Park East'
= 40, 'Crown Heights North' = 41, 'Crown Heights South' = 42, 'Cypress Hills-City Line' = 43, 'DUMBO-Vinegar Hill-
Downtown Brooklyn-Boerum Hill' = 44, 'Douglas Manor-Douglaston-Little Neck' = 45, 'Dyker Heights' = 46, 'East Concourse-
Concourse Village' = 47, 'East Elmhurst' = 48, 'East Flatbush-Farragut' = 49, 'East Flushing' = 50, 'East Harlem North' = 51,
'East Harlem South' = 52, 'East New York' = 53, 'East New York (Pennsylvania Ave)' = 54, 'East Tremont' = 55, 'East Village'
= 56, 'East Williamsburg' = 57, 'Eastchester-Edenwald-Baychester' = 58, 'Elmhurst' = 59, 'Elmhurst-Maspeth' = 60,
'Erasmus' = 61, 'Far Rockaway-Bayswater' = 62, 'Flatbush' = 63, 'Flatlands' = 64, 'Flushing' = 65, 'Fordham South' = 66,
'Forest Hills' = 67, 'Fort Greene' = 68, 'Fresh Meadows-Utopia' = 69, 'Ft. Totten-Bay Terrace-Clearview' = 70, 'Georgetown-
Marine Park-Bergen Beach-Mill Basin' = 71, 'Glen Oaks-Floral Park-New Hyde Park' = 72, 'Glendale' = 73, 'Gramercy' = 74,
'Grasmere-Arrochar-Ft. Wadsworth' = 75, 'Gravesend' = 76, 'Great Kills' = 77, 'Greenpoint' = 78, 'Grymes Hill-Clifton-Fox
Hills' = 79, 'Hamilton Heights' = 80, 'Hammels-Arverne-Edgemere' = 81, 'Highbridge' = 82, 'Hollis' = 83, 'Homecrest' = 84,
'Hudson Yards-Chelsea-Flatiron-Union Square' = 85, 'Hunters Point-Sunnyside-West Maspeth' = 86, 'Hunts Point' = 87,
'Jackson Heights' = 88, 'Jamaica' = 89, 'Jamaica Estates-Holliswood' = 90, 'Kensington-Ocean Parkway' = 91, 'Kew Gardens'
= 92, 'Kew Gardens Hills' = 93, 'Kingsbridge Heights' = 94, 'Laurelton' = 95, 'Lenox Hill-Roosevelt Island' = 96, 'Lincoln
Square' = 97, 'Lindenwood-Howard Beach' = 98, 'Longwood' = 99, 'Lower East Side' = 100, 'Madison' = 101,
'Manhattanville' = 102, 'Marble Hill-Inwood' = 103, 'Mariner's Harbor-Arlington-Port Ivory-Graniteville' = 104, 'Maspeth' =
105, 'Melrose South-Mott Haven North' = 106, 'Middle Village' = 107, 'Midtown-Midtown South' = 108, 'Midwood' = 109,
'Morningside Heights' = 110, 'Morrisania-Melrose' = 111, 'Mott Haven-Port Morris' = 112, 'Mount Hope' = 113, 'Murray Hill'
= 114, 'Murray Hill-Kips Bay' = 115, 'New Brighton-Silver Lake' = 116, 'New Dorp-Midland Beach' = 117, 'New Springville-
Bloomfield-Travis' = 118, 'North Corona' = 119, 'North Riverdale-Fieldston-Riverdale' = 120, 'North Side-South Side' = 121,
'Norwood' = 122, 'Oakland Gardens' = 123, 'Oakwood-Oakwood Beach' = 124, 'Ocean Hill' = 125, 'Ocean Parkway South' =
126, 'Old Astoria' = 127, 'Old Town-Dongan Hills-South Beach' = 128, 'Ozone Park' = 129, 'Park Slope-Gowanus' = 130,
'Parkchester' = 131, 'Pelham Bay-Country Club-City Island' = 132, 'Pelham Parkway' = 133, 'Pomonok-Flushing Heights-
Hillcrest' = 134, 'Port Richmond' = 135, 'Prospect Heights' = 136, 'Prospect Lefferts Gardens-Wingate' = 137, 'Queens
Village' = 138, 'Queensboro Hill' = 139, 'Queensbridge-Ravenswood-Long Island City' = 140, 'Rego Park' = 141, 'Richmond
Hill' = 142, 'Ridgewood' = 143, 'Rikers Island' = 144, 'Rosedale' = 145, 'Rossville-Woodrow' = 146, 'Rugby-Renssen Village'
= 147, 'Schuylerville-Throgs Neck-Edgewater Park' = 148, 'Seagate-Coney Island' = 149, 'Sheepshead Bay-Gerritsen Beach-
Manhattan Beach' = 150, 'SoHo-TriBeCa-Civic Center-Little Italy' = 151, 'Soundview-Bruckner' = 152, 'Soundview-Castle
Hill-Clason Point-Harding Park' = 153, 'South Jamaica' = 154, 'South Ozone Park' = 155, 'Springfield Gardens North' = 156,
'Springfield Gardens South-Brookville' = 157, 'Spuyten Duyvil-Kingsbridge' = 158, 'St. Albans' = 159, 'Stapleton-Rosebank'
= 160, 'Starrett City' = 161, 'Steinway' = 162, 'Stuyvesant Heights' = 163, 'Stuyvesant Town-Cooper Village' = 164,
'Sunset Park East' = 165, 'Sunset Park West' = 166, 'Todt Hill-Emerson Hill-Heartland Village-Lighthouse Hill' = 167, 'Turtle
Bay-East Midtown' = 168, 'University Heights-Morris Heights' = 169, 'Upper East Side-Carnegie Hill' = 170, 'Upper West
Side' = 171, 'Van Cortlandt Village' = 172, 'Van Nest-Morris Park-Westchester Square' = 173, 'Washington Heights North' =
174, 'Washington Heights South' = 175, 'West Brighton' = 176, 'West Concourse' = 177, 'West Farms-Bronx River' = 178,
'West New Brighton-New Brighton-St. George' = 179, 'West Village' = 180, 'Westchester-Unionport' = 181, 'Westerleigh' =
182, 'Whitestone' = 183, 'Williamsbridge-Olinville' = 184, 'Williamsburg' = 185, 'Windsor Terrace' = 186, 'Woodhaven' =
187, 'Woodlawn-Wakefield' = 188, 'Woodside' = 189, 'Yorkville' = 190, 'park-cemetery-etc-Bronx' = 191, 'park-cemetery-
etc-Brooklyn' = 192, 'park-cemetery-etc-Manhattan' = 193, 'park-cemetery-etc-Queens' = 194, 'park-cemetery-etc-Staten
Island' = 195), dropoff_puma UInt16) ENGINE = MergeTree(pickup_date, pickup_datetime, 8192)

```

On the source server:

```

CREATE TABLE trips_mergetree_x3 AS trips_mergetree_third ENGINE = Distributed(perftest, default, trips_mergetree_third,
rand())

```

The following query redistributes data:

```

INSERT INTO trips_mergetree_x3 SELECT * FROM trips_mergetree

```

This takes 2454 seconds.

On three servers:

O1: 0.212 seconds.

Q2: 0.438 seconds.
Q3: 0.733 seconds.
Q4: 1.241 seconds.

No surprises here, since the queries are scaled linearly.

We also have results from a cluster of 140 servers:

Q1: 0.028 sec.
Q2: 0.043 sec.
Q3: 0.051 sec.
Q4: 0.072 sec.

In this case, the query processing time is determined above all by network latency.

We ran queries using a client located in a Yandex datacenter in Finland on a cluster in Russia, which added about 20 ms of latency.

Summary

servers	Q1	Q2	Q3	Q4
1	0.490	1.224	2.104	3.593
3	0.212	0.438	0.733	1.241
140	0.028	0.043	0.051	0.072

AMPLab Big Data Benchmark

See <https://amplab.cs.berkeley.edu/benchmark/>

Sign up for a free account at <https://aws.amazon.com>. You will need a credit card, email and phone number. Get a new access key at https://console.aws.amazon.com/iam/home?nc2=h_m_sc#security_credential

Run the following in the console:

```
sudo apt-get install s3cmd
mkdir tiny; cd tiny;
s3cmd sync s3://big-data-benchmark/pavlo/text-deflate/tiny/ .
cd ..
mkdir 1node; cd 1node;
s3cmd sync s3://big-data-benchmark/pavlo/text-deflate/1node/ .
cd ..
mkdir 5nodes; cd 5nodes;
s3cmd sync s3://big-data-benchmark/pavlo/text-deflate/5nodes/ .
cd ..
```

Run the following ClickHouse queries:

```
CREATE TABLE rankings_tiny
(
  pageURL String,
  pageRank UInt32,
  avgDuration UInt32
) ENGINE = Log;

CREATE TABLE uservisits_tiny
(
  sourceIP String,
```

```

destinationURL String,
visitDate Date,
adRevenue Float32,
UserAgent String,
cCode FixedString(3),
lCode FixedString(6),
searchWord String,
duration UInt32
) ENGINE = MergeTree(visitDate, visitDate, 8192);

```

```

CREATE TABLE rankings_1node
(
    pageURL String,
    pageRank UInt32,
    avgDuration UInt32
) ENGINE = Log;

```

```

CREATE TABLE uservisits_1node
(
    sourceIP String,
    destinationURL String,
    visitDate Date,
    adRevenue Float32,
    UserAgent String,
    cCode FixedString(3),
    lCode FixedString(6),
    searchWord String,
    duration UInt32
) ENGINE = MergeTree(visitDate, visitDate, 8192);

```

```

CREATE TABLE rankings_5nodes_on_single
(
    pageURL String,
    pageRank UInt32,
    avgDuration UInt32
) ENGINE = Log;

```

```

CREATE TABLE uservisits_5nodes_on_single
(
    sourceIP String,
    destinationURL String,
    visitDate Date,
    adRevenue Float32,
    UserAgent String,
    cCode FixedString(3),
    lCode FixedString(6),
    searchWord String,
    duration UInt32
) ENGINE = MergeTree(visitDate, visitDate, 8192);

```

Go back to the console:

```

for i in tiny/rankings/*.deflate; do echo $i; zlib-flate -uncompress < $i | clickhouse-client --host=example-perftest01j --
query="INSERT INTO rankings_tiny FORMAT CSV"; done
for i in tiny/uservisits/*.deflate; do echo $i; zlib-flate -uncompress < $i | clickhouse-client --host=example-perftest01j --
query="INSERT INTO uservisits_tiny FORMAT CSV"; done
for i in 1node/rankings/*.deflate; do echo $i; zlib-flate -uncompress < $i | clickhouse-client --host=example-perftest01j --
query="INSERT INTO rankings_1node FORMAT CSV"; done
for i in 1node/uservisits/*.deflate; do echo $i; zlib-flate -uncompress < $i | clickhouse-client --host=example-perftest01j --
query="INSERT INTO uservisits_1node FORMAT CSV"; done
for i in 5nodes/rankings/*.deflate; do echo $i; zlib-flate -uncompress < $i | clickhouse-client --host=example-perftest01j --
query="INSERT INTO rankings_5nodes_on_single FORMAT CSV"; done
for i in 5nodes/uservisits/*.deflate; do echo $i; zlib-flate -uncompress < $i | clickhouse-client --host=example-perftest01j --

```

```
query="INSERT INTO uservisits_5nodes_on_single FORMAT CSV"; done
```

Queries for obtaining data samples:

```
SELECT pageURL, pageRank FROM rankings_1node WHERE pageRank > 1000

SELECT substring(sourceIP, 1, 8), sum(adRevenue) FROM uservisits_1node GROUP BY substring(sourceIP, 1, 8)

SELECT
  sourceIP,
  sum(adRevenue) AS totalRevenue,
  avg(pageRank) AS pageRank
FROM rankings_1node ALL INNER JOIN
(
  SELECT
    sourceIP,
    destinationURL AS pageURL,
    adRevenue
  FROM uservisits_1node
  WHERE (visitDate > '1980-01-01') AND (visitDate < '1980-04-01')
) USING pageURL
GROUP BY sourceIP
ORDER BY totalRevenue DESC
LIMIT 1
```

WikiStat

See: <http://dumps.wikimedia.org/other/pagecounts-raw/>

Creating a table:

```
CREATE TABLE wikistat
(
  date Date,
  time DateTime,
  project String,
  subproject String,
  path String,
  hits UInt64,
  size UInt64
) ENGINE = MergeTree(date, (path, time), 8192);
```

Loading data:

```
for i in {2007..2016}; do for j in {01..12}; do echo $i-$j >&2; curl -sSL "http://dumps.wikimedia.org/other/pagecounts-raw/$i/$i-$j/" | grep -oE 'pagecounts-[0-9]+-[0-9]+\.gz'; done; done | sort | uniq | tee links.txt
cat links.txt | while read link; do wget http://dumps.wikimedia.org/other/pagecounts-raw/$(echo $link | sed -r 's/pagecounts-([0-9]{4})([0-9]{2})([0-9]{2})-[0-9]+\.gz/\1/');$(echo $link | sed -r 's/pagecounts-([0-9]{4})([0-9]{2})([0-9]{2})-[0-9]+\.gz/\1-\2/');$link; done
ls -l /opt/wikistat/ | grep gz | while read i; do echo $i; gzip -cd /opt/wikistat/$i | ./wikistat-loader --time="$(echo -n $i | sed -r 's/pagecounts-([0-9]{4})([0-9]{2})([0-9]{2})-[0-9]+\.gz/\1-\2-\3 \4-00-00/')" | clickhouse-client --query="INSERT INTO wikistat FORMAT TabSeparated"; done
```

Terabyte of Click Logs from Criteo

Download the data from <http://labs.criteo.com/downloads/download-terabyte-click-logs/>

Create a table to import the log to:

```
CREATE TABLE criteo_log (date Date, clicked UInt8, int1 Int32, int2 Int32, int3 Int32, int4 Int32, int5 Int32, int6 Int32, int7 Int32, int8 Int32, int9 Int32, int10 Int32, int11 Int32, int12 Int32, int13 Int32, cat1 String, cat2 String, cat3 String, cat4 String, cat5 String, cat6 String, cat7 String, cat8 String, cat9 String, cat10 String, cat11 String, cat12 String, cat13 String, cat14 String, cat15 String, cat16 String, cat17 String, cat18 String, cat19 String, cat20 String, cat21 String, cat22 String, cat23 String, cat24 String, cat25 String, cat26 String) ENGINE = Log
```

Download the data:

```
for i in {00..23}; do echo $i; zcat datasets/criteo/day_${i#0}.gz | sed -r 's/^/2000-01-'\${i/00/24}'\t/' | clickhouse-client --host=example-perftest01j --query="INSERT INTO criteo_log FORMAT TabSeparated"; done
```

Create a table for the converted data:

```
CREATE TABLE criteo
(
    date Date,
    clicked UInt8,
    int1 Int32,
    int2 Int32,
    int3 Int32,
    int4 Int32,
    int5 Int32,
    int6 Int32,
    int7 Int32,
    int8 Int32,
    int9 Int32,
    int10 Int32,
    int11 Int32,
    int12 Int32,
    int13 Int32,
    icat1 UInt32,
    icat2 UInt32,
    icat3 UInt32,
    icat4 UInt32,
    icat5 UInt32,
    icat6 UInt32,
    icat7 UInt32,
    icat8 UInt32,
    icat9 UInt32,
    icat10 UInt32,
    icat11 UInt32,
    icat12 UInt32,
    icat13 UInt32,
    icat14 UInt32,
    icat15 UInt32,
    icat16 UInt32,
    icat17 UInt32,
    icat18 UInt32,
    icat19 UInt32,
    icat20 UInt32,
    icat21 UInt32,
    icat22 UInt32,
    icat23 UInt32,
    icat24 UInt32,
    icat25 UInt32,
    icat26 UInt32
) ENGINE = MergeTree(date, intHash32(icat1), (date, intHash32(icat1)), 8192)
```

Transform data from the raw log and put it in the second table:

```
INSERT INTO criteo SELECT date, clicked, int1, int2, int3, int4, int5, int6, int7, int8, int9, int10, int11, int12, int13,
```

```

INSERT INTO criteo SELECT date, clicked, int1, int2, int3, int4, int5, int6, int7, int8, int9, int10, int11, int12, int13,
reinterpretAsUInt32(unhex(cat1)) AS icat1, reinterpretAsUInt32(unhex(cat2)) AS icat2, reinterpretAsUInt32(unhex(cat3)) AS
icat3, reinterpretAsUInt32(unhex(cat4)) AS icat4, reinterpretAsUInt32(unhex(cat5)) AS icat5,
reinterpretAsUInt32(unhex(cat6)) AS icat6, reinterpretAsUInt32(unhex(cat7)) AS icat7, reinterpretAsUInt32(unhex(cat8)) AS
icat8, reinterpretAsUInt32(unhex(cat9)) AS icat9, reinterpretAsUInt32(unhex(cat10)) AS icat10,
reinterpretAsUInt32(unhex(cat11)) AS icat11, reinterpretAsUInt32(unhex(cat12)) AS icat12,
reinterpretAsUInt32(unhex(cat13)) AS icat13, reinterpretAsUInt32(unhex(cat14)) AS icat14,
reinterpretAsUInt32(unhex(cat15)) AS icat15, reinterpretAsUInt32(unhex(cat16)) AS icat16,
reinterpretAsUInt32(unhex(cat17)) AS icat17, reinterpretAsUInt32(unhex(cat18)) AS icat18,
reinterpretAsUInt32(unhex(cat19)) AS icat19, reinterpretAsUInt32(unhex(cat20)) AS icat20,
reinterpretAsUInt32(unhex(cat21)) AS icat21, reinterpretAsUInt32(unhex(cat22)) AS icat22,
reinterpretAsUInt32(unhex(cat23)) AS icat23, reinterpretAsUInt32(unhex(cat24)) AS icat24,
reinterpretAsUInt32(unhex(cat25)) AS icat25, reinterpretAsUInt32(unhex(cat26)) AS icat26 FROM criteo_log;

DROP TABLE criteo_log;

```

Star Schema Benchmark

Compiling dbgen:

```

git clone git@github.com:vadimtk/ssb-dbgen.git
cd ssb-dbgen
make

```

Generating data:

```

./dbgen -s 1000 -T c
./dbgen -s 1000 -T l
./dbgen -s 1000 -T p
./dbgen -s 1000 -T s
./dbgen -s 1000 -T d

```

Creating tables in ClickHouse:

```

CREATE TABLE customer
(
    C_CUSTKEY      UInt32,
    C_NAME         String,
    C_ADDRESS      String,
    C_CITY         LowCardinality(String),
    C_NATION       LowCardinality(String),
    C_REGION       LowCardinality(String),
    C_PHONE        String,
    C_MKTSEGMENT   LowCardinality(String)
)
ENGINE = MergeTree ORDER BY (C_CUSTKEY);

CREATE TABLE lineorder
(
    LO_ORDERKEY      UInt32,
    LO_LINENUMBER    UInt8,
    LO_CUSTKEY       UInt32,
    LO_PARTKEY       UInt32,
    LO_SUPPKEY       UInt32,
    LO_ORDERDATE     Date,
    LO_ORDERPRIORITY LowCardinality(String),
    LO_SHIPPRIORITY  UInt8,
    LO_QUANTITY      UInt8,
    LO_EXTENDEDPRICE UInt32,
    LO_ORDTOTALPRICE UInt32,
    LO_DISCOUNT     UInt8
)

```

```

LO_DISCOUNT      UInt8,
LO_REVENUE          UInt32,
LO_SUPPLYCOST       UInt32,
LO_TAX              UInt8,
LO_COMMITDATE       Date,
LO_SHIPMODE         LowCardinality(String)
)
ENGINE = MergeTree PARTITION BY toYear(LO_ORDERDATE) ORDER BY (LO_ORDERDATE, LO_ORDERKEY);

CREATE TABLE part
(
    P_PARTKEY      UInt32,
    P_NAME          String,
    P_MFGR          LowCardinality(String),
    P_CATEGORY      LowCardinality(String),
    P_BRAND          LowCardinality(String),
    P_COLOR          LowCardinality(String),
    P_TYPE          LowCardinality(String),
    P_SIZE          UInt8,
    P_CONTAINER      LowCardinality(String)
)
ENGINE = MergeTree ORDER BY P_PARTKEY;

CREATE TABLE supplier
(
    S_SUPPKEY      UInt32,
    S_NAME          String,
    S_ADDRESS       String,
    S_CITY          LowCardinality(String),
    S_NATION         LowCardinality(String),
    S_REGION         LowCardinality(String),
    S_PHONE          String
)
ENGINE = MergeTree ORDER BY S_SUPPKEY;

```

Inserting data:

```

clickhouse-client --query "INSERT INTO customer FORMAT CSV" < customer.tbl
clickhouse-client --query "INSERT INTO part FORMAT CSV" < part.tbl
clickhouse-client --query "INSERT INTO supplier FORMAT CSV" < supplier.tbl
clickhouse-client --query "INSERT INTO lineorder FORMAT CSV" < lineorder.tbl

```

Converting "star schema" to denormalized "flat schema":

```

SET max_memory_usage = 20000000000, allow_experimental_multiple_joins_emulation = 1;

CREATE TABLE lineorder_flat
ENGINE = MergeTree
PARTITION BY toYear(LO_ORDERDATE)
ORDER BY (LO_ORDERDATE, LO_ORDERKEY) AS
SELECT *
FROM lineorder
ANY INNER JOIN customer ON LO_CUSTKEY = C_CUSTKEY
ANY INNER JOIN supplier ON LO_SUPPKEY = S_SUPPKEY
ANY INNER JOIN part ON LO_PARTKEY = P_PARTKEY;

ALTER TABLE lineorder_flat DROP COLUMN C_CUSTKEY, DROP COLUMN S_SUPPKEY, DROP COLUMN P_PARTKEY;

```

Running the queries:

Q1.1


```
SELECT sum(LO_EXTENDEDPRICE * LO_DISCOUNT) AS revenue FROM lineorder_flat WHERE toYear(LO_ORDERDATE) = 1993 AND LO_DISCOUNT BETWEEN 1 AND 3 AND LO_QUANTITY < 25;
```

Q1.2

```
SELECT sum(LO_EXTENDEDPRICE * LO_DISCOUNT) AS revenue FROM lineorder_flat WHERE toYYYYMM(LO_ORDERDATE) = 199401 AND LO_DISCOUNT BETWEEN 4 AND 6 AND LO_QUANTITY BETWEEN 26 AND 35;
```

Q1.3

```
SELECT sum(LO_EXTENDEDPRICE * LO_DISCOUNT) AS revenue FROM lineorder_flat WHERE toISOWeek(LO_ORDERDATE) = 6 AND toYear(LO_ORDERDATE) = 1994 AND LO_DISCOUNT BETWEEN 5 AND 7 AND LO_QUANTITY BETWEEN 26 AND 35;
```

Q2.1

```
SELECT sum(LO_REVENUE), toYear(LO_ORDERDATE) AS year, P_BRAND FROM lineorder_flat WHERE P_CATEGORY = 'MFGR#12' AND S_REGION = 'AMERICA' GROUP BY year, P_BRAND ORDER BY year, P_BRAND;
```

Q2.2

```
SELECT sum(LO_REVENUE), toYear(LO_ORDERDATE) AS year, P_BRAND FROM lineorder_flat WHERE P_BRAND BETWEEN 'MFGR#2221' AND 'MFGR#2228' AND S_REGION = 'ASIA' GROUP BY year, P_BRAND ORDER BY year, P_BRAND;
```

Q2.3

```
SELECT sum(LO_REVENUE), toYear(LO_ORDERDATE) AS year, P_BRAND FROM lineorder_flat WHERE P_BRAND = 'MFGR#2239' AND S_REGION = 'EUROPE' GROUP BY year, P_BRAND ORDER BY year, P_BRAND;
```

Q3.1

```
SELECT C_NATION, S_NATION, toYear(LO_ORDERDATE) AS year, sum(LO_REVENUE) AS revenue FROM lineorder_flat WHERE C_REGION = 'ASIA' AND S_REGION = 'ASIA' AND year >= 1992 AND year <= 1997 GROUP BY C_NATION, S_NATION, year ORDER BY year asc, revenue desc;
```

Q3.2

```
SELECT C_CITY, S_CITY, toYear(LO_ORDERDATE) AS year, sum(LO_REVENUE) AS revenue FROM lineorder_flat WHERE C_NATION = 'UNITED STATES' AND S_NATION = 'UNITED STATES' AND year >= 1992 AND year <= 1997 GROUP BY C_CITY, S_CITY, year ORDER BY year asc, revenue desc;
```

Q3.3

```
SELECT C_CITY, S_CITY, toYear(LO_ORDERDATE) AS year, sum(LO_REVENUE) AS revenue FROM lineorder_flat WHERE (C_CITY = 'UNITED K11' OR C_CITY = 'UNITED K15') AND (S_CITY = 'UNITED K11' OR S_CITY = 'UNITED K15') AND year >= 1992 AND year <= 1997 GROUP BY C_CITY, S_CITY, year ORDER BY year asc, revenue desc;
```

Q3.4

```
SELECT C_CITY, S_CITY, toYear(LO_ORDERDATE) AS year, sum(LO_REVENUE) AS revenue FROM lineorder_flat WHERE (C_CITY = 'UNITED K11' OR C_CITY = 'UNITED K15') AND (S_CITY = 'UNITED K11' OR S_CITY = 'UNITED K15') AND toYYYYMM(LO_ORDERDATE) = '199712' GROUP BY C_CITY, S_CITY, year ORDER BY year asc, revenue desc;
```

Q4.1

```
SELECT toYear(LO_ORDERDATE) AS year, C_NATION, sum(LO_REVENUE - LO_SUPPLYCOST) AS profit FROM lineorder_flat WHERE C_REGION = 'AMERICA' AND S_REGION = 'AMERICA' AND (P_MFGR = 'MFGR#1' OR P_MFGR = 'MFGR#2') GROUP BY year, C_NATION ORDER BY year, C_NATION;
```

Q4.2

```
SELECT toYear(LO_ORDERDATE) AS year, S_NATION, P_CATEGORY, sum(LO_REVENUE - LO_SUPPLYCOST) AS profit FROM lineorder_flat WHERE C_REGION = 'AMERICA' AND S_REGION = 'AMERICA' AND (year = 1997 OR year = 1998) AND (P_MFGR = 'MFGR#1' OR P_MFGR = 'MFGR#2') GROUP BY year, S_NATION, P_CATEGORY ORDER BY year, S_NATION, P_CATEGORY;
```

Q4.3

```
SELECT toYear(LO_ORDERDATE) AS year, S_CITY, P_BRAND, sum(LO_REVENUE - LO_SUPPLYCOST) AS profit FROM lineorder_flat WHERE S_NATION = 'UNITED STATES' AND (year = 1997 OR year = 1998) AND P_CATEGORY = 'MFGR#14' GROUP BY year, S_CITY, P_BRAND ORDER BY year, S_CITY, P_BRAND;
```

Anonymized Yandex.Metrica Data

Dataset consists of two tables containing anonymized data about hits (`hits_v1`) and visits (`visits_v1`) of Yandex.Metrica. Each of the tables can be downloaded as a compressed `tsv.xz` file or as prepared partitions.

Obtaining Tables from Prepared Partitions

Download and import hits:

```
curl -O https://clickhouse-datasets.s3.yandex.net/hits/partitions/hits_v1.tar
tar xvf hits_v1.tar -C /var/lib/clickhouse # path to ClickHouse data directory
## check permissions on unpacked data, fix if required
sudo service clickhouse-server restart
clickhouse-client --query "SELECT COUNT(*) FROM datasets.hits_v1"
```

Download and import visits:

```
curl -O https://clickhouse-datasets.s3.yandex.net/visits/partitions/visits_v1.tar
tar xvf visits_v1.tar -C /var/lib/clickhouse # path to ClickHouse data directory
## check permissions on unpacked data, fix if required
sudo service clickhouse-server restart
clickhouse-client --query "SELECT COUNT(*) FROM datasets.visits_v1"
```

Obtaining Tables from Compressed tsv-file

Download and import hits from compressed tsv-file

```
curl https://clickhouse-datasets.s3.yandex.net/hits/tsv/hits_v1.tsv.xz | unxz --threads=`nproc` > hits_v1.tsv
## now create table
clickhouse-client --query "CREATE DATABASE IF NOT EXISTS datasets"
clickhouse-client --query "CREATE TABLE datasets.hits_v1 ( WatchID UInt64, JavaEnable UInt8, Title String, GoodEvent
Int16, EventTime DateTime, EventDate Date, CounterID UInt32, ClientIP UInt32, ClientIP6 FixedString(16), RegionID
UInt32, UserID UInt64, CounterClass Int8, OS UInt8, UserAgent UInt8, URL String, Referer String, URLEDomain String,
RefererDomain String, Refresh UInt8, IsRobot UInt8, RefererCategories Array(UInt16), URLECategories Array(UInt16),
URLRegions Array(UInt32), RefererRegions Array(UInt32), ResolutionWidth UInt16, ResolutionHeight UInt16,
ResolutionDepth UInt8, FlashMajor UInt8, FlashMinor UInt8, FlashMinor2 String, NetMajor UInt8, NetMinor UInt8,
UserAgentMajor UInt16, UserAgentMinor FixedString(2), CookieEnable UInt8, JavascriptEnable UInt8, IsMobile UInt8,
MobilePhone UInt8, MobilePhoneModel String, Params String, IPNetworkID UInt32, TrafficSourceID Int8, SearchEngineID
UInt16, SearchPhrase String, AdvEngineID UInt8, IsArtificial UInt8, WindowClientWidth UInt16, WindowClientHeight
UInt16, ClientTimeZone Int16, ClientEventTime DateTime, SilverlightVersion1 UInt8, SilverlightVersion2 UInt8,
SilverlightVersion3 UInt32, SilverlightVersion4 UInt16, PageCharset String, CodeVersion UInt32, IsLink UInt8, IsDownload
UInt8, IsNotBounce UInt8, FUniqID UInt64, HID UInt32, IsOldCounter UInt8, IsEvent UInt8, IsParameter UInt8,
DontCountHits UInt8, WithHash UInt8, HitColor FixedString(1), UTCEventTime DateTime, Age UInt8, Sex UInt8, Income
UInt8, Interests UInt16, Robotness UInt8, GeneralInterests Array(UInt16), RemoteIP UInt32, RemoteIP6 FixedString(16),
WindowName Int32, OpenerName Int32, HistoryLength Int16, BrowserLanguage FixedString(2), BrowserCountry
FixedString(2), SocialNetwork String, SocialAction String, HTTPError UInt16, SendTiming Int32, DNSTiming Int32,
ConnectTiming Int32, ResponseStartTiming Int32, ResponseEndTiming Int32, FetchTiming Int32, RedirectTiming Int32,
DOMInteractiveTiming Int32, DOMContentLoadedTiming Int32, DOMCompleteTiming Int32, LoadEventStartTiming Int32,
LoadEventEndTiming Int32, NSToDOMContentLoadedTiming Int32, FirstPaintTiming Int32, RedirectCount Int8,
SocialSourceNetworkID UInt8, SocialSourcePage String, ParamPrice Int64, ParamOrderID String, ParamCurrency
FixedString(3), ParamCurrencyID UInt16, GoalsReached Array(UInt32), OpenstatServiceName String,
OpenstatCampaignID String, OpenstatAdID String, OpenstatSourceID String, UTMSource String, UTMMedium String,
UTMCampaign String, UTMContent String, UTMTerm String, FromTag String, HasGCLID UInt8, RefererHash UInt64,
URLHash UInt64, CLID UInt32, YCLID UInt64, ShareService String, ShareURL String, ShareTitle String, ParsedParams
Nested(Key1 String, Key2 String, Key3 String, Key4 String, Key5 String, ValueDouble Float64), IslandID FixedString(16),
RequestNum UInt32, RequestTry UInt8) ENGINE = MergeTree() PARTITION BY toYYYYMM(EventDate) ORDER BY (CounterID,
EventDate, intHash32(UserID)) SAMPLE BY intHash32(UserID) SETTINGS index_granularity = 8192"
## import data
cat hits_v1.tsv | clickhouse-client --query "INSERT INTO datasets.hits_v1 FORMAT TSV" --max_insert_block_size=100000
## optionally you can optimize table
clickhouse-client --query "OPTIMIZE TABLE datasets.hits_v1 FINAL"
clickhouse-client --query "SELECT COUNT(*) FROM datasets.hits_v1"
```

Download and import visits from compressed tsv-file

```

curl https://clickhouse-datasets.s3.yandex.net/visits/tsv/visits_v1.tsv.xz | unxz --threads=`nproc` > visits_v1.tsv
## now create table
clickhouse-client --query "CREATE DATABASE IF NOT EXISTS datasets"
clickhouse-client --query "CREATE TABLE datasets.visits_v1 ( CounterID UInt32, StartDate Date, Sign Int8, IsNew UInt8,
VisitID UInt64, UserID UInt64, StartTime DateTime, Duration UInt32, UTCStartTime DateTime, PageViews Int32, Hits
Int32, IsBounce UInt8, Referer String, StartURL String, RefererDomain String, StartURLDomain String, EndURL String,
LinkURL String, IsDownload UInt8, TrafficSourceID Int8, SearchEngineID UInt16, SearchPhrase String, AdvEngineID UInt8,
PlacelD Int32, RefererCategories Array(UInt16), URLCategories Array(UInt16), URLRegions Array(UInt32), RefererRegions
Array(UInt32), IsYandex UInt8, GoalReachesDepth Int32, GoalReachesURL Int32, GoalReachesAny Int32,
SocialSourceNetworkID UInt8, SocialSourcePage String, MobilePhoneModel String, ClientEventTime DateTime, RegionID
UInt32, ClientIP UInt32, ClientIP6 FixedString(16), RemoteIP UInt32, RemoteIP6 FixedString(16), IPNetworkID UInt32,
SilverlightVersion3 UInt32, CodeVersion UInt32, ResolutionWidth UInt16, ResolutionHeight UInt16, UserAgentMajor
UInt16, UserAgentMinor UInt16, WindowClientWidth UInt16, WindowClientHeight UInt16, SilverlightVersion2 UInt8,
SilverlightVersion4 UInt16, FlashVersion3 UInt16, FlashVersion4 UInt16, ClientTimeZone Int16, OS UInt8, UserAgent
UInt8, ResolutionDepth UInt8, FlashMajor UInt8, FlashMinor UInt8, NetMajor UInt8, NetMinor UInt8, MobilePhone UInt8,
SilverlightVersion1 UInt8, Age UInt8, Sex UInt8, Income UInt8, JavaEnable UInt8, CookieEnable UInt8, JavascriptEnable
UInt8, IsMobile UInt8, BrowserLanguage UInt16, BrowserCountry UInt16, Interests UInt16, Robotness UInt8,
GeneralInterests Array(UInt16), Params Array(String), Goals Nested(ID UInt32, Serial UInt32, EventTime DateTime, Price
Int64, OrderID String, CurrencyID UInt32), WatchIDs Array(UInt64), ParamSumPrice Int64, ParamCurrency FixedString(3),
ParamCurrencyID UInt16, ClickLogID UInt64, ClickEventID Int32, ClickGoodEvent Int32, ClickEventTime DateTime,
ClickPriorityID Int32, ClickPhraseID Int32, ClickPageID Int32, ClickPlacelD Int32, ClickTypeID Int32, ClickResourceID Int32,
ClickCost UInt32, ClickClientIP UInt32, ClickDomainID UInt32, ClickURL String, ClickAttempt UInt8, ClickOrderID UInt32,
ClickBannerID UInt32, ClickMarketCategoryID UInt32, ClickMarketPP UInt32, ClickMarketCategoryName String,
ClickMarketPPName String, ClickAWAPSCampaignName String, ClickPageName String, ClickTargetType UInt16,
ClickTargetPhraseID UInt64, ClickContextType UInt8, ClickSelectType Int8, ClickOptions String, ClickGroupBannerID Int32,
OpenstatServiceName String, OpenstatCampaignID String, OpenstatAdID String, OpenstatSourceID String, UTMSource
String, UTMMedium String, UTMCampaign String, UTMContent String, UTMTerm String, FromTag String, HasGCLID UInt8,
FirstVisit DateTime, PredLastVisit Date, LastVisit Date, TotalVisits UInt32, TrafficSource Nested(ID Int8, SearchEngineID
UInt16, AdvEngineID UInt8, PlacelD UInt16, SocialSourceNetworkID UInt8, Domain String, SearchPhrase String,
SocialSourcePage String), Attendance FixedString(16), CLID UInt32, YCLID UInt64, NormalizedRefererHash UInt64,
SearchPhraseHash UInt64, RefererDomainHash UInt64, NormalizedStartURLHash UInt64, StartURLDomainHash UInt64,
NormalizedEndURLHash UInt64, TopLevelDomain UInt64, URLScheme UInt64, OpenstatServiceNameHash UInt64,
OpenstatCampaignIDHash UInt64, OpenstatAdIDHash UInt64, OpenstatSourceIDHash UInt64, UTMSourceHash UInt64,
UTMMediumHash UInt64, UTMCampaignHash UInt64, UTMContentHash UInt64, UTMTermHash UInt64, FromHash UInt64,
WebVisorEnabled UInt8, WebVisorActivity UInt32, ParsedParams Nested(Key1 String, Key2 String, Key3 String, Key4
String, Key5 String, ValueDouble Float64), Market Nested(Type UInt8, GoalID UInt32, OrderID String, OrderPrice Int64, PP
UInt32, DirectPlacelD UInt32, DirectOrderID UInt32, DirectBannerID UInt32, GoodID String, GoodName String,
GoodQuantity Int32, GoodPrice Int64), IslandID FixedString(16)) ENGINE = CollapsingMergeTree(StartDate,
intHash32(UserID), (CounterID, StartDate, intHash32(UserID), VisitID), 8192, Sign)"
## import data
cat visits_v1.tsv | clickhouse-client --query "INSERT INTO datasets.visits_v1 FORMAT TSV" --max_insert_block_size=100000
## optionally you can optimize table
clickhouse-client --query "OPTIMIZE TABLE datasets.visits_v1 FINAL"
clickhouse-client --query "SELECT COUNT(*) FROM datasets.visits_v1"

```

Queries

Examples of queries to these tables (they are named `test.hits` and `test.visits`) can be found among [stateful tests](#) and in some [performance tests](#) of ClickHouse.

Interfaces

ClickHouse provides two network interfaces (both can be optionally wrapped in TLS for additional security):

- [HTTP](#), which is documented and easy to use directly.
- [Native TCP](#), which has less overhead.

In most cases it is recommended to use appropriate tool or library instead of interacting with those directly. Officially supported by Yandex are the following:

- [Command-line client](#)
- [JDBC driver](#)
- [ODBC driver](#)

- [ODBC driver](#)

There are also a wide range of third-party libraries for working with ClickHouse:

- [Client libraries](#)
- [Integrations](#)
- [Visual interfaces](#)

Command-line Client

To work from the command line, you can use `clickhouse-client`:

```
$ clickhouse-client
ClickHouse client version 0.0.26176.
Connecting to localhost:9000.
Connected to ClickHouse server version 0.0.26176.

:)
```

The client supports command-line options and configuration files. For more information, see "[Configuring](#)".

Usage

The client can be used in interactive and non-interactive (batch) mode.

To use batch mode, specify the 'query' parameter, or send data to 'stdin' (it verifies that 'stdin' is not a terminal), or both.

Similar to the HTTP interface, when using the 'query' parameter and sending data to 'stdin', the request is a concatenation of the 'query' parameter, a line feed, and the data in 'stdin'. This is convenient for large INSERT queries.

Example of using the client to insert data:

```
echo -ne "1, 'some text', '2016-08-14 00:00:00'\n2, 'some more text', '2016-08-14 00:00:01'" | clickhouse-client --
database=test --query="INSERT INTO test FORMAT CSV";

cat <<_EOF | clickhouse-client --database=test --query="INSERT INTO test FORMAT CSV";
3, 'some text', '2016-08-14 00:00:00'
4, 'some more text', '2016-08-14 00:00:01'
_EOF

cat file.csv | clickhouse-client --database=test --query="INSERT INTO test FORMAT CSV";
```

In batch mode, the default data format is TabSeparated. You can set the format in the FORMAT clause of the query.

By default, you can only process a single query in batch mode. To make multiple queries from a "script," use the `--multiquery` parameter. This works for all queries except INSERT. Query results are output consecutively without additional separators.

Similarly, to process a large number of queries, you can run 'clickhouse-client' for each query. Note that it may take tens of milliseconds to launch the 'clickhouse-client' program.

In interactive mode, you get a command line where you can enter queries.

If 'multiline' is not specified (the default): To run the query, press Enter. The semicolon is not necessary at the end of the query. To enter a multiline query, enter a backslash `\` before the line feed. After you press Enter, you will be asked to enter the next line of the query.

If multiline is specified: To run a query, end it with a semicolon and press Enter. If the semicolon was omitted at the end of the entered line, you will be asked to enter the next line of the query.

Only a single query is run, so everything after the semicolon is ignored.

You can specify `\G` instead of or after the semicolon. This indicates Vertical format. In this format, each value is printed on a separate line, which is convenient for wide tables. This unusual feature was added for compatibility with the MySQL CLI.

The command line is based on 'readline' (and 'history' or 'libedit', or without a library, depending on the build). In other words, it uses the familiar keyboard shortcuts and keeps a history.

The history is written to `~/.clickhouse-client-history`.

By default, the format used is PrettyCompact. You can change the format in the FORMAT clause of the query, or by specifying `\G` at the end of the query, using the `--format` or `--vertical` argument in the command line, or using the client configuration file.

To exit the client, press Ctrl+D (or Ctrl+C), or enter one of the following instead of a query: "exit", "quit", "logout", "exit;", "quit;", "logout;", "q", "Q", ":", "q"

When processing a query, the client shows:

1. Progress, which is updated no more than 10 times per second (by default). For quick queries, the progress might not have time to be displayed.
2. The formatted query after parsing, for debugging.
3. The result in the specified format.
4. The number of lines in the result, the time passed, and the average speed of query processing.

You can cancel a long query by pressing Ctrl+C. However, you will still need to wait a little for the server to abort the request. It is not possible to cancel a query at certain stages. If you don't wait and press Ctrl+C a second time, the client will exit.

The command-line client allows passing external data (external temporary tables) for querying. For more information, see the section "External data for query processing".

Configuring

You can pass parameters to `clickhouse-client` (all parameters have a default value) using:

- From the Command Line

Command-line options override the default values and settings in configuration files.

- Configuration files.

Settings in the configuration files override the default values.

Command Line Options

- `--host, -h` -- The server name, 'localhost' by default. You can use either the name or the IPv4 or IPv6 address.
- `--port` -- The port to connect to. Default value: 9000. Note that the HTTP interface and the native interface use different ports.
- `--user, -u` -- The username. Default value: default.
- `--password` -- The password. Default value: empty string.
- `--query, -q` -- The query to process when using non-interactive mode.
- `--database, -d` -- Select the current default database. Default value: the current database from the server settings ('default' by default).
- `--multiline, -m` -- If specified, allow multiline queries (do not send the query on Enter).
- `--multiquery, -n` -- If specified, allow processing multiple queries separated by commas. Only works in non-interactive mode.
- `--format, -f` -- Use the specified default format to output the result.
- `--vertical, -E` -- If specified, use the Vertical format by default to output the result. This is the same as '`--format=Vertical`'. In this format, each value is printed on a separate line, which is helpful when displaying wide tables.
- `--time` -- If specified, print the query execution time to stdout in non-interactive mode.

- `--time, -t` – If specified, print the query execution time to `stderr` in non-interactive mode.
- `--stacktrace` – If specified, also print the stack trace if an exception occurs.
- `--config-file` – The name of the configuration file.
- `--secure` – If specified, will connect to server over secure connection.

Configuration Files

`clickhouse-client` uses the first existing file of the following:

- Defined in the `--config-file` parameter.
- `./clickhouse-client.xml`
- `~/.clickhouse-client/config.xml`
- `/etc/clickhouse-client/config.xml`

Example of a config file:

```
<config>
  <user>username</user>
  <password>password</password>
  <secure>False</secure>
</config>
```

Native Interface (TCP)

The native protocol is used in the [command-line client](#), for interserver communication during distributed query processing, and also in other C++ programs. Unfortunately, native ClickHouse protocol does not have formal specification yet, but it can be reverse engineered from ClickHouse source code (starting [around here](#)) and/or by intercepting and analyzing TCP traffic.

HTTP Interface

The HTTP interface lets you use ClickHouse on any platform from any programming language. We use it for working from Java and Perl, as well as shell scripts. In other departments, the HTTP interface is used from Perl, Python, and Go. The HTTP interface is more limited than the native interface, but it has better compatibility.

By default, `clickhouse-server` listens for HTTP on port 8123 (this can be changed in the config).

If you make a GET / request without parameters, it returns the string "Ok" (with a line feed at the end). You can use this in health-check scripts.

```
$ curl 'http://localhost:8123/'
Ok.
```

Send the request as a URL 'query' parameter, or as a POST. Or send the beginning of the query in the 'query' parameter, and the rest in the POST (we'll explain later why this is necessary). The size of the URL is limited to 16 KB, so keep this in mind when sending large queries.

If successful, you receive the 200 response code and the result in the response body.

If an error occurs, you receive the 500 response code and an error description text in the response body.

When using the GET method, 'readonly' is set. In other words, for queries that modify data, you can only use the POST method. You can send the query itself either in the POST body, or in the URL parameter.

Examples:

```
$ curl 'http://localhost:8123/?query=SELECT%201'
1

$ wget -O- -q 'http://localhost:8123/?query=SELECT 1'
1
```

```
$ echo -ne 'GET /?query=SELECT%201 HTTP/1.0\r\n\r\n' | nc localhost 8123
HTTP/1.0 200 OK
Connection: Close
Date: Fri, 16 Nov 2012 19:21:50 GMT
```

```
1
```

As you can see, curl is somewhat inconvenient in that spaces must be URL escaped.

Although wget escapes everything itself, we don't recommend using it because it doesn't work well over HTTP 1.1 when using keep-alive and Transfer-Encoding: chunked.

```
$ echo 'SELECT 1' | curl 'http://localhost:8123/' --data-binary @-
1

$ echo 'SELECT 1' | curl 'http://localhost:8123/?query=' --data-binary @-
1

$ echo '1' | curl 'http://localhost:8123/?query=SELECT' --data-binary @-
1
```

If part of the query is sent in the parameter, and part in the POST, a line feed is inserted between these two data parts.

Example (this won't work):

```
$ echo 'ECT 1' | curl 'http://localhost:8123/?query=SEL' --data-binary @-
Code: 59, e.displayText() = DB::Exception: Syntax error: failed at position 0: SEL
ECT 1
, expected One of: SHOW TABLES, SHOW DATABASES, SELECT, INSERT, CREATE, ATTACH, RENAME, DROP, DETACH, USE,
SET, OPTIMIZE., e.what() = DB::Exception
```

By default, data is returned in TabSeparated format (for more information, see the "Formats" section).

You use the FORMAT clause of the query to request any other format.

```
$ echo 'SELECT 1 FORMAT Pretty' | curl 'http://localhost:8123/?' --data-binary @-
```

```
1
1
```

The POST method of transmitting data is necessary for INSERT queries. In this case, you can write the beginning of the query in the URL parameter, and use POST to pass the data to insert. The data to insert could be, for example, a tab-separated dump from MySQL. In this way, the INSERT query replaces LOAD DATA LOCAL INFILE from MySQL.

Examples: Creating a table:

```
echo 'CREATE TABLE t (a UInt8) ENGINE = Memory' | curl 'http://localhost:8123/' --data-binary @-
```

Using the familiar INSERT query for data insertion:

```
echo 'INSERT INTO t VALUES (1),(2),(3)' | curl 'http://localhost:8123/' --data-binary @-
```

Data can be sent separately from the query:


```
echo '(4),(5),(6)' | curl 'http://localhost:8123/?query=INSERT%20INTO%20t%20VALUES' --data-binary @-
```

You can specify any data format. The 'Values' format is the same as what is used when writing INSERT INTO t VALUES:

```
echo '(7),(8),(9)' | curl 'http://localhost:8123/?query=INSERT%20INTO%20t%20FORMAT%20Values' --data-binary @-
```

To insert data from a tab-separated dump, specify the corresponding format:

```
echo -ne '10\n11\n12\n' | curl 'http://localhost:8123/?query=INSERT%20INTO%20t%20FORMAT%20TabSeparated' --data-binary @-
```

Reading the table contents. Data is output in random order due to parallel query processing:

```
$ curl 'http://localhost:8123/?query=SELECT%20a%20FROM%20t'
7
8
9
10
11
12
1
2
3
4
5
6
```

Deleting the table.

```
echo 'DROP TABLE t' | curl 'http://localhost:8123/' --data-binary @-
```

For successful requests that don't return a data table, an empty response body is returned.

You can use the internal ClickHouse compression format when transmitting data. The compressed data has a non-standard format, and you will need to use the special `clickhouse-compressor` program to work with it (it is installed with the `clickhouse-client` package). To increase the efficiency of data insertion, you can disable server-side checksum verification by using the `http_native_compression_disable_checksumming_on_decompress` setting.

If you specified `compress=1` in the URL, the server compresses the data it sends you.

If you specified `decompress=1` in the URL, the server decompresses the same data that you pass in the POST method.

You can also choose to use **HTTP compression**. To send a POST request compressed, append the request header `Content-Encoding: compression_method`. In order for ClickHouse to compress the response, you must append `Accept-Encoding: compression_method`. ClickHouse supports `gzip`, `br`, `deflate` [https://en.wikipedia.org/wiki/HTTP_compression#Content-Encoding_tokens]. To enable HTTP compression, you must use the ClickHouse `enable_http_compression` setting. You can configure the compression level of the data with the `http_zlib_compression_level` setting for all the compression methods.

You can use this to reduce network traffic when transmitting a large amount of data, or for creating dumps that are immediately compressed.

Examples of sending the data with compression:

```
##Sending the data to the server:
```

```
##Sending the data to the server:
```

```
curl -vsS "http://localhost:8123/?enable_http_compression=1" -d 'SELECT number FROM system.numbers LIMIT 10' -H 'Accept-Encoding: gzip'
```

```
##Sending the data to the client:
```

```
echo "SELECT 1" | gzip -c | curl -sS --data-binary @- -H 'Content-Encoding: gzip' 'http://localhost:8123/'
```

Note

Some HTTP clients might decompress data from the server by default (with `gzip` and `deflate`) and you might get decompressed data even if you use the compression settings correctly.

You can use the 'database' URL parameter to specify the default database.

```
$ echo 'SELECT number FROM numbers LIMIT 10' | curl 'http://localhost:8123/?database=system' --data-binary @-
0
1
2
3
4
5
6
7
8
9
```

By default, the database that is registered in the server settings is used as the default database. By default, this is the database called 'default'. Alternatively, you can always specify the database using a dot before the table name.

The username and password can be indicated in one of two ways:

1. Using HTTP Basic Authentication. Example:

```
echo 'SELECT 1' | curl 'http://user:password@localhost:8123/' -d @-
```

2. In the 'user' and 'password' URL parameters. Example:

```
echo 'SELECT 1' | curl 'http://localhost:8123/?user=user&password=password' -d @-
```

If the user name is not specified, the `default` name is used. If the password is not specified, the empty password is used.

You can also use the URL parameters to specify any settings for processing a single query, or entire profiles of settings. Example: http://localhost:8123/?profile=web&max_rows_to_read=1000000000&query=SELECT+1

For more information, see the [Settings][../operations/settings/index.md] section.

```
$ echo 'SELECT number FROM system.numbers LIMIT 10' | curl 'http://localhost:8123/?' --data-binary @-
0
1
2
3
4
5
6
7
8
9
```

For information about other parameters, see the section "SET".

Similarly, you can use ClickHouse sessions in the HTTP protocol. To do this, you need to add the `session_id` GET parameter to the request. You can use any string as the session ID. By default, the session is terminated after 60 seconds of inactivity. To change this timeout, modify the `default_session_timeout` setting in the server configuration, or add the `session_timeout` GET parameter to the request. To check the session status, use the `session_check=1` parameter. Only one query at a time can be executed within a single session.

You have the option to receive information about the progress of query execution in X-ClickHouse-Progress headers. To do this, enable the setting `send_progress_in_http_headers`.

Running requests don't stop automatically if the HTTP connection is lost. Parsing and data formatting are performed on the server side, and using the network might be ineffective. The optional 'query_id' parameter can be passed as the query ID (any string). For more information, see the section "Settings, replace_running_query".

The optional 'quota_key' parameter can be passed as the quota key (any string). For more information, see the section "Quotas".

The HTTP interface allows passing external data (external temporary tables) for querying. For more information, see the section "External data for query processing".

Response Buffering

You can enable response buffering on the server side. The `buffer_size` and `wait_end_of_query` URL parameters are provided for this purpose.

`buffer_size` determines the number of bytes in the result to buffer in the server memory. If the result body is larger than this threshold, the buffer is written to the HTTP channel, and the remaining data is sent directly to the HTTP channel.

To ensure that the entire response is buffered, set `wait_end_of_query=1`. In this case, the data that is not stored in memory will be buffered in a temporary server file.

Example:

```
curl -sS 'http://localhost:8123/?max_result_bytes=4000000&buffer_size=3000000&wait_end_of_query=1' -d 'SELECT toUInt8(number) FROM system.numbers LIMIT 9000000 FORMAT RowBinary'
```

Use buffering to avoid situations where a query processing error occurred after the response code and HTTP headers were sent to the client. In this situation, an error message is written at the end of the response body, and on the client side, the error can only be detected at the parsing stage.

Formats for input and output data

ClickHouse can accept (`INSERT`) and return (`SELECT`) data in various formats.

The table below lists supported formats and how they can be used in `INSERT` and `SELECT` queries.

Format	INSERT	SELECT
TabSeparated	✓	✓
TabSeparatedRaw	✗	✓
TabSeparatedWithNames	✓	✓
TabSeparatedWithNamesAndTypes	✓	✓

CSV Format	✓ INSERT	✓ SELECT
CSVWithNames	✓	✓
Values	✓	✓
Vertical	X	✓
JSON	X	✓
JSONCompact	X	✓
JSONEachRow	✓	✓
TSKV	✓	✓
Pretty	X	✓
PrettyCompact	X	✓
PrettyCompactMonoBlock	X	✓
PrettyNoEscapes	X	✓
PrettySpace	X	✓
Protobuf	✓	✓
Parquet	✓	✓
RowBinary	✓	✓
Native	✓	✓
Null	X	✓
XML	X	✓
CapnProto	✓	X

You can control some format processing parameters by the ClickHouse settings. For more information read the [Settings](#) section.

TabSeparated

In TabSeparated format, data is written by row. Each row contains values separated by tabs. Each value is followed by a tab, except the last value in the row, which is followed by a line feed. Strictly Unix line feeds are assumed everywhere. The last row also must contain a line feed at the end. Values are written in text format, without enclosing quotation marks, and with special characters escaped.

This format is also available under the name `TSV`.

The `TabSeparated` format is convenient for processing data using custom programs and scripts. It is used by default in the HTTP interface, and in the command-line client's batch mode. This format also allows transferring data between different DBMSs. For example, you can get a dump from MySQL and upload it to ClickHouse, or vice versa.

The `TabSeparated` format supports outputting total values (when using `WITH TOTALS`) and extreme values (when 'extremes' is set to 1). In these cases, the total values and extremes are output after the main data. The main result, total values, and extremes are separated from each other by an empty line. Example:

```
SELECT EventDate, count() AS c FROM test.hits GROUP BY EventDate WITH TOTALS ORDER BY EventDate FORMAT TabSeparated``
```

2014-03-17	1406958
2014-03-18	1383658
2014-03-19	1405797
2014-03-20	1353623
2014-03-21	1245779
2014-03-22	1031592
2014-03-23	1046491
0000-00-00	8873898
2014-03-17	1031592
2014-03-23	1406958

Data formatting

Integer numbers are written in decimal form. Numbers can contain an extra "+" character at the beginning (ignored when parsing, and not recorded when formatting). Non-negative numbers can't contain the negative sign. When reading, it is allowed to parse an empty string as a zero, or (for signed types) a string consisting of just a minus sign as a zero. Numbers that do not fit into the corresponding data type may be parsed as a different number, without an error message.

Floating-point numbers are written in decimal form. The dot is used as the decimal separator. Exponential entries are supported, as are 'inf', '+inf', '-inf', and 'nan'. An entry of floating-point numbers may begin or end with a decimal point.

During formatting, accuracy may be lost on floating-point numbers.

During parsing, it is not strictly required to read the nearest machine-representable number.

Dates are written in `YYYY-MM-DD` format and parsed in the same format, but with any characters as separators. Dates with times are written in the format `YYYY-MM-DD hh:mm:ss` and parsed in the same format, but with any characters as separators.

This all occurs in the system time zone at the time the client or server starts (depending on which one formats data). For dates with times, daylight saving time is not specified. So if a dump has times during daylight saving time, the dump does not unequivocally match the data, and parsing will select one of the two times.

During a read operation, incorrect dates and dates with times can be parsed with natural overflow or as null dates and times, without an error message.

As an exception, parsing dates with times is also supported in Unix timestamp format, if it consists of exactly 10 decimal digits. The result is not time zone-dependent. The formats `YYYY-MM-DD hh:mm:ss` and `NNNNNNNNNN` are differentiated automatically.

Strings are output with backslash-escaped special characters. The following escape sequences are used for output: `\b`, `\f`, `\r`, `\n`, `\t`, `\0`, `\'`, `\"`. Parsing also supports the sequences `\a`, `\v`, and `\xHH` (hex escape sequences) and any `\c` sequences, where `c` is any character (these sequences are converted to `c`). Thus, reading data supports formats where a line feed can be written as `\n` or `\`, or as a line feed. For example, the string `Hello world` with a line feed between the words instead of a space can be parsed in any of the following variations:

```
Hello\nworld
```

```
Hello\  
world
```

The second variant is supported because MySQL uses it when writing tab-separated dumps.

The minimum set of characters that you need to escape when passing data in TabSeparated format: tab, line feed (LF) and backslash.

Only a small set of symbols are escaped. You can easily stumble onto a string value that your terminal will ruin in output.

Arrays are written as a list of comma-separated values in square brackets. Number items in the array are formatted as normally, but dates, dates with times, and strings are written in single quotes with the same escaping rules as above.

NULL is formatted as \N.

TabSeparatedRaw

Differs from `TabSeparated` format in that the rows are written without escaping.

This format is only appropriate for outputting a query result, but not for parsing (retrieving data to insert in a table).

This format is also available under the name `TSVRaw`.

TabSeparatedWithNames

Differs from the `TabSeparated` format in that the column names are written in the first row.

During parsing, the first row is completely ignored. You can't use column names to determine their position or to check their correctness.

(Support for parsing the header row may be added in the future.)

This format is also available under the name `TSVWithNames`.

TabSeparatedWithNamesAndTypes

Differs from the `TabSeparated` format in that the column names are written to the first row, while the column types are in the second row.

During parsing, the first and second rows are completely ignored.

This format is also available under the name `TSVWithNamesAndTypes`.

TSKV

Similar to `TabSeparated`, but outputs a value in name=value format. Names are escaped the same way as in `TabSeparated` format, and the = symbol is also escaped.

```
SearchPhrase= count()=8267016
SearchPhrase=bathroom interior design count()=2166
SearchPhrase=yandex count()=1655
SearchPhrase=2014 spring fashion count()=1549
SearchPhrase=freeform photos count()=1480
SearchPhrase=angelina jolie count()=1245
SearchPhrase=omsk count()=1112
SearchPhrase=photos of dog breeds count()=1091
SearchPhrase=curtain designs count()=1064
SearchPhrase=baku count()=1000
```

NULL is formatted as \N.

```
SELECT * FROM t_null FORMAT TSKV
```

```
x=1 y=\N
```

When there is a large number of small columns, this format is ineffective, and there is generally no reason to use it. It is used in some departments of Yandex.

Both data output and parsing are supported in this format. For parsing, any order is supported for the values of different columns. It is acceptable for some values to be omitted – they are treated as equal to their default values. In this case, zeros and blank rows are used as default values. Complex values that could be specified in the table are not supported as defaults.

Parsing allows the presence of the additional field `tskv` without the equal sign or a value. This field is ignored.

CSV

Comma Separated Values format (RFC).

When formatting, rows are enclosed in double quotes. A double quote inside a string is output as two double quotes in a row. There are no other rules for escaping characters. Date and date-time are enclosed in double quotes. Numbers are output without quotes. Values are separated by a delimiter character, which is `,` by default. The delimiter character is defined in the setting `format_csv_delimiter`. Rows are separated using the Unix line feed (LF). Arrays are serialized in CSV as follows: first the array is serialized to a string as in `TabSeparated` format, and then the resulting string is output to CSV in double quotes. Tuples in CSV format are serialized as separate columns (that is, their nesting in the tuple is lost).

```
clickhouse-client --format_csv_delimiter="|" --query="INSERT INTO test.csv FORMAT CSV" < data.csv
```

By default, the delimiter is `,`. See the `format_csv_delimiter` setting for more information.

When parsing, all values can be parsed either with or without quotes. Both double and single quotes are supported. Rows can also be arranged without quotes. In this case, they are parsed up to the delimiter character or line feed (CR or LF). In violation of the RFC, when parsing rows without quotes, the leading and trailing spaces and tabs are ignored. For the line feed, Unix (LF), Windows (CR LF) and Mac OS Classic (CR LF) types are all supported.

`NULL` is formatted as `\N`.

The CSV format supports the output of totals and extremes the same way as `TabSeparated`.

CSVWithNames

Also prints the header row, similar to `TabSeparatedWithNames`.

JSON

Outputs data in JSON format. Besides data tables, it also outputs column names and types, along with some additional information: the total number of output rows, and the number of rows that could have been output if there weren't a LIMIT. Example:


```
SELECT SearchPhrase, count() AS c FROM test.hits GROUP BY SearchPhrase WITH TOTALS ORDER BY c DESC LIMIT 5  
FORMAT JSON
```

```
{  
  "meta":  
  [  
    {  
      "name": "SearchPhrase",  
      "type": "String"  
    },  
    {  
      "name": "c",  
      "type": "UInt64"  
    }  
  ]  
}
```

```

        "type": "UInt64"
    }
],
"data":
[
    {
        "SearchPhrase": "",
        "c": "8267016"
    },
    {
        "SearchPhrase": "bathroom interior design",
        "c": "2166"
    },
    {
        "SearchPhrase": "yandex",
        "c": "1655"
    },
    {
        "SearchPhrase": "spring 2014 fashion",
        "c": "1549"
    },
    {
        "SearchPhrase": "freeform photos",
        "c": "1480"
    }
],
"totals":
{
    "SearchPhrase": "",
    "c": "8873898"
},
"extremes":
{
    "min":
    {
        "SearchPhrase": "",
        "c": "1480"
    },
    "max":
    {
        "SearchPhrase": "",
        "c": "8267016"
    }
},
"rows": 5,
"rows_before_limit_at_least": 141137
}

```

The JSON is compatible with JavaScript. To ensure this, some characters are additionally escaped: the slash / is escaped as \; alternative line breaks U+2028 and U+2029, which break some browsers, are escaped as \uXXXX. ASCII control characters are escaped: backspace, form feed, line feed, carriage return, and horizontal tab are replaced with \b, \f, \n, \r, \t, as well as the remaining bytes in the 00-1F range using \uXXXX sequences. Invalid UTF-8 sequences are changed to the replacement character  so the output text will consist of valid UTF-8 sequences. For compatibility with JavaScript, Int64 and UInt64 integers are enclosed in double quotes by default. To remove the quotes, you can set the configuration parameter `output_format_json_quote_64bit_integers` to 0.

`rows` – The total number of output rows.

`rows` - The total number of output rows.

`rows_before_limit_at_least` The minimal number of rows there would have been without LIMIT. Output only if the query contains LIMIT.

If the query contains GROUP BY, `rows_before_limit_at_least` is the exact number of rows there would have been without a LIMIT.

`totals` - Total values (when using WITH TOTALS).

`extremes` - Extreme values (when extremes is set to 1).

This format is only appropriate for outputting a query result, but not for parsing (retrieving data to insert in a table).

ClickHouse supports **NULL**, which is displayed as `null` in the JSON output.

See also the **JSONEachRow** format.

JSONCompact

Differs from JSON only in that data rows are output in arrays, not in objects.

Example:

```
{
  "meta":
  [
    {
      "name": "SearchPhrase",
      "type": "String"
    },
    {
      "name": "c",
      "type": "UInt64"
    }
  ],
  "data":
  [
    ["", "8267016"],
    ["bathroom interior design", "2166"],
    ["yandex", "1655"],
    ["fashion trends spring 2014", "1549"],
    ["freeform photo", "1480"]
  ],
  "totals": ["", "8873898"],
  "extremes":
  {
    "min": ["", "1480"],
    "max": ["", "8267016"]
  },
  "rows": 5,
  "rows_before_limit_at_least": 141137
}
```

This format is only appropriate for outputting a query result, but not for parsing (retrieving data to insert in a table).

See also the `JSONEachRow` format.

JSONEachRow

JSONEACHROW

When using this format, ClickHouse outputs rows as separated, newline-delimited JSON objects, but the data as a whole is not valid JSON.

```
{"SearchPhrase":"curtain designs","count()":"1064"}
{"SearchPhrase":"baku","count()":"1000"}
{"SearchPhrase":"","count()":"8267016"}
```

When inserting the data, you should provide a separate JSON object for each row.

Inserting Data

```
INSERT INTO UserActivity FORMAT JSONEachRow {"PageViews":5, "UserID":"4324182021466249494",
"Duration":146,"Sign":-1} {"UserID":"4324182021466249494","PageViews":6,"Duration":185,"Sign":1}
```

ClickHouse allows:

- Any order of key-value pairs in the object.
- Omitting some values.

ClickHouse ignores spaces between elements and commas after the objects. You can pass all the objects in one line. You don't have to separate them with line breaks.

Omitted values processing

ClickHouse substitutes omitted values with the default values for the corresponding **data types**.

If `DEFAULT expr` is specified, ClickHouse uses different substitution rules depending on the `input_format_defaults_for_omitted_fields` setting.

Consider the following table:

```
CREATE TABLE IF NOT EXISTS example_table
(
  x UInt32,
  a DEFAULT x * 2
) ENGINE = Memory;
```

- If `input_format_defaults_for_omitted_fields = 0`, then the default value for `x` and `a` equals `0` (as the default value for the `UInt32` data type).
- If `input_format_defaults_for_omitted_fields = 1`, then the default value for `x` equals `0`, but the default value of `a` equals `x * 2`.

Warning

When inserting data with `insert_sample_with_metadata = 1`, ClickHouse consumes more computational resources, compared to insertion with `insert_sample_with_metadata = 0`.

Selecting Data

Consider the `UserActivity` table as an example:

UserID	PageViews	Duration	Sign
4324182021466249494	5	146	-1
4324182021466249494	6	185	1

The query `SELECT * FROM UserActivity FORMAT JSONEachRow` returns:

```
{"UserID":"4324182021466249494","PageViews":5,"Duration":146,"Sign":-1}
{"UserID":"4324182021466249494","PageViews":6,"Duration":185,"Sign":1}
```

Unlike the **JSON** format, there is no substitution of invalid UTF-8 sequences. Values are escaped in the same way as for **JSON**.

Note

Any set of bytes can be output in the strings. Use the **JSONEachRow** format if you are sure that the data in the table can be formatted as JSON without losing any information.

Native

The most efficient format. Data is written and read by blocks in binary format. For each block, the number of rows, number of columns, column names and types, and parts of columns in this block are recorded one after another. In other words, this format is "columnar" – it doesn't convert columns to rows. This is the format used in the native interface for interaction between servers, for using the command-line client, and for C++ clients.

You can use this format to quickly generate dumps that can only be read by the ClickHouse DBMS. It doesn't make sense to work with this format yourself.

Null

Nothing is output. However, the query is processed, and when using the command-line client, data is transmitted to the client. This is used for tests, including productivity testing. Obviously, this format is only appropriate for output, not for parsing.

Pretty

Outputs data as Unicode-art tables, also using ANSI-escape sequences for setting colors in the terminal. A full grid of the table is drawn, and each row occupies two lines in the terminal.

Each result block is output as a separate table. This is necessary so that blocks can be output without buffering results (buffering would be necessary in order to pre-calculate the visible width of all the values).

NULL is output as `NULL`.

Example (shown for the **PrettyCompact** format):

```
SELECT * FROM t_null
```

x	y
1	NULL

Rows are not escaped in Pretty* formats. Example is shown for the **PrettyCompact** format:

```
SELECT 'String with \'quotes\' and \t character' AS Escaping_test
```

Escaping_test
String with 'quotes' and character

To avoid dumping too much data to the terminal, only the first 10,000 rows are printed. If the number of rows is greater than or equal to 10,000, the message "Showed first 10 000" is printed.

This format is only appropriate for outputting a query result, but not for parsing (retrieving data to insert in a table).

The Pretty format supports outputting total values (when using WITH TOTALS) and extremes (when 'extremes' is set to 1). In these cases, total values and extreme values are output after the main data, in separate tables. Example (shown for the **PrettyCompact** format):

```
SELECT EventDate, count() AS c FROM test.hits GROUP BY EventDate WITH TOTALS ORDER BY EventDate FORMAT PrettyCompact
```

EventDate	c
2014-03-17	1406958
2014-03-18	1383658
2014-03-19	1405797
2014-03-20	1353623
2014-03-21	1245779
2014-03-22	1031592
2014-03-23	1046491

Totals:

EventDate	c
0000-00-00	8873898

Extremes:

EventDate	c
2014-03-17	1031592
2014-03-23	1406958

PrettyCompact

Differs from **Pretty** in that the grid is drawn between rows and the result is more compact. This format is used by default in the command-line client in interactive mode.

PrettyCompactMonoBlock

Differs from **PrettyCompact** in that up to 10,000 rows are buffered, then output as a single table, not by blocks.

PrettyNoEscapes

Differs from Pretty in that ANSI-escape sequences aren't used. This is necessary for displaying this format in a browser, as well as for using the 'watch' command-line utility.

Example:

```
watch -n1 "clickhouse-client --query='SELECT event, value FROM system.events FORMAT PrettyCompactNoEscapes'"
```

You can use the HTTP interface for displaying in the browser.

PrettyCompactNoEscapes

The same as the previous setting.

PrettySpaceNoEscapes

The same as the previous setting.

PrettySpace

Differs from **PrettyCompact** in that whitespace (space characters) is used instead of the grid.

RowBinary

RowBinary

Formats and parses data by row in binary format. Rows and values are listed consecutively, without separators. This format is less efficient than the Native format, since it is row-based.

Integers use fixed-length little endian representation. For example, UInt64 uses 8 bytes.

DateTime is represented as UInt32 containing the Unix timestamp as the value.

Date is represented as a UInt16 object that contains the number of days since 1970-01-01 as the value.

String is represented as a varint length (unsigned **LEB128**), followed by the bytes of the string.

FixedString is represented simply as a sequence of bytes.

Array is represented as a varint length (unsigned **LEB128**), followed by successive elements of the array.

For **NULL** support, an additional byte containing 1 or 0 is added before each **Nullable** value. If 1, then the value is **NULL** and this byte is interpreted as a separate value. If 0, the value after the byte is not **NULL**.

RowBinaryWithNamesAndTypes

Similar to **RowBinary**, but with added header: * **LEB128**-encoded number of columns (N) * N **Strings** specifying column names * N **Strings** specifying column types

Values

Prints every row in brackets. Rows are separated by commas. There is no comma after the last row. The values inside the brackets are also comma-separated. Numbers are output in decimal format without quotes. Arrays are output in square brackets. Strings, dates, and dates with times are output in quotes. Escaping rules and parsing are similar to the **TabSeparated** format. During formatting, extra spaces aren't inserted, but during parsing, they are allowed and skipped (except for spaces inside array values, which are not allowed). **NULL** is represented as **NULL**.

The minimum set of characters that you need to escape when passing data in Values format: single quotes and backslashes.

This is the format that is used in **INSERT INTO t VALUES ...**, but you can also use it for formatting query results.

Vertical

Prints each value on a separate line with the column name specified. This format is convenient for printing just one or a few rows, if each row consists of a large number of columns.

NULL is output as **NULL**.

Example:

```
SELECT * FROM t_null FORMAT Vertical
```

Row 1:

x: 1
y: NULL

Rows are not escaped in Vertical format:

```
SELECT 'string with \'quotes\' and \t with some special \n characters' AS test FORMAT Vertical
```

Row 1:

test: string with 'quotes' and with some special
characters

This format is only appropriate for outputting a query result, but not for parsing (retrieving data to insert in a table).

XML

XML format is suitable only for output, not for parsing. Example:

```
<?xml version='1.0' encoding='UTF-8' ?>
<result>
  <meta>
    <columns>
      <column>
        <name>SearchPhrase</name>
        <type>String</type>
      </column>
      <column>
        <name>count()</name>
        <type>UInt64</type>
      </column>
    </columns>
  </meta>
  <data>
    <row>
      <SearchPhrase></SearchPhrase>
      <field>8267016</field>
    </row>
    <row>
      <SearchPhrase>bathroom interior design</SearchPhrase>
      <field>2166</field>
    </row>
    <row>
      <SearchPhrase>yandex</SearchPhrase>
      <field>1655</field>
    </row>
    <row>
      <SearchPhrase>2014 spring fashion</SearchPhrase>
      <field>1549</field>
    </row>
    <row>
      <SearchPhrase>freeform photos</SearchPhrase>
      <field>1480</field>
    </row>
    <row>
      <SearchPhrase>angelina jolie</SearchPhrase>
      <field>1245</field>
    </row>
    <row>
      <SearchPhrase>omsk</SearchPhrase>
      <field>1112</field>
    </row>
    <row>
      <SearchPhrase>photos of dog breeds</SearchPhrase>
      <field>1091</field>
    </row>
    <row>
      <SearchPhrase>curtain designs</SearchPhrase>
      <field>1064</field>
    </row>
    <row>
      <SearchPhrase>baku</SearchPhrase>
      <field>1000</field>
    </row>
  </data>
</result>
```

```
</data>
<rows>10</rows>
<rows_before_limit_at_least>141137</rows_before_limit_at_least>
</result>
```

If the column name does not have an acceptable format, just 'field' is used as the element name. In general, the XML structure follows the JSON structure.

Just as for JSON, invalid UTF-8 sequences are changed to the replacement character `�` so the output text will consist of valid UTF-8 sequences.

In string values, the characters `<` and `&` are escaped as `<` and `&`.

Arrays are output as `<array><elem>Hello</elem><elem>World</elem>...</array>`, and tuples as `<tuple><elem>Hello</elem><elem>World</elem>...</tuple>`.

CapnProto

Cap'n Proto is a binary message format similar to Protocol Buffers and Thrift, but not like JSON or MessagePack.

Cap'n Proto messages are strictly typed and not self-describing, meaning they need an external schema description. The schema is applied on the fly and cached for each query.

```
cat capnproto_messages.bin | clickhouse-client --query "INSERT INTO test.hits FORMAT CapnProto SETTINGS
format_schema='schema:Message'"
```

Where `schema.capnp` looks like this:

```
struct Message {
    SearchPhrase @0 :Text;
    c @1 :UInt64;
}
```

Deserialization is effective and usually doesn't increase the system load.

See also [Format Schema](#).

Protobuf

Protobuf - is a [Protocol Buffers](#) format.

This format requires an external format schema. The schema is cached between queries.

ClickHouse supports both `proto2` and `proto3` syntaxes. Repeated/optional/required fields are supported.

Usage examples:

```
SELECT * FROM test.table FORMAT Protobuf SETTINGS format_schema = 'schemafile:MessageType'
```

```
cat protobuf_messages.bin | clickhouse-client --query "INSERT INTO test.table FORMAT Protobuf SETTINGS
format_schema='schemafile:MessageType'"
```

where the file `schemafile.proto` looks like this:

```
syntax = "proto3";

message MessageType {
    string name = 1;
    string surname = 2;
    uint32 birthDate = 3;
```

```
repeated string phoneNumbers = 4;
};
```

To find the correspondence between table columns and fields of Protocol Buffers' message type ClickHouse compares their names.

This comparison is case-insensitive and the characters `_` (underscore) and `.` (dot) are considered as equal. If types of a column and a field of Protocol Buffers' message are different the necessary conversion is applied.

Nested messages are supported. For example, for the field `z` in the following message type

```
message MessageType {
  message XType {
    message YType {
      int32 z;
    };
    repeated YType y;
  };
  XType x;
};
```

ClickHouse tries to find a column named `x.y.z` (or `x_y_z` or `X.y_Z` and so on).

Nested messages are suitable to input or output a [nested data structures](#).

Default values defined in a protobuf schema like this

```
syntax = "proto2";

message MessageType {
  optional int32 result_per_page = 3 [default = 10];
}
```

are not applied; the [table defaults](#) are used instead of them.

ClickHouse inputs and outputs protobuf messages in the `length-delimited` format.

It means before every message should be written its length as a [varint](#).

See also [how to read/write length-delimited protobuf messages in popular languages](#).

Parquet

[Apache Parquet](#) is a columnar storage format available to any project in the Hadoop ecosystem. ClickHouse supports read and write operations for this format.

Data Types Matching

The table below shows supported data types and how they match ClickHouse [data types](#) in `INSERT` and `SELECT` queries.

Parquet data type (<code>INSERT</code>)	ClickHouse data type	Parquet data type (<code>SELECT</code>)
UINT8, BOOL	UInt8	UINT8
INT8	Int8	INT8
UINT16	UInt16	UINT16
INT16	Int16	INT16
UINT32	UInt32	UINT32

Parquet data type (INSERT)	ClickHouse data type	Parquet data type (SELECT)
UINT64	UInt64	UINT64
INT64	Int64	INT64
FLOAT, HALF_FLOAT	Float32	FLOAT
DOUBLE	Float64	DOUBLE
DATE32	Date	UINT16
DATE64, TIMESTAMP	DateTime	UINT32
STRING, BINARY	String	STRING
—	FixedString	STRING
DECIMAL	Decimal	DECIMAL

ClickHouse supports configurable precision of `Decimal` type. The `INSERT` query treats the Parquet `DECIMAL` type as the ClickHouse `Decimal128` type.

Unsupported Parquet data types: `DATE32`, `TIME32`, `FIXED_SIZE_BINARY`, `JSON`, `UUID`, `ENUM`.

Data types of a ClickHouse table columns can differ from the corresponding fields of the Parquet data inserted. When inserting data, ClickHouse interprets data types according to the table above and then **cast** the data to that data type which is set for the ClickHouse table column.

Inserting and Selecting Data

You can insert Parquet data from a file into ClickHouse table by the following command:

```
cat {filename} | clickhouse-client --query="INSERT INTO {some_table} FORMAT Parquet"
```

You can select data from a ClickHouse table and save them into some file in the Parquet format by the following command:

```
clickhouse-client --query="SELECT * FROM {some_table} FORMAT Parquet" > {some_file.pq}
```

Also look at the `HDFS` and `URL` storage engines to process data from the remote servers.

Format Schema

The file name containing the format schema is set by the setting `format_schema`.

It's required to set this setting when it is used one of the formats `Cap'n Proto` and `Protobuf`.

The format schema is a combination of a file name and the name of a message type in this file, delimited by colon,

e.g. `schemafile.proto:MessageType`.

If the file has the standard extension for the format (for example, `.proto` for `Protobuf`), it can be omitted and in this case the format schema looks like `schemafile:MessageType`.

If you input or output data via the **client** in the **interactive mode**, the file name specified in the format schema can contain an absolute path or a path relative to the current directory on the client.

If you use the client in the **batch mode**, the path to the schema must be relative due to security reasons.

If you input or output data via the **HTTP interface** the file name specified in the format schema

in your input or output data via the `format_schema` and the name specified in the format schema should be located in the directory specified in `format_schema_path` in the server configuration.

JDBC Driver

- [Official driver](#).
- Third-party driver [ClickHouse-Native-JDBC](#).

ODBC Driver

- [Official driver](#).

Client Libraries from Third-party Developers

Disclaimer

Yandex does **not** maintain the libraries listed below and haven't done any extensive testing to ensure their quality.

- Python
 - [infi.clickhouse_orm](#)
 - [clickhouse-driver](#)
 - [clickhouse-client](#)
 - [aiochclient](#)
- PHP
 - [SeasClick](#)
 - [phpClickHouse](#)
 - [clickhouse-php-client](#)
 - [clickhouse-client](#)
 - [PhpClickHouseClient](#)
- Go
 - [clickhouse](#)
 - [go-clickhouse](#)
 - [mailrugo-clickhouse](#)
 - [golang-clickhouse](#)
- Nodejs
 - [clickhouse \(Nodejs\)](#)
 - [node-clickhouse](#)
- Perl
 - [perl-DBD-ClickHouse](#)
 - [HTTP-ClickHouse](#)
 - [AnyEvent-ClickHouse](#)
- Ruby
 - [clickhouse \(Ruby\)](#)
- R
 - [clickhouse-r](#)
 - [RClickhouse](#)
- Java
 - [clickhouse-client-java](#)
 - [clickhouse-client](#)
- Scala
 - [clickhouse-scala-client](#)
- Kotlin
 - [AORM](#)
- C#
 - [ClickHouse.Ado](#)
 - [ClickHouse.Net](#)
- C++

- [clickhouse-cpp](#)
- Elixir
 - [clickhousex](#)
- Nim
 - [nim-clickhouse](#)

Integration Libraries from Third-party Developers

Disclaimer

Yandex does **not** maintain the tools and libraries listed below and haven't done any extensive testing to ensure their quality.

Infrastructure Products

- Relational database management systems
 - [MySQL](#)
 - [ProxySQL](#)
 - [clickhouse-mysql-data-reader](#)
 - [horgh-replicator](#)
 - [PostgreSQL](#)
 - [clickhousedb_fdw](#)
 - [infi.clickhouse_fdw](#) (uses [infi.clickhouse_orm](#))
 - [pg2ch](#)
 - [MSSQL](#)
 - [ClickHouseMigrator](#)
- Message queues
 - [Kafka](#)
 - [clickhouse_sinker](#) (uses [Go client](#))
- Object storages
 - [S3](#)
 - [clickhouse-backup](#)
- Container orchestration
 - [Kubernetes](#)
 - [clickhouse-operator](#)
- Configuration management
 - [puppet](#)
 - [innogames/clickhouse](#)
 - [mfedotov/clickhouse](#)
- Monitoring
 - [Graphite](#)
 - [graphouse](#)
 - [carbon-clickhouse](#)
 - [Grafana](#)
 - [clickhouse-grafana](#)
 - [Prometheus](#)
 - [clickhouse_exporter](#)
 - [PromHouse](#)
 - [clickhouse_exporter](#) (uses [Go client](#))
 - [Nagios](#)
 - [check_clickhouse](#)
 - [Zabbix](#)
 - [clickhouse-zabbix-template](#)
 - [Sematext](#)
 - [clickhouse integration](#)
- Logging
 - [rsyslog](#)
 - [rsyslog-clickhouse](#)

- `omclickhouse`
- `fluentd`
 - `loghouse` (for `Kubernetes`)
- `logagent`
 - `logagent output-plugin-clickhouse`
- `Geo`
 - `MaxMind`
 - `clickhouse-maxmind-geoip`

Programming Language Ecosystems

- `Python`
 - `SQLAlchemy`
 - `sqlalchemy-clickhouse` (uses `infi.clickhouse_orm`)
 - `pandas`
 - `pandahouse`
- `R`
 - `dplyr`
 - `RClickhouse` (uses `clickhouse-cpp`)
- `Java`
 - `Hadoop`
 - `clickhouse-hdfs-loader` (uses `JDBC`)
- `Scala`
 - `Akka`
 - `clickhouse-scala-client`
- `C#`
 - `ADO.NET`
 - `ClickHouse.Ado`
 - `ClickHouse.Net`
 - `ClickHouse.Net.Migrations`
- `Elixir`
 - `Ecto`
 - `clickhouse_ecto`

Visual Interfaces from Third-party Developers

Open-Source

Tabix

Web interface for ClickHouse in the `Tabix` project.

Features:

- Works with ClickHouse directly from the browser, without the need to install additional software.
- Query editor with syntax highlighting.
- Auto-completion of commands.
- Tools for graphical analysis of query execution.
- Color scheme options.

[Tabix documentation](#).

HouseOps

`HouseOps` is a UI/IDE for OSX, Linux and Windows.

Features:

- Query builder with syntax highlighting. View the response in a table or JSON view.
- Export query results as CSV or JSON.
- List of processes with descriptions. Write mode. Ability to stop (`KILL`) a process.

- Database graph. Shows all tables and their columns with additional information.
- Quick view of the column size.
- Server configuration.

The following features are planned for development:

- Database management.
- User management.
- Real-time data analysis.
- Cluster monitoring.
- Cluster management.
- Monitoring replicated and Kafka tables.

LightHouse

LightHouse is a lightweight web interface for ClickHouse.

Features:

- Table list with filtering and metadata.
- Table preview with filtering and sorting.
- Read-only queries execution.

DBeaver

DBeaver - universal desktop database client with ClickHouse support.

Features:

- Query development with syntax highlight and autocompletion.
- Table list with filters and metadata search.
- Table data preview.
- Full text search.

clickhouse-cli

clickhouse-cli is an alternative command line client for ClickHouse, written in Python 3.

Features:

- Autocompletion.
- Syntax highlighting for the queries and data output.
- Pager support for the data output.
- Custom PostgreSQL-like commands.

Commercial

DataGrip

DataGrip is a database IDE from JetBrains with dedicated support for ClickHouse. It is also embedded into other IntelliJ-based tools: PyCharm, IntelliJ IDEA, GoLand, PhpStorm and others.

Features:

- Very fast code completion.
- ClickHouse syntax highlighting.
- Support for features specific to ClickHouse, for example nested columns, table engines.
- Data Editor.
- Refactorings.
- Search and Navigation.

Proxy Servers from Third-party Developers

chproxy

chproxy, is an http proxy and load balancer for ClickHouse database.

Features:

- Per-user routing and response caching.
- Flexible limits.
- Automatic SSL certificate renewal.

Implemented in Go.

KittenHouse

KittenHouse is designed to be a local proxy between ClickHouse and application server in case it's impossible or inconvenient to buffer INSERT data on your application side.

Features:

- In-memory and on-disk data buffering.
- Per-table routing.
- Load-balancing and health checking.

Implemented in Go.

ClickHouse-Bulk

ClickHouse-Bulk is a simple ClickHouse insert collector.

Features:

- Group requests and send by threshold or interval.
- Multiple remote servers.
- Basic authentication.

Implemented in Go.

Data Types

ClickHouse can store various types of data in table cells.

This section describes the supported data types and special considerations when using and/or implementing them, if any.

UInt8, UInt16, UInt32, UInt64, Int8, Int16, Int32, Int64

Fixed-length integers, with or without a sign.

Int Ranges

- Int8 - [-128 : 127]
- Int16 - [-32768 : 32767]
- Int32 - [-2147483648 : 2147483647]
- Int64 - [-9223372036854775808 : 9223372036854775807]

UInt Ranges

- UInt8 - [0 : 255]
- UInt16 - [0 : 65535]
- UInt32 - [0 : 4294967295]
- UInt64 - [0 : 18446744073709551615]

Float32, Float64

Floating point numbers.

Types are equivalent to types of C:

- Float32 - float
- Float64 - double

We recommend that you store data in integer form whenever possible. For example, convert fixed precision numbers to integer values, such as monetary amounts or page load times in milliseconds.

Using Floating-point Numbers

- Computations with floating-point numbers might produce a rounding error.

```
SELECT 1 - 0.9
```

```
┌──────────minus(1, 0.9)──┐  
│ 0.09999999999999998 │  
└────────────────────────┘
```

- The result of the calculation depends on the calculation method (the processor type and architecture of the computer system).
- Floating-point calculations might result in numbers such as infinity (`Inf`) and "not-a-number" (`NaN`). This should be taken into account when processing the results of calculations.
- When parsing floating point numbers from text, the result might not be the nearest machine-representable number.

NaN and Inf

In contrast to standard SQL, ClickHouse supports the following categories of floating-point numbers:

- `Inf` - Infinity.

```
SELECT 0.5 / 0
```

```
┌────────divide(0.5, 0)──┐  
│ inf │  
└────────────────────────┘
```

- `-Inf` - Negative infinity.

```
SELECT -0.5 / 0
```

```
┌────────divide(-0.5, 0)──┐  
│ -inf │  
└────────────────────────┘
```

- `NaN` - Not a number.

```
SELECT 0 / 0
```

```
┌────────divide(0, 0)──┐  
│ nan │  
└────────────────────────┘
```

See the rules for NaN sorting in the section [ORDER BY clause](#).

Decimal(P, S), Decimal32(S), Decimal64(S), Decimal128(S)

Signed fixed point numbers that keep precision during add, subtract and multiply operations. For division least significant digits are discarded (not rounded).

Parameters

- P - precision. Valid range: [1 : 38]. Determines how many decimal digits number can have (including fraction).
- S - scale. Valid range: [0 : P]. Determines how many decimal digits fraction can have.

Depending on P parameter value Decimal(P, S) is a synonym for:

- P from [1 : 9] - for Decimal32(S)
- P from [10 : 18] - for Decimal64(S)
- P from [19 : 38] - for Decimal128(S)

Decimal value ranges

- Decimal32(S) - ($-1 * 10^{(9 - S)}$, $1 * 10^{(9 - S)}$)
- Decimal64(S) - ($-1 * 10^{(18 - S)}$, $1 * 10^{(18 - S)}$)
- Decimal128(S) - ($-1 * 10^{(38 - S)}$, $1 * 10^{(38 - S)}$)

For example, Decimal32(4) can contain numbers from -99999.9999 to 99999.9999 with 0.0001 step.

Internal representation

Internally data is represented as normal signed integers with respective bit width. Real value ranges that can be stored in memory are a bit larger than specified above, which are checked only on conversion from string.

Because modern CPU's do not support 128 bit integers natively, operations on Decimal128 are emulated. Because of this Decimal128 works significantly slower than Decimal32/Decimal64.

Operations and result type

Binary operations on Decimal result in wider result type (with any order of arguments).

- Decimal64(S1) Decimal32(S2) -> Decimal64(S)
- Decimal128(S1) Decimal32(S2) -> Decimal128(S)
- Decimal128(S1) Decimal64(S2) -> Decimal128(S)

Rules for scale:

- add, subtract: $S = \max(S1, S2)$.
- multiply: $S = S1 + S2$.
- divide: $S = S1$.

For similar operations between Decimal and integers, the result is Decimal of the same size as argument.

Operations between Decimal and Float32/Float64 are not defined. If you really need them, you can explicitly cast one of argument using toDecimal32, toDecimal64, toDecimal128 or toFloat32, toFloat64 builtins. Keep in mind that the result will lose precision and type conversion is computationally expensive operation.

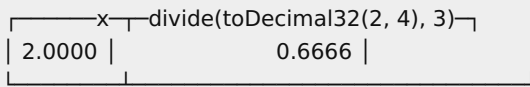
Some functions on Decimal return result as Float64 (for example, var or stddev). Intermediate calculations might still be performed in Decimal, which might lead to different results between Float64 and Decimal inputs with same values.

Overflow checks

During calculations on Decimal, integer overflow might happen. Excessing digits in fraction are discarded (not

During calculations on Decimal, integer overflows might happen. Excessive digits in fraction are discarded (not rounded). Excessive digits in integer part will lead to exception.

```
SELECT toDecimal32(2, 4) AS x, x / 3
```



```
SELECT toDecimal32(4.2, 8) AS x, x * x
```

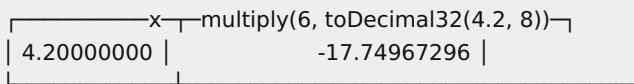
DB::Exception: Scale is out of bounds.

```
SELECT toDecimal32(4.2, 8) AS x, 6 * x
```

DB::Exception: Decimal math overflow.

Overflow checks lead to operations slowdown. If it is known that overflows are not possible, it makes sense to disable checks using `decimal_check_overflow` setting. When checks are disabled and overflow happens, the result will be incorrect:

```
SET decimal_check_overflow = 0;  
SELECT toDecimal32(4.2, 8) AS x, 6 * x
```



Overflow checks happen not only on arithmetic operations, but also on value comparison:

```
SELECT toDecimal32(1, 8) < 100
```

DB::Exception: Can't compare.

Boolean Values

There isn't a separate type for boolean values. They use the UInt8 type, restricted to the values 0 or 1.

String

Strings of an arbitrary length. The length is not limited. The value can contain an arbitrary set of bytes, including null bytes.

The String type replaces the types VARCHAR, BLOB, CLOB, and others from other DBMSs.

Encodings

ClickHouse doesn't have the concept of encodings. Strings can contain an arbitrary set of bytes, which are stored and output as-is.

If you need to store texts, we recommend using UTF-8 encoding. At the very least, if your terminal uses UTF-8 (as recommended), you can read and write your values without making conversions.

Similarly, certain functions for working with strings have separate variations that work under the assumption that the string contains a set of bytes representing a UTF-8 encoded text.

For example, the 'length' function calculates the string length in bytes, while the 'lengthUTF8' function calculates the string length in Unicode code points, assuming that the value is UTF-8 encoded.

FixedString

A fixed-length string of `N` bytes (neither characters nor code points).

To declare a column of `FixedString` type, use the following syntax:

```
<column_name> FixedString(N)
```

Where `N` is a natural number.

The `FixedString` type is efficient when data has the length of precisely `N` bytes. In all other cases, it is likely to reduce efficiency.

Examples of the values that can be efficiently stored in `FixedString`-typed columns:

- Binary representation of IP addresses (`FixedString(16)` for IPv6).
- Language codes (ru_RU, en_US ...).
- Currency codes (USD, RUB ...).
- Binary representation of hashes (`FixedString(16)` for MD5, `FixedString(32)` for SHA256).

To store UUID values, use the `UUID` data type.

When inserting the data, ClickHouse:

- Complements a string with null bytes if the string contains fewer than `N` bytes.
- Throws the `Too large value for FixedString(N)` exception if the string contains more than `N` bytes.

When selecting the data, ClickHouse does not remove the null bytes at the end of the string. If you use the `WHERE` clause, you should add null bytes manually to match the `FixedString` value. The following example illustrates how to use the `WHERE` clause with `FixedString`.

Let's consider the following table with the single `FixedString(2)` column:

```
┌name┐
└─┬─┘
  b
```

The query `SELECT * FROM FixedStringTable WHERE a = 'b'` does not return any data as a result. We should complement the filter pattern with null bytes.

```
SELECT * FROM FixedStringTable
WHERE a = 'b\0'
```

```
┌a┐
└─┘
 b
```

1 rows in set. Elapsed: 0.002 sec.

This behavior differs from MySQL behavior for the `CHAR` type (where strings are padded with spaces, and the spaces are removed for output).

Note that the length of the `FixedString(N)` value is constant. The `length` function returns `N` even if the `FixedString(N)` value is filled only with null bytes, but the `empty` function returns `1` in this case.

UUID

A universally unique identifier (UUID) is a 16-byte number used to identify records. For detailed information about the UUID, see [Wikipedia](#).

The example of UUID type value is represented below:

```
61f0c404-5cb3-11e7-907b-a6006ad3dba0
```

If you do not specify the UUID column value when inserting a new record, the UUID value is filled with zero:

```
00000000-0000-0000-0000-000000000000
```

How to generate

To generate the UUID value, ClickHouse provides the [generateUUIDv4](#) function.

Usage example

Example 1

This example demonstrates creating a table with the UUID type column and inserting a value into the table.

```
:) CREATE TABLE t_uuid (x UUID, y String) ENGINE=TinyLog

:) INSERT INTO t_uuid SELECT generateUUIDv4(), 'Example 1'

:) SELECT * FROM t_uuid
```

x	y
417ddc5d-e556-4d27-95dd-a34d84e46a50	Example 1

Example 2

In this example, the UUID column value is not specified when inserting a new record.

```
:) INSERT INTO t_uuid (y) VALUES ('Example 2')

:) SELECT * FROM t_uuid
```

x	y
417ddc5d-e556-4d27-95dd-a34d84e46a50	Example 1
00000000-0000-0000-0000-000000000000	Example 2

Restrictions

The UUID data type only supports functions which [String](#) data type also supports (for example, [min](#), [max](#), and [count](#)).

The UUID data type is not supported by arithmetic operations (for example, [abs](#)) or aggregate functions, such as [sum](#) and [avg](#).

Date

A date. Stored in two bytes as the number of days since 1970-01-01 (unsigned). Allows storing values from just after the beginning of the Unix Epoch to the upper threshold defined by a constant at the compilation stage

(currently, this is until the year 2106, but the final fully-supported year is 2105).
The minimum value is output as 0000-00-00.

The date is stored without the time zone.

DateTime

Date with time. Stored in four bytes as a Unix timestamp (unsigned). Allows storing values in the same range as for the Date type. The minimal value is output as 0000-00-00 00:00:00.

The time is stored with accuracy up to one second (without leap seconds).

Time Zones

The date with time is converted from text (divided into component parts) to binary and back, using the system's time zone at the time the client or server starts. In text format, information about daylight savings is lost.

By default, the client switches to the timezone of the server when it connects. You can change this behavior by enabling the client command-line option `--use_client_time_zone`.

So when working with a textual date (for example, when saving text dumps), keep in mind that there may be ambiguity during changes for daylight savings time, and there may be problems matching data if the time zone changed.

Enum8, Enum16

Includes the `Enum8` and `Enum16` types. `Enum` saves the finite set of pairs of `'string' = integer`. In ClickHouse, all operations with the `Enum` data type are performed as if value contains integers, although the user is working with string constants. This is more effective in terms of performance than working with the `String` data type.

- `Enum8` is described by pairs of `'String' = Int8`.
- `Enum16` is described by pairs of `'String' = Int16`.

Usage examples

Here we create a table with an `Enum8('hello' = 1, 'world' = 2)` type column:

```
CREATE TABLE t_enum
(
  x Enum8('hello' = 1, 'world' = 2)
)
ENGINE = TinyLog
```

This column `x` can only store the values that are listed in the type definition: `'hello'` or `'world'`. If you try to save any other value, ClickHouse will generate an exception.

```
:) INSERT INTO t_enum VALUES ('hello'), ('world'), ('hello')
```

```
INSERT INTO t_enum VALUES
```

```
Ok.
```

```
3 rows in set. Elapsed: 0.002 sec.
```

```
:) insert into t_enum values('a')
```

```
INSERT INTO t_enum VALUES
```

```
Exception on client:
```

```
Code: 49. DB::Exception: Unknown element 'a' for type Enum8('hello' = 1, 'world' = 2)
```

When you query data from the table, ClickHouse outputs the string values from `Enum`.

```
SELECT * FROM t_enum
```

x
hello
world
hello

If you need to see the numeric equivalents of the rows, you must cast the `Enum` value to integer type.

```
SELECT CAST(x, 'Int8') FROM t_enum
```

CAST(x, 'Int8')
1
2
1

To create an Enum value in a query, you also need to use `CAST`.

```
SELECT toTypeName(CAST('a', 'Enum8(\'a\' = 1, \'b\' = 2)'))
```

toTypeName(CAST('a', 'Enum8(\'a\' = 1, \'b\' = 2)'))
Enum8('a' = 1, 'b' = 2)

General rules and usage

Each of the values is assigned a number in the range `-128 ... 127` for `Enum8` or in the range `-32768 ... 32767` for `Enum16`. All the strings and numbers must be different. An empty string is allowed. If this type is specified (in a table definition), numbers can be in an arbitrary order. However, the order does not matter.

Neither the string nor the numeric value in an `Enum` can be **NULL**.

An `Enum` can be contained in **Nullable** type. So if you create a table using the query

```
CREATE TABLE t_enum_nullable
(
    x Nullable( Enum8('hello' = 1, 'world' = 2) )
)
ENGINE = TinyLog
```

it can store not only `'hello'` and `'world'`, but `NULL`, as well.

```
INSERT INTO t_enum_null Values('hello'),('world'),(NULL)
```

In RAM, an `Enum` column is stored in the same way as `Int8` or `Int16` of the corresponding numerical values.

When reading in text form, ClickHouse parses the value as a string and searches for the corresponding string from the set of Enum values. If it is not found, an exception is thrown. When reading in text format, the string is read and the corresponding numeric value is looked up. An exception will be thrown if it is not found.

When writing in text form, it writes the value as the corresponding string. If column data contains garbage (numbers that are not from the valid set), an exception is thrown. When reading and writing in binary form, it works the same way as for `Int8` and `Int16` data types.

The implicit default value is the value with the lowest number

The implicit default value is the value with the lowest number.

During `ORDER BY`, `GROUP BY`, `IN`, `DISTINCT` and so on, Enums behave the same way as the corresponding numbers. For example, `ORDER BY` sorts them numerically. Equality and comparison operators work the same way on Enums as they do on the underlying numeric values.

Enum values cannot be compared with numbers. Enums can be compared to a constant string. If the string compared to is not a valid value for the Enum, an exception will be thrown. The `IN` operator is supported with the Enum on the left hand side and a set of strings on the right hand side. The strings are the values of the corresponding Enum.

Most numeric and string operations are not defined for Enum values, e.g. adding a number to an Enum or concatenating a string to an Enum.

However, the Enum has a natural `toString` function that returns its string value.

Enum values are also convertible to numeric types using the `toT` function, where `T` is a numeric type. When `T` corresponds to the enum's underlying numeric type, this conversion is zero-cost.

The Enum type can be changed without cost using `ALTER`, if only the set of values is changed. It is possible to both add and remove members of the Enum using `ALTER` (removing is safe only if the removed value has never been used in the table). As a safeguard, changing the numeric value of a previously defined Enum member will throw an exception.

Using `ALTER`, it is possible to change an `Enum8` to an `Enum16` or vice versa, just like changing an `Int8` to `Int16`.

Array(T)

Array of `T`-type items.

`T` can be anything, including an array.

Creating an array

You can use a function to create an array:

```
array(T)
```

You can also use square brackets.

```
[]
```

Example of creating an array:

```
:) SELECT array(1, 2) AS x, toTypeName(x)
```

```
SELECT
  [1, 2] AS x,
  toTypeName(x)
```

```
┌─x──┐┌─toTypeName(array(1, 2))─┐
│ [1,2] │ Array(UInt8)           │
└──────┴────────────────────────┘
```

```
1 rows in set. Elapsed: 0.002 sec.
```

```
:) SELECT [1, 2] AS x, toTypeName(x)
```

```
SELECT
  [1, 2] AS x,
  toTypeName(x)
```

```
┌─x──┐┌─toTypeName([1, 2])─┐
```

```

┌ x ────────── toTypeName([1, 2]) ──────────┐
└── [1,2] ── Array(UInt8) ─────────────────┘

```

1 rows in set. Elapsed: 0.002 sec.

Working with data types

When creating an array on the fly, ClickHouse automatically defines the argument type as the narrowest data type that can store all the listed arguments. If there are any **NULL** or **Nullable** type arguments, the type of array elements is **Nullable**.

If ClickHouse couldn't determine the data type, it will generate an exception. For instance, this will happen when trying to create an array with strings and numbers simultaneously (`SELECT array(1, 'a')`).

Examples of automatic data type detection:

```

:) SELECT array(1, 2, NULL) AS x, toTypeName(x)

```

```

SELECT
  [1, 2, NULL] AS x,
  toTypeName(x)

```

```

┌ x ────────── toTypeName(array(1, 2, NULL)) ──────────┐
└── [1,2,NULL] ── Array(Nullable(UInt8)) ────────────┘

```

1 rows in set. Elapsed: 0.002 sec.

If you try to create an array of incompatible data types, ClickHouse throws an exception:

```

:) SELECT array(1, 'a')

```

```

SELECT [1, 'a']

```

Received exception from server (version 1.1.54388):

Code: 386. DB::Exception: Received from localhost:9000, 127.0.0.1. DB::Exception: There is no supertype for types UInt8, String because some of them are String/FixedString and some of them are not.

0 rows in set. Elapsed: 0.246 sec.

AggregateFunction(name, types_of_arguments...)

The intermediate state of an aggregate function. To get it, use aggregate functions with the `-State` suffix. To get aggregated data in the future, you must use the same aggregate functions with the `-Merge` suffix.

`AggregateFunction` — parametric data type.

Parameters

- Name of the aggregate function.
If the function is parametric specify its parameters too.
- Types of the aggregate function arguments.

Example

```

CREATE TABLE t
(
  column1 AggregateFunction(uniq, UInt64),

```

```
column2 AggregateFunction(anyIf, String, UInt8),
column3 AggregateFunction(quantiles(0.5, 0.9), UInt64)
) ENGINE = ...
```

uniq, **anyIf** (**any**+**If**) and **quantiles** are the aggregate functions supported in ClickHouse.

Usage

Data Insertion

To insert data, use `INSERT SELECT` with aggregate `-State-` functions.

Function examples

```
uniqState(UserID)
quantilesState(0.5, 0.9)(SendTiming)
```

In contrast to the corresponding functions `uniq` and `quantiles`, `-State-` functions return the state, instead the final value. In other words, they return a value of `AggregateFunction` type.

In the results of `SELECT` query the values of `AggregateFunction` type have implementation-specific binary representation for all of the ClickHouse output formats. If dump data into, for example, `TabSeparated` format with `SELECT` query then this dump can be loaded back using `INSERT` query.

Data Selection

When selecting data from `AggregatingMergeTree` table, use `GROUP BY` clause and the same aggregate functions as when inserting data, but using `-Merge` suffix.

An aggregate function with `-Merge` suffix takes a set of states, combines them, and returns the result of complete data aggregation.

For example, the following two queries return the same result:

```
SELECT uniq(UserID) FROM table

SELECT uniqMerge(state) FROM (SELECT uniqState(UserID) AS state FROM table GROUP BY RegionID)
```

Usage Example

See [AggregatingMergeTree](#) engine description.

Tuple(T1, T2, ...)

A tuple of elements, each having an individual **type**.

You can't store tuples in tables (other than Memory tables). They are used for temporary column grouping. Columns can be grouped when an `IN` expression is used in a query, and for specifying certain formal parameters of lambda functions. For more information, see the sections [IN operators](#) and [Higher order functions](#).

Tuples can be the result of a query. In this case, for text formats other than JSON, values are comma-separated in brackets. In JSON formats, tuples are output as arrays (in square brackets).

Creating a tuple

You can use a function to create a tuple:

```
tuple(T1, T2, ...)
```

Example of creating a tuple:


```
) SELECT tuple(1,'a') AS x, toTypeName(x)
```

```
SELECT
  (1, 'a') AS x,
  toTypeName(x)
```

```
x-----toTypeName(tuple(1, 'a'))-----
| (1,'a') | Tuple(UInt8, String) |
```

1 rows in set. Elapsed: 0.021 sec.

Working with data types

When creating a tuple on the fly, ClickHouse automatically detects the type of each argument as the minimum of the types which can store the argument value. If the argument is **NULL**, the type of the tuple element is **Nullable**.

Example of automatic data type detection:

```
SELECT tuple(1, NULL) AS x, toTypeName(x)
```

```
SELECT
  (1, NULL) AS x,
  toTypeName(x)
```

```
x-----toTypeName(tuple(1, NULL))-----
| (1,NULL) | Tuple(UInt8, Nullable(Nothing)) |
```

1 rows in set. Elapsed: 0.002 sec.

Nullable(TypeName)

Allows to store special marker (**NULL**) that denotes "missing value" alongside normal values allowed by `TypeName`. For example, a `Nullable(Int8)` type column can store `Int8` type values, and the rows that don't have a value will store `NULL`.

For a `TypeName`, you can't use composite data types **Array** and **Tuple**. Composite data types can contain `Nullable` type values, such as `Array(Nullable(Int8))`.

A `Nullable` type field can't be included in table indexes.

NULL is the default value for any `Nullable` type, unless specified otherwise in the ClickHouse server configuration.

Storage features

To store `Nullable` type values in table column, ClickHouse uses a separate file with `NULL` masks in addition to normal file with values. Entries in masks file allow ClickHouse to distinguish between `NULL` and default value of corresponding data type for each table row. Because of additional file, `Nullable` column consumes additional storage space compared to similar normal one.

Note

Using `Nullable` almost always negatively affects performance, keep this in mind when designing your databases.

Usage example

```
) CREATE TABLE t_null(x Int8, y Nullable(Int8)) ENGINE TinyLog
```

```
CREATE TABLE t_null
```

```
(  
  x Int8,  
  y Nullable(Int8)  
)
```

```
ENGINE = TinyLog
```

Ok.

0 rows in set. Elapsed: 0.012 sec.

```
:) INSERT INTO t_null VALUES (1, NULL), (2, 3)
```

```
INSERT INTO t_null VALUES
```

Ok.

1 rows in set. Elapsed: 0.007 sec.

```
:) SELECT x + y FROM t_null
```

```
SELECT x + y  
FROM t_null
```

```
┌──plus(x, y)──┐  
|  NULL  |  
|    5   |  
└────────┘
```

2 rows in set. Elapsed: 0.144 sec.

Nested Data Structures

Nested(Name1 Type1, Name2 Type2, ...)

A nested data structure is like a nested table. The parameters of a nested data structure – the column names and types – are specified the same way as in a CREATE query. Each table row can correspond to any number of rows in a nested data structure.

Example:

```
CREATE TABLE test.visits
```

```
(  
  CounterID UInt32,  
  StartDate Date,  
  Sign Int8,  
  IsNew UInt8,  
  VisitID UInt64,  
  UserID UInt64,  
  ...  
  Goals Nested  
  (  
    ID UInt32,  
    Serial UInt32,  
    EventTime DateTime,  
    Price Int64,  
    OrderID String,  
    CurrencyID UInt32  
  ),  
  ...  
)
```

```
) ENGINE = CollapsingMergeTree(StartDate, intHash32(UserID), (CounterID, StartDate, intHash32(UserID), VisitID), 8192,  
  ...)
```

Sign)

This example declares the `Goals` nested data structure, which contains data about conversions (goals reached). Each row in the 'visits' table can correspond to zero or any number of conversions.

Only a single nesting level is supported. Columns of nested structures containing arrays are equivalent to multidimensional arrays, so they have limited support (there is no support for storing these columns in tables with the MergeTree engine).

In most cases, when working with a nested data structure, its individual columns are specified. To do this, the column names are separated by a dot. These columns make up an array of matching types. All the column arrays of a single nested data structure have the same length.

Example:

```
SELECT
  Goals.ID,
  Goals.EventTime
FROM test.visits
WHERE CounterID = 101500 AND length(Goals.ID) < 5
LIMIT 10
```

Goals.ID	Goals.EventTime
[1073752,591325,591325]	['2014-03-17 16:38:10','2014-03-17 16:38:48','2014-03-17 16:42:27']
[1073752]	['2014-03-17 00:28:25']
[1073752]	['2014-03-17 10:46:20']
[1073752,591325,591325,591325]	['2014-03-17 13:59:20','2014-03-17 22:17:55','2014-03-17 22:18:07','2014-03-17 22:18:51']
[]	[]
[1073752,591325,591325]	['2014-03-17 11:37:06','2014-03-17 14:07:47','2014-03-17 14:36:21']
[]	[]
[]	[]
[591325,1073752]	['2014-03-17 00:46:05','2014-03-17 00:46:05']
[1073752,591325,591325,591325]	['2014-03-17 13:28:33','2014-03-17 13:30:26','2014-03-17 18:51:21','2014-03-17 18:51:45']

It is easiest to think of a nested data structure as a set of multiple column arrays of the same length.

The only place where a SELECT query can specify the name of an entire nested data structure instead of individual columns is the ARRAY JOIN clause. For more information, see "ARRAY JOIN clause". Example:

```
SELECT
  Goal.ID,
  Goal.EventTime
FROM test.visits
ARRAY JOIN Goals AS Goal
WHERE CounterID = 101500 AND length(Goals.ID) < 5
LIMIT 10
```

Goal.ID	Goal.EventTime
1073752	2014-03-17 16:38:10
591325	2014-03-17 16:38:48
591325	2014-03-17 16:42:27
1073752	2014-03-17 00:28:25
1073752	2014-03-17 10:46:20

1073752	2014-03-17 13:59:20
591325	2014-03-17 22:17:55
591325	2014-03-17 22:18:07
591325	2014-03-17 22:18:51
1073752	2014-03-17 11:37:06

You can't perform SELECT for an entire nested data structure. You can only explicitly list individual columns that are part of it.

For an INSERT query, you should pass all the component column arrays of a nested data structure separately (as if they were individual column arrays). During insertion, the system checks that they have the same length.

For a DESCRIBE query, the columns in a nested data structure are listed separately in the same way.

The ALTER query is very limited for elements in a nested data structure.

Special Data Types

Special data type values can't be saved to a table or output in results, but are used as the intermediate result of running a query.

Expression

Used for representing lambda expressions in high-order functions.

Set

Used for the right half of an IN expression.

Nothing

The only purpose of this data type is to represent cases where value is not expected. So you can't create a `Nothing` type value.

For example, literal `NULL` has type of `Nullable(Nothing)`. See more about [Nullable](#).

The `Nothing` type can also used to denote empty arrays:

```
:) SELECT toTypeName(array())
```

```
SELECT toTypeName([])
```

```
┌toTypeName(array())┐
└ Array(Nothing)   ┘
```

```
1 rows in set. Elapsed: 0.062 sec.
```

Domains

Domains are special-purpose types, that add some extra features atop of existing base type, leaving on-wire and on-disc format of underlying table intact. At the moment, ClickHouse does not support user-defined domains.

You can use domains anywhere corresponding base type can be used:

- Create a column of domain type
- Read/write values from/to domain column
- Use it as index if base type can be used as index
- Call functions with values of domain column

- etc.

Extra Features of Domains

- Explicit column type name in `SHOW CREATE TABLE` or `DESCRIBE TABLE`
- Input from human-friendly format with `INSERT INTO domain_table(domain_column) VALUES(...)`
- Output to human-friendly format for `SELECT domain_column FROM domain_table`
- Loading data from external source in human-friendly format: `INSERT INTO domain_table FORMAT CSV ...`

Limitations

- Can't convert index column of base type to domain type via `ALTER TABLE`.
- Can't implicitly convert string values into domain values when inserting data from another column or table.
- Domain adds no constraints on stored values.

IPv4

`IPv4` is a domain based on `UInt32` type and serves as typed replacement for storing IPv4 values. It provides compact storage with human-friendly input-output format, and column type information on inspection.

Basic Usage

```
CREATE TABLE hits (url String, from IPv4) ENGINE = MergeTree() ORDER BY url;

DESCRIBE TABLE hits;
```

name	type	default_type	default_expression	comment	codec_expression
url	String				
from	IPv4				

OR you can use IPv4 domain as a key:

```
CREATE TABLE hits (url String, from IPv4) ENGINE = MergeTree() ORDER BY from;
```

`IPv4` domain supports custom input format as IPv4-strings:

```
INSERT INTO hits (url, from) VALUES ('https://wikipedia.org', '116.253.40.133')('https://clickhouse.yandex', '183.247.232.58')
('https://clickhouse.yandex/docs/en/', '116.106.34.242');

SELECT * FROM hits;
```

url	from
https://clickhouse.yandex/docs/en/	116.106.34.242
https://wikipedia.org	116.253.40.133
https://clickhouse.yandex	183.247.232.58

Values are stored in compact binary form:

```
SELECT toTypeName(from), hex(from) FROM hits LIMIT 1;
```

toTypeName(from)	hex(from)
IPv4	B7F7E83A

Domain values are not implicitly convertible to types other than `UInt32`.

If you want to convert `IPv4` value to a string, you have to do that explicitly with `IPv4NumToString()` function:

```
SELECT toTypeName(s), IPv4NumToString(from) as s FROM hits LIMIT 1;
```

toTypeName(IPv4NumToString(from))	s
String	183.247.232.58

Or cast to a `UInt32` value:

```
SELECT toTypeName(i), CAST(from as UInt32) as i FROM hits LIMIT 1;
```

toTypeName(CAST(from, 'UInt32'))	i
UInt32	3086477370

IPv6

`IPv6` is a domain based on `FixedString(16)` type and serves as typed replacement for storing IPv6 values. It provides compact storage with human-friendly input-output format, and column type information on inspection.

Basic Usage

```
CREATE TABLE hits (url String, from IPv6) ENGINE = MergeTree() ORDER BY url;
```

```
DESCRIBE TABLE hits;
```

name	type	default_type	default_expression	comment	codec_expression
url	String				
from	IPv6				

OR you can use `IPv6` domain as a key:

```
CREATE TABLE hits (url String, from IPv6) ENGINE = MergeTree() ORDER BY from;
```

`IPv6` domain supports custom input as IPv6-strings:

```
INSERT INTO hits (url, from) VALUES ('https://wikipedia.org', '2a02:aa08:e000:3100::2')('https://clickhouse.yandex', '2001:44c8:129:2632:33:0:252:2')('https://clickhouse.yandex/docs/en/', '2a02:e980:1e::1');
```

```
SELECT * FROM hits;
```

url	from
https://clickhouse.yandex	2001:44c8:129:2632:33:0:252:2
https://clickhouse.yandex/docs/en/	2a02:e980:1e::1
https://wikipedia.org	2a02:aa08:e000:3100::2

Values are stored in compact binary form:

```
SELECT toTypeName(from), hex(from) FROM hits LIMIT 1;
```

toTypeName(from)	hex(from)
IPv6	200144C8012926320033000002520002

Domain values are not implicitly convertible to types other than `FixedString(16)`.

If you want to convert `IPv6` value to a string, you have to do that explicitly with `IPv6NumToString()` function:

```
SELECT toTypeName(s), IPv6NumToString(from) as s FROM hits LIMIT 1;
```

toTypeName(IPv6NumToString(from))	s
String	2001:44c8:129:2632:33:0:252:2

Or cast to a `FixedString(16)` value:

```
SELECT toTypeName(i), CAST(from as FixedString(16)) as i FROM hits LIMIT 1;
```

toTypeName(CAST(from, 'FixedString(16)'))	i
FixedString(16)	◆◆◆◆

Table engines

The table engine (type of table) determines:

- How and where data is stored, where to write it to, and where to read it from.
- Which queries are supported, and how.
- Concurrent data access.
- Use of indexes, if present.
- Whether multithreaded request execution is possible.
- Data replication parameters.

When reading, the engine is only required to output the requested columns, but in some cases the engine can partially process data when responding to the request.

For most serious tasks, you should use engines from the `MergeTree` family.

MergeTree

The `MergeTree` engine and other engines of this family (`*MergeTree`) are the most robust ClickHouse table engines.

The basic idea for `MergeTree` engines family is the following. When you have tremendous amount of a data that should be inserted into the table, you should write them quickly part by part and then merge parts by some rules in background. This method is much more efficient than constantly rewriting data in the storage at the insert.

Main features:

- Stores data sorted by primary key.

This allows you to create a small sparse index that helps find data faster.

- This allows you to use partitions if the **partitioning key** is specified.

ClickHouse supports certain operations with partitions that are more effective than general operations on the same data with the same result. ClickHouse also automatically cuts off the partition data where the partitioning key is specified in the query. This also increases the query performance.

- Data replication support.

The family of `ReplicatedMergeTree` tables is used for this. For more information, see the **Data replication** section.

- Data sampling support.

If necessary, you can set the data sampling method in the table.

Info

The **Merge** engine does not belong to the `*MergeTree` family.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
  name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1] [TTL expr1],
  name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2] [TTL expr2],
  ...
  INDEX index_name1 expr1 TYPE type1(...) GRANULARITY value1,
  INDEX index_name2 expr2 TYPE type2(...) GRANULARITY value2
) ENGINE = MergeTree()
[PARTITION BY expr]
[ORDER BY expr]
[PRIMARY KEY expr]
[SAMPLE BY expr]
[TTL expr]
[SETTINGS name=value, ...]
```

For a description of request parameters, see **request description**.

Query clauses

- **ENGINE** — Name and parameters of the engine. `ENGINE = MergeTree()`. `MergeTree` engine does not have parameters.
- **PARTITION BY** — The **partitioning key**.

For partitioning by month, use the `toYYYYMM(date_column)` expression, where `date_column` is a column with a date of the type **Date**. The partition names here have the "YYYYMM" format.

- **ORDER BY** — The sorting key.

A tuple of columns or arbitrary expressions. Example: `ORDER BY (CounterID, EventDate)`.

- **PRIMARY KEY** — The primary key if it **differs from the sorting key**.

By default the primary key is the same as the sorting key (which is specified by the `ORDER BY` clause). Thus in most cases it is unnecessary to specify a separate `PRIMARY KEY` clause.

- **SAMPLE BY** — An expression for sampling.

If a sampling expression is used, the primary key must contain it. Example: `SAMPLE BY intHash32(UserID) ORDER BY (CounterID, EventDate, intHash32(UserID))`.

- **TTL** — An expression for setting storage time for rows.

It must depends on `Date` or `DateTime` column and has one `Date` or `DateTime` column as a result. Example:
`TTL date + INTERVAL 1 DAY`

For more details, see [TTL for columns and tables](#)

- **SETTINGS** — Additional parameters that control the behavior of the `MergeTree`:
 - `index_granularity` — The granularity of an index. The number of data rows between the "marks" of an index. By default, 8192. The list of all available parameters you can see in [MergeTreeSettings.h](#).
 - `use_minimalistic_part_header_in_zookeeper` — Storage method of the data parts headers in ZooKeeper. If `use_minimalistic_part_header_in_zookeeper=1`, then ZooKeeper stores less data. For more information refer the [setting description](#) in the "Server configuration parameters" chapter.
 - `min_merge_bytes_to_use_direct_io` — The minimum data volume for merge operation required for using of the direct I/O access to the storage disk. During the merging of the data parts, ClickHouse calculates summary storage volume of all the data to be merged. If the volume exceeds `min_merge_bytes_to_use_direct_io` bytes, then ClickHouse reads and writes the data using direct I/O interface (`O_DIRECT` option) to the storage disk. If `min_merge_bytes_to_use_direct_io = 0`, then the direct I/O is disabled. Default value: $10 * 1024 * 1024 * 1024$ bytes.
 - `merge_with_ttl_timeout` — Minimal time in seconds, when merge with TTL can be repeated. Default value: 86400 (1 day).

Example of sections setting

```
ENGINE MergeTree() PARTITION BY toYYYYMM(EventDate) ORDER BY (CounterID, EventDate, intHash32(UserID)) SAMPLE BY intHash32(UserID) SETTINGS index_granularity=8192
```

In the example, we set partitioning by month.

We also set an expression for sampling as a hash by the user ID. This allows you to pseudorandomize the data in the table for each `CounterID` and `EventDate`. If, when selecting the data, you define a **SAMPLE** clause, ClickHouse will return an evenly pseudorandom data sample for a subset of users.

`index_granularity` could be omitted because 8192 is the default value.

▼ Deprecated Method for Creating a Table

Attention

Do not use this method in new projects and, if possible, switch the old projects to the method described above.

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE [=] MergeTree(date-column [, sampling_expression], (primary, key), index_granularity)
```

MergeTree() parameters

- `date-column` — The name of a column of the type **Date**. ClickHouse automatically creates partitions by month on the basis of this column. The partition names are in the `"YYYYMM"` format.
- `sampling_expression` — an expression for sampling.
- `(primary, key)` — primary key. Type — **Tuple()**
- `index_granularity` — The granularity of an index. The number of data rows between the "marks" of an index. The value 8192 is appropriate for most tasks.

Example

```
MergeTree(EventDate, intHash32(UserID), (CounterID, EventDate, intHash32(UserID)), 8192)
```

The MergeTree engine is configured in the same way as in the example above for the main engine configuration method.

Data Storage

A table consists of data *parts* sorted by primary key.

When data is inserted in a table, separate data parts are created and each of them is lexicographically sorted by primary key. For example, if the primary key is (CounterID, Date), the data in the part is sorted by CounterID, and within each CounterID, it is ordered by Date.

Data belonging to different partitions are separated into different parts. In the background, ClickHouse merges data parts for more efficient storage. Parts belonging to different partitions are not merged. The merge mechanism does not guarantee that all rows with the same primary key will be in the same data part.

For each data part, ClickHouse creates an index file that contains the primary key value for each index row ("mark"). Index row numbers are defined as $n * \text{index_granularity}$. The maximum value n is equal to the integer part of dividing the total number of rows by the `index_granularity`. For each column, the "marks" are also written for the same index rows as the primary key. These "marks" allow you to find the data directly in the columns.

You can use a single large table and continually add data to it in small chunks – this is what the MergeTree engine is intended for.

Primary Keys and Indexes in Queries

Let's take the (CounterID, Date) primary key. In this case, the sorting and index can be illustrated as follows:

```
Whole data:  [------]
CounterID:   [aaaaaaaaaaaaaaaaabbbbcdeeeeeeeeeeeefgggggggghhhhhhhhhiiiiiiiiklllllll]
Date:        [11111112222222333312332111112222223332111111212222223111112223311122333]
Marks:       |  |  |  |  |  |  |  |  |  |  |  |
              a,1 a,2 a,3 b,3 e,2 e,3 g,1 h,2 i,1 i,3 l,3
Marks numbers: 0   1   2   3   4   5   6   7   8   9   10
```

If the data query specifies:

- CounterID in ('a', 'h'), the server reads the data in the ranges of marks [0, 3) and [6, 8).
- CounterID IN ('a', 'h') AND Date = 3, the server reads the data in the ranges of marks [1, 3) and [7, 8).
- Date = 3, the server reads the data in the range of marks [1, 10].

The examples above show that it is always more effective to use an index than a full scan.

A sparse index allows extra data to be read. When reading a single range of the primary key, up to $\text{index_granularity} * 2$ extra rows in each data block can be read. In most cases, ClickHouse performance does not degrade when `index_granularity = 8192`.

Sparse indexes allow you to work with a very large number of table rows, because such indexes are always stored in the computer's RAM.

ClickHouse does not require a unique primary key. You can insert multiple rows with the same primary key.

Selecting the Primary Key

The number of columns in the primary key is not explicitly limited. Depending on the data structure, you can include more or fewer columns in the primary key. This may:

- Improve the performance of an index.

If the primary key is (a, b), then adding another column c will improve the performance if the following conditions are met:

There are queries with a condition on column c

- There are queries with a condition on column `c`.
- Long data ranges (several times longer than the `index_granularity`) with identical values for `(a, b)` are common. In other words, when adding another column allows you to skip quite long data ranges.
- Improve data compression.

ClickHouse sorts data by primary key, so the higher the consistency, the better the compression.

- Provide additional logic when data parts merging in the `CollapsingMergeTree` and `SummingMergeTree` engines.

In this case it makes sense to specify the *sorting key* that is different from the primary key.

A long primary key will negatively affect the insert performance and memory consumption, but extra columns in the primary key do not affect ClickHouse performance during `SELECT` queries.

Choosing the Primary Key that differs from the Sorting Key

It is possible to specify the primary key (the expression, values of which are written into the index file for each mark) that is different from the sorting key (the expression for sorting the rows in data parts). In this case the primary key expression tuple must be a prefix of the sorting key expression tuple.

This feature is helpful when using the `SummingMergeTree` and `AggregatingMergeTree` table engines. In a common case when using these engines the table has two types of columns: *dimensions* and *measures*. Typical queries aggregate values of measure columns with arbitrary `GROUP BY` and filtering by dimensions. As `SummingMergeTree` and `AggregatingMergeTree` aggregate rows with the same value of the sorting key, it is natural to add all dimensions to it. As a result the key expression consists of a long list of columns and this list must be frequently updated with newly added dimensions.

In this case it makes sense to leave only a few columns in the primary key that will provide efficient range scans and add the remaining dimension columns to the sorting key tuple.

ALTER of the sorting key is a lightweight operation because when a new column is simultaneously added to the table and to the sorting key, existing data parts don't need to be changed. Since the old sorting key is a prefix of the new sorting key and there is no data in the just added column, the data at the moment of table modification is sorted by both the old and the new sorting key.

Use of Indexes and Partitions in Queries

For `SELECT` queries, ClickHouse analyzes whether an index can be used. An index can be used if the `WHERE/PREWHERE` clause has an expression (as one of the conjunction elements, or entirely) that represents an equality or inequality comparison operation, or if it has `IN` or `LIKE` with a fixed prefix on columns or expressions that are in the primary key or partitioning key, or on certain partially repetitive functions of these columns, or logical relationships of these expressions.

Thus, it is possible to quickly run queries on one or many ranges of the primary key. In this example, queries will be fast when run for a specific tracking tag; for a specific tag and date range; for a specific tag and date; for multiple tags with a date range, and so on.

Let's look at the engine configured as follows:

```
ENGINE MergeTree() PARTITION BY toYYYYMM(EventDate) ORDER BY (CounterID, EventDate) SETTINGS
index_granularity=8192
```

In this case, in queries:

```
SELECT count() FROM table WHERE EventDate = toDate(now()) AND CounterID = 34
SELECT count() FROM table WHERE EventDate = toDate(now()) AND (CounterID = 34 OR CounterID = 42)
SELECT count() FROM table WHERE ((EventDate >= toDate('2014-01-01') AND EventDate <= toDate('2014-01-31')) OR
EventDate = toDate('2014-05-01')) AND CounterID IN (101500, 731962, 160656) AND (CounterID = 101500 OR EventDate
```

```
!= toDate('2014-05-01'))
```

ClickHouse will use the primary key index to trim improper data and the monthly partitioning key to trim partitions that are in improper date ranges.

The queries above show that the index is used even for complex expressions. Reading from the table is organized so that using the index can't be slower than a full scan.

In the example below, the index can't be used.

```
SELECT count() FROM table WHERE CounterID = 34 OR URL LIKE '%upyachka%'
```

To check whether ClickHouse can use the index when running a query, use the settings `force_index_by_date` and `force_primary_key`.

The key for partitioning by month allows reading only those data blocks which contain dates from the proper range. In this case, the data block may contain data for many dates (up to an entire month). Within a block, data is sorted by primary key, which might not contain the date as the first column. Because of this, using a query with only a date condition that does not specify the primary key prefix will cause more data to be read than for a single date.

Data Skipping Indices (Experimental)

You need to set `allow_experimental_data_skipping_indices` to 1 to use indices. (run `SET allow_experimental_data_skipping_indices = 1`).

Index declaration in the columns section of the `CREATE` query.

```
INDEX index_name expr TYPE type(...) GRANULARITY granularity_value
```

For tables from the `*MergeTree` family data skipping indices can be specified.

These indices aggregate some information about the specified expression on blocks, which consist of `granularity_value` granules (size of the granule is specified using `index_granularity` setting in the table engine), then these aggregates are used in `SELECT` queries for reducing the amount of data to read from the disk by skipping big blocks of data where `where` query cannot be satisfied.

Example

```
CREATE TABLE table_name
(
    u64 UInt64,
    i32 Int32,
    s String,
    ...
    INDEX a (u64 * i32, s) TYPE minmax GRANULARITY 3,
    INDEX b (u64 * length(s)) TYPE set(1000) GRANULARITY 4
) ENGINE = MergeTree()
...
```

Indices from the example can be used by ClickHouse to reduce the amount of data to read from disk in following queries.

```
SELECT count() FROM table WHERE s < 'z'
SELECT count() FROM table WHERE u64 * i32 == 10 AND u64 * length(s) >= 1234
```

Available Types of Indices

• `minmax`

- `minmax`

Stores extremes of the specified expression (if the expression is `tuple`, then it stores extremes for each element of `tuple`), uses stored info for skipping blocks of the data like the primary key.

- `set(max_rows)`

Stores unique values of the specified expression (no more than `max_rows` rows, `max_rows=0` means "no limits"), use them to check if the `WHERE` expression is not satisfiable on a block of the data.

- `ngrambf_v1(n, size_of_bloom_filter_in_bytes, number_of_hash_functions, random_seed)`

Stores **bloom filter** that contains all ngrams from block of data. Works only with strings. Can be used for optimization of `equals`, `like` and `in` expressions.

- `n` — ngram size,
- `size_of_bloom_filter_in_bytes` — bloom filter size in bytes (you can use big values here, for example, 256 or 512, because it can be compressed well),
- `number_of_hash_functions` — number of hash functions used in bloom filter,
- `random_seed` — seed for bloom filter hash functions.

- `tokenbf_v1(size_of_bloom_filter_in_bytes, number_of_hash_functions, random_seed)`

The same as `ngrambf_v1`, but instead of ngrams stores tokens, which are sequences separated by non-alphanumeric characters.

```
INDEX sample_index (u64 * length(s)) TYPE minmax GRANULARITY 4
INDEX sample_index2 (u64 * length(str), i32 + f64 * 100, date, str) TYPE set(100) GRANULARITY 4
INDEX sample_index3 (lower(str), str) TYPE ngrambf_v1(3, 256, 2, 0) GRANULARITY 4
```

Concurrent Data Access

For concurrent table access, we use multi-versioning. In other words, when a table is simultaneously read and updated, data is read from a set of parts that is current at the time of the query. There are no lengthy locks. Inserts do not get in the way of read operations.

Reading from a table is automatically parallelized.

TTL for columns and tables

Data with expired TTL is removed while executing merges.

If TTL is set for column, when it expires, value will be replaced by default. If all values in columns were zeroed in part, data for this column will be deleted from disk for part. You are not allowed to set TTL for all key columns. If TTL is set for table, when it expires, row will be deleted.

When TTL expires on some value or row in part, extra merge will be executed. To control frequency of merges with TTL you can set `merge_with_ttl_timeout`. If it is too low, many extra merges and lack of regular merges can reduce the performance.

Data Replication

Replication is only supported for tables in the MergeTree family:

- `ReplicatedMergeTree`
- `ReplicatedSummingMergeTree`
- `ReplicatedReplacingMergeTree`
- `ReplicatedAggregatingMergeTree`
- `ReplicatedCollapsingMergeTree`
- `ReplicatedVersionedCollapsingMergeTree`
- `ReplicatedGraphiteMergeTree`

Replication works at the level of an individual table, not the entire server. A server can store both replicated and

Replication works at the level of an individual table, not the entire server. A server can store both replicated and non-replicated tables at the same time.

Replication does not depend on sharding. Each shard has its own independent replication.

Compressed data for `INSERT` and `ALTER` queries is replicated (for more information, see the documentation for `ALTER`).

`CREATE`, `DROP`, `ATTACH`, `DETACH` and `RENAME` queries are executed on a single server and are not replicated:

- The `CREATE TABLE` query creates a new replicatable table on the server where the query is run. If this table already exists on other servers, it adds a new replica.
- The `DROP TABLE` query deletes the replica located on the server where the query is run.
- The `RENAME` query renames the table on one of the replicas. In other words, replicated tables can have different names on different replicas.

To use replication, set the addresses of the ZooKeeper cluster in the config file. Example:

```
<zookeeper>
  <node index="1">
    <host>example1</host>
    <port>2181</port>
  </node>
  <node index="2">
    <host>example2</host>
    <port>2181</port>
  </node>
  <node index="3">
    <host>example3</host>
    <port>2181</port>
  </node>
</zookeeper>
```

Use ZooKeeper version 3.4.5 or later.

You can specify any existing ZooKeeper cluster and the system will use a directory on it for its own data (the directory is specified when creating a replicatable table).

If ZooKeeper isn't set in the config file, you can't create replicated tables, and any existing replicated tables will be read-only.

ZooKeeper is not used in `SELECT` queries because replication does not affect the performance of `SELECT` and queries run just as fast as they do for non-replicated tables. When querying distributed replicated tables, ClickHouse behavior is controlled by the settings `max_replica_delay_for_distributed_queries` and `fallback_to_stale_replicas_for_distributed_queries`.

For each `INSERT` query, approximately ten entries are added to ZooKeeper through several transactions. (To be more precise, this is for each inserted block of data; an `INSERT` query contains one block or one block per `max_insert_block_size = 1048576` rows.) This leads to slightly longer latencies for `INSERT` compared to non-replicated tables. But if you follow the recommendations to insert data in batches of no more than one `INSERT` per second, it doesn't create any problems. The entire ClickHouse cluster used for coordinating one ZooKeeper cluster has a total of several hundred `INSERTs` per second. The throughput on data inserts (the number of rows per second) is just as high as for non-replicated data.

For very large clusters, you can use different ZooKeeper clusters for different shards. However, this hasn't proven necessary on the Yandex.Metrica cluster (approximately 300 servers).

Replication is asynchronous and multi-master. `INSERT` queries (as well as `ALTER`) can be sent to any available server. Data is inserted on the server where the query is run, and then it is copied to the other servers. Because it is asynchronous, recently inserted data appears on the other replicas with some latency. If part of the replicas are not available, the data is written when they become available. If a replica is available, the latency is the amount of time it takes to transfer the block of compressed data over the network.

amount of time it takes to transfer the block of compressed data over the network.

By default, an INSERT query waits for confirmation of writing the data from only one replica. If the data was successfully written to only one replica and the server with this replica ceases to exist, the stored data will be lost. To enable getting confirmation of data writes from multiple replicas, use the `insert_quorum` option.

Each block of data is written atomically. The INSERT query is divided into blocks up to `max_insert_block_size = 1048576` rows. In other words, if the INSERT query has less than 1048576 rows, it is made atomically.

Data blocks are deduplicated. For multiple writes of the same data block (data blocks of the same size containing the same rows in the same order), the block is only written once. The reason for this is in case of network failures when the client application doesn't know if the data was written to the DB, so the INSERT query can simply be repeated. It doesn't matter which replica INSERTs were sent to with identical data. INSERTs are idempotent. Deduplication parameters are controlled by `merge_tree` server settings.

During replication, only the source data to insert is transferred over the network. Further data transformation (merging) is coordinated and performed on all the replicas in the same way. This minimizes network usage, which means that replication works well when replicas reside in different datacenters. (Note that duplicating data in different datacenters is the main goal of replication.)

You can have any number of replicas of the same data. Yandex.Metrica uses double replication in production. Each server uses RAID-5 or RAID-6, and RAID-10 in some cases. This is a relatively reliable and convenient solution.

The system monitors data synchronicity on replicas and is able to recover after a failure. Failover is automatic (for small differences in data) or semi-automatic (when data differs too much, which may indicate a configuration error).

Creating Replicated Tables

The `Replicated` prefix is added to the table engine name. For example: `ReplicatedMergeTree`.

Replicated*MergeTree parameters

- `zoo_path` — The path to the table in ZooKeeper.
- `replica_name` — The replica name in ZooKeeper.

Example:

```
CREATE TABLE table_name
(
    EventDate DateTime,
    CounterID UInt32,
    UserID UInt32
) ENGINE = ReplicatedMergeTree('/clickhouse/tables/{layer}-{shard}/table_name', '{replica}')
PARTITION BY toYYYYMM(EventDate)
ORDER BY (CounterID, EventDate, intHash32(UserID))
SAMPLE BY intHash32(UserID)
```

▼ Example in deprecated syntax

```
CREATE TABLE table_name
(
    EventDate DateTime,
    CounterID UInt32,
    UserID UInt32
) ENGINE = ReplicatedMergeTree('/clickhouse/tables/{layer}-{shard}/table_name', '{replica}', EventDate,
intHash32(UserID), (CounterID, EventDate, intHash32(UserID), EventTime), 8192)
```

As the example shows, these parameters can contain substitutions in curly brackets. The substituted values are taken from the 'macros' section of the configuration file. Example:


```
<macros>
  <layer>05</layer>
  <shard>02</shard>
  <replica>example05-02-1.yandex.ru</replica>
</macros>
```

The path to the table in ZooKeeper should be unique for each replicated table. Tables on different shards should have different paths.

In this case, the path consists of the following parts:

`/clickhouse/tables/` is the common prefix. We recommend using exactly this one.

`{layer}-{shard}` is the shard identifier. In this example it consists of two parts, since the Yandex.Metrica cluster uses bi-level sharding. For most tasks, you can leave just the `{shard}` substitution, which will be expanded to the shard identifier.

`table_name` is the name of the node for the table in ZooKeeper. It is a good idea to make it the same as the table name. It is defined explicitly, because in contrast to the table name, it doesn't change after a RENAME query.

HINT: you could add a database name in front of `table_name` as well. E.g. `db_name.table_name`

The replica name identifies different replicas of the same table. You can use the server name for this, as in the example. The name only needs to be unique within each shard.

You can define the parameters explicitly instead of using substitutions. This might be convenient for testing and for configuring small clusters. However, you can't use distributed DDL queries (`ON CLUSTER`) in this case.

When working with large clusters, we recommend using substitutions because they reduce the probability of error.

Run the `CREATE TABLE` query on each replica. This query creates a new replicated table, or adds a new replica to an existing one.

If you add a new replica after the table already contains some data on other replicas, the data will be copied from the other replicas to the new one after running the query. In other words, the new replica syncs itself with the others.

To delete a replica, run `DROP TABLE`. However, only one replica is deleted – the one that resides on the server where you run the query.

Recovery After Failures

If ZooKeeper is unavailable when a server starts, replicated tables switch to read-only mode. The system periodically attempts to connect to ZooKeeper.

If ZooKeeper is unavailable during an `INSERT`, or an error occurs when interacting with ZooKeeper, an exception is thrown.

After connecting to ZooKeeper, the system checks whether the set of data in the local file system matches the expected set of data (ZooKeeper stores this information). If there are minor inconsistencies, the system resolves them by syncing data with the replicas.

If the system detects broken data parts (with the wrong size of files) or unrecognized parts (parts written to the file system but not recorded in ZooKeeper), it moves them to the `detached` subdirectory (they are not deleted). Any missing parts are copied from the replicas.

Note that ClickHouse does not perform any destructive actions such as automatically deleting a large amount of data.

When the server starts (or establishes a new session with ZooKeeper), it only checks the quantity and sizes of all files. If the file sizes match but bytes have been changed somewhere in the middle, this is not detected immediately, but only when attempting to read the data for a `SELECT` query. The query throws an exception

about a non-matching checksum or size of a compressed block. In this case, data parts are added to the verification queue and copied from the replicas if necessary.

If the local set of data differs too much from the expected one, a safety mechanism is triggered. The server enters this in the log and refuses to launch. The reason for this is that this case may indicate a configuration error, such as if a replica on a shard was accidentally configured like a replica on a different shard. However, the thresholds for this mechanism are set fairly low, and this situation might occur during normal failure recovery. In this case, data is restored semi-automatically - by "pushing a button".

To start recovery, create the node `/path_to_table/replica_name/flags/force_restore_data` in ZooKeeper with any content, or run the command to restore all replicated tables:

```
sudo -u clickhouse touch /var/lib/clickhouse/flags/force_restore_data
```

Then restart the server. On start, the server deletes these flags and starts recovery.

Recovery After Complete Data Loss

If all data and metadata disappeared from one of the servers, follow these steps for recovery:

1. Install ClickHouse on the server. Define substitutions correctly in the config file that contains the shard identifier and replicas, if you use them.
2. If you had unreplicated tables that must be manually duplicated on the servers, copy their data from a replica (in the directory `/var/lib/clickhouse/data/db_name/table_name/`).
3. Copy table definitions located in `/var/lib/clickhouse/metadata/` from a replica. If a shard or replica identifier is defined explicitly in the table definitions, correct it so that it corresponds to this replica. (Alternatively, start the server and make all the `ATTACH TABLE` queries that should have been in the .sql files in `/var/lib/clickhouse/metadata/`.)
4. To start recovery, create the ZooKeeper node `/path_to_table/replica_name/flags/force_restore_data` with any content, or run the command to restore all replicated tables: `sudo -u clickhouse touch /var/lib/clickhouse/flags/force_restore_data`

Then start the server (restart, if it is already running). Data will be downloaded from replicas.

An alternative recovery option is to delete information about the lost replica from ZooKeeper (`/path_to_table/replica_name`), then create the replica again as described in "[Creating replicated tables](#)".

There is no restriction on network bandwidth during recovery. Keep this in mind if you are restoring many replicas at once.

Converting from MergeTree to ReplicatedMergeTree

We use the term `MergeTree` to refer to all table engines in the `MergeTree` family, the same as for `ReplicatedMergeTree`.

If you had a `MergeTree` table that was manually replicated, you can convert it to a replicated table. You might need to do this if you have already collected a large amount of data in a `MergeTree` table and now you want to enable replication.

If the data differs on various replicas, first sync it, or delete this data on all the replicas except one.

Rename the existing `MergeTree` table, then create a `ReplicatedMergeTree` table with the old name.

Move the data from the old table to the `detached` subdirectory inside the directory with the new table data (`/var/lib/clickhouse/data/db_name/table_name/`).

Then run `ALTER TABLE ATTACH PARTITION` on one of the replicas to add these data parts to the working set.

Converting from ReplicatedMergeTree to MergeTree

Create a `MergeTree` table with a different name. Move all the data from the directory with the `ReplicatedMergeTree` table data to the new table's data directory. Then delete the `ReplicatedMergeTree` table and

restart the server.

If you want to get rid of a `ReplicatedMergeTree` table without launching the server:

- Delete the corresponding `.sql` file in the metadata directory (`/var/lib/clickhouse/metadata/`).
- Delete the corresponding path in ZooKeeper (`/path_to_table/replica_name`).

After this, you can launch the server, create a `MergeTree` table, move the data to its directory, and then restart the server.

Recovery When Metadata in The ZooKeeper Cluster is Lost or Damaged

If the data in ZooKeeper was lost or damaged, you can save data by moving it to an unreplicated table as described above.

Custom Partitioning Key

Partitioning is available for the `MergeTree` family tables (including `replicated` tables). `Materialized views` based on `MergeTree` tables support partitioning, as well.

A partition is a logical combination of records in a table by a specified criterion. You can set a partition by an arbitrary criterion, such as by month, by day, or by event type. Each partition is stored separately in order to simplify manipulations of this data. When accessing the data, ClickHouse uses the smallest subset of partitions possible.

The partition is specified in the `PARTITION BY expr` clause when `creating a table`. The partition key can be any expression from the table columns. For example, to specify partitioning by month, use the expression `toYYYYMM(date_column)`:

```
CREATE TABLE visits
(
    VisitDate Date,
    Hour UInt8,
    ClientID UUID
)
ENGINE = MergeTree()
PARTITION BY toYYYYMM(VisitDate)
ORDER BY Hour;
```

The partition key can also be a tuple of expressions (similar to the `primary key`). For example:

```
ENGINE = ReplicatedCollapsingMergeTree('/clickhouse/tables/name', 'replica1', Sign)
PARTITION BY (toMonday(StartDate), EventType)
ORDER BY (CounterID, StartDate, intHash32(UserID));
```

In this example, we set partitioning by the event types that occurred during the current week.

When inserting new data to a table, this data is stored as a separate part (chunk) sorted by the primary key. In 10-15 minutes after inserting, the parts of the same partition are merged into the entire part.

Info

A merge only works for data parts that have the same value for the partitioning expression. This means **you shouldn't make overly granular partitions** (more than about a thousand partitions). Otherwise, the `SELECT` query performs poorly because of an unreasonably large number of files in the file system and open file descriptors.

Use the `system.parts` table to view the table parts and partitions. For example, let's assume that we have a `visits` table with partitioning by month. Let's perform the `SELECT` query for the `system.parts` table:

```
SELECT
    partition,
    name,
    active
FROM system.parts
WHERE table = 'visits'
```

partition	name	active
201901	201901_1_3_1	0
201901	201901_1_9_2	1
201901	201901_8_8_0	0
201901	201901_9_9_0	0
201902	201902_4_6_1	1
201902	201902_10_10_0	1
201902	201902_11_11_0	1

The `partition` column contains the names of the partitions. There are two partitions in this example: `201901` and `201902`. You can use this column value to specify the partition name in **ALTER ... PARTITION** queries.

The `name` column contains the names of the partition data parts. You can use this column to specify the name of the part in the **ALTER ATTACH PART** query.

Let's break down the name of the first part: `201901_1_3_1`:

- `201901` is the partition name.
- `1` is the minimum number of the data block.
- `3` is the maximum number of the data block.
- `1` is the chunk level (the depth of the merge tree it is formed from).

Info

The parts of old-type tables have the name: `20190117_20190123_2_2_0` (minimum date - maximum date - minimum block number - maximum block number - level).

The `active` column shows the status of the part. `1` is active; `0` is inactive. The inactive parts are, for example, source parts remaining after merging to a larger part. The corrupted data parts are also indicated as inactive.

As you can see in the example, there are several separated parts of the same partition (for example, `201901_1_3_1` and `201901_1_9_2`). This means that these parts are not merged yet. ClickHouse merges the inserted parts of data periodically, approximately 15 minutes after inserting. In addition, you can perform a non-scheduled merge using the **OPTIMIZE** query. Example:

```
OPTIMIZE TABLE visits PARTITION 201902;
```

partition	name	active
201901	201901_1_3_1	0
201901	201901_1_9_2	1
201901	201901_8_8_0	0
201901	201901_9_9_0	0
201902	201902_4_6_1	0
201902	201902_4_11_2	1
201902	201902_10_10_0	0
201902	201902_11_11_0	0

Inactive parts will be deleted approximately 10 minutes after merging.

Another way to view a set of parts and partitions is to go into the directory of the table:

`/var/lib/clickhouse/data/<database>/<table>/`. For example:

```
dev:/var/lib/clickhouse/data/default/visits$ ls -l
total 40
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  1 16:48 201901_1_3_1
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  5 16:17 201901_1_9_2
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  5 15:52 201901_8_8_0
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  5 15:52 201901_9_9_0
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  5 16:17 201902_10_10_0
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  5 16:17 201902_11_11_0
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  5 16:19 201902_4_11_2
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  5 12:09 201902_4_6_1
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  1 16:48 detached
```

The folders '201901_1_1_0', '201901_1_7_1' and so on are the directories of the parts. Each part relates to a corresponding partition and contains data just for a certain month (the table in this example has partitioning by month).

The `detached` directory contains parts that were detached from the table using the `DETACH` query. The corrupted parts are also moved to this directory, instead of being deleted. The server does not use the parts from the `detached` directory. You can add, delete, or modify the data in this directory at any time – the server will not know about this until you run the `ATTACH` query.

Note that on the operating server, you cannot manually change the set of parts or their data on the file system, since the server will not know about it. For non-replicated tables, you can do this when the server is stopped, but it isn't recommended. For replicated tables, the set of parts cannot be changed in any case.

ClickHouse allows you to perform operations with the partitions: delete them, copy from one table to another, or create a backup. See the list of all operations in the section [Manipulations With Partitions and Parts](#).

ReplacingMergeTree

The engine differs from `MergeTree` in that it removes duplicate entries with the same primary key value (or more accurately, with the same `sorting key` value).

Data deduplication occurs only during a merge. Merging occurs in the background at an unknown time, so you can't plan for it. Some of the data may remain unprocessed. Although you can run an unscheduled merge using the `OPTIMIZE` query, don't count on using it, because the `OPTIMIZE` query will read and write a large amount of data.

Thus, `ReplacingMergeTree` is suitable for clearing out duplicate data in the background in order to save space, but it doesn't guarantee the absence of duplicates.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = ReplacingMergeTree([ver])
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

For a description of request parameters, see [request description](#).

ReplacingMergeTree Parameters

- `ver` — column with version. Type `UInt*`, `Date` or `DateTime`. Optional parameter.

When merging, `ReplacingMergeTree` from all the rows with the same primary key leaves only one:

- Last in the selection, if `ver` not set.
- With the maximum version, if `ver` specified.

Query clauses

When creating a `ReplacingMergeTree` table the same **clauses** are required, as when creating a `MergeTree` table.

▼ Deprecated Method for Creating a Table

Attention

Do not use this method in new projects and, if possible, switch the old projects to the method described above.

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE [=] ReplacingMergeTree(date-column [, sampling_expression], (primary, key), index_granularity, [ver])
```

All of the parameters excepting `ver` have the same meaning as in `MergeTree`.

- `ver` - column with the version. Optional parameter. For a description, see the text above.

SummingMergeTree

The engine inherits from `MergeTree`. The difference is that when merging data parts for `SummingMergeTree` tables ClickHouse replaces all the rows with the same primary key (or more accurately, with the same **sorting key**) with one row which contains summarized values for the columns with the numeric data type. If the sorting key is composed in a way that a single key value corresponds to large number of rows, this significantly reduces storage volume and speeds up data selection.

We recommend to use the engine together with `MergeTree`. Store complete data in `MergeTree` table, and use `SummingMergeTree` for aggregated data storing, for example, when preparing reports. Such an approach will prevent you from losing valuable data due to an incorrectly composed primary key.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = SummingMergeTree([columns])
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

For a description of request parameters, see [request description](#).

Parameters of SummingMergeTree

- `columns` - a tuple with the names of columns where values will be summarized. Optional parameter. The columns must be of a numeric type and must not be in the primary key.

If `columns` not specified, ClickHouse summarizes the values in all columns with a numeric data type that are

not in the primary key.

Query clauses

When creating a `SummingMergeTree` table the same **clauses** are required, as when creating a `MergeTree` table.

▼ Deprecated Method for Creating a Table

Attention

Do not use this method in new projects and, if possible, switch the old projects to the method described above.

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE [=] SummingMergeTree(date-column [, sampling_expression], (primary, key), index_granularity, [columns])
```

All of the parameters excepting `columns` have the same meaning as in `MergeTree`.

- `columns` — tuple with names of columns values of which will be summarized. Optional parameter. For a description, see the text above.

Usage Example

Consider the following table:

```
CREATE TABLE summtt
(
    key UInt32,
    value UInt32
)
ENGINE = SummingMergeTree()
ORDER BY key
```

Insert data to it:

```
:) INSERT INTO summtt Values(1,1),(1,2),(2,1)
```

ClickHouse may sum all the rows not completely (**see below**), so we use an aggregate function `sum` and `GROUP BY` clause in the query.

```
SELECT key, sum(value) FROM summtt GROUP BY key
```

key	sum(value)
2	1
1	3

Data Processing

When data are inserted into a table, they are saved as-is. Clickhouse merges the inserted parts of data periodically and this is when rows with the same primary key are summed and replaced with one for each resulting part of data.

ClickHouse can merge the data parts so that different resulting parts of data can consist rows with the same primary key, i.e. the summation will be incomplete. Therefore (`SELECT`) an aggregate function `sum()` and `GROUP`

`BY` clause should be used in a query as described in the example above.

Common rules for summation

The values in the columns with the numeric data type are summarized. The set of columns is defined by the parameter `columns`.

If the values were 0 in all of the columns for summation, the row is deleted.

If column is not in the primary key and is not summarized, an arbitrary value is selected from the existing ones.

The values are not summarized for columns in the primary key.

The Summation in the AggregateFunction Columns

For columns of `AggregateFunction` type ClickHouse behaves as `AggregatingMergeTree` engine aggregating according to the function.

Nested Structures

Table can have nested data structures that are processed in a special way.

If the name of a nested table ends with `Map` and it contains at least two columns that meet the following criteria:

- the first column is numeric (`*Int*`, `Date`, `DateTime`), let's call it `key`,
- the other columns are arithmetic (`*Int*`, `Float32/64`), let's call it `(values...)`,

then this nested table is interpreted as a mapping of `key => (values...)`, and when merging its rows, the elements of two data sets are merged by `key` with a summation of the corresponding `(values...)`.

Examples:

```
[(1, 100)] + [(2, 150)] -> [(1, 100), (2, 150)]
[(1, 100)] + [(1, 150)] -> [(1, 250)]
[(1, 100)] + [(1, 150), (2, 150)] -> [(1, 250), (2, 150)]
[(1, 100), (2, 150)] + [(1, -100)] -> [(2, 150)]
```

When requesting data, use the `sumMap(key, value)` function for aggregation of `Map`.

For nested data structure, you do not need to specify its columns in the tuple of columns for summation.

AggregatingMergeTree

The engine inherits from `MergeTree`, altering the logic for data parts merging. ClickHouse replaces all rows with the same primary key (or more accurately, with the same `sorting key`) with a single row (within a one data part) that stores a combination of states of aggregate functions.

You can use `AggregatingMergeTree` tables for incremental data aggregation, including for aggregated materialized views.

The engine processes all columns with `AggregateFunction` type.

It is appropriate to use `AggregatingMergeTree` if it reduces the number of rows by orders.

Creating a Table

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

```
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

For a description of request parameters, see [request description](#).

Query clauses

When creating a `AggregatingMergeTree` table the same [clauses](#) are required, as when creating a `MergeTree` table.

▼ Deprecated Method for Creating a Table

Attention

Do not use this method in new projects and, if possible, switch the old projects to the method described above.

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE [=] AggregatingMergeTree(date-column [, sampling_expression], (primary, key), index_granularity)
```

All of the parameters have the same meaning as in `MergeTree`.

SELECT and INSERT

To insert data, use [INSERT SELECT](#) query with aggregate `-State-` functions.

When selecting data from `AggregatingMergeTree` table, use `GROUP BY` clause and the same aggregate functions as when inserting data, but using `-Merge` suffix.

In the results of `SELECT` query the values of `AggregateFunction` type have implementation-specific binary representation for all of the ClickHouse output formats. If dump data into, for example, `TabSeparated` format with `SELECT` query then this dump can be loaded back using `INSERT` query.

Example of an Aggregated Materialized View

`AggregatingMergeTree` materialized view that watches the `test.visits` table:

```
CREATE MATERIALIZED VIEW test.basic
ENGINE = AggregatingMergeTree() PARTITION BY toYYYYMM(StartDate) ORDER BY (CounterID, StartDate)
AS SELECT
    CounterID,
    StartDate,
    sumState(Sign) AS Visits,
    uniqState(UserID) AS Users
FROM test.visits
GROUP BY CounterID, StartDate;
```

Inserting of data into the `test.visits` table.

```
INSERT INTO test.visits ...
```

The data are inserted in both the table and view `test.basic` that will perform the aggregation.

To get the aggregated data, we need to execute a query such as `SELECT ... GROUP BY ...` from the view `test.basic`:

```
SELECT
    StartDate
```



```

StartDate,
sumMerge(Visits) AS Visits,
uniqMerge(Users) AS Users
FROM test.basic
GROUP BY StartDate
ORDER BY StartDate;

```

CollapsingMergeTree

The engine inherits from [MergeTree](#) and adds the logic of rows collapsing to data parts merge algorithm.

[CollapsingMergeTree](#) asynchronously deletes (collapses) pairs of rows if all of the fields in a row are equivalent excepting the particular field `Sign` which can have `1` and `-1` values. Rows without a pair are kept. For more details see the [Collapsing](#) section of the document.

The engine may significantly reduce the volume of storage and increase efficiency of `SELECT` query as a consequence.

Creating a Table

```

CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = CollapsingMergeTree(sign)
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]

```

For a description of query parameters, see [query description](#).

CollapsingMergeTree Parameters

- `sign` — Name of the column with the type of row: `1` is a "state" row, `-1` is a "cancel" row.

Column data type — `Int8`.

Query clauses

When creating a [CollapsingMergeTree](#) table, the same [query clauses](#) are required, as when creating a [MergeTree](#) table.

▼ Deprecated Method for Creating a Table

Attention

Do not use this method in new projects and, if possible, switch the old projects to the method described above.

```

CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE [=] CollapsingMergeTree(date-column [, sampling_expression], (primary, key), index_granularity, sign)

```

All of the parameters excepting `sign` have the same meaning as in [MergeTree](#).

- `sign` — Name of the column with the type of row: `1` — "state" row, `-1` — "cancel" row.

Column Data Type — `Int8`.

Collapsing

Data

Consider the situation where you need to save continually changing data for some object. It sounds logical to have one row for an object and update it at any change, but update operation is expensive and slow for DBMS because it requires rewriting of the data in the storage. If you need to write data quickly, update not acceptable, but you can write the changes of an object sequentially as follows.

Use the particular column `Sign`. If `Sign = 1` it means that the row is a state of an object, let's call it "state" row. If `Sign = -1` it means the cancellation of the state of an object with the same attributes, let's call it "cancel" row.

For example, we want to calculate how much pages users checked at some site and how long they were there. At some moment of time we write the following row with the state of user activity:

UserID	PageViews	Duration	Sign
4324182021466249494	5	146	1

At some moment later we register the change of user activity and write it with the following two rows.

UserID	PageViews	Duration	Sign
4324182021466249494	5	146	-1
4324182021466249494	6	185	1

The first row cancels the previous state of the object (user). It should copy all of the fields of the canceled state excepting `Sign`.

The second row contains the current state.

As we need only the last state of user activity, the rows

UserID	PageViews	Duration	Sign
4324182021466249494	5	146	1
4324182021466249494	5	146	-1

can be deleted collapsing the invalid (old) state of an object. `CollapsingMergeTree` does this while merging of the data parts.

Why we need 2 rows for each change read in the [Algorithm](#) paragraph.

Peculiar properties of such approach

1. The program that writes the data should remember the state of an object to be able to cancel it. "Cancel" string should be the copy of "state" string with the opposite `Sign`. It increases the initial size of storage but allows to write the data quickly.
2. Long growing arrays in columns reduce the efficiency of the engine due to load for writing. The more straightforward data, the higher efficiency.
3. The `SELECT` results depend strongly on the consistency of object changes history. Be accurate when preparing data for inserting. You can get unpredictable results in inconsistent data, for example, negative values for non-negative metrics such as session depth.

Algorithm

When ClickHouse merges data parts, each group of consecutive rows with the same primary key is reduced to not more than two rows, one with `Sign = 1` ("state" row) and another with `Sign = -1` ("cancel" row). In other words entries collapse

words, changes collapse.

For each resulting data part ClickHouse saves:

1. The first "cancel" and the last "state" rows, if the number of "state" and "cancel" rows matches.
2. The last "state" row, if there is one more "state" row than "cancel" rows.
3. The first "cancel" row, if there is one more "cancel" row than "state" rows.
4. None of the rows, in all other cases.

The merge continues, but ClickHouse treats this situation as a logical error and records it in the server log. This error can occur if the same data were inserted more than once.

Thus, collapsing should not change the results of calculating statistics.

Changes gradually collapsed so that in the end only the last state of almost every object left.

The `Sign` is required because the merging algorithm doesn't guarantee that all of the rows with the same primary key will be in the same resulting data part and even on the same physical server. ClickHouse process `SELECT` queries with multiple threads, and it can not predict the order of rows in the result. The aggregation is required if there is a need to get completely "collapsed" data from `CollapsingMergeTree` table.

To finalize collapsing write a query with `GROUP BY` clause and aggregate functions that account for the sign. For example, to calculate quantity, use `sum(Sign)` instead of `count()`. To calculate the sum of something, use `sum(Sign * x)` instead of `sum(x)`, and so on, and also add `HAVING sum(Sign) > 0`.

The aggregates `count`, `sum` and `avg` could be calculated this way. The aggregate `uniq` could be calculated if an object has at least one state not collapsed. The aggregates `min` and `max` could not be calculated because `CollapsingMergeTree` does not save values history of the collapsed states.

If you need to extract data without aggregation (for example, to check whether rows are present whose newest values match certain conditions), you can use the `FINAL` modifier for the `FROM` clause. This approach is significantly less efficient.

Example of use

Example data:

UserID	PageViews	Duration	Sign
4324182021466249494	5	146	1
4324182021466249494	5	146	-1
4324182021466249494	6	185	1

Creation of the table:

```
CREATE TABLE UAct
(
  UserID UInt64,
  PageViews UInt8,
  Duration UInt8,
  Sign Int8
)
ENGINE = CollapsingMergeTree(Sign)
ORDER BY UserID
```

Insertion of the data:

```
INSERT INTO UAct VALUES (4324182021466249494, 5, 146, 1)
```

```
INSERT INTO UAct VALUES (4324182021466249494, 5, 146, -1),(4324182021466249494, 6, 185, 1)
```

We use two `INSERT` queries to create two different data parts. If we insert the data with one query ClickHouse creates one data part and will not perform any merge ever.

Getting the data:

```
SELECT * FROM UAct
```

UserID	PageViews	Duration	Sign
4324182021466249494	5	146	-1
4324182021466249494	6	185	1

UserID	PageViews	Duration	Sign
4324182021466249494	5	146	1

What do we see and where is collapsing?

With two `INSERT` queries, we created 2 data parts. The `SELECT` query was performed in 2 threads, and we got a random order of rows. Collapsing not occurred because there was no merge of the data parts yet. ClickHouse merges data part in an unknown moment of time which we can not predict.

Thus we need aggregation:

```
SELECT
  UserID,
  sum(PageViews * Sign) AS PageViews,
  sum(Duration * Sign) AS Duration
FROM UAct
GROUP BY UserID
HAVING sum(Sign) > 0
```

UserID	PageViews	Duration
4324182021466249494	6	185

If we do not need aggregation and want to force collapsing, we can use `FINAL` modifier for `FROM` clause.

```
SELECT * FROM UAct FINAL
```

UserID	PageViews	Duration	Sign
4324182021466249494	6	185	1

This way of selecting the data is very inefficient. Don't use it for big tables.

VersionedCollapsingMergeTree

This engine:

- Allows quick writing of object states that are continually changing.
- Deletes old object states in the background. This significantly reduces the volume of storage.

See the section [Collapsing](#) for details.

The engine inherits from `MergeTree` and adds the logic for collapsing rows to the algorithm for merging data

The engine inherits from `MergeTree` and adds the logic for collapsing rows to the algorithm for merging data parts. `VersionedCollapsingMergeTree` serves the same purpose as `CollapsingMergeTree` but uses a different collapsing algorithm that allows inserting the data in any order with multiple threads. In particular, the `Version` column helps to collapse the rows properly even if they are inserted in the wrong order. In contrast, `CollapsingMergeTree` allows only strictly consecutive insertion.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = VersionedCollapsingMergeTree(sign, version)
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

For a description of query parameters, see the [query description](#).

Engine Parameters

`VersionedCollapsingMergeTree(sign, version)`

- `sign` — Name of the column with the type of row: `1` is a "state" row, `-1` is a "cancel" row.

The column data type should be `Int8`.

- `version` — Name of the column with the version of the object state.

The column data type should be `UInt*`.

Query Clauses

When creating a `VersionedCollapsingMergeTree` table, the same [clauses](#) are required as when creating a `MergeTree` table.

▼ Deprecated Method for Creating a Table

Attention

Do not use this method in new projects. If possible, switch the old projects to the method described above.

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE [=] VersionedCollapsingMergeTree(date-column [, sampling_expression], (primary, key), index_granularity, sign, version)
```

All of the parameters except `sign` and `version` have the same meaning as in `MergeTree`.

- `sign` — Name of the column with the type of row: `1` is a "state" row, `-1` is a "cancel" row.

Column Data Type — `Int8`.

- `version` — Name of the column with the version of the object state.

The column data type should be `UInt*`.

Collapsing

Data

Consider a situation where you need to save continually changing data for some object. It is reasonable to have one row for an object and update the row whenever there are changes. However, the update operation is expensive and slow for a DBMS because it requires rewriting the data in the storage. Update is not acceptable if you need to write data quickly, but you can write the changes to an object sequentially as follows.

Use the `Sign` column when writing the row. If `Sign = 1` it means that the row is a state of an object (let's call it the "state" row). If `Sign = -1` it indicates the cancellation of the state of an object with the same attributes (let's call it the "cancel" row). Also use the `Version` column, which should identify each state of an object with a separate number.

For example, we want to calculate how many pages users visited on some site and how long they were there. At some point in time we write the following row with the state of user activity:

UserID	PageViews	Duration	Sign	Version
4324182021466249494	5	146	1	1

At some point later we register the change of user activity and write it with the following two rows.

UserID	PageViews	Duration	Sign	Version
4324182021466249494	5	146	-1	1
4324182021466249494	6	185	1	2

The first row cancels the previous state of the object (user). It should copy all of the fields of the canceled state except `Sign`.

The second row contains the current state.

Because we need only the last state of user activity, the rows

UserID	PageViews	Duration	Sign	Version
4324182021466249494	5	146	1	1
4324182021466249494	5	146	-1	1

can be deleted, collapsing the invalid (old) state of the object. `VersionedCollapsingMergeTree` does this while merging the data parts.

To find out why we need two rows for each change, see [Algorithm](#).

Notes on Usage

1. The program that writes the data should remember the state of an object in order to cancel it. The "cancel" string should be a copy of the "state" string with the opposite `Sign`. This increases the initial size of storage but allows to write the data quickly.
2. Long growing arrays in columns reduce the efficiency of the engine due to the load for writing. The more straightforward the data, the better the efficiency.
3. `SELECT` results depend strongly on the consistency of the history of object changes. Be accurate when preparing data for inserting. You can get unpredictable results with inconsistent data, such as negative values for non-negative metrics like session depth.

Algorithm

When ClickHouse merges data parts, it deletes each pair of rows that have the same primary key and version

and different `Sign`. The order of rows does not matter.

When ClickHouse inserts data, it orders rows by the primary key. If the `Version` column is not in the primary key, ClickHouse adds it to the primary key implicitly as the last field and uses it for ordering.

Selecting Data

ClickHouse doesn't guarantee that all of the rows with the same primary key will be in the same resulting data part or even on the same physical server. This is true both for writing the data and for subsequent merging of the data parts. In addition, ClickHouse processes `SELECT` queries with multiple threads, and it cannot predict the order of rows in the result. This means that aggregation is required if there is a need to get completely "collapsed" data from a `VersionedCollapsingMergeTree` table.

To finalize collapsing, write a query with a `GROUP BY` clause and aggregate functions that account for the sign. For example, to calculate quantity, use `sum(Sign)` instead of `count()`. To calculate the sum of something, use `sum(Sign * x)` instead of `sum(x)`, and add `HAVING sum(Sign) > 0`.

The aggregates `count`, `sum` and `avg` can be calculated this way. The aggregate `uniq` can be calculated if an object has at least one non-collapsed state. The aggregates `min` and `max` can't be calculated because `VersionedCollapsingMergeTree` does not save the history of values of collapsed states.

If you need to extract the data with "collapsing" but without aggregation (for example, to check whether rows are present whose newest values match certain conditions), you can use the `FINAL` modifier for the `FROM` clause. This approach is inefficient and should not be used with large tables.

Example of Use

Example data:

UserID	PageViews	Duration	Sign	Version
4324182021466249494	5	146	1	1
4324182021466249494	5	146	-1	1
4324182021466249494	6	185	1	2

Creating the table:

```
CREATE TABLE UAct
(
    UserID UInt64,
    PageViews UInt8,
    Duration UInt8,
    Sign Int8,
    Version UInt8
)
ENGINE = VersionedCollapsingMergeTree(Sign, Version)
ORDER BY UserID
```

Inserting the data:

```
INSERT INTO UAct VALUES (4324182021466249494, 5, 146, 1, 1)
```

```
INSERT INTO UAct VALUES (4324182021466249494, 5, 146, -1, 1),(4324182021466249494, 6, 185, 1, 2)
```

We use two `INSERT` queries to create two different data parts. If we insert the data with a single query, ClickHouse creates one data part and will never perform any merge.

Getting the data:

```
SELECT * FROM UAct
```

UserID	PageViews	Duration	Sign	Version
4324182021466249494	5	146	1	1
4324182021466249494	5	146	-1	1
4324182021466249494	6	185	1	2

What do we see here and where are the collapsed parts?

We created two data parts using two `INSERT` queries. The `SELECT` query was performed in two threads, and the result is a random order of rows.

Collapsing did not occur because the data parts have not been merged yet. ClickHouse merges data parts at an unknown point in time which we cannot predict.

This is why we need aggregation:

```
SELECT
  UserID,
  sum(PageViews * Sign) AS PageViews,
  sum(Duration * Sign) AS Duration,
  Version
FROM UAct
GROUP BY UserID, Version
HAVING sum(Sign) > 0
```

UserID	PageViews	Duration	Version
4324182021466249494	6	185	2

If we don't need aggregation and want to force collapsing, we can use the `FINAL` modifier for the `FROM` clause.

```
SELECT * FROM UAct FINAL
```

UserID	PageViews	Duration	Sign	Version
4324182021466249494	6	185	1	2

This is a very inefficient way to select data. Don't use it for large tables.

GraphiteMergeTree

This engine is designed for thinning and aggregating/averaging (rollup) [Graphite](#) data. It may be helpful to developers who want to use ClickHouse as a data store for Graphite.

You can use any ClickHouse table engine to store the Graphite data if you don't need rollup, but if you need a rollup use `GraphiteMergeTree`. The engine reduces the volume of storage and increases the efficiency of queries from Graphite.

The engine inherits properties from [MergeTree](#).

Creating a Table


```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    Path String,
    Time DateTime,
    Value <Numeric_type>,
    Version <Numeric_type>
    ...
) ENGINE = GraphiteMergeTree(config_section)
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

For a description of request parameters, see [request description](#).

A table for the Graphite data should have the following columns:

- Column with the metric name (Graphite sensor). Data type: `String`.
- Column with the time of measuring the metric. Data type: `DateTime`.
- Column with the value of the metric. Data type: any numeric.
- Column with the version of the metric. Data type: any numeric.

ClickHouse saves the rows with the highest version or the last written if versions are the same. Other rows are deleted during the merge of data parts.

The names of these columns should be set in the rollup configuration.

GraphiteMergeTree parameters

- `config_section` — Name of the section in the configuration file, where are the rules of rollup set.

Query clauses

When creating a `GraphiteMergeTree` table, the same [clauses](#) are required, as when creating a `MergeTree` table.

▼ Deprecated Method for Creating a Table

Attention

Do not use this method in new projects and, if possible, switch the old projects to the method described above.

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    EventDate Date,
    Path String,
    Time DateTime,
    Value <Numeric_type>,
    Version <Numeric_type>
    ...
) ENGINE [=] GraphiteMergeTree(date-column [, sampling_expression], (primary, key), index_granularity, config_section)
```

All of the parameters excepting `config_section` have the same meaning as in `MergeTree`.

- `config_section` — Name of the section in the configuration file, where are the rules of rollup set.

Rollup configuration

The settings for rollup are defined by the [graphite_rollup](#) parameter in the server configuration. The name of the parameter could be any. You can create several configurations and use them for different tables.

Rollup configuration structure:

```

required-columns
pattern
  regexp
  function
pattern
  regexp
  age + precision
  ...
pattern
  regexp
  function
  age + precision
  ...
pattern
  ...
default
  function
  age + precision
  ...

```

Important: The order of patterns should be next:

1. Patterns *without* function *or* retention.
2. Patterns *with* both function *and* retention.
3. Pattern default.

When processing a row, ClickHouse checks the rules in the `pattern` sections. Each of `pattern` (including `default`) sections could contain `function` parameter for aggregation, `retention` parameters or both. If the metric name matches the `regexp`, the rules from the `pattern` section (or sections) are applied; otherwise, the rules from the `default` section are used.

Fields for `pattern` and `default` sections:

- `regexp` – A pattern for the metric name.
- `age` – The minimum age of the data in seconds.
- `precision` – How precisely to define the age of the data in seconds. Should be a divisor for 86400 (seconds in a day).
- `function` – The name of the aggregating function to apply to data whose age falls within the range `[age, age + precision]`.

The `required-columns`:

- `path_column_name` — Column with the metric name (Graphite sensor).
- `time_column_name` — Column with the time of measuring the metric.
- `value_column_name` — Column with the value of the metric at the time set in `time_column_name`.
- `version_column_name` — Column with the version of the metric.

Example of settings:

```

<graphite_rollup>
  <path_column_name>Path</path_column_name>
  <time_column_name>Time</time_column_name>
  <value_column_name>Value</value_column_name>
  <version_column_name>Version</version_column_name>
  <pattern>
    <regexp>click_cost</regexp>
    <function>any</function>
    <retention>
      <age>0</age>
      <precision>5</precision>
    </retention>
  </retention>

```

```

    <age>86400</age>
    <precision>60</precision>
  </retention>
</pattern>
<default>
  <function>max</function>
  <retention>
    <age>0</age>
    <precision>60</precision>
  </retention>
  <retention>
    <age>3600</age>
    <precision>300</precision>
  </retention>
  <retention>
    <age>86400</age>
    <precision>3600</precision>
  </retention>
</default>
</graphite_rollup>

```

Log Engine Family

These engines were developed for scenarios when you need to write many tables with the small amount of data (less than 1 million rows).

Engines of the family:

- [StripeLog](#)
- [Log](#)
- [TinyLog](#)

Common properties

Engines:

- Store data on a disk.
- Append data to the end of file when writing.
- Do not support **mutation** operations.
- Do not support indexes.

This means that `SELECT` queries for ranges of data are not efficient.

- Do not write data atomically.

You can get a table with corrupted data if something breaks the write operation, for example, abnormal server shutdown.

Differences

The `Log` and `StripeLog` engines support:

- Locks for concurrent data access.

During `INSERT` query the table is locked, and other queries for reading and writing data both wait for unlocking. If there are no writing data queries, any number of reading data queries can be performed concurrently.

- Parallel reading of data.

When reading data ClickHouse uses multiple threads. Each thread processes separated data block.

The `Log` engine uses the separate file for each column of the table. The `StripeLog` stores all the data in one file.

Thus the `StripeLog` engine uses fewer descriptors in the operating system, but the `Log` engine provides a more efficient reading of the data.

The `TinyLog` engine is the simplest in the family and provides the poorest functionality and lowest efficiency. The `TinyLog` engine does not support a parallel reading and concurrent access and stores each column in a separate file. It reads the data slower than both other engines with parallel reading, and it uses almost as many descriptors as the `Log` engine. You can use it in simple low-load scenarios.

StripeLog

This engine belongs to the family of log engines. See the common properties of log engines and their differences in the [Log Engine Family](#) article.

Use this engine in scenarios when you need to write many tables with a small amount of data (less than 1 million rows).

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    column1_name [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    column2_name [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = StripeLog
```

See the detailed description of the [CREATE TABLE](#) query.

Writing the Data

The `StripeLog` engine stores all the columns in one file. For each `INSERT` query, ClickHouse appends the data block to the end of a table file, writing columns one by one.

For each table ClickHouse writes the files:

- `data.bin` — Data file.
- `index.mrk` — File with marks. Marks contain offsets for each column of each data block inserted.

The `StripeLog` engine does not support the `ALTER UPDATE` and `ALTER DELETE` operations.

Reading the Data

The file with marks allows ClickHouse to parallelize the reading of data. This means that a `SELECT` query returns rows in an unpredictable order. Use the `ORDER BY` clause to sort rows.

Example of Use

Creating a table:

```
CREATE TABLE stripe_log_table
(
    timestamp DateTime,
    message_type String,
    message String
)
ENGINE = StripeLog
```

Inserting data:

```
INSERT INTO stripe_log_table VALUES (now(),'REGULAR','The first regular message')
INSERT INTO stripe_log_table VALUES (now(),'REGULAR','The second regular message'),(now(),'WARNING','The first warning message')
```

We used two `INSERT` queries to create two data blocks inside the `data.bin` file.

ClickHouse uses multiple threads when selecting data. Each thread reads a separate data block and returns resulting rows independently as it finishes. As a result, the order of blocks of rows in the output does not match the order of the same blocks in the input in most cases. For example:

```
SELECT * FROM stripe_log_table
```

timestamp	message_type	message
2019-01-18 14:27:32	REGULAR	The second regular message
2019-01-18 14:34:53	WARNING	The first warning message
2019-01-18 14:23:43	REGULAR	The first regular message

Sorting the results (ascending order by default):

```
SELECT * FROM stripe_log_table ORDER BY timestamp
```

timestamp	message_type	message
2019-01-18 14:23:43	REGULAR	The first regular message
2019-01-18 14:27:32	REGULAR	The second regular message
2019-01-18 14:34:53	WARNING	The first warning message

Log

Engine belongs to the family of log engines. See the common properties of log engines and their differences in the [Log Engine Family](#) article.

Log differs from [TinyLog](#) in that a small file of "marks" resides with the column files. These marks are written on every data block and contain offsets that indicate where to start reading the file in order to skip the specified number of rows. This makes it possible to read table data in multiple threads.

For concurrent data access, the read operations can be performed simultaneously, while write operations block reads and each other.

The Log engine does not support indexes. Similarly, if writing to a table failed, the table is broken, and reading from it returns an error. The Log engine is appropriate for temporary data, write-once tables, and for testing or demonstration purposes.

TinyLog

Engine belongs to the family of log engines. See the common properties of log engines and their differences in the [Log Engine Family](#) article.

The simplest table engine, which stores data on a disk.

Each column is stored in a separate compressed file.

When writing, data is appended to the end of files.

Concurrent data access is not restricted in any way:

- If you are simultaneously reading from a table and writing to it in a different query, the read operation will complete with an error.
- If you are writing to a table in multiple queries simultaneously, the data will be broken.

The typical way to use this table is write-once: first just write the data one time, then read it as many times as needed.

Queries are executed in a single stream. In other words, this engine is intended for relatively small tables (recommended up to 1,000,000 rows).

It makes sense to use this table engine if you have many small tables, since it is simpler than the Log engine (fewer files need to be opened).

The situation when you have a large number of small tables guarantees poor productivity, but may already be used when working with another DBMS, and you may find it easier to switch to using TinyLog types of tables.

Indexes are not supported.

In Yandex.Metrica, TinyLog tables are used for intermediary data that is processed in small batches.

Kafka

This engine works with [Apache Kafka](#).

Kafka lets you:

- Publish or subscribe to data flows.
- Organize fault-tolerant storage.
- Process streams as they become available.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = Kafka()
SETTINGS
    kafka_broker_list = 'host:port',
    kafka_topic_list = 'topic1,topic2,...',
    kafka_group_name = 'group_name',
    kafka_format = 'data_format',[,]
    [kafka_row_delimiter = 'delimiter_symbol',]
    [kafka_schema = '',]
    [kafka_num_consumers = N,]
    [kafka_skip_broken_messages = <0|1>]
```

Required parameters:

- `kafka_broker_list` – A comma-separated list of brokers (for example, `localhost:9092`).
- `kafka_topic_list` – A list of Kafka topics.
- `kafka_group_name` – A group of Kafka consumers. Reading margins are tracked for each group separately. If you don't want messages to be duplicated in the cluster, use the same group name everywhere.
- `kafka_format` – Message format. Uses the same notation as the SQL `FORMAT` function, such as `JSONEachRow`. For more information, see the [Formats](#) section.

Optional parameters:

- `kafka_row_delimiter` – Delimiter character, which ends the message.
- `kafka_schema` – Parameter that must be used if the format requires a schema definition. For example, [Cap'n Proto](#) requires the path to the schema file and the name of the root `schema.capnp:Message` object.
- `kafka_num_consumers` – The number of consumers per table. Default: `1`. Specify more consumers if the throughput of one consumer is insufficient. The total number of consumers should not exceed the number of partitions in the topic, since only one consumer can be assigned per partition.
- `kafka_skip_broken_messages` – Kafka message parser mode. If `kafka_skip_broken_messages = 1` then the engine skips the Kafka messages that can't be parsed (a message equals a row of data).

Examples:

```
CREATE TABLE queue (
  timestamp UInt64,
  level String,
  message String
) ENGINE = Kafka('localhost:9092', 'topic', 'group1', 'JSONEachRow');
```

```
SELECT * FROM queue LIMIT 5;
```

```
CREATE TABLE queue2 (
  timestamp UInt64,
  level String,
  message String
) ENGINE = Kafka SETTINGS kafka_broker_list = 'localhost:9092',
                  kafka_topic_list = 'topic',
                  kafka_group_name = 'group1',
                  kafka_format = 'JSONEachRow',
                  kafka_num_consumers = 4;
```

```
CREATE TABLE queue2 (
  timestamp UInt64,
  level String,
  message String
) ENGINE = Kafka('localhost:9092', 'topic', 'group1')
  SETTINGS kafka_format = 'JSONEachRow',
          kafka_num_consumers = 4;
```

▼ Deprecated Method for Creating a Table

Attention

Do not use this method in new projects. If possible, switch old projects to the method described above.

```
Kafka(kafka_broker_list, kafka_topic_list, kafka_group_name, kafka_format
      [, kafka_row_delimiter, kafka_schema, kafka_num_consumers, kafka_skip_broken_messages])
```

Description

The delivered messages are tracked automatically, so each message in a group is only counted once. If you want to get the data twice, then create a copy of the table with another group name.

Groups are flexible and synced on the cluster. For instance, if you have 10 topics and 5 copies of a table in a cluster, then each copy gets 2 topics. If the number of copies changes, the topics are redistributed across the copies automatically. Read more about this at <http://kafka.apache.org/intro>.

`SELECT` is not particularly useful for reading messages (except for debugging), because each message can be read only once. It is more practical to create real-time threads using materialized views. To do this:

1. Use the engine to create a Kafka consumer and consider it a data stream.
2. Create a table with the desired structure.
3. Create a materialized view that converts data from the engine and puts it into a previously created table.

When the `MATERIALIZED VIEW` joins the engine, it starts collecting data in the background. This allows you to continually receive messages from Kafka and convert them to the required format using `SELECT`.

Example:

```
CREATE TABLE queue (
  timestamp UInt64,
  level String,
  message String
```

```

message String
) ENGINE = Kafka('localhost:9092', 'topic', 'group1', 'JSONEachRow');

CREATE TABLE daily (
    day Date,
    level String,
    total UInt64
) ENGINE = SummingMergeTree(day, (day, level), 8192);

CREATE MATERIALIZED VIEW consumer TO daily
AS SELECT toDate(toDateTime(timestamp)) AS day, level, count() as total
FROM queue GROUP BY day, level;

SELECT level, sum(total) FROM daily GROUP BY level;

```

To improve performance, received messages are grouped into blocks the size of `max_insert_block_size`. If the block wasn't formed within `stream_flush_interval_ms` milliseconds, the data will be flushed to the table regardless of the completeness of the block.

To stop receiving topic data or to change the conversion logic, detach the materialized view:

```

DETACH TABLE consumer;
ATTACH MATERIALIZED VIEW consumer;

```

If you want to change the target table by using `ALTER`, we recommend disabling the material view to avoid discrepancies between the target table and the data from the view.

Configuration

Similar to `GraphiteMergeTree`, the Kafka engine supports extended configuration using the ClickHouse config file. There are two configuration keys that you can use: global (`kafka`) and topic-level (`kafka_*`). The global configuration is applied first, and then the topic-level configuration is applied (if it exists).

```

<!-- Global configuration options for all tables of Kafka engine type -->
<kafka>
  <debug>cgrp</debug>
  <auto_offset_reset>smallest</auto_offset_reset>
</kafka>

<!-- Configuration specific for topic "logs" -->
<kafka_logs>
  <retry_backoff_ms>250</retry_backoff_ms>
  <fetch_min_bytes>100000</fetch_min_bytes>
</kafka_logs>

```

For a list of possible configuration options, see the [librdkafka configuration reference](#). Use the underscore (`_`) instead of a dot in the ClickHouse configuration. For example, `check.crcs=true` will be `<check_crcs>true</check_crcs>`.

MySQL

The MySQL engine allows you to perform `SELECT` queries on data that is stored on a remote MySQL server.

Creating a Table

```

CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1] [TTL expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2] [TTL expr2],
    ...

```



```
INDEX index_name1 expr1 TYPE type1(...) GRANULARITY value1,  
INDEX index_name2 expr2 TYPE type2(...) GRANULARITY value2  
) ENGINE = MySQL('host:port', 'database', 'table', 'user', 'password'[, replace_query, 'on_duplicate_clause']);
```

See the detailed description of the [CREATE TABLE](#) query.

The table structure can be not the same as the original MySQL table structure:

- Names of columns should be the same as in the original MySQL table, but you can use just some of these columns in any order.
- Types of columns may differ from the types in the original MySQL table. ClickHouse tries to [cast](#) values into the ClickHouse data types.

Engine Parameters

- `host:port` — MySQL server address.
- `database` — Remote database name.
- `table` — Remote table name.
- `user` — MySQL user.
- `password` — User password.
- `replace_query` — Flag that sets query substitution `INSERT INTO` to `REPLACE INTO`. If `replace_query=1`, the query is replaced.
- `on_duplicate_clause` — The `ON DUPLICATE KEY on_duplicate_clause` expression that is added to the `INSERT` query.

Example: `INSERT INTO t (c1,c2) VALUES ('a', 2) ON DUPLICATE KEY UPDATE c2 = c2 + 1`, where `on_duplicate_clause` is `UPDATE c2 = c2 + 1`. See MySQL documentation to find which `on_duplicate_clause` you can use with `ON DUPLICATE KEY` clause.

To specify `on_duplicate_clause` you need to pass `0` to the `replace_query` parameter. If you simultaneously pass `replace_query = 1` and `on_duplicate_clause`, ClickHouse generates an exception.

At this time, simple `WHERE` clauses such as `=, !=, >, >=, <, <=` are executed on the MySQL server.

The rest of the conditions and the `LIMIT` sampling constraint are executed in ClickHouse only after the query to MySQL finishes.

Usage Example

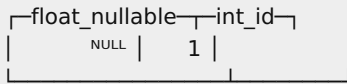
Table in MySQL:

```
mysql> CREATE TABLE `test`.`test` (  
-> `int_id` INT NOT NULL AUTO_INCREMENT,  
-> `int_nullable` INT NULL DEFAULT NULL,  
-> `float` FLOAT NOT NULL,  
-> `float_nullable` FLOAT NULL DEFAULT NULL,  
-> PRIMARY KEY (`int_id`));  
Query OK, 0 rows affected (0,09 sec)  
  
mysql> insert into test (`int_id`, `float`) VALUES (1,2);  
Query OK, 1 row affected (0,00 sec)  
  
mysql> select * from test;  
+-----+-----+-----+-----+  
| int_id | int_nullable | float | float_nullable |  
+-----+-----+-----+-----+  
| 1 | NULL | 2 | NULL |  
+-----+-----+-----+-----+  
1 row in set (0,00 sec)
```

Table in ClickHouse, getting data from the MySQL table:

```
CREATE TABLE mysql_table
(
  `float_nullable` Nullable(Float32),
  `int_id` Int32
)
ENGINE = MySQL('localhost:3306', 'test', 'test', 'bayonet', '123')
```

```
SELECT * FROM mysql_table6
```



See Also

- [The 'mysql' table function](#)
- [Using MySQL as a source of external dictionary](#)

Distributed

The Distributed engine does not store data itself, but allows distributed query processing on multiple servers.

Reading is automatically parallelized. During a read, the table indexes on remote servers are used, if there are any.

The Distributed engine accepts parameters: the cluster name in the server's config file, the name of a remote database, the name of a remote table, and (optionally) a sharding key.

Example:

```
Distributed(logs, default, hits[, sharding_key])
```

Data will be read from all servers in the 'logs' cluster, from the default.hits table located on every server in the cluster.

Data is not only read, but is partially processed on the remote servers (to the extent that this is possible). For example, for a query with GROUP BY, data will be aggregated on remote servers, and the intermediate states of aggregate functions will be sent to the requestor server. Then data will be further aggregated.

Instead of the database name, you can use a constant expression that returns a string. For example: `currentDatabase()`.

logs - The cluster name in the server's config file.

Clusters are set like this:

```
<remote_servers>
<logs>
  <shard>
    <!-- Optional. Shard weight when writing data. Default: 1. -->
    <weight>1</weight>
    <!-- Optional. Whether to write data to just one of the replicas. Default: false (write data to all replicas). -->
    <internal_replication>false</internal_replication>
    <replica>
      <host>example01-01-1</host>
      <port>9000</port>
    </replica>
    <replica>
      <host>example01-01-2</host>
      <port>9000</port>
```

```

    </replica>
  </shard>
  <shard>
    <weight>2</weight>
    <internal_replication>false</internal_replication>
    <replica>
      <host>example01-02-1</host>
      <port>9000</port>
    </replica>
    <replica>
      <host>example01-02-2</host>
      <secure>1</secure>
      <port>9440</port>
    </replica>
  </shard>
</logs>
</remote_servers>

```

Here a cluster is defined with the name 'logs' that consists of two shards, each of which contains two replicas. Shards refer to the servers that contain different parts of the data (in order to read all the data, you must access all the shards).

Replicas are duplicating servers (in order to read all the data, you can access the data on any one of the replicas).

Cluster names must not contain dots.

The parameters `host`, `port`, and optionally `user`, `password`, `secure`, `compression` are specified for each server:

- `host` – The address of the remote server. You can use either the domain or the IPv4 or IPv6 address. If you specify the domain, the server makes a DNS request when it starts, and the result is stored as long as the server is running. If the DNS request fails, the server doesn't start. If you change the DNS record, restart the server.
- `port` – The TCP port for messenger activity ('tcp_port' in the config, usually set to 9000). Do not confuse it with http_port.
- `user` – Name of the user for connecting to a remote server. Default value: default. This user must have access to connect to the specified server. Access is configured in the users.xml file. For more information, see the section "Access rights".
- `password` – The password for connecting to a remote server (not masked). Default value: empty string.
- `secure` – Use ssl for connection, usually you also should define `port` = 9440. Server should listen on 9440 and have correct certificates.
- `compression` – Use data compression. Default value: true.

When specifying replicas, one of the available replicas will be selected for each of the shards when reading. You can configure the algorithm for load balancing (the preference for which replica to access) – see the 'load_balancing' setting.

If the connection with the server is not established, there will be an attempt to connect with a short timeout. If the connection failed, the next replica will be selected, and so on for all the replicas. If the connection attempt failed for all the replicas, the attempt will be repeated the same way, several times.

This works in favor of resiliency, but does not provide complete fault tolerance: a remote server might accept the connection, but might not work, or work poorly.

You can specify just one of the shards (in this case, query processing should be called remote, rather than distributed) or up to any number of shards. In each shard, you can specify from one to any number of replicas. You can specify a different number of replicas for each shard.

You can specify as many clusters as you wish in the configuration.

To view your clusters, use the 'system.clusters' table.

The Distributed engine allows working with a cluster like a local server. However, the cluster is inextensible: you

must write its configuration in the server config file (even better, for all the cluster's servers).

There is no support for Distributed tables that look at other Distributed tables (except in cases when a Distributed table only has one shard). As an alternative, make the Distributed table look at the "final" tables.

The Distributed engine requires writing clusters to the config file. Clusters from the config file are updated on the fly, without restarting the server. If you need to send a query to an unknown set of shards and replicas each time, you don't need to create a Distributed table – use the 'remote' table function instead. See the section "Table functions".

There are two methods for writing data to a cluster:

First, you can define which servers to write which data to, and perform the write directly on each shard. In other words, perform INSERT in the tables that the distributed table "looks at".

This is the most flexible solution – you can use any sharding scheme, which could be non-trivial due to the requirements of the subject area.

This is also the most optimal solution, since data can be written to different shards completely independently.

Second, you can perform INSERT in a Distributed table. In this case, the table will distribute the inserted data across servers itself.

In order to write to a Distributed table, it must have a sharding key set (the last parameter). In addition, if there is only one shard, the write operation works without specifying the sharding key, since it doesn't have any meaning in this case.

Each shard can have a weight defined in the config file. By default, the weight is equal to one. Data is distributed across shards in the amount proportional to the shard weight. For example, if there are two shards and the first has a weight of 9 while the second has a weight of 10, the first will be sent $9 / 19$ parts of the rows, and the second will be sent $10 / 19$.

Each shard can have the 'internal_replication' parameter defined in the config file.

If this parameter is set to 'true', the write operation selects the first healthy replica and writes data to it. Use this alternative if the Distributed table "looks at" replicated tables. In other words, if the table where data will be written is going to replicate them itself.

If it is set to 'false' (the default), data is written to all replicas. In essence, this means that the Distributed table replicates data itself. This is worse than using replicated tables, because the consistency of replicas is not checked, and over time they will contain slightly different data.

To select the shard that a row of data is sent to, the sharding expression is analyzed, and its remainder is taken from dividing it by the total weight of the shards. The row is sent to the shard that corresponds to the half-interval of the remainders from 'prev_weight' to 'prev_weights + weight', where 'prev_weights' is the total weight of the shards with the smallest number, and 'weight' is the weight of this shard. For example, if there are two shards, and the first has a weight of 9 while the second has a weight of 10, the row will be sent to the first shard for the remainders from the range $[0, 9)$, and to the second for the remainders from the range $[9, 19)$.

The sharding expression can be any expression from constants and table columns that returns an integer. For example, you can use the expression 'rand()' for random distribution of data, or 'UserID' for distribution by the remainder from dividing the user's ID (then the data of a single user will reside on a single shard, which simplifies running IN and JOIN by users). If one of the columns is not distributed evenly enough, you can wrap it in a hash function: `intHash64(UserID)`.

A simple remainder from division is a limited solution for sharding and isn't always appropriate. It works for medium and large volumes of data (dozens of servers), but not for very large volumes of data (hundreds of servers or more). In the latter case, use the sharding scheme required by the subject area, rather than using entries in Distributed tables.

SELECT queries are sent to all the shards, and work regardless of how data is distributed across the shards (they can be distributed completely randomly). When you add a new shard, you don't have to transfer the old data to it. You can write new data with a heavier weight – the data will be distributed slightly unevenly, but queries will work correctly and efficiently.

You should be concerned about the sharding scheme in the following cases:

- Queries are used that require joining data (IN or JOIN) by a specific key. If data is sharded by this key, you can use local IN or JOIN instead of GLOBAL IN or GLOBAL JOIN, which is much more efficient.
- A large number of servers is used (hundreds or more) with a large number of small queries (queries of individual clients - websites, advertisers, or partners). In order for the small queries to not affect the entire cluster, it makes sense to locate data for a single client on a single shard. Alternatively, as we've done in Yandex.Metrica, you can set up bi-level sharding: divide the entire cluster into "layers", where a layer may consist of multiple shards. Data for a single client is located on a single layer, but shards can be added to a layer as necessary, and data is randomly distributed within them. Distributed tables are created for each layer, and a single shared distributed table is created for global queries.

Data is written asynchronously. For an INSERT to a Distributed table, the data block is just written to the local file system. The data is sent to the remote servers in the background as soon as possible. You should check whether data is sent successfully by checking the list of files (data waiting to be sent) in the table directory: `/var/lib/clickhouse/data/database/table/`.

If the server ceased to exist or had a rough restart (for example, after a device failure) after an INSERT to a Distributed table, the inserted data might be lost. If a damaged data part is detected in the table directory, it is transferred to the 'broken' subdirectory and no longer used.

When the `max_parallel_replicas` option is enabled, query processing is parallelized across all replicas within a single shard. For more information, see the section "Settings, `max_parallel_replicas`".

External Data for Query Processing

ClickHouse allows sending a server the data that is needed for processing a query, together with a SELECT query. This data is put in a temporary table (see the section "Temporary tables") and can be used in the query (for example, in IN operators).

For example, if you have a text file with important user identifiers, you can upload it to the server along with a query that uses filtration by this list.

If you need to run more than one query with a large volume of external data, don't use this feature. It is better to upload the data to the DB ahead of time.

External data can be uploaded using the command-line client (in non-interactive mode), or using the HTTP interface.

In the command-line client, you can specify a parameters section in the format

```
--external --file=... [--name=...] [--format=...] [--types=...|--structure=...]
```

You may have multiple sections like this, for the number of tables being transmitted.

--external - Marks the beginning of a clause.

--file - Path to the file with the table dump, or -, which refers to stdin.

Only a single table can be retrieved from stdin.

The following parameters are optional: **--name** - Name of the table. If omitted, `_data` is used.

--format - Data format in the file. If omitted, `TabSeparated` is used.

One of the following parameters is required: **--types** - A list of comma-separated column types. For example: `UInt64,String`. The columns will be named `_1, _2, ...`

--structure - The table structure in the format `UserID UInt64, URL String`. Defines the column names and types.

The files specified in 'file' will be parsed by the format specified in 'format', using the data types specified in 'types' or 'structure'. The table will be uploaded to the server and accessible there as a temporary table with the name in 'name'.

Examples:

```
echo -ne "\n2\n3\n" | clickhouse-client --query="SELECT count() FROM test.visits WHERE TrafficSourceID IN _data" --external --file=- --types=Int8
849897
cat /etc/passwd | sed 's:/\t/g' | clickhouse-client --query="SELECT shell, count() AS c FROM passwd GROUP BY shell ORDER BY c DESC" --external --file=- --name=passwd --structure='login String, unused String, uid UInt16, gid UInt16, comment String, home String, shell String'
/bin/sh 20
/bin/false 5
/bin/bash 4
/usr/sbin/nologin 1
/bin/sync 1
```

When using the HTTP interface, external data is passed in the multipart/form-data format. Each table is transmitted as a separate file. The table name is taken from the file name. The 'query_string' is passed the parameters 'name_format', 'name_types', and 'name_structure', where 'name' is the name of the table that these parameters correspond to. The meaning of the parameters is the same as when using the command-line client.

Example:

```
cat /etc/passwd | sed 's:/\t/g' > passwd.tsv

curl -F 'passwd=@passwd.tsv;' 'http://localhost:8123/?query=SELECT+shell,+count()+AS+c+FROM+passwd+GROUP+BY+shell+ORDER+BY+c+DESC&passwd_structure=login+String,+unused+String,+uid+UInt16,+gid+UInt16,+comment+String,+home+String,+shell+String'
/bin/sh 20
/bin/false 5
/bin/bash 4
/usr/sbin/nologin 1
/bin/sync 1
```

For distributed query processing, the temporary tables are sent to all the remote servers.

Dictionary

The Dictionary engine displays the **dictionary** data as a ClickHouse table.

As an example, consider a dictionary of **products** with the following configuration:

```
<ictionaries>
<dictionary>
  <name>products</name>
  <source>
    <odbc>
      <table>products</table>
      <connection_string>DSN=some-db-server</connection_string>
    </odbc>
  </source>
  <lifetime>
    <min>300</min>
    <max>360</max>
  </lifetime>
  <layout>
    <flat/>
  </layout>
  <structure>
    <id>
      <name>product_id</name>
```

```

</id>
<attribute>
  <name>title</name>
  <type>String</type>
  <null_value></null_value>
</attribute>
</structure>
</dictionary>
</dictionaries>

```

Query the dictionary data:

```

select name, type, key, attribute.names, attribute.types, bytes_allocated, element_count, source from system.dictionaries
where name = 'products';

```

```

SELECT
  name,
  type,
  key,
  attribute.names,
  attribute.types,
  bytes_allocated,
  element_count,
  source
FROM system.dictionaries
WHERE name = 'products'

```

name	type	key	attribute.names	attribute.types	bytes_allocated	element_count	source
products	Flat	UInt64	['title']	['String']	23065376	175032	ODBC: .products

You can use the **dictGet*** function to get the dictionary data in this format.

This view isn't helpful when you need to get raw data, or when performing a **JOIN** operation. For these cases, you can use the **Dictionary** engine, which displays the dictionary data in a table.

Syntax:

```

CREATE TABLE %table_name% (%fields%) engine = Dictionary(%dictionary_name%)`

```

Usage example:

```

create table products (product_id UInt64, title String) Engine = Dictionary(products);

CREATE TABLE products
(
  product_id UInt64,
  title String,
)
ENGINE = Dictionary(products)

```

Ok.

0 rows in set. Elapsed: 0.004 sec.

Take a look at what's in the table.

```
select * from products limit 1;
```

```
SELECT *  
FROM products  
LIMIT 1
```

product_id	title
152689	Some item

1 rows in set. Elapsed: 0.006 sec.

Merge

The `Merge` engine (not to be confused with `MergeTree`) does not store data itself, but allows reading from any number of other tables simultaneously.

Reading is automatically parallelized. Writing to a table is not supported. When reading, the indexes of tables that are actually being read are used, if they exist.

The `Merge` engine accepts parameters: the database name and a regular expression for tables.

Example:

```
Merge(hits, '^WatchLog')
```

Data will be read from the tables in the `hits` database that have names that match the regular expression `'^WatchLog'`.

Instead of the database name, you can use a constant expression that returns a string. For example, `currentDatabase()`.

Regular expressions — `re2` (supports a subset of PCRE), case-sensitive.

See the notes about escaping symbols in regular expressions in the "match" section.

When selecting tables to read, the `Merge` table itself will not be selected, even if it matches the regex. This is to avoid loops.

It is possible to create two `Merge` tables that will endlessly try to read each others' data, but this is not a good idea.

The typical way to use the `Merge` engine is for working with a large number of `TinyLog` tables as if with a single table.

Example 2:

Let's say you have a old table (`WatchLog_old`) and decided to change partitioning without moving data to a new table (`WatchLog_new`) and you need to see data from both tables.

```
CREATE TABLE WatchLog_old(date Date, UserId Int64, EventType String, Cnt UInt64)  
ENGINE=MergeTree(date, (UserId, EventType), 8192);  
INSERT INTO WatchLog_old VALUES ('2018-01-01', 1, 'hit', 3);
```

```
CREATE TABLE WatchLog_new(date Date, UserId Int64, EventType String, Cnt UInt64)  
ENGINE=MergeTree PARTITION BY date ORDER BY (UserId, EventType) SETTINGS index_granularity=8192;  
INSERT INTO WatchLog_new VALUES ('2018-01-02', 2, 'hit', 3);
```

```
CREATE TABLE WatchLog as WatchLog_old ENGINE=Merge(currentDatabase(), '^WatchLog');
```



```
SELECT *
FROM WatchLog
```

date	UserId	EventType	Cnt
2018-01-01	1	hit	3
2018-01-02	2	hit	3

Virtual Columns

Virtual columns are columns that are provided by the table engine, regardless of the table definition. In other words, these columns are not specified in `CREATE TABLE`, but they are accessible for `SELECT`.

Virtual columns differ from normal columns in the following ways:

- They are not specified in table definitions.
- Data can't be added to them with `INSERT`.
- When using `INSERT` without specifying the list of columns, virtual columns are ignored.
- They are not selected when using the asterisk (`SELECT *`).
- Virtual columns are not shown in `SHOW CREATE TABLE` and `DESC TABLE` queries.

The `Merge` type table contains a virtual `_table` column of the `String` type. (If the table already has a `_table` column, the virtual column is called `_table1`; if you already have `_table1`, it's called `_table2`, and so on.) It contains the name of the table that data was read from.

If the `WHERE/PREWHERE` clause contains conditions for the `_table` column that do not depend on other table columns (as one of the conjunction elements, or as an entire expression), these conditions are used as an index. The conditions are performed on a data set of table names to read data from, and the read operation will be performed from only those tables that the condition was triggered on.

File(InputFormat)

The data source is a file that stores data in one of the supported input formats (TabSeparated, Native, etc.).

Usage examples:

- Data export from ClickHouse to file.
- Convert data from one format to another.
- Updating data in ClickHouse via editing a file on a disk.

Usage in ClickHouse Server

File(Format)

`Format` should be supported for either `INSERT` and `SELECT`. For the full list of supported formats see [Formats](#).

ClickHouse does not allow to specify filesystem path for `File`. It will use folder defined by `path` setting in server configuration.

When creating table using `File(Format)` it creates empty subdirectory in that folder. When data is written to that table, it's put into `data.Format` file in that subdirectory.

You may manually create this subfolder and file in server filesystem and then `ATTACH` it to table information with matching name, so you can query data from that file.

Warning

Be careful with this functionality, because ClickHouse does not keep track of external changes to such files. The result of simultaneous writes via ClickHouse and outside of ClickHouse is undefined.

Inserts of simultaneous writes via ClickHouse and output of ClickHouse is asynchronous.

Example:

1. Set up the `file_engine_table` table:

```
CREATE TABLE file_engine_table (name String, value UInt32) ENGINE=File(TabSeparated)
```

By default ClickHouse will create folder `/var/lib/clickhouse/data/default/file_engine_table`.

2. Manually create `/var/lib/clickhouse/data/default/file_engine_table/data.TabSeparated` containing:

```
$ cat data.TabSeparated
one 1
two 2
```

3. Query the data:

```
SELECT * FROM file_engine_table
```

name	value
one	1
two	2

Usage in Clickhouse-local

In **clickhouse-local** File engine accepts file path in addition to `Format`. Default input/output streams can be specified using numeric or human-readable names like `0` or `stdin`, `1` or `stdout`.

Example:

```
$ echo -e "1,2\n3,4" | clickhouse-local -q "CREATE TABLE table (a Int64, b Int64) ENGINE = File(CSV, stdin); SELECT a, b FROM table; DROP TABLE table"
```

Details of Implementation

- Reads can be parallel, but not writes
- Not supported:
 - `ALTER`
 - `SELECT ... SAMPLE`
 - Indices
 - Replication

Null

When writing to a Null table, data is ignored. When reading from a Null table, the response is empty.

However, you can create a materialized view on a Null table. So the data written to the table will end up in the view.

Set

A data set that is always in RAM. It is intended for use on the right side of the IN operator (see the section "IN operators").

You can use INSERT to insert data in the table. New elements will be added to the data set, while duplicates will be ignored.

But you can't perform SELECT from the table. The only way to retrieve data is by using it in the right half of the IN operator.

Data is always located in RAM. For INSERT, the blocks of inserted data are also written to the directory of tables on the disk. When starting the server, this data is loaded to RAM. In other words, after restarting, the data remains in place.

For a rough server restart, the block of data on the disk might be lost or damaged. In the latter case, you may need to manually delete the file with damaged data.

Join

A prepared data structure for JOIN that is always located in RAM.

```
Join(ANY|ALL, LEFT|INNER, k1[, k2, ...])
```

Engine parameters: `ANY|ALL` – strictness; `LEFT|INNER` – type. For more information, see the [JOIN Clause](#) section. These parameters are set without quotes and must match the JOIN that the table will be used for. `k1`, `k2`, ... are the key columns from the USING clause that the join will be made on.

The table can't be used for GLOBAL JOINS.

You can use INSERT to add data to the table, similar to the Set engine. For ANY, data for duplicated keys will be ignored. For ALL, it will be counted.

You can't perform SELECT directly from the table. There are two ways to retrieve data from table with Join engine * use it as the "right-hand" table for JOIN * use `joinGet` function, which allows to extract data from Join table with dictionary-like syntax.

Storing data on the disk is the same as for the Set engine.

You can customize several settings when creating JOIN table with the following syntax:

```
CREATE TABLE join_any_left_null ( ... ) ENGINE = Join(ANY, LEFT, ...) SETTINGS join_use_nulls = 1;
```

The following setting are supported by JOIN engine: * `join_use_nulls` * `max_rows_in_join` * `max_bytes_in_join` * `join_overflow_mode` * `join_any_take_last_row`

URL(URL, Format)

Manages data on a remote HTTP/HTTPS server. This engine is similar to the [File](#) engine.

Using the engine in the ClickHouse server

The `format` must be one that ClickHouse can use in `SELECT` queries and, if necessary, in `INSERTs`. For the full list of supported formats, see [Formats](#).

The `URL` must conform to the structure of a Uniform Resource Locator. The specified URL must point to a server that uses HTTP or HTTPS. This does not require any additional headers for getting a response from the server.

`INSERT` and `SELECT` queries are transformed to `POST` and `GET` requests, respectively. For processing `POST` requests, the remote server must support [Chunked transfer encoding](#).

Example:

1. Create a `url_engine_table` table on the server :

```
CREATE TABLE url_engine_table (word String, value UInt64)
ENGINE=URL('http://127.0.0.1:12345/', CSV)
```

2. Create a basic HTTP server using the standard Python 3 tools and start it:

```
from http.server import BaseHTTPRequestHandler, HTTPServer

class CSVHTTPServer(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/csv')
        self.end_headers()

        self.wfile.write(bytes('Hello,1\nWorld,2\n', "utf-8"))

if __name__ == "__main__":
    server_address = ('127.0.0.1', 12345)
    HTTPServer(server_address, CSVHTTPServer).serve_forever()
```

```
python3 server.py
```

3. Request data:

```
SELECT * FROM url_engine_table
```

word	value
Hello	1
World	2

Details of Implementation

- Reads and writes can be parallel
- Not supported:
 - ALTER and SELECT...SAMPLE operations.
 - Indexes.
 - Replication.

View

Used for implementing views (for more information, see the `CREATE VIEW query`). It does not store data, but only stores the specified `SELECT` query. When reading from a table, it runs this query (and deletes all unnecessary columns from the query).

MaterializedView

Used for implementing materialized views (for more information, see `CREATE TABLE`). For storing data, it uses a different engine that was specified when creating the view. When reading from a table, it just uses this engine.

Memory

The Memory engine stores data in RAM, in uncompressed form. Data is stored in exactly the same form as it is received when read. In other words, reading from this table is completely free.

Concurrent data access is synchronized. Locks are short: read and write operations don't block each other.

Indexes are not supported. Reading is parallelized.

Maximal productivity (over 10 GB/sec) is reached on simple queries, because there is no reading from the disk, decompressing, or deserializing data. (We should note that in many cases, the productivity of the MergeTree engine is almost as high.)

When restarting a server, data disappears from the table and the table becomes empty.

Normally, using this table engine is not justified. However, it can be used for tests, and for tasks where maximum speed is required on a relatively small number of rows (up to approximately 100,000,000).

The Memory engine is used by the system for temporary tables with external query data (see the section "External data for processing a query"), and for implementing GLOBAL IN (see the section "IN operators").

Buffer

Buffers the data to write in RAM, periodically flushing it to another table. During the read operation, data is read from the buffer and the other table simultaneously.

```
Buffer(database, table, num_layers, min_time, max_time, min_rows, max_rows, min_bytes, max_bytes)
```

Engine parameters: database, table – The table to flush data to. Instead of the database name, you can use a constant expression that returns a string. num_layers – Parallelism layer. Physically, the table will be represented as 'num_layers' of independent buffers. Recommended value: 16. min_time, max_time, min_rows, max_rows, min_bytes, and max_bytes are conditions for flushing data from the buffer.

Data is flushed from the buffer and written to the destination table if all the 'min' conditions or at least one 'max' condition are met. min_time, max_time – Condition for the time in seconds from the moment of the first write to the buffer. min_rows, max_rows – Condition for the number of rows in the buffer. min_bytes, max_bytes – Condition for the number of bytes in the buffer.

During the write operation, data is inserted to a 'num_layers' number of random buffers. Or, if the data part to insert is large enough (greater than 'max_rows' or 'max_bytes'), it is written directly to the destination table, omitting the buffer.

The conditions for flushing the data are calculated separately for each of the 'num_layers' buffers. For example, if num_layers = 16 and max_bytes = 100000000, the maximum RAM consumption is 1.6 GB.

Example:

```
CREATE TABLE merge.hits_buffer AS merge.hits ENGINE = Buffer(merge, hits, 16, 10, 100, 10000, 1000000, 100000000, 100000000)
```

Creating a 'merge.hits_buffer' table with the same structure as 'merge.hits' and using the Buffer engine. When writing to this table, data is buffered in RAM and later written to the 'merge.hits' table. 16 buffers are created. The data in each of them is flushed if either 100 seconds have passed, or one million rows have been written, or 100 MB of data have been written; or if simultaneously 10 seconds have passed and 10,000 rows and 10 MB of data have been written. For example, if just one row has been written, after 100 seconds it will be flushed, no matter what. But if many rows have been written, the data will be flushed sooner.

When the server is stopped, with DROP TABLE or DETACH TABLE, buffer data is also flushed to the destination table.

You can set empty strings in single quotation marks for the database and table name. This indicates the absence of a destination table. In this case, when the data flush conditions are reached, the buffer is simply cleared. This may be useful for keeping a window of data in memory.

When reading from a Buffer table, data is processed both from the buffer and from the destination table (if there is one).

Note that the Buffer tables does not support an index. In other words, data in the buffer is fully scanned, which might be slow for large buffers. (For data in a subordinate table, the index that it supports will be used.)

If the set of columns in the Buffer table doesn't match the set of columns in a subordinate table, a subset of columns that exist in both tables is inserted.

If the types don't match for one of the columns in the Buffer table and a subordinate table, an error message is entered in the server log and the buffer is cleared.

The same thing happens if the subordinate table doesn't exist when the buffer is flushed.

If you need to run ALTER for a subordinate table and the Buffer table, we recommend first deleting the Buffer table, running ALTER for the subordinate table, then creating the Buffer table again.

If the server is restarted abnormally, the data in the buffer is lost.

FINAL and SAMPLE do not work correctly for Buffer tables. These conditions are passed to the destination table, but are not used for processing data in the buffer. If these features are required we recommend only using the Buffer table for writing, while reading from the destination table.

When adding data to a Buffer, one of the buffers is locked. This causes delays if a read operation is simultaneously being performed from the table.

Data that is inserted to a Buffer table may end up in the subordinate table in a different order and in different blocks. Because of this, a Buffer table is difficult to use for writing to a CollapsingMergeTree correctly. To avoid problems, you can set 'num_layers' to 1.

If the destination table is replicated, some expected characteristics of replicated tables are lost when writing to a Buffer table. The random changes to the order of rows and sizes of data parts cause data deduplication to quit working, which means it is not possible to have a reliable 'exactly once' write to replicated tables.

Due to these disadvantages, we can only recommend using a Buffer table in rare cases.

A Buffer table is used when too many INSERTs are received from a large number of servers over a unit of time and data can't be buffered before insertion, which means the INSERTs can't run fast enough.

Note that it doesn't make sense to insert data one row at a time, even for Buffer tables. This will only produce a speed of a few thousand rows per second, while inserting larger blocks of data can produce over a million rows per second (see the section "Performance").

JDBC

Allows ClickHouse to connect to external databases via [JDBC](#).

To implement the JDBC connection, ClickHouse uses the separate program [clickhouse-jdbc-bridge](#) that should run as a daemon.

This engine supports the [Nullable](#) data type.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name
ENGINE = JDBC(dbms_uri, external_database, external_table)
```

Engine Parameters

- `dbms_uri` — URI of an external DBMS.

Format: `jdbc:<driver_name>://<host_name>:<port>/?user=<username>&password=<password>`.

Example for MySQL: `jdbc:mysql://localhost:3306/?user=root&password=root`.

- `external_database` — Database in an external DBMS.
- `external_table` — Name of the table in `external_database`.

Usage Example

Usage Example

Creating a table in MySQL server by connecting directly with it's console client:

```
mysql> CREATE TABLE `test`.`test` (  
-> `int_id` INT NOT NULL AUTO_INCREMENT,  
-> `int_nullable` INT NULL DEFAULT NULL,  
-> `float` FLOAT NOT NULL,  
-> `float_nullable` FLOAT NULL DEFAULT NULL,  
-> PRIMARY KEY (`int_id`));  
Query OK, 0 rows affected (0,09 sec)  
  
mysql> insert into test (`int_id`, `float`) VALUES (1,2);  
Query OK, 1 row affected (0,00 sec)  
  
mysql> select * from test;  
+-----+-----+-----+-----+  
| int_id | int_nullable | float | float_nullable |  
+-----+-----+-----+-----+  
| 1 | NULL | 2 | NULL |  
+-----+-----+-----+-----+  
1 row in set (0,00 sec)
```

Creating a table in ClickHouse server and selecting data from it:

```
CREATE TABLE jdbc_table ENGINE JDBC('jdbc:mysql://localhost:3306/?user=root&password=root', 'test', 'test')  
  
Ok.  
  
DESCRIBE TABLE jdbc_table  
  
+-----+-----+-----+-----+  
| name | type | default_type | default_expression |  
+-----+-----+-----+-----+  
| int_id | Int32 | | |  
| int_nullable | Nullable(Int32) | | |  
| float | Float32 | | |  
| float_nullable | Nullable(Float32) | | |  
+-----+-----+-----+-----+  
  
10 rows in set. Elapsed: 0.031 sec.  
  
SELECT *  
FROM jdbc_table  
  
+-----+-----+-----+-----+  
| int_id | int_nullable | float | float_nullable |  
+-----+-----+-----+-----+  
| 1 | NULL | 2 | NULL |  
+-----+-----+-----+-----+  
  
1 rows in set. Elapsed: 0.055 sec.
```

See Also

- [JDBC table function](#).

ODBC

Allows ClickHouse to connect to external databases via [ODBC](#).

To implement ODBC connection safely, ClickHouse uses the separate program `clickhouse-odbc-bridge`. If the ODBC driver is loaded directly from the `clickhouse-server` program, the problems in the driver can crash the ClickHouse server. ClickHouse starts the `clickhouse-odbc-bridge` program automatically when it is required. The ODBC bridge program is installed by the same package as the `clickhouse-server`.

This article documents the [MySQL](#) data source.

This engine supports the **Nullable** data type.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1],
    name2 [type2],
    ...
)
ENGINE = ODBC(connection_settings, external_database, external_table)
```

See the detailed description of the **CREATE TABLE** query.

The table structure can be not the same as the source table structure:

- Names of columns should be the same as in the source table, but you can use just some of these columns in any order.
- Types of columns may differ from the types in the source table. ClickHouse tries to **cast** values into the ClickHouse data types.

Engine Parameters

- `connection_settings` — Name of the section with connection settings in the `odbc.ini` file.
- `external_database` — Name of a database in an external DBMS.
- `external_table` — Name of a table in the `external_database`.

Usage Example

Getting data from the local MySQL installation via ODBC

This example is for linux Ubuntu 18.04 and MySQL server 5.7.

Ensure that there are unixODBC and MySQL Connector are installed.

By default (if installed from packages) ClickHouse starts on behalf of the user `clickhouse`. Thus, you need to create and configure this user at MySQL server.

```
sudo mysql
mysql> CREATE USER 'clickhouse'@'localhost' IDENTIFIED BY 'clickhouse';
mysql> GRANT ALL PRIVILEGES ON *.* TO 'clickhouse'@'clickhouse' WITH GRANT OPTION;
```

Then configure the connection in `/etc/odbc.ini`.

```
$ cat /etc/odbc.ini
[mysqlconn]
DRIVER = /usr/local/lib/libmyodbc5w.so
SERVER = 127.0.0.1
PORT = 3306
DATABASE = test
USERNAME = clickhouse
PASSWORD = clickhouse
```

You can check the connection by the `isql` utility from the unixODBC installation.

```
isql -v mysqlconn
+-----+
| Connected! |
|           |
|           |
```


Table in MySQL:

```
mysql> CREATE TABLE `test`.`test` (  
-> `int_id` INT NOT NULL AUTO_INCREMENT,  
-> `int_nullable` INT NULL DEFAULT NULL,  
-> `float` FLOAT NOT NULL,  
-> `float_nullable` FLOAT NULL DEFAULT NULL,  
-> PRIMARY KEY (`int_id`));  
Query OK, 0 rows affected (0,09 sec)  
  
mysql> insert into test (`int_id`, `float`) VALUES (1,2);  
Query OK, 1 row affected (0,00 sec)  
  
mysql> select * from test;  
+-----+-----+-----+-----+  
| int_id | int_nullable | float | float_nullable |  
+-----+-----+-----+-----+  
| 1 | NULL | 2 | NULL |  
+-----+-----+-----+-----+  
1 row in set (0,00 sec)
```

Table in ClickHouse, getting data from the MySQL table:

```
CREATE TABLE odbc_t  
(  
  `int_id` Int32,  
  `float_nullable` Nullable(Float32)  
)  
ENGINE = ODBC('DSN=mysqlconn', 'test', 'test')
```

```
SELECT * FROM odbc_t
```

```
┌ int_id ─┴─ float_nullable ─┐  
| 1 | NULL |  
└──────────┴──────────┘
```

See Also

- [ODBC external dictionaries](#)
- [ODBC table function](#).

SQL Reference

- [SELECT](#)
- [INSERT INTO](#)
- [CREATE](#)
- [ALTER](#)
- [Other types of queries](#)

SELECT Queries Syntax

`SELECT` performs data retrieval.

```
SELECT [DISTINCT] expr_list  
[FROM [db.]table | (subquery) | table_function] [FINAL]  
[SAMPLE sample_coeff]
```

```
[ARRAY JOIN ...]  
[GLOBAL] [ANY|ALL] [INNER|LEFT|RIGHT|FULL|CROSS] [OUTER] JOIN (subquery)|table USING columns_list  
[PREWHERE expr]  
[WHERE expr]  
[GROUP BY expr_list] [WITH TOTALS]  
[HAVING expr]  
[ORDER BY expr_list]  
[LIMIT [n, ]m]  
[UNION ALL ...]  
[INTO OUTFILE filename]  
[FORMAT format]  
[LIMIT [offset_value, ]n BY columns]
```

All the clauses are optional, except for the required list of expressions immediately after SELECT. The clauses below are described in almost the same order as in the query execution conveyor.

If the query omits the `DISTINCT`, `GROUP BY` and `ORDER BY` clauses and the `IN` and `JOIN` subqueries, the query will be completely stream processed, using $O(1)$ amount of RAM.

Otherwise, the query might consume a lot of RAM if the appropriate restrictions are not specified:

`max_memory_usage`, `max_rows_to_group_by`, `max_rows_to_sort`, `max_rows_in_distinct`, `max_bytes_in_distinct`, `max_rows_in_set`, `max_bytes_in_set`, `max_rows_in_join`, `max_bytes_in_join`, `max_bytes_before_external_sort`, `max_bytes_before_external_group_by`. For more information, see the section "Settings". It is possible to use external sorting (saving temporary tables to a disk) and external aggregation. The system does not have "merge join".

FROM Clause

If the FROM clause is omitted, data will be read from the `system.one` table.

The 'system.one' table contains exactly one row (this table fulfills the same purpose as the DUAL table found in other DBMSs).

The FROM clause specifies the table to read data from, or a subquery, or a table function; ARRAY JOIN and the regular JOIN may also be included (see below).

Instead of a table, the SELECT subquery may be specified in brackets.

In this case, the subquery processing pipeline will be built into the processing pipeline of an external query.

In contrast to standard SQL, a synonym does not need to be specified after a subquery. For compatibility, it is possible to write 'AS name' after a subquery, but the specified name isn't used anywhere.

A table function may be specified instead of a table. For more information, see the section "Table functions".

To execute a query, all the columns listed in the query are extracted from the appropriate table. Any columns not needed for the external query are thrown out of the subqueries.

If a query does not list any columns (for example, `SELECT count() FROM t`), some column is extracted from the table anyway (the smallest one is preferred), in order to calculate the number of rows.

The FINAL modifier can be used only for a SELECT from a CollapsingMergeTree table. When you specify FINAL, data is selected fully "collapsed". Keep in mind that using FINAL leads to a selection that includes columns related to the primary key, in addition to the columns specified in the SELECT. Additionally, the query will be executed in a single stream, and data will be merged during query execution. This means that when using FINAL, the query is processed more slowly. In most cases, you should avoid using FINAL. For more information, see the section "CollapsingMergeTree engine".

SAMPLE Clause

The `SAMPLE` clause allows for approximated query processing.

When data sampling is enabled, the query is not performed on all the data, but only on a certain fraction of data (sample). For example, if you need to calculate statistics for all the visits, it is enough to execute the query on the 1/10 fraction of all the visits and then multiply the result by 10.

Approximated query processing can be useful in the following cases:

- When you have strict timing requirements (like <100ms) but you can't justify the cost of additional hardware resources to meet them.
- When your raw data is not accurate, so approximation doesn't noticeably degrade the quality.
- Business requirements target approximate results (for cost-effectiveness, or in order to market exact results to premium users).

Note

You can only use sampling with the tables in the [MergeTree](#) family, and only if the sampling expression was specified during table creation (see [MergeTree engine](#)).

The features of data sampling are listed below:

- Data sampling is a deterministic mechanism. The result of the same `SELECT .. SAMPLE` query is always the same.
- Sampling works consistently for different tables. For tables with a single sampling key, a sample with the same coefficient always selects the same subset of possible data. For example, a sample of user IDs takes rows with the same subset of all the possible user IDs from different tables. This means that you can use the sample in subqueries in the `IN` clause. Also, you can join samples using the `JOIN` clause.
- Sampling allows reading less data from a disk. Note that you must specify the sampling key correctly. For more information, see [Creating a MergeTree Table](#).

For the `SAMPLE` clause the following syntax is supported:

SAMPLE Clause Syntax	Description
<code>SAMPLE k</code>	Here <code>k</code> is the number from 0 to 1.

The query is executed on `k` fraction of data. For example, `SAMPLE 0.1` runs the query on 10% of data. [Read more](#)

`SAMPLE n` Here `n` is a sufficiently large integer. The query is executed on a sample of at least `n` rows (but not significantly more than this). For example, `SAMPLE 10000000` runs the query on a minimum of 10,000,000 rows. [Read more](#)

`SAMPLE k OFFSET m` Here `k` and `m` are the numbers from 0 to 1. The query is executed on a sample of `k` fraction of the data. The data used for the sample is offset by `m` fraction. [Read more](#)

SAMPLE k

Here `k` is the number from 0 to 1 (both fractional and decimal notations are supported). For example, `SAMPLE 1/2` or `SAMPLE 0.5`.

In a `SAMPLE k` clause, the sample is taken from the `k` fraction of data. The example is shown below:

```
SELECT
  Title,
  count() * 10 AS PageViews
FROM hits_distributed
SAMPLE 0.1
WHERE
  CounterID = 34
GROUP BY Title
ORDER BY PageViews DESC LIMIT 1000
```

In this example, the query is executed on a sample from 0.1 (10%) of data. Values of aggregate functions are not corrected automatically, so to get an approximate result, the value `count()` is manually multiplied by 10.

SAMPLE n

Here `n` is a sufficiently large integer. For example, `SAMPLE 10000000`.

In this case, the query is executed on a sample of at least `n` rows (but not significantly more than this). For example, `SAMPLE 10000000` runs the query on a minimum of 10,000,000 rows.

Since the minimum unit for data reading is one granule (its size is set by the `index_granularity` setting), it makes sense to set a sample that is much larger than the size of the granule.

When using the `SAMPLE n` clause, you don't know which relative percent of data was processed. So you don't know the coefficient the aggregate functions should be multiplied by. Use the `_sample_factor` virtual column to get the approximate result.

The `_sample_factor` column contains relative coefficients that are calculated dynamically. This column is created automatically when you `create` a table with the specified sampling key. The usage examples of the `_sample_factor` column are shown below.

Let's consider the table `visits`, which contains the statistics about site visits. The first example shows how to calculate the number of page views:

```
SELECT sum(PageViews * _sample_factor)
FROM visits
SAMPLE 10000000
```

The next example shows how to calculate the total number of visits:

```
SELECT sum(_sample_factor)
FROM visits
SAMPLE 10000000
```

The example below shows how to calculate the average session duration. Note that you don't need to use the relative coefficient to calculate the average values.

```
SELECT avg(Duration)
FROM visits
SAMPLE 10000000
```

SAMPLE k OFFSET m

Here `k` and `m` are numbers from 0 to 1. Examples are shown below.

Example 1

```
SAMPLE 1/10
```

In this example, the sample is 1/10th of all data:

[++-----]

Example 2

```
SAMPLE 1/10 OFFSET 1/2
```

Here, a sample of 10% is taken from the second half of the data.

[------++-----]

ARRAY JOIN Clause

Allows executing `JOIN` with an array or nested data structure. The intent is similar to the `arrayJoin` function, but its functionality is broader.

```
SELECT <expr_list>
FROM <left_subquery>
[LEFT] ARRAY JOIN <array>
[WHERE|PREWHERE <expr>]
...
```

You can specify only a single `ARRAY JOIN` clause in a query.

The query execution order is optimized when running **ARRAY JOIN**. Although **ARRAY JOIN** must always be specified before the **WHERE/PREWHERE** clause, it can be performed either before **WHERE/PREWHERE** (if the result is needed in this clause), or after completing it (to reduce the volume of calculations). The processing order is controlled by the query optimizer.

Supported types of **ARRAY JOIN** are listed below:

- **ARRAY JOIN** - In this case, empty arrays are not included in the result of **JOIN**.
- **LEFT ARRAY JOIN** - The result of **JOIN** contains rows with empty arrays. The value for an empty array is set to the default value for the array element type (usually 0, empty string or NULL).

The examples below demonstrate the usage of the **ARRAY JOIN** and **LEFT ARRAY JOIN** clauses. Let's create a table with an **Array** type column and insert values into it:

```
CREATE TABLE arrays_test
(
  s String,
  arr Array(UInt8)
) ENGINE = Memory;

INSERT INTO arrays_test
VALUES ('Hello', [1,2]), ('World', [3,4,5]), ('Goodbye', []);
```

s	arr
Hello	[1,2]
World	[3,4,5]
Goodbye	[]

The example below uses the **ARRAY JOIN** clause:

```
SELECT s, arr
FROM arrays_test
ARRAY JOIN arr;
```

s	arr
Hello	1
Hello	2
World	3
World	4
World	5

The next example uses the **LEFT ARRAY JOIN** clause:

```
SELECT s, arr
FROM arrays_test
LEFT ARRAY JOIN arr;
```

s	arr
Hello	1
Hello	2
World	3
World	4
World	5
Goodbye	0

Using Aliases

An alias can be specified for an array in the **ARRAY JOIN** clause. In this case, an array item can be accessed by this alias, but the array itself is accessed by the original name. Example:

```
SELECT s, arr, a
FROM arrays_test
ARRAY JOIN arr AS a;
```

s	arr	a
Hello	[1,2]	1
Hello	[1,2]	2
World	[3,4,5]	3
World	[3,4,5]	4
World	[3,4,5]	5

Using aliases, you can perform **ARRAY JOIN** with an external array. For example:

```
SELECT s, arr_external
FROM arrays_test
ARRAY JOIN [1, 2, 3] AS arr_external;
```

s	arr_external
Hello	1
Hello	2
Hello	3
World	1
World	2
World	3
Goodbye	1
Goodbye	2
Goodbye	3

Multiple arrays can be comma-separated in the **ARRAY JOIN** clause. In this case, **JOIN** is performed with them simultaneously (the direct sum, not the cartesian product). Note that all the arrays must have the same size. Example:

```
SELECT s, arr, a, num, mapped
FROM arrays_test
ARRAY JOIN arr AS a, arrayEnumerate(arr) AS num, arrayMap(x -> x + 1, arr) AS mapped;
```

s	arr	a	num	mapped
Hello	[1,2]	1	1	2
Hello	[1,2]	2	2	3
World	[3,4,5]	3	1	4
World	[3,4,5]	4	2	5
World	[3,4,5]	5	3	6

The example below uses the **arrayEnumerate** function:

```
SELECT s, arr, a, num, arrayEnumerate(arr)
FROM arrays_test
ARRAY JOIN arr AS a, arrayEnumerate(arr) AS num;
```

s	arr	a	num	arrayEnumerate(arr)
Hello	[1,2]	1	1	[1,2]
Hello	[1,2]	2	2	[1,2]
World	[3,4,5]	3	1	[1,2,3]
World	[3,4,5]	4	2	[1,2,3]
World	[3,4,5]	5	3	[1,2,3]

ARRAY JOIN With Nested Data Structure

ARRAY JOIN also works with **nested data structures**. Example:

```
CREATE TABLE nested_test
(
  s String,
  nest Nested(
    x UInt8,
    y UInt32)
) ENGINE = Memory;

INSERT INTO nested_test
VALUES ('Hello', [1,2], [10,20]), ('World', [3,4,5], [30,40,50]), ('Goodbye', [], []);
```

s	nest.x	nest.y
Hello	[1,2]	[10,20]
World	[3,4,5]	[30,40,50]
Goodbye	[]	[]

```
SELECT s, `nest.x`, `nest.y`
FROM nested_test
ARRAY JOIN nest;
```

s	nest.x	nest.y
Hello	1	10
Hello	2	20
World	3	30
World	4	40
World	5	50

When specifying names of nested data structures in **ARRAY JOIN**, the meaning is the same as **ARRAY JOIN** with all the array elements that it consists of. Examples are listed below:

```
SELECT s, `nest.x`, `nest.y`
FROM nested_test
ARRAY JOIN `nest.x`, `nest.y`;
```

s	nest.x	nest.y
Hello	1	10
Hello	2	20
World	3	30
World	4	40
World	5	50

This variation also makes sense:

```
SELECT s, `nest.x`, `nest.y`
FROM nested_test
ARRAY JOIN `nest.x`;
```

s	nest.x	nest.y
Hello	1	[10,20]
Hello	2	[10,20]
World	3	[30,40,50]
World	4	[30,40,50]
World	5	[30,40,50]

An alias may be used for a nested data structure, in order to select either the **JOIN** result or the source array. Example:

```
SELECT s, `n.x`, `n.y`, `nest.x`, `nest.y`
FROM nested_test
ARRAY JOIN nest AS n;
```

s	n.x	n.y	nest.x	nest.y
Hello	1	10	[1,2]	[10,20]
Hello	2	20	[1,2]	[10,20]
World	3	30	[3,4,5]	[30,40,50]
World	4	40	[3,4,5]	[30,40,50]
World	5	50	[3,4,5]	[30,40,50]

Example of using the `arrayEnumerate` function:

```
SELECT s, `n.x`, `n.y`, `nest.x`, `nest.y`, num
FROM nested_test
ARRAY JOIN nest AS n, arrayEnumerate(`nest.x`) AS num;
```

s	n.x	n.y	nest.x	nest.y	num
Hello	1	10	[1,2]	[10,20]	1
Hello	2	20	[1,2]	[10,20]	2
World	3	30	[3,4,5]	[30,40,50]	1
World	4	40	[3,4,5]	[30,40,50]	2
World	5	50	[3,4,5]	[30,40,50]	3

JOIN Clause

Joins the data in the normal `SQL JOIN` sense.

Note

Not related to `ARRAY JOIN`.

```
SELECT <expr_list>
FROM <left_subquery>
[GLOBAL] [ANY|ALL] [INNER|LEFT|RIGHT|FULL|CROSS] [OUTER] JOIN <right_subquery>
(ON <expr_list>)|(USING <column_list>) ...
```

The table names can be specified instead of `<left_subquery>` and `<right_subquery>`. This is equivalent to the `SELECT * FROM table` subquery, except in a special case when the table has the `Join` engine – an array prepared for joining.

Supported Types of JOIN

- `INNER JOIN` (or `JOIN`)
- `LEFT JOIN` (or `LEFT OUTER JOIN`)
- `RIGHT JOIN` (or `RIGHT OUTER JOIN`)
- `FULL JOIN` (or `FULL OUTER JOIN`)
- `CROSS JOIN` (or `,`)

See the standard `SQL JOIN` description.

Multiple JOIN

Performing queries, ClickHouse rewrites multiple joins into the sequence of two-table joins. If there are four tables for join ClickHouse joins the first and the second, then joins the result with the third table, and at the last step, it joins the fourth one.

If a query contains `WHERE` clause, ClickHouse tries to push down filters from this clause into the intermediate join. If it cannot apply the filter to each intermediate join, ClickHouse applies the filters after all joins are completed.

We recommend the `JOIN ON` or `JOIN USING` syntax for creating a query. For example:

```
SELECT * FROM t1 JOIN t2 ON t1.a = t2.a JOIN t3 ON t1.a = t3.a
```

Also, you can use comma separated list of tables for join. Works only with the `allow_experimental_cross_to_join_conversion = 1` setting.

```
For example, `SELECT * FROM t1, t2, t3 WHERE t1.a = t2.a AND t1.a = t3.a`
```


Don't mix these syntaxes.

ClickHouse doesn't support the syntax with commas directly, so we don't recommend to use it. The algorithm tries to rewrite the query in terms of **CROSS** and **INNER JOIN** clauses and then proceeds the query processing. When rewriting the query, ClickHouse tries to optimize performance and memory consumption. By default, ClickHouse treats comma as an **INNER JOIN** clause and converts it to **CROSS JOIN** when the algorithm cannot guaranty that **INNER JOIN** returns required data.

ANY or ALL Strictness

If **ALL** is specified and the right table has several matching rows, the data will be multiplied by the number of these rows. This is the normal **JOIN** behavior for standard SQL.

If **ANY** is specified and the right table has several matching rows, only the first one found is joined. If the right table has only one matching row, the results of **ANY** and **ALL** are the same.

To set the default strictness value, use the session configuration parameter **join_default_strictness**.

GLOBAL JOIN

When using a normal **JOIN**, the query is sent to remote servers. Subqueries are run on each of them in order to make the right table, and the join is performed with this table. In other words, the right table is formed on each server separately.

When using **GLOBAL ... JOIN**, first the requestor server runs a subquery to calculate the right table. This temporary table is passed to each remote server, and queries are run on them using the temporary data that was transmitted.

Be careful when using **GLOBAL**. For more information, see the section **Distributed subqueries**.

Usage Recommendations

All columns that are not needed for the **JOIN** are deleted from the subquery.

When running a **JOIN**, there is no optimization of the order of execution in relation to other stages of the query. The join (a search in the right table) is run before filtering in **WHERE** and before aggregation. In order to explicitly set the processing order, we recommend running a **JOIN** subquery with a subquery.

Example:

```
SELECT
  CounterID,
  hits,
  visits
FROM
(
  SELECT
    CounterID,
    count() AS hits
  FROM test.hits
  GROUP BY CounterID
) ANY LEFT JOIN
(
  SELECT
    CounterID,
    sum(Sign) AS visits
  FROM test.visits
  GROUP BY CounterID
) USING CounterID
ORDER BY hits DESC
LIMIT 10
```

CounterID	hits	visits
1143050	523264	13665
731962	475698	102716
722545	337212	108187
722889	252197	10547
2237260	196036	9522
23057320	147211	7689
722818	90109	17847
48221	85379	4652
19762435	77807	7026
722884	77492	11056

Subqueries don't allow you to set names or use them for referencing a column from a specific subquery. The columns specified in **USING** must have the same names in both subqueries, and the other columns must be named differently. You can use aliases to change the names of columns in subqueries (the example uses the aliases **hits** and **visits**).

The **USING** clause specifies one or more columns to join, which establishes the equality of these columns. The list of columns is set without brackets. More complex join conditions are not supported.

The right table (the subquery result) resides in RAM. If there isn't enough memory, you can't run a **JOIN**.

Each time a query is run with the same **JOIN**, the subquery is run again because the result is not cached. To avoid this, use the special **Join** table engine, which is a prepared array for joining that is always in RAM.

In some cases, it is more efficient to use **IN** instead of **JOIN**.

Among the various types of **JOIN**, the most efficient is **ANY LEFT JOIN**, then **ANY INNER JOIN**. The least efficient are **ALL LEFT JOIN** and **ALL INNER JOIN**.

If you need a **JOIN** for joining with dimension tables (these are relatively small tables that contain dimension properties, such as names for advertising campaigns), a **JOIN** might not be very convenient due to the bulky syntax and the fact that the right table is re-accessed for every query. For such cases, there is an "external dictionaries" feature that you should use instead of **JOIN**. For more information, see the section **External dictionaries**.

Processing of Empty or NULL Cells

While joining tables, the empty cells may appear. The setting **join_use_nulls** define how ClickHouse fills these cells.

If the **JOIN** keys are **Nullable** fields, the rows where at least one of the keys has the value **NULL** are not joined.

Syntax Limitations

For multiple **JOIN** clauses in the single **SELECT** query:

- Taking all the columns via ***** is available only if tables are joined, not subqueries.
- The **PREWHERE** clause is not available.

For **ON**, **WHERE** and **GROUP BY** clauses:

- Arbitrary expressions cannot be used in **ON**, **WHERE** and **GROUP BY** clauses, but you can define an expression in **SELECT** clause and then use it via alias in these clauses.

WHERE Clause

If there is a **WHERE** clause, it must contain an expression with the **UInt8** type. This is usually an expression with comparison and logical operators.

This expression will be used for filtering data before all other transformations.

If indexes are supported by the database table engine, the expression is evaluated on the ability to use indexes.

PREWHERE Clause

This clause has the same meaning as the WHERE clause. The difference is in which data is read from the table. When using PREWHERE, first only the columns necessary for executing PREWHERE are read. Then the other columns are read that are needed for running the query, but only those blocks where the PREWHERE expression is true.

It makes sense to use PREWHERE if there are filtration conditions that are used by a minority of the columns in the query, but that provide strong data filtration. This reduces the volume of data to read.

For example, it is useful to write PREWHERE for queries that extract a large number of columns, but that only have filtration for a few columns.

PREWHERE is only supported by tables from the **MergeTree* family.

A query may simultaneously specify PREWHERE and WHERE. In this case, PREWHERE precedes WHERE.

If the 'optimize_move_to_prewrite' setting is set to 1 and PREWHERE is omitted, the system uses heuristics to automatically move parts of expressions from WHERE to PREWHERE.

GROUP BY Clause

This is one of the most important parts of a column-oriented DBMS.

If there is a GROUP BY clause, it must contain a list of expressions. Each expression will be referred to here as a "key".

All the expressions in the SELECT, HAVING, and ORDER BY clauses must be calculated from keys or from aggregate functions. In other words, each column selected from the table must be used either in keys or inside aggregate functions.

If a query contains only table columns inside aggregate functions, the GROUP BY clause can be omitted, and aggregation by an empty set of keys is assumed.

Example:

```
SELECT
    count(),
    median(FetchTiming > 60 ? 60 : FetchTiming),
    count() - sum(Refresh)
FROM hits
```

However, in contrast to standard SQL, if the table doesn't have any rows (either there aren't any at all, or there aren't any after using WHERE to filter), an empty result is returned, and not the result from one of the rows containing the initial values of aggregate functions.

As opposed to MySQL (and conforming to standard SQL), you can't get some value of some column that is not in a key or aggregate function (except constant expressions). To work around this, you can use the 'any' aggregate function (get the first encountered value) or 'min/max'.

Example:

```
SELECT
    domainWithoutWWW(URL) AS domain,
    count(),
    any(Title) AS title -- getting the first occurred page header for each domain.
FROM hits
GROUP BY domain
```

For every different key value encountered, GROUP BY calculates a set of aggregate function values.

GROUP BY is not supported for array columns.

A constant can't be specified as arguments for aggregate functions. Example: sum(1). Instead of this, you can get rid of the constant. Example: *count()*.

NULL processing

For grouping, ClickHouse interprets **NULL** as a value, and **NULL=NULL**.

Here's an example to show what this means.

Assume you have this table:

x	y
1	2
2	NULL
3	2
3	3
3	NULL

The query **SELECT sum(x), y FROM t_null_big GROUP BY y** results in:

sum(x)	y
4	2
3	3
5	NULL

You can see that **GROUP BY** for **y = NULL** summed up **x**, as if **NULL** is this value.

If you pass several keys to **GROUP BY**, the result will give you all the combinations of the selection, as if **NULL** were a specific value.

WITH TOTALS Modifier

If the **WITH TOTALS** modifier is specified, another row will be calculated. This row will have key columns containing default values (zeros or empty lines), and columns of aggregate functions with the values calculated across all the rows (the "total" values).

This extra row is output in **JSON***, **TabSeparated***, and **Pretty*** formats, separately from the other rows. In the other formats, this row is not output.

In **JSON*** formats, this row is output as a separate 'totals' field. In **TabSeparated*** formats, the row comes after the main result, preceded by an empty row (after the other data). In **Pretty*** formats, the row is output as a separate table after the main result.

WITH TOTALS can be run in different ways when **HAVING** is present. The behavior depends on the 'totals_mode' setting.

By default, **totals_mode = 'before_having'**. In this case, 'totals' is calculated across all rows, including the ones that don't pass through **HAVING** and 'max_rows_to_group_by'.

The other alternatives include only the rows that pass through **HAVING** in 'totals', and behave differently with the setting **max_rows_to_group_by** and **group_by_overflow_mode = 'any'**.

after_having_exclusive – Don't include rows that didn't pass through **max_rows_to_group_by**. In other words, 'totals' will have less than or the same number of rows as it would if **max_rows_to_group_by** were omitted.

after_having_inclusive – Include all the rows that didn't pass through 'max_rows_to_group_by' in 'totals'. In other words, 'totals' will have more than or the same number of rows as it would if **max_rows_to_group_by** were omitted.

after_having_auto – Count the number of rows that passed through **HAVING**. If it is more than a certain amount (by default, 50%), include all the rows that didn't pass through 'max_rows_to_group_by' in 'totals'. Otherwise, do not include them.

totals_auto_threshold – By default, 0.5. The coefficient for **after_having_auto**.

If **max_rows_to_group_by** and **group_by_overflow_mode = 'any'** are not used, all variations of **after_having** are the same, and you can use any of them (for example, **after_having_auto**).

You can use WITH TOTALS in subqueries, including subqueries in the JOIN clause (in this case, the respective total values are combined).

GROUP BY in External Memory

You can enable dumping temporary data to the disk to restrict memory usage during GROUP BY.

The `max_bytes_before_external_group_by` setting determines the threshold RAM consumption for dumping GROUP BY temporary data to the file system. If set to 0 (the default), it is disabled.

When using `max_bytes_before_external_group_by`, we recommend that you set `max_memory_usage` about twice as high. This is necessary because there are two stages to aggregation: reading the data and forming intermediate data (1) and merging the intermediate data (2). Dumping data to the file system can only occur during stage 1. If the temporary data wasn't dumped, then stage 2 might require up to the same amount of memory as in stage 1.

For example, if `max_memory_usage` was set to 10000000000 and you want to use external aggregation, it makes sense to set `max_bytes_before_external_group_by` to 10000000000, and `max_memory_usage` to 20000000000. When external aggregation is triggered (if there was at least one dump of temporary data), maximum consumption of RAM is only slightly more than `max_bytes_before_external_group_by`.

With distributed query processing, external aggregation is performed on remote servers. In order for the requestor server to use only a small amount of RAM, set `distributed_aggregation_memory_efficient` to 1.

When merging data flushed to the disk, as well as when merging results from remote servers when the `distributed_aggregation_memory_efficient` setting is enabled, consumes up to $1/256 * \text{the number of threads}$ from the total amount of RAM.

When external aggregation is enabled, if there was less than `max_bytes_before_external_group_by` of data (i.e. data was not flushed), the query runs just as fast as without external aggregation. If any temporary data was flushed, the run time will be several times longer (approximately three times).

If you have an ORDER BY with a small LIMIT after GROUP BY, then the ORDER BY CLAUSE will not use significant amounts of RAM.

But if the ORDER BY doesn't have LIMIT, don't forget to enable external sorting (`max_bytes_before_external_sort`).

LIMIT BY Clause

The query with the `LIMIT n BY expressions` clause selects the first `n` rows for each distinct value of `expressions`. The key for `LIMIT BY` can contain any number of `expressions`.

ClickHouse supports the following syntax:

- `LIMIT [offset_value,]n BY expressions`
- `LIMIT n OFFSET offset_value BY expressions`

During the query processing, ClickHouse selects data ordered by sorting key. Sorting key is set explicitly by `ORDER BY` clause or implicitly as a property of table engine. Then ClickHouse applies `LIMIT n BY expressions` and returns the first `n` rows for each distinct combination of `expressions`. If `OFFSET` is specified, then for each data block, belonging to a distinct combination of `expressions`, ClickHouse skips `offset_value` rows from the beginning of the block, and returns not more than `n` rows as a result. If `offset_value` is bigger than the number of rows in the data block, then ClickHouse returns no rows from the block.

`LIMIT BY` is not related to `LIMIT`, they can both be used in the same query.

Examples

Sample table:

```
CREATE TABLE limit_by(id Int, val Int) ENGINE = Memory;  
INSERT INTO limit_by values(1, 10), (1, 11), (1, 12), (2, 20), (2, 21);
```

Queries:

```
SELECT * FROM limit_by ORDER BY id, val LIMIT 2 BY id
```

id	val
1	10
1	11
2	20
2	21

```
SELECT * FROM limit_by ORDER BY id, val LIMIT 1, 2 BY id
```

id	val
1	11
1	12
2	21

The `SELECT * FROM limit_by ORDER BY id, val LIMIT 2 OFFSET 1 BY id` query returns the same result.

The following query returns the top 5 referrers for each `domain`, `device_type` pair, but not more than 100 rows (`LIMIT n BY + LIMIT`).

```
SELECT
  domainWithoutWWW(URL) AS domain,
  domainWithoutWWW(REFERRER_URL) AS referrer,
  device_type,
  count() cnt
FROM hits
GROUP BY domain, referrer, device_type
ORDER BY cnt DESC
LIMIT 5 BY domain, device_type
LIMIT 100
```

HAVING Clause

Allows filtering the result received after GROUP BY, similar to the WHERE clause.

WHERE and HAVING differ in that WHERE is performed before aggregation (GROUP BY), while HAVING is performed after it.

If aggregation is not performed, HAVING can't be used.

ORDER BY Clause

The ORDER BY clause contains a list of expressions, which can each be assigned DESC or ASC (the sorting direction). If the direction is not specified, ASC is assumed. ASC is sorted in ascending order, and DESC in descending order. The sorting direction applies to a single expression, not to the entire list. Example: `ORDER BY Visits DESC, SearchPhrase`

For sorting by String values, you can specify collation (comparison). Example: `ORDER BY SearchPhrase COLLATE 'tr'` - for sorting by keyword in ascending order, using the Turkish alphabet, case insensitive, assuming that strings are UTF-8 encoded. COLLATE can be specified or not for each expression in ORDER BY independently. If ASC or DESC is specified, COLLATE is specified after it. When using COLLATE, sorting is always case-insensitive.

We only recommend using COLLATE for final sorting of a small number of rows, since sorting with COLLATE is less efficient than normal sorting by bytes.

Rows that have identical values for the list of sorting expressions are output in an arbitrary order, which can also be nondeterministic (different each time).

If the ORDER BY clause is omitted, the order of the rows is also undefined, and may be nondeterministic as well.

NaN and **NULL** sorting order:

- With the modifier **NULLS FIRST** — First **NULL**, then **NaN**, then other values.
- With the modifier **NULLS LAST** — First the values, then **NaN**, then **NULL**.
- Default — The same as with the **NULLS LAST** modifier.

Example:

For the table

x	y
1	NULL
2	2
1	nan
2	2
3	4
5	6
6	nan
7	NULL
6	7
8	9

Run the query `SELECT * FROM t_null_nan ORDER BY y NULLS FIRST` to get:

x	y
1	NULL
7	NULL
1	nan
6	nan
2	2
2	2
3	4
5	6
6	7
8	9

When floating point numbers are sorted, NaNs are separate from the other values. Regardless of the sorting order, NaNs come at the end. In other words, for ascending sorting they are placed as if they are larger than all the other numbers, while for descending sorting they are placed as if they are smaller than the rest.

Less RAM is used if a small enough LIMIT is specified in addition to ORDER BY. Otherwise, the amount of memory spent is proportional to the volume of data for sorting. For distributed query processing, if GROUP BY is omitted, sorting is partially done on remote servers, and the results are merged on the requestor server. This means that for distributed sorting, the volume of data to sort can be greater than the amount of memory on a single server.

If there is not enough RAM, it is possible to perform sorting in external memory (creating temporary files on a disk). Use the setting `max_bytes_before_external_sort` for this purpose. If it is set to 0 (the default), external sorting is disabled. If it is enabled, when the volume of data to sort reaches the specified number of bytes, the collected data is sorted and dumped into a temporary file. After all data is read, all the sorted files are merged and the results are output. Files are written to the `/var/lib/clickhouse/tmp/` directory in the config (by default, but you can use the `'tmp_path'` parameter to change this setting).

Running a query may use more memory than `'max_bytes_before_external_sort'`. For this reason, this setting must have a value significantly smaller than `'max_memory_usage'`. As an example, if your server has 128 GB of RAM and you need to run a single query, set `'max_memory_usage'` to 100 GB, and `'max_bytes_before_external_sort'` to 80 GB.

External sorting works much less effectively than sorting in RAM.

SELECT Clause

The expressions specified in the `SELECT` clause are analyzed after the calculations for all the clauses listed above are completed.

More specifically, expressions are analyzed that are above the aggregate functions, if there are any aggregate functions.

The aggregate functions and everything below them are calculated during aggregation (`GROUP BY`).

These expressions work as if they are applied to separate rows in the result.

DISTINCT Clause

If `DISTINCT` is specified, only a single row will remain out of all the sets of fully matching rows in the result.

The result will be the same as if `GROUP BY` were specified across all the fields specified in `SELECT` without aggregate functions. But there are several differences from `GROUP BY`:

- `DISTINCT` can be applied together with `GROUP BY`.
- When `ORDER BY` is omitted and `LIMIT` is defined, the query stops running immediately after the required number of different rows has been read.
- Data blocks are output as they are processed, without waiting for the entire query to finish running.

`DISTINCT` is not supported if `SELECT` has at least one array column.

`DISTINCT` works with `NULL` as if `NULL` were a specific value, and `NULL=NULL`. In other words, in the `DISTINCT` results, different combinations with `NULL` only occur once.

ClickHouse supports using the `DISTINCT` and `ORDER BY` clauses for different columns in one query. The `DISTINCT` clause is executed before the `ORDER BY` clause.

The sample table:

a	b
2	1
1	2
3	3
2	4

When selecting data by the `SELECT DISTINCT a FROM t1 ORDER BY b ASC` query, we get the following result:

a
2
1
3

If we change the direction of ordering `SELECT DISTINCT a FROM t1 ORDER BY b DESC`, we get the following result:

a
3
1
2

Row 2, 4 was cut before sorting.

Take into account this implementation specificity when programming queries.

LIMIT Clause

`LIMIT m` allows you to select the first `m` rows from the result.

`LIMIT n, m` allows you to select the first `m` rows from the result after skipping the first `n` rows. The `LIMIT m OFFSET n` syntax is also supported.

`n` and `m` must be non-negative integers.

If there isn't an **ORDER BY** clause that explicitly sorts results, the result may be arbitrary and nondeterministic.

UNION ALL Clause

You can use UNION ALL to combine any number of queries. Example:

```
SELECT CounterID, 1 AS table, toInt64(count()) AS c
  FROM test.hits
  GROUP BY CounterID

UNION ALL

SELECT CounterID, 2 AS table, sum(Sign) AS c
  FROM test.visits
  GROUP BY CounterID
  HAVING c > 0
```

Only UNION ALL is supported. The regular UNION (UNION DISTINCT) is not supported. If you need UNION DISTINCT, you can write SELECT DISTINCT from a subquery containing UNION ALL.

Queries that are parts of UNION ALL can be run simultaneously, and their results can be mixed together.

The structure of results (the number and type of columns) must match for the queries. But the column names can differ. In this case, the column names for the final result will be taken from the first query. Type casting is performed for unions. For example, if two queries being combined have the same field with non-**Nullable** and **Nullable** types from a compatible type, the resulting **UNION ALL** has a **Nullable** type field.

Queries that are parts of UNION ALL can't be enclosed in brackets. ORDER BY and LIMIT are applied to separate queries, not to the final result. If you need to apply a conversion to the final result, you can put all the queries with UNION ALL in a subquery in the FROM clause.

INTO OUTFILE Clause

Add the **INTO OUTFILE filename** clause (where filename is a string literal) to redirect query output to the specified file. In contrast to MySQL, the file is created on the client side. The query will fail if a file with the same filename already exists.

This functionality is available in the command-line client and clickhouse-local (a query sent via HTTP interface will fail).

The default output format is TabSeparated (the same as in the command-line client batch mode).

FORMAT Clause

Specify 'FORMAT format' to get data in any specified format.

You can use this for convenience, or for creating dumps.

For more information, see the section "Formats".

If the FORMAT clause is omitted, the default format is used, which depends on both the settings and the interface used for accessing the DB. For the HTTP interface and the command-line client in batch mode, the default format is TabSeparated. For the command-line client in interactive mode, the default format is PrettyCompact (it has attractive and compact tables).

When using the command-line client, data is passed to the client in an internal efficient format. The client independently interprets the FORMAT clause of the query and formats the data itself (thus relieving the network and the server from the load).

IN Operators

The **IN**, **NOT IN**, **GLOBAL IN**, and **GLOBAL NOT IN** operators are covered separately, since their functionality is quite rich.

The left side of the operator is either a single column or a tuple.

Examples:

```
SELECT UserID IN (123, 456) FROM ...  
SELECT (CounterID, UserID) IN ((34, 123), (101500, 456)) FROM ...
```

If the left side is a single column that is in the index, and the right side is a set of constants, the system uses the index for processing the query.

Don't list too many values explicitly (i.e. millions). If a data set is large, put it in a temporary table (for example, see the section "External data for query processing"), then use a subquery.

The right side of the operator can be a set of constant expressions, a set of tuples with constant expressions (shown in the examples above), or the name of a database table or SELECT subquery in brackets.

If the right side of the operator is the name of a table (for example, `UserID IN users`), this is equivalent to the subquery `UserID IN (SELECT * FROM users)`. Use this when working with external data that is sent along with the query. For example, the query can be sent together with a set of user IDs loaded to the 'users' temporary table, which should be filtered.

If the right side of the operator is a table name that has the Set engine (a prepared data set that is always in RAM), the data set will not be created over again for each query.

The subquery may specify more than one column for filtering tuples.

Example:

```
SELECT (CounterID, UserID) IN (SELECT CounterID, UserID FROM ...) FROM ...
```

The columns to the left and right of the IN operator should have the same type.

The IN operator and subquery may occur in any part of the query, including in aggregate functions and lambda functions.

Example:

```
SELECT  
  EventDate,  
  avg(UserID IN  
    (  
      SELECT UserID  
      FROM test.hits  
      WHERE EventDate = toDate('2014-03-17')  
    )) AS ratio  
FROM test.hits  
GROUP BY EventDate  
ORDER BY EventDate ASC
```

EventDate	ratio
2014-03-17	1
2014-03-18	0.807696
2014-03-19	0.755406
2014-03-20	0.723218
2014-03-21	0.697021
2014-03-22	0.647851
2014-03-23	0.648416

For each day after March 17th, count the percentage of pageviews made by users who visited the site on March 17th.

A subquery in the IN clause is always run just one time on a single server. There are no dependent subqueries.

NULL processing

During request processing, the IN operator assumes that the result of an operation with `NULL` is always equal to `0`, regardless of whether `NULL` is on the right or left side of the operator. `NULL` values are not included in any dataset, do not correspond to each other and cannot be compared.

Here is an example with the `t_null` table:

x	y
1	NULL
2	3

Running the query `SELECT x FROM t_null WHERE y IN (NULL,3)` gives you the following result:

x
2

You can see that the row in which `y = NULL` is thrown out of the query results. This is because ClickHouse can't decide whether `NULL` is included in the `(NULL,3)` set, returns `0` as the result of the operation, and `SELECT` excludes this row from the final output.

```
SELECT y IN (NULL, 3)
FROM t_null
```

in(y, tuple(NULL, 3))
0
1

Distributed Subqueries

There are two options for IN-s with subqueries (similar to JOINS): normal `IN / JOIN` and `GLOBAL IN / GLOBAL JOIN`. They differ in how they are run for distributed query processing.

Attention

Remember that the algorithms described below may work differently depending on the `settings distributed_product_mode` setting.

When using the regular `IN`, the query is sent to remote servers, and each of them runs the subqueries in the `IN` or `JOIN` clause.

When using `GLOBAL IN / GLOBAL JOINs`, first all the subqueries are run for `GLOBAL IN / GLOBAL JOINs`, and the results are collected in temporary tables. Then the temporary tables are sent to each remote server, where the queries are run using this temporary data.

For a non-distributed query, use the regular `IN / JOIN`.

Be careful when using subqueries in the `IN / JOIN` clauses for distributed query processing.

Let's look at some examples. Assume that each server in the cluster has a normal **local_table**. Each server also has a **distributed_table** with the **Distributed** type, which looks at all the servers in the cluster.

For a query to the **distributed_table**, the query will be sent to all the remote servers and run on them using the **local_table**.

For example, the query

```
SELECT uniq(UserID) FROM distributed_table
```

will be sent to all remote servers as

```
SELECT uniq(UserID) FROM local_table
```

and run on each of them in parallel, until it reaches the stage where intermediate results can be combined. Then the intermediate results will be returned to the requestor server and merged on it, and the final result will be sent to the client.

Now let's examine a query with `IN`:

```
SELECT uniq(UserID) FROM distributed_table WHERE CounterID = 101500 AND UserID IN (SELECT UserID FROM local_table WHERE CounterID = 34)
```

- Calculation of the intersection of audiences of two sites.

This query will be sent to all remote servers as

```
SELECT uniq(UserID) FROM local_table WHERE CounterID = 101500 AND UserID IN (SELECT UserID FROM local_table WHERE CounterID = 34)
```

In other words, the data set in the IN clause will be collected on each server independently, only across the data that is stored locally on each of the servers.

This will work correctly and optimally if you are prepared for this case and have spread data across the cluster servers such that the data for a single UserID resides entirely on a single server. In this case, all the necessary data will be available locally on each server. Otherwise, the result will be inaccurate. We refer to this variation of the query as "local IN".

To correct how the query works when data is spread randomly across the cluster servers, you could specify **distributed_table** inside a subquery. The query would look like this:

```
SELECT uniq(UserID) FROM distributed_table WHERE CounterID = 101500 AND UserID IN (SELECT UserID FROM distributed_table WHERE CounterID = 34)
```

This query will be sent to all remote servers as

```
SELECT uniq(UserID) FROM local_table WHERE CounterID = 101500 AND UserID IN (SELECT UserID FROM distributed_table WHERE CounterID = 34)
```

The subquery will begin running on each remote server. Since the subquery uses a distributed table, the subquery that is on each remote server will be resent to every remote server as

```
SELECT UserID FROM local_table WHERE CounterID = 34
```

For example, if you have a cluster of 100 servers, executing the entire query will require 10,000 elementary requests, which is generally considered unacceptable.

In such cases, you should always use GLOBAL IN instead of IN. Let's look at how it works for the query

```
SELECT uniq(UserID) FROM distributed_table WHERE CounterID = 101500 AND UserID GLOBAL IN (SELECT UserID FROM distributed_table WHERE CounterID = 34)
```

The requestor server will run the subquery

```
SELECT UserID FROM distributed_table WHERE CounterID = 34
```

and the result will be put in a temporary table in RAM. Then the request will be sent to each remote server as

```
SELECT uniq(UserID) FROM local_table WHERE CounterID = 101500 AND UserID GLOBAL IN _data1
```

and the temporary table `_data1` will be sent to every remote server with the query (the name of the temporary table is implementation-defined).

This is more optimal than using the normal IN. However, keep the following points in mind:

1. When creating a temporary table, data is not made unique. To reduce the volume of data transmitted over the network, specify DISTINCT in the subquery. (You don't need to do this for a normal IN.)
2. The temporary table will be sent to all the remote servers. Transmission does not account for network topology. For example, if 10 remote servers reside in a datacenter that is very remote in relation to the requestor server, the data will be sent 10 times over the channel to the remote datacenter. Try to avoid large data sets when using GLOBAL IN.
3. When transmitting data to remote servers, restrictions on network bandwidth are not configurable. You might overload the network.
4. Try to distribute data across servers so that you don't need to use GLOBAL IN on a regular basis.

5. If you need to use GLOBAL IN often, plan the location of the ClickHouse cluster so that a single group of replicas resides in no more than one data center with a fast network between them, so that a query can be processed entirely within a single data center.

It also makes sense to specify a local table in the **GLOBAL IN** clause, in case this local table is only available on the requestor server and you want to use data from it on remote servers.

Extreme Values

In addition to results, you can also get minimum and maximum values for the results columns. To do this, set the **extremes** setting to 1. Minimums and maximums are calculated for numeric types, dates, and dates with times. For other columns, the default values are output.

An extra two rows are calculated – the minimums and maximums, respectively. These extra two rows are output in JSON*, TabSeparated*, and Pretty* formats, separate from the other rows. They are not output for other formats.

In JSON* formats, the extreme values are output in a separate 'extremes' field. In TabSeparated* formats, the row comes after the main result, and after 'totals' if present. It is preceded by an empty row (after the other data). In Pretty* formats, the row is output as a separate table after the main result, and after 'totals' if present.

Extreme values are calculated for rows that have passed through LIMIT. However, when using 'LIMIT offset, size', the rows before 'offset' are included in 'extremes'. In stream requests, the result may also include a small number of rows that passed through LIMIT.

Notes

The **GROUP BY** and **ORDER BY** clauses do not support positional arguments. This contradicts MySQL, but conforms to standard SQL.

For example, **GROUP BY 1, 2** will be interpreted as grouping by constants (i.e. aggregation of all rows into one).

You can use synonyms (**AS** aliases) in any part of a query.

You can put an asterisk in any part of a query instead of an expression. When the query is analyzed, the asterisk is expanded to a list of all table columns (excluding the **MATERIALIZED** and **ALIAS** columns). There are only a few cases when using an asterisk is justified:

- When creating a table dump.
- For tables containing just a few columns, such as system tables.
- For getting information about what columns are in a table. In this case, set **LIMIT 1**. But it is better to use the **DESC TABLE** query.
- When there is strong filtration on a small number of columns using **PREWHERE**.
- In subqueries (since columns that aren't needed for the external query are excluded from subqueries).

In all other cases, we don't recommend using the asterisk, since it only gives you the drawbacks of a columnar DBMS instead of the advantages. In other words using the asterisk is not recommended.

INSERT

Adding data.

Basic query format:

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

The query can specify a list of columns to insert [(c1, c2, c3)]. In this case, the rest of the columns are filled with:

- The values calculated from the **DEFAULT** expressions specified in the table definition.
- Zeros and empty strings, if **DEFAULT** expressions are not defined.

If **strict_insert_defaults=1**, columns that do not have **DEFAULT** defined must be listed in the query.

Data can be passed to the INSERT in any **format** supported by ClickHouse. The format must be specified explicitly in the query:

```
INSERT INTO [db.]table [(c1, c2, c3)] FORMAT format_name data_set
```

For example, the following query format is identical to the basic version of INSERT ... VALUES:

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

ClickHouse removes all spaces and one line feed (if there is one) before the data. When forming a query, we recommend putting the data on a new line after the query operators (this is important if the data begins with spaces).

Example:

```
INSERT INTO t FORMAT TabSeparated
11 Hello, world!
22 Qwerty
```

You can insert data separately from the query by using the command-line client or the HTTP interface. For more information, see the section "**Interfaces**".

Inserting The Results of **SELECT**

```
INSERT INTO [db.]table [(c1, c2, c3)] SELECT ...
```

Columns are mapped according to their position in the SELECT clause. However, their names in the SELECT expression and the table for INSERT may differ. If necessary, type casting is performed.

None of the data formats except Values allow setting values to expressions such as **now()**, **1 + 2**, and so on. The Values format allows limited use of expressions, but this is not recommended, because in this case inefficient code is used for their execution.

Other queries for modifying data parts are not supported: **UPDATE**, **DELETE**, **REPLACE**, **MERGE**, **UPSERT**, **INSERT UPDATE**. However, you can delete old data using **ALTER TABLE ... DROP PARTITION**.

Performance Considerations

INSERT sorts the input data by primary key and splits them into partitions by month. If you insert data for mixed months, it can significantly reduce the performance of the **INSERT** query. To avoid this:

- Add data in fairly large batches, such as 100,000 rows at a time.
- Group data by month before uploading it to ClickHouse.

Performance will not decrease if:

- Data is added in real time.
- You upload data that is usually sorted by time.

CREATE DATABASE

Creating db_name databases

```
CREATE DATABASE [IF NOT EXISTS] db_name [ON CLUSTER cluster]
```

A **database** is just a directory for tables.

If **IF NOT EXISTS** is included, the query won't return an error if the database already exists.

CREATE TABLE

The **CREATE TABLE** query can have several forms.

```
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
```

Creates a table named 'name' in the 'db' database or the current database if 'db' is not set, with the structure specified in brackets and the 'engine' engine.

The structure of the table is a list of column descriptions. If indexes are supported by the engine, they are indicated as parameters for the table engine.

A column description is **name type** in the simplest case. Example: **RegionID UInt32**. Expressions can also be defined for default values (see below).

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name AS [db2.]name2 [ENGINE = engine]
```

Creates a table with the same structure as another table. You can specify a different engine for the table. If the engine is not specified, the same engine will be used as for the **db2.name2** table.

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name ENGINE = engine AS SELECT ...
```

Creates a table with a structure like the result of the **SELECT** query, with the 'engine' engine, and fills it with data from **SELECT**.

In all cases, if **IF NOT EXISTS** is specified, the query won't return an error if the table already exists. In this case, the query won't do anything.

There can be other clauses after the **ENGINE** clause in the query. See detailed documentation on how to create tables in the descriptions of **table engines**.

Default Values

The column description can specify an expression for a default value, in one of the following ways: **DEFAULT expr**, **MATERIALIZED expr**, **ALIAS expr**.

Example: **URLDomain String DEFAULT domain(URL)**.

If an expression for the default value is not defined, the default values will be set to zeros for numbers, empty strings for strings, empty arrays for arrays, and **0000-00-00** for dates or **0000-00-00 00:00:00** for dates with time. NULLs are not supported.

If the default expression is defined, the column type is optional. If there isn't an explicitly defined type, the default expression type is used. Example: **EventDate DEFAULT toDate(EventTime)** – the 'Date' type will be used for the 'EventDate' column.

If the data type and default expression are defined explicitly, this expression will be cast to the specified type using type casting functions. Example: **Hits UInt32 DEFAULT 0** means the same thing as **Hits UInt32 DEFAULT toUInt32(0)**.

Default expressions may be defined as an arbitrary expression from table constants and columns. When creating and changing the table structure, it checks that expressions don't contain loops. For **INSERT**, it checks that expressions are resolvable – that all columns they can be calculated from have been passed.

DEFAULT expr

Normal default value. If the **INSERT** query doesn't specify the corresponding column, it will be filled in by computing the corresponding expression.

MATERIALIZED expr

Materialized expression. Such a column can't be specified for INSERT, because it is always calculated. For an INSERT without a list of columns, these columns are not considered. In addition, this column is not substituted when using an asterisk in a SELECT query. This is to preserve the invariant that the dump obtained using `SELECT *` can be inserted back into the table using INSERT without specifying the list of columns.

ALIAS expr

Synonym. Such a column isn't stored in the table at all. Its values can't be inserted in a table, and it is not substituted when using an asterisk in a SELECT query. It can be used in SELECTs if the alias is expanded during query parsing.

When using the ALTER query to add new columns, old data for these columns is not written. Instead, when reading old data that does not have values for the new columns, expressions are computed on the fly by default. However, if running the expressions requires different columns that are not indicated in the query, these columns will additionally be read, but only for the blocks of data that need it.

If you add a new column to a table but later change its default expression, the values used for old data will change (for data where values were not stored on the disk). Note that when running background merges, data for columns that are missing in one of the merging parts is written to the merged part.

It is not possible to set default values for elements in nested data structures.

TTL expression

Can be specified only for MergeTree-family tables. An expression for setting storage time for values. It must depends on `Date` or `DateTime` column and has one `Date` or `DateTime` column as a result. Example:

`TTL date + INTERVAL 1 DAY`

You are not allowed to set TTL for key columns. For more details, see [TTL for columns and tables](#)

Column Compression Codecs

Besides default data compression, defined in [server settings](#), per-column specification is also available.

Supported compression algorithms:

- `NONE` - no compression for data applied
- `LZ4`
- `LZ4HC(level)` - (level) - LZ4_HC compression algorithm with defined level. Possible `level` range: [3, 12]. Default value: 9. Greater values stands for better compression and higher CPU usage. Recommended value range: [4,9].
- `ZSTD(level)` - ZSTD compression algorithm with defined `level`. Possible `level` value range: [1, 22]. Default value: 1. Greater values stands for better compression and higher CPU usage.
- `Delta(delta_bytes)` - compression approach when raw values are replace with difference of two neighbour values. Up to `delta_bytes` are used for storing delta value. Possible `delta_bytes` values: 1, 2, 4, 8. Default value for delta bytes is `sizeof(type)`, if it is equals to 1, 2, 4, 8 and equals to 1 otherwise.

Syntax example:

```
CREATE TABLE codec_example
(
    dt Date CODEC(ZSTD), /* используется уровень сжатия по-умолчанию */
    ts DateTime CODEC(LZ4HC),
    float_value Float32 CODEC(NONE),
    double_value Float64 CODEC(LZ4HC(9))
)
ENGINE = MergeTree
PARTITION BY tuple()
ORDER BY dt
```


Codecs can be combined in a pipeline. Default table codec is not included into pipeline (if it should be applied to a column, you have to specify it explicitly in pipeline). Example below shows an optimization approach for storing timeseries metrics.

Usually, values for particular metric, stored in **path** does not differ significantly from point to point. Using delta-encoding allows to reduce disk space usage significantly.

```
CREATE TABLE timeseries_example
(
    dt Date,
    ts DateTime,
    path String,
    value Float32 CODEC(Delta, ZSTD)
)
ENGINE = MergeTree
PARTITION BY dt
ORDER BY (path, ts)
```

Temporary Tables

ClickHouse supports temporary tables which have the following characteristics:

- Temporary tables disappear when the session ends, including if the connection is lost.
- A temporary table use the Memory engine only.
- The DB can't be specified for a temporary table. It is created outside of databases.
- If a temporary table has the same name as another one and a query specifies the table name without specifying the DB, the temporary table will be used.
- For distributed query processing, temporary tables used in a query are passed to remote servers.

To create a temporary table, use the following syntax:

```
CREATE TEMPORARY TABLE [IF NOT EXISTS] table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
)
```

In most cases, temporary tables are not created manually, but when using external data for a query, or for distributed **(GLOBAL) IN**. For more information, see the appropriate sections

Distributed DDL queries (ON CLUSTER clause)

The **CREATE**, **DROP**, **ALTER**, and **RENAME** queries support distributed execution on a cluster.

For example, the following query creates the **all_hits Distributed** table on each host in **cluster**:

```
CREATE TABLE IF NOT EXISTS all_hits ON CLUSTER cluster (p Date, i Int32) ENGINE = Distributed(cluster, default, hits)
```

In order to run these queries correctly, each host must have the same cluster definition (to simplify syncing configs, you can use substitutions from ZooKeeper). They must also connect to the ZooKeeper servers.

The local version of the query will eventually be implemented on each host in the cluster, even if some hosts are currently not available. The order for executing queries within a single host is guaranteed.

ALTER queries are not yet supported for replicated tables.

CREATE VIEW

```
CREATE [MATERIALIZED] VIEW [IF NOT EXISTS] [db.]table_name [TO[db.]name] [ENGINE = engine] [POPULATE] AS SELECT ...
```

Creates a view. There are two types of views: normal and MATERIALIZED.

Normal views don't store any data, but just perform a read from another table. In other words, a normal view is nothing more than a saved query. When reading from a view, this saved query is used as a subquery in the FROM clause.

As an example, assume you've created a view:

```
CREATE VIEW view AS SELECT ...
```

and written a query:

```
SELECT a, b, c FROM view
```

This query is fully equivalent to using the subquery:

```
SELECT a, b, c FROM (SELECT ...)
```

Materialized views store data transformed by the corresponding SELECT query.

When creating a materialized view, you must specify **ENGINE** – the table engine for storing data.

A materialized view is arranged as follows: when inserting data to the table specified in SELECT, part of the inserted data is converted by this SELECT query, and the result is inserted in the view.

If you specify **POPULATE**, the existing table data is inserted in the view when creating it, as if making a **CREATE TABLE ... AS SELECT ...**. Otherwise, the query contains only the data inserted in the table after creating the view. We don't recommend using **POPULATE**, since data inserted in the table during the view creation will not be inserted in it.

A **SELECT** query can contain **DISTINCT**, **GROUP BY**, **ORDER BY**, **LIMIT**... Note that the corresponding conversions are performed independently on each block of inserted data. For example, if **GROUP BY** is set, data is aggregated during insertion, but only within a single packet of inserted data. The data won't be further aggregated. The exception is when using an **ENGINE** that independently performs data aggregation, such as **SummingMergeTree**.

The execution of **ALTER** queries on materialized views has not been fully developed, so they might be inconvenient. If the materialized view uses the construction **TO [db.]name**, you can **DETACH** the view, run **ALTER** for the target table, and then **ATTACH** the previously detached (**DETACH**) view.

Views look the same as normal tables. For example, they are listed in the result of the **SHOW TABLES** query.

There isn't a separate query for deleting views. To delete a view, use **DROP TABLE**.

ALTER

The **ALTER** query is only supported for ***MergeTree** tables, as well as **Merge** and **Distributed**. The query has several variations.

Column Manipulations

Changing the table structure.

```
ALTER TABLE [db].name [ON CLUSTER cluster] ADD|DROP|CLEAR|COMMENT|MODIFY COLUMN ...
```

In the query, specify a list of one or more comma-separated actions. Each action is an operation on a column.

The following actions are supported:

- **ADD COLUMN** — Adds a new column to the table.
- **DROP COLUMN** — Deletes the column.
- **CLEAR COLUMN** — Resets column values.
- **COMMENT COLUMN** — Adds a text comment to the column.
- **MODIFY COLUMN** — Changes column's type and/or default expression.

Detailed description of these actions is shown below.

ADD COLUMN

```
ADD COLUMN [IF NOT EXISTS] name [type] [default_expr] [AFTER name_after]
```

Adds a new column to the table with the specified **name**, **type**, and **default_expr** (see the section [Default expressions](#)).

If the **IF NOT EXISTS** clause is included, the query won't return an error if the column already exists. If you specify **AFTER name_after** (the name of another column), the column is added after the specified one in the list of table columns. Otherwise, the column is added to the end of the table. Note that there is no way to add a column to the beginning of a table. For a chain of actions, **name_after** can be the name of a column that is added in one of the previous actions.

Adding a column just changes the table structure, without performing any actions with data. The data doesn't appear on the disk after **ALTER**. If the data is missing for a column when reading from the table, it is filled in with default values (by performing the default expression if there is one, or using zeros or empty strings). The column appears on the disk after merging data parts (see [MergeTree](#)).

This approach allows us to complete the **ALTER** query instantly, without increasing the volume of old data.

Example:

```
ALTER TABLE visits ADD COLUMN browser String AFTER user_id
```

DROP COLUMN

```
DROP COLUMN [IF EXISTS] name
```

Deletes the column with the name **name**. If the **IF EXISTS** clause is specified, the query won't return an error if the column doesn't exist.

Deletes data from the file system. Since this deletes entire files, the query is completed almost instantly.

Example:

```
ALTER TABLE visits DROP COLUMN browser
```

CLEAR COLUMN

```
CLEAR COLUMN [IF EXISTS] name IN PARTITION partition_name
```

Resets all data in a column for a specified partition. Read more about setting the partition name in the section [How to specify the partition expression](#).

If the **IF EXISTS** clause is specified, the query won't return an error if the column doesn't exist.

Example:

```
ALTER TABLE visits CLEAR COLUMN browser IN PARTITION tuple()
```

COMMENT COLUMN

```
COMMENT COLUMN [IF EXISTS] name 'comment'
```

Adds a comment to the column. If the **IF EXISTS** clause is specified, the query won't return an error if the column doesn't exist.

Each column can have one comment. If a comment already exists for the column, a new comment overwrites the previous comment.

Comments are stored in the **comment_expression** column returned by the **DESCRIBE TABLE** query.

Example:

```
ALTER TABLE visits COMMENT COLUMN browser 'The table shows the browser used for accessing the site.'
```

MODIFY COLUMN

```
MODIFY COLUMN [IF EXISTS] name [type] [default_expr]
```

This query changes the `name` column's type to `type` and/or the default expression to `default_expr`. If the `IF EXISTS` clause is specified, the query won't return an error if the column doesn't exist.

When changing the type, values are converted as if the `toType` functions were applied to them. If only the default expression is changed, the query doesn't do anything complex, and is completed almost instantly.

Example:

```
ALTER TABLE visits MODIFY COLUMN browser Array(String)
```

Changing the column type is the only complex action – it changes the contents of files with data. For large tables, this may take a long time.

There are several processing stages:

- Preparing temporary (new) files with modified data.
- Renaming old files.
- Renaming the temporary (new) files to the old names.
- Deleting the old files.

Only the first stage takes time. If there is a failure at this stage, the data is not changed.

If there is a failure during one of the successive stages, data can be restored manually. The exception is if the old files were deleted from the file system but the data for the new files did not get written to the disk and was lost.

The `ALTER` query for changing columns is replicated. The instructions are saved in ZooKeeper, then each replica applies them. All `ALTER` queries are run in the same order. The query waits for the appropriate actions to be completed on the other replicas. However, a query to change columns in a replicated table can be interrupted, and all actions will be performed asynchronously.

ALTER Query Limitations

The `ALTER` query lets you create and delete separate elements (columns) in nested data structures, but not whole nested data structures. To add a nested data structure, you can add columns with a name like `name.nested_name` and the type `Array(T)`. A nested data structure is equivalent to multiple array columns with a name that has the same prefix before the dot.

There is no support for deleting columns in the primary key or the sampling key (columns that are used in the `ENGINE` expression). Changing the type for columns that are included in the primary key is only possible if this change does not cause the data to be modified (for example, it is allowed to add values to an Enum or to change a type from `DateTime` to `UInt32`).

If the `ALTER` query is not sufficient to make the table changes you need, you can create a new table, copy the data to it using the `INSERT SELECT` query, then switch the tables using the `RENAME` query and delete the old table. You can use the `clickhouse-copier` as an alternative to the `INSERT SELECT` query.

The `ALTER` query blocks all reads and writes for the table. In other words, if a long `SELECT` is running at the time of the `ALTER` query, the `ALTER` query will wait for it to complete. At the same time, all new queries to the same table will wait while this `ALTER` is running.

For tables that don't store data themselves (such as `Merge` and `Distributed`), `ALTER` just changes the table structure, and does not change the structure of subordinate tables. For example, when running `ALTER` for a `Distributed` table, you will also need to run `ALTER` for the tables on all remote servers.

Manipulations With Key Expressions

The following command is supported:

```
MODIFY ORDER BY new_expression
```

It only works for tables in the [MergeTree](#) family (including [replicated](#) tables). The command changes the [sorting key](#) of the table to [new_expression](#) (an expression or a tuple of expressions). Primary key remains the same.

The command is lightweight in a sense that it only changes metadata. To keep the property that data part rows are ordered by the sorting key expression you cannot add expressions containing existing columns to the sorting key (only columns added by the [ADD COLUMN](#) command in the same [ALTER](#) query).

Manipulations With Data Skipping Indices

It only works for tables in the [*MergeTree](#) family (including [replicated](#) tables). The following operations are available:

- [ALTER TABLE \[db\].name ADD INDEX name expression TYPE type GRANULARITY value AFTER name \[AFTER name2\]](#)- Adds index description to tables metadata.
- [ALTER TABLE \[db\].name DROP INDEX name](#) - Removes index description from tables metadata and deletes index files from disk.

These commands are lightweight in a sense that they only change metadata or remove files. Also, they are replicated (syncing indices metadata through ZooKeeper).

Manipulations With Partitions and Parts

The following operations with [partitions](#) are available:

- [DETACH PARTITION](#) - Moves a partition to the [detached](#) directory and forget it.
- [DROP PARTITION](#) - Deletes a partition.
- [ATTACH PART|PARTITION](#) - Adds a part or partition from the [detached](#) directory to the table.
- [REPLACE PARTITION](#) - Copies the data partition from one table to another.
- [CLEAR COLUMN IN PARTITION](#) - Resets the value of a specified column in a partition.
- [FREEZE PARTITION](#) - Creates a backup of a partition.
- [FETCH PARTITION](#) - Downloads a partition from another server.

DETACH PARTITION

```
ALTER TABLE table_name DETACH PARTITION partition_expr
```

Moves all data for the specified partition to the [detached](#) directory. The server forgets about the detached data partition as if it does not exist. The server will not know about this data until you make the [ATTACH](#) query.

Example:

```
ALTER TABLE visits DETACH PARTITION 201901
```

Read about setting the partition expression in a section [How to specify the partition expression](#).

After the query is executed, you can do whatever you want with the data in the [detached](#) directory — delete it from the file system, or just leave it.

This query is replicated – it moves the data to the [detached](#) directory on all replicas. Note that you can execute this query only on a leader replica. To find out if a replica is a leader, perform the [SELECT](#) query to the [system.replicas](#) table. Alternatively, it is easier to make a [DETACH](#) query on all replicas - all the replicas throw an exception, except the leader replica.

DROP PARTITION

```
ALTER TABLE table_name DROP PARTITION partition_expr
```

Deletes the specified partition from the table. This query tags the partition as inactive and deletes data completely, approximately in 10 minutes.

Read about setting the partition expression in a section [How to specify the partition expression](#).

The query is replicated – it deletes data on all replicas.

ATTACH PARTITION|PART

```
ALTER TABLE table_name ATTACH PARTITION|PART partition_expr
```

Adds data to the table from the **detached** directory. It is possible to add data for an entire partition or for a separate part. Examples:

```
ALTER TABLE visits ATTACH PARTITION 201901;  
ALTER TABLE visits ATTACH PART 201901_2_2_0;
```

Read more about setting the partition expression in a section [How to specify the partition expression](#).

This query is replicated. Each replica checks whether there is data in the **detached** directory. If the data is in this directory, the query checks the integrity, verifies that it matches the data on the server that initiated the query. If everything is correct, the query adds data to the replica. If not, it downloads data from the query requestor replica, or from another replica where the data has already been added.

So you can put data to the **detached** directory on one replica, and use the **ALTER ... ATTACH** query to add it to the table on all replicas.

REPLACE PARTITION

```
ALTER TABLE table2 REPLACE PARTITION partition_expr FROM table1
```

This query copies the data partition from the **table1** to **table2**. Note that data won't be deleted from **table1**.

For the query to run successfully, the following conditions must be met:

- Both tables must have the same structure.
- Both tables must have the same partition key.

CLEAR COLUMN IN PARTITION

```
ALTER TABLE table_name CLEAR COLUMN column_name IN PARTITION partition_expr
```

Resets all values in the specified column in a partition. If the **DEFAULT** clause was determined when creating a table, this query sets the column value to a specified default value.

Example:

```
ALTER TABLE visits CLEAR COLUMN hour in PARTITION 201902
```

FREEZE PARTITION

```
ALTER TABLE table_name FREEZE [PARTITION partition_expr]
```

This query creates a local backup of a specified partition. If the **PARTITION** clause is omitted, the query creates the backup of all partitions at once.

Note that for old-styled tables you can specify the prefix of the partition name (for example, '2019') - then the query creates the backup for all the corresponding partitions. Read about setting the partition expression in a section [How to specify the partition expression](#).

Note

The entire backup process is performed without stopping the server.

At the time of execution, for a data snapshot, the query creates hardlinks to a table data. Hardlinks are placed in the directory **/var/lib/clickhouse/shadow/N/...**, where:

- **/var/lib/clickhouse/** is the working ClickHouse directory specified in the config.
- **N** is the incremental number of the backup.

The same structure of directories is created inside the backup as inside `/var/lib/clickhouse/`. The query performs 'chmod' for all files, forbidding writing into them.

After creating the backup, you can copy the data from `/var/lib/clickhouse/shadow/` to the remote server and then delete it from the local server. Note that the `ALTER t FREEZE PARTITION` query is not replicated. It creates a local backup only on the local server.

The query creates backup almost instantly (but first it waits for the current queries to the corresponding table to finish running).

`ALTER TABLE t FREEZE PARTITION` copies only the data, not table metadata. To make a backup of table metadata, copy the file `/var/lib/clickhouse/metadata/database/table.sql`

To restore data from a backup, do the following:

1. Create the table if it does not exist. To view the query, use the .sql file (replace `ATTACH` in it with `CREATE`).
2. Copy the data from the `data/database/table/` directory inside the backup to the `/var/lib/clickhouse/data/database/table/detached/` directory.
3. Run `ALTER TABLE t ATTACH PARTITION` queries to add the data to a table.

Restoring from a backup doesn't require stopping the server.

For more information about backups and restoring data, see the [Data Backup](#) section.

FETCH PARTITION

```
ALTER TABLE table_name FETCH PARTITION partition_expr FROM 'path-in-zookeeper'
```

Downloads a partition from another server. This query only works for the replicated tables.

The query does the following:

1. Downloads the partition from the specified shard. In 'path-in-zookeeper' you must specify a path to the shard in ZooKeeper.
2. Then the query puts the downloaded data to the `detached` directory of the `table_name` table. Use the `ATTACH PARTITION|PART` query to add the data to the table.

For example:

```
ALTER TABLE users FETCH PARTITION 201902 FROM '/clickhouse/tables/01-01/visits';  
ALTER TABLE users ATTACH PARTITION 201902;
```

Note that:

- The `ALTER ... FETCH PARTITION` query isn't replicated. It places the partition to the `detached` directory only on the local server.
- The `ALTER TABLE ... ATTACH` query is replicated. It adds the data to all replicas. The data is added to one of the replicas from the `detached` directory, and to the others - from neighboring replicas.

Before downloading, the system checks if the partition exists and the table structure matches. The most appropriate replica is selected automatically from the healthy replicas.

Although the query is called `ALTER TABLE`, it does not change the table structure and does not immediately change the data available in the table.

How To Set Partition Expression

You can specify the partition expression in `ALTER ... PARTITION` queries in different ways:

- As a value from the `partition` column of the `system.parts` table. For example, `ALTER TABLE visits DETACH PARTITION 201901`.
- As the expression from the table column. Constants and constant expressions are supported. For example, `ALTER TABLE visits DETACH PARTITION toYYYYMM(toDate('2019-01-25'))`.

- Using the partition ID. Partition ID is a string identifier of the partition (human-readable, if possible) that is used as the names of partitions in the file system and in ZooKeeper. The partition ID must be specified in the `PARTITION ID` clause, in a single quotes. For example, `ALTER TABLE visits DETACH PARTITION ID '201901'`.
- In the `ALTER ATTACH PART` query, to specify the name of a part, use a value from the `name` column of the `system.parts` table. For example, `ALTER TABLE visits ATTACH PART 201901_1_1_0`.

Usage of quotes when specifying the partition depends on the type of partition expression. For example, for the `String` type, you have to specify its name in quotes ('). For the `Date` and `Int*` types no quotes are needed.

For old-style tables, you can specify the partition either as a number `201901` or a string `'201901'`. The syntax for the new-style tables is stricter with types (similar to the parser for the `VALUES` input format).

All the rules above are also true for the `OPTIMIZE` query. If you need to specify the only partition when optimizing a non-partitioned table, set the expression `PARTITION tuple()`. For example:

```
OPTIMIZE TABLE table_not_partitioned PARTITION tuple() FINAL;
```

The examples of `ALTER ... PARTITION` queries are demonstrated in the tests `00502_custom_partitioning_local` and `00502_custom_partitioning_replicated_zookeeper`.

Synchronicity of ALTER Queries

For non-replicable tables, all `ALTER` queries are performed synchronously. For replicatable tables, the query just adds instructions for the appropriate actions to `ZooKeeper`, and the actions themselves are performed as soon as possible. However, the query can wait for these actions to be completed on all the replicas.

For `ALTER ... ATTACH|DETACH|DROP` queries, you can use the `replication_alter_partitions_sync` setting to set up waiting. Possible values: `0` – do not wait; `1` – only wait for own execution (default); `2` – wait for all.

Mutations

Mutations are an `ALTER` query variant that allows changing or deleting rows in a table. In contrast to standard `UPDATE` and `DELETE` queries that are intended for point data changes, mutations are intended for heavy operations that change a lot of rows in a table.

The functionality is in beta stage and is available starting with the 1.1.54388 version. Currently `*MergeTree` table engines are supported (both replicated and unreplicated).

Existing tables are ready for mutations as-is (no conversion necessary), but after the first mutation is applied to a table, its metadata format becomes incompatible with previous server versions and falling back to a previous version becomes impossible.

Currently available commands:

```
ALTER TABLE [db.]table DELETE WHERE filter_expr
```

The `filter_expr` must be of type `UInt8`. The query deletes rows in the table for which this expression takes a non-zero value.

```
ALTER TABLE [db.]table UPDATE column1 = expr1 [, ...] WHERE filter_expr
```

The command is available starting with the 18.12.14 version. The `filter_expr` must be of type `UInt8`. This query updates values of specified columns to the values of corresponding expressions in rows for which the `filter_expr` takes a non-zero value. Values are casted to the column type using the `CAST` operator. Updating columns that are used in the calculation of the primary or the partition key is not supported.

One query can contain several commands separated by commas.

For `*MergeTree` tables mutations execute by rewriting whole data parts. There is no atomicity - parts are substituted for mutated parts as soon as they are ready and a `SELECT` query that started executing during a mutation will see data from parts that have already been mutated along with data from parts that have not been mutated yet.

Mutations are totally ordered by their creation order and are applied to each part in that order. Mutations are also partially ordered with INSERTs - data that was inserted into the table before the mutation was submitted will be mutated and data that was inserted after that will not be mutated. Note that mutations do not block INSERTs in any way.

A mutation query returns immediately after the mutation entry is added (in case of replicated tables to ZooKeeper, for nonreplicated tables - to the filesystem). The mutation itself executes asynchronously using the system profile settings. To track the progress of mutations you can use the `system.mutations` table. A mutation that was successfully submitted will continue to execute even if ClickHouse servers are restarted. There is no way to roll back the mutation once it is submitted, but if the mutation is stuck for some reason it can be cancelled with the `KILL MUTATION` query.

Entries for finished mutations are not deleted right away (the number of preserved entries is determined by the `finished_mutations_to_keep` storage engine parameter). Older mutation entries are deleted.

Miscellaneous Queries

ATTACH

This query is exactly the same as `CREATE`, but

- Instead of the word `CREATE` it uses the word `ATTACH`.
 - The query does not create data on the disk, but assumes that data is already in the appropriate places, and just adds information about the table to the server.
- After executing an `ATTACH` query, the server will know about the existence of the table.

If the table was previously detached (`DETACH`), meaning that its structure is known, you can use shorthand without defining the structure.

```
ATTACH TABLE [IF NOT EXISTS] [db.]name [ON CLUSTER cluster]
```

This query is used when starting the server. The server stores table metadata as files with `ATTACH` queries, which it simply runs at launch (with the exception of system tables, which are explicitly created on the server).

CHECK TABLE

Checks if the data in the table is corrupted.

```
CHECK TABLE [db.]name
```

The `CHECK TABLE` query compares actual file sizes with the expected values which are stored on the server. If the file sizes do not match the stored values, it means the data is corrupted. This can be caused, for example, by a system crash during query execution.

The query response contains the `result` column with a single row. The row has a value of `Boolean` type:

- 0 - The data in the table is corrupted.
- 1 - The data maintains integrity.

The `CHECK TABLE` query is only supported for the following table engines:

- `Log`
- `TinyLog`
- `StripeLog`

These engines do not provide automatic data recovery on failure. Use the `CHECK TABLE` query to track data loss in a timely manner.

To avoid data loss use the `MergeTree` family tables.

If the data is corrupted

If the table is corrupted, you can copy the non-corrupted data to another table. To do this:

1. Create a new table with the same structure as damaged table. To do this execute the query `CREATE TABLE <new_table_name> AS <damaged_table_name>`.
2. Set the `max_threads` value to 1 to process the next query in a single thread. To do this run the query `SET max_threads = 1`.
3. Execute the query `INSERT INTO <new_table_name> SELECT * FROM <damaged_table_name>`. This request copies the non-corrupted data from the damaged table to another table. Only the data before the corrupted part will be copied.
4. Restart the `clickhouse-client` to reset the `max_threads` value.

DESCRIBE TABLE

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

Returns the following `String` type columns:

- `name` — Column name.
- `type` — Column type.
- `default_type` — Clause that is used in `default expression` (`DEFAULT`, `MATERIALIZED` or `ALIAS`). Column contains an empty string, if the default expression isn't specified.
- `default_expression` — Value specified in the `DEFAULT` clause.
- `comment_expression` — Comment text.

Nested data structures are output in "expanded" format. Each column is shown separately, with the name after a dot.

DETACH

Deletes information about the 'name' table from the server. The server stops knowing about the table's existence.

```
DETACH TABLE [IF EXISTS] [db.]name [ON CLUSTER cluster]
```

This does not delete the table's data or metadata. On the next server launch, the server will read the metadata and find out about the table again.

Similarly, a "detached" table can be re-attached using the `ATTACH` query (with the exception of system tables, which do not have metadata stored for them).

There is no `DETACH DATABASE` query.

DROP

This query has two types: `DROP DATABASE` and `DROP TABLE`.

```
DROP DATABASE [IF EXISTS] db [ON CLUSTER cluster]
```

Deletes all tables inside the 'db' database, then deletes the 'db' database itself.

If `IF EXISTS` is specified, it doesn't return an error if the database doesn't exist.

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

Deletes the table.

If `IF EXISTS` is specified, it doesn't return an error if the table doesn't exist or the database doesn't exist.

EXISTS

```
EXISTS [TEMPORARY] TABLE [db.]name [INTO OUTFILE filename] [FORMAT format]
```

Returns a single `UInt8`-type column, which contains the single value `0` if the table or database doesn't exist, or `1` if the table exists in the specified database.

KILL QUERY

```
KILL QUERY [ON CLUSTER cluster]
WHERE <where expression to SELECT FROM system.processes query>
[SYNC|ASYNC|TEST]
[FORMAT format]
```

Attempts to forcibly terminate the currently running queries.

The queries to terminate are selected from the `system.processes` table using the criteria defined in the `WHERE` clause of the `KILL` query.

Examples:

```
-- Forcibly terminates all queries with the specified query_id:
KILL QUERY WHERE query_id='2-857d-4a57-9ee0-327da5d60a90'

-- Synchronously terminates all queries run by 'username':
KILL QUERY WHERE user='username' SYNC
```

Read-only users can only stop their own queries.

By default, the asynchronous version of queries is used (`ASYNC`), which doesn't wait for confirmation that queries have stopped.

The synchronous version (`SYNC`) waits for all queries to stop and displays information about each process as it stops.

The response contains the `kill_status` column, which can take the following values:

1. 'finished' – The query was terminated successfully.
2. 'waiting' – Waiting for the query to end after sending it a signal to terminate.
3. The other values explain why the query can't be stopped.

A test query (`TEST`) only checks the user's rights and displays a list of queries to stop.

KILL MUTATION

```
KILL MUTATION [ON CLUSTER cluster]
WHERE <where expression to SELECT FROM system.mutations query>
[TEST]
[FORMAT format]
```

Tries to cancel and remove `mutations` that are currently executing. Mutations to cancel are selected from the `system.mutations` table using the filter specified by the `WHERE` clause of the `KILL` query.

A test query (`TEST`) only checks the user's rights and displays a list of queries to stop.

Examples:

```
-- Cancel and remove all mutations of the single table:
KILL MUTATION WHERE database = 'default' AND table = 'table'

-- Cancel the specific mutation:
KILL MUTATION WHERE database = 'default' AND table = 'table' AND mutation_id = 'mutation_3.txt'
```

The query is useful when a mutation is stuck and cannot finish (e.g. if some function in the mutation query throws an exception when applied to the data contained in the table).

Changes already made by the mutation are not rolled back.

OPTIMIZE

```
OPTIMIZE TABLE [db.]name [ON CLUSTER cluster] [PARTITION partition] [FINAL]
```

Asks the table engine to do something for optimization.

Supported only by ***MergeTree** engines, in which this query initializes a non-scheduled merge of data parts.

If you specify a **PARTITION**, only the specified partition will be optimized.

If you specify **FINAL**, optimization will be performed even when all the data is already in one part.

Warning

OPTIMIZE can't fix the "Too many parts" error.

RENAME

Renames one or more tables.

```
RENAME TABLE [db11.]name11 TO [db12.]name12, [db21.]name21 TO [db22.]name22, ... [ON CLUSTER cluster]
```

All tables are renamed under global locking. Renaming tables is a light operation. If you indicated another database after TO, the table will be moved to this database. However, the directories with databases must reside in the same file system (otherwise, an error is returned).

SET

```
SET param = value
```

Allows you to set **param** to **value**. You can also make all the settings from the specified settings profile in a single query. To do this, specify 'profile' as the setting name. For more information, see the section "Settings".

The setting is made for the session, or for the server (globally) if **GLOBAL** is specified.

When making a global setting, the setting is not applied to sessions already running, including the current session. It will only be used for new sessions.

When the server is restarted, global settings made using **SET** are lost.

To make settings that persist after a server restart, you can only use the server's config file.

SHOW CREATE TABLE

```
SHOW CREATE [TEMPORARY] TABLE [db.]table [INTO OUTFILE filename] [FORMAT format]
```

Returns a single **String**-type 'statement' column, which contains a single value – the **CREATE** query used for creating the specified table.

SHOW DATABASES

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

Prints a list of all databases.

This query is identical to **SELECT name FROM system.databases [INTO OUTFILE filename] [FORMAT format]**

See also the section "Formats".

SHOW PROCESSLIST

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

Outputs a list of queries currently being processed, other than **SHOW PROCESSLIST** queries.

Prints a table containing the columns:

user – The user who made the query. Keep in mind that for distributed processing, queries are sent to remote servers under the 'default' user. SHOW PROCESSLIST shows the username for a specific query, not for a query that this query initiated.

address – The name of the host that the query was sent from. For distributed processing, on remote servers, this is the name of the query requestor host. To track where a distributed query was originally made from, look at SHOW PROCESSLIST on the query requestor server.

elapsed – The execution time, in seconds. Queries are output in order of decreasing execution time.

rows_read, bytes_read – How many rows and bytes of uncompressed data were read when processing the query. For distributed processing, data is totaled from all the remote servers. This is the data used for restrictions and quotas.

memory_usage – Current RAM usage in bytes. See the setting 'max_memory_usage'.

query – The query itself. In INSERT queries, the data for insertion is not output.

query_id – The query identifier. Non-empty only if it was explicitly defined by the user. For distributed processing, the query ID is not passed to remote servers.

This query is nearly identical to: `SELECT * FROM system.processes`. The difference is that the `SHOW PROCESSLIST` query does not show itself in a list, when the `SELECT .. FROM system.processes` query does.

Tip (execute in the console):

```
watch -n1 "clickhouse-client --query='SHOW PROCESSLIST'"
```

SHOW TABLES

```
SHOW [TEMPORARY] TABLES [FROM db] [LIKE 'pattern'] [INTO OUTFILE filename] [FORMAT format]
```

Displays a list of tables

- Tables from the current database, or from the 'db' database if "FROM db" is specified.
- All tables, or tables whose name matches the pattern, if "LIKE 'pattern'" is specified.

This query is identical to: `SELECT name FROM system.tables WHERE database = 'db' [AND name LIKE 'pattern'] [INTO OUTFILE filename] [FORMAT format]`.

See also the section "LIKE operator".

TRUNCATE

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

Removes all data from a table. When the clause `IF EXISTS` is omitted, the query returns an error if the table does not exist.

The `TRUNCATE` query is not supported for `View`, `File`, `URL` and `Null` table engines.

USE

```
USE db
```

Lets you set the current database for the session.

The current database is used for searching for tables if the database is not explicitly defined in the query with a dot before the table name.

This query can't be made when using the HTTP protocol, since there is no concept of a session.

Functions

There are at least* two types of functions - regular functions (they are just called "functions") and aggregate functions. These are completely different concepts. Regular functions work as if they are applied to each row separately (for each row, the result of the function doesn't depend on the other rows). Aggregate functions accumulate a set of values from various rows (i.e. they depend on the entire set of rows).

In this section we discuss regular functions. For aggregate functions, see the section "Aggregate functions".

* - There is a third type of function that the 'arrayJoin' function belongs to; table functions can also be mentioned separately.*

Strong typing

In contrast to standard SQL, ClickHouse has strong typing. In other words, it doesn't make implicit conversions between types. Each function works for a specific set of types. This means that sometimes you need to use type conversion functions.

Common subexpression elimination

All expressions in a query that have the same AST (the same record or same result of syntactic parsing) are considered to have identical values. Such expressions are concatenated and executed once. Identical subqueries are also eliminated this way.

Types of results

All functions return a single result as the result (not several values, and not zero values). The type of result is usually defined only by the types of arguments, not by the values. Exceptions are the `tupleElement` function (the `a.N` operator), and the `toFixedString` function.

Constants

For simplicity, certain functions can only work with constants for some arguments. For example, the right argument of the `LIKE` operator must be a constant.

Almost all functions return a constant for constant arguments. The exception is functions that generate random numbers.

The `'now'` function returns different values for queries that were run at different times, but the result is considered a constant, since constancy is only important within a single query.

A constant expression is also considered a constant (for example, the right half of the `LIKE` operator can be constructed from multiple constants).

Functions can be implemented in different ways for constant and non-constant arguments (different code is executed). But the results for a constant and for a true column containing only the same value should match each other.

NULL processing

Functions have the following behaviors:

- If at least one of the arguments of the function is `NULL`, the function result is also `NULL`.
- Special behavior that is specified individually in the description of each function. In the ClickHouse source code, these functions have `UseDefaultImplementationForNulls=false`.

Constancy

Functions can't change the values of their arguments – any changes are returned as the result. Thus, the result of calculating separate functions does not depend on the order in which the functions are written in the query.

Error handling

Some functions might throw an exception if the data is invalid. In this case, the query is canceled and an error text is returned to the client. For distributed processing, when an exception occurs on one of the servers, the other servers also attempt to abort the query.

Evaluation of argument expressions

In almost all programming languages, one of the arguments might not be evaluated for certain operators. This is usually the operators `&&`, `||`, and `?:`.

But in ClickHouse, arguments of functions (operators) are always evaluated. This is because entire parts of columns are evaluated at once, instead of calculating each row separately.

Performing functions for distributed query processing

For distributed query processing, as many stages of query processing as possible are performed on remote servers, and the rest of the stages (merging intermediate results and everything after that) are performed on the requestor server.

This means that functions can be performed on different servers.

For example, in the query `SELECT f(sum(g(x))) FROM distributed_table GROUP BY h(y),`

- if a `distributed_table` has at least two shards, the functions 'g' and 'h' are performed on remote servers, and the function 'f' is performed on the requestor server.
- if a `distributed_table` has only one shard, all the 'f', 'g', and 'h' functions are performed on this shard's server.

The result of a function usually doesn't depend on which server it is performed on. However, sometimes this is important.

For example, functions that work with dictionaries use the dictionary that exists on the server they are running on. Another example is the `hostName` function, which returns the name of the server it is running on in order to make `GROUP BY` by servers in a `SELECT` query.

If a function in a query is performed on the requestor server, but you need to perform it on remote servers, you can wrap it in an 'any' aggregate function or add it to a key in `GROUP BY`.

Arithmetic functions

For all arithmetic functions, the result type is calculated as the smallest number type that the result fits in, if there is such a type. The minimum is taken simultaneously based on the number of bits, whether it is signed, and whether it floats. If there are not enough bits, the highest bit type is taken.

Example:

SELECT toTypeName(0), toTypeName(0 + 0), toTypeName(0 + 0 + 0), toTypeName(0 + 0 + 0 + 0)				
toTypeName(0)	toTypeName(plus(0, 0))	toTypeName(plus(plus(0, 0), 0))	toTypeName(plus(plus(plus(0, 0), 0), 0))	
UInt8	UInt16	UInt32	UInt64	

Arithmetic functions work for any pair of types from UInt8, UInt16, UInt32, UInt64, Int8, Int16, Int32, Int64, Float32, or Float64.

Overflow is produced the same way as in C++.

plus(a, b), a + b operator

Calculates the sum of the numbers.

You can also add integer numbers with a date or date and time. In the case of a date, adding an integer means adding the corresponding number of days. For a date with time, it means adding the corresponding number of seconds.

minus(a, b), a - b operator

Calculates the difference. The result is always signed.

You can also calculate integer numbers from a date or date with time. The idea is the same – see above for 'plus'.

multiply(a, b), a * b operator

Calculates the product of the numbers.

divide(a, b), a / b operator

Calculates the quotient of the numbers. The result type is always a floating-point type.

It is not integer division. For integer division, use the 'intDiv' function.

When dividing by zero you get 'inf', '-inf', or 'nan'.

intDiv(a, b)

Calculates the quotient of the numbers. Divides into integers, rounding down (by the absolute value).

An exception is thrown when dividing by zero or when dividing a minimal negative number by minus one.

intDivOrZero(a, b)

Differs from 'intDiv' in that it returns zero when dividing by zero or when dividing a minimal negative number by minus one.

modulo(a, b), a % b operator

Calculates the remainder after division.

If arguments are floating-point numbers, they are pre-converted to integers by dropping the decimal portion.

The remainder is taken in the same sense as in C++. Truncated division is used for negative numbers.

An exception is thrown when dividing by zero or when dividing a minimal negative number by minus one.

negate(a), -a operator

Calculates a number with the reverse sign. The result is always signed.

abs(a)

Calculates the absolute value of the number (a). That is, if $a < 0$, it returns $-a$. For unsigned types it doesn't do anything. For signed integer types, it returns an unsigned number.

gcd(a, b)

Returns the greatest common divisor of the numbers.

An exception is thrown when dividing by zero or when dividing a minimal negative number by minus one.

lcm(a, b)

Returns the least common multiple of the numbers.

An exception is thrown when dividing by zero or when dividing a minimal negative number by minus one.

Comparison functions

Comparison functions always return 0 or 1 (UInt8).

The following types can be compared:

- numbers
- strings and fixed strings
- dates
- dates with times

within each group, but not between different groups.

For example, you can't compare a date with a string. You have to use a function to convert the string to a date, or vice versa.

Strings are compared by bytes. A shorter string is smaller than all strings that start with it and that contain at least one more character.

Note. Up until version 1.1.54134, signed and unsigned numbers were compared the same way as in C++. In other words, you could get an incorrect result in cases like `SELECT 9223372036854775807 > -1`. This behavior changed in version 1.1.54134 and is now mathematically correct.

equals, a = b and a == b operator

notEquals, a != b and a <> b

less, < operator

greater, > operator

lessOrEquals, <= operator

greaterOrEquals, >= operator

Logical functions

Logical functions accept any numeric types, but return a UInt8 number equal to 0 or 1.

Zero as an argument is considered "false," while any non-zero value is considered "true".

and, AND operator

or, OR operator

not, NOT operator

xor

Type conversion functions

toUInt8, toUInt16, toUInt32, toUInt64

toInt8, toInt16, toInt32, toInt64

toFloat32, toFloat64

toDate, toDateTime

toUInt8OrZero, toUInt16OrZero, toUInt32OrZero, toUInt64OrZero,
toInt8OrZero, toInt16OrZero, toInt32OrZero, toInt64OrZero,
toFloat32OrZero, toFloat64OrZero, toDateOrZero,
toDateTimeOrZero

toUInt8OrNull, toUInt16OrNull, toUInt32OrNull, toUInt64OrNull,
toInt8OrNull, toInt16OrNull, toInt32OrNull, toInt64OrNull,
toFloat32OrNull, toFloat64OrNull, toDateOrNull,
toDateTimeOrNull

toDecimal32(value, S), toDecimal64(value, S),
toDecimal128(value, S)

Converts **value** to **Decimal** of precision **S**. The **value** can be a number or a string. The **S** (scale) parameter specifies the number of decimal places.

toString

Functions for converting between numbers, strings (but not fixed strings), dates, and dates with times.
All these functions accept one argument.

When converting to or from a string, the value is formatted or parsed using the same rules as for the TabSeparated format (and almost all other text formats). If the string can't be parsed, an exception is thrown and the request is canceled.

When converting dates to numbers or vice versa, the date corresponds to the number of days since the beginning of the Unix epoch.

When converting dates with times to numbers or vice versa, the date with time corresponds to the number of seconds since the beginning of the Unix epoch.

The date and date-with-time formats for the `toDate/toDateTime` functions are defined as follows:

```
YYYY-MM-DD
YYYY-MM-DD hh:mm:ss
```

As an exception, if converting from `UInt32`, `Int32`, `UInt64`, or `Int64` numeric types to `Date`, and if the number is greater than or equal to 65536, the number is interpreted as a Unix timestamp (and not as the number of days) and is rounded to the date. This allows support for the common occurrence of writing `'toDate(unix_timestamp)'`, which otherwise would be an error and would require writing the more cumbersome `'toDate(toDateTime(unix_timestamp))'`.

Conversion between a date and date with time is performed the natural way: by adding a null time or dropping the time.

Conversion between numeric types uses the same rules as assignments between different numeric types in C++.

Additionally, the `toString` function of the `DateTime` argument can take a second `String` argument containing the name of the time zone. Example: `Asia/Yekaterinburg`. In this case, the time is formatted according to the specified time zone.

```
SELECT
  now() AS now_local,
  toString(now(), 'Asia/Yekaterinburg') AS now_yekat
```

```
┌──────────now_local──┐┌──────────now_yekat──┐
│ 2016-06-15 00:11:21 │ │ 2016-06-15 02:11:21 │
└──────────┘          └──────────┘
```

Also see the `toUnixTimestamp` function.

toFixedString(s, N)

Converts a `String` type argument to a `FixedString(N)` type (a string with fixed length `N`). `N` must be a constant. If the string has fewer bytes than `N`, it is passed with null bytes to the right. If the string has more bytes than `N`, an exception is thrown.

toStringCutToZero(s)

Accepts a `String` or `FixedString` argument. Returns the `String` with the content truncated at the first zero byte found.

Example:

```
SELECT toFixedString('foo', 8) AS s, toStringCutToZero(s) AS s_cut
```

```
┌──s──┐┌──s_cut──┐
│foo\0\0\0\0\0│ │foo │
└──┘   └──┘
```

```
SELECT toFixedString('foo\0bar', 8) AS s, toStringCutToZero(s) AS s_cut
```

```
┌──s──┐┌──s_cut──┐
│foo\0bar\0│ │foo │
└──┘   └──┘
```

`reinterpretAsUInt8`, `reinterpretAsUInt16`, `reinterpretAsUInt32`,
`reinterpretAsUInt64`

reinterpretAsDate, reinterpretAsDateTime

These functions accept a string and interpret the bytes placed at the beginning of the string as a number in host order (little endian). If the string isn't long enough, the functions work as if the string is padded with the necessary number of null bytes. If the string is longer than needed, the extra bytes are ignored. A date is interpreted as the number of days since the beginning of the Unix Epoch, and a date with time is interpreted as the number of seconds since the beginning of the Unix Epoch.

reinterpretAsString

This function accepts a number or date or date with time, and returns a string containing bytes representing the corresponding value in host order (little endian). Null bytes are dropped from the end. For example, a UInt32 type value of 255 is a string that is one byte long.

reinterpretAsString

This function accepts a number or date or date with time, and returns a `FixedString` containing bytes representing the corresponding value in host order (little endian). Null bytes are dropped from the end. For example, a `UInt32` type value of 255 is a `FixedString` that is one byte long.

CAST(x, t)

Converts 'x' to the 't' data type. The syntax CAST(x AS t) is also supported.

Example:

```
SELECT
  '2016-06-15 23:00:00' AS timestamp,
  CAST(timestamp AS DateTime) AS datetime,
  CAST(timestamp AS Date) AS date,
  CAST(timestamp, 'String') AS string,
  CAST(timestamp, 'FixedString(22)') AS fixed_string
```

timestamp	datetime	date	string	fixed_string
2016-06-15 23:00:00	2016-06-15 23:00:00	2016-06-15	2016-06-15 23:00:00	2016-06-15 23:00:00\0\0\0

Conversion to `FixedString(N)` only works for arguments of type `String` or `FixedString(N)`.

Type conversion to **Nullable** and back is supported. Example:

```
SELECT toTypeName(x) FROM t null
```

```
┌toTypeName(x)┐
| Int8         |
| Int8         |
|              |
|              |
```

```
SELECT toTypeName(CAST(x, 'Nullable(UInt16)')) FROM t null
```

```
└─toTypeName(CAST(x, 'Nullable(UInt16)'))┐
| Nullable(UInt16)                        |
| Nullable(UInt16)                        |
```

toIntervalYear, toIntervalQuarter, toIntervalMonth, toIntervalWeek, toIntervalDay, toIntervalHour, toIntervalMinute, toIntervalSecond

Converts a Number type argument to a Interval type (duration).

The interval type is actually very useful, you can use this type of data to perform arithmetic operations directly with Date or DateTime. At the same time, ClickHouse provides a more convenient syntax for declaring Interval type data. For example:

```
WITH
    toDate('2019-01-01') AS date,
    INTERVAL 1 WEEK AS interval_week,
    toIntervalWeek(1) AS interval_to_week
SELECT
    date + interval_week,
    date + interval_to_week
```

plus(date, interval_week)	plus(date, interval_to_week)
2019-01-08	2019-01-08

parseDateTimeBestEffort

Parse a number type argument to a Date or DateTime type.

different from toDate and toDateTime, parseDateTimeBestEffort can progress more complex date format.

For more information, see the link: [Complex Date Format](#)

parseDateTimeBestEffortOrNull

Same as for [parseDateTimeBestEffort](#) except that it returns null when it encounters a date format that cannot be processed.

parseDateTimeBestEffortOrZero

Same as for [parseDateTimeBestEffort](#) except that it returns zero date or zero date time when it encounters a date format that cannot be processed.

Functions for working with dates and times

Support for time zones

All functions for working with the date and time that have a logical use for the time zone can accept a second optional time zone argument. Example: Asia/Yekaterinburg. In this case, they use the specified time zone instead of the local (default) one.

```
SELECT
    toDateTime('2016-06-15 23:00:00') AS time,
    toDate(time) AS date_local,
    toDate(time, 'Asia/Yekaterinburg') AS date_yekat,
    toString(time, 'US/Samoa') AS time_samoa
```

time	date_local	date_yekat	time_samoa
2016-06-15 23:00:00	2016-06-15	2016-06-16	2016-06-15 09:00:00

Only time zones that differ from UTC by a whole number of hours are supported.

toTimeZone

Convert time or date and time to the specified time zone.

toYear

Converts a date or date with time to a UInt16 number containing the year number (AD).

toQuarter

Converts a date or date with time to a UInt8 number containing the quarter number.

toMonth

Converts a date or date with time to a UInt8 number containing the month number (1-12).

toDayOfYear

Converts a date or date with time to a UInt8 number containing the number of the day of the year (1-366).

toDayOfMonth

Converts a date or date with time to a UInt8 number containing the number of the day of the month (1-31).

toDayOfWeek

Converts a date or date with time to a UInt8 number containing the number of the day of the week (Monday is 1, and Sunday is 7).

toHour

Converts a date with time to a UInt8 number containing the number of the hour in 24-hour time (0-23).

This function assumes that if clocks are moved ahead, it is by one hour and occurs at 2 a.m., and if clocks are moved back, it is by one hour and occurs at 3 a.m. (which is not always true – even in Moscow the clocks were twice changed at a different time).

toMinute

Converts a date with time to a UInt8 number containing the number of the minute of the hour (0-59).

toSecond

Converts a date with time to a UInt8 number containing the number of the second in the minute (0-59).

Leap seconds are not accounted for.

toUnixTimestamp

Converts a date with time to a unix timestamp.

toStartOfYear

Rounds down a date or date with time to the first day of the year.

Returns the date.

toStartOfISOYear

Rounds down a date or date with time to the first day of ISO year.

Returns the date.

toStartOfQuarter

Rounds down a date or date with time to the first day of the quarter.

The first day of the quarter is either 1 January, 1 April, 1 July, or 1 October.

Returns the date.

toStartOfMonth

Rounds down a date or date with time to the first day of the month.

Returns the date.

Attention

The behavior of parsing incorrect dates is implementation specific. ClickHouse may return zero date, throw an exception or do "natural" overflow.

toMonday

Rounds down a date or date with time to the nearest Monday.
Returns the date.

toStartOfDay

Rounds down a date with time to the start of the day.

toStartOfHour

Rounds down a date with time to the start of the hour.

toStartOfMinute

Rounds down a date with time to the start of the minute.

toStartOfFiveMinute

Rounds down a date with time to the start of the five-minute interval.

toStartOfTenMinutes

Rounds down a date with time to the start of the ten-minute interval.

toStartOfFifteenMinutes

Rounds down the date with time to the start of the fifteen-minute interval.

toStartOfInterval(time_or_data, INTERVAL x unit [, time_zone])

This is a generalization of other functions named `toStartOf*`. For example,
`toStartOfInterval(t, INTERVAL 1 year)` returns the same as `toStartOfYear(t)`,
`toStartOfInterval(t, INTERVAL 1 month)` returns the same as `toStartOfMonth(t)`,
`toStartOfInterval(t, INTERVAL 1 day)` returns the same as `toStartOfDay(t)`,
`toStartOfInterval(t, INTERVAL 15 minute)` returns the same as `toStartOfFifteenMinutes(t)` etc.

toTime

Converts a date with time to a certain fixed date, while preserving the time.

toRelativeYearNum

Converts a date with time or date to the number of the year, starting from a certain fixed point in the past.

toRelativeQuarterNum

Converts a date with time or date to the number of the quarter, starting from a certain fixed point in the past.

toRelativeMonthNum

Converts a date with time or date to the number of the month, starting from a certain fixed point in the past.

toRelativeWeekNum

Converts a date with time or date to the number of the week, starting from a certain fixed point in the past.

toRelativeDayNum

Converts a date with time or date to the number of the day, starting from a certain fixed point in the past.

toRelativeHourNum

Converts a date with time or date to the number of the hour, starting from a certain fixed point in the past.

toRelativeMinuteNum

Converts a date with time or date to the number of the minute, starting from a certain fixed point in the past.

toRelativeSecondNum

Converts a date with time or date to the number of the second, starting from a certain fixed point in the past.

toISOYear

Converts a date or date with time to a UInt16 number containing the ISO Year number.

toISOWeek

Converts a date or date with time to a UInt8 number containing the ISO Week number.

now

Accepts zero arguments and returns the current time at one of the moments of request execution. This function returns a constant, even if the request took a long time to complete.

today

Accepts zero arguments and returns the current date at one of the moments of request execution. The same as 'toDate(now())'.

yesterday

Accepts zero arguments and returns yesterday's date at one of the moments of request execution. The same as 'today() - 1'.

timeSlot

Rounds the time to the half hour.

This function is specific to Yandex.Metrica, since half an hour is the minimum amount of time for breaking a session into two sessions if a tracking tag shows a single user's consecutive pageviews that differ in time by strictly more than this amount. This means that tuples (the tag ID, user ID, and time slot) can be used to search for pageviews that are included in the corresponding session.

toYYYYMM

Converts a date or date with time to a UInt32 number containing the year and month number ($YYYY * 100 + MM$).

toYYYYMMDD

Converts a date or date with time to a UInt32 number containing the year and month number ($YYYY * 10000 + MM * 100 + DD$).

toYYYYMMDDhhmmss

Converts a date or date with time to a UInt64 number containing the year and month number ($YYYY * 10000000000 + MM * 100000000 + DD * 1000000 + hh * 10000 + mm * 100 + ss$).

addYears, addMonths, addWeeks, addDays, addHours, addMinutes, addSeconds, addQuarters

Function adds a Date/DateTime interval to a Date/DateTime and then return the Date/DateTime. For example:

```
WITH
  toDate('2018-01-01') AS date,
  toDateTime('2018-01-01 00:00:00') AS date_time
SELECT
  addYears(date, 1) AS add_years_with_date,
  addYears(date_time, 1) AS add_years_with_date_time
```

add_years_with_date	add_years_with_date_time
2019-01-01	2019-01-01 00:00:00

subtractYears, subtractMonths, subtractWeeks, subtractDays, subtractHours, subtractMinutes, subtractSeconds, subtractQuarters

Function subtract a Date/DateTime interval to a Date/DateTime and then return the Date/DateTime. For example:

```
WITH
  toDate('2019-01-01') AS date,
  toDateTime('2019-01-01 00:00:00') AS date_time
SELECT
  subtractYears(date, 1) AS subtract_years_with_date,
  subtractYears(date_time, 1) AS subtract_years_with_date_time
```

subtract_years_with_date	subtract_years_with_date_time
2018-01-01	2018-01-01 00:00:00

dateDiff('unit', t1, t2, [timezone])

Return the difference between two times expressed in 'unit' e.g. '[hours](#)'. 't1' and 't2' can be Date or DateTime, If 'timezone' is specified, it applied to both arguments. If not, timezones from datatypes 't1' and 't2' are used. If that timezones are not the same, the result is unspecified.

Supported unit values:

unit

second
minute
hour
day
week
month
quarter
year

timeSlots(StartTime, Duration[, Size])

For a time interval starting at 'StartTime' and continuing for 'Duration' seconds, it returns an array of moments in time, consisting of points from this interval rounded down to the 'Size' in seconds. 'Size' is an optional parameter: a constant UInt32, set to 1800 by default.

For example, `timeSlots(toDateTime('2012-01-01 12:20:00'), 600) = [toDateTime('2012-01-01 12:00:00'), toDateTime('2012-01-01 12:30:00')]`.

This is necessary for searching for pageviews in the corresponding session.

formatDateTime(Time, Format[, Timezone])

Function formats a Time according given Format string. N.B.: Format is a constant expression, e.g. you can not have multiple formats for single result column.

Supported modifiers for Format:

("Example" column shows formatting result for time 2018-01-02 22:33:44)

Modifier	Description	Example
%C	year divided by 100 and truncated to integer (00-99)	20
%d	day of the month, zero-padded (01-31)	02
%D	Short MM/DD/YY date, equivalent to %m/%d/%y	01/02/2018
%e	day of the month, space-padded (1-31)	2
%F	short YYYY-MM-DD date, equivalent to %Y-%m-%d	2018-01-02
%H	hour in 24h format (00-23)	22
%I	hour in 12h format (01-12)	10
%j	day of the year (001-366)	002
%m	month as a decimal number (01-12)	01
%M	minute (00-59)	33
%n	new-line character ('\n')	
%p	AM or PM designation	PM
%R	24-hour HH:MM time, equivalent to %H:%M	22:33
%S	second (00-59)	44
%t	horizontal-tab character ('\t')	
%T	ISO 8601 time format (HH:MM:SS), equivalent to %H:%M:%S	22:33:44
%u	ISO 8601 weekday as number with Monday as 1 (1-7)	2
%V	ISO 8601 week number (01-53)	01
%w	weekday as a decimal number with Sunday as 0 (0-6)	2
%y	Year, last two digits (00-99)	18
%Y	Year	2018
%%	a % sign	%

Functions for working with strings

empty

Returns 1 for an empty string or 0 for a non-empty string.

The result type is UInt8.

A string is considered non-empty if it contains at least one byte, even if this is a space or a null byte.

The function also works for arrays.

notEmpty

Returns 0 for an empty string or 1 for a non-empty string.

The result type is UInt8.

The function also works for arrays.

length

Returns the length of a string in bytes (not in characters, and not in code points).

The result type is UInt64.

The function also works for arrays.

lengthUTF8

Returns the length of a string in Unicode code points (not in characters), assuming that the string contains a set of bytes that make up UTF-8 encoded text. If this assumption is not met, it returns some result (it doesn't throw an exception).

The result type is UInt64.

char_length, CHAR_LENGTH

Returns the length of a string in Unicode code points (not in characters), assuming that the string contains a set of bytes that make up UTF-8 encoded text. If this assumption is not met, it returns some result (it doesn't throw an exception).

The result type is UInt64.

character_length, CHARACTER_LENGTH

Returns the length of a string in Unicode code points (not in characters), assuming that the string contains a set of bytes that make up UTF-8 encoded text. If this assumption is not met, it returns some result (it doesn't throw an exception).

The result type is UInt64.

lower, lcase

Converts ASCII Latin symbols in a string to lowercase.

upper, ucase

Converts ASCII Latin symbols in a string to uppercase.

lowerUTF8

Converts a string to lowercase, assuming the string contains a set of bytes that make up a UTF-8 encoded text. It doesn't detect the language. So for Turkish the result might not be exactly correct.

If the length of the UTF-8 byte sequence is different for upper and lower case of a code point, the result may be incorrect for this code point.

If the string contains a set of bytes that is not UTF-8, then the behavior is undefined.

upperUTF8

Converts a string to uppercase, assuming the string contains a set of bytes that make up a UTF-8 encoded text. It doesn't detect the language. So for Turkish the result might not be exactly correct.


If the length of the UTF-8 byte sequence is different for upper and lower case of a code point, the result may be incorrect for this code point.

If the string contains a set of bytes that is not UTF-8, then the behavior is undefined.

isValidUTF8

Returns 1, if the set of bytes is valid UTF-8 encoded, otherwise 0.

toValidUTF8

Replaces invalid UTF-8 characters by the  (U+FFFD) character. All running in a row invalid characters are collapsed into the one replacement character.

```
toValidUTF8( input_string )
```




Parameters:

- input_string — Any set of bytes represented as the **String** data type object.

Returned value: Valid UTF-8 string.

Example

```
SELECT toValidUTF8('\x61\xF0\x80\x80\x80b')
```

```
┌toValidUTF8('ab')┐  
└ab┘
```

reverse

Reverses the string (as a sequence of bytes).

reverseUTF8

Reverses a sequence of Unicode code points, assuming that the string contains a set of bytes representing a UTF-8 text. Otherwise, it does something else (it doesn't throw an exception).

format(pattern, s0, s1, ...)

Formatting constant pattern with the string listed in the arguments. **pattern** is a simplified Python format pattern. Format string contains "replacement fields" surrounded by curly braces `{}`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: `{{` and `}}`. Field names can be numbers (starting from zero) or empty (then they are treated as consequence numbers).

```
SELECT format('{1} {0} {1}', 'World', 'Hello')
```

```
└─format('{1} {0} {1}', 'World', 'Hello')─┐
| Hello World Hello                        |
└────────────────────────────────────────┘
```

```
SELECT format('{} {}'. 'Hello', 'World')
```

```
└─format('{} {}'. 'Hello', 'World')─┐
| Hello World                        |
└──────────────────────────────────┘
```

concat(s1, s2, ...)

Concatenates the strings listed in the arguments, without a separator.

concatAssumeInjective(s1, s2, ...)

Same as **concat**, the difference is that you need to ensure that `concat(s1, s2, s3) -> s4` is injective, it will be used for optimization of GROUP BY

substring(s, offset, length), mid(s, offset, length), substr(s, offset, length)

Returns a substring starting with the byte from the 'offset' index that is 'length' bytes long. Character indexing starts from one (as in standard SQL). The 'offset' and 'length' arguments must be constants.

substringUTF8(s, offset, length)

The same as 'substring', but for Unicode code points. Works under the assumption that the string contains a set of bytes representing a UTF-8 encoded text. If this assumption is not met, it returns some result (it doesn't throw an exception).

appendTrailingCharIfAbsent(s, c)

If the 's' string is non-empty and does not contain the 'c' character at the end, it appends the 'c' character to the end.

convertCharset(s, from, to)

Returns the string 's' that was converted from the encoding in 'from' to the encoding in 'to'.

base64Encode(s)

Encodes 's' string into base64

base64Decode(s)

Decode base64-encoded string 's' into original string. In case of failure raises an exception.

tryBase64Decode(s)

Similar to `base64Decode`, but in case of error an empty string would be returned.

endsWith(s, suffix)

Returns whether to end with the specified suffix. Returns 1 if the string ends with the specified suffix, otherwise it returns 0.

startsWith(s, prefix)

Returns whether to start with the specified prefix. Returns 1 if the string starts with the specified prefix, otherwise it returns 0.

trimLeft(s)

Returns a string that removes the whitespace characters on left side.

trimRight(s)

Returns a string that removes the whitespace characters on right side.

trimBoth(s)

Returns a string that removes the whitespace characters on either side.

Functions for Searching Strings

The search is case-sensitive by default in all these functions. There are separate variants for case insensitive search.

position(haystack, needle), locate(haystack, needle)

Search for the substring `needle` in the string `haystack`.

Returns the position (in bytes) of the found substring, starting from 1, or returns 0 if the substring was not found.

For a case-insensitive search, use the function `positionCaseInsensitive`.

positionUTF8(haystack, needle)

The same as `position`, but the position is returned in Unicode code points. Works under the assumption that the string contains a set of bytes representing a UTF-8 encoded text. If this assumption is not met, it returns some result (it doesn't throw an exception).

For a case-insensitive search, use the function `positionCaseInsensitiveUTF8`.

multiSearchAllPositions(haystack, [needle₁, needle₂, ..., needle_n])

The same as `position`, but returns `Array` of the `positions` for all `needlei`.

For a case-insensitive search or/and in UTF-8 format use functions `multiSearchAllPositionsCaseInsensitive`, `multiSearchAllPositionsUTF8`, `multiSearchAllPositionsCaseInsensitiveUTF8`.

multiSearchFirstPosition(haystack, [needle₁, needle₂, ..., needle_n])

The same as `position` but returns the leftmost offset of the string `haystack` that is matched to some of the needles.

For a case-insensitive search or/and in UTF-8 format use functions `multiSearchFirstPositionCaseInsensitive`, `multiSearchFirstPositionUTF8`, `multiSearchFirstPositionCaseInsensitiveUTF8`.

multiSearchFirstIndex(haystack, [needle₁, needle₂, ..., needle_n])

Returns the index `i` (starting from 1) of the leftmost found `needlei` in the string `haystack` and 0 otherwise.

For a case-insensitive search or/and in UTF-8 format use functions [multiSearchFirstIndexCaseInsensitive](#), [multiSearchFirstIndexUTF8](#), [multiSearchFirstIndexCaseInsensitiveUTF8](#).

multiSearchAny(haystack, [needle₁, needle₂, ..., needle_n])

Returns 1, if at least one string needle_i matches the string [haystack](#) and 0 otherwise.

For a case-insensitive search or/and in UTF-8 format use functions [multiSearchAnyCaseInsensitive](#), [multiSearchAnyUTF8](#), [multiSearchAnyCaseInsensitiveUTF8](#).

Note: in all [multiSearch*](#) functions the number of needles should be less than 2⁸ because of implementation specification.

match(haystack, pattern)

Checks whether the string matches the [pattern](#) regular expression. A [re2](#) regular expression. The [syntax](#) of the [re2](#) regular expressions is more limited than the syntax of the Perl regular expressions.

Returns 0 if it doesn't match, or 1 if it matches.

Note that the backslash symbol (`\`) is used for escaping in the regular expression. The same symbol is used for escaping in string literals. So in order to escape the symbol in a regular expression, you must write two backslashes (`\\`) in a string literal.

The regular expression works with the string as if it is a set of bytes. The regular expression can't contain null bytes.

For patterns to search for substrings in a string, it is better to use `LIKE` or 'position', since they work much faster.

multiMatchAny(haystack, [pattern₁, pattern₂, ..., pattern_n])

The same as [match](#), but returns 0 if none of the regular expressions are matched and 1 if any of the patterns matches. It uses [hyperscan](#) library. For patterns to search substrings in a string, it is better to use [multiSearchAny](#) since it works much faster.

Note: the length of any of the [haystack](#) string must be less than 2³² bytes otherwise the exception is thrown. This restriction takes place because of hyperscan API.

multiMatchAnyIndex(haystack, [pattern₁, pattern₂, ..., pattern_n])

The same as [multiMatchAny](#), but returns any index that matches the haystack.

multiFuzzyMatchAny(haystack, distance, [pattern₁, pattern₂, ..., pattern_n])

The same as [multiMatchAny](#), but returns 1 if any pattern matches the haystack within a constant [edit distance](#). This function is also in an experimental mode and can be extremely slow. For more information see [hyperscan documentation](#).

multiFuzzyMatchAnyIndex(haystack, distance, [pattern₁, pattern₂, ..., pattern_n])

The same as [multiFuzzyMatchAny](#), but returns any index that matches the haystack within a constant edit distance.

Note: [multiFuzzyMatch*](#) functions do not support UTF-8 regular expressions, and such expressions are treated as bytes because of hyperscan restriction.

Note: to turn off all functions that use hyperscan, use setting [SET allow_hyperscan = 0](#);

extract(haystack, pattern)

Extracts a fragment of a string using a regular expression. If 'haystack' doesn't match the 'pattern' regex, an empty string is returned. If the regex doesn't contain subpatterns, it takes the fragment that matches the entire regex. Otherwise, it takes the fragment that matches the first subpattern.

extractAll(haystack, pattern)

Extracts all the fragments of a string using a regular expression. If 'haystack' doesn't match the 'pattern' regex, an empty string is returned. Returns an array of strings consisting of all matches to the regex. In general, the behavior is the same as the 'extract' function (it takes the first subpattern, or the entire expression if there isn't a subpattern).

like(haystack, pattern), haystack LIKE pattern operator

Checks whether a string matches a simple regular expression.

The regular expression can contain the metasymbols % and _.

% indicates any quantity of any bytes (including zero characters).

_ indicates any one byte.

Use the backslash (\) for escaping metasymbols. See the note on escaping in the description of the 'match' function.

For regular expressions like %needle%, the code is more optimal and works as fast as the position function.

For other regular expressions, the code is the same as for the 'match' function.

notLike(haystack, pattern), haystack NOT LIKE pattern operator

The same thing as 'like', but negative.

ngramDistance(haystack, needle)

Calculates the 4-gram distance between haystack and needle: counts the symmetric difference between two multisets of 4-grams and normalizes it by the sum of their cardinalities. Returns float number from 0 to 1 -- the closer to zero, the more strings are similar to each other. If the constant needle or haystack is more than 32Kb, throws an exception. If some of the non-constant haystack or needle strings are more than 32Kb, the distance is always one.

For case-insensitive search or/and in UTF-8 format use functions ngramDistanceCaseInsensitive, ngramDistanceUTF8, ngramDistanceCaseInsensitiveUTF8.

ngramSearch(haystack, needle)

Same as ngramDistance but calculates the non-symmetric difference between needle and haystack -- the number of n-grams from needle minus the common number of n-grams normalized by the number of needle n-grams. Can be useful for fuzzy string search.

For case-insensitive search or/and in UTF-8 format use functions ngramSearchCaseInsensitive, ngramSearchUTF8, ngramSearchCaseInsensitiveUTF8.

Note: For UTF-8 case we use 3-gram distance. All these are not perfectly fair n-gram distances. We use 2-byte hashes to hash n-grams and then calculate the (non-)symmetric difference between these hash tables -- collisions may occur. With UTF-8 case-insensitive format we do not use fair tolower function -- we zero the 5-th bit (starting from zero) of each codepoint byte and first bit of zeroth byte if bytes more than one -- this works for Latin and mostly for all Cyrillic letters.

Functions for searching and replacing in strings

replaceOne(haystack, pattern, replacement)

Replaces the first occurrence, if it exists, of the 'pattern' substring in 'haystack' with the 'replacement' substring. Hereafter, 'pattern' and 'replacement' must be constants.

replaceAll(haystack, pattern, replacement), replace(haystack, pattern, replacement)

Replaces all occurrences of the 'pattern' substring in 'haystack' with the 'replacement' substring.

replaceRegexpOne(haystack, pattern, replacement)

Replacement using the 'pattern' regular expression. A re2 regular expression.

Replaces only the first occurrence, if it exists.

A pattern can be specified as 'replacement'. This pattern can include substitutions `\0-19`.

The substitution `\0` includes the entire regular expression. Substitutions `\1-19` correspond to the subpattern numbers. To use the `\` character in a template, escape it using `\\`.

Also keep in mind that a string literal requires an extra escape.

Example 1. Converting the date to American format:

```
SELECT DISTINCT
  EventDate,
  replaceRegexpOne(toString(EventDate), '\\d{4}-\\d{2}-\\d{2}', '\\2/\\3/\\1') AS res
FROM test.hits
LIMIT 7
FORMAT TabSeparated
```

2014-03-17	03/17/2014
2014-03-18	03/18/2014
2014-03-19	03/19/2014
2014-03-20	03/20/2014
2014-03-21	03/21/2014
2014-03-22	03/22/2014
2014-03-23	03/23/2014

Example 2. Copying a string ten times:

```
SELECT replaceRegexpOne('Hello, World!', '.*', '\\0\\0\\0\\0\\0\\0\\0\\0\\0\\0') AS res
```

```
┌res┐
│ Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello, World! |
└───┘
```

replaceRegexpAll(haystack, pattern, replacement)

This does the same thing, but replaces all the occurrences. Example:

```
SELECT replaceRegexpAll('Hello, World!', '.', '\\0\\0') AS res
```

```
┌res┐
│ HHeelllloo,, WWoorrrlidd!! |
└───┘
```

As an exception, if a regular expression worked on an empty substring, the replacement is not made more than once.

Example:

```
SELECT replaceRegexpAll('Hello, World!', '^', 'here: ') AS res
```

```
┌res┐
│ here: Hello, World! |
└───┘
```

regexpQuoteMeta(s)

The function adds a backslash before some predefined characters in the string.

Predefined characters: '0', '\', '|', '(', ')', '^', '\$', ':', '[', ']', '?', '*', '+', '{', '}', '-', ' '.

This implementation slightly differs from `re2::RE2::QuoteMeta`. It escapes zero byte as `\0` instead of `\x00` and it escapes only required characters.

For more information, see the link: [RE2](#)

Conditional functions

`if(cond, then, else), cond ? operator then : else`

Returns `then` if `cond != 0`, or `else` if `cond = 0`.

`cond` must be of type `UInt8`, and `then` and `else` must have the lowest common type.

`then` and `else` can be `NULL`

`multif`

Allows you to write the `CASE` operator more compactly in the query.

```
multif(cond_1, then_1, cond_2, then_2...else)
```

Parameters:

- `cond_N` — The condition for the function to return `then_N`.
- `then_N` — The result of the function when executed.
- `else` — The result of the function if none of the conditions is met.

The function accepts `2N+1` parameters.

Returned values

The function returns one of the values `then_N` or `else`, depending on the conditions `cond_N`.

Example

Take the table

x	y
1	NULL
2	3

Run the query `SELECT multif(isNull(y) x, y < 3, y, NULL) FROM t_null`. Result:

multif(isNull(y), x, less(y, 3), y, NULL)
1
NULL

Mathematical functions

All the functions return a `Float64` number. The accuracy of the result is close to the maximum precision possible, but the result might not coincide with the machine representable number nearest to the corresponding real number.

`e()`

Returns a `Float64` number that is close to the number `e`.

`pi()`

Returns a `Float64` number that is close to the number π .

`exp(x)`

Accepts a numeric argument and returns a Float64 number close to the exponent of the argument.

log(x), ln(x)

Accepts a numeric argument and returns a Float64 number close to the natural logarithm of the argument.

exp2(x)

Accepts a numeric argument and returns a Float64 number close to 2 to the power of x.

log2(x)

Accepts a numeric argument and returns a Float64 number close to the binary logarithm of the argument.

exp10(x)

Accepts a numeric argument and returns a Float64 number close to 10 to the power of x.

log10(x)

Accepts a numeric argument and returns a Float64 number close to the decimal logarithm of the argument.

sqrt(x)

Accepts a numeric argument and returns a Float64 number close to the square root of the argument.

cbrt(x)

Accepts a numeric argument and returns a Float64 number close to the cubic root of the argument.

erf(x)

If 'x' is non-negative, then $\text{erf}(x / \sigma\sqrt{2})$ is the probability that a random variable having a normal distribution with standard deviation ' σ ' takes the value that is separated from the expected value by more than 'x'.

Example (three sigma rule):

```
SELECT erf(3 / sqrt(2))
```

```
┌erf(divide(3, sqrt(2)))┐  
└ 0.9973002039367398 ─┘
```

erfc(x)

Accepts a numeric argument and returns a Float64 number close to $1 - \text{erf}(x)$, but without loss of precision for large 'x' values.

lgamma(x)

The logarithm of the gamma function.

tgamma(x)

Gamma function.

sin(x)

The sine.

cos(x)

The cosine.

tan(x)

The tangent.

asin(x)

The arc sine.

acos(x)

The arc cosine.

atan(x)

The arc tangent.

pow(x, y), power(x, y)

Takes two numeric arguments x and y. Returns a Float64 number close to x to the power of y.

intExp2

Accepts a numeric argument and returns a UInt64 number close to 2 to the power of x.

intExp10

Accepts a numeric argument and returns a UInt64 number close to 10 to the power of x.

Rounding functions

floor(x[, N])

Returns the largest round number that is less than or equal to x. A round number is a multiple of $1/10^N$, or the nearest number of the appropriate data type if $1 / 10^N$ isn't exact.

'N' is an integer constant, optional parameter. By default it is zero, which means to round to an integer.

'N' may be negative.

Examples: `floor(123.45, 1) = 123.4`, `floor(123.45, -1) = 120`.

x is any numeric type. The result is a number of the same type.

For integer arguments, it makes sense to round with a negative 'N' value (for non-negative 'N', the function doesn't do anything).

If rounding causes overflow (for example, `floor(-128, -1)`), an implementation-specific result is returned.

ceil(x[, N]), ceiling(x[, N])

Returns the smallest round number that is greater than or equal to 'x'. In every other way, it is the same as the 'floor' function (see above).

round(x[, N])

Rounds a value to a specified number of decimal places.

The function returns the nearest number of the specified order. In case when given number has equal distance to surrounding numbers the function returns the number having the nearest even digit (banker's rounding).

round(expression [, decimal_places])

Parameters:

- expression — A number to be rounded. Can be any expression returning the numeric data type.

- **decimal-places** — An integer value.
 - If **decimal-places** > 0 then the function rounds the value to the right of the decimal point.
 - If **decimal-places** < 0 then the function rounds the value to the left of the decimal point.
 - If **decimal-places** = 0 then the function rounds the value to integer. In this case the argument can be omitted.

Returned value:

The rounded number of the same type as the input number.

Examples

Example of use

```
SELECT number / 2 AS x, round(x) FROM system.numbers LIMIT 3
```

x	round(divide(number, 2))
0	0
0.5	0
1	1

Examples of rounding

Rounding to the nearest number.

```
round(3.2, 0) = 3
round(4.1267, 2) = 4.13
round(22,-1) = 20
round(467,-2) = 500
round(-467,-2) = -500
```

Banker's rounding.

```
round(3.5) = 4
round(4.5) = 4
round(3.55, 1) = 3.6
round(3.65, 1) = 3.6
```

roundToExp2(num)

Accepts a number. If the number is less than one, it returns 0. Otherwise, it rounds the number down to the nearest (whole non-negative) degree of two.

roundDuration(num)

Accepts a number. If the number is less than one, it returns 0. Otherwise, it rounds the number down to numbers from the set: 1, 10, 30, 60, 120, 180, 240, 300, 600, 1200, 1800, 3600, 7200, 18000, 36000. This function is specific to Yandex.Metrica and used for implementing the report on session length

roundAge(num)

Accepts a number. If the number is less than 18, it returns 0. Otherwise, it rounds the number down to a number from the set: 18, 25, 35, 45, 55. This function is specific to Yandex.Metrica and used for implementing the report on user age.

roundDown(num, arr)

Accept a number, round it down to an element in the specified array. If the value is less than the lowest bound, the lowest bound is returned.

Functions for working with arrays

empty

Returns 1 for an empty array, or 0 for a non-empty array.
The result type is UInt8.
The function also works for strings.

notEmpty

Returns 0 for an empty array, or 1 for a non-empty array.
The result type is UInt8.
The function also works for strings.

length

Returns the number of items in the array.
The result type is UInt64.
The function also works for strings.

emptyArrayUInt8, emptyArrayUInt16, emptyArrayUInt32,
emptyArrayUInt64

emptyArrayInt8, emptyArrayInt16, emptyArrayInt32,
emptyArrayInt64

emptyArrayFloat32, emptyArrayFloat64

emptyArrayDate, emptyArrayDateTime

emptyArrayString

Accepts zero arguments and returns an empty array of the appropriate type.

emptyArrayToSingle

Accepts an empty array and returns a one-element array that is equal to the default value.

range(N)

Returns an array of numbers from 0 to N-1.
Just in case, an exception is thrown if arrays with a total length of more than 100,000,000 elements are created in a data block.

array(x1, ...), operator [x1, ...]

Creates an array from the function arguments.
The arguments must be constants and have types that have the smallest common type. At least one argument must be passed, because otherwise it isn't clear which type of array to create. That is, you can't use this function to create an empty array (to do that, use the 'emptyArray*' function described above).
Returns an 'Array(T)' type result, where 'T' is the smallest common type out of the passed arguments.

arrayConcat

Combines arrays passed as arguments.

```
arrayConcat(arrays)
```

Parameters

- **arrays** – Arbitrary number of arguments of **Array** type.

Example

```
SELECT arrayConcat([1, 2], [3, 4], [5, 6]) AS res
```

```
┌res┐
└─┬─┘
  │ [1,2,3,4,5,6] │
  └────────────────┘
```

arrayElement(arr, n), operator arr[n]

Get the element with the index **n** from the array **arr**. **n** must be any integer type.

Indexes in an array begin from one.

Negative indexes are supported. In this case, it selects the corresponding element numbered from the end. For example, **arr[-1]** is the last item in the array.

If the index falls outside of the bounds of an array, it returns some default value (0 for numbers, an empty string for strings, etc.).

has(arr, elem)

Checks whether the 'arr' array has the 'elem' element.

Returns 0 if the the element is not in the array, or 1 if it is.

NULL is processed as a value.

```
SELECT has([1, 2, NULL], NULL)
```

```
┌has([1, 2, NULL], NULL)┐
└──────────────────┴──┘
1
```

hasAll

Checks whether one array is a subset of another.

```
hasAll(set, subset)
```

Parameters

- **set** – Array of any type with a set of elements.
- **subset** – Array of any type with elements that should be tested to be a subset of **set**.

Return values

- **1**, if **set** contains all of the elements from **subset**.
- **0**, otherwise.

Peculiar properties

- An empty array is a subset of any array.
- **Null** processed as a value.
- Order of values in both of arrays doesn't matter.

Examples

SELECT hasAll([], []) returns 1.

SELECT hasAll([1, Null], [Null]) returns 1.

SELECT hasAll([1.0, 2, 3, 4], [1, 3]) returns 1.

SELECT hasAll(['a', 'b'], ['a']) returns 1.

SELECT hasAll([1], ['a']) returns 0.

SELECT hasAll([[1, 2], [3, 4]], [[1, 2], [3, 5]]) returns 0.

hasAny

Checks whether two arrays have intersection by some elements.

```
hasAny(array1, array2)
```

Parameters

- **array1** – Array of any type with a set of elements.
- **array2** – Array of any type with a set of elements.

Return values

- **1**, if **array1** and **array2** have one similar element at least.
- **0**, otherwise.

Peculiar properties

- **Null** processed as a value.
- Order of values in both of arrays doesn't matter.

Examples

SELECT hasAny([1], []) returns **0**.

SELECT hasAny([Null], [Null, 1]) returns **1**.

SELECT hasAny([-128, 1., 512], [1]) returns **1**.

SELECT hasAny([[1, 2], [3, 4]], ['a', 'c']) returns **0**.

SELECT hasAll([[1, 2], [3, 4]], [[1, 2], [1, 2]]) returns **1**.

indexOf(arr, x)

Returns the index of the first 'x' element (starting from 1) if it is in the array, or 0 if it is not.

Example:

```
:) SELECT indexOf([1,3,NULL,NULL],NULL)
```

```
SELECT indexOf([1, 3, NULL, NULL], NULL)
```

```
┌indexOf([1, 3, NULL, NULL], NULL)┐
```

```
3 |
```

Elements set to **NULL** are handled as normal values.

countEqual(arr, x)

Returns the number of elements in the array equal to x. Equivalent to arrayCount (elem -> elem = x, arr).

NULL elements are handled as separate values.

Example:

```
SELECT countEqual([1, 2, NULL, NULL], NULL)
```

```
┌countEqual([1, 2, NULL, NULL], NULL)┐
```

```
2 |
```

arrayEnumerate(arr)

Returns the array [1, 2, 3, ..., length (arr)]

This function is normally used with ARRAY JOIN. It allows counting something just once for each array after applying ARRAY JOIN. Example:

```
SELECT
    count() AS Reaches,
    countIf(num = 1) AS Hits
FROM test.hits
ARRAY JOIN
    GoalsReached,
    arrayEnumerate(GoalsReached) AS num
WHERE CounterID = 160656
LIMIT 10
```

Reaches	Hits
95606	31406

In this example, Reaches is the number of conversions (the strings received after applying ARRAY JOIN), and Hits is the number of pageviews (strings before ARRAY JOIN). In this particular case, you can get the same result in an easier way:

```
SELECT
    sum(length(GoalsReached)) AS Reaches,
    count() AS Hits
FROM test.hits
WHERE (CounterID = 160656) AND notEmpty(GoalsReached)
```

Reaches	Hits
95606	31406

This function can also be used in higher-order functions. For example, you can use it to get array indexes for elements that match a condition.

arrayEnumerateUniq(arr, ...)

Returns an array the same size as the source array, indicating for each element what its position is among elements with the same value.

For example: `arrayEnumerateUniq([10, 20, 10, 30]) = [1, 1, 2, 1]`.

This function is useful when using ARRAY JOIN and aggregation of array elements.

Example:

```
SELECT
    Goals.ID AS GoalID,
    sum(Sign) AS Reaches,
    sumIf(Sign, num = 1) AS Visits
FROM test.visits
ARRAY JOIN
    Goals,
    arrayEnumerateUniq(Goals.ID) AS num
WHERE CounterID = 160656
GROUP BY GoalID
ORDER BY Reaches DESC
LIMIT 10
```

GoalID	Reaches	Visits
53225	3214	1097
2825062	3188	1097
56600	2803	488
1989037	2401	365
2830064	2396	910
1113562	2372	373
3270895	2262	812
1084657	2262	345
56599	2260	799
3271094	2256	812

In this example, each goal ID has a calculation of the number of conversions (each element in the Goals nested data structure is a goal that was reached, which we refer to as a conversion) and the number of sessions. Without ARRAY JOIN, we would have counted the number of sessions as sum(Sign). But in this particular case, the rows were multiplied by the nested Goals structure, so in order to count each session one time after this, we apply a condition to the value of the arrayEnumerateUniq(Goals.ID) function.

The arrayEnumerateUniq function can take multiple arrays of the same size as arguments. In this case, uniqueness is considered for tuples of elements in the same positions in all the arrays.

```
SELECT arrayEnumerateUniq([1, 1, 1, 2, 2, 2], [1, 1, 2, 1, 1, 2]) AS res
```

res
[1,2,1,1,2,1]

This is necessary when using ARRAY JOIN with a nested data structure and further aggregation across multiple elements in this structure.

arrayPopBack

Removes the last item from the array.

```
arrayPopBack(array)
```

Parameters

- **array** – Array.

Example

```
SELECT arrayPopBack([1, 2, 3]) AS res
```

res
[1,2]

arrayPopFront

Removes the first item from the array.

```
arrayPopFront(array)
```

Parameters

- **array** – Array.

Example

```
SELECT arrayPopFront([1, 2, 3]) AS res
```



```
┌res┐
│ [2,3] │
```

arrayPushBack

Adds one item to the end of the array.

```
arrayPushBack(array, single_value)
```

Parameters

- **array** – Array.
- **single_value** – A single value. Only numbers can be added to an array with numbers, and only strings can be added to an array of strings. When adding numbers, ClickHouse automatically sets the **single_value** type for the data type of the array. For more information about the types of data in ClickHouse, see "[Data types](#)". Can be **NULL**. The function adds a **NULL** element to an array, and the type of array elements converts to **Nullable**.

Example

```
SELECT arrayPushBack(['a'], 'b') AS res
```

```
┌res┐
│ ['a','b'] │
```

arrayPushFront

Adds one element to the beginning of the array.

```
arrayPushFront(array, single_value)
```

Parameters

- **array** – Array.
- **single_value** – A single value. Only numbers can be added to an array with numbers, and only strings can be added to an array of strings. When adding numbers, ClickHouse automatically sets the **single_value** type for the data type of the array. For more information about the types of data in ClickHouse, see "[Data types](#)". Can be **NULL**. The function adds a **NULL** element to an array, and the type of array elements converts to **Nullable**.

Example

```
SELECT arrayPushBack(['b'], 'a') AS res
```

```
┌res┐
│ ['a','b'] │
```

arrayResize

Changes the length of the array.

```
arrayResize(array, size[, extender])
```

Parameters:

- **array** — Array.
- **size** — Required length of the array.
 - If **size** is less than the original size of the array, the array is truncated from the right.
- If **size** is larger than the initial size of the array, the array is extended to the right with **extender** values or default values for the data type of the array items.
- **extender** — Value for extending an array. Can be **NULL**.

Returned value:

An array of length **size**.

Examples of calls

```
SELECT arrayResize([1], 3)
```

```
┌arrayResize([1], 3)┐
└ [1,0,0]          ┘
```

```
SELECT arrayResize([1], 3, NULL)
```

```
┌arrayResize([1], 3, NULL)┐
└ [1,NULL,NULL]          ┘
```

arraySlice

Returns a slice of the array.

```
arraySlice(array, offset[, length])
```

Parameters

- **array** - Array of data.
- **offset** - Indent from the edge of the array. A positive value indicates an offset on the left, and a negative value is an indent on the right. Numbering of the array items begins with 1.
- **length** - The length of the required slice. If you specify a negative value, the function returns an open slice [**offset**, **array_length** - **length**). If you omit the value, the function returns the slice [**offset**, **the_end_of_array**].

Example

```
SELECT arraySlice([1, 2, NULL, 4, 5], 2, 3) AS res
```

```
┌res┐
└ [2,NULL,4] ┘
```

Array elements set to **NULL** are handled as normal values.

arraySort([func,] arr, ...)

Sorts the elements of the **arr** array in ascending order. If the **func** function is specified, sorting order is determined by the result of the **func** function applied to the elements of the array. If **func** accepts multiple arguments, the **arraySort** function is passed several arrays that the arguments of **func** will correspond to. Detailed examples are shown at the end of **arraySort** description.

Example of integer values sorting:

```
SELECT arraySort([1, 3, 3, 0]);
```

```
┌arraySort([1, 3, 3, 0])┐
└ [0,1,3,3]            ┘
```

Example of string values sorting:

```
SELECT arraySort(['hello', 'world', '!']);
```

```
┌arraySort(['hello', 'world', '!'])┐
└ ['!','hello','world']            ┘
```

Consider the following sorting order for the **NULL**, **NaN** and **Inf** values:

```
SELECT arraySort([1, nan, 2, NULL, 3, nan, -4, NULL, inf, -inf]);
```

```
┌arraySort([1, nan, 2, NULL, 3, nan, -4, NULL, inf, -inf])┐  
└[-inf,-4,1,2,3,inf,nan,nan,NULL,NULL]┘
```

- **-Inf** values are first in the array.
- **NULL** values are last in the array.
- **NaN** values are right before **NULL**.
- **Inf** values are right before **NaN**.

Note that **arraySort** is a **higher-order function**. You can pass a lambda function to it as the first argument. In this case, sorting order is determined by the result of the lambda function applied to the elements of the array.

Let's consider the following example:

```
SELECT arraySort((x) -> -x, [1, 2, 3]) as res;
```

```
┌res┐  
└[3,2,1]┘
```

For each element of the source array, the lambda function returns the sorting key, that is, [1 -> -1, 2 -> -2, 3 -> -3]. Since the **arraySort** function sorts the keys in ascending order, the result is [3, 2, 1]. Thus, the **(x) -> -x** lambda function sets the **descending order** in a sorting.

The lambda function can accept multiple arguments. In this case, you need to pass the **arraySort** function several arrays of identical length that the arguments of lambda function will correspond to. The resulting array will consist of elements from the first input array; elements from the next input array(s) specify the sorting keys. For example:

```
SELECT arraySort((x, y) -> y, ['hello', 'world'], [2, 1]) as res;
```

```
┌res┐  
└['world', 'hello']┘
```

Here, the elements that are passed in the second array ([2, 1]) define a sorting key for the corresponding element from the source array (['hello', 'world']), that is, ['hello' -> 2, 'world' -> 1]. Since the lambda function doesn't use **x**, actual values of the source array don't affect the order in the result. So, 'hello' will be the second element in the result, and 'world' will be the first.

Other examples are shown below.

```
SELECT arraySort((x, y) -> y, [0, 1, 2], ['c', 'b', 'a']) as res;
```

```
┌res┐  
└[2,1,0]┘
```

```
SELECT arraySort((x, y) -> -y, [0, 1, 2], [1, 2, 3]) as res;
```

```
┌res┐  
└[2,1,0]┘
```

Note

To improve sorting efficiency, the **Schwartzian transform** is used.

arrayReverseSort([func,] arr, ...)

Sorts the elements of the **arr** array in descending order. If the **func** function is specified, **arr** is sorted according to the result of the **func** function applied to the elements of the array, and then the sorted array is reversed. If **func** accepts multiple arguments, the **arrayReverseSort** function is passed several arrays that the arguments of **func** will correspond to. Detailed examples are shown at the end of **arrayReverseSort** description.

Example of integer values sorting:

```
SELECT arrayReverseSort([1, 3, 3, 0]);
```

```
┌arrayReverseSort([1, 3, 3, 0])┐
└ [3,3,1,0]                    ┘
```

Example of string values sorting:

```
SELECT arrayReverseSort(['hello', 'world', '!']);
```

```
┌arrayReverseSort(['hello', 'world', '!'])┐
└ ['world','hello','!']                  ┘
```

Consider the following sorting order for the **NULL**, **NaN** and **Inf** values:

```
SELECT arrayReverseSort([1, nan, 2, NULL, 3, nan, -4, NULL, inf, -inf]) as res;
```

```
┌res┐
└ [inf,3,2,1,-4,-inf,nan,nan,NULL,NULL] ┘
```

- **Inf** values are first in the array.
- **NULL** values are last in the array.
- **NaN** values are right before **NULL**.
- **-Inf** values are right before **NaN**.

Note that the **arrayReverseSort** is a **higher-order function**. You can pass a lambda function to it as the first argument. Example is shown below.

```
SELECT arrayReverseSort((x) -> -x, [1, 2, 3]) as res;
```

```
┌res┐
└ [1,2,3] ┘
```

The array is sorted in the following way:

1. At first, the source array ([1, 2, 3]) is sorted according to the result of the lambda function applied to the elements of the array. The result is an array [3, 2, 1].
2. Array that is obtained on the previous step, is reversed. So, the final result is [1, 2, 3].

The lambda function can accept multiple arguments. In this case, you need to pass the **arrayReverseSort** function several arrays of identical length that the arguments of lambda function will correspond to. The resulting array will consist of elements from the first input array; elements from the next input array(s) specify the sorting keys. For example:

```
SELECT arrayReverseSort((x, y) -> y, ['hello', 'world'], [2, 1]) as res;
```

```
┌res┐
└ ['hello','world'] ┘
```

In this example, the array is sorted in the following way:

1. At first, the source array (['hello', 'world']) is sorted according to the result of the lambda function applied to the elements of the arrays. The elements that are passed in the second array ([2, 1]), define the sorting keys for corresponding elements from the source array. The result is an array ['world', 'hello'].
2. Array that was sorted on the previous step, is reversed. So, the final result is ['hello', 'world'].

Other examples are shown below.

```
SELECT arrayReverseSort((x, y) -> y, [4, 3, 5], ['a', 'b', 'c']) AS res;
```

```
┌res┐
│ [5,3,4] │
```

```
SELECT arrayReverseSort((x, y) -> -y, [4, 3, 5], [1, 2, 3]) AS res;
```

```
┌res┐
│ [4,3,5] │
```

arrayUniq(arr, ...)

If one argument is passed, it counts the number of different elements in the array.

If multiple arguments are passed, it counts the number of different tuples of elements at corresponding positions in multiple arrays.

If you want to get a list of unique items in an array, you can use `arrayReduce('groupUniqArray', arr)`.

arrayJoin(arr)

A special function. See the section "[ArrayJoin function](#)".

arrayDifference(arr)

Takes an array, returns an array with the difference between all pairs of neighboring elements. For example:

```
SELECT arrayDifference([1, 2, 3, 4])
```

```
┌arrayDifference([1, 2, 3, 4])┐
│ [0,1,1,1] │
```

arrayDistinct(arr)

Takes an array, returns an array containing the different elements in all the arrays. For example:

```
SELECT arrayDistinct([1, 2, 2, 3, 1])
```

```
┌arrayDistinct([1, 2, 2, 3, 1])┐
│ [1,2,3] │
```

arrayEnumerateDense(arr)

Returns an array of the same size as the source array, indicating where each element first appears in the source array. For example: `arrayEnumerateDense([10,20,10,30]) = [1,2,1,3]`.

arrayIntersect(arr)

Takes an array, returns the intersection of all array elements. For example:

```
SELECT
  arrayIntersect([1, 2], [1, 3], [2, 3]) AS no_intersect,
  arrayIntersect([1, 2], [1, 3], [1, 4]) AS intersect
```

```
┌no_intersect┐┌intersect┐
| []         | [1]       |
└──────────┘└────────┘
```

arrayReduce(agg_func, arr1, ...)

Applies an aggregate function to array and returns its result. If aggregate function has multiple arguments, then this function can be applied to multiple arrays of the same size.

arrayReduce('agg_func', arr1, ...) - apply the aggregate function **agg_func** to arrays **arr1....** If multiple arrays passed, then elements on corresponding positions are passed as multiple arguments to the aggregate function. For example: SELECT arrayReduce('max', [1,2,3]) = 3

arrayReverse(arr)

Returns an array of the same size as the source array, containing the result of inverting all elements of the source array.

Functions for splitting and merging strings and arrays

splitByChar(separator, s)

Splits a string into substrings separated by 'separator'. 'separator' must be a string constant consisting of exactly one character.

Returns an array of selected substrings. Empty substrings may be selected if the separator occurs at the beginning or end of the string, or if there are multiple consecutive separators.

splitByString(separator, s)

The same as above, but it uses a string of multiple characters as the separator. The string must be non-empty.

arrayStringConcat(arr[, separator])

Concatenates the strings listed in the array with the separator. 'separator' is an optional parameter: a constant string, set to an empty string by default.

Returns the string.

alphaTokens(s)

Selects substrings of consecutive bytes from the ranges a-z and A-Z. Returns an array of substrings.

Example:

```
SELECT alphaTokens('abca1abc')
```

```
┌alphaTokens('abca1abc')┐
| ['abca','abc']        |
└──────────────────┘
```

Bit functions

Bit functions work for any pair of types from UInt8, UInt16, UInt32, UInt64, Int8, Int16, Int32, Int64, Float32, or Float64.

The result type is an integer with bits equal to the maximum bits of its arguments. If at least one of the arguments is signed, the result is a signed number. If an argument is a floating-point number, it is cast to Int64.

bitAnd(a, b)

bitOr(a, b)

bitXor(a, b)

bitNot(a)

bitShiftLeft(a, b)

bitShiftRight(a, b)

bitRotateLeft(a, b)

bitRotateRight(a, b)

bitTest(a, b)

bitTestAll(a, b)

bitTestAny(a, b)

Bitmap functions

Bitmap functions work for two bitmaps Object value calculation, it is to return new bitmap or cardinality while using formula calculation, such as and, or, xor, and not, etc.

There are 2 kinds of construction methods for Bitmap Object. One is to be constructed by aggregation function groupBitmap with -State, the other is to be constructed by Array Object. It is also to convert Bitmap Object to Array Object.

RoaringBitmap is wrapped into a data structure while actual storage of Bitmap objects. When the cardinality is less than or equal to 32, it uses Set objet. When the cardinality is greater than 32, it uses RoaringBitmap object. That is why storage of low cardinality set is faster.

For more information on RoaringBitmap, see: [CRoaring](#).

bitmapBuild

Build a bitmap from unsigned integer array.

```
bitmapBuild(array)
```

Parameters

- **array** – unsigned integer array.

Example

```
SELECT bitmapBuild([1, 2, 3, 4, 5]) AS res
```

bitmapToArray

Convert bitmap to integer array.

```
bitmapToArray(bitmap)
```

Parameters

- **bitmap** – bitmap object.

Example

```
SELECT bitmapToArray(bitmapBuild([1, 2, 3, 4, 5])) AS res
```

```
┌res┐  
└───┘  
| [1,2,3,4,5] |  
└───┘
```

bitmapHasAny

Analogous to `hasAny(array, array)` returns 1 if bitmaps have any common elements, 0 otherwise.
For empty bitmaps returns 0.

```
bitmapHasAny(bitmap,bitmap)
```

Parameters

- `bitmap` – bitmap object.

Example

```
SELECT bitmapHasAny(bitmapBuild([1,2,3]),bitmapBuild([3,4,5])) AS res
```

```
┌res┐  
│ 1 │  
└───┘
```

bitmapHasAll

Analogous to `hasAll(array, array)` returns 1 if the first bitmap contains all the elements of the second one, 0 otherwise.
If the second argument is an empty bitmap then returns 1.

```
bitmapHasAll(bitmap,bitmap)
```

Parameters

- `bitmap` – bitmap object.

Example

```
SELECT bitmapHasAll(bitmapBuild([1,2,3]),bitmapBuild([3,4,5])) AS res
```

```
┌res┐  
│ 0 │  
└───┘
```

bitmapAnd

Two bitmap and calculation, the result is a new bitmap.

```
bitmapAnd(bitmap,bitmap)
```

Parameters

- `bitmap` – bitmap object.

Example

```
SELECT bitmapToArray(bitmapAnd(bitmapBuild([1,2,3]),bitmapBuild([3,4,5]))) AS res
```

```
┌res┐  
│ [3] │  
└───┘
```

bitmapOr

Two bitmap or calculation, the result is a new bitmap.

```
bitmapOr(bitmap,bitmap)
```

Parameters

- `bitmap` – bitmap object.

Example


```
SELECT bitmapToArray(bitmapOr(bitmapBuild([1,2,3]),bitmapBuild([3,4,5]))) AS res
```

```
┌res┐  
└─┘  
| [1,2,3,4,5] |
```

bitmapXor

Two bitmap xor calculation, the result is a new bitmap.

```
bitmapXor(bitmap,bitmap)
```

Parameters

- **bitmap** – bitmap object.

Example

```
SELECT bitmapToArray(bitmapXor(bitmapBuild([1,2,3]),bitmapBuild([3,4,5]))) AS res
```

```
┌res┐  
└─┘  
| [1,2,4,5] |
```

bitmapAndnot

Two bitmap andnot calculation, the result is a new bitmap.

```
bitmapAndnot(bitmap,bitmap)
```

Parameters

- **bitmap** – bitmap object.

Example

```
SELECT bitmapToArray(bitmapAndnot(bitmapBuild([1,2,3]),bitmapBuild([3,4,5]))) AS res
```

```
┌res┐  
└─┘  
| [1,2] |
```

bitmapCardinality

Retrun bitmap cardinality of type UInt64.

```
bitmapCardinality(bitmap)
```

Parameters

- **bitmap** – bitmap object.

Example

```
SELECT bitmapCardinality(bitmapBuild([1, 2, 3, 4, 5])) AS res
```

```
┌res┐  
└─┘  
| 5 |
```

bitmapAndCardinality

Two bitmap and calculation, return cardinality of type UInt64.

```
bitmapAndCardinality(bitmap,bitmap)
```

Parameters

- **bitmap** – bitmap object.

Example

```
SELECT bitmapAndCardinality(bitmapBuild([1,2,3]),bitmapBuild([3,4,5])) AS res;
```

```
┌res┐  
└───┘  
| 1 |  
└───┘
```

bitmapOrCardinality

Two bitmap or calculation, return cardinality of type UInt64.

```
bitmapOrCardinality(bitmap,bitmap)
```

Parameters

- **bitmap** – bitmap object.

Example

```
SELECT bitmapOrCardinality(bitmapBuild([1,2,3]),bitmapBuild([3,4,5])) AS res;
```

```
┌res┐  
└───┘  
| 5 |  
└───┘
```

bitmapXorCardinality

Two bitmap xor calculation, return cardinality of type UInt64.

```
bitmapXorCardinality(bitmap,bitmap)
```

Parameters

- **bitmap** – bitmap object.

Example

```
SELECT bitmapXorCardinality(bitmapBuild([1,2,3]),bitmapBuild([3,4,5])) AS res;
```

```
┌res┐  
└───┘  
| 4 |  
└───┘
```

bitmapAndnotCardinality

Two bitmap andnot calculation, return cardinality of type UInt64.

```
bitmapAndnotCardinality(bitmap,bitmap)
```

Parameters

- **bitmap** – bitmap object.

Example

```
SELECT bitmapAndnotCardinality(bitmapBuild([1,2,3]),bitmapBuild([3,4,5])) AS res;
```

```
┌res┐  
└───┘  
| 2 |  
└───┘
```

Hash functions

Hash functions can be used for deterministic pseudo-random shuffling of elements.

halfMD5

Interprets all the input parameters as strings and calculates the MD5 hash value for each of them. Then combines hashes. Then from the resulting string, takes the first 8 bytes of the hash and interprets them as **UInt64** in big-endian byte order.

```
halfMD5(par1, ...)
```

The function works relatively slow (5 million short strings per second per processor core). Consider using the **sipHash64** function instead.

Parameters

The function takes a variable number of input parameters. Parameters can be any of the **supported data types**.

Returned Value

Hash value having the **UInt64** data type.

Example

```
SELECT halfMD5(array('e','x','a'), 'mple', 10, toDateTime('2019-06-15 23:00:00')) AS halfMD5hash, toTypeName(halfMD5hash) AS type
```

halfMD5hash	type
186182704141653334	UInt64

MD5

Calculates the MD5 from a string and returns the resulting set of bytes as FixedString(16).

If you don't need MD5 in particular, but you need a decent cryptographic 128-bit hash, use the 'sipHash128' function instead.

If you want to get the same result as output by the md5sum utility, use lower(hex(MD5(s))).

sipHash64

Produces 64-bit **SipHash** hash value.

```
sipHash64(par1,...)
```

This function **interprets** all the input parameters as strings and calculates the hash value for each of them. Then combines hashes.

This is a cryptographic hash function. It works at least three times faster than the **MD5** function.

Parameters

The function takes a variable number of input parameters. Parameters can be any of the **supported data types**.

Returned Value

Hash value having the **UInt64** data type.

Example

```
SELECT sipHash64(array('e','x','a'), 'mple', 10, toDateTime('2019-06-15 23:00:00')) AS SipHash, toTypeName(SipHash) AS type
```

SipHash	type
13726873534472839665	UInt64

sipHash128

Calculates SipHash from a string.
Accepts a String-type argument. Returns FixedString(16).
Differs from sipHash64 in that the final xor-folding state is only done up to 128 bytes.

cityHash64

Produces 64-bit hash value.

```
cityHash64(par1,...)
```

This is the fast non-cryptographic hash function. It uses [CityHash](#) algorithm for string parameters and implementation-specific fast non-cryptographic hash function for the parameters with other data types. To get the final result, the function uses the CityHash combinator.

Parameters

The function takes a variable number of input parameters. Parameters can be any of the [supported data types](#).

Returned Value

Hash value having the [UInt64](#) data type.

Examples

Call example:

```
SELECT cityHash64(array('e','x','a'), 'mple', 10, toDateTime('2019-06-15 23:00:00')) AS CityHash, toTypeName(CityHash) AS type
```



The following example shows how to compute the checksum of the entire table with accuracy up to the row order:

```
SELECT sum(cityHash64(*)) FROM table
```

intHash32

Calculates a 32-bit hash code from any type of integer.
This is a relatively fast non-cryptographic hash function of average quality for numbers.

intHash64

Calculates a 64-bit hash code from any type of integer.
It works faster than intHash32. Average quality.

SHA1

SHA224

SHA256

Calculates SHA-1, SHA-224, or SHA-256 from a string and returns the resulting set of bytes as FixedString(20), FixedString(28), or FixedString(32).
The function works fairly slowly (SHA-1 processes about 5 million short strings per second per processor core, while SHA-224 and SHA-256 process about 2.2 million).
We recommend using this function only in cases when you need a specific hash function and you can't select it. Even in these cases, we recommend applying the function offline and pre-calculating values when inserting them into the table, instead of applying it in SELECTS.

URLHash(url[, N])

A fast, decent-quality non-cryptographic hash function for a string obtained from a URL using some type of normalization.

URLHash(s) – Calculates a hash from a string without one of the trailing symbols **/**, **?** or **#** at the end, if present.

URLHash(s, N) – Calculates a hash from a string up to the N level in the URL hierarchy, without one of the trailing symbols **/**, **?** or **#** at the end, if present.

Levels are the same as in URLHierarchy. This function is specific to Yandex.Metrica.

farmHash64

Produces a 64-bit **FarmHash** hash value.

```
farmHash64(par1, ...)
```

The function uses the **Hash64** method from all **available methods**.

Parameters

The function takes a variable number of input parameters. Parameters can be any of the **supported data types**.

Returned Value

Hash value having the **UInt64** data type.

Example

```
SELECT farmHash64(array('e','x','a'), 'mple', 10, toDateTime('2019-06-15 23:00:00')) AS FarmHash, toTypeName(FarmHash) AS type
```

FarmHash		type
17790458267262532859	UInt64	

javaHash

Calculates JavaHash from a string.

Accepts a String-type argument. Returns Int32.

For more information, see the link: **JavaHash**

hiveHash

Calculates HiveHash from a string.

Accepts a String-type argument. Returns Int32.

Same as for **JavaHash**, except that the return value never has a negative number.

metroHash64

Produces a 64-bit **MetroHash** hash value.

```
metroHash64(par1, ...)
```

Parameters

The function takes a variable number of input parameters. Parameters can be any of the **supported data types**.

Returned Value

Hash value having the **UInt64** data type.

Example

```
SELECT metroHash64(array('e','x','a'), 'mple', 10, toDateTime('2019-06-15 23:00:00')) AS MetroHash, toTypeName(MetroHash) AS type
```

MetroHash		type
14235658766382344533		UInt64

jumpConsistentHash

Calculates JumpConsistentHash form a UInt64.
 Accepts a UInt64-type argument. Returns Int32.
 For more information, see the link: [JumpConsistentHash](#)

murmurHash2_32, murmurHash2_64

Produces a **MurmurHash2** hash value.

murmurHash2_32(par1, ...)
murmurHash2_64(par1, ...)

Parameters

Both functions take a variable number of input parameters. Parameters can be any of the **supported data types**.

Returned Value

- The **murmurHash2_32** function returns hash value having the **UInt32** data type.
- The **murmurHash2_64** function returns hash value having the **UInt64** data type.

Example

SELECT murmurHash2_64(array('e','x','a'), 'mple', 10, toDateTime('2019-06-15 23:00:00')) AS MurmurHash2, toTypeName(MurmurHash2) AS type
--

MurmurHash2		type
11832096901709403633		UInt64

murmurHash3_32, murmurHash3_64

Produces a **MurmurHash3** hash value.

murmurHash3_32(par1, ...)
murmurHash3_64(par1, ...)

Parameters

Both functions take a variable number of input parameters. Parameters can be any of the **supported data types**.

Returned Value

- The **murmurHash3_32** function returns hash value having the **UInt32** data type.
- The **murmurHash3_64** function returns hash value having the **UInt64** data type.

Example

SELECT murmurHash3_32(array('e','x','a'), 'mple', 10, toDateTime('2019-06-15 23:00:00')) AS MurmurHash3, toTypeName(MurmurHash3) AS type
--

MurmurHash3		type
2152717		UInt32

murmurHash3_128

Produces a 128-bit **MurmurHash3** hash value.

murmurHash3_128(expr)

Parameters

- `expr` — Expressions returning `String`-typed value.

Returned Value

Hash value having `FixedString(16)` data type.

Example

```
SELECT murmurHash3_128('example_string') AS MurmurHash3, toTypeName(MurmurHash3) AS type
```

MurmurHash3	type
614"55KT~q	FixedString(16)

xxHash32, xxHash64

Calculates xxHash from a string.

Accepts a `String`-type argument. Returns `UInt64` Or `UInt32`.

For more information, see the link: [xxHash](#)

Functions for generating pseudo-random numbers

Non-cryptographic generators of pseudo-random numbers are used.

All the functions accept zero arguments or one argument.

If an argument is passed, it can be any type, and its value is not used for anything.

The only purpose of this argument is to prevent common subexpression elimination, so that two different instances of the same function return different columns with different random numbers.

rand

Returns a pseudo-random `UInt32` number, evenly distributed among all `UInt32`-type numbers.

Uses a linear congruential generator.

rand64

Returns a pseudo-random `UInt64` number, evenly distributed among all `UInt64`-type numbers.

Uses a linear congruential generator.

randConstant

Returns a pseudo-random `UInt32` number, The value is one for different blocks.

Encoding functions

hex

Accepts arguments of types: `String`, `unsigned integer`, `Date`, or `DateTime`. Returns a string containing the argument's hexadecimal representation. Uses uppercase letters `A-F`. Does not use `0x` prefixes or `h` suffixes. For strings, all bytes are simply encoded as two hexadecimal numbers. Numbers are converted to big endian ("human readable") format. For numbers, older zeros are trimmed, but only by entire bytes. For example, `hex(1) = '01'`. `Date` is encoded as the number of days since the beginning of the Unix epoch. `DateTime` is encoded as the number of seconds since the beginning of the Unix epoch.

unhex(str)

Accepts a string containing any number of hexadecimal digits, and returns a string containing the corresponding bytes. Supports both uppercase and lowercase letters A-F. The number of hexadecimal digits does not have to be even. If it is odd, the last digit is interpreted as the younger half of the 00-0F byte. If the argument string contains anything other than hexadecimal digits, some implementation-defined result is returned (an exception isn't thrown).

If you want to convert the result to a number, you can use the 'reverse' and 'reinterpretAsType' functions.

UUIDStringToNum(str)

Accepts a string containing 36 characters in the format `123e4567-e89b-12d3-a456-426655440000`, and returns it as a set of bytes in a `FixedString(16)`.

UUIDNumToString(str)

Accepts a `FixedString(16)` value. Returns a string containing 36 characters in text format.

bitmaskToList(num)

Accepts an integer. Returns a string containing the list of powers of two that total the source number when summed. They are comma-separated without spaces in text format, in ascending order.

bitmaskToArray(num)

Accepts an integer. Returns an array of `UInt64` numbers containing the list of powers of two that total the source number when summed. Numbers in the array are in ascending order.

Functions for working with UUID

The functions for working with UUID are listed below.

generateUUIDv4

Generates the **UUID** of **version 4**.

```
generateUUIDv4()
```

Returned value

The UUID type value.

Usage example

This example demonstrates creating a table with the UUID type column and inserting a value into the table.

```
:) CREATE TABLE t_uuid (x UUID) ENGINE=TinyLog
:) INSERT INTO t_uuid SELECT generateUUIDv4()
:) SELECT * FROM t_uuid
```

f4bf890f-f9dc-4332-ad5c-0c18e73f28e9

toUUID (x)

Converts String type value to UUID type.

```
toUUID(String)
```

Returned value

The UUID type value.

Usage example

```
:) SELECT toUUID('61f0c404-5cb3-11e7-907b-a6006ad3dba0') AS uuid
```

uuid
61f0c404-5cb3-11e7-907b-a6006ad3dba0

UUIDStringToNum

Accepts a string containing 36 characters in the format `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`, and returns it as a set of bytes in a [FixedString\(16\)](#).

```
UUIDStringToNum(String)
```

Returned value

FixedString(16)

Usage examples

```
:) SELECT
  '612f3c40-5d3b-217e-707b-6a546a3d7b29' AS uuid,
  UUIDStringToNum(uuid) AS bytes
```

uuid	bytes
612f3c40-5d3b-217e-707b-6a546a3d7b29	a/<@];!~p{jTj={}

UUIDNumToString

Accepts a [FixedString\(16\)](#) value, and returns a string containing 36 characters in text format.

```
UUIDNumToString(FixedString(16))
```

Returned value

String.

Usage example

```
SELECT
  'a/<@];!~p{jTj={}' AS bytes,
  UUIDNumToString(toFixedString(bytes, 16)) AS uuid
```

bytes	uuid
a/<@];!~p{jTj={}	612f3c40-5d3b-217e-707b-6a546a3d7b29

See also

- [dictGetUUID](#)
- [dictGetUUIDOrDefault](#)

Functions for working with URLs

All these functions don't follow the RFC. They are maximally simplified for improved performance.

Functions that extract part of a URL

If there isn't anything similar in a URL, an empty string is returned.

protocol

Returns the protocol. Examples: http, ftp, mailto, magnet...

domain

Gets the domain.

domainWithoutWWW

Returns the domain and removes no more than one 'www.' from the beginning of it, if present.

topLevelDomain

Returns the top-level domain. Example: .ru.

firstSignificantSubdomain

Returns the "first significant subdomain". This is a non-standard concept specific to Yandex.Metrica. The first significant subdomain is a second-level domain if it is 'com', 'net', 'org', or 'co'. Otherwise, it is a third-level domain. For example, `firstSignificantSubdomain('https://news.yandex.ru/') = 'yandex '`, `firstSignificantSubdomain('https://news.yandex.com.tr/') = 'yandex '`. The list of "insignificant" second-level domains and other implementation details may change in the future.

cutToFirstSignificantSubdomain

Returns the part of the domain that includes top-level subdomains up to the "first significant subdomain" (see the explanation above).

For example, `cutToFirstSignificantSubdomain('https://news.yandex.com.tr/') = 'yandex.com.tr'`.

path

Returns the path. Example: `/top/news.html` The path does not include the query string.

pathFull

The same as above, but including query string and fragment. Example: `/top/news.html?page=2#comments`

queryString

Returns the query string. Example: `page=1&lr=213`. query-string does not include the initial question mark, as well as `#` and everything after `#`.

fragment

Returns the fragment identifier. fragment does not include the initial hash symbol.

queryStringAndFragment

Returns the query string and fragment identifier. Example: `page=1#29390`.

extractURLParameter(URL, name)

Returns the value of the 'name' parameter in the URL, if present. Otherwise, an empty string. If there are many parameters with this name, it returns the first occurrence. This function works under the assumption that the parameter name is encoded in the URL exactly the same way as in the passed argument.

extractURLParameters(URL)

Returns an array of name=value strings corresponding to the URL parameters. The values are not decoded in any way.

extractURLParameterNames(URL)

Returns an array of name strings corresponding to the names of URL parameters. The values are not decoded in any way.

URLHierarchy(URL)

Returns an array containing the URL, truncated at the end by the symbols `/,?` in the path and query-string. Consecutive separator characters are counted as one. The cut is made in the position after all the consecutive separator characters. Example:

URLPathHierarchy(URL)

The same as above, but without the protocol and host in the result. The `/` element (root) is not included. Example: the function is used to implement tree reports the URL in Yandex. Metric.

```
URLPathHierarchy('https://example.com/browse/CONV-6788') =  
[  
  '/browse/',  
  '/browse/CONV-6788'  
]
```

decodeURLComponent(URL)

Returns the decoded URL.

Example:

```
SELECT decodeURLComponent('http://127.0.0.1:8123/?query=SELECT%201%3B') AS DecodedURL;
```

```
└─DecodedURL─┐  
| http://127.0.0.1:8123/?query=SELECT 1; |  
└───────────┘
```

Functions that remove part of a URL.

If the URL doesn't have anything similar, the URL remains unchanged.

cutWWW

Removes no more than one `'www.'` from the beginning of the URL's domain, if present.

cutQueryString

Removes query string. The question mark is also removed.

cutFragment

Removes the fragment identifier. The number sign is also removed.

cutQueryStringAndFragment

Removes the query string and fragment identifier. The question mark and number sign are also removed.

cutURLParameter(URL, name)

Removes the `'name'` URL parameter, if present. This function works under the assumption that the parameter name is encoded in the URL exactly the same way as in the passed argument.

Functions for working with IP addresses

IPv4NumToString(num)

Takes a `UInt32` number. Interprets it as an IPv4 address in big endian. Returns a string containing the corresponding IPv4 address in the format `A.B.C.d` (dot-separated numbers in decimal form).

IPv4StringToNum(s)

The reverse function of `IPv4NumToString`. If the IPv4 address has an invalid format, it returns 0.

IPv4NumToStringClassC(num)

Similar to `IPv4NumToString`, but using `xxx` instead of the last octet.

Example:

```
SELECT
  IPv4NumToStringClassC(ClientIP) AS k,
  count() AS c
FROM test.hits
GROUP BY k
ORDER BY c DESC
LIMIT 10
```

k	c
83.149.9.xxx	26238
217.118.81.xxx	26074
213.87.129.xxx	25481
83.149.8.xxx	24984
217.118.83.xxx	22797
78.25.120.xxx	22354
213.87.131.xxx	21285
78.25.121.xxx	20887
188.162.65.xxx	19694
83.149.48.xxx	17406

Since using 'xxx' is highly unusual, this may be changed in the future. We recommend that you don't rely on the exact format of this fragment.

IPv6NumToString(x)

Accepts a FixedString(16) value containing the IPv6 address in binary format. Returns a string containing this address in text format.

IPv6-mapped IPv4 addresses are output in the format ::ffff:111.222.33.44. Examples:

```
SELECT IPv6NumToString(toFixedString(unhex('2A0206B8000000000000000000000011'), 16)) AS addr
```

addr
2a02:6b8::11

```
SELECT
  IPv6NumToString(ClientIP6 AS k),
  count() AS c
FROM hits_all
WHERE EventDate = today() AND substring(ClientIP6, 1, 12) != unhex('00000000000000000000FFFF')
GROUP BY k
ORDER BY c DESC
LIMIT 10
```

IPv6NumToString(ClientIP6)	c
2a02:2168:aaa:bbbb::2	24695
2a02:2698:abcd:abcd:abcd:8888:5555	22408
2a02:6b8:0:fff::ff	16389
2a01:4f8:111:6666::2	16016
2a02:2168:888:222::1	15896
2a01:7e00::ffff:ffff:ffff:222	14774
2a02:8109:eee:ee:eeee:eeee:eeee:eeee	14443
2a02:810b:8888:888:8888:8888:8888:8888	14345
2a02:6b8:0:444:4444:4444:4444:4444	14279
2a01:7e00::ffff:ffff:ffff:ffff	13880

```
SELECT
  IPv6NumToString(ClientIP6 AS k),
  count() AS c
FROM hits_all
WHERE EventDate = today()
GROUP BY k
ORDER BY c DESC
LIMIT 10
```

IPv6NumToString(ClientIP6)	c
::ffff:94.26.111.111	747440
::ffff:37.143.222.4	529483
::ffff:5.166.111.99	317707
::ffff:46.38.11.77	263086
::ffff:79.105.111.111	186611
::ffff:93.92.111.88	176773
::ffff:84.53.111.33	158709
::ffff:217.118.11.22	154004
::ffff:217.118.11.33	148449
::ffff:217.118.11.44	148243

IPv6StringToNum(s)

The reverse function of IPv6NumToString. If the IPv6 address has an invalid format, it returns a string of null bytes. HEX can be uppercase or lowercase.

IPv4ToIPv6(x)

Takes a UInt32 number. Interprets it as an IPv4 address in big endian. Returns a FixedString(16) value containing the IPv6 address in binary format. Examples:

```
SELECT IPv6NumToString(IPv4ToIPv6(IPv4StringToNum('192.168.0.1'))) AS addr
```

addr
::ffff:192.168.0.1

cutIPv6(x, bitsToCutForIPv6, bitsToCutForIPv4)

Accepts a FixedString(16) value containing the IPv6 address in binary format. Returns a string containing the address of the specified number of bits removed in text format. For example:

```
WITH
  IPv6StringToNum('2001:0DB8:AC10:FE01:FEED:BABE:CAFE:F00D') AS ipv6,
  IPv4ToIPv6(IPv4StringToNum('192.168.0.1')) AS ipv4
SELECT
  cutIPv6(ipv6, 2, 0),
  cutIPv6(ipv4, 0, 2)
```

cutIPv6(ipv6, 2, 0)	cutIPv6(ipv4, 0, 2)
2001:db8:ac10:fe01:feed:babe:cafe:0	::ffff:192.168.0.0

IPv4CIDRtoIPv4Range(ipv4, cidr),

Accepts an IPv4 and an UInt8 value containing the CIDR. Return a tuple with two IPv4 containing the lower range and the higher range of the subnet.

```
SELECT IPv4CIDRtoIPv4Range(toIPv4('192.168.5.2'), 16)
```

IPv4CIDRtoIPv4Range(toIPv4('192.168.5.2'), 16)
('192.168.0.0','192.168.255.255')

IPv6CIDRtoIPv6Range(ipv6, cidr),

Accepts an IPv6 and an UInt8 value containing the CIDR. Return a tuple with two IPv6 containing the lower range and the higher range of the subnet.

```
SELECT IPv6CIDRtoIPv6Range(toIPv6('2001:0db8:0000:85a3:0000:0000:ac1f:8001'), 32);
```

```
┌IPv6CIDRtoIPv6Range(toIPv6('2001:0db8:0000:85a3:0000:0000:ac1f:8001'), 32)┐  
├ ('2001:db8::','2001:db8:ffff:ffff:ffff:ffff:ffff:ffff')                │  
└──────────────────────────────────────────────────────────────────────────┘
```

toIPv4(string)

An alias to [IPv4StringToNum\(\)](#) that takes a string form of IPv4 address and returns value of **IPv4** type, which is binary equal to value returned by [IPv4StringToNum\(\)](#).

```
WITH  
  '171.225.130.45' as IPv4_string  
SELECT  
  toTypeName(IPv4StringToNum(IPv4_string)),  
  toTypeName(toIPv4(IPv4_string))
```

```
┌toTypeName(IPv4StringToNum(IPv4_string))┐toTypeName(toIPv4(IPv4_string))┐  
├ UInt32                                │ IPv4                            │  
└────────────────────────────────────────┘└────────────────────────────────┘
```

```
WITH  
  '171.225.130.45' as IPv4_string  
SELECT  
  hex(IPv4StringToNum(IPv4_string)),  
  hex(toIPv4(IPv4_string))
```

```
┌hex(IPv4StringToNum(IPv4_string))┐hex(toIPv4(IPv4_string))┐  
├ ABE1822D                        │ ABE1822D                    │  
└──────────────────────────────────┘└──────────────────────────┘
```

toIPv6(string)

An alias to [IPv6StringToNum\(\)](#) that takes a string form of IPv6 address and returns value of **IPv6** type, which is binary equal to value returned by [IPv6StringToNum\(\)](#).

```
WITH  
  '2001:438:ffff::407d:1bc1' as IPv6_string  
SELECT  
  toTypeName(IPv6StringToNum(IPv6_string)),  
  toTypeName(toIPv6(IPv6_string))
```

```
┌toTypeName(IPv6StringToNum(IPv6_string))┐toTypeName(toIPv6(IPv6_string))┐  
├ FixedString(16)                        │ IPv6                            │  
└────────────────────────────────────────┘└────────────────────────────────┘
```

```
WITH  
  '2001:438:ffff::407d:1bc1' as IPv6_string  
SELECT  
  hex(IPv6StringToNum(IPv6_string)),  
  hex(toIPv6(IPv6_string))
```

```
┌hex(IPv6StringToNum(IPv6_string))┐hex(toIPv6(IPv6_string))┐  
├ 20010438FFFF00000000000000407D1BC1 │ 20010438FFFF00000000000000407D1BC1 │  
└────────────────────────────────────────┘└────────────────────────────────┘
```

Functions for working with JSON

In Yandex.Metrica, JSON is transmitted by users as session parameters. There are some special functions for working with this JSON. (Although in most of the cases, the JSONs are additionally pre-processed, and the resulting values are put in separate columns in their processed format.) All these functions are based on strong assumptions about what the JSON can be, but they try to do as little as possible to get the job done.

The following assumptions are made:

1. The field name (function argument) must be a constant.
2. The field name is somehow canonically encoded in JSON. For example: `visitParamHas({'"abc": "def"}', 'abc') = 1`, but `visitParamHas({'"\u0061\u0062\u0063": "def"}', 'abc') = 0`
3. Fields are searched for on any nesting level, indiscriminately. If there are multiple matching fields, the first occurrence is used.
4. The JSON doesn't have space characters outside of string literals.

visitParamHas(params, name)

Checks whether there is a field with the 'name' name.

visitParamExtractUInt(params, name)

Parses UInt64 from the value of the field named 'name'. If this is a string field, it tries to parse a number from the beginning of the string. If the field doesn't exist, or it exists but doesn't contain a number, it returns 0.

visitParamExtractInt(params, name)

The same as for Int64.

visitParamExtractFloat(params, name)

The same as for Float64.

visitParamExtractBool(params, name)

Parses a true/false value. The result is UInt8.

visitParamExtractRaw(params, name)

Returns the value of a field, including separators.

Examples:

```
visitParamExtractRaw({'"abc": "\n\u0000"}', 'abc') = '"\n\u0000"'
visitParamExtractRaw({'"abc": {"def": [1,2,3]}'}', 'abc') = '{"def": [1,2,3]}'
```

visitParamExtractString(params, name)

Parses the string in double quotes. The value is unescaped. If unescaping failed, it returns an empty string.

Examples:

```
visitParamExtractString({'"abc": "\n\u0000"}', 'abc') = '\n\0'
visitParamExtractString({'"abc": "\u263a"}', 'abc') = '☺'
visitParamExtractString({'"abc": "\u263"}', 'abc') = ''
visitParamExtractString({'"abc": "hello"}', 'abc') = ''
```

There is currently no support for code points in the format `\uXXXX\uYYYY` that are not from the basic multilingual plane (they are converted to CESU-8 instead of UTF-8).

The following functions are based on `simdjson` designed for more complex JSON parsing requirements. The assumption 2 mentioned above still applies.

JSONHas(json[, indices_or_keys]...)

If the value exists in the JSON document, `1` will be returned.

If the value does not exist, **0** will be returned.

Examples:

```
select JSONHas('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b') = 1
select JSONHas('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b', 4) = 0
```

indices_or_keys is a list of zero or more arguments each of them can be either string or integer.

- String = access object member by key.
- Positive integer = access the n-th member/key from the beginning.
- Negative integer = access the n-th member/key from the end.

You may use integers to access both JSON arrays and JSON objects.

So, for example:

```
select JSONExtractKey('{ "a": "hello", "b": [-100, 200.0, 300] }', 1) = 'a'
select JSONExtractKey('{ "a": "hello", "b": [-100, 200.0, 300] }', 2) = 'b'
select JSONExtractKey('{ "a": "hello", "b": [-100, 200.0, 300] }', -1) = 'b'
select JSONExtractKey('{ "a": "hello", "b": [-100, 200.0, 300] }', -2) = 'a'
select JSONExtractString('{ "a": "hello", "b": [-100, 200.0, 300] }', 1) = 'hello'
```

JSONLength(json[, indices_or_keys]...)

Return the length of a JSON array or a JSON object.

If the value does not exist or has a wrong type, **0** will be returned.

Examples:

```
select JSONLength('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b') = 3
select JSONLength('{ "a": "hello", "b": [-100, 200.0, 300] }') = 2
```

JSONType(json[, indices_or_keys]...)

Return the type of a JSON value.

If the value does not exist, **Null** will be returned.

Examples:

```
select JSONType('{ "a": "hello", "b": [-100, 200.0, 300] }') = 'Object'
select JSONType('{ "a": "hello", "b": [-100, 200.0, 300] }', 'a') = 'String'
select JSONType('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b') = 'Array'
```

JSONExtractUInt(json[, indices_or_keys]...)

JSONExtractInt(json[, indices_or_keys]...)

JSONExtractFloat(json[, indices_or_keys]...)

JSONExtractBool(json[, indices_or_keys]...)

Parses a JSON and extract a value. These functions are similar to **visitParam** functions.

If the value does not exist or has a wrong type, **0** will be returned.

Examples:

```
select JSONExtractInt('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b', 1) = -100
select JSONExtractFloat('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b', 2) = 200.0
select JSONExtractUInt('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b', -1) = 300
```

JSONExtractString(json[, indices_or_keys]...)

Parses a JSON and extract a string. This function is similar to [visitParamExtractString](#) functions.

If the value does not exist or has a wrong type, an empty string will be returned.

The value is unescaped. If unescaping failed, it returns an empty string.

Examples:

```
select JSONExtractString('{ "a": "hello", "b": [-100, 200.0, 300] }', 'a') = 'hello'
select JSONExtractString('{ "abc": "\n\u0000" }', 'abc') = '\n\0'
select JSONExtractString('{ "abc": "\u263a" }', 'abc') = '☺'
select JSONExtractString('{ "abc": "\u263" }', 'abc') = ''
select JSONExtractString('{ "abc": "hello" }', 'abc') = ''
```

JSONExtract(json[, indices_or_keys...], return_type)

Parses a JSON and extract a value of the given ClickHouse data type.

This is a generalization of the previous [JSONExtract<type>](#) functions.

This means

[JSONExtract\(..., 'String'\)](#) returns exactly the same as [JSONExtractString\(\)](#),

[JSONExtract\(..., 'Float64'\)](#) returns exactly the same as [JSONExtractFloat\(\)](#).

Examples:

```
SELECT JSONExtract('{ "a": "hello", "b": [-100, 200.0, 300] }', 'Tuple(String, Array(Float64))') = ('hello',[-100,200,300])
SELECT JSONExtract('{ "a": "hello", "b": [-100, 200.0, 300] }', 'Tuple(b Array(Float64), a String)') = ([-100,200,300],'hello')
SELECT JSONExtract('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b', 'Array(Nullable(Int8))') = [-100, NULL, NULL]
SELECT JSONExtract('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b', 4, 'Nullable(Int64)') = NULL
SELECT JSONExtract('{ "passed": true }', 'passed', 'UInt8') = 1
SELECT JSONExtract('{ "day": "Thursday" }', 'day', 'Enum8(\'Sunday\' = 0, \'Monday\' = 1, \'Tuesday\' = 2, \'Wednesday\' = 3, \'Thursday\' = 4, \'Friday\' = 5, \'Saturday\' = 6)') = 'Thursday'
SELECT JSONExtract('{ "day": 5 }', 'day', 'Enum8(\'Sunday\' = 0, \'Monday\' = 1, \'Tuesday\' = 2, \'Wednesday\' = 3, \'Thursday\' = 4, \'Friday\' = 5, \'Saturday\' = 6)') = 'Friday'
```

JSONExtractKeysAndValues(json[, indices_or_keys...], value_type)

Parse key-value pairs from a JSON where the values are of the given ClickHouse data type.

Example:

```
SELECT JSONExtractKeysAndValues('{ "x": { "a": 5, "b": 7, "c": 11 } }', 'x', 'Int8') = [(('a',5),('b',7),('c',11))];
```

JSONExtractRaw(json[, indices_or_keys]...)

Returns a part of JSON.

If the part does not exist or has a wrong type, an empty string will be returned.

Example:

```
select JSONExtractRaw('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b') = '[-100, 200.0, 300]'
```

Higher-order functions

-> operator, lambda(params, expr) function

Allows describing a lambda function for passing to a higher-order function. The left side of the arrow has a formal parameter, which is any ID, or multiple formal parameters – any IDs in a tuple. The right side of the arrow has an expression that can use these formal parameters, as well as any table columns.

Examples: [x -> 2 * x](#), [str -> str != Referer](#).

Higher-order functions can only accept lambda functions as their functional argument.

A lambda function that accepts multiple arguments can be passed to a higher-order function. In this case, the higher-order function is passed several arrays of identical length that these arguments will correspond to.

For all functions other than 'arrayMap' and 'arrayFilter', the first argument (the lambda function) can be omitted. In this case, identical mapping is assumed.

arrayMap(func, arr1, ...)

Returns an array obtained from the original application of the 'func' function to each element in the 'arr' array.

arrayFilter(func, arr1, ...)

Returns an array containing only the elements in 'arr1' for which 'func' returns something other than 0.

Examples:

```
SELECT arrayFilter(x -> x LIKE '%World%', ['Hello', 'abc World']) AS res
```

```
┌res┐
│ ['abc World'] │
```

```
SELECT
  arrayFilter(
    (i, x) -> x LIKE '%World%',
    arrayEnumerate(arr),
    ['Hello', 'abc World'] AS arr)
  AS res
```

```
┌res┐
│ [2] │
```

arrayCount([func,] arr1, ...)

Returns the number of elements in the arr array for which func returns something other than 0. If 'func' is not specified, it returns the number of non-zero elements in the array.

arrayExists([func,] arr1, ...)

Returns 1 if there is at least one element in 'arr' for which 'func' returns something other than 0. Otherwise, it returns 0.

arrayAll([func,] arr1, ...)

Returns 1 if 'func' returns something other than 0 for all the elements in 'arr'. Otherwise, it returns 0.

arraySum([func,] arr1, ...)

Returns the sum of the 'func' values. If the function is omitted, it just returns the sum of the array elements.

arrayFirst(func, arr1, ...)

Returns the first element in the 'arr1' array for which 'func' returns something other than 0.

arrayFirstIndex(func, arr1, ...)

Returns the index of the first element in the 'arr1' array for which 'func' returns something other than 0.

arrayCumSum([func,] arr1, ...)

Returns an array of partial sums of elements in the source array (a running sum). If the **func** function is specified, then the values of the array elements are converted by this function before summing.

Example:

```
SELECT arrayCumSum([1, 1, 1, 1]) AS res
```

```
┌res┐
└─┘
| [1, 2, 3, 4] |
```

arrayCumSumNonNegative(arr)

Same as arrayCumSum, returns an array of partial sums of elements in the source array (a running sum). Different arrayCumSum, when then returned value contains a value less than zero, the value is replace with zero and the subsequent calculation is performed with zero parameters. For example:

```
SELECT arrayCumSumNonNegative([1, 1, -4, 1]) AS res
```

```
┌res┐
└─┘
| [1,2,0,1] |
```

arraySort([func,] arr1, ...)

Returns an array as result of sorting the elements of **arr1** in ascending order. If the **func** function is specified, sorting order is determined by the result of the function **func** applied to the elements of array (arrays)

The **Schwartzian transform** is used to improve sorting efficiency.

Example:

```
SELECT arraySort((x, y) -> y, ['hello', 'world'], [2, 1]);
```

```
┌res┐
└─┘
| ['world', 'hello'] |
```

Note that NULLs and NaNs go last (NaNs go before NULLs). For example:

```
SELECT arraySort([1, nan, 2, NULL, 3, nan, 4, NULL])
```

```
┌arraySort([1, nan, 2, NULL, 3, nan, 4, NULL])┐
└─┘
| [1,2,3,4,nan,nan,NULL,NULL] |
```

arrayReverseSort([func,] arr1, ...)

Returns an array as result of sorting the elements of **arr1** in descending order. If the **func** function is specified, sorting order is determined by the result of the function **func** applied to the elements of array (arrays)

Note that NULLs and NaNs go last (NaNs go before NULLs). For example:

```
SELECT arrayReverseSort([1, nan, 2, NULL, 3, nan, 4, NULL])
```

```
┌arrayReverseSort([1, nan, 2, NULL, 3, nan, 4, NULL])┐
└─┘
| [4,3,2,1,nan,nan,NULL,NULL] |
```

Functions for working with external dictionaries

For information on connecting and configuring external dictionaries, see [External dictionaries](#).

dictGetUInt8, dictGetUInt16, dictGetUInt32, dictGetUInt64

dictGetInt8, dictGetInt16, dictGetInt32, dictGetInt64

dictGetFloat32, dictGetFloat64

dictGetDate, dictGetDateTime

dictGetUUID

dictGetString

`dictGetT('dict_name', 'attr_name', id)`

- Get the value of the `attr_name` attribute from the `dict_name` dictionary using the 'id' key. `dict_name` and `attr_name` are constant strings. `id` must be UInt64.
If there is no `id` key in the dictionary, it returns the default value specified in the dictionary description.

dictGetTOrDefault

`dictGetTOrDefault('dict_name', 'attr_name', id, default)`

The same as the `dictGetT` functions, but the default value is taken from the function's last argument.

dictIsIn

`dictIsIn ('dict_name', child_id, ancestor_id)`

- For the 'dict_name' hierarchical dictionary, finds out whether the 'child_id' key is located inside 'ancestor_id' (or matches 'ancestor_id'). Returns UInt8.

dictGetHierarchy

`dictGetHierarchy('dict_name', id)`

- For the 'dict_name' hierarchical dictionary, returns an array of dictionary keys starting from 'id' and continuing along the chain of parent elements. Returns Array(UInt64).

dictHas

`dictHas('dict_name', id)`

- Check whether the dictionary has the key. Returns a UInt8 value equal to 0 if there is no key and 1 if there is a key.

Functions for working with Yandex.Metrica dictionaries

In order for the functions below to work, the server config must specify the paths and addresses for getting all the Yandex.Metrica dictionaries. The dictionaries are loaded at the first call of any of these functions. If the reference lists can't be loaded, an exception is thrown.

For information about creating reference lists, see the section "Dictionaries".

Multiple geobases

ClickHouse supports working with multiple alternative geobases (regional hierarchies) simultaneously, in order to support various perspectives on which countries certain regions belong to.

The 'clickhouse-server' config specifies the file with the regional hierarchy: `<path_to_regions_hierarchy_file>/opt/geo/regions_hierarchy.txt</path_to_regions_hierarchy_file>`

Besides this file, it also searches for files nearby that have the `_` symbol and any suffix appended to the name (before the file extension).

For example, it will also find the file `/opt/geo/regions_hierarchy_ua.txt`, if present.

`ua` is called the dictionary key. For a dictionary without a suffix, the key is an empty string.

All the dictionaries are re-loaded in runtime (once every certain number of seconds, as defined in the `builtin_dictionaries_reload_interval` config parameter, or once an hour by default). However, the list of available dictionaries is defined one time, when the server starts.

All functions for working with regions have an optional argument at the end - the dictionary key. It is referred to as the geobase.

Example:

```
regionToCountry(RegionID) - Uses the default dictionary: /opt/geo/regions_hierarchy.txt
regionToCountry(RegionID, '') - Uses the default dictionary: /opt/geo/regions_hierarchy.txt
regionToCountry(RegionID, 'ua') - Uses the dictionary for the 'ua' key: /opt/geo/regions_hierarchy_ua.txt
```

regionToCity(id[, geobase])

Accepts a UInt32 number - the region ID from the Yandex geobase. If this region is a city or part of a city, it returns the region ID for the appropriate city. Otherwise, returns 0.

regionToArea(id[, geobase])

Converts a region to an area (type 5 in the geobase). In every other way, this function is the same as 'regionToCity'.

```
SELECT DISTINCT regionToName(regionToArea(toUInt32(number), 'ua'))
FROM system.numbers
LIMIT 15
```

```
┌regionToName(regionToArea(toUInt32(number), \'ua\'))┐
├──────────────────────────────────────────────────┤
│ Moscow and Moscow region                        │
│ St. Petersburg and Leningrad region              │
│ Belgorod region                                │
│ Ivanovsk region                                │
│ Kaluga region                                  │
│ Kostroma region                               │
│ Kursk region                                  │
│ Lipetsk region                                │
│ Orlov region                                  │
│ Ryazan region                                 │
│ Smolensk region                               │
│ Tambov region                                 │
│ Tver region                                   │
│ Tula region                                   │
└──────────────────────────────────────────────────┘
```

regionToDistrict(id[, geobase])

Converts a region to a federal district (type 4 in the geobase). In every other way, this function is the same as 'regionToCity'.

```
SELECT DISTINCT regionToName(regionToDistrict(toUInt32(number), 'ua'))
FROM system.numbers
LIMIT 15
```

regionToName(regionToDistrict(toUInt32(number), \ua\'))		
Central federal district		
Northwest federal district		
South federal district		
North Caucasus federal district		
Privolga federal district		
Ural federal district		
Siberian federal district		
Far East federal district		
Scotland		
Faroe Islands		
Flemish region		
Brussels capital region		
Wallonia		
Federation of Bosnia and Herzegovina		

regionToCountry(id[, geobase])

Converts a region to a country. In every other way, this function is the same as 'regionToCity'.

Example: `regionToCountry(toUInt32(213)) = 225` converts Moscow (213) to Russia (225).

regionToContinent(id[, geobase])

Converts a region to a continent. In every other way, this function is the same as 'regionToCity'.

Example: `regionToContinent(toUInt32(213)) = 10001` converts Moscow (213) to Eurasia (10001).

regionToPopulation(id[, geobase])

Gets the population for a region.

The population can be recorded in files with the geobase. See the section "External dictionaries".

If the population is not recorded for the region, it returns 0.

In the Yandex geobase, the population might be recorded for child regions, but not for parent regions.

regionIn(lhs, rhs[, geobase])

Checks whether a 'lhs' region belongs to a 'rhs' region. Returns a UInt8 number equal to 1 if it belongs, or 0 if it doesn't belong.

The relationship is reflexive – any region also belongs to itself.

regionHierarchy(id[, geobase])

Accepts a UInt32 number – the region ID from the Yandex geobase. Returns an array of region IDs consisting of the passed region and all parents along the chain.

Example: `regionHierarchy(toUInt32(213)) = [213,1,3,225,10001,10000]`.

regionToName(id[, lang])

Accepts a UInt32 number – the region ID from the Yandex geobase. A string with the name of the language can be passed as a second argument. Supported languages are: ru, en, ua, uk, by, kz, tr. If the second argument is omitted, the language 'ru' is used. If the language is not supported, an exception is thrown. Returns a string – the name of the region in the corresponding language. If the region with the specified ID doesn't exist, an empty string is returned.

`ua` and `uk` both mean Ukrainian.

Functions for implementing the IN operator

in, notIn, globalIn, globalNotIn

See the section [IN operators](#).

tuple(x, y, ...), operator (x, y, ...)

A function that allows grouping multiple columns.

For columns with the types T1, T2, ..., it returns a Tuple(T1, T2, ...) type tuple containing these columns. There is no cost to execute the function.

Tuples are normally used as intermediate values for an argument of IN operators, or for creating a list of formal parameters of lambda functions. Tuples can't be written to a table.

tupleElement(tuple, n), operator x.N

A function that allows getting a column from a tuple.

'N' is the column index, starting from 1. N must be a constant. 'N' must be a constant. 'N' must be a strict positive integer no greater than the size of the tuple.

There is no cost to execute the function.

arrayJoin function

This is a very unusual function.

Normal functions don't change a set of rows, but just change the values in each row (map).

Aggregate functions compress a set of rows (fold or reduce).

The 'arrayJoin' function takes each row and generates a set of rows (unfold).

This function takes an array as an argument, and propagates the source row to multiple rows for the number of elements in the array.

All the values in columns are simply copied, except the values in the column where this function is applied; it is replaced with the corresponding array value.

A query can use multiple **arrayJoin** functions. In this case, the transformation is performed multiple times.

Note the ARRAY JOIN syntax in the SELECT query, which provides broader possibilities.

Example:

```
SELECT arrayJoin([1, 2, 3] AS src) AS dst, 'Hello', src
```

dst	'Hello'	src
1	Hello	[1,2,3]
2	Hello	[1,2,3]
3	Hello	[1,2,3]

Functions for working with geographical coordinates

greatCircleDistance

Calculate the distance between two points on the Earth's surface using **the great-circle formula**.

```
greatCircleDistance(lon1Deg, lat1Deg, lon2Deg, lat2Deg)
```

Input parameters

- **lon1Deg** — Longitude of the first point in degrees. Range: [-180°, 180°].
- **lat1Deg** — Latitude of the first point in degrees. Range: [-90°, 90°].
- **lon2Deg** — Longitude of the second point in degrees. Range: [-180°, 180°].
- **lat2Deg** — Latitude of the second point in degrees. Range: [-90°, 90°].

Positive values correspond to North latitude and East longitude, and negative values correspond to South latitude and West longitude.

Returned value

The distance between two points on the Earth's surface, in meters.

Generates an exception when the input parameter values fall outside of the range.

Example

```
SELECT greatCircleDistance(55.755831, 37.617673, -55.755831, -37.617673)
```

```
┌greatCircleDistance(55.755831, 37.617673, -55.755831, -37.617673)┐  
| 14132374.194975413 |
```

pointInEllipses

Checks whether the point belongs to at least one of the ellipses.

```
pointInEllipses(x, y, x0, y0, a0, b0, ..., xn, yn, an, bn)
```

Input parameters

- x, y — Coordinates of a point on the plane.
- x_i, y_i — Coordinates of the center of the i -th ellipsis.
- a_i, b_i — Axes of the i -th ellipsis in meters.

The input parameters must be $2+4\cdot n$, where n is the number of ellipses.

Returned values

1 if the point is inside at least one of the ellipses; **0** if it is not.

Example

```
SELECT pointInEllipses(55.755831, 37.617673, 55.755831, 37.617673, 1.0, 2.0)
```

```
┌pointInEllipses(55.755831, 37.617673, 55.755831, 37.617673, 1., 2.)┐  
| 1 |
```

pointInPolygon

Checks whether the point belongs to the polygon on the plane.

```
pointInPolygon((x, y), [(a, b), (c, d) ...], ...)
```

Input values

- (x, y) — Coordinates of a point on the plane. Data type — **Tuple** — A tuple of two numbers.
- $[(a, b), (c, d) \dots]$ — Polygon vertices. Data type — **Array**. Each vertex is represented by a pair of coordinates (a, b) . Vertices should be specified in a clockwise or counterclockwise order. The minimum number of vertices is 3. The polygon must be constant.
- The function also supports polygons with holes (cut out sections). In this case, add polygons that define the cut out sections using additional arguments of the function. The function does not support non-simply-connected polygons.

Returned values

1 if the point is inside the polygon, **0** if it is not.

If the point is on the polygon boundary, the function may return either 0 or 1.

Example

```
SELECT pointInPolygon((3., 3.), [(6, 0), (8, 4), (5, 8), (0, 2)]) AS res
```

```
┌res┐  
| 1 |
```

geohashEncode

Encodes latitude and longitude as a geohash-string, please see (<http://geohash.org/>, <https://en.wikipedia.org/wiki/Geohash>).

```
geohashEncode(longitude, latitude, [precision])
```

Input values

- longitude - longitude part of the coordinate you want to encode. Floating in range `[-180°, 180°]`
- latitude - latitude part of the coordinate you want to encode. Floating in range `[-90°, 90°]`
- precision - Optional, length of the resulting encoded string, defaults to `12`. Integer in range `[1, 12]`. Any value less than `1` or greater than `12` is silently converted to `12`.

Returned values

- alphanumeric `String` of encoded coordinate (modified version of the base32-encoding alphabet is used).

Example

```
SELECT geohashEncode(-5.60302734375, 42.593994140625, 0) AS res
```

```
res  
| ezs42d000000 |
```

geohashDecode

Decodes any geohash-encoded string into longitude and latitude.

Input values

- encoded string - geohash-encoded string.

Returned values

- (longitude, latitude) - 2-tuple of `Float64` values of longitude and latitude.

Example

```
SELECT geohashDecode('ezs42') AS res
```

```
res  
| (-5.60302734375,42.60498046875) |
```

Functions for working with Nullable aggregates

isNull

Checks whether the argument is `NULL`.

```
isNull(x)
```

Parameters

- `x` — A value with a non-compound data type.

Returned value

- `1` if `x` is `NULL`.
- `0` if `x` is not `NULL`.

Example

Input table

x	y
1	NULL
2	3

Query

```

:) SELECT x FROM t_null WHERE isNull(y)

```

```

SELECT x
FROM t_null
WHERE isNull(y)

```

x
1

1 rows in set. Elapsed: 0.010 sec.

isNotNull

Checks whether the argument is **NULL**.

```
isNotNull(x)
```

Parameters:

- **x** — A value with a non-compound data type.

Returned value

- **0** if **x** is **NULL**.
- **1** if **x** is not **NULL**.

Example

Input table

x	y
1	NULL
2	3

Query

```

:) SELECT x FROM t_null WHERE isNotNull(y)

```

```

SELECT x
FROM t_null
WHERE isNotNull(y)

```

x
2

1 rows in set. Elapsed: 0.010 sec.

coalesce

Checks from left to right whether **NULL** arguments were passed and returns the first non-**NULL** argument.

```
coalesce(x,...)
```

Parameters:

- Any number of parameters of a non-compound type. All parameters must be compatible by data type.

Returned values

- The first non-**NULL** argument.
- NULL**, if all arguments are **NULL**.

Example

Consider a list of contacts that may specify multiple ways to contact a customer.

name	mail	phone	icq
client 1	NULL	123-45-67	123
client 2	NULL	NULL	NULL

The **mail** and **phone** fields are of type String, but the **icq** field is **UInt32**, so it needs to be converted to **String**.

Get the first available contact method for the customer from the contact list:

```
;) SELECT coalesce(mail, phone, CAST(icq,'Nullable(String)')) FROM aBook
```

```
SELECT coalesce(mail, phone, CAST(icq, 'Nullable(String)'))
FROM aBook
```

name	coalesce(mail, phone, CAST(icq, 'Nullable(String)'))
client 1	123-45-67
client 2	NULL

2 rows in set. Elapsed: 0.006 sec.

ifNull

Returns an alternative value if the main argument is **NULL**.

ifNull(x,alt)

Parameters:

- x** — The value to check for **NULL**.
- alt** — The value that the function returns if **x** is **NULL**.

Returned values

- The value **x**, if **x** is not **NULL**.
- The value **alt**, if **x** is **NULL**.

Example

```
SELECT ifNull('a', 'b')
```

ifNull('a', 'b')
a

```
SELECT ifNull(NULL, 'b')
```

ifNull(NULL, 'b')
b

nullIf

Returns **NULL** if the arguments are equal.

```
nullIf(x, y)
```

Parameters:

x, y — Values for comparison. They must be compatible types, or ClickHouse will generate an exception.

Returned values

- **NULL**, if the arguments are equal.
- The **x** value, if the arguments are not equal.

Example

```
SELECT nullIf(1, 1)
```

nullIf(1, 1)
NULL

```
SELECT nullIf(1, 2)
```

nullIf(1, 2)
1

assumeNotNull

Results in a value of type **Nullable** for a non- **Nullable**, if the value is not **NULL**.

```
assumeNotNull(x)
```

Parameters:

- **x** — The original value.

Returned values

- The original value from the non-**Nullable** type, if it is not **NULL**.
- The default value for the non-**Nullable** type if the original value was **NULL**.

Example

Consider the **t_null** table.

```
SHOW CREATE TABLE t_null
```

statement
CREATE TABLE default.t_null (x Int8, y Nullable(Int8)) ENGINE = TinyLog

x	y
1	NULL
2	3

Apply the **resumenotnull** function to the **y** column.

```
SELECT assumeNotNull(y) FROM t_null
```

assumeNotNull(y)
0
3

```
SELECT toTypeName(assumeNotNull(y)) FROM t_null
```

```
┌toTypeName(assumeNotNull(y))┐
├ Int8                        │
├ Int8                        │
└──────────────────────────┘
```

toNullable

Converts the argument type to **Nullable**.

```
toNullable(x)
```

Parameters:

- **x** — The value of any non-compound type.

Returned value

- The input value with a non-**Nullable** type.

Example

```
SELECT toTypeName(10)
```

```
┌toTypeName(10)┐
├ UInt8        │
└──────────┘
```

```
SELECT toTypeName(toNullable(10))
```

```
┌toTypeName(toNullable(10))┐
├ Nullable(UInt8)          │
└────────────────────────┘
```

Other functions

hostName()

Returns a string with the name of the host that this function was performed on. For distributed processing, this is the name of the remote server host, if the function is performed on a remote server.

basename

Extracts trailing part of a string after the last slash or backslash. This function is often used to extract the filename from the path.

```
basename( expr )
```

Parameters

- **expr** — Expression, resulting in the **String**-type value. All the backslashes must be escaped in the resulting value.

Returned Value

A **String**-type value that contains:

- Trailing part of a string after the last slash or backslash in it.

If the input string contains a path, ending with slash or backslash, for example, **/** or **c:**, the function returns an empty string.

- Original string if there are no slashes or backslashes in it.

Example

```
SELECT 'some/long/path/to/file' AS a, basename(a)
```

```
└─a┐ ┌─basename('some\\long\\path\\to\\file')┐
  some\\long\\path\\to\\file | file           |
```

```
SELECT 'some\\long\\path\\to\\file' AS a, basename(a)
```

a	basename('some\\long\\path\\to\\file')
some\\long\\path\\to\\file file	

```
SELECT 'some-file-name' AS a, basename(a)
```

```
└─a────────┐basename('some-file-name')┐
└some-file-name┤some-file-name┤
```

visibleWidth(x)

Calculates the approximate width when outputting values to the console in text format (tab-separated).

This function is used by the system for implementing Pretty formats.

`NULL` is represented as a string corresponding to `NULL` in `Pretty` formats.

```
SELECT visibleWidth(NULL)
```

```
└─visibleWidth(NULL)─┘
|           4 |
|
```

toTypeName(x)

Returns a string containing the type name of the passed argument.

If **NULL** is passed to the function as input, then it returns the **Nullable(Nothing)** type, which corresponds to an internal **NULL** representation in ClickHouse.

blockSize()

Gets the size of the block.

In ClickHouse, queries are always run on blocks (sets of column parts). This function allows getting the size of the block that you called it for.

materialize(x)

Turns a constant into a full column containing just one value.

In ClickHouse, full columns and constants are represented differently in memory. Functions work differently for constant arguments and normal arguments (different code is executed), although the result is almost always the same. This function is for debugging this behavior.

```
ignore(...)
```

Accepts any arguments, including **NULL**. Always returns 0.

However, the argument is still evaluated. This can be used for benchmarks.

sleep(seconds)

Sleeps 'seconds' seconds on each data block. You can specify an integer or a floating-point number.

sleepEachRow(seconds)

Sleeps 'seconds' seconds on each row. You can specify an integer or a floating-point number.

currentDatabase()

Returns the name of the current database.

You can use this function in table engine parameters in a CREATE TABLE query where you need to specify the database.

isFinite(x)

Accepts Float32 and Float64 and returns UInt8 equal to 1 if the argument is not infinite and not a NaN, otherwise 0.

isInfinite(x)

Accepts Float32 and Float64 and returns UInt8 equal to 1 if the argument is infinite, otherwise 0. Note that 0 is returned for a NaN.

isNaN(x)

Accepts Float32 and Float64 and returns UInt8 equal to 1 if the argument is a NaN, otherwise 0.

hasColumnInTable(['hostname'[, 'username'[, 'password']], 'database', 'table', 'column')

Accepts constant strings: database name, table name, and column name. Returns a UInt8 constant expression equal to 1 if there is a column, otherwise 0. If the hostname parameter is set, the test will run on a remote server. The function throws an exception if the table does not exist.

For elements in a nested data structure, the function checks for the existence of a column. For the nested data structure itself, the function returns 0.

bar

Allows building a unicode-art diagram.

bar(x, min, max, width) draws a band with a width proportional to (x - min) and equal to width characters when x = max.

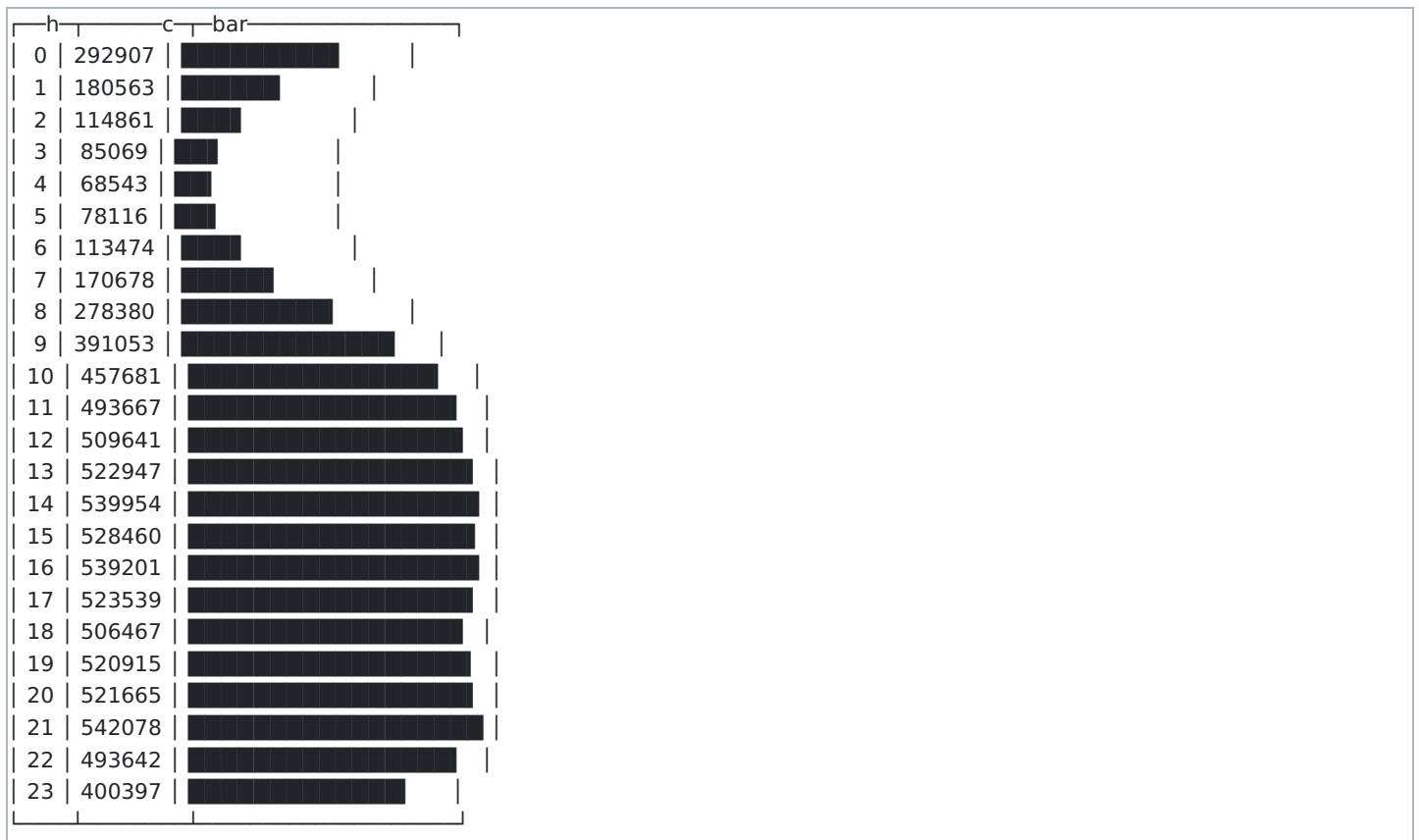
Parameters:

- **x** — Size to display.
- **min, max** — Integer constants. The value must fit in **Int64**.
- **width** — Constant, positive integer, can be fractional.

The band is drawn with accuracy to one eighth of a symbol.

Example:

```
SELECT
  toHour(EventTime) AS h,
  count() AS c,
  bar(c, 0, 600000, 20) AS bar
FROM test.hits
GROUP BY h
ORDER BY h ASC
```



transform

Transforms a value according to the explicitly defined mapping of some elements to other ones. There are two variations of this function:

- 1. `transform(x, array_from, array_to, default)`

`x` - What to transform.

`array_from` - Constant array of values for converting.

`array_to` - Constant array of values to convert the values in 'from' to.

`default` - Which value to use if 'x' is not equal to any of the values in 'from'.

`array_from` and `array_to` - Arrays of the same size.

Types:

`transform(T, Array(T), Array(U), U) -> U`

`T` and `U` can be numeric, string, or Date or DateTime types.

Where the same letter is indicated (T or U), for numeric types these might not be matching types, but types that have a common type.

For example, the first argument can have the Int64 type, while the second has the Array(UInt16) type.

If the 'x' value is equal to one of the elements in the 'array_from' array, it returns the existing element (that is numbered the same) from the 'array_to' array. Otherwise, it returns 'default'. If there are multiple matching elements in 'array_from', it returns one of the matches.

Example:


```
SELECT
  transform(SearchEngineID, [2, 3], ['Yandex', 'Google'], 'Other') AS title,
  count() AS c
FROM test.hits
WHERE SearchEngineID != 0
GROUP BY title
ORDER BY c DESC
```

title	c
Yandex	498635
Google	229872
Other	104472

2. `transform(x, array_from, array_to)`

Differs from the first variation in that the 'default' argument is omitted.

If the 'x' value is equal to one of the elements in the 'array_from' array, it returns the matching element (that is numbered the same) from the 'array_to' array. Otherwise, it returns 'x'.

Types:

`transform(T, Array(T), Array(T)) -> T`

Example:

```
SELECT
  transform(domain(Referer), ['yandex.ru', 'google.ru', 'vk.com'], ['www.yandex', 'example.com']) AS s,
  count() AS c
FROM test.hits
GROUP BY domain(Referer)
ORDER BY count() DESC
LIMIT 10
```

s	c
	2906259
www.yandex	867767
██████.ru	313599
mail.yandex.ru	107147
██████.ru	100355
██████.ru	65040
news.yandex.ru	64515
██████.net	59141
example.com	57316

`formatReadableSize(x)`

Accepts the size (number of bytes). Returns a rounded size with a suffix (KiB, MiB, etc.) as a string.

Example:

```
SELECT
  arrayJoin([1, 1024, 1024*1024, 192851925]) AS filesize_bytes,
  formatReadableSize(filesize_bytes) AS filesize
```

filesize_bytes	filesize
1	1.00 B
1024	1.00 KiB
1048576	1.00 MiB
192851925	183.92 MiB

`least(a, b)`

Returns the smallest value from a and b.

greatest(a, b)

Returns the largest value of a and b.

uptime()

Returns the server's uptime in seconds.

version()

Returns the version of the server as a string.

timezone()

Returns the timezone of the server.

blockNumber

Returns the sequence number of the data block where the row is located.

rowNumberInBlock

Returns the ordinal number of the row in the data block. Different data blocks are always recalculated.

rowNumberInAllBlocks()

Returns the ordinal number of the row in the data block. This function only considers the affected data blocks.

runningDifference(x)

Calculates the difference between successive row values in the data block.

Returns 0 for the first row and the difference from the previous row for each subsequent row.

The result of the function depends on the affected data blocks and the order of data in the block.

If you make a subquery with ORDER BY and call the function from outside the subquery, you can get the expected result.

Example:

```
SELECT
  EventID,
  EventTime,
  runningDifference(EventTime) AS delta
FROM
(
  SELECT
    EventID,
    EventTime
  FROM events
  WHERE EventDate = '2016-11-24'
  ORDER BY EventTime ASC
  LIMIT 5
)
```

EventID	EventTime	delta
1106	2016-11-24 00:00:04	0
1107	2016-11-24 00:00:05	1
1108	2016-11-24 00:00:05	0
1109	2016-11-24 00:00:09	4
1110	2016-11-24 00:00:10	1

runningDifferenceStartingWithFirstValue

Same as for **runningDifference**, the difference is the value of the first row, returned the value of the first row, and each subsequent row returns the difference from the previous row.

MACNumToString(num)

Accepts a UInt64 number. Interprets it as a MAC address in big endian. Returns a string containing the corresponding MAC address in the format AA:BB:CC:DD:EE:FF (colon-separated numbers in hexadecimal form).

MACStringToNum(s)

The inverse function of MACNumToString. If the MAC address has an invalid format, it returns 0.

MACStringToOUI(s)

Accepts a MAC address in the format AA:BB:CC:DD:EE:FF (colon-separated numbers in hexadecimal form). Returns the first three octets as a UInt64 number. If the MAC address has an invalid format, it returns 0.

getsizeofEnumType

Returns the number of fields in **Enum**.

```
getsizeofEnumType(value)
```

Parameters:

- **value** — Value of type **Enum**.

Returned values

- The number of fields with **Enum** input values.
- An exception is thrown if the type is not **Enum**.

Example

SELECT sizeofEnumType(CAST('a' AS Enum8('a' = 1, 'b' = 2))) AS x

X

2

toColumnName

Returns the name of the class that represents the data type of the column in RAM.

```
toColumnName(value)
```

Parameters:

- **value** — Any type of value.

Returned values

- A string with the name of the class that is used for representing the **value** data type in RAM.

Example of the difference between **toTypeName ' and ' **toColumnName****

```

:) select toTypeName(cast('2018-01-01 01:02:03' AS DateTime))

SELECT toTypeName(CAST('2018-01-01 01:02:03', 'DateTime'))

┌toTypeName(CAST('2018-01-01 01:02:03', 'DateTime'))┐
│ DateTime                                         │
└───────────────────────────────────────────────────┘

1 rows in set. Elapsed: 0.008 sec.

:) select toColumnName(cast('2018-01-01 01:02:03' AS DateTime))

SELECT toColumnName(CAST('2018-01-01 01:02:03', 'DateTime'))

┌toColumnName(CAST('2018-01-01 01:02:03', 'DateTime'))┐
│ Const(UInt32)                                     │
└───────────────────────────────────────────────────┘

```

The example shows that the `DateTime` data type is stored in memory as `Const(UInt32)`.

dumpColumnStructure

Outputs a detailed description of data structures in RAM

```
dumpColumnStructure(value)
```

Parameters:

- `value` — Any type of value.

Returned values

- A string describing the structure that is used for representing the `value` data type in RAM.

Example

```

SELECT dumpColumnStructure(CAST('2018-01-01 01:02:03', 'DateTime'))

┌dumpColumnStructure(CAST('2018-01-01 01:02:03', 'DateTime'))┐
│ DateTime, Const(size = 1, UInt32(size = 1))                │
└───────────────────────────────────────────────────────────┘

```

defaultValueOfArgumentType

Outputs the default value for the data type.

Does not include default values for custom columns set by the user.

```
defaultValueOfArgumentType(expression)
```

Parameters:

- `expression` — Arbitrary type of value or an expression that results in a value of an arbitrary type.

Returned values

- `0` for numbers.
- Empty string for strings.
- `NULL` for `Nullable`.

Example

```

:) SELECT defaultValueType( CAST(1 AS Int8) )

SELECT defaultValueType(CAST(1, 'Int8'))

┌defaultValueType(CAST(1, 'Int8'))┐
│                                0 │
└──────────────────────────────────┘

1 rows in set. Elapsed: 0.002 sec.

:) SELECT defaultValueType( CAST(1 AS Nullable(Int8) ) )

SELECT defaultValueType(CAST(1, 'Nullable(Int8)'))

┌defaultValueType(CAST(1, 'Nullable(Int8)'))┐
│                                NULL │
└──────────────────────────────────────────┘

1 rows in set. Elapsed: 0.002 sec.

```

indexHint

Outputs data in the range selected by the index without filtering by the expression specified as an argument.

The expression passed to the function is not calculated, but ClickHouse applies the index to this expression in the same way as if the expression was in the query without `indexHint`.

Returned value

- 1.

Example

Here is a table with the test data for `ontime`.

```

SELECT count() FROM ontime

┌count()┐
│ 4276457 │
└────────┘

```

The table has indexes for the fields (`FlightDate`, (`Year`, `FlightDate`)).

Create a selection by date like this:

```

:) SELECT FlightDate AS k, count() FROM ontime GROUP BY k ORDER BY k

SELECT
  FlightDate AS k,
  count()
FROM ontime
GROUP BY k
ORDER BY k ASC

┌k┐count()┐
├──┴──┤
│ 2017-01-01 │ 13970 │
│ 2017-01-02 │ 15882 │
├──┴──┤
.....
│ 2017-09-28 │ 16411 │
│ 2017-09-29 │ 16384 │
│ 2017-09-30 │ 12520 │
├──┴──┤

273 rows in set. Elapsed: 0.072 sec. Processed 4.28 million rows, 8.55 MB (59.00 million rows/s., 118.01 MB/s.)

```

In this selection, the index is not used and ClickHouse processed the entire table (**Processed 4.28 million rows**). To apply the index, select a specific date and run the following query:

```
:) SELECT FlightDate AS k, count() FROM ontime WHERE k = '2017-09-15' GROUP BY k ORDER BY k
```

```
SELECT
  FlightDate AS k,
  count()
FROM ontime
WHERE k = '2017-09-15'
GROUP BY k
ORDER BY k ASC
```

k	count()
2017-09-15	16428

1 rows in set. Elapsed: 0.014 sec. Processed 32.74 thousand rows, 65.49 KB (2.31 million rows/s., 4.63 MB/s.)

The last line of output shows that by using the index, ClickHouse processed a significantly smaller number of rows (**Processed 32.74 thousand rows**).

Now pass the expression `k = '2017-09-15'` to the `indexHint` function:

```
:) SELECT FlightDate AS k, count() FROM ontime WHERE indexHint(k = '2017-09-15') GROUP BY k ORDER BY k
```

```
SELECT
  FlightDate AS k,
  count()
FROM ontime
WHERE indexHint(k = '2017-09-15')
GROUP BY k
ORDER BY k ASC
```

k	count()
2017-09-14	7071
2017-09-15	16428
2017-09-16	1077
2017-09-30	8167

4 rows in set. Elapsed: 0.004 sec. Processed 32.74 thousand rows, 65.49 KB (8.97 million rows/s., 17.94 MB/s.)

The response to the request shows that ClickHouse applied the index in the same way as the previous time (**Processed 32.74 thousand rows**). However, the resulting set of rows shows that the expression `k = '2017-09-15'` was not used when generating the result.

Because the index is sparse in ClickHouse, "extra" data ends up in the response when reading a range (in this case, the adjacent dates). Use the `indexHint` function to see it.

replicate

Creates an array with a single value.

Used for internal implementation of `arrayJoin`.

```
replicate(x, arr)
```

Parameters:

- **arr** — Original array. ClickHouse creates a new array of the same length as the original and fills it with the value **x**.
- **x** — The value that the resulting array will be filled with.

Output value

- An array filled with the value **x**.

Example

```
SELECT replicate(1, ['a', 'b', 'c'])
```

```
└─replicate(1, ['a', 'b', 'c'])─┐  
| [1,1,1]                      |  
└────────────────────────────────┘
```

filesystemAvailable

Returns the remaining space information of the disk, in bytes. This information is evaluated using the configured by path.

filesystemCapacity

Returns the capacity information of the disk, in bytes. This information is evaluated using the configured by path.

finalizeAggregation

Takes state of aggregate function. Returns result of aggregation (finalized state).

runningAccumulate

Takes the states of the aggregate function and returns a column with values, are the result of the accumulation of these states for a set of block lines, from the first to the current line.

For example, takes state of aggregate function (example `runningAccumulate(uniqState(UserID))`), and for each row of block, return result of aggregate function on merge of states of all previous rows and current row. So, result of function depends on partition of data to blocks and on order of data in block.

joinGet('join_storage_table_name', 'get_column', join_key)

Get data from a table of type Join using the specified join key.

modelEvaluate(model_name, ...)

Evaluate external model.

Accepts a model name and model arguments. Returns Float64.

throwIf(x)

Throw an exception if the argument is non zero.

Aggregate functions

Aggregate functions work in the **normal** way as expected by database experts.

ClickHouse also supports:

- **Parametric aggregate functions**, which accept other parameters in addition to columns.
- **Combinators**, which change the behavior of aggregate functions.

NULL processing

During aggregation, all **NULLs** are skipped.

Examples:

Consider this table:

x	y
1	2
2	NULL
3	2
3	3
3	NULL

Let's say you need to total the values in the **y** column:

```
:) SELECT sum(y) FROM t_null_big
```

```
SELECT sum(y)
FROM t_null_big
```

sum(y)
7

1 rows in set. Elapsed: 0.002 sec.

The **sum** function interprets **NULL** as **0**. In particular, this means that if the function receives input of a selection where all the values are **NULL**, then the result will be **0**, not **NULL**.

Now you can use the **groupArray** function to create an array from the **y** column:

```
:) SELECT groupArray(y) FROM t_null_big
```

```
SELECT groupArray(y)
FROM t_null_big
```

groupArray(y)
[2,2,3]

1 rows in set. Elapsed: 0.002 sec.

groupArray does not include **NULL** in the resulting array.

Function reference

count()

Counts the number of rows. Accepts zero arguments and returns UInt64.

The syntax **COUNT(DISTINCT x)** is not supported. The separate **uniq** aggregate function exists for this purpose.

A **SELECT count() FROM table** query is not optimized, because the number of entries in the table is not stored separately. It will select some small column from the table and count the number of values in it.

any(x)

Selects the first encountered value.

The query can be executed in any order and even in a different order each time, so the result of this function is indeterminate.

To get a determinate result, you can use the 'min' or 'max' function instead of 'any'.

In some cases, you can rely on the order of execution. This applies to cases when **SELECT** comes from a subquery that uses **ORDER BY**.

When a **SELECT** query has the **GROUP BY** clause or at least one aggregate function, ClickHouse (in contrast to MySQL) requires that all expressions in the **SELECT**, **HAVING**, and **ORDER BY** clauses be calculated from keys or from aggregate functions. In other words, each column selected from the table must be used either in keys or inside aggregate functions. To get behavior like in MySQL, you can put the other columns in the **any** aggregate function.

anyHeavy(x)

Selects a frequently occurring value using the **heavy hitters** algorithm. If there is a value that occurs more than in half the cases in each of the query's execution threads, this value is returned. Normally, the result is nondeterministic.

```
anyHeavy(column)
```

Arguments

- **column** – The column name.

Example

Take the **OnTime** data set and select any frequently occurring value in the **AirlineID** column.

```
SELECT anyHeavy(AirlineID) AS res
FROM ontime
```

```
┌ res ─┐
│ 19690 │
```

anyLast(x)

Selects the last value encountered.

The result is just as indeterminate as for the **any** function.

groupBitAnd

Applies bitwise **AND** for series of numbers.

```
groupBitAnd(expr)
```

Parameters

expr – An expression that results in **UInt*** type.

Return value

Value of the **UInt*** type.

Example

Test data:

```
binary  decimal
00101100 = 44
00011100 = 28
00001101 = 13
01010101 = 85
```

Query:

```
SELECT groupBitAnd(num) FROM t
```

Where **num** is the column with the test data.

Result:

```
binary  decimal
00000100 = 4
```

groupBitOr

Applies bitwise **OR** for series of numbers.

```
groupBitOr(expr)
```

Parameters

expr – An expression that results in **UInt*** type.

Return value

Value of the **UInt*** type.

Example

Test data:

```
binary  decimal
00101100 = 44
00011100 = 28
00001101 = 13
01010101 = 85
```

Query:

```
SELECT groupBitOr(num) FROM t
```

Where **num** is the column with the test data.

Result:

```
binary  decimal
01111101 = 125
```

groupBitXor

Applies bitwise **XOR** for series of numbers.

```
groupBitXor(expr)
```

Parameters

expr – An expression that results in **UInt*** type.

Return value

Value of the **UInt*** type.

Example

Test data:

```
binary  decimal
00101100 = 44
00011100 = 28
00001101 = 13
01010101 = 85
```

Query:

```
SELECT groupBitXor(num) FROM t
```

Where **num** is the column with the test data.

Result:

```
binary  decimal
01101000 = 104
```

groupBitmap

Bitmap or Aggregate calculations from a unsigned integer column, return cardinality of type UInt64, if add suffix - State, then return **bitmap object**.

```
groupBitmap(expr)
```

Parameters

expr – An expression that results in **UInt*** type.

Return value

Value of the **UInt64** type.

Example

Test data:

```
userid
1
1
2
3
```

Query:

```
SELECT groupBitmap(userid) as num FROM t
```

Result:

```
num
3
```

min(x)

Calculates the minimum.

max(x)

Calculates the maximum.

argMin(arg, val)

Calculates the 'arg' value for a minimal 'val' value. If there are several different values of 'arg' for minimal values of 'val', the first of these values encountered is output.

Example:

```
┌──user──┐┌──salary┐
│director│ 5000 │
│manager │ 3000 │
│worker  │ 1000 │
└────────┘└────────┘
```

```
SELECT argMin(user, salary) FROM salary
```

```
┌──argMin(user, salary)──┐
│worker                  │
└────────────────────────┘
```

argMax(arg, val)

Calculates the 'arg' value for a maximum 'val' value. If there are several different values of 'arg' for maximum values of 'val', the first of these values encountered is output.

sum(x)

Calculates the sum.

Only works for numbers.

sumWithOverflow(x)

Computes the sum of the numbers, using the same data type for the result as for the input parameters. If the sum exceeds the maximum value for this data type, the function returns an error.

Only works for numbers.

sumMap(key, value)

Totals the 'value' array according to the keys specified in the 'key' array.

The number of elements in 'key' and 'value' must be the same for each row that is totaled.

Returns a tuple of two arrays: keys in sorted order, and values summed for the corresponding keys.

Example:

```
CREATE TABLE sum_map(
  date Date,
  timeslot DateTime,
  statusMap Nested(
    status UInt16,
    requests UInt64
  )
) ENGINE = Log;
INSERT INTO sum_map VALUES
  ('2000-01-01', '2000-01-01 00:00:00', [1, 2, 3], [10, 10, 10]),
  ('2000-01-01', '2000-01-01 00:00:00', [3, 4, 5], [10, 10, 10]),
  ('2000-01-01', '2000-01-01 00:01:00', [4, 5, 6], [10, 10, 10]),
  ('2000-01-01', '2000-01-01 00:01:00', [6, 7, 8], [10, 10, 10]);
SELECT
  timeslot,
  sumMap(statusMap.status, statusMap.requests)
FROM sum_map
GROUP BY timeslot
```

timeslot	sumMap(statusMap.status, statusMap.requests)
2000-01-01 00:00:00	[(1,2,3,4,5),[10,10,20,10,10]]
2000-01-01 00:01:00	[(4,5,6,7,8),[10,10,20,10,10]]

timeSeriesGroupSum(uid,timestamp, value)

timeSeriesGroupSum can aggregate different time series that sample timestamp not alignment.

It will use linear interpolation between two sample timestamp and then sum time-series together.

uid is the time series unique id, UInt64.

timestamp is Int64 type in order to support millisecond or microsecond.

value is the metric.

Before use this function, timestamp should be in ascend order

Example:

uid	timestamp	value
1	2	0.2
1	7	0.7
1	12	1.2
1	17	1.7
1	25	2.5
2	3	0.6
2	8	1.6
2	12	2.4
2	18	3.6
2	24	4.8

```
CREATE TABLE time_series(
  uid      UInt64,
  timestamp Int64,
  value    Float64
) ENGINE = Memory;
INSERT INTO time_series VALUES
  (1,2,0.2),(1,7,0.7),(1,12,1.2),(1,17,1.7),(1,25,2.5),
  (2,3,0.6),(2,8,1.6),(2,12,2.4),(2,18,3.6),(2,24,4.8);

SELECT timeSeriesGroupSum(uid, timestamp, value)
FROM (
  SELECT * FROM time_series order by timestamp ASC
);
```

And the result will be:

```
[(2,0.2),(3,0.9),(7,2.1),(8,2.4),(12,3.6),(17,5.1),(18,5.4),(24,7.2),(25,2.5)]
```

timeSeriesGroupRateSum(uid, ts, val)

Similarly timeSeriesGroupRateSum, timeSeriesGroupRateSum will Calculate the rate of time-series and then sum rates together.

Also, timestamp should be in ascend order before use this function.

Use this function, the result above case will be:

```
[(2,0),(3,0.1),(7,0.3),(8,0.3),(12,0.3),(17,0.3),(18,0.3),(24,0.3),(25,0.1)]
```

avg(x)

Calculates the average.

Only works for numbers.

The result is always Float64.

uniq(x)

Calculates the approximate number of different values of the argument. Works for numbers, strings, dates, date-with-time, and for multiple arguments and tuple arguments.

Uses an adaptive sampling algorithm: for the calculation state, it uses a sample of element hash values with a size up to 65536.

This algorithm is also very accurate for data sets with low cardinality (up to 65536) and very efficient on CPU (when computing not too many of these functions, using **uniq** is almost as fast as using other aggregate functions).

The result is determinate (it doesn't depend on the order of query processing).

This function provides excellent accuracy even for data sets with extremely high cardinality (over 10 billion elements). It is recommended for default use.

uniqCombined(HLL_precision)(x)

Calculates the approximate number of different values of the argument. Works for numbers, strings, dates, date-with-time, and for multiple arguments and tuple arguments.

A combination of three algorithms is used: array, hash table and **HyperLogLog** with an error correction table. For small number of distinct elements, the array is used. When the set size becomes larger the hash table is used, while it is smaller than HyperLogLog data structure. For larger number of elements, the HyperLogLog is used, and it will occupy fixed amount of memory.

The parameter "HLL_precision" is the base-2 logarithm of the number of cells in HyperLogLog. You can omit the parameter (omit first parens). The default value is 17, that is effectively 96 KiB of space (2^{17} cells of 6 bits each). The memory consumption is several times smaller than for the **uniq** function, and the accuracy is several times higher. Performance is slightly lower than for the **uniq** function, but sometimes it can be even higher than it, such as with distributed queries that transmit a large number of aggregation states over the network.

The result is deterministic (it doesn't depend on the order of query processing).

The **uniqCombined** function is a good default choice for calculating the number of different values, but keep in mind that the estimation error for large sets (200 million elements and more) will become larger than theoretical value due to poor choice of hash function.

uniqHLL12(x)

Uses the **HyperLogLog** algorithm to approximate the number of different values of the argument. 212 5-bit cells are used. The size of the state is slightly more than 2.5 KB. The result is not very accurate (up to ~10% error) for small data sets (<10K elements). However, the result is fairly accurate for high-cardinality data sets (10K-100M), with a maximum error of ~1.6%. Starting from 100M, the estimation error increases, and the function will return very inaccurate results for data sets with extremely high cardinality (1B+ elements).

The result is determinate (it doesn't depend on the order of query processing).

We don't recommend using this function. In most cases, use the **uniq** or **uniqCombined** function.

uniqExact(x)

Calculates the number of different values of the argument, exactly. There is no reason to fear approximations. It's better to use the **uniq** function. Use the **uniqExact** function if you definitely need an exact result.

The **uniqExact** function uses more memory than the **uniq** function, because the size of the state has unbounded growth as the number of different values increases.

groupArray(x), groupArray(max_size)(x)

Creates an array of argument values. Values can be added to the array in any (indeterminate) order.

The second version (with the **max_size** parameter) limits the size of the resulting array to **max_size** elements. For example, **groupArray(1)(x)** is equivalent to **[any(x)]**.

In some cases, you can still rely on the order of execution. This applies to cases when **SELECT** comes from a subquery that uses **ORDER BY**.

groupArrayInsertAt(x)

Inserts a value into the array in the specified position.

Accepts the value and position as input. If several values are inserted into the same position, any of them might end up in the resulting array (the first one will be used in the case of single-threaded execution). If no value is inserted into a position, the position is assigned the default value.

Optional parameters:

- The default value for substituting in empty positions.
- The length of the resulting array. This allows you to receive arrays of the same size for all the aggregate keys. When using this parameter, the default value must be specified.

groupUniqArray(x), groupUniqArray(max_size)(x)

Creates an array from different argument values. Memory consumption is the same as for the [uniqExact](#) function.

The second version (with the [max_size](#) parameter) limits the size of the resulting array to [max_size](#) elements. For example, [groupUniqArray\(1\)\(x\)](#) is equivalent to [\[any\(x\)\]](#).

quantile(level)(x)

Approximates the [level](#) quantile. [level](#) is a constant, a floating-point number from 0 to 1.

We recommend using a [level](#) value in the range of [\[0.01, 0.99\]](#)

Don't use a [level](#) value equal to 0 or 1 – use the [min](#) and [max](#) functions for these cases.

In this function, as well as in all functions for calculating quantiles, the [level](#) parameter can be omitted. In this case, it is assumed to be equal to 0.5 (in other words, the function will calculate the median).

Works for numbers, dates, and dates with times.

Returns: for numbers – [Float64](#); for dates – a date; for dates with times – a date with time.

Uses [reservoir sampling](#) with a reservoir size up to 8192.

If necessary, the result is output with linear approximation from the two neighboring values.

This algorithm provides very low accuracy. See also: [quantileTiming](#), [quantileTDigest](#), [quantileExact](#).

The result depends on the order of running the query, and is nondeterministic.

When using multiple [quantile](#) (and similar) functions with different levels in a query, the internal states are not combined (that is, the query works less efficiently than it could). In this case, use the [quantiles](#) (and similar) functions.

quantileDeterministic(level)(x, determinator)

Works the same way as the [quantile](#) function, but the result is deterministic and does not depend on the order of query execution.

To achieve this, the function takes a second argument – the "determinator". This is a number whose hash is used instead of a random number generator in the reservoir sampling algorithm. For the function to work correctly, the same determinator value should not occur too often. For the determinator, you can use an event ID, user ID, and so on.

Don't use this function for calculating timings. There is a more suitable function for this purpose: [quantileTiming](#).

quantileTiming(level)(x)

Computes the quantile of 'level' with a fixed precision.

Works for numbers. Intended for calculating quantiles of page loading time in milliseconds.

If the value is greater than 30,000 (a page loading time of more than 30 seconds), the result is equated to 30,000.

If the total value is not more than about 5670, then the calculation is accurate.

Otherwise:

- if the time is less than 1024 ms, then the calculation is accurate.
- otherwise the calculation is rounded to a multiple of 16 ms.

When passing negative values to the function, the behavior is undefined.

The returned value has the Float32 type. If no values were passed to the function (when using `quantileTimingIf`), 'nan' is returned. The purpose of this is to differentiate these instances from zeros. See the note on sorting NaNs in "ORDER BY clause".

The result is determinate (it doesn't depend on the order of query processing).

For its purpose (calculating quantiles of page loading times), using this function is more effective and the result is more accurate than for the `quantile` function.

quantileTimingWeighted(level)(x, weight)

Differs from the `quantileTiming` function in that it has a second argument, "weights". Weight is a non-negative integer.

The result is calculated as if the `x` value were passed `weight` number of times to the `quantileTiming` function.

quantileExact(level)(x)

Computes the quantile of 'level' exactly. To do this, all the passed values are combined into an array, which is then partially sorted. Therefore, the function consumes $O(n)$ memory, where 'n' is the number of values that were passed. However, for a small number of values, the function is very effective.

quantileExactWeighted(level)(x, weight)

Computes the quantile of 'level' exactly. In addition, each value is counted with its weight, as if it is present 'weight' times. The arguments of the function can be considered as histograms, where the value 'x' corresponds to a histogram "column" of the height 'weight', and the function itself can be considered as a summation of histograms.

A hash table is used as the algorithm. Because of this, if the passed values are frequently repeated, the function consumes less RAM than `quantileExact`. You can use this function instead of `quantileExact` and specify the weight as 1.

quantileTDigest(level)(x)

Approximates the quantile level using the `t-digest` algorithm. The maximum error is 1%. Memory consumption by State is proportional to the logarithm of the number of passed values.

The performance of the function is lower than for `quantile` or `quantileTiming`. In terms of the ratio of State size to precision, this function is much better than `quantile`.

The result depends on the order of running the query, and is nondeterministic.

median(x)

All the quantile functions have corresponding median functions: `median`, `medianDeterministic`, `medianTiming`, `medianTimingWeighted`, `medianExact`, `medianExactWeighted`, `medianTDigest`. They are synonyms and their behavior is identical.

quantiles(level1, level2, ...)(x)

All the quantile functions also have corresponding quantiles functions: `quantiles`, `quantilesDeterministic`, `quantilesTiming`, `quantilesTimingWeighted`, `quantilesExact`, `quantilesExactWeighted`, `quantilesTDigest`. These functions calculate all the quantiles of the listed levels in one pass, and return an array of the resulting values.

varSamp(x)

Calculates the amount $\Sigma((x - \bar{x})^2) / (n - 1)$, where `n` is the sample size and \bar{x} is the average value of `x`.

It represents an unbiased estimate of the variance of a random variable, if the values passed to the function are a sample of this random amount.

Returns `Float64`. When `n <= 1`, returns $+\infty$.

varPop(x)

Calculates the amount $\Sigma((x - \bar{x})^2) / n$, where n is the sample size and \bar{x} is the average value of x .

In other words, dispersion for a set of values. Returns **Float64**.

stddevSamp(x)

The result is equal to the square root of **varSamp(x)**.

stddevPop(x)

The result is equal to the square root of **varPop(x)**.

topK(N)(column)

Returns an array of the most frequent values in the specified column. The resulting array is sorted in descending order of frequency of values (not by the values themselves).

Implements the **Filtered Space-Saving** algorithm for analyzing TopK, based on the reduce-and-combine algorithm from **Parallel Space Saving**.

```
topK(N)(column)
```

This function doesn't provide a guaranteed result. In certain situations, errors might occur and it might return frequent values that aren't the most frequent values.

We recommend using the $N < 10$ value; performance is reduced with large N values. Maximum value of $N = 65536$.

Arguments

- 'N' is the number of values.
- 'x' – The column.

Example

Take the **OnTime** data set and select the three most frequently occurring values in the **AirlineID** column.

```
SELECT topK(3)(AirlineID) AS res
FROM ontime
```

```
┌res┐
└───┘
| [19393,19790,19805] |
└───┘
```

covarSamp(x, y)

Calculates the value of $\Sigma((x - \bar{x})(y - \bar{y})) / (n - 1)$.

Returns **Float64**. When $n \leq 1$, returns $+\infty$.

covarPop(x, y)

Calculates the value of $\Sigma((x - \bar{x})(y - \bar{y})) / n$.

corr(x, y)

Calculates the Pearson correlation coefficient: $\Sigma((x - \bar{x})(y - \bar{y})) / \sqrt{\Sigma((x - \bar{x})^2) * \Sigma((y - \bar{y})^2)}$

simpleLinearRegression

Performs simple (unidimensional) linear regression.

```
simpleLinearRegression(x, y)
```

Parameters:

- **x** — Column with values of dependent variable.
- **y** — Column with explanatory variable.

Returned values:

Parameters (**a**, **b**) of the resulting line $x = a*y + b$.

Examples

```
SELECT arrayReduce('simpleLinearRegression', [0, 1, 2, 3], [0, 1, 2, 3])
```

```
└─arrayReduce('simpleLinearRegression', [0, 1, 2, 3], [0, 1, 2, 3])─┐
| (1,0) |
└──────────────────────────────────────────────────────────────────┘
```

```
SELECT arrayReduce('simpleLinearRegression', [0, 1, 2, 3], [3, 4, 5, 6])
```

```
└─arrayReduce('simpleLinearRegression', [0, 1, 2, 3], [3, 4, 5, 6])─┐
| (1,3) |
└──────────────────────────────────────────────────────────────────┘
```

Aggregate function combinators

The name of an aggregate function can have a suffix appended to it. This changes the way the aggregate function works.

-If

The suffix -If can be appended to the name of any aggregate function. In this case, the aggregate function accepts an extra argument – a condition (UInt8 type). The aggregate function processes only the rows that trigger the condition. If the condition was not triggered even once, it returns a default value (usually zeros or empty strings).

Examples: `sumIf(column, cond)`, `countIf(cond)`, `avgIf(x, cond)`, `quantilesTimingIf(level1, level2)(x, cond)`, `argMinIf(arg, val, cond)` and so on.

With conditional aggregate functions, you can calculate aggregates for several conditions at once, without using subqueries and **JOINS**. For example, in Yandex.Metrica, conditional aggregate functions are used to implement the segment comparison functionality.

-Array

The -Array suffix can be appended to any aggregate function. In this case, the aggregate function takes arguments of the 'Array(T)' type (arrays) instead of 'T' type arguments. If the aggregate function accepts multiple arguments, this must be arrays of equal lengths. When processing arrays, the aggregate function works like the original aggregate function across all array elements.

Example 1: `sumArray(arr)` - Totals all the elements of all 'arr' arrays. In this example, it could have been written more simply: `sum(arraySum(arr))`.

Example 2: `uniqArray(arr)` – Count the number of unique elements in all 'arr' arrays. This could be done an easier way: `uniq(arrayJoin(arr))`, but it's not always possible to add 'arrayJoin' to a query.

-If and -Array can be combined. However, 'Array' must come first, then 'If'. Examples: `uniqArrayIf(arr, cond)`, `quantilesTimingArrayIf(level1, level2)(arr, cond)`. Due to this order, the 'cond' argument can't be an array.

-State

If you apply this combinator, the aggregate function doesn't return the resulting value (such as the number of unique values for the `uniq` function), but an intermediate state of the aggregation (for `uniq`, this is the hash table for calculating the number of unique values). This is an `AggregateFunction(...)` that can be used for further processing or stored in a table to finish aggregating later. See the sections "AggregatingMergeTree" and "Functions for working with intermediate aggregation states".

-Merge

If you apply this combinator, the aggregate function takes the intermediate aggregation state as an argument, combines the states to finish aggregation, and returns the resulting value.

-MergeState.

Merges the intermediate aggregation states in the same way as the -Merge combinator. However, it doesn't return the resulting value, but an intermediate aggregation state, similar to the -State combinator.

-ForEach

Converts an aggregate function for tables into an aggregate function for arrays that aggregates the corresponding array items and returns an array of results. For example, `sumForEach` for the arrays `[1, 2]`, `[3, 4, 5]` and `[6, 7]` returns the result `[10, 13, 5]` after adding together the corresponding array items.

Parametric aggregate functions

Some aggregate functions can accept not only argument columns (used for compression), but a set of parameters – constants for initialization. The syntax is two pairs of brackets instead of one. The first is for parameters, and the second is for arguments.

`sequenceMatch(pattern)(time, cond1, cond2, ...)`

Pattern matching for event chains.

`pattern` is a string containing a pattern to match. The pattern is similar to a regular expression.

`time` is the time of the event, type support: `Date`, `DateTime`, and other unsigned integer types.

`cond1`, `cond2` ... is from one to 32 arguments of type `UInt8` that indicate whether a certain condition was met for the event.

The function collects a sequence of events in RAM. Then it checks whether this sequence matches the pattern. It returns `UInt8`: 0 if the pattern isn't matched, or 1 if it matches.

Example: `sequenceMatch ('(?1).*(?2)')(EventTime, URL LIKE '%company%', URL LIKE '%cart%')`

- whether there was a chain of events in which a pageview with 'company' in the address occurred earlier than a pageview with 'cart' in the address.

This is a singular example. You could write it using other aggregate functions:

```
minIf(EventTime, URL LIKE '%company%') < maxIf(EventTime, URL LIKE '%cart%').
```

However, there is no such solution for more complex situations.

Pattern syntax:

`(?1)` refers to the condition (any number can be used in place of 1).

`.*` is any number of any events.

`(?t>=1800)` is a time condition.

Any quantity of any type of events is allowed over the specified time.

Instead of `>=`, the following operators can be used: `<`, `>`, `<=`.

Any number may be specified in place of 1800.

Events that occur during the same second can be put in the chain in any order. This may affect the result of the function.

sequenceCount(pattern)(time, cond1, cond2, ...)

Works the same way as the sequenceMatch function, but instead of returning whether there is an event chain, it returns UInt64 with the number of event chains found.

Chains are searched for without overlapping. In other words, the next chain can start only after the end of the previous one.

windowFunnel(window)(timestamp, cond1, cond2, cond3, ...)

Searches for event chains in a sliding time window and calculates the maximum number of events that occurred from the chain.

```
windowFunnel(window)(timestamp, cond1, cond2, cond3, ...)
```

Parameters:

- **window** — Length of the sliding window in seconds.
- **timestamp** — Name of the column containing the timestamp. Data type support: **Date**, **DateTime**, and other unsigned integer types (Note that though timestamp support **UInt64** type, there is a limitation it's value can't overflow maximum of Int64, which is $2^{63} - 1$).
- **cond1, cond2...** — Conditions or data describing the chain of events. Data type: **UInt8**. Values can be 0 or 1.

Algorithm

- The function searches for data that triggers the first condition in the chain and sets the event counter to 1. This is the moment when the sliding window starts.
- If events from the chain occur sequentially within the window, the counter is incremented. If the sequence of events is disrupted, the counter isn't incremented.
- If the data has multiple event chains at varying points of completion, the function will only output the size of the longest chain.

Returned value

- Integer. The maximum number of consecutive triggered conditions from the chain within the sliding time window. All the chains in the selection are analyzed.

Example

Determine if one hour is enough for the user to select a phone and purchase it in the online store.

Set the following chain of events:

1. The user logged in to their account on the store (**eventID=1001**).
2. The user searches for a phone (**eventID = 1003, product = 'phone'**).
3. The user placed an order (**eventID = 1009**).

To find out how far the user **user_id** could get through the chain in an hour in January of 2017, make the query:

```
SELECT
  level,
  count() AS c
FROM
(
  SELECT
    user_id,
    windowFunnel(3600)(timestamp, eventID = 1001, eventID = 1003 AND product = 'phone', eventID = 1009) AS level
  FROM trend_event
  WHERE (event_date >= '2017-01-01') AND (event_date <= '2017-01-31')
  GROUP BY user_id
)
GROUP BY level
ORDER BY level
```

Simply, the level value could only be 0, 1, 2, 3, it means the maximum event action stage that one user could reach.

retention(cond1, cond2, ...)

Retention refers to the ability of a company or product to retain its customers over some specified periods.

cond1, **cond2** ... is from one to 32 arguments of type UInt8 that indicate whether a certain condition was met for the event

Example:

Consider you are doing a website analytics, intend to calculate the retention of customers

This could be easily calculate by **retention**

```
SELECT
  sum(r[1]) AS r1,
  sum(r[2]) AS r2,
  sum(r[3]) AS r3
FROM
(
  SELECT
    uid,
    retention(date = '2018-08-10', date = '2018-08-11', date = '2018-08-12') AS r
  FROM events
  WHERE date IN ('2018-08-10', '2018-08-11', '2018-08-12')
  GROUP BY uid
)
```

Simply, **r1** means the number of unique visitors who met the **cond1** condition, **r2** means the number of unique visitors who met **cond1** and **cond2** conditions, **r3** means the number of unique visitors who met **cond1** and **cond3** conditions.

uniqUpTo(N)(x)

Calculates the number of different argument values if it is less than or equal to N. If the number of different argument values is greater than N, it returns N + 1.

Recommended for use with small Ns, up to 10. The maximum value of N is 100.

For the state of an aggregate function, it uses the amount of memory equal to 1 + N * the size of one value of bytes.

For strings, it stores a non-cryptographic hash of 8 bytes. That is, the calculation is approximated for strings.

The function also works for several arguments.

It works as fast as possible, except for cases when a large N value is used and the number of unique values is slightly less than N.

Usage example:

Problem: Generate a report that shows only keywords that produced at least 5 unique users.
Solution: Write in the GROUP BY query SearchPhrase HAVING uniqUpTo(4)(UserID) >= 5

sumMapFiltered(keys_to_keep)(keys, values)

Same behavior as **sumMap** except that an array of keys is passed as a parameter. This can be especially useful when working with a high cardinality of keys.

Table functions

Table functions can be specified in the FROM clause instead of the database and table names.

Table functions can only be used if 'readonly' is not set.

Table functions aren't related to other functions.

file

Creates a table from a file.

```
file(path, format, structure)
```

Input parameters

- **path** — The relative path to the file from **user_files_path**.
- **format** — The **format** of the file.
- **structure** — Structure of the table. Format '**column1_name column1_type, column2_name column2_type, ...**'.

Returned value

A table with the specified structure for reading or writing data in the specified file.

Example

Setting **user_files_path** and the contents of the file **test.csv**:

```
$ grep user_files_path /etc/clickhouse-server/config.xml
<user_files_path>/var/lib/clickhouse/user_files/</user_files_path>

$ cat /var/lib/clickhouse/user_files/test.csv
1,2,3
3,2,1
78,43,45
```

Table from **test.csv** and selection of the first two rows from it:

```
SELECT *
FROM file('test.csv', 'CSV', 'column1 UInt32, column2 UInt32, column3 UInt32')
LIMIT 2
```

column1	column2	column3
1	2	3
3	2	1

```
-- getting the first 10 lines of a table that contains 3 columns of UInt32 type from a CSV file
SELECT * FROM file('test.csv', 'CSV', 'column1 UInt32, column2 UInt32, column3 UInt32') LIMIT 10
```

merge

merge(db_name, 'tables_regexp') – Creates a temporary Merge table. For more information, see the section "Table engines, Merge".

The table structure is taken from the first table encountered that matches the regular expression.

numbers

numbers(N) – Returns a table with the single 'number' column (UInt64) that contains integers from 0 to N-1.

numbers(N, M) - Returns a table with the single 'number' column (UInt64) that contains integers from N to (N + M - 1).

Similar to the **system.numbers** table, it can be used for testing and generating successive values, **numbers(N, M)** more efficient than **system.numbers**.

The following queries are equivalent:

```
SELECT * FROM numbers(10);
SELECT * FROM numbers(0, 10);
SELECT * FROM system.numbers LIMIT 10;
```

Examples:

```
-- Generate a sequence of dates from 2010-01-01 to 2010-12-31
select toDate('2010-01-01') + number as d FROM numbers(365);
```

remote, remoteSecure

Allows you to access remote servers without creating a **Distributed** table.

Signatures:

```
remote('addresses_expr', db, table[, 'user'[, 'password']])
remote('addresses_expr', db.table[, 'user'[, 'password']])
```

addresses_expr – An expression that generates addresses of remote servers. This may be just one server address. The server address is **host:port**, or just **host**. The host can be specified as the server name, or as the IPv4 or IPv6 address. An IPv6 address is specified in square brackets. The port is the TCP port on the remote server. If the port is omitted, it uses **tcp_port** from the server's config file (by default, 9000).

Important

The port is required for an IPv6 address.

Examples:

```
example01-01-1
example01-01-1:9000
localhost
127.0.0.1
[::]:9000
[2a02:6b8:0:1111::11]:9000
```

Multiple addresses can be comma-separated. In this case, ClickHouse will use distributed processing, so it will send the query to all specified addresses (like to shards with different data).

Example:

```
example01-01-1,example01-02-1
```

Part of the expression can be specified in curly brackets. The previous example can be written as follows:

```
example01-0{1,2}-1
```

Curly brackets can contain a range of numbers separated by two dots (non-negative integers). In this case, the range is expanded to a set of values that generate shard addresses. If the first number starts with zero, the values are formed with the same zero alignment. The previous example can be written as follows:

```
example01-{01..02}-1
```

If you have multiple pairs of curly brackets, it generates the direct product of the corresponding sets.

Addresses and parts of addresses in curly brackets can be separated by the pipe symbol (|). In this case, the corresponding sets of addresses are interpreted as replicas, and the query will be sent to the first healthy replica. However, the replicas are iterated in the order currently set in the **load_balancing** setting.

Example:

```
example01-{01..02}-{1|2}
```

This example specifies two shards that each have two replicas.

The number of addresses generated is limited by a constant. Right now this is 1000 addresses.

Using the `remote` table function is less optimal than creating a `Distributed` table, because in this case, the server connection is re-established for every request. In addition, if host names are set, the names are resolved, and errors are not counted when working with various replicas. When processing a large number of queries, always create the `Distributed` table ahead of time, and don't use the `remote` table function.

The `remote` table function can be useful in the following cases:

- Accessing a specific server for data comparison, debugging, and testing.
- Queries between various ClickHouse clusters for research purposes.
- Infrequent distributed requests that are made manually.
- Distributed requests where the set of servers is re-defined each time.

If the user is not specified, `default` is used.

If the password is not specified, an empty password is used.

`remoteSecure` - same as `remote` but with secured connection. Default port — `tcp_port_secure` from config or 9440.

url

`url(URL, format, structure)` - returns a table created from the `URL` with given `format` and `structure`.

`URL` - HTTP or HTTPS server address, which can accept `GET` and/or `POST` requests.

`format` - `format` of the data.

`structure` - table structure in `'UserID UInt64, Name String'` format. Determines column names and types.

Example

```
-- getting the first 3 lines of a table that contains columns of String and UInt32 type from HTTP-server which answers in CSV format.  
SELECT * FROM url('http://127.0.0.1:12345/', CSV, 'column1 String, column2 UInt32') LIMIT 3
```

mysql

Allows to perform `SELECT` queries on data that is stored on a remote MySQL server.

```
mysql('host:port', 'database', 'table', 'user', 'password'[, replace_query, 'on_duplicate_clause']);
```

Parameters

- `host:port` — MySQL server address.
- `database` — Remote database name.
- `table` — Remote table name.
- `user` — MySQL user.
- `password` — User password.
- `replace_query` — If `replace_query=1`, the `REPLACE` query will be performed instead of `INSERT`.
- `on_duplicate_clause` — The `ON DUPLICATE KEY on_duplicate_clause` expression that is added to the `INSERT` query.

Example: `INSERT INTO t (c1,c2) VALUES ('a', 2) ON DUPLICATE KEY UPDATE c2 = c2 + 1` where `on_duplicate_clause` is `UPDATE c2 = c2 + 1`. See MySQL documentation to find which `on_duplicate_clause` you can use with `ON DUPLICATE KEY` clause.

To specify `on_duplicate_clause` you need to pass `0` to the `replace_query` parameter. If you simultaneously pass `replace_query = 1` and `on_duplicate_clause`, ClickHouse generates an exception.

At this time, simple `WHERE` clauses such as `=`, `!=`, `>`, `>=`, `<`, `<=` are executed on the MySQL server.

The rest of the conditions and the `LIMIT` sampling constraint are executed in ClickHouse only after the query to MySQL finishes.

Returned Value

A table object with the same columns as the original MySQL table.

Usage Example

Table in MySQL:

```
mysql> CREATE TABLE `test`.`test` (  
-> `int_id` INT NOT NULL AUTO_INCREMENT,  
-> `int_nullable` INT NULL DEFAULT NULL,  
-> `float` FLOAT NOT NULL,  
-> `float_nullable` FLOAT NULL DEFAULT NULL,  
-> PRIMARY KEY (`int_id`));  
Query OK, 0 rows affected (0,09 sec)  
  
mysql> insert into test (`int_id`, `float`) VALUES (1,2);  
Query OK, 1 row affected (0,00 sec)  
  
mysql> select * from test;  
+-----+-----+-----+-----+  
| int_id | int_nullable | float | float_nullable |  
+-----+-----+-----+-----+  
| 1 | NULL | 2 | NULL |  
+-----+-----+-----+-----+  
1 row in set (0,00 sec)
```

Selection of the data from ClickHouse:

```
SELECT * FROM mysql('localhost:3306', 'test', 'test', 'bayonet', '123')
```

int_id	int_nullable	float	float_nullable
1	NULL	2	NULL

See Also

- [The 'MySQL' table engine](#)
- [Using MySQL as a source of external dictionary](#)

jdbc

`jdbc(jdbc_connection_uri, schema, table)` - returns table that is connected via JDBC driver.

This table function requires separate [clickhouse-jdbc-bridge](#) program to be running.
It supports Nullable types (based on DDL of remote table that is queried).

Examples

```
SELECT * FROM jdbc('jdbc:mysql://localhost:3306/?user=root&password=root', 'schema', 'table')
```

```
SELECT * FROM jdbc('mysql://localhost:3306/?user=root&password=root', 'schema', 'table')
```

```
SELECT * FROM jdbc('datasource://mysql-local', 'schema', 'table')
```

odbc

Returns table that is connected via [ODBC](#).

```
odbc(connection_settings, external_database, external_table)
```

Parameters:

- `connection_settings` — Name of the section with connection settings in the `odbc.ini` file.
- `external_database` — Name of a database in an external DBMS.
- `external_table` — Name of a table in the `external_database`.

To implement ODBC connection safely, ClickHouse uses the separate program `clickhouse-odbc-bridge`. If the ODBC driver is loaded directly from the `clickhouse-server` program, the problems in the driver can crash the ClickHouse server. ClickHouse starts the `clickhouse-odbc-bridge` program automatically when it is required. The ODBC bridge program is installed by the same package as the `clickhouse-server`.

The fields with the **NULL** values from the external table are converted into the default values for the base data type. For example, if a remote MySQL table field has the **INT NULL** type it is converted to 0 (the default value for ClickHouse **Int32** data type).

Usage example

Getting data from the local MySQL installation via ODBC

This example is for linux Ubuntu 18.04 and MySQL server 5.7.

Ensure that there are unixODBC and MySQL Connector are installed.

By default (if installed from packages) ClickHouse starts on behalf of the user `clickhouse`. Thus you need to create and configure this user at MySQL server.

```
sudo mysql
mysql> CREATE USER 'clickhouse'@'localhost' IDENTIFIED BY 'clickhouse';
mysql> GRANT ALL PRIVILEGES ON *.* TO 'clickhouse'@'clickhouse' WITH GRANT OPTION;
```

Then configure the connection in `/etc/odbc.ini`.

```
$ cat /etc/odbc.ini
[mysqlconn]
DRIVER = /usr/local/lib/libmyodbc5w.so
SERVER = 127.0.0.1
PORT = 3306
DATABASE = test
USERNAME = clickhouse
PASSWORD = clickhouse
```

You can check the connection by the `isql` utility from the unixODBC installation.

```
isql -v mysqlconn
+-----+
| Connected! |
|           |
|           |
| ...       |
```

Table in MySQL:

```
mysql> CREATE TABLE `test`.`test` (
-> `int_id` INT NOT NULL AUTO_INCREMENT,
-> `int_nullable` INT NULL DEFAULT NULL,
-> `float` FLOAT NOT NULL,
-> `float_nullable` FLOAT NULL DEFAULT NULL,
-> PRIMARY KEY (`int_id`));

Query OK, 0 rows affected (0,09 sec)

mysql> insert into test (`int_id`, `float`) VALUES (1,2);

Query OK, 1 row affected (0,00 sec)

mysql> select * from test;

+-----+-----+-----+-----+
| int_id | int_nullable | float | float_nullable |
+-----+-----+-----+-----+
| 1 | NULL | 2 | NULL |
+-----+-----+-----+-----+

1 row in set (0,00 sec)
```

Getting data from the MySQL table:

```
SELECT * FROM odbcd('DSN=mysqlconn', 'test', 'test')
```

int_id	int_nullable	float	float_nullable
1	0	2	0

See Also

- [ODBC external dictionaries](#)
- [ODBC table engine](#).

Dictionaries

A dictionary is a mapping ([key -> attributes](#)) that is convenient for various types of reference lists.

ClickHouse supports special functions for working with dictionaries that can be used in queries. It is easier and more efficient to use dictionaries with functions than a [JOIN](#) with reference tables.

NULL values can't be stored in a dictionary.

ClickHouse supports:

- [Built-in dictionaries](#) with a specific [set of functions](#).
- [Plug-in \(external\) dictionaries](#) with a [set of functions](#).

External Dictionaries

You can add your own dictionaries from various data sources. The data source for a dictionary can be a local text or executable file, an HTTP(s) resource, or another DBMS. For more information, see "[Sources for external dictionaries](#)".

ClickHouse:

- Fully or partially stores dictionaries in RAM.
- Periodically updates dictionaries and dynamically loads missing values. In other words, dictionaries can be loaded dynamically.

The configuration of external dictionaries is located in one or more files. The path to the configuration is specified in the [dictionaries_config](#) parameter.

Dictionaries can be loaded at server startup or at first use, depending on the [dictionaries_lazy_load](#) setting.

The dictionary config file has the following format:

```
<yandex>
  <comment>An optional element with any content. Ignored by the ClickHouse server.</comment>

  <!--Optional element. File name with substitutions-->
  <include_from>/etc/metrika.xml</include_from>

  <dictionary>
    <!-- Dictionary configuration -->
  </dictionary>

  ...

  <dictionary>
    <!-- Dictionary configuration -->
  </dictionary>
</yandex>
```

You can **configure** any number of dictionaries in the same file. The file format is preserved even if there is only one dictionary (i.e. `<yandex><dictionary> <!--configuration --> </dictionary></yandex>`).

See also "**Functions for working with external dictionaries**".

Attention

You can convert values for a small dictionary by describing it in a **SELECT** query (see the **transform** function). This functionality is not related to external dictionaries.

Configuring an External Dictionary

The dictionary configuration has the following structure:

```
<dictionary>
  <name>dict_name</name>

  <source>
    <!-- Source configuration -->
  </source>

  <layout>
    <!-- Memory layout configuration -->
  </layout>

  <structure>
    <!-- Complex key configuration -->
  </structure>

  <lifetime>
    <!-- Lifetime of dictionary in memory -->
  </lifetime>
</dictionary>
```

- **name** – The identifier that can be used to access the dictionary. Use the characters `[a-zA-Z0-9_\-]`.
- **source** — Source of the dictionary.
- **layout** — Dictionary layout in memory.
- **structure** — Structure of the dictionary . A key and attributes that can be retrieved by this key.
- **lifetime** — Frequency of dictionary updates.

Storing Dictionaries in Memory

There are a variety of ways to store dictionaries in memory.

We recommend **flat**, **hashed** and **complex_key_hashed**. which provide optimal processing speed.

Caching is not recommended because of potentially poor performance and difficulties in selecting optimal parameters. Read more in the section "**cache**".

There are several ways to improve dictionary performance:

- Call the function for working with the dictionary after **GROUP BY**.
- Mark attributes to extract as injective. An attribute is called injective if different attribute values correspond to different keys. So when **GROUP BY** uses a function that fetches an attribute value by the key, this function is automatically taken out of **GROUP BY**.

ClickHouse generates an exception for errors with dictionaries. Examples of errors:

- The dictionary being accessed could not be loaded.
- Error querying a **cached** dictionary.

You can view the list of external dictionaries and their statuses in the **system.dictionaries** table.

The configuration looks like this:

```
<yandex>
  <dictionary>
    ...
    <layout>
      <layout_type>
        <!-- layout settings -->
      </layout_type>
    </layout>
    ...
  </dictionary>
</yandex>
```

Ways to Store Dictionaries in Memory

- flat
- hashed
- cache
- range_hashed
- complex_key_hashed
- complex_key_cache
- ip_trie

flat

The dictionary is completely stored in memory in the form of flat arrays. How much memory does the dictionary use? The amount is proportional to the size of the largest key (in space used).

The dictionary key has the `UInt64` type and the value is limited to 500,000. If a larger key is discovered when creating the dictionary, ClickHouse throws an exception and does not create the dictionary.

All types of sources are supported. When updating, data (from a file or from a table) is read in its entirety.

This method provides the best performance among all available methods of storing the dictionary.

Configuration example:

```
<layout>
  <flat />
</layout>
```

hashed

The dictionary is completely stored in memory in the form of a hash table. The dictionary can contain any number of elements with any identifiers. In practice, the number of keys can reach tens of millions of items.

All types of sources are supported. When updating, data (from a file or from a table) is read in its entirety.

Configuration example:

```
<layout>
  <hashed />
</layout>
```

complex_key_hashed

This type of storage is for use with composite `keys`. Similar to `hashed`.

Configuration example:

```
<layout>
  <complex_key_hashed />
</layout>
```

range_hashed

The dictionary is stored in memory in the form of a hash table with an ordered array of ranges and their corresponding values.

This storage method works the same way as hashed and allows using date/time ranges in addition to the key, if they appear in the dictionary.

Example: The table contains discounts for each advertiser in the format:

advertiser id	discount start date	discount end date	amount
123	2015-01-01	2015-01-15	0.15
123	2015-01-16	2015-01-31	0.25
456	2015-01-01	2015-01-15	0.05

To use a sample for date ranges, define the `range_min` and `range_max` elements in the `structure`.

Example:

```
<structure>
  <id>
    <name>Id</name>
  </id>
  <range_min>
    <name>first</name>
  </range_min>
  <range_max>
    <name>last</name>
  </range_max>
  ...
</structure>
```

To work with these dictionaries, you need to pass an additional date argument to the `dictGetT` function:

```
dictGetT('dict_name', 'attr_name', id, date)
```

This function returns the value for the specified `ids` and the date range that includes the passed date.

Details of the algorithm:

- If the `id` is not found or a range is not found for the `id`, it returns the default value for the dictionary.
- If there are overlapping ranges, you can use any.
- If the range delimiter is `NULL` or an invalid date (such as 1900-01-01 or 2039-01-01), the range is left open. The range can be open on both sides.

Configuration example:

```

<yandex>
  <dictionary>

    ...

    <layout>
      <range_hashed />
    </layout>

    <structure>
      <id>
        <name>Abcdef</name>
      </id>
      <range_min>
        <name>StartDate</name>
      </range_min>
      <range_max>
        <name>EndDate</name>
      </range_max>
      <attribute>
        <name>XXXType</name>
        <type>String</type>
        <null_value />
      </attribute>
    </structure>

  </dictionary>
</yandex>

```

cache

The dictionary is stored in a cache that has a fixed number of cells. These cells contain frequently used elements.

When searching for a dictionary, the cache is searched first. For each block of data, all keys that are not found in the cache or are outdated are requested from the source using `SELECT attrs... FROM db.table WHERE id IN (k1, k2, ...)`. The received data is then written to the cache.

For cache dictionaries, the expiration **lifetime** of data in the cache can be set. If more time than **lifetime** has passed since loading the data in a cell, the cell's value is not used, and it is re-requested the next time it needs to be used.

This is the least effective of all the ways to store dictionaries. The speed of the cache depends strongly on correct settings and the usage scenario. A cache type dictionary performs well only when the hit rates are high enough (recommended 99% and higher). You can view the average hit rate in the `system.dictionaries` table.

To improve cache performance, use a subquery with **LIMIT**, and call the function with the dictionary externally.

Supported **sources**: MySQL, ClickHouse, executable, HTTP.

Example of settings:

```

<layout>
  <cache>
    <!-- The size of the cache, in number of cells. Rounded up to a power of two. -->
    <size_in_cells>1000000000</size_in_cells>
  </cache>
</layout>

```

Set a large enough cache size. You need to experiment to select the number of cells:

1. Set some value.
2. Run queries until the cache is completely full.
3. Assess memory consumption using the `system.dictionaries` table.
4. Increase or decrease the number of cells until the required memory consumption is reached.

Warning

Do not use ClickHouse as a source, because it is slow to process queries with random reads.

complex_key_cache

This type of storage is for use with composite **keys**. Similar to **cache**.

ip_trie

This type of storage is for mapping network prefixes (IP addresses) to metadata such as ASN.

Example: The table contains network prefixes and their corresponding AS number and country code:

prefix	asn	cca2
202.79.32.0/20	17501	NP
2620:0:870::/48	3856	US
2a02:6b8:1::/48	13238	RU
2001:db8::/32	65536	ZZ

When using this type of layout, the structure must have a composite key.

Example:

```
<structure>
  <key>
    <attribute>
      <name>prefix</name>
      <type>String</type>
    </attribute>
  </key>
  <attribute>
    <name>asn</name>
    <type>UInt32</type>
    <null_value />
  </attribute>
  <attribute>
    <name>cca2</name>
    <type>String</type>
    <null_value>??</null_value>
  </attribute>
  ...
</structure>
```

The key must have only one String type attribute that contains an allowed IP prefix. Other types are not supported yet.

For queries, you must use the same functions (**dictGet** with a tuple) as for dictionaries with composite keys:

```
dictGet('dict_name', 'attr_name', tuple(ip))
```

The function takes either **UInt32** for IPv4, or **FixedString(16)** for IPv6:

```
dictGetString('prefix', 'asn', tuple(IPv6StringToNum('2001:db8::1')))
```

Other types are not supported yet. The function returns the attribute for the prefix that corresponds to this IP address. If there are overlapping prefixes, the most specific one is returned.

Data is stored in a **trie**. It must completely fit into RAM.

Dictionary Updates

ClickHouse periodically updates the dictionaries. The update interval for fully downloaded dictionaries and the invalidation interval for cached dictionaries are defined in the `<lifetime>` tag in seconds.

Dictionary updates (other than loading for first use) do not block queries. During updates, the old version of a dictionary is used. If an error occurs during an update, the error is written to the server log, and queries continue using the old version of dictionaries.

Example of settings:

```
<dictionary>
...
<lifetime>300</lifetime>
...
</dictionary>
```

Setting `<lifetime> 0</lifetime>` prevents updating dictionaries.

You can set a time interval for upgrades, and ClickHouse will choose a uniformly random time within this range. This is necessary in order to distribute the load on the dictionary source when upgrading on a large number of servers.

Example of settings:

```
<dictionary>
...
<lifetime>
  <min>300</min>
  <max>360</max>
</lifetime>
...
</dictionary>
```

When upgrading the dictionaries, the ClickHouse server applies different logic depending on the type of `source`:

- For a text file, it checks the time of modification. If the time differs from the previously recorded time, the dictionary is updated.
- For MyISAM tables, the time of modification is checked using a `SHOW TABLE STATUS` query.
- Dictionaries from other sources are updated every time by default.

For MySQL (InnoDB), ODBC and ClickHouse sources, you can set up a query that will update the dictionaries only if they really changed, rather than each time. To do this, follow these steps:

- The dictionary table must have a field that always changes when the source data is updated.
- The settings of the source must specify a query that retrieves the changing field. The ClickHouse server interprets the query result as a row, and if this row has changed relative to its previous state, the dictionary is updated. Specify the query in the `<invalidate_query>` field in the settings for the `source`.

Example of settings:

```
<dictionary>
...
<odbc>
...
<invalidate_query>SELECT update_time FROM dictionary_source where id = 1</invalidate_query>
</odbc>
...
</dictionary>
```

Sources of External Dictionaries

An external dictionary can be connected from many different sources.

The configuration looks like this:

```
<yandex>
<dictionary>
...
<source>
  <source_type>
    <!-- Source configuration -->
  </source_type>
</source>
...
</dictionary>
...
</yandex>
```

The source is configured in the **source** section.

Types of sources (**source_type**):

- **Local file**
- **Executable file**
- **HTTP(s)**
- **DBMS**
 - **MySQL**
 - **ClickHouse**
 - **MongoDB**
 - **ODBC**

Local File

Example of settings:

```
<source>
<file>
  <path>/opt/dictionaries/os.tsv</path>
  <format>TabSeparated</format>
</file>
</source>
```

Setting fields:

- **path** – The absolute path to the file.
- **format** – The file format. All the formats described in "**Formats**" are supported.

Executable File

Working with executable files depends on **how the dictionary is stored in memory**. If the dictionary is stored using **cache** and **complex_key_cache**, ClickHouse requests the necessary keys by sending a request to the executable file's STDIN. Otherwise, ClickHouse starts executable file and treats its output as dictionary data.

Example of settings:

```
<source>
<executable>
  <command>cat /opt/dictionaries/os.tsv</command>
  <format>TabSeparated</format>
</executable>
</source>
```

Setting fields:

- **command** – The absolute path to the executable file, or the file name (if the program directory is written to **PATH**).

- **format** – The file format. All the formats described in "**Formats**" are supported.

HTTP(s)

Working with an HTTP(s) server depends on **how the dictionary is stored in memory**. If the dictionary is stored using **cache** and **complex_key_cache**, ClickHouse requests the necessary keys by sending a request via the **POST** method.

Example of settings:

```
<source>
  <http>
    <url>http://[::1]/os.tsv</url>
    <format>TabSeparated</format>
  </http>
</source>
```

In order for ClickHouse to access an HTTPS resource, you must **configure openssl** in the server configuration.

Setting fields:

- **url** – The source URL.
- **format** – The file format. All the formats described in "**Formats**" are supported.

ODBC

You can use this method to connect any database that has an ODBC driver.

Example of settings:

```
<odbc>
  <db>DatabaseName</db>
  <table>ShemaName.TableName</table>
  <connection_string>DSN=some_parameters</connection_string>
  <invalidate_query>SQL_QUERY</invalidate_query>
</odbc>
```

Setting fields:

- **db** – Name of the database. Omit it if the database name is set in the **<connection_string>** parameters.
- **table** – Name of the table and schema if exists.
- **connection_string** – Connection string.
- **invalidate_query** – Query for checking the dictionary status. Optional parameter. Read more in the section **Updating dictionaries**.

ClickHouse receives quoting symbols from ODBC-driver and quote all settings in queries to driver, so it's necessary to set table name accordingly to table name case in database.

If you have a problems with encodings when using Oracle, see the corresponding **FAQ** article.

Known vulnerability of the ODBC dictionary functionality

Attention

When connecting to the database through the ODBC driver connection parameter **Servename** can be substituted. In this case values of **USERNAME** and **PASSWORD** from **odbc.ini** are sent to the remote server and can be compromised.

Example of insecure use

Let's configure unixODBC for PostgreSQL. Content of **/etc/odbc.ini**:

```
[gregtest]
Driver = /usr/lib/psqlodbc.so
Servername = localhost
PORT = 5432
DATABASE = test_db
##OPTION = 3
USERNAME = test
PASSWORD = test
```

If you then make a query such as

```
SELECT * FROM odbc('DSN=gregtest;Servername=some-server.com', 'test_db');
```

ODBC driver will send values of **USERNAME** and **PASSWORD** from **odbc.ini** to **some-server.com**.

Example of Connecting PostgreSQL

Ubuntu OS.

Installing unixODBC and the ODBC driver for PostgreSQL:

```
sudo apt-get install -y unixodbc odbcinst odbc-postgresql
```

Configuring **/etc/odbc.ini** (or **~/odbc.ini**):

```
[DEFAULT]
Driver = myconnection

[myconnection]
Description      = PostgreSQL connection to my_db
Driver           = PostgreSQL Unicode
Database         = my_db
Servername       = 127.0.0.1
UserName         = username
Password         = password
Port             = 5432
Protocol         = 9.3
ReadOnly         = No
RowVersioning    = No
ShowSystemTables = No
ConnSettings     =
```

The dictionary configuration in ClickHouse:

```

<yandex>
  <dictionary>
    <name>table_name</name>
    <source>
      <odbc>
        <!-- You can specify the following parameters in connection_string: -->
        <!-- DSN=myconnection;UID=username;PWD=password;HOST=127.0.0.1;PORT=5432;DATABASE=my_db -->
        <connection_string>DSN=myconnection</connection_string>
        <table>postgresql_table</table>
      </odbc>
    </source>
    <lifetime>
      <min>300</min>
      <max>360</max>
    </lifetime>
    <layout>
      <hashed/>
    </layout>
    <structure>
      <id>
        <name>id</name>
      </id>
      <attribute>
        <name>some_column</name>
        <type>UInt64</type>
        <null_value>0</null_value>
      </attribute>
    </structure>
  </dictionary>
</yandex>

```

You may need to edit `odbc.ini` to specify the full path to the library with the driver `DRIVER=/usr/local/lib/psqlodbcw.so`.

Example of Connecting MS SQL Server

Ubuntu OS.

Installing the driver: :

```
sudo apt-get install tdsodbc freetds-bin sqsh
```

Configuring the driver: :

```
$ cat /etc/freetds/freetds.conf
```

```
...
```

```
[MSSQL]
```

```
host = 192.168.56.101
```

```
port = 1433
```

```
tds version = 7.0
```

```
client charset = UTF-8
```

```
$ cat /etc/odbcinst.ini
```

```
...
```

```
[FreeTDS]
```

```
Description      = FreeTDS
```

```
Driver           = /usr/lib/x86_64-linux-gnu/odbc/libtdsodbc.so
```

```
Setup            = /usr/lib/x86_64-linux-gnu/odbc/libtdsS.so
```

```
FileUsage        = 1
```

```
UsageCount       = 5
```

```
$ cat ~/.odbc.ini
```

```
...
```

```
[MSSQL]
```

```
Description      = FreeTDS
```

```
Driver           = FreeTDS
```

```
Servename        = MSSQL
```

```
Database         = test
```

```
UID              = test
```

```
PWD              = test
```

```
Port             = 1433
```

Configuring the dictionary in ClickHouse:

```
<yandex>
```

```
<dictionary>
```

```
<name>test</name>
```

```
<source>
```

```
<odbc>
```

```
<table>dict</table>
```

```
<connection_string>DSN=MSSQL;UID=test;PWD=test</connection_string>
```

```
</odbc>
```

```
</source>
```

```
<lifetime>
```

```
<min>300</min>
```

```
<max>360</max>
```

```
</lifetime>
```

```
<layout>
```

```
<flat />
```

```
</layout>
```

```
<structure>
```

```
<id>
```

```
<name>k</name>
```

```
</id>
```

```
<attribute>
```

```
<name>s</name>
```

```
<type>String</type>
```

```
<null_value></null_value>
```

```
</attribute>
```

```
</structure>
```

```
</dictionary>
```

```
</yandex>
```

DBMS

MySQL

Example of settings:

```
<source>
<mysql>
  <port>3306</port>
  <user>clickhouse</user>
  <password>qwerty</password>
  <replica>
    <host>example01-1</host>
    <priority>1</priority>
  </replica>
  <replica>
    <host>example01-2</host>
    <priority>1</priority>
  </replica>
  <db>db_name</db>
  <table>table_name</table>
  <where>id=10</where>
  <invalidate_query>SQL_QUERY</invalidate_query>
</mysql>
</source>
```

Setting fields:

- **port** – The port on the MySQL server. You can specify it for all replicas, or for each one individually (inside **<replica>**).
- **user** – Name of the MySQL user. You can specify it for all replicas, or for each one individually (inside **<replica>**).
- **password** – Password of the MySQL user. You can specify it for all replicas, or for each one individually (inside **<replica>**).
- **replica** – Section of replica configurations. There can be multiple sections.
 - **replica/host** – The MySQL host.
 - * **replica/priority** – The replica priority. When attempting to connect, ClickHouse traverses the replicas in order of priority. The lower the number, the higher the priority.
- **db** – Name of the database.
- **table** – Name of the table.
- **where** – The selection criteria. Optional parameter.
- **invalidate_query** – Query for checking the dictionary status. Optional parameter. Read more in the section [Updating dictionaries](#).

MySQL can be connected on a local host via sockets. To do this, set **host** and **socket**.

Example of settings:

```
<source>
  <mysql>
    <host>localhost</host>
    <socket>/path/to/socket/file.sock</socket>
    <user>clickhouse</user>
    <password>qwerty</password>
    <db>db_name</db>
    <table>table_name</table>
    <where>id=10</where>
    <invalidate_query>SQL_QUERY</invalidate_query>
  </mysql>
</source>
```

ClickHouse

Example of settings:

```
<source>
  <clickhouse>
    <host>example01-01-1</host>
    <port>9000</port>
    <user>default</user>
    <password></password>
    <db>default</db>
    <table>ids</table>
    <where>id=10</where>
  </clickhouse>
</source>
```

Setting fields:

- **host** – The ClickHouse host. If it is a local host, the query is processed without any network activity. To improve fault tolerance, you can create a **Distributed** table and enter it in subsequent configurations.
- **port** – The port on the ClickHouse server.
- **user** – Name of the ClickHouse user.
- **password** – Password of the ClickHouse user.
- **db** – Name of the database.
- **table** – Name of the table.
- **where** – The selection criteria. May be omitted.
- **invalidate_query** – Query for checking the dictionary status. Optional parameter. Read more in the section **Updating dictionaries**.

MongoDB

Example of settings:

```
<source>
  <mongodb>
    <host>localhost</host>
    <port>27017</port>
    <user></user>
    <password></password>
    <db>test</db>
    <collection>dictionary_source</collection>
  </mongodb>
</source>
```

Setting fields:

- **host** – The MongoDB host.
- **port** – The port on the MongoDB server.
- **user** – Name of the MongoDB user.
- **password** – Password of the MongoDB user.

- **db** - Name of the database.
- **collection** - Name of the collection.

Dictionary Key and Fields

The **<structure>** clause describes the dictionary key and fields available for queries.

Overall structure:

```
<dictionary>
  <structure>
    <id>
      <name>Id</name>
    </id>

    <attribute>
      <!-- Attribute parameters -->
    </attribute>

    ...

  </structure>
</dictionary>
```

Columns are described in the structure:

- **<id>** - **key column**.
- **<attribute>** - **data column**. There can be a large number of columns.

Key

ClickHouse supports the following types of keys:

- Numeric key. UInt64. Defined in the tag **<id>** .
- Composite key. Set of values of different types. Defined in the tag **<key>** .

A structure can contain either **<id>** or **<key>** .

Warning

The key doesn't need to be defined separately in attributes.

Numeric Key

Format: **UInt64**.

Configuration example:

```
<id>
  <name>Id</name>
</id>
```

Configuration fields:

- name - The name of the column with keys.

Composite Key

The key can be a **tuple** from any types of fields. The **layout** in this case must be **complex_key_hashed** or **complex_key_cache**.

Tip

A composite key can consist of a single element. This makes it possible to use a string as the key, for instance.

The key structure is set in the element `<key>`. Key fields are specified in the same format as the dictionary **attributes**. Example:

```
<structure>
  <key>
    <attribute>
      <name>field1</name>
      <type>String</type>
    </attribute>
    <attribute>
      <name>field2</name>
      <type>UInt32</type>
    </attribute>
    ...
  </key>
...
```

For a query to the `dictGet*` function, a tuple is passed as the key. Example: `dictGetString('dict_name', 'attr_name', tuple('string for field1', num_for_field2))`.

Attributes

Configuration example:

```
<structure>
...
<attribute>
  <name>Name</name>
  <type>Type</type>
  <null_value></null_value>
  <expression>rand64()</expression>
  <hierarchical>true</hierarchical>
  <injective>true</injective>
  <is_object_id>true</is_object_id>
</attribute>
</structure>
```

Configuration fields:

- **name** – The column name.
- **type** – The column type. Sets the method for interpreting data in the source. For example, for MySQL, the field might be **TEXT**, **VARCHAR**, or **BLOB** in the source table, but it can be uploaded as **String**.
- **null_value** – The default value for a non-existing element. In the example, it is an empty string.
- **expression** – The attribute can be an expression. The tag is not required.
- **hierarchical** – Hierarchical support. Mirrored to the parent identifier. By default, **false**.
- **injective** – Whether the **id -> attribute** image is injective. If **true**, then you can optimize the **GROUP BY** clause. By default, **false**.
- **is_object_id** – Whether the query is executed for a MongoDB document by **ObjectID**.

Internal dictionaries

ClickHouse contains a built-in feature for working with a geobase.

This allows you to:

- Use a region's ID to get its name in the desired language.
- Use a region's ID to get the ID of a city, area, federal district, country, or continent.
- Check whether a region is part of another region.
- Get a chain of parent regions.

All the functions support "translocality," the ability to simultaneously use different perspectives on region ownership. For more information, see the section "Functions for working with Yandex.Metrica dictionaries".

The internal dictionaries are disabled in the default package.

To enable them, uncomment the parameters `path_to_regions_hierarchy_file` and `path_to_regions_names_files` in the server configuration file.

The geobase is loaded from text files.

Place the `regions_hierarchy*.txt` files into the `path_to_regions_hierarchy_file` directory. This configuration parameter must contain the path to the `regions_hierarchy.txt` file (the default regional hierarchy), and the other files (`regions_hierarchy_ua.txt`) must be located in the same directory.

Put the `regions_names_*.txt` files in the `path_to_regions_names_files` directory.

You can also create these files yourself. The file format is as follows:

`regions_hierarchy*.txt`: TabSeparated (no header), columns:

- region ID (`UInt32`)
- parent region ID (`UInt32`)
- region type (`UInt8`): 1 - continent, 3 - country, 4 - federal district, 5 - region, 6 - city; other types don't have values
- population (`UInt32`) — optional column

`regions_names_*.txt`: TabSeparated (no header), columns:

- region ID (`UInt32`)
- region name (`String`) — Can't contain tabs or line feeds, even escaped ones.

A flat array is used for storing in RAM. For this reason, IDs shouldn't be more than a million.

Dictionaries can be updated without restarting the server. However, the set of available dictionaries is not updated.

For updates, the file modification times are checked. If a file has changed, the dictionary is updated.

The interval to check for changes is configured in the `builtin_dictionaries_reload_interval` parameter.

Dictionary updates (other than loading at first use) do not block queries. During updates, queries use the old versions of dictionaries. If an error occurs during an update, the error is written to the server log, and queries continue using the old version of dictionaries.

We recommend periodically updating the dictionaries with the geobase. During an update, generate new files and write them to a separate location. When everything is ready, rename them to the files used by the server.

There are also functions for working with OS identifiers and Yandex.Metrica search engines, but they shouldn't be used.

Operators

All operators are transformed to the corresponding functions at the query parsing stage, in accordance with their precedence and associativity.

Groups of operators are listed in order of priority (the higher it is in the list, the earlier the operator is connected to its arguments).

Access Operators

`a[N]` Access to an element of an array; `arrayElement(a, N)` function.

`a.N` – Access to a tuple element; `tupleElement(a, N)` function.

Numeric Negation Operator

`-a` – The `negate(a)` function.

Multiplication and Division Operators

`a * b` – The `multiply(a, b)` function.

`a / b` – The `divide(a, b)` function.

`a % b` – The `modulo(a, b)` function.

Addition and Subtraction Operators

`a + b` – The `plus(a, b)` function.

`a - b` – The `minus(a, b)` function.

Comparison Operators

`a = b` – The `equals(a, b)` function.

`a == b` – The `equals(a, b)` function.

`a != b` – The `notEquals(a, b)` function.

`a <> b` – The `notEquals(a, b)` function.

`a <= b` – The `lessOrEquals(a, b)` function.

`a >= b` – The `greaterOrEquals(a, b)` function.

`a < b` – The `less(a, b)` function.

`a > b` – The `greater(a, b)` function.

`a LIKE s` – The `like(a, b)` function.

`a NOT LIKE s` – The `notLike(a, b)` function.

`a BETWEEN b AND c` – The same as `a >= b AND a <= c`.

`a NOT BETWEEN b AND c` – The same as `a < b OR a > c`.

Operators for Working With Data Sets

See the section *IN operators*.

`a IN ...` – The `in(a, b)` function

`a NOT IN ...` – The `notIn(a, b)` function.

`a GLOBAL IN ...` – The `globalIn(a, b)` function.

`a GLOBAL NOT IN ...` – The `globalNotIn(a, b)` function.

Operator for Working With Dates and Times

```
EXTRACT(part FROM date);
```

Extracts a part from a given date. For example, you can retrieve a month from a given date, or a second from a time.

The `part` parameter specifies which part of the date to retrieve. The following values are available:

- `DAY` — The day of the month. Possible values: 1-31.
- `MONTH` — The number of a month. Possible values: 1-12.
- `YEAR` — The year.
- `SECOND` — The second. Possible values: 0-59.
- `MINUTE` — The minute. Possible values: 0-59.
- `HOURL` — The hour. Possible values: 0-23.

The **part** parameter is case-insensitive.

The **date** parameter specifies the date or the time to process. Either **Date** or **DateTime** type is supported.

Examples:

```
SELECT EXTRACT(DAY FROM toDate('2017-06-15'));
SELECT EXTRACT(MONTH FROM toDate('2017-06-15'));
SELECT EXTRACT(YEAR FROM toDate('2017-06-15'));
```

In the following example we create a table and insert into it a value with the **DateTime** type.

```
CREATE TABLE test.Orders
(
  OrderId UInt64,
  OrderName String,
  OrderDate DateTime
)
ENGINE = Log;
```

```
INSERT INTO test.Orders VALUES (1, 'Jarlberg Cheese', toDateTime('2008-10-11 13:23:44'));
```

```
SELECT
  toYear(OrderDate) AS OrderYear,
  toMonth(OrderDate) AS OrderMonth,
  toDayOfMonth(OrderDate) AS OrderDay,
  toHour(OrderDate) AS OrderHour,
  toMinute(OrderDate) AS OrderMinute,
  toSecond(OrderDate) AS OrderSecond
FROM test.Orders;
```

OrderYear	OrderMonth	OrderDay	OrderHour	OrderMinute	OrderSecond
2008	10	11	13	23	44

You can see more examples in [tests](#).

Logical Negation Operator

NOT a - The **not(a)** function.

Logical AND Operator

a AND b - The **and(a, b)** function.

Logical OR Operator

a OR b - The **or(a, b)** function.

Conditional Operator

a ? b : c - The **if(a, b, c)** function.

Note:

The conditional operator calculates the values of **b** and **c**, then checks whether condition **a** is met, and then returns the corresponding value. If **b** or **c** is an **arrayJoin()** function, each row will be replicated regardless of the "a" condition.

Conditional Expression

```
CASE [x]
  WHEN a THEN b
  [WHEN ... THEN ...]
  [ELSE c]
END
```

If **x** is specified, then `transform(x, [a, ...], [b, ...], c)` function is used. Otherwise – `multif(a, b, ..., c)`.

If there is no **ELSE c** clause in the expression, the default value is **NULL**.

The **transform** function does not work with **NULL**.

Concatenation Operator

`s1 || s2` – The `concat(s1, s2)` function.

Lambda Creation Operator

`x -> expr` – The `lambda(x, expr)` function.

The following operators do not have a priority, since they are brackets:

Array Creation Operator

`[x1, ...]` – The `array(x1, ...)` function.

Tuple Creation Operator

`(x1, x2, ...)` – The `tuple(x2, x2, ...)` function.

Associativity

All binary operators have left associativity. For example, `1 + 2 + 3` is transformed to `plus(plus(1, 2), 3)`.

Sometimes this doesn't work the way you expect. For example, `SELECT 4 > 2 > 3` will result in 0.

For efficiency, the **and** and **or** functions accept any number of arguments. The corresponding chains of **AND** and **OR** operators are transformed to a single call of these functions.

Checking for NULL

ClickHouse supports the **IS NULL** and **IS NOT NULL** operators.

IS NULL

- For **Nullable** type values, the **IS NULL** operator returns:
 - 1**, if the value is **NULL**.
 - 0** otherwise.
- For other values, the **IS NULL** operator always returns **0**.

```
:) SELECT x+100 FROM t_null WHERE y IS NULL
```

```
SELECT x + 100
FROM t_null
WHERE isNull(y)
```

```
┌─plus(x, 100)─┐
│      101    │
└──────────┘
```

1 rows in set. Elapsed: 0.002 sec.

IS NOT NULL

- For **Nullable** type values, the **IS NOT NULL** operator returns:
 - **0**, if the value is **NULL**.
 - **1** otherwise.
- For other values, the **IS NOT NULL** operator always returns **1**.

```
;) SELECT * FROM t_null WHERE y IS NOT NULL
```

```
SELECT *
FROM t_null
WHERE isNotNull(y)
```

x	y
2	3

1 rows in set. Elapsed: 0.002 sec.

Syntax

There are two types of parsers in the system: the full SQL parser (a recursive descent parser), and the data format parser (a fast stream parser).

In all cases except the **INSERT** query, only the full SQL parser is used.

The **INSERT** query uses both parsers:

```
INSERT INTO t VALUES (1, 'Hello, world'), (2, 'abc'), (3, 'def')
```

The **INSERT INTO t VALUES** fragment is parsed by the full parser, and the data **(1, 'Hello, world'), (2, 'abc'), (3, 'def')** is parsed by the fast stream parser. You can also turn on the full parser for the data by using the **input_format_values_interpret_expressions** setting. When **input_format_values_interpret_expressions = 1**, ClickHouse first tries to parse values with the fast stream parser. If it fails, ClickHouse tries to use the full parser for the data, treating it like an SQL **expression**.

Data can have any format. When a query is received, the server calculates no more than **max_query_size** bytes of the request in RAM (by default, 1 MB), and the rest is stream parsed.

This means the system doesn't have problems with large **INSERT** queries, like MySQL does.

When using the **Values** format in an **INSERT** query, it may seem that data is parsed the same as expressions in a **SELECT** query, but this is not true. The **Values** format is much more limited.

Next we will cover the full parser. For more information about format parsers, see the **Formats** section.

Spaces

There may be any number of space symbols between syntactical constructions (including the beginning and end of a query). Space symbols include the space, tab, line feed, CR, and form feed.

Comments

SQL-style and C-style comments are supported.

SQL-style comments: from **--** to the end of the line. The space after **--** can be omitted.

Comments in C-style: from **/*** to ***/**. These comments can be multiline. Spaces are not required here, either.

Keywords

Keywords (such as **SELECT**) are not case-sensitive. Everything else (column names, functions, and so on), in contrast to standard SQL, is case-sensitive.

Keywords are not reserved (they are just parsed as keywords in the corresponding context). If you use **identifiers** the same as the keywords, enclose them into quotes. For example, the query **SELECT "FROM" FROM table_name** is valid if the table **table_name** has column with the name **"FROM"**.

Identifiers

Identifiers are:

- Cluster, database, table, partition and column names.
- Functions.
- Data types.
- **Expression aliases**.

Identifiers can be quoted or non-quoted. It is recommended to use non-quoted identifiers.

Non-quoted identifiers must match the regex `^[a-zA-Z_][0-9a-zA-Z_]*$` and can not be equal to **keywords**. Examples: `x, _1, X_y__Z123_`.

If you want to use identifiers the same as keywords or you want to use other symbols in identifiers, quote it using double quotes or backticks, for example, `"id"`, ``id``.

Literals

There are: numeric, string, compound and **NULL** literals.

Numeric

A numeric literal tries to be parsed:

- First as a 64-bit signed number, using the **strtoull** function.
- If unsuccessful, as a 64-bit unsigned number, using the **strtoll** function.
- If unsuccessful, as a floating-point number using the **strtod** function.
- Otherwise, an error is returned.

The corresponding value will have the smallest type that the value fits in.

For example, 1 is parsed as **UInt8**, but 256 is parsed as **UInt16**. For more information, see **Data types**.

Examples: `1, 18446744073709551615, 0xDEADBEEF, 01, 0.1, 1e100, -1e-100, inf, nan`.

String

Only string literals in single quotes are supported. The enclosed characters can be backslash-escaped. The following escape sequences have a corresponding special value: `\b, \f, \r, \n, \t, \0, \a, \v, \xHH`. In all other cases, escape sequences in the format `\c`, where `c` is any character, are converted to `c`. This means that you can use the sequences `\'` and `\\`. The value will have the **String** type.

The minimum set of characters that you need to escape in string literals: `'` and `\`. Single quote can be escaped with the single quote, literals `'It\'s'` and `'It\'s'` are equal.

Compound

Constructions are supported for arrays: `[1, 2, 3]` and tuples: `(1, 'Hello, world!', 2)`.

Actually, these are not literals, but expressions with the array creation operator and the tuple creation operator, respectively.

An array must consist of at least one item, and a tuple must have at least two items.

Tuples have a special purpose for use in the **IN** clause of a **SELECT** query. Tuples can be obtained as the result of a query, but they can't be saved to a database (with the exception of **Memory** tables).

NULL

Indicates that the value is missing.

In order to store **NULL** in a table field, it must be of the **Nullable** type.

Depending on the data format (input or output), **NULL** may have a different representation. For more information, see the documentation for **data formats**.

There are many nuances to processing **NULL**. For example, if at least one of the arguments of a comparison operation is **NULL**, the result of this operation will also be **NULL**. The same is true for multiplication, addition, and other operations. For more information, read the documentation for each operation.

In queries, you can check **NULL** using the **IS NULL** and **IS NOT NULL** operators and the related functions **isNull** and **isNotNull**.

Functions

Functions are written like an identifier with a list of arguments (possibly empty) in brackets. In contrast to standard SQL, the brackets are required, even for an empty arguments list. Example: **now()**.

There are regular and aggregate functions (see the section "Aggregate functions"). Some aggregate functions can contain two lists of arguments in brackets. Example: **quantile (0.9) (x)**. These aggregate functions are called "parametric" functions, and the arguments in the first list are called "parameters". The syntax of aggregate functions without parameters is the same as for regular functions.

Operators

Operators are converted to their corresponding functions during query parsing, taking their priority and associativity into account.

For example, the expression **1 + 2 * 3 + 4** is transformed to **plus(plus(1, multiply(2, 3)), 4)**.

Data Types and Database Table Engines

Data types and table engines in the **CREATE** query are written the same way as identifiers or functions. In other words, they may or may not contain an arguments list in brackets. For more information, see the sections "Data types," "Table engines," and "CREATE".

Expression Aliases

An alias is a user-defined name for an expression in a query.

expr AS alias

- **AS** — The keyword for defining aliases. You can define the alias for a table name or a column name in a **SELECT** clause without using the **AS** keyword.

For example, **SELECT table_name_alias.column_name FROM table_name table_name_alias**.

In the **CAST** function, the **AS** keyword has another meaning. See the description of the function.

- **expr** — Any expression supported by ClickHouse.

For example, **SELECT column_name * 2 AS double FROM some_table**.

- **alias** — Name for **expr**. Aliases should comply with the **identifiers** syntax.

For example, **SELECT "table t".column_name FROM table_name AS "table t"**.

Notes on Usage

Aliases are global for a query or subquery and you can define an alias in any part of a query for any expression. For example, **SELECT (1 AS n) + 2, n**

Aliases are not visible in subqueries and between subqueries. For example, while executing the query **SELECT (SELECT sum(b.a) + num FROM b) - a.a AS num FROM a** ClickHouse generates the exception **Unknown identifier: num**.

If an alias is defined for the result columns in the **SELECT** clause of a subquery, these columns are visible in the outer query. For example, **SELECT n + m FROM (SELECT 1 AS n, 2 AS m)**

Be careful with aliases that are the same as column or table names. Let's consider the following example:

```
CREATE TABLE t
(
    a Int,
    b Int
)
ENGINE = TinyLog()
```

```
SELECT
    argMax(a, b),
    sum(b) AS b
FROM t
```

Received exception from server (version 18.14.17):

Code: 184. DB::Exception: Received from localhost:9000, 127.0.0.1. DB::Exception: Aggregate function sum(b) is found inside another aggregate function in query.

In this example, we declared table `t` with column `b`. Then, when selecting data, we defined the `sum(b) AS b` alias. As aliases are global, ClickHouse substituted the literal `b` in the expression `argMax(a, b)` with the expression `sum(b)`. This substitution caused the exception.

Asterisk

In a `SELECT` query, an asterisk can replace the expression. For more information, see the section "SELECT".

Expressions

An expression is a function, identifier, literal, application of an operator, expression in brackets, subquery, or asterisk. It can also contain an alias.

A list of expressions is one or more expressions separated by commas.

Functions and operators, in turn, can have expressions as arguments.

Operations

ClickHouse operations manual consists of the following major sections:

- [Requirements](#)
- [Monitoring](#)
- [Troubleshooting](#)
- [Usage Recommendations](#)
- [Update Procedure](#)
- [Access Rights](#)
- [Data Backup](#)
- [Configuration Files](#)
- [Quotas](#)
- [System Tables](#)
- [Server Configuration Parameters](#)
- [Settings](#)
- [Utilities](#)

Requirements

CPU

For installation from prebuilt deb packages, use a CPU with x86_64 architecture and support for SSE 4.2 instructions. To run ClickHouse with processors that do not support SSE 4.2 or have AArch64 or PowerPC64LE architecture, you should build ClickHouse from sources.

ClickHouse implements parallel data processing and uses all the hardware resources available. When choosing a processor, take into account that ClickHouse works more efficiently at configurations with a large number of cores but a lower clock rate than at configurations with fewer cores and a higher clock rate. For example, 16 cores with 2600 MHz is preferable to 8 cores with 3600 MHz.

Use of **Turbo Boost** and **hyper-threading** technologies is recommended. It significantly improves performance with a typical load.

RAM

We recommend to use a minimum of 4GB of RAM in order to perform non-trivial queries. The ClickHouse server can run with a much smaller amount of RAM, but it requires memory for processing queries.

The required volume of RAM depends on:

- The complexity of queries.
- The amount of data that is processed in queries.

To calculate the required volume of RAM, you should estimate the size of temporary data for **GROUP BY**, **DISTINCT**, **JOIN** and other operations you use.

ClickHouse can use external memory for temporary data. See **GROUP BY in External Memory** for details.

Swap File

Disable the swap file for production environments.

Storage Subsystem

You need to have 2GB of free disk space to install ClickHouse.

The volume of storage required for your data should be calculated separately. Assessment should include:

- Estimation of the data volume.

You can take a sample of the data and get the average size of a row from it. Then multiply the value by the number of rows you plan to store.

- The data compression coefficient.

To estimate the data compression coefficient, load a sample of your data into ClickHouse and compare the actual size of the data with the size of the table stored. For example, clickstream data is usually compressed by 6-10 times.

To calculate the final volume of data to be stored, apply the compression coefficient to the estimated data volume. If you plan to store data in several replicas, then multiply the estimated volume by the number of replicas.

Network

If possible, use networks of 10G or higher class.

The network bandwidth is critical for processing distributed queries with a large amount of intermediate data. In addition, network speed affects replication processes.

Software

ClickHouse is developed for the Linux family of operating systems. The recommended Linux distribution is Ubuntu. The **tzdata** package should be installed in the system.

ClickHouse can also work in other operating system families. See details in the **Getting started** section of the documentation.

Monitoring

You can monitor:

- Utilization of hardware resources.
- ClickHouse server metrics.

Resource Utilization

ClickHouse does not monitor the state of hardware resources by itself.

It is highly recommended to set up monitoring for:

- Load and temperature on processors.
You can use [dmesg](#), [turbostat](#) or other instruments.
- Utilization of storage system, RAM and network.

ClickHouse Server Metrics

ClickHouse server has embedded instruments for self-state monitoring.

To track server events use server logs. See the [logger](#) section of the configuration file.

ClickHouse collects:

- Different metrics of how the server uses computational resources.
- Common statistics on query processing.

You can find metrics in the [system.metrics](#), [system.events](#), and [system.asynchronous_metrics](#) tables.

You can configure ClickHouse to export metrics to [Graphite](#). See the [Graphite section](#) in the ClickHouse server configuration file. Before configuring export of metrics, you should set up Graphite by following their official guide <https://graphite.readthedocs.io/en/latest/install.html>.

Additionally, you can monitor server availability through the HTTP API. Send the [HTTP GET](#) request to [/](#). If the server is available, it responds with [200 OK](#).

To monitor servers in a cluster configuration, you should set the [max_replica_delay_for_distributed_queries](#) parameter and use the HTTP resource [/replicas-delay](#). A request to [/replicas-delay](#) returns [200 OK](#) if the replica is available and is not delayed behind the other replicas. If a replica is delayed, it returns information about the gap.

Troubleshooting

- [Installation](#)
- [Connecting to the server](#)
- [Query processing](#)
- [Efficiency of query processing](#)

Installation

You Cannot Get Deb Packages from ClickHouse Repository With apt-get

- Check firewall settings.
- If you cannot access the repository for any reason, download packages as described in the [Getting started](#) article and install them manually using the [sudo dpkg -i <packages>](#) command. You will also need the [tzdata](#) package.

Connecting to the Server

Possible issues:

- The server is not running.
- Unexpected or wrong configuration parameters.

Server Is Not Running

Check if server is running

Command:

```
sudo service clickhouse-server status
```

If the server is not running, start it with the command:

```
sudo service clickhouse-server start
```

Check logs

The main log of `clickhouse-server` is in `/var/log/clickhouse-server/clickhouse-server.log` by default.

If the server started successfully, you should see the strings:

- `<Information> Application: starting up.` — Server started.
- `<Information> Application: Ready for connections.` — Server is running and ready for connections.

If `clickhouse-server` start failed with a configuration error, you should see the `<Error>` string with an error description. For example:

```
2019.01.11 15:23:25.549505 [ 45 ] {} <Error> ExternalDictionaries: Failed reloading 'event2id' external dictionary: Poco::Exception. Code: 1000, e.code() = 111, e.displayText() = Connection refused, e.what() = Connection refused
```

If you don't see an error at the end of the file, look through the entire file starting from the string:

```
<Information> Application: starting up.
```

If you try to start a second instance of `clickhouse-server` on the server, you see the following log:

```
2019.01.11 15:25:11.151730 [ 1 ] {} <Information> : Starting ClickHouse 19.1.0 with revision 54413
2019.01.11 15:25:11.154578 [ 1 ] {} <Information> Application: starting up
2019.01.11 15:25:11.156361 [ 1 ] {} <Information> StatusFile: Status file ./status already exists - unclean restart. Contents:
PID: 8510
Started at: 2019-01-11 15:24:23
Revision: 54413

2019.01.11 15:25:11.156673 [ 1 ] {} <Error> Application: DB::Exception: Cannot lock file ./status. Another server instance in same
directory is already running.
2019.01.11 15:25:11.156682 [ 1 ] {} <Information> Application: shutting down
2019.01.11 15:25:11.156686 [ 1 ] {} <Debug> Application: Uninitializing subsystem: Logging Subsystem
2019.01.11 15:25:11.156716 [ 2 ] {} <Information> BaseDaemon: Stop SignalListener thread
```

See system.d logs

If you don't find any useful information in `clickhouse-server` logs or there aren't any logs, you can view `system.d` logs using the command:

```
sudo journalctl -u clickhouse-server
```

Start clickhouse-server in interactive mode

```
sudo -u clickhouse /usr/bin/clickhouse-server --config-file /etc/clickhouse-server/config.xml
```

This command starts the server as an interactive app with standard parameters of the autostart script. In this mode `clickhouse-server` prints all the event messages in the console.

Configuration Parameters

Check:

- Docker settings.

If you run ClickHouse in Docker in an IPv6 network, make sure that `network=host` is set.

- Endpoint settings.

Check `listen_host` and `tcp_port` settings.

ClickHouse server accepts localhost connections only by default.

- HTTP protocol settings.

Check protocol settings for the HTTP API.

- Secure connection settings.

Check:

- The `tcp_port_secure` setting.
- Settings for `SSL certificates`.

Use proper parameters while connecting. For example, use the `port_secure` parameter with `clickhouse_client`.

- User settings.

You might be using the wrong user name or password.

Query Processing

If ClickHouse is not able to process the query, it sends an error description to the client. In the `clickhouse-client` you get a description of the error in the console. If you are using the HTTP interface, ClickHouse sends the error description in the response body. For example:

```
$ curl 'http://localhost:8123/' --data-binary "SELECT a"
Code: 47, e.displayText() = DB::Exception: Unknown identifier: a. Note that there are no tables (FROM clause) in your query, context: required_names: 'a' source_tables: table_aliases: private_aliases: column_aliases: public_columns: 'a' masked_columns: array_join_columns: source_columns: , e.what() = DB::Exception
```

If you start `clickhouse-client` with the `stack-trace` parameter, ClickHouse returns the server stack trace with the description of an error.

You might see a message about a broken connection. In this case, you can repeat the query. If the connection breaks every time you perform the query, check the server logs for errors.

Efficiency of Query Processing

If you see that ClickHouse is working too slowly, you need to profile the load on the server resources and network for your queries.

You can use the `clickhouse-benchmark` utility to profile queries. It shows the number of queries processed per second, the number of rows processed per second, and percentiles of query processing times.

Usage Recommendations

CPU Scaling Governor

Always use the `performance` scaling governor. The `on-demand` scaling governor works much worse with constantly high demand.

```
echo 'performance' | sudo tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
```

CPU Limitations

Processors can overheat. Use `dmesg` to see if the CPU's clock rate was limited due to overheating.

The restriction can also be set externally at the datacenter level. You can use `turbostat` to monitor it under a load.

RAM

For small amounts of data (up to ~200 GB compressed), it is best to use as much memory as the volume of data. For large amounts of data and when processing interactive (online) queries, you should use a reasonable amount of RAM (128 GB or more) so the hot data subset will fit in the cache of pages. Even for data volumes of ~50 TB per server, using 128 GB of RAM significantly improves query performance compared to 64 GB.

Do not disable overcommit. The value `cat /proc/sys/vm/overcommit_memory` should be 0 or 1. Run

```
echo 0 | sudo tee /proc/sys/vm/overcommit_memory
```

Huge Pages

Always disable transparent huge pages. It interferes with memory allocators, which leads to significant performance degradation.

```
echo 'never' | sudo tee /sys/kernel/mm/transparent_hugepage/enabled
```

Use `perf top` to watch the time spent in the kernel for memory management. Permanent huge pages also do not need to be allocated.

Storage Subsystem

If your budget allows you to use SSD, use SSD.
If not, use HDD. SATA HDDs 7200 RPM will do.

Give preference to a lot of servers with local hard drives over a smaller number of servers with attached disk shelves.
But for storing archives with rare queries, shelves will work.

RAID

When using HDD, you can combine their RAID-10, RAID-5, RAID-6 or RAID-50.
For Linux, software RAID is better (with `mdadm`). We don't recommend using LVM.
When creating RAID-10, select the `far` layout.
If your budget allows, choose RAID-10.

If you have more than 4 disks, use RAID-6 (preferred) or RAID-50, instead of RAID-5.
When using RAID-5, RAID-6 or RAID-50, always increase `stripe_cache_size`, since the default value is usually not the best choice.

```
echo 4096 | sudo tee /sys/block/md2/md/stripe_cache_size
```

Calculate the exact number from the number of devices and the block size, using the formula: $2 * \text{num_devices} * \text{chunk_size_in_bytes} / 4096$.

A block size of 1024 KB is sufficient for all RAID configurations.
Never set the block size too small or too large.

You can use RAID-0 on SSD.
Regardless of RAID use, always use replication for data security.

Enable NCQ with a long queue. For HDD, choose the CFQ scheduler, and for SSD, choose `noop`. Don't reduce the 'readahead' setting.
For HDD, enable the write cache.

File System

Ext4 is the most reliable option. Set the mount options `noatime`, `nobarrier`.
XFS is also suitable, but it hasn't been as thoroughly tested with ClickHouse.
Most other file systems should also work fine. File systems with delayed allocation work better.

Linux Kernel

Don't use an outdated Linux kernel.

Network

If you are using IPv6, increase the size of the route cache.

The Linux kernel prior to 3.2 had a multitude of problems with IPv6 implementation.

Use at least a 10 GB network, if possible. 1 Gb will also work, but it will be much worse for patching replicas with tens of terabytes of data, or for processing distributed queries with a large amount of intermediate data.

ZooKeeper

You are probably already using ZooKeeper for other purposes. You can use the same installation of ZooKeeper, if it isn't already overloaded.

It's best to use a fresh version of ZooKeeper – 3.4.9 or later. The version in stable Linux distributions may be outdated.

You should never use manually written scripts to transfer data between different ZooKeeper clusters, because the result will be incorrect for sequential nodes. Never use the "zkcopy" utility for the same reason:

<https://github.com/ksprojects/zkcopy/issues/15>

If you want to divide an existing ZooKeeper cluster into two, the correct way is to increase the number of its replicas and then reconfigure it as two independent clusters.

Do not run ZooKeeper on the same servers as ClickHouse. Because ZooKeeper is very sensitive for latency and ClickHouse may utilize all available system resources.

With the default settings, ZooKeeper is a time bomb:

The ZooKeeper server won't delete files from old snapshots and logs when using the default configuration (see autopurge), and this is the responsibility of the operator.

This bomb must be defused.

The ZooKeeper (3.5.1) configuration below is used in the Yandex.Metrica production environment as of May 20, 2017:

zoo.cfg:


```
## http://hadoop.apache.org/zookeeper/docs/current/zookeeperAdmin.html

## The number of milliseconds of each tick
tickTime=2000
## The number of ticks that the initial
## synchronization phase can take
initLimit=30000
## The number of ticks that can pass between
## sending a request and getting an acknowledgement
syncLimit=10

maxClientCnxns=2000

maxSessionTimeout=60000000
## the directory where the snapshot is stored.
dataDir=/opt/zookeeper/{ { cluster['name'] } }/data
## Place the dataLogDir to a separate physical disc for better performance
dataLogDir=/opt/zookeeper/{ { cluster['name'] } }/logs

autopurge.snapRetainCount=10
autopurge.purgeInterval=1

## To avoid seeks ZooKeeper allocates space in the transaction log file in
## blocks of preAllocSize kilobytes. The default block size is 64M. One reason
## for changing the size of the blocks is to reduce the block size if snapshots
## are taken more often. (Also, see snapCount).
preAllocSize=131072

## Clients can submit requests faster than ZooKeeper can process them,
## especially if there are a lot of clients. To prevent ZooKeeper from running
## out of memory due to queued requests, ZooKeeper will throttle clients so that
## there is no more than globalOutstandingLimit outstanding requests in the
## system. The default limit is 1,000. ZooKeeper logs transactions to a
## transaction log. After snapCount transactions are written to a log file a
## snapshot is started and a new transaction log file is started. The default
## snapCount is 10,000.
snapCount=3000000

## If this option is defined, requests will be will logged to a trace file named
## traceFile.year.month.day.
##traceFile=

## Leader accepts client connections. Default value is "yes". The leader machine
## coordinates updates. For higher update throughput at the slight expense of
## read throughput the leader can be configured to not accept clients and focus
## on coordination.
leaderServes=yes

standaloneEnabled=false
dynamicConfigFile=/etc/zookeeper-{ { cluster['name'] } }/conf/zoo.cfg.dynamic
```

Java version:

```
java(TM) SE Runtime Environment (build 1.8.0_25-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
```

JVM parameters:

```

NAME=zookeeper-{{ cluster['name'] }}
ZOO_CFG_DIR=/etc/$NAME/conf

## TODO this is really ugly
## How to find out, which jars are needed?
## seems, that log4j requires the log4j.properties file to be in the classpath
CLASSPATH="$ZOO_CFG_DIR:/usr/build/classes:/usr/build/lib/*.jar:/usr/share/zookeeper/zookeeper-3.5.1-
metrika.jar:/usr/share/zookeeper/slf4j-log4j12-1.7.5.jar:/usr/share/zookeeper/slf4j-api-1.7.5.jar:/usr/share/zookeeper/servlet-api-
2.5-20081211.jar:/usr/share/zookeeper/netty-3.7.0.Final.jar:/usr/share/zookeeper/log4j-1.2.16.jar:/usr/share/zookeeper/jline-
2.11.jar:/usr/share/zookeeper/jetty-util-6.1.26.jar:/usr/share/zookeeper/jetty-
6.1.26.jar:/usr/share/zookeeper/javacc.jar:/usr/share/zookeeper/jackson-mapper-asl-1.9.11.jar:/usr/share/zookeeper/jackson-core-
asl-1.9.11.jar:/usr/share/zookeeper/commons-cli-1.2.jar:/usr/src/java/lib/*.jar:/usr/etc/zookeeper"

ZOO_CFG="$ZOO_CFG_DIR/zoo.cfg"
ZOO_LOG_DIR=/var/log/$NAME
USER=zookeeper
GROUP=zookeeper
PIDDIR=/var/run/$NAME
PIDFILE=$PIDDIR/$NAME.pid
SCRIPTNAME=/etc/init.d/$NAME
JAVA=/usr/bin/java
ZOO_MAIN="org.apache.zookeeper.server.quorum.QuorumPeerMain"
ZOO_LOG4J_PROP="INFO,ROLLINGFILE"
JMXLOCALONLY=false
JAVA_OPTS="-Xms{{ cluster.get('xms','128M') }} \
-Xmx{{ cluster.get('xmx','1G') }} \
-Xloggc:/var/log/$NAME/zookeeper-gc.log \
-XX:+UseGCLogFileRotation \
-XX:NumberOfGCLogFiles=16 \
-XX:GCLogFileSize=16M \
-verbose:gc \
-XX:+PrintGCTimeStamps \
-XX:+PrintGCDateStamps \
-XX:+PrintGCDetails \
-XX:+PrintTenuringDistribution \
-XX:+PrintGCApplicationStoppedTime \
-XX:+PrintGCApplicationConcurrentTime \
-XX:+PrintSafepointStatistics \
-XX:+UseParNewGC \
-XX:+UseConcMarkSweepGC \
-XX:+CMSParallelRemarkEnabled"

```

Salt init:

```
description "zookeeper-{{ cluster['name'] }} centralized coordination service"
```

```
start on runlevel [2345]
```

```
stop on runlevel [!2345]
```

```
respawn
```

```
limit nofile 8192 8192
```

```
pre-start script
```

```
  [ -r "/etc/zookeeper-{{ cluster['name'] }}/conf/environment" ] || exit 0
```

```
  . /etc/zookeeper-{{ cluster['name'] }}/conf/environment
```

```
  [ -d $ZOO_LOG_DIR ] || mkdir -p $ZOO_LOG_DIR
```

```
  chown $USER:$GROUP $ZOO_LOG_DIR
```

```
end script
```

```
script
```

```
  . /etc/zookeeper-{{ cluster['name'] }}/conf/environment
```

```
  [ -r /etc/default/zookeeper ] && . /etc/default/zookeeper
```

```
  if [ -z "$JMXDISABLE" ]; then
```

```
    JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote -
```

```
Dcom.sun.management.jmxremote.local.only=$JMXLOCALONLY"
```

```
  fi
```

```
  exec start-stop-daemon --start -c $USER --exec $JAVA --name zookeeper-{{ cluster['name'] }} \
```

```
    -- -cp $CLASSPATH $JAVA_OPTS -Dzookeeper.log.dir=${ZOO_LOG_DIR} \
```

```
    -Dzookeeper.root.logger=${ZOO_LOG4J_PROP} $ZOOMAIN $ZOOCFG
```

```
end script
```

ClickHouse Update

If ClickHouse was installed from deb packages, execute the following commands on the server:

```
sudo apt-get update
```

```
sudo apt-get install clickhouse-client clickhouse-server
```

```
sudo service clickhouse-server restart
```

If you installed ClickHouse using something other than the recommended deb packages, use the appropriate update method.

ClickHouse does not support a distributed update. The operation should be performed consecutively on each separate server. Do not update all the servers on a cluster simultaneously, or the cluster will be unavailable for some time.

Access Rights

Users and access rights are set up in the user config. This is usually [users.xml](#).

Users are recorded in the [users](#) section. Here is a fragment of the [users.xml](#) file:

◀ ▶

The **default** user is chosen in cases when the username is not passed. The **default** user is also used for distributed query processing, if the configuration of the server or cluster doesn't specify the **user** and **password** (see the section on the **Distributed** engine).

The user that is used for exchanging information between servers combined in a cluster must not have substantial restrictions or quotas – otherwise, distributed queries will fail.

The password is specified in clear text (not recommended) or in SHA-256. The hash isn't salted. In this regard, you should not consider these passwords as providing security against potential malicious attacks. Rather, they are necessary for protection from employees.

A list of networks is specified that access is allowed from. In this example, the list of networks for both users is loaded from a separate file (`/etc/metrika.xml`) containing the `networks` substitution. Here is a fragment of it:

```
<yandex>
...
<networks>
  <ip>::/64</ip>
  <ip>203.0.113.0/24</ip>
  <ip>2001:DB8::/32</ip>
...
</networks>
</yandex>
```

You could define this list of networks directly in `users.xml`, or in a file in the `users.d` directory (for more information, see the section "[Configuration files](#)").

The config includes comments explaining how to open access from everywhere.

For use in production, only specify `ip` elements (IP addresses and their masks), since using `host` and `hoost_regexp` might cause extra latency.

Next the user settings profile is specified (see the section "[Settings profiles](#)"). You can specify the default profile, `default`. The profile can have any name. You can specify the same profile for different users. The most important thing you can write in the settings profile is `readonly=1`, which ensures read-only access.

Then specify the quota to be used (see the section "[Quotas](#)"). You can specify the default quota: `default`. It is set in the config by default to only count resource usage, without restricting it. The quota can have any name. You can specify the same quota for different users – in this case, resource usage is calculated for each user individually.

In the optional `<allow_databases>` section, you can also specify a list of databases that the user can access. By default, all databases are available to the user. You can specify the `default` database. In this case, the user will receive access to the database by default.

Access to the `system` database is always allowed (since this database is used for processing queries).

The user can get a list of all databases and tables in them by using `SHOW` queries or system tables, even if access to individual databases isn't allowed.

Database access is not related to the `readonly` setting. You can't grant full access to one database and `readonly` access to another one.

Data Backup

While `replication` provides protection from hardware failures, it does not protect against human errors: accidental deletion of data, deletion of the wrong table or a table on the wrong cluster, and software bugs that result in incorrect data processing or data corruption. In many cases mistakes like these will affect all replicas. ClickHouse has built-in safeguards to prevent some types of mistakes — for example, by default **you can't just drop tables with a MergeTree-like engine containing more than 50 Gb of data**. However, these safeguards don't cover all possible cases and can be circumvented.

In order to effectively mitigate possible human errors, you should carefully prepare a strategy for backing up and restoring your data **in advance**.

Each company has different resources available and business requirements, so there's no universal solution for ClickHouse backups and restores that will fit every situation. What works for one gigabyte of data likely won't work for tens of petabytes. There are a variety of possible approaches with their own pros and cons, which will be discussed below. It is a good idea to use several approaches instead of just one in order to compensate for their various shortcomings.

Note

Keep in mind that if you backed something up and never tried to restore it, chances are that restore will not work properly when you actually need it (or at least it will take longer than business can tolerate). So whatever backup approach you choose, make sure to automate the restore process as well, and practice it on a spare ClickHouse cluster regularly.

Duplicating Source Data Somewhere Else

Often data that is ingested into ClickHouse is delivered through some sort of persistent queue, such as [Apache Kafka](#). In this case it is possible to configure an additional set of subscribers that will read the same data stream while it is being written to ClickHouse and store it in cold storage somewhere. Most companies already have some default recommended cold storage, which could be an object store or a distributed filesystem like [HDFS](#).

Filesystem Snapshots

Some local filesystems provide snapshot functionality (for example, [ZFS](#)), but they might not be the best choice for serving live queries. A possible solution is to create additional replicas with this kind of filesystem and exclude them from the [Distributed](#) tables that are used for [SELECT](#) queries. Snapshots on such replicas will be out of reach of any queries that modify data. As a bonus, these replicas might have special hardware configurations with more disks attached per server, which would be cost-effective.

clickhouse-copier

[clickhouse-copier](#) is a versatile tool that was initially created to re-shard petabyte-sized tables. It can also be used for backup and restore purposes because it reliably copies data between ClickHouse tables and clusters.

For smaller volumes of data, a simple [INSERT INTO ... SELECT ...](#) to remote tables might work as well.

Manipulations with Parts

ClickHouse allows using the [ALTER TABLE ... FREEZE PARTITION ...](#) query to create a local copy of table partitions. This is implemented using hardlinks to the [/var/lib/clickhouse/shadow/](#) folder, so it usually does not consume extra disk space for old data. The created copies of files are not handled by ClickHouse server, so you can just leave them there: you will have a simple backup that doesn't require any additional external system, but it will still be prone to hardware issues. For this reason, it's better to remotely copy them to another location and then remove the local copies. Distributed filesystems and object stores are still a good options for this, but normal attached file servers with a large enough capacity might work as well (in this case the transfer will occur via the network filesystem or maybe [rsync](#)).

For more information about queries related to partition manipulations, see the [ALTER documentation](#).

A third-party tool is available to automate this approach: [clickhouse-backup](#).

Configuration Files

The main server config file is [config.xml](#). It resides in the [/etc/clickhouse-server/](#) directory.

Individual settings can be overridden in the [*.xml](#) and [*.conf](#) files in the [config.d](#) directory next to the config file.

The [replace](#) or [remove](#) attributes can be specified for the elements of these config files.

If neither is specified, it combines the contents of elements recursively, replacing values of duplicate children.

If [replace](#) is specified, it replaces the entire element with the specified one.

If **remove** is specified, it deletes the element.

The config can also define "substitutions". If an element has the **incl** attribute, the corresponding substitution from the file will be used as the value. By default, the path to the file with substitutions is **/etc/metrika.xml**. This can be changed in the **include_from** element in the server config. The substitution values are specified in **/yandex/substitution_name** elements in this file. If a substitution specified in **incl** does not exist, it is recorded in the log. To prevent ClickHouse from logging missing substitutions, specify the **optional="true"** attribute (for example, settings for **macros**).

Substitutions can also be performed from ZooKeeper. To do this, specify the attribute **from_zk = "/path/to/node"**. The element value is replaced with the contents of the node at **/path/to/node** in ZooKeeper. You can also put an entire XML subtree on the ZooKeeper node and it will be fully inserted into the source element.

The **config.xml** file can specify a separate config with user settings, profiles, and quotas. The relative path to this config is set in the 'users_config' element. By default, it is **users.xml**. If **users_config** is omitted, the user settings, profiles, and quotas are specified directly in **config.xml**.

In addition, **users_config** may have overrides in files from the **users_config.d** directory (for example, **users.d**) and substitutions. For example, you can have separate config file for each user like this:

```
$ cat /etc/clickhouse-server/users.d/alice.xml
<yandex>
  <users>
    <alice>
      <profile>analytics</profile>
      <networks>
        <ip>::/0</ip>
      </networks>
      <password_sha256_hex>...</password_sha256_hex>
      <quota>analytics</quota>
    </alice>
  </users>
</yandex>
```

For each config file, the server also generates **file-preprocessed.xml** files when starting. These files contain all the completed substitutions and overrides, and they are intended for informational use. If ZooKeeper substitutions were used in the config files but ZooKeeper is not available on the server start, the server loads the configuration from the preprocessed file.

The server tracks changes in config files, as well as files and ZooKeeper nodes that were used when performing substitutions and overrides, and reloads the settings for users and clusters on the fly. This means that you can modify the cluster, users, and their settings without restarting the server.

Quotas

Quotas allow you to limit resource usage over a period of time, or simply track the use of resources. Quotas are set up in the user config. This is usually 'users.xml'.

The system also has a feature for limiting the complexity of a single query. See the section "Restrictions on query complexity").

In contrast to query complexity restrictions, quotas:

- Place restrictions on a set of queries that can be run over a period of time, instead of limiting a single query.
- Account for resources spent on all remote servers for distributed query processing.

Let's look at the section of the 'users.xml' file that defines quotas.

```

<!-- Quotas -->
<quotas>
  <!-- Quota name. -->
  <default>
    <!-- Restrictions for a time period. You can set many intervals with different restrictions. -->
    <interval>
      <!-- Length of the interval. -->
      <duration>3600</duration>

      <!-- Unlimited. Just collect data for the specified time interval. -->
      <queries>0</queries>
      <errors>0</errors>
      <result_rows>0</result_rows>
      <read_rows>0</read_rows>
      <execution_time>0</execution_time>
    </interval>
  </default>

```

By default, the quota just tracks resource consumption for each hour, without limiting usage. The resource consumption calculated for each interval is output to the server log after each request.

```

<statbox>
  <!-- Restrictions for a time period. You can set many intervals with different restrictions. -->
  <interval>
    <!-- Length of the interval. -->
    <duration>3600</duration>

    <queries>1000</queries>
    <errors>100</errors>
    <result_rows>1000000000</result_rows>
    <read_rows>100000000000</read_rows>
    <execution_time>900</execution_time>
  </interval>

  <interval>
    <duration>86400</duration>

    <queries>10000</queries>
    <errors>1000</errors>
    <result_rows>5000000000</result_rows>
    <read_rows>500000000000</read_rows>
    <execution_time>7200</execution_time>
  </interval>
</statbox>

```

For the 'statbox' quota, restrictions are set for every hour and for every 24 hours (86,400 seconds). The time interval is counted starting from an implementation-defined fixed moment in time. In other words, the 24-hour interval doesn't necessarily begin at midnight.

When the interval ends, all collected values are cleared. For the next hour, the quota calculation starts over.

Here are the amounts that can be restricted:

queries – The total number of requests.

errors – The number of queries that threw an exception.

result_rows – The total number of rows given as the result.

read_rows – The total number of source rows read from tables for running the query, on all remote servers.

execution_time – The total query execution time, in seconds (wall time).

If the limit is exceeded for at least one time interval, an exception is thrown with a text about which restriction was exceeded, for which interval, and when the new interval begins (when queries can be sent again).

Quotas can use the "quota key" feature in order to report on resources for multiple keys independently. Here is an example of this:

```
<!-- For the global reports designer. -->
<web_global>
  <!-- keyed - The quota_key "key" is passed in the query parameter,
    and the quota is tracked separately for each key value.
    For example, you can pass a Yandex.Metrica username as the key,
    so the quota will be counted separately for each username.
    Using keys makes sense only if quota_key is transmitted by the program, not by a user.

    You can also write <keyed_by_ip /> so the IP address is used as the quota key.
    (But keep in mind that users can change the IPv6 address fairly easily.)
  -->
  <keyed />
```

The quota is assigned to users in the 'users' section of the config. See the section "Access rights".

For distributed query processing, the accumulated amounts are stored on the requestor server. So if the user goes to another server, the quota there will "start over".

When the server is restarted, quotas are reset.

System tables

System tables are used for implementing part of the system's functionality, and for providing access to information about how the system is working.

You can't delete a system table (but you can perform DETACH).

System tables don't have files with data on the disk or files with metadata. The server creates all the system tables when it starts.

System tables are read-only.

They are located in the 'system' database.

system.asynchronous_metrics

Contain metrics used for profiling and monitoring.

They usually reflect the number of events currently in the system, or the total resources consumed by the system.

Example: The number of SELECT queries currently running; the amount of memory in

use. [system.asynchronous_metrics](#) and [system.metrics](#) differ in their sets of metrics and how they are calculated.

system.clusters

Contains information about clusters available in the config file and the servers in them.

Columns:

```
cluster String      — The cluster name.
shard_num UInt32    — The shard number in the cluster, starting from 1.
shard_weight UInt32 — The relative weight of the shard when writing data.
replica_num UInt32  — The replica number in the shard, starting from 1.
host_name String    — The host name, as specified in the config.
String host_address — The host IP address obtained from DNS.
port UInt16         — The port to use for connecting to the server.
user String         — The name of the user for connecting to the server.
```

system.columns

Contains information about the columns in all the tables.

You can use this table to get information similar to the [DESCRIBE TABLE](#) query, but for multiple tables at once.

The `system.columns` table contains the following columns (the type of the corresponding column is shown in brackets):

- `database` (String) — Database name.
- `table` (String) — Table name.
- `name` (String) — Column name.
- `type` (String) — Column type.
- `default_kind` (String) — Expression type (`DEFAULT`, `MATERIALIZED`, `ALIAS`) for the default value, or an empty string if it is not defined.
- `default_expression` (String) — Expression for the default value, or an empty string if it is not defined.
- `data_compressed_bytes` (UInt64) — The size of compressed data, in bytes.
- `data_uncompressed_bytes` (UInt64) — The size of decompressed data, in bytes.
- `marks_bytes` (UInt64) — The size of marks, in bytes.
- `comment` (String) — The comment about column, or an empty string if it is not defined.
- `is_in_partition_key` (UInt8) — Flag that indicates whether the column is in partition expression.
- `is_in_sorting_key` (UInt8) — Flag that indicates whether the column is in sorting key expression.
- `is_in_primary_key` (UInt8) — Flag that indicates whether the column is in primary key expression.
- `is_in_sampling_key` (UInt8) — Flag that indicates whether the column is in sampling key expression.

system.databases

This table contains a single String column called 'name' – the name of a database. Each database that the server knows about has a corresponding entry in the table. This system table is used for implementing the `SHOW DATABASES` query.

system.detached_parts

Contains information about detached parts of `MergeTree` tables. The `reason` column specifies why the part was detached. For user-detached parts, the reason is empty. Such parts can be attached with `ALTER TABLE ATTACH PARTITION|PART` command. For the description of other columns, see `system.parts`.

system.dictionaries

Contains information about external dictionaries.

Columns:

- `name` String — Dictionary name.
- `type` String — Dictionary type: Flat, Hashed, Cache.
- `origin` String — Path to the configuration file that describes the dictionary.
- `attribute.names` Array(String) — Array of attribute names provided by the dictionary.
- `attribute.types` Array(String) — Corresponding array of attribute types that are provided by the dictionary.
- `has_hierarchy` UInt8 — Whether the dictionary is hierarchical.
- `bytes_allocated` UInt64 — The amount of RAM the dictionary uses.
- `hit_rate` Float64 — For cache dictionaries, the percentage of uses for which the value was in the cache.
- `element_count` UInt64 — The number of items stored in the dictionary.
- `load_factor` Float64 — The percentage full of the dictionary (for a hashed dictionary, the percentage filled in the hash table).
- `creation_time` DateTime — The time when the dictionary was created or last successfully reloaded.
- `last_exception` String — Text of the error that occurs when creating or reloading the dictionary if the dictionary couldn't be created.
- `source` String — Text describing the data source for the dictionary.

Note that the amount of memory used by the dictionary is not proportional to the number of items stored in it. So for flat and cached dictionaries, all the memory cells are pre-assigned, regardless of how full the dictionary actually is.

system.events

Contains information about the number of events that have occurred in the system. This is used for profiling and monitoring purposes.

Example: The number of processed SELECT queries.

Columns: 'event String' – the event name, and 'value UInt64' – the quantity.

system.functions

Contains information about normal and aggregate functions.

Columns:

- `name(String)` – The name of the function.
- `is_aggregate(UInt8)` – Whether the function is aggregate.

system.graphite_retentions

Contains information about parameters `graphite_rollup` which use in tables with `*GraphiteMergeTree` engines.

Columns:

- `config_name (String)` - `graphite_rollup` parameter name.
- `regexp (String)` - A pattern for the metric name.
- `function (String)` - The name of the aggregating function.
- `age (UInt64)` - The minimum age of the data in seconds.
- `precision (UInt64)` - How precisely to define the age of the data in seconds.
- `priority (UInt16)` - Pattern priority.
- `is_default (UInt8)` - Is pattern default or not.
- `Tables.database (Array(String))` - Array of databases names of tables, which use `config_name` parameter.
- `Tables.table (Array(String))` - Array of tables names, which use `config_name` parameter.

system.merges

Contains information about merges and part mutations currently in process for tables in the MergeTree family.

Columns:

- `database String` — The name of the database the table is in.
- `table String` — Table name.
- `elapsed Float64` — The time elapsed (in seconds) since the merge started.
- `progress Float64` — The percentage of completed work from 0 to 1.
- `num_parts UInt64` — The number of pieces to be merged.
- `result_part_name String` — The name of the part that will be formed as the result of merging.
- `is_mutation UInt8` - 1 if this process is a part mutation.
- `total_size_bytes_compressed UInt64` — The total size of the compressed data in the merged chunks.
- `total_size_marks UInt64` — The total number of marks in the merged parts.
- `bytes_read_uncompressed UInt64` — Number of bytes read, uncompressed.
- `rows_read UInt64` — Number of rows read.
- `bytes_written_uncompressed UInt64` — Number of bytes written, uncompressed.
- `rows_written UInt64` — Number of lines rows written.

system.metrics

system.numbers

This table contains a single UInt64 column named 'number' that contains almost all the natural numbers starting from zero.

You can use this table for tests, or if you need to do a brute force search.

Reads from this table are not parallelized.

system.numbers_mt

The same as 'system.numbers' but reads are parallelized. The numbers can be returned in any order.
Used for tests.

system.one

This table contains a single row with a single 'dummy' UInt8 column containing the value 0.
This table is used if a SELECT query doesn't specify the FROM clause.
This is similar to the DUAL table found in other DBMSs.

system.parts

Contains information about parts of MergeTree tables.

Each row describes one part of the data.

Columns:

- partition (String) – The partition name. To learn what a partition is, see the description of the ALTER query.

Formats:

- YYYYMM for automatic partitioning by month.
- any_string when partitioning manually.

- name (String) – Name of the data part.
- active (UInt8) – Indicates whether the part is active. If a part is active, it is used in a table; otherwise, it will be deleted. Inactive data parts remain after merging.
- marks (UInt64) – The number of marks. To get the approximate number of rows in a data part, multiply marks by the index granularity (usually 8192).
- marks_size (UInt64) – The size of the file with marks.
- rows (UInt64) – The number of rows.
- bytes (UInt64) – The number of bytes when compressed.
- modification_time (DateTime) – The modification time of the directory with the data part. This usually corresponds to the time of data part creation.
- remove_time (DateTime) – The time when the data part became inactive.
- refcount (UInt32) – The number of places where the data part is used. A value greater than 2 indicates that the data part is used in queries or merges.
- min_date (Date) – The minimum value of the date key in the data part.
- max_date (Date) – The maximum value of the date key in the data part.
- min_block_number (UInt64) – The minimum number of data parts that make up the current part after merging.
- max_block_number (UInt64) – The maximum number of data parts that make up the current part after merging.
- level (UInt32) – Depth of the merge tree. If a merge was not performed, level=0.
- primary_key_bytes_in_memory (UInt64) – The amount of memory (in bytes) used by primary key values.
- primary_key_bytes_in_memory_allocated (UInt64) – The amount of memory (in bytes) reserved for primary key values.

- database (String) – Name of the database.
- table (String) – Name of the table.
- engine (String) – Name of the table engine without parameters.

system.part_log

The `system.part_log` table is created only if the `part_log` server setting is specified.

This table contains information about the events that occurred with the `data parts` in the `MergeTree` family tables. For instance, adding or merging data.

The `system.part_log` table contains the following columns:

- `event_type` (Enum) — Type of the event that occurred with the data part. Can have one of the following values: `NEW_PART` — inserting, `MERGE_PARTS` — merging, `DOWNLOAD_PART` — downloading, `REMOVE_PART` — removing or detaching using `DETACH PARTITION`, `MUTATE_PART` — updating.
- `event_date` (Date) — Event date.
- `event_time` (DateTime) — Event time.
- `duration_ms` (UInt64) — Duration.
- `database` (String) — Name of the database the data part is in.
- `table` (String) — Name of the table the data part is in.
- `part_name` (String) — Name of the data part.
- `partition_id` (String) — ID of the partition that the data part was inserted to. The column takes the 'all' value if the partitioning is by `tuple()`.
- `rows` (UInt64) — The number of rows in the data part.
- `size_in_bytes` (UInt64) — Size of the data part in bytes.
- `merged_from` (Array(String)) — An array of names of the parts which the current part was made up from (after the merge).
- `bytes_uncompressed` (UInt64) — Size of uncompressed bytes.
- `read_rows` (UInt64) — The number of rows was read during the merge.
- `read_bytes` (UInt64) — The number of bytes was read during the merge.
- `error` (UInt16) — The code number of the occurred error.
- `exception` (String) — Text message of the occurred error.

The `system.part_log` table is created after the first inserting data to the `MergeTree` table.

system.processes

This system table is used for implementing the `SHOW PROCESSLIST` query.

Columns:

user String	- Name of the user who made the request. For distributed query processing, this is the user who helped the requestor server send the query to this server, not the user who made the distributed request on the requestor server.
address String	- The IP address the request was made from. The same for distributed processing.
elapsed Float64	- The time in seconds since request execution started.
rows_read UInt64	- The number of rows read from the table. For distributed processing, on the requestor server, this is the total for all remote servers.
bytes_read UInt64	- The number of uncompressed bytes read from the table. For distributed processing, on the requestor server, this is the total for all remote servers.
total_rows_approx UInt64	- The approximation of the total number of rows that should be read. For distributed processing, on the requestor server, this is the total for all remote servers. It can be updated during request processing, when new sources to process become known.
memory_usage UInt64	- How much memory the request uses. It might not include some types of dedicated memory.
query String	- The query text. For INSERT, it doesn't include the data to insert.
query_id String	- Query ID, if defined.

system.query_log

Contains information about queries execution. For each query, you can see processing start time, duration of processing, error message and other information.

Note

The table doesn't contain input data for **INSERT** queries.

ClickHouse creates this table only if the **query_log** server parameter is specified. This parameter sets the logging rules. For example, a logging interval or name of a table the queries will be logged in.

To enable query logging, set the parameter **log_queries** to 1. For details, see the **Settings** section.

The **system.query_log** table registers two kinds of queries:

1. Initial queries, that were run directly by the client.
2. Child queries that were initiated by other queries (for distributed query execution). For such a kind of queries, information about the parent queries is shown in the **initial_*** columns.

Columns:

- **type** (UInt8) — Type of event that occurred when executing the query. Possible values:
 - 1 — Successful start of query execution.
 - 2 — Successful end of query execution.
 - 3 — Exception before the start of query execution.
 - 4 — Exception during the query execution.
- **event_date** (Date) — Event date.
- **event_time** (DateTime) — Event time.
- **query_start_time** (DateTime) — Time of the query processing start.
- **query_duration_ms** (UInt64) — Duration of the query processing.
- **read_rows** (UInt64) — Number of read rows.
- **read_bytes** (UInt64) — Number of read bytes.
- **written_rows** (UInt64) — For **INSERT** queries, number of written rows. For other queries, the column value is 0.
- **written_bytes** (UInt64) — For **INSERT** queries, number of written bytes. For other queries, the column value is 0.
- **result_rows** (UInt64) — Number of rows in a result.
- **result_bytes** (UInt64) — Number of bytes in a result.
- **memory_usage** (UInt64) — Memory consumption by the query.

- `query` (String) — Query string.
- `exception` (String) — Exception message.
- `stack_trace` (String) — Stack trace (a list of methods called before the error occurred). An empty string, if the query is completed successfully.
- `is_initial_query` (UInt8) — Kind of query. Possible values:
 - 1 — Query was initiated by the client.
 - 0 — Query was initiated by another query for distributed query execution.
- `user` (String) — Name of the user initiated the current query.
- `query_id` (String) — ID of the query.
- `address` (FixedString(16)) — IP address the query was initiated from.
- `port` (UInt16) — A server port that was used to receive the query.
- `initial_user` (String) — Name of the user who run the parent query (for distributed query execution).
- `initial_query_id` (String) — ID of the parent query.
- `initial_address` (FixedString(16)) — IP address that the parent query was launched from.
- `initial_port` (UInt16) — A server port that was used to receive the parent query from the client.
- `interface` (UInt8) — Interface that the query was initiated from. Possible values:
 - 1 — TCP.
 - 2 — HTTP.
- `os_user` (String) — User's OS.
- `client_hostname` (String) — Server name that the `clickhouse-client` is connected to.
- `client_name` (String) — The `clickhouse-client` name.
- `client_revision` (UInt32) — Revision of the `clickhouse-client`.
- `client_version_major` (UInt32) — Major version of the `clickhouse-client`.
- `client_version_minor` (UInt32) — Minor version of the `clickhouse-client`.
- `client_version_patch` (UInt32) — Patch component of the `clickhouse-client` version.
- `http_method` (UInt8) — HTTP method initiated the query. Possible values:
 - 0 — The query was launched from the TCP interface.
 - 1 — `GET` method is used.
 - 2 — `POST` method is used.
- `http_user_agent` (String) — The `UserAgent` header passed in the HTTP request.
- `quota_key` (String) — The quota key specified in `quotas` setting.
- `revision` (UInt32) — ClickHouse revision.
- `thread_numbers` (Array(UInt32)) — Number of threads that are participating in query execution.
- `ProfileEvents.Names` (Array(String)) — Counters that measure the following metrics:
 - Time spent on reading and writing over the network.
 - Time spent on reading and writing to a disk.
 - Number of network errors.
 - Time spent on waiting when the network bandwidth is limited.
- `ProfileEvents.Values` (Array(UInt64)) — Values of metrics that are listed in the `ProfileEvents.Names` column.
- `Settings.Names` (Array(String)) — Names of settings that were changed when the client run a query. To enable logging of settings changing, set the `log_query_settings` parameter to 1.
- `Settings.Values` (Array(String)) — Values of settings that are listed in the `Settings.Names` column.

Each query creates one or two rows in the `query_log` table, depending on the status of the query:

1. If the query execution is successful, two events with types 1 and 2 are created (see the `type` column).
2. If the error occurred during the query processing, two events with types 1 and 4 are created.
3. If the error occurred before the query launching, a single event with type 3 is created.

By default, logs are added into the table at intervals of 7,5 seconds. You can set this interval in the `query_log` server setting (see the `flush_interval_milliseconds` parameter). To flush the logs forcibly from the memory buffer into the table, use the `SYSTEM FLUSH LOGS` query.

When the table is deleted manually, it will be automatically created on the fly. Note that all the previous logs will be deleted.

Note

The storage period for logs is unlimited; the logs aren't automatically deleted from the table. You need to organize the removing of non-actual logs yourself.

You can specify an arbitrary partitioning key for the `system.query_log` table in the `query_log` server setting (see the `partition_by` parameter).

system.replicas

Contains information and status for replicated tables residing on the local server.

This table can be used for monitoring. The table contains a row for every Replicated* table.

Example:

```
SELECT *  
FROM system.replicas  
WHERE table = 'visits'  
FORMAT Vertical
```

Row 1:

```
database:      merge  
table:         visits  
engine:        ReplicatedCollapsingMergeTree  
is_leader:     1  
is_readonly:   0  
is_session_expired: 0  
future_parts:  1  
parts_to_check: 0  
zookeeper_path: /clickhouse/tables/01-06/visits  
replica_name:   example01-06-1.yandex.ru  
replica_path:   /clickhouse/tables/01-06/visits/replicas/example01-06-1.yandex.ru  
columns_version: 9  
queue_size:    1  
inserts_in_queue: 0  
merges_in_queue: 1  
log_max_index: 596273  
log_pointer:   596274  
total_replicas: 2  
active_replicas: 2
```

Columns:

database:	Database name
table:	Table name
engine:	Table engine name
is_leader:	Whether the replica is the leader.
<p>Only one replica at a time can be the leader. The leader is responsible for selecting background merges to perform. Note that writes can be performed to any replica that is available and has a session in ZK, regardless of whether it is a leader.</p>	
is_readonly:	Whether the replica is in read-only mode.
<p>This mode is turned on if the config doesn't have sections with ZooKeeper, if an unknown error occurred when reinitializing sessions in ZooKeeper, and during session reinitialization in ZooKeeper.</p>	
is_session_expired:	Whether the session with ZooKeeper has expired.
<p>Basically the same as 'is_readonly'.</p>	
future_parts:	The number of data parts that will appear as the result of INSERTs or merges that haven't been done yet.
parts_to_check:	The number of data parts in the queue for verification.
<p>A part is put in the verification queue if there is suspicion that it might be damaged.</p>	
zookeeper_path:	Path to table data in ZooKeeper.
replica_name:	Replica name in ZooKeeper. Different replicas of the same table have different names.
replica_path:	Path to replica data in ZooKeeper. The same as concatenating 'zookeeper_path/replicas/replica_path'.
columns_version:	Version number of the table structure.
<p>Indicates how many times ALTER was performed. If replicas have different versions, it means some replicas haven't made all of the ALTERs yet.</p>	
queue_size:	Size of the queue for operations waiting to be performed.
<p>Operations include inserting blocks of data, merges, and certain other actions. It usually coincides with 'future_parts'.</p>	
inserts_in_queue:	Number of inserts of blocks of data that need to be made.
<p>Insertions are usually replicated fairly quickly. If this number is large, it means something is wrong.</p>	
merges_in_queue:	The number of merges waiting to be made.
<p>Sometimes merges are lengthy, so this value may be greater than zero for a long time.</p>	
<p>The next 4 columns have a non-zero value only where there is an active session with ZK.</p>	
log_max_index:	Maximum entry number in the log of general activity.
log_pointer:	Maximum entry number in the log of general activity that the replica copied to its execution queue, plus one.
<p>If log_pointer is much smaller than log_max_index, something is wrong.</p>	
total_replicas:	The total number of known replicas of this table.
active_replicas:	The number of replicas of this table that have a session in ZooKeeper (i.e., the number of functioning replicas).

If you request all the columns, the table may work a bit slowly, since several reads from ZooKeeper are made for each row.

If you don't request the last 4 columns (log_max_index, log_pointer, total_replicas, active_replicas), the table works quickly.

For example, you can check that everything is working correctly like this:

```

SELECT
  database,
  table,
  is_leader,
  is_readonly,
  is_session_expired,
  future_parts,
  parts_to_check,
  columns_version,
  queue_size,
  inserts_in_queue,
  merges_in_queue,
  log_max_index,
  log_pointer,
  total_replicas,
  active_replicas
FROM system.replicas
WHERE
  is_readonly
  OR is_session_expired
  OR future_parts > 20
  OR parts_to_check > 10
  OR queue_size > 20
  OR inserts_in_queue > 10
  OR log_max_index - log_pointer > 10
  OR total_replicas < 2
  OR active_replicas < total_replicas

```

If this query doesn't return anything, it means that everything is fine.

system.settings

Contains information about settings that are currently in use.

I.e. used for executing the query you are using to read from the system.settings table.

Columns:

```

name String — Setting name.
value String — Setting value.
changed UInt8 — Whether the setting was explicitly defined in the config or explicitly changed.

```

Example:

```

SELECT *
FROM system.settings
WHERE changed

```

name	value	changed
max_threads	8	1
use_uncompressed_cache	0	1
load_balancing	random	1
max_memory_usage	10000000000	1

system.tables

Contains metadata of each table that the server knows about. Detached tables are not shown in [system.tables](#).

This table contains the following columns (the type of the corresponding column is shown in brackets):

- [database](#) (String) — The name of database the table is in.
- [name](#) (String) — Table name.
- [engine](#) (String) — Table engine name (without parameters).
- [is_temporary](#) (UInt8) - Flag that indicates whether the table is temporary.

- `data_path` (String) - Path to the table data in the file system.
- `metadata_path` (String) - Path to the table metadata in the file system.
- `metadata_modification_time` (DateTime) - Time of latest modification of the table metadata.
- `dependencies_database` (Array(String)) - Database dependencies.
- `dependencies_table` (Array(String)) - Table dependencies (`MaterializedView` tables based on the current table).
- `create_table_query` (String) - The query that was used to create the table.
- `engine_full` (String) - Parameters of the table engine.
- `partition_key` (String) - The partition key expression specified in the table.
- `sorting_key` (String) - The sorting key expression specified in the table.
- `primary_key` (String) - The primary key expression specified in the table.
- `sampling_key` (String) - The sampling key expression specified in the table.

The `system.tables` is used in `SHOW TABLES` query implementation.

system.zookeeper

The table does not exist if ZooKeeper is not configured. Allows reading data from the ZooKeeper cluster defined in the config.

The query must have a 'path' equality condition in the WHERE clause. This is the path in ZooKeeper for the children that you want to get data for.

The query `SELECT * FROM system.zookeeper WHERE path = '/clickhouse'` outputs data for all children on the `/clickhouse` node.

To output data for all root nodes, write `path = '/'`.

If the path specified in 'path' doesn't exist, an exception will be thrown.

Columns:

- `name String` — The name of the node.
- `path String` — The path to the node.
- `value String` — Node value.
- `dataLength Int32` — Size of the value.
- `numChildren Int32` — Number of descendants.
- `czxid Int64` — ID of the transaction that created the node.
- `mzxid Int64` — ID of the transaction that last changed the node.
- `pzxid Int64` — ID of the transaction that last deleted or added descendants.
- `ctime DateTime` — Time of node creation.
- `mtime DateTime` — Time of the last modification of the node.
- `version Int32` — Node version: the number of times the node was changed.
- `cversion Int32` — Number of added or removed descendants.
- `aversion Int32` — Number of changes to the ACL.
- `ephemeralOwner Int64` — For ephemeral nodes, the ID of the session that owns this node.

Example:

```
SELECT *
FROM system.zookeeper
WHERE path = '/clickhouse/tables/01-08/visits/replicas'
FORMAT Vertical
```

Row 1:	
name:	example01-08-1.yandex.ru
value:	
czxid:	932998691229
mzxid:	932998691229
ctime:	2015-03-27 16:49:51
mtime:	2015-03-27 16:49:51
version:	0
cversion:	47
aversion:	0
ephemeralOwner:	0
dataLength:	0
numChildren:	7
pzxid:	987021031383
path:	/clickhouse/tables/01-08/visits/replicas
Row 2:	
name:	example01-08-2.yandex.ru
value:	
czxid:	933002738135
mzxid:	933002738135
ctime:	2015-03-27 16:57:01
mtime:	2015-03-27 16:57:01
version:	0
cversion:	37
aversion:	0
ephemeralOwner:	0
dataLength:	0
numChildren:	7
pzxid:	987021252247
path:	/clickhouse/tables/01-08/visits/replicas

system.mutations

The table contains information about **mutations** of MergeTree tables and their progress. Each mutation command is represented by a single row. The table has the following columns:

- database, table** - The name of the database and table to which the mutation was applied.
- mutation_id** - The ID of the mutation. For replicated tables these IDs correspond to znode names in the `<table_path_in_zookeeper>/mutations/` directory in ZooKeeper. For unreplicated tables the IDs correspond to file names in the data directory of the table.
- command** - The mutation command string (the part of the query after `ALTER TABLE [db.]table`).
- create_time** - When this mutation command was submitted for execution.
- block_numbers.partition_id, block_numbers.number** - A Nested column. For mutations of replicated tables contains one record for each partition: the partition ID and the block number that was acquired by the mutation (in each partition only parts that contain blocks with numbers less than the block number acquired by the mutation in that partition will be mutated). Because in non-replicated tables blocks numbers in all partitions form a single sequence, for mutations of non-replicated tables the column will contain one record with a single block number acquired by the mutation.
- parts_to_do** - The number of data parts that need to be mutated for the mutation to finish.
- is_done** - Is the mutation done? Note that even if `parts_to_do = 0` it is possible that a mutation of a replicated table is not done yet because of a long-running INSERT that will create a new data part that will need to be mutated.

If there were problems with mutating some parts the following columns contain additional information:

latest_failed_part - The name of the most recent part that could not be mutated.

latest_fail_time - The time of the most recent part mutation failure.

latest_fail_reason - The exception message that caused the most recent part mutation failure.

Server configuration parameters

This section contains descriptions of server settings that cannot be changed at the session or query level.

These settings are stored in the `config.xml` file on the ClickHouse server.

Other settings are described in the "[Settings](#)" section.

Before studying the settings, read the [Configuration files](#) section and note the use of substitutions (the `incl` and `optional` attributes).

Server settings

`builtin_dictionaries_reload_interval`

The interval in seconds before reloading built-in dictionaries.

ClickHouse reloads built-in dictionaries every x seconds. This makes it possible to edit dictionaries "on the fly" without restarting the server.

Default value: 3600.

Example

```
<builtin_dictionaries_reload_interval>3600</builtin_dictionaries_reload_interval>
```

compression

Data compression settings.

Warning

Don't use it if you have just started using ClickHouse.

The configuration looks like this:

```
<compression>
  <case>
    <parameters/>
  </case>
  ...
</compression>
```

You can configure multiple sections `<case>`.

Block field `<case>`:

- `min_part_size` – The minimum size of a table part.
- `min_part_size_ratio` – The ratio of the minimum size of a table part to the full size of the table.
- `method` – Compression method. Acceptable values : `lz4` or `zstd`(experimental).

ClickHouse checks `min_part_size` and `min_part_size_ratio` and processes the `case` blocks that match these conditions. If none of the `<case>` matches, ClickHouse applies the `lz4` compression algorithm.

Example

```
<compression incl="clickhouse_compression">
  <case>
    <min_part_size>10000000000</min_part_size>
    <min_part_size_ratio>0.01</min_part_size_ratio>
    <method>zstd</method>
  </case>
</compression>
```

default_database

The default database.

To get a list of databases, use the **SHOW DATABASES** query.

Example

```
<default_database>default</default_database>
```

default_profile

Default settings profile.

Settings profiles are located in the file specified in the parameter **user_config**.

Example

```
<default_profile>default</default_profile>
```

dictionaries_config

The path to the config file for external dictionaries.

Path:

- Specify the absolute path or the path relative to the server config file.
- The path can contain wildcards * and ?.

See also "**External dictionaries**".

Example

```
<dictionaries_config>*_dictionary.xml</dictionaries_config>
```

dictionaries_lazy_load

Lazy loading of dictionaries.

If **true**, then each dictionary is created on first use. If dictionary creation failed, the function that was using the dictionary throws an exception.

If **false**, all dictionaries are created when the server starts, and if there is an error, the server shuts down.

The default is **true**.

Example

```
<dictionaries_lazy_load>true</dictionaries_lazy_load>
```

format_schema_path

The path to the directory with the schemas for the input data, such as schemas for the **CapnProto** format.

Example

```
<!-- Directory containing schema files for various input formats. -->
<format_schema_path>format_schemas/</format_schema_path>
```

graphite

Sending data to [Graphite](#).

Settings:

- host – The Graphite server.
- port – The port on the Graphite server.
- interval – The interval for sending, in seconds.
- timeout – The timeout for sending data, in seconds.
- root_path – Prefix for keys.
- metrics – Sending data from a :ref:system_tables-system.metrics table.
- events – Sending data from a :ref:system_tables-system.events table.
- asynchronous_metrics – Sending data from a :ref:system_tables-system.asynchronous_metrics table.

You can configure multiple `<graphite>` clauses. For instance, you can use this for sending different data at different intervals.

Example

```
<graphite>
  <host>localhost</host>
  <port>42000</port>
  <timeout>0.1</timeout>
  <interval>60</interval>
  <root_path>one_min</root_path>
  <metrics>true</metrics>
  <events>true</events>
  <asynchronous_metrics>true</asynchronous_metrics>
</graphite>
```

graphite_rollup

Settings for thinning data for Graphite.

For more details, see [GraphiteMergeTree](#).

Example

```
<graphite_rollup_example>
  <default>
    <function>max</function>
    <retention>
      <age>0</age>
      <precision>60</precision>
    </retention>
    <retention>
      <age>3600</age>
      <precision>300</precision>
    </retention>
    <retention>
      <age>86400</age>
      <precision>3600</precision>
    </retention>
  </default>
</graphite_rollup_example>
```

http_port/https_port

The port for connecting to the server over HTTP(s).

If `https_port` is specified, `openSSL` must be configured.

If `http_port` is specified, the openSSL configuration is ignored even if it is set.

Example

```
<https>0000</https>
```

http_server_default_response

The page that is shown by default when you access the ClickHouse HTTP(s) server.

Example

Opens <https://tabix.io/> when accessing http://localhost: http_port.

```
<http_server_default_response>
<![CDATA[<html ng-app="SMI2"><head><base href="http://ui.tabix.io/"></head><body><div ui-view="" class="content-ui">
</div><script src="http://loader.tabix.io/master.js"></script></body></html>]]>
</http_server_default_response>
```

include_from

The path to the file with substitutions.

For more information, see the section "[Configuration files](#)".

Example

```
<include_from>/etc/metrica.xml</include_from>
```

interserver_http_port

Port for exchanging data between ClickHouse servers.

Example

```
<interserver_http_port>9009</interserver_http_port>
```

interserver_http_host

The host name that can be used by other servers to access this server.

If omitted, it is defined in the same way as the [hostname-f](#) command.

Useful for breaking away from a specific network interface.

Example

```
<interserver_http_host>example.yandex.ru</interserver_http_host>
```

keep_alive_timeout

The number of seconds that ClickHouse waits for incoming requests before closing the connection. Defaults to 3 seconds.

Example

```
<keep_alive_timeout>3</keep_alive_timeout>
```

listen_host

Restriction on hosts that requests can come from. If you want the server to answer all of them, specify [::](#).

Examples:

```
<listen_host>::1</listen_host>
<listen_host>127.0.0.1</listen_host>
```

logger

Logging settings.

Keys:

- level – Logging level. Acceptable values: `trace`, `debug`, `information`, `warning`, `error`.
- log – The log file. Contains all the entries according to `level`.
- errorlog – Error log file.
- size – Size of the file. Applies to `log` and `errorlog`. Once the file reaches `size`, ClickHouse archives and renames it, and creates a new log file in its place.
- count – The number of archived log files that ClickHouse stores.

Example

```
<logger>
  <level>trace</level>
  <log>/var/log/clickhouse-server/clickhouse-server.log</log>
  <errorlog>/var/log/clickhouse-server/clickhouse-server.err.log</errorlog>
  <size>1000M</size>
  <count>10</count>
</logger>
```

Writing to the syslog is also supported. Config example:

```
<logger>
  <use_syslog>1</use_syslog>
  <syslog>
    <address>syslog.remote:10514</address>
    <hostname>myhost.local</hostname>
    <facility>LOG_LOCAL6</facility>
    <format>syslog</format>
  </syslog>
</logger>
```

Keys:

- user_syslog — Required setting if you want to write to the syslog.
- address — The host[:port] of syslogd. If omitted, the local daemon is used.
- hostname — Optional. The name of the host that logs are sent from.
- facility — **The syslog facility keyword** in uppercase letters with the "LOG_" prefix: (`LOG_USER`, `LOG_DAEMON`, `LOG_LOCAL3`, and so on).
Default value: `LOG_USER` if `address` is specified, `LOG_DAEMON` otherwise.
- format – Message format. Possible values: `bsd` and `syslog`.

macros

Parameter substitutions for replicated tables.

Can be omitted if replicated tables are not used.

For more information, see the section "**Creating replicated tables**".

Example

```
<macros incl="macros" optional="true" />
```

mark_cache_size

Approximate size (in bytes) of the cache of "marks" used by **MergeTree**.

The cache is shared for the server and memory is allocated as needed. The cache size must be at least 5368709120.

Example

```
<mark_cache_size>5368709120</mark_cache_size>
```

max_concurrent_queries

The maximum number of simultaneously processed requests.

Example

```
<max_concurrent_queries>100</max_concurrent_queries>
```

max_connections

The maximum number of inbound connections.

Example

```
<max_connections>4096</max_connections>
```

max_open_files

The maximum number of open files.

By default: [maximum](#).

We recommend using this option in Mac OS X, since the [getrlimit\(\)](#) function returns an incorrect value.

Example

```
<max_open_files>262144</max_open_files>
```

max_table_size_to_drop

Restriction on deleting tables.

If the size of a [MergeTree](#) table exceeds [max_table_size_to_drop](#) (in bytes), you can't delete it using a DROP query.

If you still need to delete the table without restarting the ClickHouse server, create the [<clickhouse-path>/flags/force_drop_table](#) file and run the DROP query.

Default value: 50 GB.

The value 0 means that you can delete all tables without any restrictions.

Example

```
<max_table_size_to_drop>0</max_table_size_to_drop>
```

merge_tree

Fine tuning for tables in the [MergeTree](#).

For more information, see the MergeTreeSettings.h header file.

Example

```
<merge_tree>  
  <max_suspicious_broken_parts>5</max_suspicious_broken_parts>  
</merge_tree>
```

openSSL

SSL client/server configuration.

Support for SSL is provided by the [libpoco](#) library. The interface is described in the file [SSLManager.h](#)

Keys for server/client settings:

- `privateKeyFile` – The path to the file with the secret key of the PEM certificate. The file may contain a key and certificate at the same time.
- `certificateFile` – The path to the client/server certificate file in PEM format. You can omit it if `privateKeyFile` contains the certificate.
- `caConfig` – The path to the file or directory that contains trusted root certificates.
- `verificationMode` – The method for checking the node's certificates. Details are in the description of the `Context` class. Possible values: `none`, `relaxed`, `strict`, `once`.
- `verificationDepth` – The maximum length of the verification chain. Verification will fail if the certificate chain length exceeds the set value.
- `loadDefaultCAFile` – Indicates that built-in CA certificates for OpenSSL will be used. Acceptable values: `true`, `false`.
- `cipherList` – Supported OpenSSL encryptions. For example: `ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH`.
- `cacheSessions` – Enables or disables caching sessions. Must be used in combination with `sessionIdContext`. Acceptable values: `true`, `false`.
- `sessionIdContext` – A unique set of random characters that the server appends to each generated identifier. The length of the string must not exceed `SSL_MAX_SSL_SESSION_ID_LENGTH`. This parameter is always recommended, since it helps avoid problems both if the server caches the session and if the client requested caching. Default value: `${application.name}`.
- `sessionCacheSize` – The maximum number of sessions that the server caches. Default value: `1024*20`. `0` – Unlimited sessions.
- `sessionTimeout` – Time for caching the session on the server.
- `extendedVerification` – Automatically extended verification of certificates after the session ends. Acceptable values: `true`, `false`.
- `requireTLSv1` – Require a TLSv1 connection. Acceptable values: `true`, `false`.
- `requireTLSv1_1` – Require a TLSv1.1 connection. Acceptable values: `true`, `false`.
- `requireTLSv1_2` – Require a TLSv1.2 connection. Acceptable values: `true`, `false`.
- `fips` – Activates OpenSSL FIPS mode. Supported if the library's OpenSSL version supports FIPS.
- `privateKeyPassphraseHandler` – Class (`PrivateKeyPassphraseHandler` subclass) that requests the passphrase for accessing the private key. For example: `<privateKeyPassphraseHandler>`, `<name>KeyFileHandler</name>`, `<options><password>test</password></options>`, `</privateKeyPassphraseHandler>`.
- `invalidCertificateHandler` – Class (subclass of `CertificateHandler`) for verifying invalid certificates. For example: `<invalidCertificateHandler>` `<name>ConsoleCertificateHandler</name>` `</invalidCertificateHandler>`.
- `disableProtocols` – Protocols that are not allowed to use.
- `preferServerCiphers` – Preferred server ciphers on the client.

Example of settings:

```

<openssl>
  <server>
    <!-- openssl req -subj "/CN=localhost" -new -newkey rsa:2048 -days 365 -nodes -x509 -keyout /etc/clickhouse-
server/server.key -out /etc/clickhouse-server/server.crt -->
    <certificateFile>/etc/clickhouse-server/server.crt</certificateFile>
    <privateKeyFile>/etc/clickhouse-server/server.key</privateKeyFile>
    <!-- openssl dhparam -out /etc/clickhouse-server/dhparam.pem 4096 -->
    <dhParamsFile>/etc/clickhouse-server/dhparam.pem</dhParamsFile>
    <verificationMode>none</verificationMode>
    <loadDefaultCAFile>true</loadDefaultCAFile>
    <cacheSessions>true</cacheSessions>
    <disableProtocols>ssl2,ssl3</disableProtocols>
    <preferServerCiphers>true</preferServerCiphers>
  </server>
  <client>
    <loadDefaultCAFile>true</loadDefaultCAFile>
    <cacheSessions>true</cacheSessions>
    <disableProtocols>ssl2,ssl3</disableProtocols>
    <preferServerCiphers>true</preferServerCiphers>
    <!-- Use for self-signed: <verificationMode>none</verificationMode> -->
    <invalidCertificateHandler>
      <!-- Use for self-signed: <name>AcceptCertificateHandler</name> -->
      <name>RejectCertificateHandler</name>
    </invalidCertificateHandler>
  </client>
</openssl>

```

part_log

Logging events that are associated with **MergeTree**. For instance, adding or merging data. You can use the log to simulate merge algorithms and compare their characteristics. You can visualize the merge process.

Queries are logged in the **system.part_log** table, not in a separate file. You can configure the name of this table in the **table** parameter (see below).

Use the following parameters to configure logging:

- **database** – Name of the database.
- **table** – Name of the system table.
- **partition_by** – Sets a **custom partitioning key**.
- **flush_interval_milliseconds** – Interval for flushing data from the buffer in memory to the table.

Example

```

<part_log>
  <database>system</database>
  <table>part_log</table>
  <partition_by>toMonday(event_date)</partition_by>
  <flush_interval_milliseconds>7500</flush_interval_milliseconds>
</part_log>

```

path

The path to the directory containing data.

Note

The trailing slash is mandatory.

Example

```

<path>/var/lib/clickhouse/</path>

```

query_log

Setting for logging queries received with the `log_queries=1` setting.

Queries are logged in the `system.query_log` table, not in a separate file. You can change the name of the table in the `table` parameter (see below).

Use the following parameters to configure logging:

- `database` – Name of the database.
- `table` – Name of the system table the queries will be logged in.
- `partition_by` – Sets a `custom partitioning key` for a system table.
- `flush_interval_milliseconds` – Interval for flushing data from the buffer in memory to the table.

If the table doesn't exist, ClickHouse will create it. If the structure of the query log changed when the ClickHouse server was updated, the table with the old structure is renamed, and a new table is created automatically.

Example

```
<query_log>
  <database>system</database>
  <table>query_log</table>
  <partition_by>toMonday(event_date)</partition_by>
  <flush_interval_milliseconds>7500</flush_interval_milliseconds>
</query_log>
```

remote_servers

Configuration of clusters used by the Distributed table engine.

For more information, see the section "[Table engines/Distributed](#)".

Example

```
<remote_servers incl="clickhouse_remote_servers" />
```

For the value of the `incl` attribute, see the section "[Configuration files](#)".

timezone

The server's time zone.

Specified as an IANA identifier for the UTC time zone or geographic location (for example, Africa/Abidjan).

The time zone is necessary for conversions between String and DateTime formats when DateTime fields are output to text format (printed on the screen or in a file), and when getting DateTime from a string. In addition, the time zone is used in functions that work with the time and date if they didn't receive the time zone in the input parameters.

Example

```
<timezone>Europe/Moscow</timezone>
```

tcp_port

Port for communicating with clients over the TCP protocol.

Example

```
<tcp_port>9000</tcp_port>
```

tcp_port_secure

TCP port for secure communication with clients. Use it with [OpenSSL](#) settings.

Possible values

Positive integer.

Default value

```
<tcp_port_secure>9440</tcp_port_secure>
```

tmp_path

Path to temporary data for processing large queries.

Note

The trailing slash is mandatory.

Example

```
<tmp_path>/var/lib/clickhouse/tmp/</tmp_path>
```

uncompressed_cache_size

Cache size (in bytes) for uncompressed data used by table engines from the [MergeTree](#).

There is one shared cache for the server. Memory is allocated on demand. The cache is used if the option [use_uncompressed_cache](#) is enabled.

The uncompressed cache is advantageous for very short queries in individual cases.

Example

```
<uncompressed_cache_size>8589934592</uncompressed_cache_size>
```

user_files_path

The directory with user files. Used in the table function [file\(\)](#).

Example

```
<user_files_path>/var/lib/clickhouse/user_files/</user_files_path>
```

users_config

Path to the file that contains:

- User configurations.
- Access rights.
- Settings profiles.
- Quota settings.

Example

```
<users_config>users.xml</users_config>
```

zookeeper

Contains settings that allow ClickHouse to interact with [ZooKeeper](#) cluster.

ClickHouse uses ZooKeeper for storing metadata of replicas when using replicated tables. If replicated tables are not used, this parameter section can be omitted.

This parameter section contains the following parameters:

- **node** — ZooKeeper endpoint. You can set a few endpoints.

For example:

```
xml <node index="1"> <host>example_host</host> <port>2181</port> </node>
```

The **index** attribute is not used in ClickHouse. The only reason for this attribute is to allow some other programs to use the same configuraton.

- **session_timeout_ms** — Maximum timeout for client session in milliseconds (default: 30000).
- **operation_timeout_ms** — Maximum timeout for operation in milliseconds (default: 10000).
- **root** — ZNode, that is used as root for znodes used by ClickHouse server. Optional.
- **identity** — User and password, required by ZooKeeper to give access to requested znodes. Optional.

Example configuration

```
<zookeeper>
  <node>
    <host>example1</host>
    <port>2181</port>
  </node>
  <node>
    <host>example2</host>
    <port>2181</port>
  </node>
  <session_timeout_ms>30000</session_timeout_ms>
  <operation_timeout_ms>10000</operation_timeout_ms>
  <!-- Optional. Chroot suffix. Should exist. -->
  <root>/path/to/zookeeper/node</root>
  <!-- Optional. Zookeeper digest ACL string. -->
  <identity>user:password</identity>
</zookeeper>
```

See Also

- [Replication](#)
- [ZooKeeper Programmer's Guide](#)

use_minimalistic_part_header_in_zookeeper

Storage method for data part headers in ZooKeeper.

This setting only applies to the **MergeTree** family. It can be specified:

- Globally in the **merge_tree** section of the **config.xml** file.

ClickHouse uses the setting for all the tables on the server. You can change the setting at any time. Existing tables change their behavior when the setting changes.

- For each individual table.

When creating a table, specify the corresponding **engine setting**. The behavior of an existing table with this setting does not change, even if the global setting changes.

Possible values

- 0 — Functionality is turned off.
- 1 — Functionality is turned on.

If **use_minimalistic_part_header_in_zookeeper = 1**, then **replicated** tables store the headers of the data parts compactly using a single **znode**. If the table contains many columns, this storage method significantly reduces the volume of the data stored in Zookeeper.

Attention

After applying `use_minimalistic_part_header_in_zookeeper = 1`, you can't downgrade the ClickHouse server to a version that doesn't support this setting. Be careful when upgrading ClickHouse on servers in a cluster. Don't upgrade all the servers at once. It is safer to test new versions of ClickHouse in a test environment, or on just a few servers of a cluster.

Data part headers already stored with this setting can't be restored to their previous (non-compact) representation.

Default value: 0.

Settings

There are multiple ways to make all the settings described below.

Settings are configured in layers, so each subsequent layer redefines the previous settings.

Ways to configure settings, in order of priority:

- Settings in the `users.xml` server configuration file.

Set in the element `<profiles>`.

- Session settings.

Send `SET setting=value` from the ClickHouse console client in interactive mode.

Similarly, you can use ClickHouse sessions in the HTTP protocol. To do this, you need to specify the `session_id` HTTP parameter.

- Query settings.
 - When starting the ClickHouse console client in non-interactive mode, set the startup parameter `--setting=value`.
 - When using the HTTP API, pass CGI parameters (`URL?setting_1=value&setting_2=value...`).

Settings that can only be made in the server config file are not covered in this section.

Permissions for queries

Queries in ClickHouse can be divided into several types:

1. Read data queries: `SELECT`, `SHOW`, `DESCRIBE`, `EXISTS`.
2. Write data queries: `INSERT`, `OPTIMIZE`.
3. Change settings queries: `SET`, `USE`.
4. DDL queries: `CREATE`, `ALTER`, `RENAME`, `ATTACH`, `DETACH`, `DROP TRUNCATE`.
5. `KILL QUERY`.

The following settings regulate user permissions by the type of query:

- `readonly` — Restricts permissions for all types of queries except DDL queries.
- `allow_ddl` — Restricts permissions for DDL queries.

`KILL QUERY` can be performed with any settings.

readonly

Restricts permissions for read data, write data and change settings queries.

See how the queries are divided into types [above](#).

Possible values

- 0 — All queries are allowed.

- 1 — Only read data queries are allowed.
- 2 — Read data and change settings queries are allowed.

After setting `readonly = 1`, the user can't change `readonly` and `allow_ddl` settings in the current session.

When using the `GET` method in the `HTTP interface`, `readonly = 1` is set automatically. To modify data, use the `POST` method.

Default value

0

allow_ddl

Allows/denies `DDL` queries.

See how the queries are divided into types `above`.

Possible values

- 0 — DDL queries are not allowed.
- 1 — DDL queries are allowed.

You cannot execute `SET allow_ddl = 1` if `allow_ddl = 0` for the current session.

Default value

1

Restrictions on query complexity

Restrictions on query complexity are part of the settings.

They are used in order to provide safer execution from the user interface.

Almost all the restrictions only apply to `SELECT`. For distributed query processing, restrictions are applied on each server separately.

ClickHouse checks the restrictions for data parts, not for each row. It means that you can exceed the value of restriction with a size of the data part.

Restrictions on the "maximum amount of something" can take the value 0, which means "unrestricted".

Most restrictions also have an 'overflow_mode' setting, meaning what to do when the limit is exceeded.

It can take one of two values: `throw` or `break`. Restrictions on aggregation (`group_by_overflow_mode`) also have the value `any`.

`throw` – Throw an exception (default).

`break` – Stop executing the query and return the partial result, as if the source data ran out.

`any (only for group_by_overflow_mode)` – Continuing aggregation for the keys that got into the set, but don't add new keys to the set.

max_memory_usage

The maximum amount of RAM to use for running a query on a single server.

In the default configuration file, the maximum is 10 GB.

The setting doesn't consider the volume of available memory or the total volume of memory on the machine.

The restriction applies to a single query within a single server.

You can use `SHOW PROCESSLIST` to see the current memory consumption for each query.

In addition, the peak memory consumption is tracked for each query and written to the log.

Memory usage is not monitored for the states of certain aggregate functions.

Memory usage is not fully tracked for states of the aggregate functions `min`, `max`, `any`, `anyLast`, `argMin`, `argMax` from `String` and `Array` arguments.

Memory consumption is also restricted by the parameters `max_memory_usage_for_user` and `max_memory_usage_for_all_queries`.

max_memory_usage_for_user

The maximum amount of RAM to use for running a user's queries on a single server.

Default values are defined in `Settings.h`. By default, the amount is not restricted (`max_memory_usage_for_user = 0`).

See also the description of `max_memory_usage`.

max_memory_usage_for_all_queries

The maximum amount of RAM to use for running all queries on a single server.

Default values are defined in `Settings.h`. By default, the amount is not restricted (`max_memory_usage_for_all_queries = 0`).

See also the description of `max_memory_usage`.

max_rows_to_read

The following restrictions can be checked on each block (instead of on each row). That is, the restrictions can be broken a little.

When running a query in multiple threads, the following restrictions apply to each thread separately.

Maximum number of rows that can be read from a table when running a query.

max_bytes_to_read

Maximum number of bytes (uncompressed data) that can be read from a table when running a query.

read_overflow_mode

What to do when the volume of data read exceeds one of the limits: 'throw' or 'break'. By default, throw.

max_rows_to_group_by

Maximum number of unique keys received from aggregation. This setting lets you limit memory consumption when aggregating.

group_by_overflow_mode

What to do when the number of unique keys for aggregation exceeds the limit: 'throw', 'break', or 'any'. By default, throw.

Using the 'any' value lets you run an approximation of GROUP BY. The quality of this approximation depends on the statistical nature of the data.

max_rows_to_sort

Maximum number of rows before sorting. This allows you to limit memory consumption when sorting.

max_bytes_to_sort

Maximum number of bytes before sorting.

sort_overflow_mode

What to do if the number of rows received before sorting exceeds one of the limits: 'throw' or 'break'. By default, throw.

max_result_rows

Limit on the number of rows in the result. Also checked for subqueries, and on remote servers when running parts of a distributed query.

max_result_bytes

Limit on the number of bytes in the result. The same as the previous setting.

result_overflow_mode

What to do if the volume of the result exceeds one of the limits: 'throw' or 'break'. By default, throw. Using 'break' is similar to using LIMIT.

max_execution_time

Maximum query execution time in seconds.

At this time, it is not checked for one of the sorting stages, or when merging and finalizing aggregate functions.

timeout_overflow_mode

What to do if the query is run longer than 'max_execution_time': 'throw' or 'break'. By default, throw.

min_execution_speed

Minimal execution speed in rows per second. Checked on every data block when

'timeout_before_checking_execution_speed' expires. If the execution speed is lower, an exception is thrown.

min_execution_speed_bytes

Minimum number of execution bytes per second. Checked on every data block when

'timeout_before_checking_execution_speed' expires. If the execution speed is lower, an exception is thrown.

max_execution_speed

Maximum number of execution rows per second. Checked on every data block when

'timeout_before_checking_execution_speed' expires. If the execution speed is high, the execution speed will be reduced.

max_execution_speed_bytes

Maximum number of execution bytes per second. Checked on every data block when

'timeout_before_checking_execution_speed' expires. If the execution speed is high, the execution speed will be reduced.

timeout_before_checking_execution_speed

Checks that execution speed is not too slow (no less than 'min_execution_speed'), after the specified time in seconds has expired.

max_columns_to_read

Maximum number of columns that can be read from a table in a single query. If a query requires reading a greater number of columns, it throws an exception.

max_temporary_columns

Maximum number of temporary columns that must be kept in RAM at the same time when running a query, including constant columns. If there are more temporary columns than this, it throws an exception.

max_temporary_non_const_columns

The same thing as 'max_temporary_columns', but without counting constant columns.

Note that constant columns are formed fairly often when running a query, but they require approximately zero computing resources.

max_subquery_depth

Maximum nesting depth of subqueries. If subqueries are deeper, an exception is thrown. By default, 100.

max_pipeline_depth

Maximum pipeline depth. Corresponds to the number of transformations that each data block goes through during query processing. Counted within the limits of a single server. If the pipeline depth is greater, an exception is thrown. By default, 1000.

max_ast_depth

Maximum nesting depth of a query syntactic tree. If exceeded, an exception is thrown.

At this time, it isn't checked during parsing, but only after parsing the query. That is, a syntactic tree that is too deep can be created during parsing, but the query will fail. By default, 1000.

max_ast_elements

Maximum number of elements in a query syntactic tree. If exceeded, an exception is thrown.

In the same way as the previous setting, it is checked only after parsing the query. By default, 50,000.

max_rows_in_set

Maximum number of rows for a data set in the IN clause created from a subquery.

max_bytes_in_set

Maximum number of bytes (uncompressed data) used by a set in the IN clause created from a subquery.

set_overflow_mode

What to do when the amount of data exceeds one of the limits: 'throw' or 'break'. By default, throw.

max_rows_in_distinct

Maximum number of different rows when using DISTINCT.

max_bytes_in_distinct

Maximum number of bytes used by a hash table when using DISTINCT.

distinct_overflow_mode

What to do when the amount of data exceeds one of the limits: 'throw' or 'break'. By default, throw.

max_rows_to_transfer

Maximum number of rows that can be passed to a remote server or saved in a temporary table when using GLOBAL IN.

max_bytes_to_transfer

Maximum number of bytes (uncompressed data) that can be passed to a remote server or saved in a temporary table when using GLOBAL IN.

transfer_overflow_mode

What to do when the amount of data exceeds one of the limits: 'throw' or 'break'. By default, throw.

max_partitions_per_insert_block

Limits the maximum number of partitions in a single inserted block.

Possible values:

- Positive integer.
- 0 — Unlimited number of partitions.

Default value: 100.

Details

When inserting data ClickHouse calculates the number of partitions in the inserted block, and if the number of partitions is more than `max_partitions_per_insert_block` then ClickHouse throws an exception with the following text:

"Too many partitions for single INSERT block (more than " + toString(max_parts) + "). The limit is controlled by 'max_partitions_per_insert_block' setting. Large number of partitions is a common misconception. It will lead to severe negative performance impact, including slow server startup, slow INSERT queries and slow SELECT queries. Recommended total number of partitions for a table is under 1000..10000. Please note, that partitioning is not intended to speed up SELECT queries (ORDER BY key is sufficient to make range queries fast). Partitions are intended for data manipulation (DROP PARTITION, etc)."

Settings

distributed_product_mode

Changes the behavior of `distributed subqueries`.

ClickHouse applies this setting when the query contains the product of distributed tables, i.e. when the query for a distributed table contains a non-GLOBAL subquery for the distributed table.

Restrictions:

- Only applied for IN and JOIN subqueries.
- Only if the FROM section uses a distributed table containing more than one shard.
- If the subquery concerns a distributed table containing more than one shard,
- Not used for a table-valued `remote` function.

Possible values:

- `deny` — Default value. Prohibits using these types of subqueries (returns the "Double-distributed in/JOIN subqueries is denied" exception).
- `local` — Replaces the database and table in the subquery with local ones for the destination server (shard), leaving the normal `IN/JOIN`.
- `global` — Replaces the `IN/JOIN` query with `GLOBAL IN/GLOBAL JOIN`.
- `allow` — Allows the use of these types of subqueries.

enable_optimize_predicate_expression

Turns on predicate pushdown in `SELECT` queries.

Predicate pushdown may significantly reduce network traffic for distributed queries.

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 0.

Usage

Consider the following queries:

1. `SELECT count() FROM test_table WHERE date = '2018-10-10'`
2. `SELECT count() FROM (SELECT * FROM test_table) WHERE date = '2018-10-10'`

If `enable_optimize_predicate_expression = 1`, then the execution time of these queries is equal, because ClickHouse applies `WHERE` to the subquery when processing it.

If `enable_optimize_predicate_expression = 0`, then the execution time of the second query is much longer, because the `WHERE` clause applies to all the data after the subquery finishes.

fallback_to_stale_replicas_for_distributed_queries

Forces a query to an out-of-date replica if updated data is not available. See "[Replication](#)".

ClickHouse selects the most relevant from the outdated replicas of the table.

Used when performing `SELECT` from a distributed table that points to replicated tables.

By default, 1 (enabled).

force_index_by_date

Disables query execution if the index can't be used by date.

Works with tables in the MergeTree family.

If `force_index_by_date=1`, ClickHouse checks whether the query has a date key condition that can be used for restricting data ranges. If there is no suitable condition, it throws an exception. However, it does not check whether the condition actually reduces the amount of data to read. For example, the condition `Date != '2000-01-01'` is acceptable even when it matches all the data in the table (i.e., running the query requires a full scan). For more information about ranges of data in MergeTree tables, see "[MergeTree](#)".

force_primary_key

Disables query execution if indexing by the primary key is not possible.

Works with tables in the MergeTree family.

If `force_primary_key=1`, ClickHouse checks to see if the query has a primary key condition that can be used for restricting data ranges. If there is no suitable condition, it throws an exception. However, it does not check whether the condition actually reduces the amount of data to read. For more information about data ranges in MergeTree tables, see "[MergeTree](#)".

fsync_metadata

Enables or disables `fsync` when writing `.sql` files. Enabled by default.

It makes sense to disable it if the server has millions of tiny table chunks that are constantly being created and destroyed.

enable_http_compression

Enables or disables data compression in the response to an HTTP request.

For more information, read the [HTTP interface description](#).

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 0.

http_zlib_compression_level

Sets the level of data compression in the response to an HTTP request if `enable_http_compression = 1`.

Possible values: Numbers from 1 to 9.

Default value: 3.

http_native_compression_disable_checksumming_on_decompress

Enables or disables checksum verification when decompressing the HTTP POST data from the client. Used only for ClickHouse native compression format (not used with `gzip` or `deflate`).

For more information, read the [HTTP interface description](#).

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 0.

input_format_allow_errors_num

Sets the maximum number of acceptable errors when reading from text formats (CSV, TSV, etc.).

The default value is 0.

Always pair it with `input_format_allow_errors_ratio`. To skip errors, both settings must be greater than 0.

If an error occurred while reading rows but the error counter is still less than `input_format_allow_errors_num`, ClickHouse ignores the row and moves on to the next one.

If `input_format_allow_errors_num` is exceeded, ClickHouse throws an exception.

input_format_allow_errors_ratio

Sets the maximum percentage of errors allowed when reading from text formats (CSV, TSV, etc.).

The percentage of errors is set as a floating-point number between 0 and 1.

The default value is 0.

Always pair it with `input_format_allow_errors_num`. To skip errors, both settings must be greater than 0.

If an error occurred while reading rows but the error counter is still less than `input_format_allow_errors_ratio`, ClickHouse ignores the row and moves on to the next one.

If `input_format_allow_errors_ratio` is exceeded, ClickHouse throws an exception.

input_format_values_interpret_expressions

Enables or disables the full SQL parser if the fast stream parser can't parse the data. This setting is used only for the `Values` format at the data insertion. For more information about syntax parsing, see the [Syntax](#) section.

Possible values:

- 0 — Disabled.

In this case, you must provide formatted data. See the [Formats](#) section.

- 1 — Enabled.

In this case, you can use an SQL expression as a value, but data insertion is much slower this way. If you insert only formatted data, then ClickHouse behaves as if the setting value is 0.

Default value: 1.

Example of Use

Insert the **DateTime** type value with the different settings.

```
SET input_format_values_interpret_expressions = 0;  
INSERT INTO datetime_t VALUES (now())
```

Exception on client:

Code: 27. DB::Exception: Cannot parse input: expected) before: now(): (at row 1)

```
SET input_format_values_interpret_expressions = 1;  
INSERT INTO datetime_t VALUES (now())
```

Ok.

The last query is equivalent to the following:

```
SET input_format_values_interpret_expressions = 0;  
INSERT INTO datetime_t SELECT now()
```

Ok.

input_format_defaults_for_omitted_fields

Turns on/off the extended data exchange between a ClickHouse client and a ClickHouse server. This setting applies for **INSERT** queries.

When executing the **INSERT** query, the ClickHouse client prepares data and sends it to the server for writing. The client gets the table structure from the server when preparing the data. In some cases, the client needs more information than the server sends by default. Turn on the extended data exchange with **input_format_defaults_for_omitted_fields = 1**.

When the extended data exchange is enabled, the server sends the additional metadata along with the table structure. The composition of the metadata depends on the operation.

Operations where you may need the extended data exchange enabled:

- Inserting data in **JSONEachRow** format.

For all other operations, ClickHouse doesn't apply the setting.

Note

The extended data exchange functionality consumes additional computing resources on the server and can reduce performance.

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 0.

input_format_skip_unknown_fields

Enables or disables skipping of insertion of extra data.

When writing data, ClickHouse throws an exception if input data contain columns that do not exist in the target table. If skipping is enabled, ClickHouse doesn't insert extra data and doesn't throw an exception.

Supported formats: **JSONEachRow**, **CSVWithNames**, **TabSeparatedWithNames**, **TSKV**.

Possible values:

- 0 — Disabled.

- 1 — Enabled.

Default value: 0.

input_format_with_names_use_header

Enables or disables checking the column order when inserting data.

We recommend disabling check, if you are sure that the column order of the input data is the same as in the target table. It increases ClickHouse performance.

Supported formats: [CSVWithNames](#), [TabSeparatedWithNames](#).

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 1.

join_default_strictness

Sets default strictness for [JOIN clauses](#).

Possible values

- [ALL](#) — If the right table has several matching rows, the data is multiplied by the number of these rows. This is the normal [JOIN](#) behavior from standard SQL.
- [ANY](#) — If the right table has several matching rows, only the first one found is joined. If the right table has only one matching row, the results of [ANY](#) and [ALL](#) are the same.
- [Empty string](#) — If [ALL](#) or [ANY](#) is not specified in the query, ClickHouse throws an exception.

Default value: [ALL](#)

join_use_nulls

Sets the type of [JOIN](#) behavior. When merging tables, empty cells may appear. ClickHouse fills them differently based on this setting.

Possible values

- 0 — The empty cells are filled with the default value of the corresponding field type.
- 1 — [JOIN](#) behaves the same way as in standard SQL. The type of the corresponding field is converted to [Nullable](#), and empty cells are filled with [NULL](#).

Default value: 0.

max_block_size

In ClickHouse, data is processed by blocks (sets of column parts). The internal processing cycles for a single block are efficient enough, but there are noticeable expenditures on each block. The [max_block_size](#) setting is a recommendation for what size of block (in number of rows) to load from tables. The block size shouldn't be too small, so that the expenditures on each block are still noticeable, but not too large, so that the query with LIMIT that is completed after the first block is processed quickly. The goal is to avoid consuming too much memory when extracting a large number of columns in multiple threads, and to preserve at least some cache locality.

Default value: 65,536.

Blocks the size of [max_block_size](#) are not always loaded from the table. If it is obvious that less data needs to be retrieved, a smaller block is processed.

preferred_block_size_bytes

Used for the same purpose as `max_block_size`, but it sets the recommended block size in bytes by adapting it to the number of rows in the block.

However, the block size cannot be more than `max_block_size` rows.

By default: 1,000,000. It only works when reading from MergeTree engines.

merge_tree_uniform_read_distribution

ClickHouse uses multiple threads when reading from MergeTree* tables. This setting turns on/off the uniform distribution of reading tasks over the working threads. The algorithm of the uniform distribution aims to make execution time for all the threads approximately equal in a `SELECT` query.

Possible values

- 0 — Do not use uniform read distribution.
- 1 — Use uniform read distribution.

Default value: 1.

merge_tree_min_rows_for_concurrent_read

If the number of rows to be read from a file of a MergeTree* table exceeds `merge_tree_min_rows_for_concurrent_read` then ClickHouse tries to perform a concurrent reading from this file on several threads.

Possible values

Any positive integer.

Default value: 163840.

merge_tree_min_rows_for_seek

If the distance between two data blocks to be read in one file is less than `merge_tree_min_rows_for_seek` rows, then ClickHouse does not seek through the file, but reads the data sequentially.

Possible values

Any positive integer.

Default value: 0.

merge_tree_coarse_index_granularity

When searching data, ClickHouse checks the data marks in the index file. If ClickHouse finds that required keys are in some range, it divides this range into `merge_tree_coarse_index_granularity` subranges and searches the required keys there recursively.

Possible values

Any positive even integer.

Default value: 8.

merge_tree_max_rows_to_use_cache

If ClickHouse should read more than `merge_tree_max_rows_to_use_cache` rows in one query, it does not use the cache of uncompressed blocks. The `uncompressed_cache_size` server setting defines the size of the cache of uncompressed blocks.

Possible values

Any positive integer.

Default value: 1048576.

min_bytes_to_use_direct_io

The minimum data volume required for using direct I/O access to the storage disk.

ClickHouse uses this setting when reading data from tables. If the total storage volume of all the data to be read exceeds `min_bytes_to_use_direct_io` bytes, then ClickHouse reads the data from the storage disk with the `O_DIRECT` option.

Possible values

- 0 — Direct I/O is disabled.
- Positive integer.

Default value: 0.

log_queries

Setting up query logging.

Queries sent to ClickHouse with this setup are logged according to the rules in the `query_log` server configuration parameter.

Example:

```
log_queries=1
```

max_insert_block_size

The size of blocks to form for insertion into a table.

This setting only applies in cases when the server forms the blocks.

For example, for an INSERT via the HTTP interface, the server parses the data format and forms blocks of the specified size.

But when using clickhouse-client, the client parses the data itself, and the 'max_insert_block_size' setting on the server doesn't affect the size of the inserted blocks.

The setting also doesn't have a purpose when using INSERT SELECT, since data is inserted using the same blocks that are formed after SELECT.

Default value: 1,048,576.

The default is slightly more than `max_block_size`. The reason for this is because certain table engines (`*MergeTree`) form a data part on the disk for each inserted block, which is a fairly large entity. Similarly, `*MergeTree` tables sort data during insertion, and a large enough block size allows sorting more data in RAM.

max_replica_delay_for_distributed_queries

Disables lagging replicas for distributed queries. See "`Replication`".

Sets the time in seconds. If a replica lags more than the set value, this replica is not used.

Default value: 300.

Used when performing `SELECT` from a distributed table that points to replicated tables.

max_threads

The maximum number of query processing threads, excluding threads for retrieving data from remote servers (see the 'max_distributed_connections' parameter).

This parameter applies to threads that perform the same stages of the query processing pipeline in parallel. For example, when reading from a table, if it is possible to evaluate expressions with functions, filter with WHERE and pre-aggregate for GROUP BY in parallel using at least 'max_threads' number of threads, then 'max_threads' are used.

Default value: 2.

If less than one SELECT query is normally run on a server at a time, set this parameter to a value slightly less than the actual number of processor cores.

For queries that are completed quickly because of a LIMIT, you can set a lower 'max_threads'. For example, if the necessary number of entries are located in every block and max_threads = 8, then 8 blocks are retrieved, although it would have been enough to read just one.

The smaller the max_threads value, the less memory is consumed.

max_compress_block_size

The maximum size of blocks of uncompressed data before compressing for writing to a table. By default, 1,048,576 (1 MiB). If the size is reduced, the compression rate is significantly reduced, the compression and decompression speed increases slightly due to cache locality, and memory consumption is reduced. There usually isn't any reason to change this setting.

Don't confuse blocks for compression (a chunk of memory consisting of bytes) with blocks for query processing (a set of rows from a table).

min_compress_block_size

For MergeTree" tables. In order to reduce latency when processing queries, a block is compressed when writing the next mark if its size is at least 'min_compress_block_size'. By default, 65,536.

The actual size of the block, if the uncompressed data is less than 'max_compress_block_size', is no less than this value and no less than the volume of data for one mark.

Let's look at an example. Assume that 'index_granularity' was set to 8192 during table creation.

We are writing a UInt32-type column (4 bytes per value). When writing 8192 rows, the total will be 32 KB of data. Since min_compress_block_size = 65,536, a compressed block will be formed for every two marks.

We are writing a URL column with the String type (average size of 60 bytes per value). When writing 8192 rows, the average will be slightly less than 500 KB of data. Since this is more than 65,536, a compressed block will be formed for each mark. In this case, when reading data from the disk in the range of a single mark, extra data won't be decompressed.

There usually isn't any reason to change this setting.

max_query_size

The maximum part of a query that can be taken to RAM for parsing with the SQL parser.

The INSERT query also contains data for INSERT that is processed by a separate stream parser (that consumes O(1) RAM), which is not included in this restriction.

Default value: 256 KiB.

interactive_delay

The interval in microseconds for checking whether request execution has been canceled and sending the progress.

Default value: 100,000 (checks for canceling and sends the progress ten times per second).

connect_timeout, receive_timeout, send_timeout

Timeouts in seconds on the socket used for communicating with the client.

Default value: 10, 300, 300.

poll_interval

Lock in a wait loop for the specified number of seconds.

Default value: 10.

max_distributed_connections

The maximum number of simultaneous connections with remote servers for distributed processing of a single query to a single Distributed table. We recommend setting a value no less than the number of servers in the cluster.

Default value: 1024.

The following parameters are only used when creating Distributed tables (and when launching a server), so there is no reason to change them at runtime.

distributed_connections_pool_size

The maximum number of simultaneous connections with remote servers for distributed processing of all queries to a single Distributed table. We recommend setting a value no less than the number of servers in the cluster.

Default value: 1024.

connect_timeout_with_failover_ms

The timeout in milliseconds for connecting to a remote server for a Distributed table engine, if the 'shard' and 'replica' sections are used in the cluster definition.

If unsuccessful, several attempts are made to connect to various replicas.

Default value: 50.

connections_with_failover_max_tries

The maximum number of connection attempts with each replica for the Distributed table engine.

Default value: 3.

extremes

Whether to count extreme values (the minimums and maximums in columns of a query result). Accepts 0 or 1. By default, 0 (disabled).

For more information, see the section "Extreme values".

use_uncompressed_cache

Whether to use a cache of uncompressed blocks. Accepts 0 or 1. By default, 0 (disabled).

Using the uncompressed cache (only for tables in the MergeTree family) can significantly reduce latency and increase throughput when working with a large number of short queries. Enable this setting for users who send frequent short requests. Also pay attention to the [uncompressed_cache_size](#) configuration parameter (only set in the config file) – the size of uncompressed cache blocks. By default, it is 8 GiB. The uncompressed cache is filled in as needed and the least-used data is automatically deleted.

For queries that read at least a somewhat large volume of data (one million rows or more), the uncompressed cache is disabled automatically in order to save space for truly small queries. This means that you can keep the 'use_uncompressed_cache' setting always set to 1.

replace_running_query

When using the HTTP interface, the 'query_id' parameter can be passed. This is any string that serves as the query identifier.

If a query from the same user with the same 'query_id' already exists at this time, the behavior depends on the 'replace_running_query' parameter.

0 (default) – Throw an exception (don't allow the query to run if a query with the same 'query_id' is already running).

1 – Cancel the old query and start running the new one.

Yandex.Metrica uses this parameter set to 1 for implementing suggestions for segmentation conditions. After entering the next character, if the old query hasn't finished yet, it should be canceled.

schema

This parameter is useful when you are using formats that require a schema definition, such as [Cap'n Proto](#). The value depends on the format.

stream_flush_interval_ms

Works for tables with streaming in the case of a timeout, or when a thread generates [max_insert_block_size](#) rows.

The default value is 7500.

The smaller the value, the more often data is flushed into the table. Setting the value too low leads to poor performance.

load_balancing

Specifies the algorithm of replicas selection that is used for distributed query processing.

ClickHouse supports the following algorithms of choosing replicas:

- [Random](#) (by default)
- [Nearest hostname](#)
- [In order](#)
- [First or random](#)

Random (by default)

```
load_balancing = random
```

The number of errors is counted for each replica. The query is sent to the replica with the fewest errors, and if there are several of these, to any one of them.

Disadvantages: Server proximity is not accounted for; if the replicas have different data, you will also get different data.

Nearest Hostname

```
load_balancing = nearest_hostname
```

The number of errors is counted for each replica. Every 5 minutes, the number of errors is integrally divided by 2. Thus, the number of errors is calculated for a recent time with exponential smoothing. If there is one replica with a minimal number of errors (i.e. errors occurred recently on the other replicas), the query is sent to it. If there are multiple replicas with the same minimal number of errors, the query is sent to the replica with a host name that is most similar to the server's host name in the config file (for the number of different characters in identical positions, up to the minimum length of both host names).

For instance, example01-01-1 and example01-01-2.yandex.ru are different in one position, while example01-01-1 and example01-02-2 differ in two places.

This method might seem primitive, but it doesn't require external data about network topology, and it doesn't compare IP addresses, which would be complicated for our IPv6 addresses.

Thus, if there are equivalent replicas, the closest one by name is preferred.

We can also assume that when sending a query to the same server, in the absence of failures, a distributed query will also go to the same servers. So even if different data is placed on the replicas, the query will return mostly the same results.

In Order

```
load_balancing = in_order
```

Replicas with the same number of errors are accessed in the same order as they are specified in configuration. This method is appropriate when you know exactly which replica is preferable.

First or Random

```
load_balancing = first_or_random
```

This algorithm chooses the first replica in order or a random replica if the first one is unavailable. It is effective in cross-replication topology setups, but it is useless in other configurations.

The **first or random** algorithm solves the problem of the **in order** algorithm. The problem is: if one replica goes down, the next one handles twice the usual load while remaining ones handle usual traffic. When using the **first or random** algorithm, the load on replicas is leveled.

prefer_localhost_replica

Enables/disables preferable using the localhost replica when processing distributed queries.

Possible values:

- 1 — ClickHouse always sends a query to the localhost replica if it exists.
- 0 — ClickHouse uses the balancing strategy specified by the **load_balancing** setting.

Default value: 1.

totals_mode

How to calculate TOTALS when HAVING is present, as well as when max_rows_to_group_by and group_by_overflow_mode = 'any' are present. See the section "WITH TOTALS modifier".

totals_auto_threshold

The threshold for **totals_mode = 'auto'**. See the section "WITH TOTALS modifier".

max_parallel_replicas

The maximum number of replicas for each shard when executing a query. For consistency (to get different parts of the same data split), this option only works when the sampling key is set. Replica lag is not controlled.

compile

Enable compilation of queries. By default, 0 (disabled).

Compilation is only used for part of the query-processing pipeline: for the first stage of aggregation (GROUP BY). If this portion of the pipeline was compiled, the query may run faster due to deployment of short cycles and inlining aggregate function calls. The maximum performance improvement (up to four times faster in rare cases) is seen for queries with multiple simple aggregate functions. Typically, the performance gain is insignificant. In very rare cases, it may slow down query execution.

min_count_to_compile

How many times to potentially use a compiled chunk of code before running compilation. By default, 3. For testing, the value can be set to 0: compilation runs synchronously and the query waits for the end of the compilation process before continuing execution. For all other cases, use values starting with 1. Compilation normally takes about 5-10 seconds. If the value is 1 or more, compilation occurs asynchronously in a separate thread. The result will be used as soon as it is ready, including queries that are currently running.

Compiled code is required for each different combination of aggregate functions used in the query and the type of keys in the GROUP BY clause.

The results of compilation are saved in the build directory in the form of .so files. There is no restriction on the number of compilation results, since they don't use very much space. Old results will be used after server restarts, except in the case of a server upgrade – in this case, the old results are deleted.

output_format_json_quote_64bit_integers

If the value is true, integers appear in quotes when using JSON* Int64 and UInt64 formats (for compatibility with most JavaScript implementations); otherwise, integers are output without the quotes.

format_csv_delimiter

The character interpreted as a delimiter in the CSV data. By default, the delimiter is ,.

insert_quorum

Enables quorum writes.

- If `insert_quorum < 2`, the quorum writes are disabled.
- If `insert_quorum >= 2`, the quorum writes are enabled.

Default value: 0.

Quorum writes

INSERT succeeds only when ClickHouse manages to correctly write data to the `insert_quorum` of replicas during the `insert_quorum_timeout`. If for any reason the number of replicas with successful writes does not reach the `insert_quorum`, the write is considered failed and ClickHouse will delete the inserted block from all the replicas where data has already been written.

All the replicas in the quorum are consistent, i.e., they contain data from all previous **INSERT** queries. The **INSERT** sequence is linearized.

When reading the data written from the `insert_quorum`, you can use the `select_sequential_consistency` option.

ClickHouse generates an exception

- If the number of available replicas at the time of the query is less than the `insert_quorum`.
- At an attempt to write data when the previous block has not yet been inserted in the `insert_quorum` of replicas. This situation may occur if the user tries to perform an **INSERT** before the previous one with the `insert_quorum` is completed.

See also the following parameters:

- `insert_quorum_timeout`
- `select_sequential_consistency`

insert_quorum_timeout

Quorum write timeout in seconds. If the timeout has passed and no write has taken place yet, ClickHouse will generate an exception and the client must repeat the query to write the same block to the same or any other replica.

Default value: 60 seconds.

See also the following parameters:

- [insert_quorum](#)
- [select_sequential_consistency](#)

select_sequential_consistency

Enables or disables sequential consistency for **SELECT** queries:

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 0.

Usage

When sequential consistency is enabled, ClickHouse allows the client to execute the **SELECT** query only for those replicas that contain data from all previous **INSERT** queries executed with [insert_quorum](#). If the client refers to a partial replica, ClickHouse will generate an exception. The **SELECT** query will not include data that has not yet been written to the quorum of replicas.

See Also

- [insert_quorum](#)
- [insert_quorum_timeout](#)

allow_experimental_cross_to_join_conversion

Enables or disables:

1. Rewriting of queries with multiple **JOIN clauses** from the syntax with commas to the **JOIN ON/USING** syntax. If the setting value is 0, ClickHouse doesn't process queries with the syntax with commas, and throws an exception.
2. Converting of **CROSS JOIN** into **INNER JOIN** if conditions of join allow it.

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 1.

Settings profiles

A settings profile is a collection of settings grouped under the same name. Each ClickHouse user has a profile. To apply all the settings in a profile, set the **profile** setting.

Example:

Install the **web** profile.

```
SET profile = 'web'
```

Settings profiles are declared in the user config file. This is usually **users.xml**.

Example:

```

<!-- Settings profiles -->
<profiles>
  <!-- Default settings -->
  <default>
    <!-- The maximum number of threads when running a single query. -->
    <max_threads>8</max_threads>
  </default>

  <!-- Settings for queries from the user interface -->
  <web>
    <max_rows_to_read>1000000000</max_rows_to_read>
    <max_bytes_to_read>100000000000</max_bytes_to_read>

    <max_rows_to_group_by>1000000</max_rows_to_group_by>
    <group_by_overflow_mode>any</group_by_overflow_mode>

    <max_rows_to_sort>1000000</max_rows_to_sort>
    <max_bytes_to_sort>1000000000</max_bytes_to_sort>

    <max_result_rows>100000</max_result_rows>
    <max_result_bytes>1000000000</max_result_bytes>
    <result_overflow_mode>break</result_overflow_mode>

    <max_execution_time>600</max_execution_time>
    <min_execution_speed>1000000</min_execution_speed>
    <timeout_before_checking_execution_speed>15</timeout_before_checking_execution_speed>

    <max_columns_to_read>25</max_columns_to_read>
    <max_temporary_columns>100</max_temporary_columns>
    <max_temporary_non_const_columns>50</max_temporary_non_const_columns>

    <max_subquery_depth>2</max_subquery_depth>
    <max_pipeline_depth>25</max_pipeline_depth>
    <max_ast_depth>50</max_ast_depth>
    <max_ast_elements>100</max_ast_elements>

    <readonly>1</readonly>
  </web>
</profiles>

```

The example specifies two profiles: **default** and **web**. The **default** profile has a special purpose: it must always be present and is applied when starting the server. In other words, the **default** profile contains default settings. The **web** profile is a regular profile that can be set using the **SET** query or using a URL parameter in an HTTP query.

Settings profiles can inherit from each other. To use inheritance, indicate the **profile** setting before the other settings that are listed in the profile.

ClickHouse Utility

- **clickhouse-local** — Allows running SQL queries on data without stopping the ClickHouse server, similar to how **awk** does this.
- **clickhouse-copier** — Copies (and reshard) data from one cluster to another cluster.

clickhouse-copier

Copies data from the tables in one cluster to tables in another (or the same) cluster.

You can run multiple **clickhouse-copier** instances on different servers to perform the same job. ZooKeeper is used for syncing the processes.

After starting, **clickhouse-copier**:

- Connects to ZooKeeper and receives:
 - Copying jobs.
 - The state of the copying jobs.
- It performs the jobs.

Each running process chooses the "closest" shard of the source cluster and copies the data into the destination cluster, resharding the data if necessary.

`clickhouse-copier` tracks the changes in ZooKeeper and applies them on the fly.

To reduce network traffic, we recommend running `clickhouse-copier` on the same server where the source data is located.

Running clickhouse-copier

The utility should be run manually:

```
clickhouse-copier copier --daemon --config zookeeper.xml --task-path /task/path --base-dir /path/to/dir
```

Parameters:

- `daemon` — Starts `clickhouse-copier` in daemon mode.
- `config` — The path to the `zookeeper.xml` file with the parameters for the connection to ZooKeeper.
- `task-path` — The path to the ZooKeeper node. This node is used for syncing `clickhouse-copier` processes and storing tasks. Tasks are stored in `$task-path/description`.
- `task-file` — Optional path to file with task configuration for initial upload to ZooKeeper.
- `task-upload-force` — Force upload `task-file` even if node already exists.
- `base-dir` — The path to logs and auxiliary files. When it starts, `clickhouse-copier` creates `clickhouse-copier_YYYYMMHHSS_<PID>` subdirectories in `$base-dir`. If this parameter is omitted, the directories are created in the directory where `clickhouse-copier` was launched.

Format of zookeeper.xml

```
<yandex>
  <logger>
    <level>trace</level>
    <size>100M</size>
    <count>3</count>
  </logger>

  <zookeeper>
    <node index="1">
      <host>127.0.0.1</host>
      <port>2181</port>
    </node>
  </zookeeper>
</yandex>
```

Configuration of copying tasks

```
<yandex>
  <!-- Configuration of clusters as in an ordinary server config -->
  <remote_servers>
    <source_cluster>
      <shard>
        <internal_replication>false</internal_replication>
        <replica>
          <host>127.0.0.1</host>
          <port>9000</port>
        </replica>
      </shard>
      ...
    </source_cluster>
```

```

<destination_cluster>
...
</destination_cluster>
</remote_servers>

<!-- How many simultaneously active workers are possible. If you run more workers superfluous workers will sleep. -->
<max_workers>2</max_workers>

<!-- Setting used to fetch (pull) data from source cluster tables -->
<settings_pull>
  <readonly>1</readonly>
</settings_pull>

<!-- Setting used to insert (push) data to destination cluster tables -->
<settings_push>
  <readonly>0</readonly>
</settings_push>

<!-- Common setting for fetch (pull) and insert (push) operations. Also, copier process context uses it.
They are overlaid by <settings_pull/> and <settings_push/> respectively. -->
<settings>
  <connect_timeout>3</connect_timeout>
  <!-- Sync insert is set forcibly, leave it here just in case. -->
  <insert_distributed_sync>1</insert_distributed_sync>
</settings>

<!-- Copying tasks description.
You could specify several table task in the same task description (in the same ZooKeeper node), they will be performed
sequentially.
-->
<tables>
  <!-- A table task, copies one table. -->
  <table_hits>
    <!-- Source cluster name (from <remote_servers/> section) and tables in it that should be copied -->
    <cluster_pull>source_cluster</cluster_pull>
    <database_pull>test</database_pull>
    <table_pull>hits</table_pull>

    <!-- Destination cluster name and tables in which the data should be inserted -->
    <cluster_push>destination_cluster</cluster_push>
    <database_push>test</database_push>
    <table_push>hits2</table_push>

    <!-- Engine of destination tables.
    If destination tables have not be created, workers create them using columns definition from source tables and engine
    definition from here.

    NOTE: If the first worker starts insert data and detects that destination partition is not empty then the partition will
    be dropped and refilled, take it into account if you already have some data in destination tables. You could directly
    specify partitions that should be copied in <enabled_partitions/>, they should be in quoted format like partition column
    of
    system.parts table.
    -->
    <engine>
    ENGINE=ReplicatedMergeTree('/clickhouse/tables/{cluster}/{shard}/hits2', '{replica}')
    PARTITION BY toMonday(date)
    ORDER BY (CounterID, EventDate)
    </engine>

    <!-- Sharding key used to insert data to destination cluster -->
    <sharding_key>jumpConsistentHash(intHash64(userID), 2)</sharding_key>

    <!-- Optional expression that filter data while pull them from source servers -->

```

```

<where_condition>CounterID != 0</where_condition>

<!-- This section specifies partitions that should be copied, other partition will be ignored.
Partition names should have the same format as
partition column of system.parts table (i.e. a quoted text).
Since partition key of source and destination cluster could be different,
these partition names specify destination partitions.

NOTE: In spite of this section is optional (if it is not specified, all partitions will be copied),
it is strictly recommended to specify them explicitly.
If you already have some ready partitions on destination cluster they
will be removed at the start of the copying since they will be interpreted
as unfinished data from the previous copying!!!
-->
<enabled_partitions>
  <partition>'2018-02-26'</partition>
  <partition>'2018-03-05'</partition>
  ...
</enabled_partitions>
</table_hits>

<!-- Next table to copy. It is not copied until previous table is copying. -->
</table_visits>
...
</table_visits>
...
</tables>
</yandex>

```

`clickhouse-copier` tracks the changes in `/task/path/description` and applies them on the fly. For instance, if you change the value of `max_workers`, the number of processes running tasks will also change.

clickhouse-local

The `clickhouse-local` program enables you to perform fast processing on local files, without having to deploy and configure the ClickHouse server.

Accepts data that represent tables and queries them using `ClickHouse SQL dialect`.

`clickhouse-local` uses the same core as ClickHouse server, so it supports most of the features and the same set of formats and table engines.

By default `clickhouse-local` does not have access to data on the same host, but it supports loading server configuration using `--config-file` argument.

Warning

It is not recommended to load production server configuration into `clickhouse-local` because data can be damaged in case of human error.

Usage

Basic usage:

```
clickhouse-local --structure "table_structure" --input-format "format_of_incoming_data" -q "query"
```

Arguments:

- `-S`, `--structure` — table structure for input data.
- `-if`, `--input-format` — input format, `TSV` by default.
- `-f`, `--file` — path to data, `stdin` by default.
- `-q` `--query` — queries to execute with `;` as delimiter.
- `-N`, `--table` — table name where to put output data, `table` by default.

- `-of`, `--format`, `--output-format` — output format, **TSV** by default.
- `--stacktrace` — whether to dump debug output in case of exception.
- `--verbose` — more details on query execution.
- `-s` — disables **stderr** logging.
- `--config-file` — path to configuration file in same format as for ClickHouse server, by default the configuration empty.
- `--help` — arguments references for **clickhouse-local**.

Also there are arguments for each ClickHouse configuration variable which are more commonly used instead of `--config-file`.

Examples

```
echo -e "1,2\n3,4" | clickhouse-local -S "a Int64, b Int64" -if "CSV" -q "SELECT * FROM table"
Read 2 rows, 32.00 B in 0.000 sec., 5182 rows/sec., 80.97 KiB/sec.
1 2
3 4
```

Previous example is the same as:

```
$ echo -e "1,2\n3,4" | clickhouse-local -q "CREATE TABLE table (a Int64, b Int64) ENGINE = File(CSV, stdin); SELECT a, b FROM table; DROP TABLE table"
Read 2 rows, 32.00 B in 0.000 sec., 4987 rows/sec., 77.93 KiB/sec.
1 2
3 4
```

Now let's output memory user for each Unix user:

```
$ ps aux | tail -n +2 | awk '{ printf("%s\t%s\n", $1, $4) }' | clickhouse-local -S "user String, mem Float64" -q "SELECT user, round(sum(mem), 2) as memTotal FROM table GROUP BY user ORDER BY memTotal DESC FORMAT Pretty"
Read 186 rows, 4.15 KiB in 0.035 sec., 5302 rows/sec., 118.34 KiB/sec.
```

user	memTotal
bayonet	113.5
root	8.8
...	

General Questions

Why Not Use Something Like MapReduce?

We can refer to systems like MapReduce as distributed computing systems in which the reduce operation is based on distributed sorting. The most common open source solution in this class is **Apache Hadoop**. Yandex uses their in-house solution, YT.

These systems aren't appropriate for online queries due to their high latency. In other words, they can't be used as the back-end for a web interface.

These types of systems aren't useful for real-time data updates.

Distributed sorting isn't the best way to perform reduce operations if the result of the operation and all the intermediate results (if there are any) are located in the RAM of a single server, which is usually the case for online queries. In such a case, a hash table is the optimal way to perform reduce operations. A common approach to optimizing map-reduce tasks is pre-aggregation (partial reduce) using a hash table in RAM. The user performs this optimization manually.

Distributed sorting is one of the main causes of reduced performance when running simple map-reduce tasks.

Most MapReduce implementations allow you to execute arbitrary code on a cluster. But a declarative query language is better suited to OLAP in order to run experiments quickly. For example, Hadoop has Hive and Pig. Also consider Cloudera Impala or Shark (outdated) for Spark, as well as Spark SQL, Presto, and Apache Drill. Performance when running such tasks is highly sub-optimal compared to specialized systems, but relatively high latency makes it unrealistic to use these systems as the backend for a web interface.

What to do if I have a problem with encodings when using Oracle through ODBC?

If you use Oracle through ODBC driver as a source of external dictionaries, you need to set up correctly value for the `NLS_LANG` variable in the `/etc/default/clickhouse`. For more details see the [Oracle NLS_LANG FAQ](#).

Example

```
NLS_LANG=RUSSIAN_RUSSIA.UTF8
```

ClickHouse Development

Overview of ClickHouse Architecture

ClickHouse is a true column-oriented DBMS. Data is stored by columns, and during the execution of arrays (vectors or chunks of columns). Whenever possible, operations are dispatched on arrays, rather than on individual values. This is called "vectorized query execution," and it helps lower the cost of actual data processing.

This idea is nothing new. It dates back to the [APL](#) programming language and its descendants: [A](#), [+](#), [J](#), [K](#), and [Q](#). Array programming is used in scientific data processing. Neither is this idea something new in relational databases: for example, it is used in the [Vectorwise](#) system.

There are two different approaches for speeding up the query processing: vectorized query execution and runtime code generation. In the latter, the code is generated for every kind of query on the fly, removing all indirection and dynamic dispatch. Neither of these approaches is strictly better than the other. Runtime code generation can be better when it fuses many operations together, thus fully utilizing CPU execution units and the pipeline. Vectorized query execution can be less practical, because it involves temporary vectors that must be written to the cache and read back. If the temporary data does not fit in the L2 cache, this becomes an issue. But vectorized query execution more easily utilizes the SIMD capabilities of the CPU. A [research paper](#) written by our friends shows that it is better to combine both approaches. ClickHouse uses vectorized query execution and has limited initial support for runtime code generation.

Columns

To represent columns in memory (actually, chunks of columns), the `IColumn` interface is used. This interface provides helper methods for implementation of various relational operators. Almost all operations are immutable: they do not modify the original column, but create a new modified one. For example, the `IColumn::filter` method accepts a filter byte mask. It is used for the `WHERE` and `HAVING` relational operators. Additional examples: the `IColumn::permute` method to support `ORDER BY`, the `IColumn::cut` method to support `LIMIT`, and so on.

Various `IColumn` implementations (`ColumnUInt8`, `ColumnString` and so on) are responsible for the memory layout of columns. Memory layout is usually a contiguous array. For the integer type of columns it is just one contiguous array, like `std::vector`. For `String` and `Array` columns, it is two vectors: one for all array elements, placed contiguously, and a second one for offsets to the beginning of each array. There is also `ColumnConst` that stores just one value in memory, but looks like a column.

Field

Nevertheless, it is possible to work with individual values as well. To represent an individual value, the `Field` is used. `Field` is just a discriminated union of `UInt64`, `Int64`, `Float64`, `String` and `Array`. `IColumn` has the `operator[]` method to get the *n*-th value as a `Field`, and the `insert` method to append a `Field` to the end of a column. These methods are not very efficient, because they require dealing with temporary `Field` objects representing an individual value. There are more efficient methods, such as `insertFrom`, `insertRangeFrom`, and so on.

`Field` doesn't have enough information about a specific data type for a table. For example, `UInt8`, `UInt16`, `UInt32`, and `UInt64` are all represented as `UInt64` in a `Field`.

Leaky Abstractions

`IColumn` has methods for common relational transformations of data, but they don't meet all needs. For example, `ColumnUInt64` doesn't have a method to calculate the sum of two columns, and `ColumnString` doesn't have a method to run a substring search. These countless routines are implemented outside of `IColumn`.

Various functions on columns can be implemented in a generic, non-efficient way using `IColumn` methods to extract `Field` values, or in a specialized way using knowledge of inner memory layout of data in a specific `IColumn` implementation. To do this, functions are cast to a specific `IColumn` type and deal with internal representation directly. For example, `ColumnUInt64` has the `getData` method that returns a reference to an internal array, then a separate routine reads or fills that array directly. In fact, we have "leaky abstractions" to allow efficient specializations of various routines.

Data Types

`IDataType` is responsible for serialization and deserialization: for reading and writing chunks of columns or individual values in binary or text form.

`IDataType` directly corresponds to data types in tables. For example, there are `DataTypeUInt32`, `DataTypeDateTime`, `DataTypeString` and so on.

`IDataType` and `IColumn` are only loosely related to each other. Different data types can be represented in memory by the same `IColumn` implementations. For example, `DataTypeUInt32` and `DataTypeDateTime` are both represented by `ColumnUInt32` or `ColumnConstUInt32`. In addition, the same data type can be represented by different `IColumn` implementations. For example, `DataTypeUInt8` can be represented by `ColumnUInt8` or `ColumnConstUInt8`.

`IDataType` only stores metadata. For instance, `DataTypeUInt8` doesn't store anything at all (except `vptr`) and `DataTypeFixedString` stores just `N` (the size of fixed-size strings).

`IDataType` has helper methods for various data formats. Examples are methods to serialize a value with possible quoting, to serialize a value for JSON, and to serialize a value as part of XML format. There is no direct correspondence to data formats. For example, the different data formats `Pretty` and `TabSeparated` can use the same `serializeTextEscaped` helper method from the `IDataType` interface.

Block

A `Block` is a container that represents a subset (chunk) of a table in memory. It is just a set of triples: (`IColumn`, `IDataType`, `column name`). During query execution, data is processed by `Blocks`. If we have a `Block`, we have data (in the `IColumn` object), we have information about its type (in `IDataType`) that tells us how to deal with that column, and we have the column name (either the original column name from the table, or some artificial name assigned for getting temporary results of calculations).

When we calculate some function over columns in a block, we add another column with its result to the block, and we don't touch columns for arguments of the function because operations are immutable. Later, unneeded columns can be removed from the block, but not modified. This is convenient for elimination of common subexpressions.

Blocks are created for every processed chunk of data. Note that for the same type of calculation, the column names and types remain the same for different blocks, and only column data changes. It is better to split block data from the block header, because small block sizes will have a high overhead of temporary strings for copying `shared_ptrs` and column names.

Block Streams

Block streams are for processing data. We use streams of blocks to read data from somewhere, perform data transformations, or write data to somewhere. `IBlockInputStream` has the `read` method to fetch the next block while available. `IBlockOutputStream` has the `write` method to push the block somewhere.

Streams are responsible for:

1. Reading or writing to a table. The table just returns a stream for reading or writing blocks.
2. Implementing data formats. For example, if you want to output data to a terminal in `Pretty` format, you create a block output stream where you push blocks, and it formats them.
3. Performing data transformations. Let's say you have `IBlockInputStream` and want to create a filtered stream. You create `FilterBlockInputStream` and initialize it with your stream. Then when you pull a block from `FilterBlockInputStream`, it pulls a block from your stream, filters it, and returns the filtered block to you. Query execution pipelines are represented this way.

There are more sophisticated transformations. For example, when you pull from `AggregatingBlockInputStream`, it reads all data from its source, aggregates it, and then returns a stream of aggregated data for you. Another example: `UnionBlockInputStream` accepts many input sources in the constructor and also a number of threads. It launches multiple threads and reads from multiple sources in parallel.

Block streams use the "pull" approach to control flow: when you pull a block from the first stream, it consequently pulls the required blocks from nested streams, and the entire execution pipeline will work. Neither "pull" nor "push" is the best solution, because control flow is implicit, and that limits implementation of various features like simultaneous execution of multiple queries (merging many pipelines together). This limitation could be overcome with coroutines or just running extra threads that wait for each other. We may have more possibilities if we make control flow explicit: if we locate the logic for passing data from one calculation unit to another outside of those calculation units. Read this [article](#) for more thoughts.

We should note that the query execution pipeline creates temporary data at each step. We try to keep block size small enough so that temporary data fits in the CPU cache. With that assumption, writing and reading temporary data is almost free in comparison with other calculations. We could consider an alternative, which is to fuse many operations in the pipeline together, to make the pipeline as short as possible and remove much of the temporary data. This could be an advantage, but it also has drawbacks. For example, a split pipeline makes it easy to implement caching intermediate data, stealing intermediate data from similar queries running at the same time, and merging pipelines for similar queries.

Formats

Data formats are implemented with block streams. There are "presentational" formats only suitable for output of data to the client, such as `Pretty` format, which provides only `IBlockOutputStream`. And there are input/output formats, such as `TabSeparated` or `JSONEachRow`.

There are also row streams: `IRowInputStream` and `IRowOutputStream`. They allow you to pull/push data by individual rows, not by blocks. And they are only needed to simplify implementation of row-oriented formats. The wrappers `BlockInputStreamFromRowInputStream` and `BlockOutputStreamFromRowOutputStream` allow you to convert row-oriented streams to regular block-oriented streams.

I/O

For byte-oriented input/output, there are `ReadBuffer` and `WriteBuffer` abstract classes. They are used instead of C++ `istream`s. Don't worry: every mature C++ project is using something other than `istream`s for good reasons.

`ReadBuffer` and `WriteBuffer` are just a contiguous buffer and a cursor pointing to the position in that buffer. Implementations may own or not own the memory for the buffer. There is a virtual method to fill the buffer with the following data (for `ReadBuffer`) or to flush the buffer somewhere (for `WriteBuffer`). The virtual methods are rarely called.

Implementations of `ReadBuffer`/`WriteBuffer` are used for working with files and file descriptors and network sockets, for implementing compression (`CompressedWriteBuffer` is initialized with another `WriteBuffer` and performs compression before writing data to it), and for other purposes – the names `ConcatReadBuffer`, `LimitReadBuffer`, and `HashingWriteBuffer` speak for themselves.

`Read/WriteBuffers` only deal with bytes. To help with formatted input/output (for instance, to write a number in decimal format), there are functions from `ReadHelpers` and `WriteHelpers` header files.

Let's look at what happens when you want to write a result set in `JSON` format to stdout. You have a result set ready to be fetched from `IBlockInputStream`. You create `WriteBufferFromFileDescriptor(STDOUT_FILENO)` to write bytes to stdout. You create `JSONRowOutputStream`, initialized with that `WriteBuffer`, to write rows in `JSON` to stdout. You create `BlockOutputStreamFromRowOutputStream` on top of it, to represent it as `IBlockOutputStream`. Then you call `copyData` to transfer data from `IBlockInputStream` to `IBlockOutputStream`, and everything works. Internally, `JSONRowOutputStream` will write various `JSON` delimiters and call the `IDataType::serializeTextJSON` method with a reference to `IColumn` and the row number as arguments. Consequently, `IDataType::serializeTextJSON` will call a method from `WriteHelpers.h`: for example, `writeText` for numeric types and `writeJSONString` for `DataTypeString`.

Tables

Tables are represented by the `IStorage` interface. Different implementations of that interface are different table engines. Examples are `StorageMergeTree`, `StorageMemory`, and so on. Instances of these classes are just tables.

The most important `IStorage` methods are `read` and `write`. There are also `alter`, `rename`, `drop`, and so on. The `read` method accepts the following arguments: the set of columns to read from a table, the `AST` query to consider, and the desired number of streams to return. It returns one or multiple `IBlockInputStream` objects and information about the stage of data processing that was completed inside a table engine during query execution.

In most cases, the `read` method is only responsible for reading the specified columns from a table, not for any further data processing. All further data processing is done by the query interpreter and is outside the responsibility of `IStorage`.

But there are notable exceptions:

- The `AST` query is passed to the `read` method and the table engine can use it to derive index usage and to read less data from a table.
- Sometimes the table engine can process data itself to a specific stage. For example, `StorageDistributed` can send a query to remote servers, ask them to process data to a stage where data from different remote servers can be merged, and return that preprocessed data. The query interpreter then finishes processing the data.

The table's `read` method can return multiple `IBlockInputStream` objects to allow parallel data processing. These multiple block input streams can read from a table in parallel. Then you can wrap these streams with various transformations (such as expression evaluation or filtering) that can be calculated independently and create a `UnionBlockInputStream` on top of them, to read from multiple streams in parallel.

There are also `TableFunctions`. These are functions that return a temporary `IStorage` object to use in the `FROM` clause of a query.

To get a quick idea of how to implement your own table engine, look at something simple, like `StorageMemory` or `StorageTinyLog`.

As the result of the `read` method, `IStorage` returns `QueryProcessingStage` – information about what parts of the query were already calculated inside storage. Currently we have only very coarse granularity for that information. There is no way for the storage to say "I have already processed this part of the expression in `WHERE`, for this range of data". We need to work on that.

Parsers

A query is parsed by a hand-written recursive descent parser. For example, `ParserSelectQuery` just recursively calls the underlying parsers for various parts of the query. Parsers create an `AST`. The `AST` is represented by nodes, which are instances of `IAST`.

Parser generators are not used for historical reasons.

Interpreters

Interpreters are responsible for creating the query execution pipeline from an `AST`. There are simple interpreters, such as `InterpreterExistsQuery` and `InterpreterDropQuery`, or the more sophisticated `InterpreterSelectQuery`. The query execution pipeline is a combination of block input or output streams. For example, the result of interpreting the `SELECT` query is the `IBlockInputStream` to read the result set from; the result of the `INSERT` query is the `IBlockOutputStream` to write data for insertion to; and the result of interpreting the `INSERT SELECT` query is the `IBlockInputStream` that returns an empty result set on the first read, but that copies data from `SELECT` to `INSERT` at the same time.

`InterpreterSelectQuery` uses `ExpressionAnalyzer` and `ExpressionActions` machinery for query analysis and transformations. This is where most rule-based query optimizations are done. `ExpressionAnalyzer` is quite messy and should be rewritten: various query transformations and optimizations should be extracted to separate classes to allow modular transformations or query.

Functions

There are ordinary functions and aggregate functions. For aggregate functions, see the next section.

Ordinary functions don't change the number of rows – they work as if they are processing each row independently. In fact, functions are not called for individual rows, but for `Block`'s of data to implement vectorized query execution.

There are some miscellaneous functions, like `blockSize`, `rowNumberInBlock`, and `runningAccumulate`, that exploit block processing and violate the independence of rows.

ClickHouse has strong typing, so implicit type conversion doesn't occur. If a function doesn't support a specific combination of types, an exception will be thrown. But functions can work (be overloaded) for many different combinations of types. For example, the `plus` function (to implement the `+` operator) works for any combination of numeric types: `UInt8 + Float32`, `UInt16 + Int8`, and so on. Also, some variadic functions can accept any number of arguments, such as the `concat` function.

Implementing a function may be slightly inconvenient because a function explicitly dispatches supported data types and supported `IColumns`. For example, the `plus` function has code generated by instantiation of a C++ template for each combination of numeric types, and for constant or non-constant left and right arguments.

This is a nice place to implement runtime code generation to avoid template code bloat. Also, it will make it possible to add fused functions like fused multiply-add, or to make multiple comparisons in one loop iteration.

Due to vectorized query execution, functions are not short-circuit. For example, if you write `WHERE f(x) AND g(y)`, both sides will be calculated, even for rows, when `f(x)` is zero (except when `f(x)` is a zero constant expression). But if selectivity of the `f(x)` condition is high, and calculation of `f(x)` is much cheaper than `g(y)`, it's better to implement multi-pass calculation: first calculate `f(x)`, then filter columns by the result, and then calculate `g(y)` only for smaller, filtered chunks of data.

Aggregate Functions

Aggregate functions are stateful functions. They accumulate passed values into some state, and allow you to get results from that state. They are managed with the `IAggregateFunction` interface. States can be rather simple (the state for `AggregateFunctionCount` is just a single `UInt64` value) or quite complex (the state of `AggregateFunctionUniqCombined` is a combination of a linear array, a hash table and a `HyperLogLog` probabilistic data structure).

To deal with multiple states while executing a high-cardinality **GROUP BY** query, states are allocated in **Arena** (a memory pool), or they could be allocated in any suitable piece of memory. States can have a non-trivial constructor and destructor: for example, complex aggregation states can allocate additional memory themselves. This requires some attention to creating and destroying states and properly passing their ownership, to keep track of who and when will destroy states.

Aggregation states can be serialized and deserialized to pass over the network during distributed query execution or to write them on disk where there is not enough RAM. They can even be stored in a table with the **DataTypeAggregateFunction** to allow incremental aggregation of data.

The serialized data format for aggregate function states is not versioned right now. This is ok if aggregate states are only stored temporarily. But we have the **AggregatingMergeTree** table engine for incremental aggregation, and people are already using it in production. This is why we should add support for backward compatibility when changing the serialized format for any aggregate function in the future.

Server

The server implements several different interfaces:

- An HTTP interface for any foreign clients.
- A TCP interface for the native ClickHouse client and for cross-server communication during distributed query execution.
- An interface for transferring data for replication.

Internally, it is just a basic multithreaded server without coroutines, fibers, etc. Since the server is not designed to process a high rate of simple queries but is intended to process a relatively low rate of complex queries, each of them can process a vast amount of data for analytics.

The server initializes the **Context** class with the necessary environment for query execution: the list of available databases, users and access rights, settings, clusters, the process list, the query log, and so on. This environment is used by interpreters.

We maintain full backward and forward compatibility for the server TCP protocol: old clients can talk to new servers and new clients can talk to old servers. But we don't want to maintain it eternally, and we are removing support for old versions after about one year.

For all external applications, we recommend using the HTTP interface because it is simple and easy to use. The TCP protocol is more tightly linked to internal data structures: it uses an internal format for passing blocks of data and it uses custom framing for compressed data. We haven't released a C library for that protocol because it requires linking most of the ClickHouse codebase, which is not practical.

Distributed Query Execution

Servers in a cluster setup are mostly independent. You can create a **Distributed** table on one or all servers in a cluster. The **Distributed** table does not store data itself – it only provides a "view" to all local tables on multiple nodes of a cluster. When you **SELECT** from a **Distributed** table, it rewrites that query, chooses remote nodes according to load balancing settings, and sends the query to them. The **Distributed** table requests remote servers to process a query just up to a stage where intermediate results from different servers can be merged. Then it receives the intermediate results and merges them. The distributed table tries to distribute as much work as possible to remote servers, and does not send much intermediate data over the network.

Things become more complicated when you have subqueries in **IN** or **JOIN** clauses and each of them uses a **Distributed** table. We have different strategies for execution of these queries.

There is no global query plan for distributed query execution. Each node has its own local query plan for its part of the job. We only have simple one-pass distributed query execution: we send queries for remote nodes and then merge the results. But this is not feasible for difficult queries with high cardinality GROUP BYs or with a large amount of temporary data for JOIN: in such cases, we need to "reshuffle" data between servers, which requires additional coordination. ClickHouse does not support that kind of query execution, and we need to work on it.

Merge Tree

MergeTree is a family of storage engines that supports indexing by primary key. The primary key can be an arbitrary tuple of columns or expressions. Data in a **MergeTree** table is stored in "parts". Each part stores data in the primary key order (data is ordered lexicographically by the primary key tuple). All the table columns are stored in separate **column.bin** files in these parts. The files consist of compressed blocks. Each block is usually from 64 KB to 1 MB of uncompressed data, depending on the average value size. The blocks consist of column values placed contiguously one after the other. Column values are in the same order for each column (the order is defined by the primary key), so when you iterate by many columns, you get values for the corresponding rows.

The primary key itself is "sparse". It doesn't address each single row, but only some ranges of data. A separate **primary.idx** file has the value of the primary key for each N-th row, where N is called **index_granularity** (usually, N = 8192). Also, for each column, we have **column.mrk** files with "marks," which are offsets to each N-th row in the data file. Each mark is a pair: the offset in the file to the beginning of the compressed block, and the offset in the decompressed block to the beginning of data. Usually compressed blocks are aligned by marks, and the offset in the decompressed block is zero. Data for **primary.idx** always resides in memory and data for **column.mrk** files is cached.

When we are going to read something from a part in **MergeTree**, we look at **primary.idx** data and locate ranges that could possibly contain requested data, then look at **column.mrk** data and calculate offsets for where to start reading those ranges. Because of sparseness, excess data may be read. ClickHouse is not suitable for a high load of simple point queries, because the entire range with **index_granularity** rows must be read for each key, and the entire compressed block must be decompressed for each column. We made the index sparse because we must be able to maintain trillions of rows per single server without noticeable memory consumption for the index. Also, because the primary key is sparse, it is not unique: it cannot check the existence of the key in the table at INSERT time. You could have many rows with the same key in a table.

When you **INSERT** a bunch of data into **MergeTree**, that bunch is sorted by primary key order and forms a new part. To keep the number of parts relatively low, there are background threads that periodically select some parts and merge them to a single sorted part. That's why it is called **MergeTree**. Of course, merging leads to "write amplification". All parts are immutable: they are only created and deleted, but not modified. When SELECT is run, it holds a snapshot of the table (a set of parts). After merging, we also keep old parts for some time to make recovery after failure easier, so if we see that some merged part is probably broken, we can replace it with its source parts.

MergeTree is not an LSM tree because it doesn't contain "memtable" and "log": inserted data is written directly to the filesystem. This makes it suitable only to INSERT data in batches, not by individual row and not very frequently – about once per second is ok, but a thousand times a second is not. We did it this way for simplicity's sake, and because we are already inserting data in batches in our applications.

MergeTree tables can only have one (primary) index: there aren't any secondary indices. It would be nice to allow multiple physical representations under one logical table, for example, to store data in more than one physical order or even to allow representations with pre-aggregated data along with original data.

There are MergeTree engines that are doing additional work during background merges. Examples are **CollapsingMergeTree** and **AggregatingMergeTree**. This could be treated as special support for updates. Keep in mind that these are not real updates because users usually have no control over the time when background merges will be executed, and data in a **MergeTree** table is almost always stored in more than one part, not in completely merged form.

Replication

Replication in ClickHouse is implemented on a per-table basis. You could have some replicated and some non-replicated tables on the same server. You could also have tables replicated in different ways, such as one table with two-factor replication and another with three-factor.

Replication is implemented in the **ReplicatedMergeTree** storage engine. The path in **ZooKeeper** is specified as a parameter for the storage engine. All tables with the same path in **ZooKeeper** become replicas of each other: they synchronize their data and maintain consistency. Replicas can be added and removed dynamically simply by creating or dropping a table.

Replication uses an asynchronous multi-master scheme. You can insert data into any replica that has a session with **ZooKeeper**, and data is replicated to all other replicas asynchronously. Because ClickHouse doesn't support UPDATES, replication is conflict-free. As there is no quorum acknowledgment of inserts, just-inserted data might be lost if one node fails.

Metadata for replication is stored in ZooKeeper. There is a replication log that lists what actions to do. Actions are: get part; merge parts; drop partition, etc. Each replica copies the replication log to its queue and then executes the actions from the queue. For example, on insertion, the "get part" action is created in the log, and every replica downloads that part. Merges are coordinated between replicas to get byte-identical results. All parts are merged in the same way on all replicas. To achieve this, one replica is elected as the leader, and that replica initiates merges and writes "merge parts" actions to the log.

Replication is physical: only compressed parts are transferred between nodes, not queries. To lower the network cost (to avoid network amplification), merges are processed on each replica independently in most cases. Large merged parts are sent over the network only in cases of significant replication lag.

In addition, each replica stores its state in ZooKeeper as the set of parts and its checksums. When the state on the local filesystem diverges from the reference state in ZooKeeper, the replica restores its consistency by downloading missing and broken parts from other replicas. When there is some unexpected or broken data in the local filesystem, ClickHouse does not remove it, but moves it to a separate directory and forgets it.

The ClickHouse cluster consists of independent shards, and each shard consists of replicas. The cluster is not elastic, so after adding a new shard, data is not rebalanced between shards automatically. Instead, the cluster load will be uneven. This implementation gives you more control, and it is fine for relatively small clusters such as tens of nodes. But for clusters with hundreds of nodes that we are using in production, this approach becomes a significant drawback. We should implement a table engine that will span its data across the cluster with dynamically replicated regions that could be split and balanced between clusters automatically.

How to Build ClickHouse Release Package

Install Git and Pbuilder

```
sudo apt-get update
sudo apt-get install git pbuilder debhelper lsb-release fakeroot sudo debian-archive-keyring debian-keyring
```

Checkout ClickHouse Sources

```
git clone --recursive --branch stable https://github.com/yandex/ClickHouse.git
cd ClickHouse
```

Run Release Script

```
./release
```

How to Build ClickHouse for Development

The following tutorial is based on the Ubuntu Linux system.

With appropriate changes, it should also work on any other Linux distribution.

Only x86_64 with SSE 4.2 is supported. Support for AArch64 is experimental.

To test for SSE 4.2, do


```
grep -q sse4_2 /proc/cpuinfo && echo "SSE 4.2 supported" || echo "SSE 4.2 not supported"
```

Install Git and CMake

```
sudo apt-get install git cmake ninja-build
```

Or cmake3 instead of cmake on older systems.

Install GCC 7

There are several ways to do this.

Install from a PPA Package

```
sudo apt-get install software-properties-common
sudo apt-add-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
sudo apt-get install gcc-7 g++-7
```

Install from Sources

Look at [ci/build-gcc-from-sources.sh](#)

Use GCC 7 for Builds

```
export CC=gcc-7
export CXX=g++-7
```

Install Required Libraries from Packages

```
sudo apt-get install libicu-dev libreadline-dev
```

Checkout ClickHouse Sources

```
git clone --recursive git@github.com:yandex/ClickHouse.git
## or: git clone --recursive https://github.com/yandex/ClickHouse.git

cd ClickHouse
```

For the latest stable version, switch to the [stable](#) branch.

Build ClickHouse

```
mkdir build
cd build
cmake ..
ninja
cd ..
```

To create an executable, run [ninja clickhouse](#).

This will create the [dbms/programs/clickhouse](#) executable, which can be used with [client](#) or [server](#) arguments.

How to Build ClickHouse on Mac OS X

Build should work on Mac OS X 10.12.

Install Homebrew

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Install Required Compilers, Tools, and Libraries

```
brew install cmake ninja gcc icu4c openssl libtool gettext readline
```

Checkout ClickHouse Sources

```
git clone --recursive git@github.com:yandex/ClickHouse.git
## or: git clone --recursive https://github.com/yandex/ClickHouse.git
```

```
cd ClickHouse
```

For the latest stable version, switch to the [stable](#) branch.

Build ClickHouse

```
mkdir build
cd build
cmake .. -DCMAKE_CXX_COMPILER=`which g++-8` -DCMAKE_C_COMPILER=`which gcc-8`
ninja
cd ..
```

Caveats

If you intend to run clickhouse-server, make sure to increase the system's maxfiles variable.

Note

You'll need to use sudo.

To do so, create the following file:

/Library/LaunchDaemons/limit.maxfiles.plist:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>Label</key>
    <string>limit.maxfiles</string>
    <key>ProgramArguments</key>
    <array>
      <string>launchctl</string>
      <string>limit</string>
      <string>maxfiles</string>
      <string>524288</string>
      <string>524288</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
    <key>ServiceIPC</key>
    <false/>
  </dict>
</plist>
```

Execute the following command:

```
$ sudo chown root:wheel /Library/LaunchDaemons/limit.maxfiles.plist
```

Reboot.

To check if it's working, you can use [ulimit -n](#) command.

How to Write C++ Code

General Recommendations

1. The following are recommendations, not requirements.
2. If you are editing code, it makes sense to follow the formatting of the existing code.

3. Code style is needed for consistency. Consistency makes it easier to read the code, and it also makes it easier to search the code.

4. Many of the rules do not have logical reasons; they are dictated by established practices.

Formatting

1. Most of the formatting will be done automatically by **clang-format**.

2. Indents are 4 spaces. Configure your development environment so that a tab adds four spaces.

3. Opening and closing curly brackets must be on a separate line.

```
inline void readBoolText(bool & x, ReadBuffer & buf)
{
    char tmp = '0';
    readChar(tmp, buf);
    x = tmp != '0';
}
```

4. If the entire function body is a single **statement**, it can be placed on a single line. Place spaces around curly braces (besides the space at the end of the line).

```
inline size_t mask() const          { return buf_size() - 1; }
inline size_t place(HashValue x) const { return x & mask(); }
```

5. For functions. Don't put spaces around brackets.

```
void reinsert(const Value & x)
```

```
memcpy(&buf[place_value], &x, sizeof(x));
```

6. In **if**, **for**, **while** and other expressions, a space is inserted in front of the opening bracket (as opposed to function calls).

```
for (size_t i = 0; i < rows; i += storage.index_granularity)
```

7. Add spaces around binary operators (**+**, **-**, *****, **/**, **%**, ...) and the ternary operator **?:**.

```
UInt16 year = (s[0] - '0') * 1000 + (s[1] - '0') * 100 + (s[2] - '0') * 10 + (s[3] - '0');
UInt8 month = (s[5] - '0') * 10 + (s[6] - '0');
UInt8 day = (s[8] - '0') * 10 + (s[9] - '0');
```

8. If a line feed is entered, put the operator on a new line and increase the indent before it.

```
if (elapsed_ns)
    message << " ("
        << rows_read_on_server * 1000000000 / elapsed_ns << " rows/s., "
        << bytes_read_on_server * 1000.0 / elapsed_ns << " MB/s.) ";
```

9. You can use spaces for alignment within a line, if desired.

```
dst.ClickLogID      = click.LogID;
dst.ClickEventID    = click.EventID;
dst.ClickGoodEvent  = click.GoodEvent;
```

10. Don't use spaces around the operators **.**, **->**.

If necessary, the operator can be wrapped to the next line. In this case, the offset in front of it is increased.

11. Do not use a space to separate unary operators (**--**, **++**, *****, **&**, ...) from the argument.

12. Put a space after a comma, but not before it. The same rule goes for a semicolon inside a **for** expression.

13. Do not use spaces to separate the **[]** operator.

14. In a **template** `<...>` expression, use a space between **template** and `<`; no spaces after `<` or before `>`.

```
template <typename TKey, typename TValue>
struct AggregatedStatElement
{}
```

15. In classes and structures, write **public**, **private**, and **protected** on the same level as **class/struct**, and indent the rest of the code.

```
template <typename T>
class MultiVersion
{
public:
    /// Version of object for usage. shared_ptr manage lifetime of version.
    using Version = std::shared_ptr<const T>;
    ...
}
```

16. If the same **namespace** is used for the entire file, and there isn't anything else significant, an offset is not necessary inside **namespace**.

17. If the block for an **if**, **for**, **while**, or other expression consists of a single **statement**, the curly brackets are optional. Place the **statement** on a separate line, instead. This rule is also valid for nested **if**, **for**, **while**, ...

But if the inner **statement** contains curly brackets or **else**, the external block should be written in curly brackets.

```
/// Finish write.
for (auto & stream : streams)
    stream.second->finalize();
```

18. There shouldn't be any spaces at the ends of lines.

19. Source files are UTF-8 encoded.

20. Non-ASCII characters can be used in string literals.

```
<< " , " << (timer.elapsed() / chunks_stats.hits) << " μsec/hit.";
```

21 Do not write multiple expressions in a single line.

22. Group sections of code inside functions and separate them with no more than one empty line.

23. Separate functions, classes, and so on with one or two empty lines.

24. A **const** (related to a value) must be written before the type name.

```
//correct
const char * pos
const std::string & s
//incorrect
char const * pos
```

25. When declaring a pointer or reference, the ***** and **&** symbols should be separated by spaces on both sides.

```
//correct
const char * pos
//incorrect
const char* pos
const char *pos
```

26. When using template types, alias them with the **using** keyword (except in the simplest cases).

In other words, the template parameters are specified only in **using** and aren't repeated in the code.

using can be declared locally, such as inside a function.

```
//correct
using FileStreams = std::map<std::string, std::shared_ptr<Stream>>;
FileStreams streams;
//incorrect
std::map<std::string, std::shared_ptr<Stream>> streams;
```

27. Do not declare several variables of different types in one statement.

```
//incorrect
int x, *y;
```

28. Do not use C-style casts.

```
//incorrect
std::cerr << (int)c <<; std::endl;
//correct
std::cerr << static_cast<int>(c) << std::endl;
```

29. In classes and structs, group members and functions separately inside each visibility scope.

30. For small classes and structs, it is not necessary to separate the method declaration from the implementation.

The same is true for small methods in any classes or structs.

For templated classes and structs, don't separate the method declarations from the implementation (because otherwise they must be defined in the same translation unit).

31. You can wrap lines at 140 characters, instead of 80.

32. Always use the prefix increment/decrement operators if postfix is not required.

```
for (Names::const_iterator it = column_names.begin(); it != column_names.end(); ++it)
```

Comments

1. Be sure to add comments for all non-trivial parts of code.

This is very important. Writing the comment might help you realize that the code isn't necessary, or that it is designed wrong.

```
/** Part of piece of memory, that can be used.
 * For example, if internal_buffer is 1MB, and there was only 10 bytes loaded to buffer from file for reading,
 * then working_buffer will have size of only 10 bytes
 * (working_buffer.end() will point to position right after those 10 bytes available for read).
 */
```

2. Comments can be as detailed as necessary.

3. Place comments before the code they describe. In rare cases, comments can come after the code, on the same line.

```
/** Parses and executes the query.
 */
void executeQuery(
    ReadBuffer & istr, /// Where to read the query from (and data for INSERT, if applicable)
    WriteBuffer & ostr, /// Where to write the result
    Context & context, /// DB, tables, data types, engines, functions, aggregate functions...
    BlockInputStreamPtr & query_plan, /// Here could be written the description on how query was executed
    QueryProcessingStage::Enum stage = QueryProcessingStage::Complete /// Up to which stage process the SELECT query
)
```

4. Comments should be written in English only.

5. If you are writing a library, include detailed comments explaining it in the main header file.

6. Do not add comments that do not provide additional information. In particular, do not leave empty comments like this:

```
/*
 * Procedure Name:
 * Original procedure name:
 * Author:
 * Date of creation:
 * Dates of modification:
 * Modification authors:
 * Original file name:
 * Purpose:
 * Intent:
 * Designation:
 * Classes used:
 * Constants:
 * Local variables:
 * Parameters:
 * Date of creation:
 * Purpose:
 */
```

The example is borrowed from the resource <http://home.tamk.fi/~jaalto/course/coding-style/doc/unmaintainable-code/>.

7. Do not write garbage comments (author, creation date ..) at the beginning of each file.

8. Single-line comments begin with three slashes: `///` and multi-line comments begin with `/**`. These comments are considered "documentation".

Note: You can use Doxygen to generate documentation from these comments. But Doxygen is not generally used because it is more convenient to navigate the code in the IDE.

9. Multi-line comments must not have empty lines at the beginning and end (except the line that closes a multi-line comment).

10. For commenting out code, use basic comments, not "documenting" comments.

11. Delete the commented out parts of the code before committing.

12. Do not use profanity in comments or code.

13. Do not use uppercase letters. Do not use excessive punctuation.

```
/// WHAT THE FAIL???
```

14. Do not use comments to make delimiters.

```
///*****
```

15. Do not start discussions in comments.

```
/// Why did you do this stuff?
```

16. There's no need to write a comment at the end of a block describing what it was about.

```
/// for
```

Names

1. Use lowercase letters with underscores in the names of variables and class members.

```
size_t max_block_size;
```

2. For the names of functions (methods), use camelCase beginning with a lowercase letter.

```
std::string getName() const override { return "Memory"; }
```

3. For the names of classes (structs), use CamelCase beginning with an uppercase letter. Prefixes other than `I` are not used for interfaces.

```
class StorageMemory : public IStorage
```

4. `using` are named the same way as classes, or with `_t` on the end.

5. Names of template type arguments: in simple cases, use `T`; `T`, `U`; `T1`, `T2`.

For more complex cases, either follow the rules for class names, or add the prefix `T`.

```
template <typename TKey, typename TValue>
struct AggregatedStatElement
```

6. Names of template constant arguments: either follow the rules for variable names, or use `N` in simple cases.

```
template <bool without_www>
struct ExtractDomain
```

7. For abstract classes (interfaces) you can add the `I` prefix.

```
class IBlockInputStream
```

8. If you use a variable locally, you can use the short name.

In all other cases, use a name that describes the meaning.

```
bool info_successfully_loaded = false;
```

9. Names of `defines` and global constants use ALL_CAPS with underscores.

```
##define MAX_SRC_TABLE_NAMES_TO_STORE 1000
```

10. File names should use the same style as their contents.

If a file contains a single class, name the file the same way as the class (CamelCase).

If the file contains a single function, name the file the same way as the function (camelCase).

11. If the name contains an abbreviation, then:

- For variable names, the abbreviation should use lowercase letters `mysql_connection` (not `mySQL_connection`).
- For names of classes and functions, keep the uppercase letters in the abbreviation `MySQLConnection` (not `MySqlConnection`).

12. Constructor arguments that are used just to initialize the class members should be named the same way as the class members, but with an underscore at the end.

```
FileQueueProcessor(
    const std::string & path_,
    const std::string & prefix_,
    std::shared_ptr<FileHandler> handler_)
: path(path_),
  prefix(prefix_),
  handler(handler_),
  log(&Logger::get("FileQueueProcessor"))
{
}
```

The underscore suffix can be omitted if the argument is not used in the constructor body.

13. There is no difference in the names of local variables and class members (no prefixes required).

```
timer (not m_timer)
```

14. For the constants in an `enum`, use CamelCase with a capital letter. ALL_CAPS is also acceptable. If the `enum` is non-local, use an `enum class`.

```
enum class CompressionMethod
{
    QuickLZ = 0,
    LZ4     = 1,
};
```

15. All names must be in English. Transliteration of Russian words is not allowed.

not Stroka

16. Abbreviations are acceptable if they are well known (when you can easily find the meaning of the abbreviation in Wikipedia or in a search engine).

`AST`, `SQL`.

Not `NVDH` (some random letters)

Incomplete words are acceptable if the shortened version is common use.

You can also use an abbreviation if the full name is included next to it in the comments.

17. File names with C++ source code must have the `.cpp` extension. Header files must have the `.h` extension.

How to Write Code

1. Memory management.

Manual memory deallocation (`delete`) can only be used in library code.

In library code, the `delete` operator can only be used in destructors.

In application code, memory must be freed by the object that owns it.

Examples:

- The easiest way is to place an object on the stack, or make it a member of another class.
- For a large number of small objects, use containers.
- For automatic deallocation of a small number of objects that reside in the heap, use `shared_ptr/unique_ptr`.

2. Resource management.

Use `RAII` and see above.

3. Error handling.

Use exceptions. In most cases, you only need to throw an exception, and don't need to catch it (because of `RAII`).

In offline data processing applications, it's often acceptable to not catch exceptions.

In servers that handle user requests, it's usually enough to catch exceptions at the top level of the connection handler.

In thread functions, you should catch and keep all exceptions to rethrow them in the main thread after `join`.

```
/// If there weren't any calculations yet, calculate the first block synchronously
if (!started)
{
    calculate();
    started = true;
}
else /// If calculations are already in progress, wait for the result
    pool.wait();

if (exception)
    exception->rethrow();
```

Never hide exceptions without handling. Never just blindly put all exceptions to log.

```
//Not correct
catch (...) {}
```

If you need to ignore some exceptions, do so only for specific ones and rethrow the rest.

```
catch (const DB::Exception & e)
{
    if (e.code() == ErrorCodes::UNKNOWN_AGGREGATE_FUNCTION)
        return nullptr;
    else
        throw;
}
```

When using functions with response codes or **errno**, always check the result and throw an exception in case of error.

```
if (0 != close(fd))
    throwFromErrno("Cannot close file " + file_name, ErrorCodes::CANNOT_CLOSE_FILE);
```

Do not use **assert**.

4. Exception types.

There is no need to use complex exception hierarchy in application code. The exception text should be understandable to a system administrator.

5. Throwing exceptions from destructors.

This is not recommended, but it is allowed.

Use the following options:

- Create a function (**done()** or **finalize()**) that will do all the work in advance that might lead to an exception. If that function was called, there should be no exceptions in the destructor later.
- Tasks that are too complex (such as sending messages over the network) can be put in separate method that the class user will have to call before destruction.
- If there is an exception in the destructor, it's better to log it than to hide it (if the logger is available).
- In simple applications, it is acceptable to rely on **std::terminate** (for cases of **noexcept** by default in C++11) to handle exceptions.

6. Anonymous code blocks.

You can create a separate code block inside a single function in order to make certain variables local, so that the destructors are called when exiting the block.

```
Block block = data.in->read();

{
    std::lock_guard<std::mutex> lock(mutex);
    data.ready = true;
    data.block = block;
}

ready_any.set();
```

7. Multithreading.

In offline data processing programs:

- Try to get the best possible performance on a single CPU core. You can then parallelize your code if necessary.

In server applications:

- Use the thread pool to process requests. At this point, we haven't had any tasks that required userspace context switching.

Fork is not used for parallelization.

8. Syncing threads.

Often it is possible to make different threads use different memory cells (even better: different cache lines,) and to not use any thread synchronization (except `joinAll`).

If synchronization is required, in most cases, it is sufficient to use mutex under `lock_guard`.

In other cases use system synchronization primitives. Do not use busy wait.

Atomic operations should be used only in the simplest cases.

Do not try to implement lock-free data structures unless it is your primary area of expertise.

9. Pointers vs references.

In most cases, prefer references.

10. const.

Use constant references, pointers to constants, `const_iterator`, and const methods.

Consider `const` to be default and use non-`const` only when necessary.

When passing variables by value, using `const` usually does not make sense.

11. unsigned.

Use `unsigned` if necessary.

12. Numeric types.

Use the types `UInt8`, `UInt16`, `UInt32`, `UInt64`, `Int8`, `Int16`, `Int32`, and `Int64`, as well as `size_t`, `ssize_t`, and `ptrdiff_t`.

Don't use these types for numbers: `signed/unsigned long`, `long long`, `short`, `signed/unsigned char`, `char`.

13. Passing arguments.

Pass complex values by reference (including `std::string`).

If a function captures ownership of an object created in the heap, make the argument type `shared_ptr` or `unique_ptr`.

14. Return values.

In most cases, just use `return`. Do not write `[return std::move(res)]{.strike}`.

If the function allocates an object on heap and returns it, use `shared_ptr` or `unique_ptr`.

In rare cases you might need to return the value via an argument. In this case, the argument should be a reference.

```
using AggregateFunctionPtr = std::shared_ptr<IAggregateFunction>;

/** Allows creating an aggregate function by its name.
 */
class AggregateFunctionFactory
{
public:
    AggregateFunctionFactory();
    AggregateFunctionPtr get(const String & name, const DataTypes & argument_types) const;
```

15. namespace.

There is no need to use a separate `namespace` for application code.

Small libraries don't need this, either.

For medium to large libraries, put everything in a `namespace`.

In the library's `.h` file, you can use `namespace detail` to hide implementation details not needed for the application code.

In a `.cpp` file, you can use a `static` or anonymous namespace to hide symbols.

Also, a `namespace` can be used for an `enum` to prevent the corresponding names from falling into an external `namespace` (but it's better to use an `enum class`).

16. Deferred initialization.

If arguments are required for initialization, then you normally shouldn't write a default constructor.

If later you'll need to delay initialization, you can add a default constructor that will create an invalid object. Or, for a small number of objects, you can use `shared_ptr/unique_ptr`.

```
Loader(DB::Connection * connection_, const std::string & query, size_t max_block_size_);

/// For deferred initialization
Loader() {}
```

17. Virtual functions.

If the class is not intended for polymorphic use, you do not need to make functions virtual. This also applies to the destructor.

18. Encodings.

Use UTF-8 everywhere. Use `std::string` and `char *`. Do not use `std::wstring` and `wchar_t`.

19. Logging.

See the examples everywhere in the code.

Before committing, delete all meaningless and debug logging, and any other types of debug output.

Logging in cycles should be avoided, even on the Trace level.

Logs must be readable at any logging level.

Logging should only be used in application code, for the most part.

Log messages must be written in English.

The log should preferably be understandable for the system administrator.

Do not use profanity in the log.

Use UTF-8 encoding in the log. In rare cases you can use non-ASCII characters in the log.

20. Input-output.

Don't use `iostreams` in internal cycles that are critical for application performance (and never use `stringstream`).

Use the `DB/IO` library instead.

21. Date and time.

See the `DateLUT` library.

22. include.

Always use `#pragma once` instead of include guards.

23. using.

`using namespace` is not used. You can use `using` with something specific. But make it local inside a class or function.

24. Do not use trailing return type for functions unless necessary.

```
[auto f() -&gt; void;]{.strike}
```

25. Declaration and initialization of variables.

```
//right way
std::string s = "Hello";
std::string s{"Hello"};

//wrong way
auto s = std::string{"Hello"};
```

26. For virtual functions, write `virtual` in the base class, but write `override` instead of `virtual` in descendent classes.

Unused Features of C++

1. Virtual inheritance is not used.

2. Exception specifiers from C++03 are not used.

Platform

1. We write code for a specific platform.

But other things being equal, cross-platform or portable code is preferred.

2. Language: C++17.

3. Compiler: `gcc`. At this time (December 2017), the code is compiled using version 7.2. (It can also be compiled using `clang 4`.)

The standard library is used (`libstdc++` or `libc++`).

4.OS: Linux Ubuntu, not older than Precise.

5.Code is written for x86_64 CPU architecture.

The CPU instruction set is the minimum supported set among our servers. Currently, it is SSE 4.2.

6. Use `-Wall -Wextra -Werror` compilation flags.
7. Use static linking with all libraries except those that are difficult to connect to statically (see the output of the `ldd` command).
8. Code is developed and debugged with release settings.

Tools

1. KDevelop is a good IDE.
2. For debugging, use `gdb`, `valgrind` (`memcheck`), `strace`, `-fsanitize=...`, or `tcmalloc_minimal_debug`.
3. For profiling, use `Linux Perf`, `valgrind` (`callgrind`), or `strace -cf`.
4. Sources are in Git.
5. Assembly uses `CMake`.
6. Programs are released using `deb` packages.
7. Commits to master must not break the build.

Though only selected revisions are considered workable.

8. Make commits as often as possible, even if the code is only partially ready.

Use branches for this purpose.

If your code in the `master` branch is not buildable yet, exclude it from the build before the `push`. You'll need to finish it or remove it within a few days.

9. For non-trivial changes, use branches and publish them on the server.
10. Unused code is removed from the repository.

Libraries

1. The C++14 standard library is used (experimental extensions are allowed), as well as `boost` and `Poco` frameworks.

2. If necessary, you can use any well-known libraries available in the OS package.

If there is a good solution already available, then use it, even if it means you have to install another library.

(But be prepared to remove bad libraries from code.)

3. You can install a library that isn't in the packages, if the packages don't have what you need or have an outdated version or the wrong type of compilation.
4. If the library is small and doesn't have its own complex build system, put the source files in the `contrib` folder.
5. Preference is always given to libraries that are already in use.

General Recommendations

1. Write as little code as possible.
2. Try the simplest solution.
3. Don't write code until you know how it's going to work and how the inner loop will function.
4. In the simplest cases, use `using` instead of classes or structs.

5. If possible, do not write copy constructors, assignment operators, destructors (other than a virtual one, if the class contains at least one virtual function), move constructors or move assignment operators. In other words, the compiler-generated functions must work correctly. You can use `default`.

6. Code simplification is encouraged. Reduce the size of your code where possible.

Additional Recommendations

1. Explicitly specifying `std::` for types from `stddef.h`

is not recommended. In other words, we recommend writing `size_t` instead of `std::size_t`, because it's shorter.

It is acceptable to add `std::`.

2. Explicitly specifying `std::` for functions from the standard C library

is not recommended. In other words, write `memcpy` instead of `std::memcpy`.

The reason is that there are similar non-standard functions, such as `memmem`. We do use these functions on occasion. These functions do not exist in `namespace std`.

If you write `std::memcpy` instead of `memcpy` everywhere, then `memmem` without `std::` will look strange.

Nevertheless, you can still use `std::` if you prefer it.

3. Using functions from C when the same ones are available in the standard C++ library.

This is acceptable if it is more efficient.

For example, use `memcpy` instead of `std::copy` for copying large chunks of memory.

4. Multiline function arguments.

Any of the following wrapping styles are allowed:

```
function(  
    T1 x1,  
    T2 x2)
```

```
function(  
    size_t left, size_t right,  
    const & RangesInDataParts ranges,  
    size_t limit)
```

```
function(size_t left, size_t right,  
    const & RangesInDataParts ranges,  
    size_t limit)
```

```
function(size_t left, size_t right,  
    const & RangesInDataParts ranges,  
    size_t limit)
```

```
function(  
    size_t left,  
    size_t right,  
    const & RangesInDataParts ranges,  
    size_t limit)
```

ClickHouse Testing

Functional Tests

Functional tests are the most simple and convenient to use. Most of ClickHouse features can be tested with functional tests and they are mandatory to use for every change in ClickHouse code that can be tested that way.

Each functional test sends one or multiple queries to the running ClickHouse server and compares the result with reference.

Tests are located in `dbms/tests/queries` directory. There are two subdirectories: `stateless` and `stateful`. Stateless tests run queries without any preloaded test data - they often create small synthetic datasets on the fly, within the test itself. Stateful tests require preloaded test data from Yandex.Metrica and not available to general public. We tend to use only `stateless` tests and avoid adding new `stateful` tests.

Each test can be one of two types: `.sql` and `.sh`. `.sql` test is the simple SQL script that is piped to `clickhouse-client --multiquery --testmode`. `.sh` test is a script that is run by itself.

To run all tests, use `dbms/tests/clickhouse-test` tool. Look `--help` for the list of possible options. You can simply run all tests or run subset of tests filtered by substring in test name: `./clickhouse-test substring`.

The most simple way to invoke functional tests is to copy `clickhouse-client` to `/usr/bin/`, run `clickhouse-server` and then run `./clickhouse-test` from its own directory.

To add new test, create a `.sql` or `.sh` file in `dbms/tests/queries/0_stateless` directory, check it manually and then generate `.reference` file in the following way: `clickhouse-client -n --testmode < 00000_test.sql > 00000_test.reference` or `./00000_test.sh > ./00000_test.reference`.

Tests should use (create, drop, etc) only tables in `test` database that is assumed to be created beforehand; also tests can use temporary tables.

If you want to use distributed queries in functional tests, you can leverage `remote` table function with `127.0.0.{1..2}` addresses for the server to query itself; or you can use predefined test clusters in server configuration file like `test_shard_localhost`.

Some tests are marked with `zookeeper`, `shard` or `long` in their names. `zookeeper` is for tests that are using ZooKeeper. `shard` is for tests that requires server to listen `127.0.0.*`; `distributed` or `global` have the same meaning. `long` is for tests that run slightly longer than one second. You can disable these groups of tests using `--no-zookeeper`, `--no-shard` and `--no-long` options, respectively.

Known bugs

If we know some bugs that can be easily reproduced by functional tests, we place prepared functional tests in `dbms/tests/queries/bugs` directory. These tests will be moved to `dbms/tests/queries/0_stateless` when bugs are fixed.

Integration Tests

Integration tests allow to test ClickHouse in clustered configuration and ClickHouse interaction with other servers like MySQL, Postgres, MongoDB. They are useful to emulate network splits, packet drops, etc. These tests are run under Docker and create multiple containers with various software.

See `dbms/tests/integration/README.md` on how to run these tests.

Note that integration of ClickHouse with third-party drivers is not tested. Also we currently don't have integration tests with our JDBC and ODBC drivers.

Unit Tests

Unit tests are useful when you want to test not the ClickHouse as a whole, but a single isolated library or class. You can enable or disable build of tests with `ENABLE_TESTS` CMake option. Unit tests (and other test programs) are located in `tests` subdirectories across the code. To run unit tests, type `ninja test`. Some tests use `gtest`, but some are just programs that return non-zero exit code on test failure.

It's not necessarily to have unit tests if the code is already covered by functional tests (and functional tests are usually much more simple to use).

Performance Tests

Performance tests allow to measure and compare performance of some isolated part of ClickHouse on synthetic queries. Tests are located at `dbms/tests/performance`. Each test is represented by `.xml` file with description of test case. Tests are run with `clickhouse performance-test` tool (that is embedded in `clickhouse` binary). See `--help` for invocation.

Each test run one or multiple queries (possibly with combinations of parameters) in a loop with some conditions for stop (like "maximum execution speed is not changing in three seconds") and measure some metrics about query performance (like "maximum execution speed"). Some tests can contain preconditions on preloaded test dataset.

If you want to improve performance of ClickHouse in some scenario, and if improvements can be observed on simple queries, it is highly recommended to write a performance test. It always makes sense to use `perf top` or other perf tools during your tests.

Test Tools And Scripts

Some programs in `tests` directory are not prepared tests, but are test tools. For example, for `Lexer` there is a tool `dbms/src/Parsers/tests/lexer` that just do tokenization of stdin and writes colored result to stdout. You can use these kind of tools as a code examples and for exploration and manual testing.

You can also place pair of files `.sh` and `.reference` along with the tool to run it on some predefined input - then script result can be compared to `.reference` file. These kind of tests are not automated.

Miscellaneous Tests

There are tests for external dictionaries located at `dbms/tests/external_dictionaries` and for machine learned models in `dbms/tests/external_models`. These tests are not updated and must be transferred to integration tests.

There is separate test for quorum inserts. This test run ClickHouse cluster on separate servers and emulate various failure cases: network split, packet drop (between ClickHouse nodes, between ClickHouse and ZooKeeper, between ClickHouse server and client, etc.), `kill -9`, `kill -STOP` and `kill -CONT`, like `Jepsen`. Then the test checks that all acknowledged inserts was written and all rejected inserts was not.

Quorum test was written by separate team before ClickHouse was open-sourced. This team no longer work with ClickHouse. Test was accidentally written in Java. For these reasons, quorum test must be rewritten and moved to integration tests.

Manual Testing

When you develop a new feature, it is reasonable to also test it manually. You can do it with the following steps:

Build ClickHouse. Run ClickHouse from the terminal: change directory to `dbms/src/programs/clickhouse-server` and run it with `./clickhouse-server`. It will use configuration (`config.xml`, `users.xml` and files within `config.d` and `users.d` directories) from the current directory by default. To connect to ClickHouse server, run `dbms/src/programs/clickhouse-client/clickhouse-client`.

Note that all clickhouse tools (server, client, etc) are just symlinks to a single binary named `clickhouse`. You can find this binary at `dbms/src/programs/clickhouse`. All tools can also be invoked as `clickhouse tool` instead of `clickhouse-tool`.

Alternatively you can install ClickHouse package: either stable release from Yandex repository or you can build package for yourself with `./release` in ClickHouse sources root. Then start the server with `sudo service clickhouse-server start` (or `stop` to stop the server). Look for logs at `/etc/clickhouse-server/clickhouse-server.log`.

When ClickHouse is already installed on your system, you can build a new `clickhouse` binary and replace the existing binary:

```
sudo service clickhouse-server stop
sudo cp ./clickhouse /usr/bin/
sudo service clickhouse-server start
```

Also you can stop system clickhouse-server and run your own with the same configuration but with logging to terminal:

```
sudo service clickhouse-server stop
sudo -u clickhouse /usr/bin/clickhouse server --config-file /etc/clickhouse-server/config.xml
```

Example with gdb:

```
sudo -u clickhouse gdb --args /usr/bin/clickhouse server --config-file /etc/clickhouse-server/config.xml
```

If the system clickhouse-server is already running and you don't want to stop it, you can change port numbers in your `config.xml` (or override them in a file in `config.d` directory), provide appropriate data path, and run it.

`clickhouse` binary has almost no dependencies and works across wide range of Linux distributions. To quick and dirty test your changes on a server, you can simply `scp` your fresh built `clickhouse` binary to your server and then run it as in examples above.

Testing Environment

Before publishing release as stable we deploy it on testing environment. Testing environment is a cluster that process 1/39 part of `Yandex.Metrica` data. We share our testing environment with Yandex.Metrica team. ClickHouse is upgraded without downtime on top of existing data. We look at first that data is processed successfully without lagging from realtime, the replication continue to work and there is no issues visible to Yandex.Metrica team. First check can be done in the following way:

```
SELECT hostName() AS h, any(version()), any(uptime()), max(UTCEventTime), count() FROM remote('example01-01-{1..3}t', merge, hits) WHERE EventDate >= today() - 2 GROUP BY h ORDER BY h;
```

In some cases we also deploy to testing environment of our friend teams in Yandex: Market, Cloud, etc. Also we have some hardware servers that are used for development purposes.

Load Testing

After deploying to testing environment we run load testing with queries from production cluster. This is done manually.

Make sure you have enabled `query_log` on your production cluster.

Collect query log for a day or more:

```
clickhouse-client --query="SELECT DISTINCT query FROM system.query_log WHERE event_date = today() AND query LIKE '%ym:%' AND query NOT LIKE '%system.query_log%' AND type = 2 AND is_initial_query" > queries.tsv
```

This is a way complicated example. `type = 2` will filter queries that are executed successfully. `query LIKE '%ym:%'` is to select relevant queries from Yandex.Metrica. `is_initial_query` is to select only queries that are initiated by client, not by ClickHouse itself (as parts of distributed query processing).

`scp` this log to your testing cluster and run it as following:

```
clickhouse benchmark --concurrency 16 < queries.tsv
```

(probably you also want to specify a `--user`)

Then leave it for a night or weekend and go take a rest.

You should check that `clickhouse-server` doesn't crash, memory footprint is bounded and performance not degrading over time.

Precise query execution timings are not recorded and not compared due to high variability of queries and environment.

Build Tests

Build tests allow to check that build is not broken on various alternative configurations and on some foreign systems. Tests are located at [ci](#) directory. They run build from source inside Docker, Vagrant, and sometimes with [qemu-user-static](#) inside Docker. These tests are under development and test runs are not automated.

Motivation:

Normally we release and run all tests on a single variant of ClickHouse build. But there are alternative build variants that are not thoroughly tested. Examples:

- build on FreeBSD;
- build on Debian with libraries from system packages;
- build with shared linking of libraries;
- build on AArch64 platform;
- build on PowerPc platform.

For example, build with system packages is bad practice, because we cannot guarantee what exact version of packages a system will have. But this is really needed by Debian maintainers. For this reason we at least have to support this variant of build. Another example: shared linking is a common source of trouble, but it is needed for some enthusiasts.

Though we cannot run all tests on all variant of builds, we want to check at least that various build variants are not broken. For this purpose we use build tests.

Testing For Protocol Compatibility

When we extend ClickHouse network protocol, we test manually that old clickhouse-client works with new clickhouse-server and new clickhouse-client works with old clickhouse-server (simply by running binaries from corresponding packages).

Help From The Compiler

Main ClickHouse code (that is located in [dbms](#) directory) is built with [-Wall](#) [-Wextra](#) [-Werror](#) and with some additional enabled warnings. Although these options are not enabled for third-party libraries.

Clang has even more useful warnings - you can look for them with [-Weverything](#) and pick something to default build.

For production builds, gcc is used (it still generates slightly more efficient code than clang). For development, clang is usually more convenient to use. You can build on your own machine with debug mode (to save battery of your laptop), but please note that compiler is able to generate more warnings with [-O3](#) due to better control flow and inter-procedure analysis. When building with clang, [libc++](#) is used instead of [libstdc++](#) and when building with debug mode, debug version of [libc++](#) is used that allows to catch more errors at runtime.

Sanitizers

Address sanitizer.

We run functional and integration tests under ASan on per-commit basis.

Valgrind (Memcheck).

We run functional tests under Valgrind overnight. It takes multiple hours. Currently there is one known false positive in [re2](#) library, see [this article](#).

Undefined behaviour sanitizer.

We run functional and integration tests under ASan on per-commit basis.

Thread sanitizer.

We run functional tests under TSan on per-commit basis. We still don't run integration tests under TSan on per-commit basis.

Memory sanitizer.

Currently we still don't use MSan.

Debug allocator.

Debug version of [jemalloc](#) is used for debug build.

Fuzzing

We use simple fuzz test to generate random SQL queries and to check that the server doesn't die. Fuzz testing is performed with Address sanitizer. You can find it in [00746_sql_fuzzy.pl](#). This test should be run continuously (overnight and longer).

As of December 2018, we still don't use isolated fuzz testing of library code.

Security Audit

People from Yandex Cloud department do some basic overview of ClickHouse capabilities from the security standpoint.

Static Analyzers

We run [PVS-Studio](#) on per-commit basis. We have evaluated [clang-tidy](#), [Coverity](#), [cppcheck](#), [PVS-Studio](#), [tscancode](#). You will find instructions for usage in [dbms/tests/instructions/](#) directory. Also you can read [the article in russian](#).

If you use [CLion](#) as an IDE, you can leverage some [clang-tidy](#) checks out of the box.

Hardening

[FORTIFY_SOURCE](#) is used by default. It is almost useless, but still makes sense in rare cases and we don't disable it.

Code Style

Code style rules are described [here](#).

To check for some common style violations, you can use [utils/check-style](#) script.

To force proper style of your code, you can use [clang-format](#). File [.clang-format](#) is located at the sources root. It mostly corresponding with our actual code style. But it's not recommended to apply [clang-format](#) to existing files because it makes formatting worse. You can use [clang-format-diff](#) tool that you can find in clang source repository.

Alternatively you can try [uncrustify](#) tool to reformat your code. Configuration is in [uncrustify.cfg](#) in the sources root. It is less tested than [clang-format](#).

[CLion](#) has its own code formatter that has to be tuned for our code style.

Metrika B2B Tests

Each ClickHouse release is tested with Yandex Metrika and AppMetrika engines. Testing and stable versions of ClickHouse are deployed on VMs and run with a small copy of Metrika engine that is processing fixed sample of input data. Then results of two instances of Metrika engine are compared together.

These tests are automated by separate team. Due to high number of moving parts, tests are fail most of the time by completely unrelated reasons, that are very difficult to figure out. Most likely these tests have negative value for us. Nevertheless these tests was proved to be useful in about one or two times out of hundreds.

Test Coverage

As of July 2018 we don't track test coverage.

Test Automation

We run tests with Yandex internal CI and job automation system named "Sandbox".

Build jobs and tests are run in Sandbox on per commit basis. Resulting packages and test results are published in GitHub and can be downloaded by direct links. Artifacts are stored eternally. When you send a pull request on GitHub, we tag it as "can be tested" and our CI system will build ClickHouse packages (release, debug, with address sanitizer, etc) for you.

We don't use Travis CI due to the limit on time and computational power.

We don't use Jenkins. It was used before and now we are happy we are not using Jenkins.

Third-Party Libraries Used

Library	License
base64	BSD 2-Clause License
boost	Boost Software License 1.0
brotli	MIT
capnproto	MIT
cctz	Apache License 2.0
double-conversion	BSD 3-Clause License
FastMemcpy	MIT
googletest	BSD 3-Clause License
hyperscan	BSD 3-Clause License
libbtrie	BSD 2-Clause License
libcxxabi	BSD + MIT
libdivide	Zlib License
libgsasl	LGPL v2.1
libhdfs3	Apache License 2.0
libmetrohash	Apache License 2.0
libpcg-random	Apache License 2.0
libressl	OpenSSL License
librdkafka	BSD 2-Clause License
libwidechar_width	CC0 1.0 Universal
llvm	BSD 3-Clause License
lz4	BSD 2-Clause License
mariadb-connector-c	LGPL v2.1
murmurhash	Public Domain
pdqsort	Zlib License
poco	Boost Software License - Version 1.0
protobuf	BSD 3-Clause License
re2	BSD 3-Clause License
UnixODBC	LGPL v2.1
zlib-ng	Zlib License
zstd	BSD 3-Clause License

Roadmap

Q2 2019

- DDL for dictionaries
- Integration with S3-like object stores
- Multiple storages for hot/cold data, JBOD support

Q3 2019

- JOIN execution improvements:
 - Distributed join not limited by memory
- Resource pools for more precise distribution of cluster capacity between users
- Fine-grained authorization

- Integration with external authentication services

ClickHouse release 19.6.2.11, 2019-05-13

New Features

- TTL expressions for columns and tables. [#4212](#) (Anton Popov)
- Added support for [brotli](#) compression for HTTP responses (Accept-Encoding: br) [#4388](#) (Mikhail)
- Added new function [isValidUTF8](#) for checking whether a set of bytes is correctly utf-8 encoded. [#4934](#) (Danila Kutenin)
- Add new load balancing policy [first_or_random](#) which sends queries to the first specified host and if it's inaccessible send queries to random hosts of shard. Useful for cross-replication topology setups. [#5012](#) (nvartolomei)

Experimental Features

- Add setting [index_granularity_bytes](#) (adaptive index granularity) for MergeTree* tables family. [#4826](#) (alesapin)

Improvements

- Added support for non-constant and negative size and length arguments for function [substringUTF8](#). [#4989](#) (alexey-milovidov)
- Disable push-down to right table in left join, left table in right join, and both tables in full join. This fixes wrong JOIN results in some cases. [#4846](#) (Ivan)
- [clickhouse-copier](#): auto upload task configuration from [--task-file](#) option [#4876](#) (proller)
- Added typos handler for storage factory and table functions factory. [#4891](#) (Danila Kutenin)
- Support asterisks and qualified asterisks for multiple joins without subqueries [#4898](#) (Artem Zuikov)
- Make missing column error message more user friendly. [#4915](#) (Artem Zuikov)

Performance Improvements

- Significant speedup of ASOF JOIN [#4924](#) (Martijn Bakker)

Backward Incompatible Changes

- HTTP header [Query-Id](#) was renamed to [X-ClickHouse-Query-Id](#) for consistency. [#4972](#) (Mikhail)

Bug Fixes

- Fixed potential null pointer dereference in [clickhouse-copier](#). [#4900](#) (proller)
- Fixed error on query with JOIN + ARRAY JOIN [#4938](#) (Artem Zuikov)
- Fixed hanging on start of the server when a dictionary depends on another dictionary via a database with engine=Dictionary. [#4962](#) (Vitaly Baranov)
- Partially fix distributed_product_mode = local. It's possible to allow columns of local tables in where/having/order by/... via table aliases. Throw exception if table does not have alias. There's not possible to access to the columns without table aliases yet. [#4986](#) (Artem Zuikov)
- Fix potentially wrong result for [SELECT DISTINCT](#) with JOIN [#5001](#) (Artem Zuikov)

Build/Testing/Packaging Improvements

- Fixed test failures when running clickhouse-server on different host [#4713](#) (Vasily Nemkov)
- clickhouse-test: Disable color control sequences in non tty environment. [#4937](#) (alesapin)
- clickhouse-test: Allow use any test database (remove [test.](#) qualification where it possible) [#5008](#) (proller)
- Fix ubsan errors [#5037](#) (Vitaly Baranov)
- Yandex LFAIloc was added to ClickHouse to allocate MarkCache and UncompressedCache data in different ways to catch segfaults more reliable [#4995](#) (Danila Kutenin)
- Python util to help with backports and changelogs. [#4949](#) (Ivan)

ClickHouse release 19.5.4.22, 2019-05-13

Bug fixes

- Fixed possible crash in bitmap* functions [#5220](#) [#5228](#) (Andy Yang)

- Fixed very rare data race condition that could happen when executing a query with UNION ALL involving at least two SELECTs from system.columns, system.tables, system.parts, system.parts_tables or tables of Merge family and performing ALTER of columns of the related tables concurrently. [#5189](#) (alexey-milovidov)
- Fixed error [Set for IN is not created yet in case of using single LowCardinality column in the left part of IN](#) This error happened if LowCardinality column was the part of primary key. [#5031](#) [#5154](#) (Nikolai Kochetov)
- Modification of retention function: If a row satisfies both the first and NTH condition, only the first satisfied condition is added to the data state. Now all conditions that satisfy in a row of data are added to the data state. [#5119](#) (小路)

ClickHouse release 19.5.3.8, 2019-04-18

Bug fixes

- Fixed type of setting [max_partitions_per_insert_block](#) from boolean to UInt64. [#5028](#) (Mohammad Hossein Sekhavat)

ClickHouse release 19.5.2.6, 2019-04-15

New Features

- [Hyperscan](#) multiple regular expression matching was added (functions [multiMatchAny](#), [multiMatchAnyIndex](#), [multiFuzzyMatchAny](#), [multiFuzzyMatchAnyIndex](#)). [#4780](#), [#4841](#) (Danila Kutenin)
- [multiSearchFirstPosition](#) function was added. [#4780](#) (Danila Kutenin)
- Implement the predefined expression filter per row for tables. [#4792](#) (Ivan)
- A new type of data skipping indices based on bloom filters (can be used for [equal](#), [in](#) and [like](#) functions). [#4499](#) (Nikita Vasilev)
- Added [ASOF JOIN](#) which allows to run queries that join to the most recent value known. [#4774](#) [#4867](#) [#4863](#) [#4875](#) (Martijn Bakker, Artem Zuikov)
- Rewrite multiple [COMMA JOIN](#) to [CROSS JOIN](#). Then rewrite them to [INNER JOIN](#) if possible. [#4661](#) (Artem Zuikov)

Improvement

- [topK](#) and [topKWeighted](#) now supports custom [loadFactor](#) (fixes issue [#4252](#)). [#4634](#) (Kirill Danshin)
- Allow to use [parallel_replicas_count > 1](#) even for tables without sampling (the setting is simply ignored for them). In previous versions it was lead to exception. [#4637](#) (Alexey Elymanov)
- Support for [CREATE OR REPLACE VIEW](#). Allow to create a view or set a new definition in a single statement. [#4654](#) (Boris Granveaud)
- [Buffer](#) table engine now supports [PREWHERE](#). [#4671](#) (Yangkuan Liu)
- Add ability to start replicated table without metadata in zookeeper in [readonly](#) mode. [#4691](#) (alesapin)
- Fixed flicker of progress bar in clickhouse-client. The issue was most noticeable when using [FORMAT Null](#) with streaming queries. [#4811](#) (alexey-milovidov)
- Allow to disable functions with [hyperscan](#) library on per user basis to limit potentially excessive and uncontrolled resource usage. [#4816](#) (alexey-milovidov)
- Add version number logging in all errors. [#4824](#) (proller)
- Added restriction to the [multiMatch](#) functions which requires string size to fit into [unsigned int](#). Also added the number of arguments limit to the [multiSearch](#) functions. [#4834](#) (Danila Kutenin)
- Improved usage of scratch space and error handling in Hyperscan. [#4866](#) (Danila Kutenin)
- Fill [system.graphite_detentions](#) from a table config of [*GraphiteMergeTree](#) engine tables. [#4584](#) (Mikhail f. Shiryaev)
- Rename [trigramDistance](#) function to [ngramDistance](#) and add more functions with [CaseInsensitive](#) and [UTF](#). [#4602](#) (Danila Kutenin)
- Improved data skipping indices calculation. [#4640](#) (Nikita Vasilev)
- Keep ordinary, [DEFAULT](#), [MATERIALIZED](#) and [ALIAS](#) columns in a single list (fixes issue [#2867](#)). [#4707](#) (Alex Zatelepin)

Bug Fix

- Avoid [std::terminate](#) in case of memory allocation failure. Now [std::bad_alloc](#) exception is thrown as expected. [#4665](#) (alexey-milovidov)
- Fixes capnproto reading from buffer. Sometimes files wasn't loaded successfully by HTTP. [#4674](#) (Vladislav)

- Fix error `Unknown log entry type: 0` after `OPTIMIZE TABLE FINAL` query. #4683 (Amos Bird)
- Wrong arguments to `hasAny` or `hasAll` functions may lead to segfault. #4698 (alexey-milovidov)
- Deadlock may happen while executing `DROP DATABASE` dictionary query. #4701 (alexey-milovidov)
- Fix undefined behavior in `median` and `quantile` functions. #4702 (hcz)
- Fix compression level detection when `network_compression_method` in lowercase. Broken in v19.1. #4706 (proller)
- Fixed ignorance of `<timezone>UTC</timezone>` setting (fixes issue #4658). #4718 (proller)
- Fix `histogram` function behaviour with `Distributed` tables. #4741 (olegkv)
- Fixed tsan report `destroy of a locked mutex`. #4742 (alexey-milovidov)
- Fixed TSan report on shutdown due to race condition in system logs usage. Fixed potential use-after-free on shutdown when `part_log` is enabled. #4758 (alexey-milovidov)
- Fix recheck parts in `ReplicatedMergeTreeAlterThread` in case of error. #4772 (Nikolai Kochetov)
- Arithmetic operations on intermediate aggregate function states were not working for constant arguments (such as subquery results). #4776 (alexey-milovidov)
- Always backquote column names in metadata. Otherwise it's impossible to create a table with column named `index` (server won't restart due to malformed `ATTACH` query in metadata). #4782 (alexey-milovidov)
- Fix crash in `ALTER ... MODIFY ORDER BY` on `Distributed` table. #4790 (TCeason)
- Fix segfault in `JOIN ON` with enabled `enable_optimize_predicate_expression`. #4794 (Winter Zhang)
- Fix bug with adding an extraneous row after consuming a protobuf message from Kafka. #4808 (Vitaly Baranov)
- Fix crash of `JOIN` on not-nullable vs nullable column. Fix `NULLs` in right keys in `ANY JOIN + join_use_nulls`. #4815 (Artem Zuikov)
- Fix segmentation fault in `clickhouse-copier`. #4835 (proller)
- Fixed race condition in `SELECT` from `system.tables` if the table is renamed or altered concurrently. #4836 (alexey-milovidov)
- Fixed data race when fetching data part that is already obsolete. #4839 (alexey-milovidov)
- Fixed rare data race that can happen during `RENAME` table of `MergeTree` family. #4844 (alexey-milovidov)
- Fixed segmentation fault in function `arrayIntersect`. Segmentation fault could happen if function was called with mixed constant and ordinary arguments. #4847 (Lixiang Qian)
- Fixed reading from `Array(LowCardinality)` column in rare case when column contained a long sequence of empty arrays. #4850 (Nikolai Kochetov)
- Fix crash in `FULL/RIGHT JOIN` when we joining on nullable vs not nullable. #4855 (Artem Zuikov)
- Fix `No message received` exception while fetching parts between replicas. #4856 (alesapin)
- Fixed `arrayIntersect` function wrong result in case of several repeated values in single array. #4871 (Nikolai Kochetov)
- Fix a race condition during concurrent `ALTER COLUMN` queries that could lead to a server crash (fixes issue #3421). #4592 (Alex Zatelepin)
- Fix incorrect result in `FULL/RIGHT JOIN` with const column. #4723 (Artem Zuikov)
- Fix duplicates in `GLOBAL JOIN` with asterisk. #4705 (Artem Zuikov)
- Fix parameter deduction in `ALTER MODIFY` of column `CODEC` when column type is not specified. #4883 (alesapin)
- Functions `cutQueryStringAndFragment()` and `queryStringAndFragment()` now works correctly when `URL` contains a fragment and no query. #4894 (Vitaly Baranov)
- Fix rare bug when setting `min_bytes_to_use_direct_io` is greater than zero, which occurs when thread have to seek backward in column file. #4897 (alesapin)
- Fix wrong argument types for aggregate functions with `LowCardinality` arguments (fixes issue #4919). #4922 (Nikolai Kochetov)
- Fix wrong name qualification in `GLOBAL JOIN`. #4969 (Artem Zuikov)
- Fix function `toISOWeek` result for year 1970. #4988 (alexey-milovidov)
- Fix `DROP`, `TRUNCATE` and `OPTIMIZE` queries duplication, when executed on `ON CLUSTER` for `ReplicatedMergeTree*` tables family. #4991 (alesapin)

Backward Incompatible Change

- Rename setting `insert_sample_with_metadata` to setting `input_format_defaults_for_omitted_fields`. #4771 (Artem Zuikov)

- Added setting `max_partitions_per_insert_block` (with value 100 by default). If inserted block contains larger number of partitions, an exception is thrown. Set it to 0 if you want to remove the limit (not recommended). [#4845](#) (alexey-milovidov)
- Multi-search functions were renamed (`multiPosition` to `multiSearchAllPositions`, `multiSearch` to `multiSearchAny`, `firstMatch` to `multiSearchFirstIndex`). [#4780](#) (Danila Kutenin)

Performance Improvement

- Optimize Volnitsky searcher by inlining, giving about 5-10% search improvement for queries with many needles or many similar bigrams. [#4862](#) (Danila Kutenin)
- Fix performance issue when setting `use_uncompressed_cache` is greater than zero, which appeared when all read data contained in cache. [#4913](#) (alesapin)

Build/Testing/Packaging Improvement

- Hardening debug build: more granular memory mappings and ASLR; add memory protection for mark cache and index. This allows to find more memory stomping bugs in case when ASan and MSan cannot do it. [#4632](#) (alexey-milovidov)
- Add support for cmake variables `ENABLE_PROTOBUF`, `ENABLE_PARQUET` and `ENABLE_BROTLI` which allows to enable/disable the above features (same as we can do for librdkafka, mysql, etc). [#4669](#) (Silviu Caragea)
- Add ability to print process list and stacktraces of all threads if some queries are hung after test run. [#4675](#) (alesapin)
- Add retries on `Connection loss` error in `clickhouse-test`. [#4682](#) (alesapin)
- Add freebsd build with vagrant and build with thread sanitizer to packager script. [#4712](#) [#4748](#) (alesapin)
- Now user asked for password for user 'default' during installation. [#4725](#) (proller)
- Suppress warning in `rdkafka` library. [#4740](#) (alexey-milovidov)
- Allow ability to build without ssl. [#4750](#) (proller)
- Add a way to launch clickhouse-server image from a custom user. [#4753](#) (Mikhail f. Shiryayev)
- Upgrade contrib boost to 1.69. [#4793](#) (proller)
- Disable usage of `mremap` when compiled with Thread Sanitizer. Surprisingly enough, TSan does not intercept `mremap` (though it does intercept `mmap`, `munmap`) that leads to false positives. Fixed TSan report in stateful tests. [#4859](#) (alexey-milovidov)
- Add test checking using format schema via HTTP interface. [#4864](#) (Vitaly Baranov)

ClickHouse release 19.4.4.33, 2019-04-17

Bug Fixes

- Avoid `std::terminate` in case of memory allocation failure. Now `std::bad_alloc` exception is thrown as expected. [#4665](#) (alexey-milovidov)
- Fixes capnproto reading from buffer. Sometimes files wasn't loaded successfully by HTTP. [#4674](#) (Vladislav)
- Fix error `Unknown log entry type: 0` after `OPTIMIZE TABLE FINAL` query. [#4683](#) (Amos Bird)
- Wrong arguments to `hasAny` or `hasAll` functions may lead to segfault. [#4698](#) (alexey-milovidov)
- Deadlock may happen while executing `DROP DATABASE dictionary` query. [#4701](#) (alexey-milovidov)
- Fix undefined behavior in `median` and `quantile` functions. [#4702](#) (hcz)
- Fix compression level detection when `network_compression_method` in lowercase. Broken in v19.1. [#4706](#) (proller)
- Fixed ignorance of `<timezone>UTC</timezone>` setting (fixes issue [#4658](#)). [#4718](#) (proller)
- Fix `histogram` function behaviour with `Distributed` tables. [#4741](#) (olegkv)
- Fixed tsan report `destroy of a locked mutex`. [#4742](#) (alexey-milovidov)
- Fixed TSan report on shutdown due to race condition in system logs usage. Fixed potential use-after-free on shutdown when `part_log` is enabled. [#4758](#) (alexey-milovidov)
- Fix recheck parts in `ReplicatedMergeTreeAlterThread` in case of error. [#4772](#) (Nikolai Kochetov)
- Arithmetic operations on intermediate aggregate function states were not working for constant arguments (such as subquery results). [#4776](#) (alexey-milovidov)
- Always backquote column names in metadata. Otherwise it's impossible to create a table with column named `index` (server won't restart due to malformed `ATTACH` query in metadata). [#4782](#) (alexey-milovidov)
- Fix crash in `ALTER ... MODIFY ORDER BY` on `Distributed` table. [#4790](#) (TCeason)
- Fix segfault in `JOIN ON` with enabled `enable_optimize_predicate_expression`. [#4794](#) (Winter Zhang)

- Fix bug with adding an extraneous row after consuming a protobuf message from Kafka. [#4808](#) (Vitaly Baranov)
- Fix segmentation fault in `clickhouse-copier`. [#4835](#) (proller)
- Fixed race condition in `SELECT` from `system.tables` if the table is renamed or altered concurrently. [#4836](#) (alexey-milovidov)
- Fixed data race when fetching data part that is already obsolete. [#4839](#) (alexey-milovidov)
- Fixed rare data race that can happen during `RENAME` table of MergeTree family. [#4844](#) (alexey-milovidov)
- Fixed segmentation fault in function `arrayIntersect`. Segmentation fault could happen if function was called with mixed constant and ordinary arguments. [#4847](#) (Lixiang Qian)
- Fixed reading from `Array(LowCardinality)` column in rare case when column contained a long sequence of empty arrays. [#4850](#) (Nikolai Kochetov)
- Fix `No message received` exception while fetching parts between replicas. [#4856](#) (alesapin)
- Fixed `arrayIntersect` function wrong result in case of several repeated values in single array. [#4871](#) (Nikolai Kochetov)
- Fix a race condition during concurrent `ALTER COLUMN` queries that could lead to a server crash (fixes issue [#3421](#)). [#4592](#) (Alex Zatelepin)
- Fix parameter deduction in `ALTER MODIFY` of column `CODEC` when column type is not specified. [#4883](#) (alesapin)
- Functions `cutQueryStringAndFragment()` and `queryStringAndFragment()` now works correctly when `URL` contains a fragment and no query. [#4894](#) (Vitaly Baranov)
- Fix rare bug when setting `min_bytes_to_use_direct_io` is greater than zero, which occurs when thread have to seek backward in column file. [#4897](#) (alesapin)
- Fix wrong argument types for aggregate functions with `LowCardinality` arguments (fixes issue [#4919](#)). [#4922](#) (Nikolai Kochetov)
- Fix function `toISOWeek` result for year 1970. [#4988](#) (alexey-milovidov)
- Fix `DROP`, `TRUNCATE` and `OPTIMIZE` queries duplication, when executed on `ON CLUSTER` for `ReplicatedMergeTree*` tables family. [#4991](#) (alesapin)

Improvements

- Keep ordinary, `DEFAULT`, `MATERIALIZED` and `ALIAS` columns in a single list (fixes issue [#2867](#)). [#4707](#) (Alex Zatelepin)

ClickHouse release 19.4.3.11, 2019-04-02

Bug Fixes

- Fix crash in `FULL/RIGHT JOIN` when we joining on nullable vs not nullable. [#4855](#) (Artem Zuikov)
- Fix segmentation fault in `clickhouse-copier`. [#4835](#) (proller)

Build/Testing/Packaging Improvement

- Add a way to launch clickhouse-server image from a custom user. [#4753](#) (Mikhail f. Shiryayev)

ClickHouse release 19.4.2.7, 2019-03-30

Bug Fixes

- Fixed reading from `Array(LowCardinality)` column in rare case when column contained a long sequence of empty arrays. [#4850](#) (Nikolai Kochetov)

ClickHouse release 19.4.1.3, 2019-03-19

Bug Fixes

- Fixed remote queries which contain both `LIMIT BY` and `LIMIT`. Previously, if `LIMIT BY` and `LIMIT` were used for remote query, `LIMIT` could happen before `LIMIT BY`, which led to too filtered result. [#4708](#) (Constantin S. Pan)

ClickHouse release 19.4.0.49, 2019-03-09

New Features

- Added full support for **Protobuf** format (input and output, nested data structures). [#4174](#) [#4493](#) (Vitaly Baranov)
- Added bitmap functions with Roaring Bitmaps. [#4207](#) (Andy Yang) [#4568](#) (Vitaly Baranov)
- Parquet format support. [#4448](#) (proller)
- N-gram distance was added for fuzzy string comparison. It is similar to q-gram metrics in R language. [#4466](#) (Danila Kutenin)
- Combine rules for graphite rollup from dedicated aggregation and retention patterns. [#4426](#) (Mikhail f. Shiryaev)
- Added **max_execution_speed** and **max_execution_speed_bytes** to limit resource usage. Added **min_execution_speed_bytes** setting to complement the **min_execution_speed**. [#4430](#) (Winter Zhang)
- Implemented function **flatten**. [#4555](#) [#4409](#) (alexey-milovidov, kzon)
- Added functions **arrayEnumerateDenseRanked** and **arrayEnumerateUniqRanked** (it's like **arrayEnumerateUniq** but allows to fine tune array depth to look inside multidimensional arrays). [#4475](#) (proller) [#4601](#) (alexey-milovidov)
- Multiple JOINS with some restrictions: no asterisks, no complex aliases in ON/WHERE/GROUP BY/... [#4462](#) (Artem Zuikov)

Bug Fixes

- This release also contains all bug fixes from 19.3 and 19.1.
- Fixed bug in data skipping indices: order of granules after INSERT was incorrect. [#4407](#) (Nikita Vasilev)
- Fixed **set** index for **Nullable** and **LowCardinality** columns. Before it, **set** index with **Nullable** or **LowCardinality** column led to error **Data type must be deserialized with multiple streams** while selecting. [#4594](#) (Nikolai Kochetov)
- Correctly set **update_time** on full **executable** dictionary update. [#4551](#) (Tema Novikov)
- Fix broken progress bar in 19.3. [#4627](#) (filimonov)
- Fixed inconsistent values of MemoryTracker when memory region was shrinked, in certain cases. [#4619](#) (alexey-milovidov)
- Fixed undefined behaviour in ThreadPool. [#4612](#) (alexey-milovidov)
- Fixed a very rare crash with the message **mutex lock failed: Invalid argument** that could happen when a MergeTree table was dropped concurrently with a SELECT. [#4608](#) (Alex Zatelepin)
- ODBC driver compatibility with **LowCardinality** data type. [#4381](#) (proller)
- FreeBSD: Fixup for **AIOcontextPool: Found io_event with unknown id 0** error. [#4438](#) (urgordeadbeef)
- **system.part_log** table was created regardless to configuration. [#4483](#) (alexey-milovidov)
- Fix undefined behaviour in **dictIsIn** function for cache dictionaries. [#4515](#) (alesapin)
- Fixed a deadlock when a SELECT query locks the same table multiple times (e.g. from different threads or when executing multiple subqueries) and there is a concurrent DDL query. [#4535](#) (Alex Zatelepin)
- Disable **compile_expressions** by default until we get own **llvm** contrib and can test it with **clang** and **asan**. [#4579](#) (alesapin)
- Prevent **std::terminate** when **invalidate_query** for **clickhouse** external dictionary source has returned wrong resultset (empty or more than one row or more than one column). Fixed issue when the **invalidate_query** was performed every five seconds regardless to the **lifetime**. [#4583](#) (alexey-milovidov)
- Avoid deadlock when the **invalidate_query** for a dictionary with **clickhouse** source was involving **system.dictionaries** table or **Dictionaries** database (rare case). [#4599](#) (alexey-milovidov)
- Fixes for CROSS JOIN with empty WHERE. [#4598](#) (Artem Zuikov)
- Fixed segfault in function "replicate" when constant argument is passed. [#4603](#) (alexey-milovidov)
- Fix lambda function with predicate optimizer. [#4408](#) (Winter Zhang)
- Multiple JOINS multiple fixes. [#4595](#) (Artem Zuikov)

Improvements

- Support aliases in JOIN ON section for right table columns. [#4412](#) (Artem Zuikov)
- Result of multiple JOINS need correct result names to be used in subselects. Replace flat aliases with source names in result. [#4474](#) (Artem Zuikov)
- Improve push-down logic for joined statements. [#4387](#) (Ivan)

Performance Improvements

- Improved heuristics of "move to PREWHERE" optimization. [#4405](#) (alexey-milovidov)

- Use proper lookup tables that uses HashTable's API for 8-bit and 16-bit keys. [#4536](#) (Amos Bird)
- Improved performance of string comparison. [#4564](#) (alexey-milovidov)
- Cleanup distributed DDL queue in a separate thread so that it doesn't slow down the main loop that processes distributed DDL tasks. [#4502](#) (Alex Zatelepin)
- When `min_bytes_to_use_direct_io` is set to 1, not every file was opened with `O_DIRECT` mode because the data size to read was sometimes underestimated by the size of one compressed block. [#4526](#) (alexey-milovidov)

Build/Testing/Packaging Improvement

- Added support for clang-9 [#4604](#) (alexey-milovidov)
- Fix wrong `__asm__` instructions (again) [#4621](#) (Konstantin Podshumok)
- Add ability to specify settings for `clickhouse-performance-test` from command line. [#4437](#) (alesapin)
- Add dictionaries tests to integration tests. [#4477](#) (alesapin)
- Added queries from the benchmark on the website to automated performance tests. [#4496](#) (alexey-milovidov)
- `xxhash.h` does not exist in external lz4 because it is an implementation detail and its symbols are namespaced with `XXH_NAMESPACE` macro. When lz4 is external, xxHash has to be external too, and the dependents have to link to it. [#4495](#) (Orivej Desh)
- Fixed a case when `quantileTiming` aggregate function can be called with negative or floating point argument (this fixes fuzz test with undefined behaviour sanitizer). [#4506](#) (alexey-milovidov)
- Spelling error correction. [#4531](#) (sdk2)
- Fix compilation on Mac. [#4371](#) (Vitaly Baranov)
- Build fixes for FreeBSD and various unusual build configurations. [#4444](#) (proller)

ClickHouse release 19.3.9.1, 2019-04-02

Bug Fixes

- Fix crash in `FULL/RIGHT JOIN` when we joining on nullable vs not nullable. [#4855](#) (Artem Zuikov)
- Fix segmentation fault in `clickhouse-copier`. [#4835](#) (proller)
- Fixed reading from `Array(LowCardinality)` column in rare case when column contained a long sequence of empty arrays. [#4850](#) (Nikolai Kochetov)

Build/Testing/Packaging Improvement

- Add a way to launch clickhouse-server image from a custom user [#4753](#) (Mikhail f. Shiryayev)

ClickHouse release 19.3.7, 2019-03-12

Bug fixes

- Fixed error in [#3920](#). This error manifestate itself as random cache corruption (messages `Unknown codec family code`, `Cannot seek through file`) and segfaults. This bug first appeared in version 19.1 and is present in versions up to 19.1.10 and 19.3.6. [#4623](#) (alexey-milovidov)

ClickHouse release 19.3.6, 2019-03-02

Bug fixes

- When there are more than 1000 threads in a thread pool, `std::terminate` may happen on thread exit. Azat Khuzhin [#4485](#) [#4505](#) (alexey-milovidov)
- Now it's possible to create `ReplicatedMergeTree*` tables with comments on columns without defaults and tables with columns codecs without comments and defaults. Also fix comparison of codecs. [#4523](#) (alesapin)
- Fixed crash on JOIN with array or tuple. [#4552](#) (Artem Zuikov)
- Fixed crash in clickhouse-copier with the message `ThreadStatus not created`. [#4540](#) (Artem Zuikov)
- Fixed hangup on server shutdown if distributed DDLs were used. [#4472](#) (Alex Zatelepin)
- Incorrect column numbers were printed in error message about text format parsing for columns with number greater than 10. [#4484](#) (alexey-milovidov)

Build/Testing/Packaging Improvements

- Fixed build with AVX enabled. [#4527](#) (alexey-milovidov)

- Enable extended accounting and IO accounting based on good known version instead of kernel under which it is compiled. [#4541](#) (nvartolomei)
- Allow to skip setting of core_dump.size_limit, warning instead of throw if limit set fail. [#4473](#) (proller)
- Removed the inline tags of void readBinary(...) in Field.cpp. Also merged redundant namespace DB blocks. [#4530](#) (hcz)

ClickHouse release 19.3.5, 2019-02-21

Bug fixes

- Fixed bug with large http insert queries processing. [#4454](#) (alesapin)
- Fixed backward incompatibility with old versions due to wrong implementation of send_logs_level setting. [#4445](#) (alexey-milovidov)
- Fixed backward incompatibility of table function remote introduced with column comments. [#4446](#) (alexey-milovidov)

ClickHouse release 19.3.4, 2019-02-16

Improvements

- Table index size is not accounted for memory limits when doing ATTACH TABLE query. Avoided the possibility that a table cannot be attached after being detached. [#4396](#) (alexey-milovidov)
- Slightly raised up the limit on max string and array size received from ZooKeeper. It allows to continue to work with increased size of CLIENT_JVMFLAGS=-Djute.maxbuffer=... on ZooKeeper. [#4398](#) (alexey-milovidov)
- Allow to repair abandoned replica even if it already has huge number of nodes in its queue. [#4399](#) (alexey-milovidov)
- Add one required argument to SET index (max stored rows number). [#4386](#) (Nikita Vasilev)

Bug Fixes

- Fixed WITH ROLLUP result for group by single LowCardinality key. [#4384](#) (Nikolai Kochetov)
- Fixed bug in the set index (dropping a granule if it contains more than max_rows rows). [#4386](#) (Nikita Vasilev)
- A lot of FreeBSD build fixes. [#4397](#) (proller)
- Fixed aliases substitution in queries with subquery containing same alias (issue [#4110](#)). [#4351](#) (Artem Zuikov)

Build/Testing/Packaging Improvements

- Add ability to run clickhouse-server for stateless tests in docker image. [#4347](#) (Vasily Nemkov)

ClickHouse release 19.3.3, 2019-02-13

New Features

- Added the KILL MUTATION statement that allows removing mutations that are for some reasons stuck. Added latest_failed_part, latest_fail_time, latest_fail_reason fields to the system.mutations table for easier troubleshooting. [#4287](#) (Alex Zatelepin)
- Added aggregate function entropy which computes Shannon entropy. [#4238](#) (Quid37)
- Added ability to send queries INSERT INTO tbl VALUES (...) to server without splitting on query and data parts. [#4301](#) (alesapin)
- Generic implementation of arrayWithConstant function was added. [#4322](#) (alexey-milovidov)
- Implented NOT BETWEEN comparison operator. [#4228](#) (Dmitry Naumov)
- Implement sumMapFiltered in order to be able to limit the number of keys for which values will be summed by sumMap. [#4129](#) (Léo Ercolanelli)
- Added support of Nullable types in mysql table function. [#4198](#) (Emmanuel Donin de Rosière)
- Support for arbitrary constant expressions in LIMIT clause. [#4246](#) (k3box)
- Added topKWeighted aggregate function that takes additional argument with (unsigned integer) weight. [#4245](#) (Andrew Golman)
- StorageJoin now supports join_any_take_last_row setting that allows overwriting existing values of the same key. [#3973](#) (Amos Bird)
- Added function toStartOfInterval. [#4304](#) (Vitaly Baranov)

- Added `RowBinaryWithNamesAndTypes` format. #4200 (Oleg V. Kozlyuk)
- Added `IPv4` and `IPv6` data types. More effective implementations of `IPv*` functions. #3669 (Vasily Nemkov)
- Added function `toStartOfTenMinutes()`. #4298 (Vitaly Baranov)
- Added `Protobuf` output format. #4005 #4158 (Vitaly Baranov)
- Added brotli support for HTTP interface for data import (INSERTs). #4235 (Mikhail)
- Added hints while user make typo in function name or type in command line client. #4239 (Danila Kutenin)
- Added `Query-Id` to Server's HTTP Response header. #4231 (Mikhail)

Experimental features

- Added `minmax` and `set` data skipping indices for MergeTree table engines family. #4143 (Nikita Vasilev)
- Added conversion of `CROSS JOIN` to `INNER JOIN` if possible. #4221 #4266 (Artem Zuikov)

Bug Fixes

- Fixed `Not found column` for duplicate columns in `JOIN ON` section. #4279 (Artem Zuikov)
- Make `START REPLICATED SENDS` command start replicated sends. #4229 (nvartolomei)
- Fixed aggregate functions execution with `Array(LowCardinality)` arguments. #4055 (KochetovNicolai)
- Fixed wrong behaviour when doing `INSERT ... SELECT ... FROM file(...)` query and file has `CSVWithNames` or `TSVWithNames` format and the first data row is missing. #4297 (alexey-milovidov)
- Fixed crash on dictionary reload if dictionary not available. This bug was appeared in 19.1.6. #4188 (proller)
- Fixed `ALL JOIN` with duplicates in right table. #4184 (Artem Zuikov)
- Fixed segmentation fault with `use_uncompressed_cache=1` and exception with wrong uncompressed size. This bug was appeared in 19.1.6. #4186 (alesapin)
- Fixed `compile_expressions` bug with comparison of big (more than int16) dates. #4341 (alesapin)
- Fixed infinite loop when selecting from table function `numbers(0)`. #4280 (alexey-milovidov)
- Temporarily disable predicate optimization for `ORDER BY`. #3890 (Winter Zhang)
- Fixed `Illegal instruction` error when using base64 functions on old CPUs. This error has been reproduced only when ClickHouse was compiled with gcc-8. #4275 (alexey-milovidov)
- Fixed `No message received` error when interacting with PostgreSQL ODBC Driver through TLS connection. Also fixes segfault when using MySQL ODBC Driver. #4170 (alexey-milovidov)
- Fixed incorrect result when `Date` and `DateTime` arguments are used in branches of conditional operator (function `if`). Added generic case for function `if`. #4243 (alexey-milovidov)
- ClickHouse dictionaries now load within `clickhouse` process. #4166 (alexey-milovidov)
- Fixed deadlock when `SELECT` from a table with `File` engine was retried after `No such file or directory` error. #4161 (alexey-milovidov)
- Fixed race condition when selecting from `system.tables` may give `table doesn't exist` error. #4313 (alexey-milovidov)
- `clickhouse-client` can segfault on exit while loading data for command line suggestions if it was run in interactive mode. #4317 (alexey-milovidov)
- Fixed a bug when the execution of mutations containing `IN` operators was producing incorrect results. #4099 (Alex Zatelepin)
- Fixed error: if there is a database with `Dictionary` engine, all dictionaries forced to load at server startup, and if there is a dictionary with ClickHouse source from localhost, the dictionary cannot load. #4255 (alexey-milovidov)
- Fixed error when system logs are tried to create again at server shutdown. #4254 (alexey-milovidov)
- Correctly return the right type and properly handle locks in `joinGet` function. #4153 (Amos Bird)
- Added `sumMapWithOverflow` function. #4151 (Léo Ercolanelli)
- Fixed segfault with `allow_experimental_multiple_joins_emulation`. 52de2c (Artem Zuikov)
- Fixed bug with incorrect `Date` and `DateTime` comparison. #4237 (valexey)
- Fixed fuzz test under undefined behavior sanitizer: added parameter type check for `quantile*Weighted` family of functions. #4145 (alexey-milovidov)
- Fixed rare race condition when removing of old data parts can fail with `File not found` error. #4378 (alexey-milovidov)
- Fix install package with missing `/etc/clickhouse-server/config.xml`. #4343 (proller)

Build/Testing/Packaging Improvements

- Debian package: correct /etc/clickhouse-server/preprocessed link according to config. [#4205](#) (proller)
- Various build fixes for FreeBSD. [#4225](#) (proller)
- Added ability to create, fill and drop tables in perfctest. [#4220](#) (alesapin)
- Added a script to check for duplicate includes. [#4326](#) (alexey-milovidov)
- Added ability to run queries by index in performance test. [#4264](#) (alesapin)
- Package with debug symbols is suggested to be installed. [#4274](#) (alexey-milovidov)
- Refactoring of performance-test. Better logging and signals handling. [#4171](#) (alesapin)
- Added docs to anonymized Yandex.Metrika datasets. [#4164](#) (alesapin)
- Added tool for converting an old month-partitioned part to the custom-partitioned format. [#4195](#) (Alex Zatelepin)
- Added docs about two datasets in s3. [#4144](#) (alesapin)
- Added script which creates changelog from pull requests description. [#4169](#) [#4173](#) (KochetovNicolai) (KochetovNicolai)
- Added puppet module for Clickhouse. [#4182](#) (Maxim Fedotov)
- Added docs for a group of undocumented functions. [#4168](#) (Winter Zhang)
- ARM build fixes. [#4210](#)[#4306](#) [#4291](#) (proller) (proller)
- Dictionary tests now able to run from ctest. [#4189](#) (proller)
- Now /etc/ssl is used as default directory with SSL certificates. [#4167](#) (alexey-milovidov)
- Added checking SSE and AVX instruction at start. [#4234](#) (lgr)
- Init script will wait server until start. [#4281](#) (proller)

Backward Incompatible Changes

- Removed `allow_experimental_low_cardinality_type` setting. `LowCardinality` data types are production ready. [#4323](#) (alexey-milovidov)
- Reduce mark cache size and uncompressed cache size accordingly to available memory amount. [#4240](#) (Lopatin Konstantin)
- Added keyword `INDEX` in `CREATE TABLE` query. A column with name `index` must be quoted with backticks or double quotes: ``index``. [#4143](#) (Nikita Vasilev)
- `sumMap` now promote result type instead of overflow. The old `sumMap` behavior can be obtained by using `sumMapWithOverflow` function. [#4151](#) (Léo Ercolanelli)

Performance Improvements

- `std::sort` replaced by `pdqsort` for queries without `LIMIT`. [#4236](#) (Evgenii Pravda)
- Now server reuse threads from global thread pool. This affects performance in some corner cases. [#4150](#) (alexey-milovidov)

Improvements

- Implemented AIO support for FreeBSD. [#4305](#) (urgordeadbeef)
- `SELECT * FROM a JOIN b USING a, b` now return `a` and `b` columns only from the left table. [#4141](#) (Artem Zuikov)
- Allow `-C` option of client to work as `-c` option. [#4232](#) (syominsergey)
- Now option `--password` used without value requires password from stdin. [#4230](#) (BSD_Conqueror)
- Added highlighting of unescaped metacharacters in string literals that contain `LIKE` expressions or regexps. [#4327](#) (alexey-milovidov)
- Added cancelling of HTTP read only queries if client socket goes away. [#4213](#) (nvartolomei)
- Now server reports progress to keep client connections alive. [#4215](#) (Ivan)
- Slightly better message with reason for OPTIMIZE query with `optimize_throw_if_noop` setting enabled. [#4294](#) (alexey-milovidov)
- Added support of `--version` option for clickhouse server. [#4251](#) (Lopatin Konstantin)
- Added `--help/-h` option to `clickhouse-server`. [#4233](#) (Yuriy Baranov)
- Added support for scalar subqueries with aggregate function state result. [#4348](#) (Nikolai Kochetov)
- Improved server shutdown time and ALTERs waiting time. [#4372](#) (alexey-milovidov)
- Added info about the `replicated_can_become_leader` setting to `system.replicas` and add logging if the replica won't try to become leader. [#4379](#) (Alex Zatelepin)

ClickHouse release 19.1.14, 2019-03-14

- Fixed error `Column ... queried more than once` that may happen if the setting `asterisk_left_columns_only` is set to 1 in case of using `GLOBAL JOIN` with `SELECT *` (rare case). The issue does not exist in 19.3 and newer. [6bac7d8d](#) (Artem Zuikov)

ClickHouse release 19.1.13, 2019-03-12

This release contains exactly the same set of patches as 19.3.7.

ClickHouse release 19.1.10, 2019-03-03

This release contains exactly the same set of patches as 19.3.6.

ClickHouse release 19.1.9, 2019-02-21

Bug fixes

- Fixed backward incompatibility with old versions due to wrong implementation of `send_logs_level` setting. [#4445](#) (alexey-milovidov)
- Fixed backward incompatibility of table function `remote` introduced with column comments. [#4446](#) (alexey-milovidov)

ClickHouse release 19.1.8, 2019-02-16

Bug Fixes

- Fix install package with missing `/etc/clickhouse-server/config.xml`. [#4343](#) (proller)

ClickHouse release 19.1.7, 2019-02-15

Bug Fixes

- Correctly return the right type and properly handle locks in `joinGet` function. [#4153](#) (Amos Bird)
- Fixed error when system logs are tried to create again at server shutdown. [#4254](#) (alexey-milovidov)
- Fixed error: if there is a database with `Dictionary` engine, all dictionaries forced to load at server startup, and if there is a dictionary with ClickHouse source from localhost, the dictionary cannot load. [#4255](#) (alexey-milovidov)
- Fixed a bug when the execution of mutations containing `IN` operators was producing incorrect results. [#4099](#) (Alex Zatelepin)
- `clickhouse-client` can segfault on exit while loading data for command line suggestions if it was run in interactive mode. [#4317](#) (alexey-milovidov)
- Fixed race condition when selecting from `system.tables` may give `table doesn't exist` error. [#4313](#) (alexey-milovidov)
- Fixed deadlock when `SELECT` from a table with `File` engine was retried after `No such file or directory` error. [#4161](#) (alexey-milovidov)
- Fixed an issue: local ClickHouse dictionaries are loaded via TCP, but should load within process. [#4166](#) (alexey-milovidov)
- Fixed `No message received` error when interacting with PostgreSQL ODBC Driver through TLS connection. Also fixes segfault when using MySQL ODBC Driver. [#4170](#) (alexey-milovidov)
- Temporarily disable predicate optimization for `ORDER BY`. [#3890](#) (Winter Zhang)
- Fixed infinite loop when selecting from table function `numbers(0)`. [#4280](#) (alexey-milovidov)
- Fixed `compile_expressions` bug with comparison of big (more than int16) dates. [#4341](#) (alesapin)
- Fixed segmentation fault with `uncompressed_cache=1` and exception with wrong uncompressed size. [#4186](#) (alesapin)
- Fixed `ALL JOIN` with duplicates in right table. [#4184](#) (Artem Zuikov)
- Fixed wrong behaviour when doing `INSERT ... SELECT ... FROM file(...)` query and file has `CSVWithNames` or `TSVWithNames` format and the first data row is missing. [#4297](#) (alexey-milovidov)
- Fixed aggregate functions execution with `Array(LowCardinality)` arguments. [#4055](#) (KochetovNicolai)
- Debian package: correct `/etc/clickhouse-server/preprocessed` link according to config. [#4205](#) (proller)
- Fixed fuzz test under undefined behavior sanitizer: added parameter type check for `quantile*Weighted` family of functions. [#4145](#) (alexey-milovidov)

- Make `START REPLICATED SENDS` command start replicated sends. [#4229](#) (nvartolomei)
- Fixed `Not found column` for duplicate columns in JOIN ON section. [#4279](#) (Artem Zuikov)
- Now `/etc/ssl` is used as default directory with SSL certificates. [#4167](#) (alexey-milovidov)
- Fixed crash on dictionary reload if dictionary not available. [#4188](#) (proller)
- Fixed bug with incorrect `Date` and `DateTime` comparison. [#4237](#) (valexey)
- Fixed incorrect result when `Date` and `DateTime` arguments are used in branches of conditional operator (function `if`). Added generic case for function `if`. [#4243](#) (alexey-milovidov)

ClickHouse release 19.1.6, 2019-01-24

New Features

- Custom per column compression codecs for tables. [#3899](#) [#4111](#) (alesapin, Winter Zhang, Anatoly)
- Added compression codec `Delta`. [#4052](#) (alesapin)
- Allow to `ALTER` compression codecs. [#4054](#) (alesapin)
- Added functions `left`, `right`, `trim`, `ltrim`, `rtrim`, `timestampadd`, `timestampsub` for SQL standard compatibility. [#3826](#) (Ivan Blinkov)
- Support for write in `HDFS` tables and `hdfs` table function. [#4084](#) (alesapin)
- Added functions to search for multiple constant strings from big haystack: `multiPosition`, `multiSearch`, `firstMatch` also with `-UTF8`, `-CaseInsensitive`, and `-CaseInsensitiveUTF8` variants. [#4053](#) (Danila Kutenin)
- Pruning of unused shards if `SELECT` query filters by sharding key (setting `optimize_skip_unused_shards`). [#3851](#) (Gleb Kanterov, Ivan)
- Allow `Kafka` engine to ignore some number of parsing errors per block. [#4094](#) (Ivan)
- Added support for `CatBoost` multiclass models evaluation. Function `modelEvaluate` returns tuple with per-class raw predictions for multiclass models. `libcatboostmodel.so` should be built with [#607](#). [#3959](#) (KochetovNicolai)
- Added functions `filesystemAvailable`, `filesystemFree`, `filesystemCapacity`. [#4097](#) (Boris Granveaud)
- Added hashing functions `xxHash64` and `xxHash32`. [#3905](#) (filimonov)
- Added `gccMurmurHash` hashing function (GCC flavoured Murmur hash) which uses the same hash seed as `gcc` [#4000](#) (sundyli)
- Added hashing functions `javaHash`, `hiveHash`. [#3811](#) (shangshujie365)
- Added table function `remoteSecure`. Function works as `remote`, but uses secure connection. [#4088](#) (proller)

Experimental features

- Added multiple JOINS emulation (`allow_experimental_multiple_joins_emulation` setting). [#3946](#) (Artem Zuikov)

Bug Fixes

- Make `compiled_expression_cache_size` setting limited by default to lower memory consumption. [#4041](#) (alesapin)
- Fix a bug that led to hangups in threads that perform ALTERs of Replicated tables and in the thread that updates configuration from ZooKeeper. [#2947](#) [#3891](#) [#3934](#) (Alex Zatelepin)
- Fixed a race condition when executing a distributed ALTER task. The race condition led to more than one replica trying to execute the task and all replicas except one failing with a ZooKeeper error. [#3904](#) (Alex Zatelepin)
- Fix a bug when `from_zk` config elements weren't refreshed after a request to ZooKeeper timed out. [#2947](#) [#3947](#) (Alex Zatelepin)
- Fix bug with wrong prefix for IPv4 subnet masks. [#3945](#) (alesapin)
- Fixed crash (`std::terminate`) in rare cases when a new thread cannot be created due to exhausted resources. [#3956](#) (alexey-milovidov)
- Fix bug when in `remote` table function execution when wrong restrictions were used for in `getStructureOfRemoteTable`. [#4009](#) (alesapin)
- Fix a leak of netlink sockets. They were placed in a pool where they were never deleted and new sockets were created at the start of a new thread when all current sockets were in use. [#4017](#) (Alex Zatelepin)
- Fix bug with closing `/proc/self/fd` directory earlier than all fds were read from `/proc` after forking `odbc-bridge` subprocess. [#4120](#) (alesapin)
- Fixed String to UInt monotonic conversion in case of usage String in primary key. [#3870](#) (Winter Zhang)
- Fixed error in calculation of integer conversion function monotonicity. [#3921](#) (alexey-milovidov)
- Fixed segfault in `arrayEnumerateUniq`, `arrayEnumerateDense` functions in case of some invalid arguments. [#3909](#) (alexey-milovidov)

- Fix UB in StorageMerge. [#3910](#) (Amos Bird)
- Fixed segfault in functions `addDays`, `subtractDays`. [#3913](#) (alexey-milovidov)
- Fixed error: functions `round`, `floor`, `trunc`, `ceil` may return bogus result when executed on integer argument and large negative scale. [#3914](#) (alexey-milovidov)
- Fixed a bug induced by 'kill query sync' which leads to a core dump. [#3916](#) (muVulDeePecker)
- Fix bug with long delay after empty replication queue. [#3928](#) [#3932](#) (alesapin)
- Fixed excessive memory usage in case of inserting into table with `LowCardinality` primary key. [#3955](#) (KochetovNicolai)
- Fixed `LowCardinality` serialization for `Native` format in case of empty arrays. [#3907](#) [#4011](#) (KochetovNicolai)
- Fixed incorrect result while using `distinct` by single `LowCardinality` numeric column. [#3895](#) [#4012](#) (KochetovNicolai)
- Fixed specialized aggregation with `LowCardinality` key (in case when `compile` setting is enabled). [#3886](#) (KochetovNicolai)
- Fix user and password forwarding for replicated tables queries. [#3957](#) (alesapin) (小路)
- Fixed very rare race condition that can happen when listing tables in Dictionary database while reloading dictionaries. [#3970](#) (alexey-milovidov)
- Fixed incorrect result when `HAVING` was used with `ROLLUP` or `CUBE`. [#3756](#) [#3837](#) (Sam Chou)
- Fixed column aliases for query with `JOIN ON` syntax and distributed tables. [#3980](#) (Winter Zhang)
- Fixed error in internal implementation of `quantileTDigest` (found by Artem Vakhrushev). This error never happens in ClickHouse and was relevant only for those who use ClickHouse codebase as a library directly. [#3935](#) (alexey-milovidov)

Improvements

- Support for `IF NOT EXISTS` in `ALTER TABLE ADD COLUMN` statements along with `IF EXISTS` in `DROP/MODIFY/CLEAR/COMMENT COLUMN`. [#3900](#) (Boris Granveaud)
- Function `parseDateTimeBestEffort`: support for formats `DD.MM.YYYY`, `DD.MM.YY`, `DD-MM-YYYY`, `DD-Mon-YYYY`, `DD/Month/YYYY` and similar. [#3922](#) (alexey-milovidov)
- `CapnProtoInputStream` now support jagged structures. [#4063](#) (Odin Hultgren Van Der Horst)
- Usability improvement: added a check that server process is started from the data directory's owner. Do not allow to start server from root if the data belongs to non-root user. [#3785](#) (sergey-v-galtsev)
- Better logic of checking required columns during analysis of queries with `JOINS`. [#3930](#) (Artem Zuikov)
- Decreased the number of connections in case of large number of Distributed tables in a single server. [#3726](#) (Winter Zhang)
- Supported totals row for `WITH TOTALS` query for ODBC driver. [#3836](#) (Maksim Koritckiy)
- Allowed to use `Enums` as integers inside `if` function. [#3875](#) (Ivan)
- Added `low_cardinality_allow_in_native_format` setting. If disabled, do not use `LowCardinality` type in `Native` format. [#3879](#) (KochetovNicolai)
- Removed some redundant objects from compiled expressions cache to lower memory usage. [#4042](#) (alesapin)
- Add check that `SET send_logs_level = 'value'` query accept appropriate value. [#3873](#) (Sabyanin Maxim)
- Fixed data type check in type conversion functions. [#3896](#) (Winter Zhang)

Performance Improvements

- Add a MergeTree setting `use_minimalistic_part_header_in_zookeeper`. If enabled, Replicated tables will store compact part metadata in a single part znode. This can dramatically reduce ZooKeeper snapshot size (especially if the tables have a lot of columns). Note that after enabling this setting you will not be able to downgrade to a version that doesn't support it. [#3960](#) (Alex Zatelepin)
- Add an DFA-based implementation for functions `sequenceMatch` and `sequenceCount` in case pattern doesn't contain time. [#4004](#) (Léo Ercolanelli)
- Performance improvement for integer numbers serialization. [#3968](#) (Amos Bird)
- Zero left padding `PODArray` so that -1 element is always valid and zeroed. It's used for branchless calculation of offsets. [#3920](#) (Amos Bird)
- Reverted `jemalloc` version which lead to performance degradation. [#4018](#) (alexey-milovidov)

Backward Incompatible Changes

- Removed undocumented feature `ALTER MODIFY PRIMARY KEY` because it was superseded by the `ALTER MODIFY ORDER BY` command. [#3887](#) (Alex Zatelepin)
- Removed function `shardByHash`. [#3833](#) (alexey-milovidov)
- Forbid using scalar subqueries with result of type `AggregateFunction`. [#3865](#) (Ivan)

Build/Testing/Packaging Improvements

- Added support for PowerPC (`ppc64le`) build. [#4132](#) (Danila Kutenin)
- Stateful functional tests are run on public available dataset. [#3969](#) (alexey-milovidov)
- Fixed error when the server cannot start with the `bash: /usr/bin/clickhouse-extract-from-config: Operation not permitted` message within Docker or `systemd-nspawn`. [#4136](#) (alexey-milovidov)
- Updated `rdkafka` library to v1.0.0-RC5. Used `cppkafka` instead of raw C interface. [#4025](#) (Ivan)
- Updated `mariadb-client` library. Fixed one of issues found by UBSan. [#3924](#) (alexey-milovidov)
- Some fixes for UBSan builds. [#3926](#) [#3021](#) [#3948](#) (alexey-milovidov)
- Added per-commit runs of tests with UBSan build.
- Added per-commit runs of PVS-Studio static analyzer.
- Fixed bugs found by PVS-Studio. [#4013](#) (alexey-milovidov)
- Fixed glibc compatibility issues. [#4100](#) (alexey-milovidov)
- Move Docker images to 18.10 and add compatibility file for glibc ≥ 2.28 [#3965](#) (alesapin)
- Add env variable if user don't want to chown directories in server Docker image. [#3967](#) (alesapin)
- Enabled most of the warnings from `-Weverything` in clang. Enabled `-Wpedantic`. [#3986](#) (alexey-milovidov)
- Added a few more warnings that are available only in clang 8. [#3993](#) (alexey-milovidov)
- Link to `libLLVM` rather than to individual LLVM libs when using shared linking. [#3989](#) (Orivej Desh)
- Added sanitizer variables for test images. [#4072](#) (alesapin)
- `clickhouse-server` debian package will recommend `libcap2-bin` package to use `setcap` tool for setting capabilities. This is optional. [#4093](#) (alexey-milovidov)
- Improved compilation time, fixed includes. [#3898](#) (proller)
- Added performance tests for hash functions. [#3918](#) (filimonov)
- Fixed cyclic library dependences. [#3958](#) (proller)
- Improved compilation with low available memory. [#4030](#) (proller)
- Added test script to reproduce performance degradation in `jemalloc`. [#4036](#) (alexey-milovidov)
- Fixed misspells in comments and string literals under `dbms`. [#4122](#) (maiha)
- Fixed typos in comments. [#4089](#) (Evgenii Pravda)

ClickHouse release 18.16.1, 2018-12-21

Bug fixes:

- Fixed an error that led to problems with updating dictionaries with the ODBC source. [#3825](#), [#3829](#)
- JIT compilation of aggregate functions now works with LowCardinality columns. [#3838](#)

Improvements:

- Added the `low_cardinality_allow_in_native_format` setting (enabled by default). When disabled, LowCardinality columns will be converted to ordinary columns for SELECT queries and ordinary columns will be expected for INSERT queries. [#3879](#)

Build improvements:

- Fixes for builds on macOS and ARM.

ClickHouse release 18.16.0, 2018-12-14

New features:

- `DEFAULT` expressions are evaluated for missing fields when loading data in semi-structured input formats (`JSONEachRow`, `TSKV`). The feature is enabled with the `insert_sample_with_metadata` setting. [#3555](#)
- The `ALTER TABLE` query now has the `MODIFY ORDER BY` action for changing the sorting key when adding or removing a table column. This is useful for tables in the `MergeTree` family that perform additional tasks when merging based on this sorting key, such as `SummingMergeTree`, `AggregatingMergeTree`, and so on. [#3581](#) [#3755](#)

- For tables in the `MergeTree` family, now you can specify a different sorting key (`ORDER BY`) and index (`PRIMARY KEY`). The sorting key can be longer than the index. [#3581](#)
- Added the `hdfs` table function and the `HDFS` table engine for importing and exporting data to HDFS. [chenxing-xc](#)
- Added functions for working with base64: `base64Encode`, `base64Decode`, `tryBase64Decode`. [Alexander Krashennnikov](#)
- Now you can use a parameter to configure the precision of the `uniqCombined` aggregate function (select the number of HyperLogLog cells). [#3406](#)
- Added the `system.contributors` table that contains the names of everyone who made commits in ClickHouse. [#3452](#)
- Added the ability to omit the partition for the `ALTER TABLE ... FREEZE` query in order to back up all partitions at once. [#3514](#)
- Added `dictGet` and `dictGetOrDefault` functions that don't require specifying the type of return value. The type is determined automatically from the dictionary description. [Amos Bird](#)
- Now you can specify comments for a column in the table description and change it using `ALTER`. [#3377](#)
- Reading is supported for `Join` type tables with simple keys. [Amos Bird](#)
- Now you can specify the options `join_use_nulls`, `max_rows_in_join`, `max_bytes_in_join`, and `join_overflow_mode` when creating a `Join` type table. [Amos Bird](#)
- Added the `joinGet` function that allows you to use a `Join` type table like a dictionary. [Amos Bird](#)
- Added the `partition_key`, `sorting_key`, `primary_key`, and `sampling_key` columns to the `system.tables` table in order to provide information about table keys. [#3609](#)
- Added the `is_in_partition_key`, `is_in_sorting_key`, `is_in_primary_key`, and `is_in_sampling_key` columns to the `system.columns` table. [#3609](#)
- Added the `min_time` and `max_time` columns to the `system.parts` table. These columns are populated when the partitioning key is an expression consisting of `DateTime` columns. [Emmanuel Donin de Rosière](#)

Bug fixes:

- Fixes and performance improvements for the `LowCardinality` data type. `GROUP BY` using `LowCardinality(Nullable(...))`. Getting the values of `extremes`. Processing high-order functions. `LEFT ARRAY JOIN`. Distributed `GROUP BY`. Functions that return `Array`. Execution of `ORDER BY`. Writing to `Distributed` tables (`nicelulu`). Backward compatibility for `INSERT` queries from old clients that implement the `Native` protocol. Support for `LowCardinality` for `JOIN`. Improved performance when working in a single stream. [#3823](#) [#3803](#) [#3799](#) [#3769](#) [#3744](#) [#3681](#) [#3651](#) [#3649](#) [#3641](#) [#3632](#) [#3568](#) [#3523](#) [#3518](#)
- Fixed how the `select_sequential_consistency` option works. Previously, when this setting was enabled, an incomplete result was sometimes returned after beginning to write to a new partition. [#2863](#)
- Databases are correctly specified when executing DDL `ON CLUSTER` queries and `ALTER UPDATE/DELETE`. [#3772](#) [#3460](#)
- Databases are correctly specified for subqueries inside a `VIEW`. [#3521](#)
- Fixed a bug in `PREWHERE` with `FINAL` for `VersionedCollapsingMergeTree`. [7167bfd7](#)
- Now you can use `KILL QUERY` to cancel queries that have not started yet because they are waiting for the table to be locked. [#3517](#)
- Corrected date and time calculations if the clocks were moved back at midnight (this happens in Iran, and happened in Moscow from 1981 to 1983). Previously, this led to the time being reset a day earlier than necessary, and also caused incorrect formatting of the date and time in text format. [#3819](#)
- Fixed bugs in some cases of `VIEW` and subqueries that omit the database. [Winter Zhang](#)
- Fixed a race condition when simultaneously reading from a `MATERIALIZED VIEW` and deleting a `MATERIALIZED VIEW` due to not locking the internal `MATERIALIZED VIEW`. [#3404](#) [#3694](#)
- Fixed the error `Lock handler cannot be nullptr`. [#3689](#)
- Fixed query processing when the `compile_expressions` option is enabled (it's enabled by default). Nondeterministic constant expressions like the `now` function are no longer unfolded. [#3457](#)
- Fixed a crash when specifying a non-constant scale argument in `toDecimal32/64/128` functions.
- Fixed an error when trying to insert an array with `NULL` elements in the `Values` format into a column of type `Array` without `Nullable` (if `input_format_values_interpret_expressions` = 1). [#3487](#) [#3503](#)
- Fixed continuous error logging in `DDLWorker` if ZooKeeper is not available. [8f50c620](#)
- Fixed the return type for `quantile*` functions from `Date` and `DateTime` types of arguments. [#3580](#)
- Fixed the `WITH` clause if it specifies a simple alias without expressions. [#3570](#)

- Fixed processing of queries with named sub-queries and qualified column names when `enable_optimize_predicate_expression` is enabled. [Winter Zhang](#)
- Fixed the error `Attempt to attach to nullptr thread group` when working with materialized views. [Marek Vavruša](#)
- Fixed a crash when passing certain incorrect arguments to the `arrayReverse` function. [73e3a7b6](#)
- Fixed the buffer overflow in the `extractURLParameter` function. Improved performance. Added correct processing of strings containing zero bytes. [141e9799](#)
- Fixed buffer overflow in the `lowerUTF8` and `upperUTF8` functions. Removed the ability to execute these functions over `FixedString` type arguments. [#3662](#)
- Fixed a rare race condition when deleting `MergeTree` tables. [#3680](#)
- Fixed a race condition when reading from `Buffer` tables and simultaneously performing `ALTER` or `DROP` on the target tables. [#3719](#)
- Fixed a segfault if the `max_temporary_non_const_columns` limit was exceeded. [#3788](#)

Improvements:

- The server does not write the processed configuration files to the `/etc/clickhouse-server/` directory. Instead, it saves them in the `preprocessed_configs` directory inside `path`. This means that the `/etc/clickhouse-server/` directory doesn't have write access for the `clickhouse` user, which improves security. [#2443](#)
- The `min_merge_bytes_to_use_direct_io` option is set to 10 GiB by default. A merge that forms large parts of tables from the `MergeTree` family will be performed in `O_DIRECT` mode, which prevents excessive page cache eviction. [#3504](#)
- Accelerated server start when there is a very large number of tables. [#3398](#)
- Added a connection pool and HTTP `Keep-Alive` for connections between replicas. [#3594](#)
- If the query syntax is invalid, the `400 Bad Request` code is returned in the `HTTP` interface (500 was returned previously). [31bc680a](#)
- The `join_default_strictness` option is set to `ALL` by default for compatibility. [120e2cbe](#)
- Removed logging to `stderr` from the `re2` library for invalid or complex regular expressions. [#3723](#)
- Added for the `Kafka` table engine: checks for subscriptions before beginning to read from Kafka; the `kafka_max_block_size` setting for the table. [Marek Vavruša](#)
- The `cityHash64`, `farmHash64`, `metroHash64`, `sipHash64`, `halfMD5`, `murmurHash2_32`, `murmurHash2_64`, `murmurHash3_32`, and `murmurHash3_64` functions now work for any number of arguments and for arguments in the form of tuples. [#3451](#) [#3519](#)
- The `arrayReverse` function now works with any types of arrays. [73e3a7b6](#)
- Added an optional parameter: the slot size for the `timeSlots` function. [Kirill Shvakov](#)
- For `FULL` and `RIGHT JOIN`, the `max_block_size` setting is used for a stream of non-joined data from the right table. [Amos Bird](#)
- Added the `--secure` command line parameter in `clickhouse-benchmark` and `clickhouse-performance-test` to enable TLS. [#3688](#) [#3690](#)
- Type conversion when the structure of a `Buffer` type table does not match the structure of the destination table. [Vitaly Baranov](#)
- Added the `tcp_keep_alive_timeout` option to enable keep-alive packets after inactivity for the specified time interval. [#3441](#)
- Removed unnecessary quoting of values for the partition key in the `system.parts` table if it consists of a single column. [#3652](#)
- The modulo function works for `Date` and `DateTime` data types. [#3385](#)
- Added synonyms for the `POWER`, `LN`, `LCASE`, `UCASE`, `REPLACE`, `LOCATE`, `SUBSTR`, and `MID` functions. [#3774](#) [#3763](#)
Some function names are case-insensitive for compatibility with the SQL standard. Added syntactic sugar `SUBSTRING(expr FROM start FOR length)` for compatibility with SQL. [#3804](#)
- Added the ability to `mlock` memory pages corresponding to `clickhouse-server` executable code to prevent it from being forced out of memory. This feature is disabled by default. [#3553](#)
- Improved performance when reading from `O_DIRECT` (with the `min_bytes_to_use_direct_io` option enabled). [#3405](#)
- Improved performance of the `dictGet...OrDefault` function for a constant key argument and a non-constant default argument. [Amos Bird](#)
- The `firstSignificantSubdomain` function now processes the domains `gov`, `mil`, and `edu`. [Igor Hatarist](#) Improved performance. [#3628](#)

- Ability to specify custom environment variables for starting `clickhouse-server` using the `SYS-V init.d` script by defining `CLICKHOUSE_PROGRAM_ENV` in `/etc/default/clickhouse`. [Pavlo Bashynskyi](#)
- Correct return code for the `clickhouse-server` init script. [#3516](#)
- The `system.metrics` table now has the `VersionInteger` metric, and `system.build_options` has the added line `VERSION_INTEGER`, which contains the numeric form of the ClickHouse version, such as `18016000`. [#3644](#)
- Removed the ability to compare the `Date` type with a number to avoid potential errors like `date = 2018-12-17`, where quotes around the date are omitted by mistake. [#3687](#)
- Fixed the behavior of stateful functions like `rowNumberInAllBlocks`. They previously output a result that was one number larger due to starting during query analysis. [Amos Bird](#)
- If the `force_restore_data` file can't be deleted, an error message is displayed. [Amos Bird](#)

Build improvements:

- Updated the `jemalloc` library, which fixes a potential memory leak. [Amos Bird](#)
- Profiling with `jemalloc` is enabled by default in order to debug builds. [2cc82f5c](#)
- Added the ability to run integration tests when only `Docker` is installed on the system. [#3650](#)
- Added the fuzz expression test in `SELECT` queries. [#3442](#)
- Added a stress test for commits, which performs functional tests in parallel and in random order to detect more race conditions. [#3438](#)
- Improved the method for starting `clickhouse-server` in a Docker image. [Elghazal Ahmed](#)
- For a Docker image, added support for initializing databases using files in the `/docker-entrypoint-initdb.d` directory. [Konstantin Lebedev](#)
- Fixes for builds on ARM. [#3709](#)

Backward incompatible changes:

- Removed the ability to compare the `Date` type with a number. Instead of `toDate('2018-12-18') = 17883`, you must use explicit type conversion `= toDate(17883)` [#3687](#)

ClickHouse release 18.14.19, 2018-12-19

Bug fixes:

- Fixed an error that led to problems with updating dictionaries with the ODBC source. [#3825](#), [#3829](#)
- Databases are correctly specified when executing DDL `ON CLUSTER` queries. [#3460](#)
- Fixed a segfault if the `max_temporary_non_const_columns` limit was exceeded. [#3788](#)

Build improvements:

- Fixes for builds on ARM.

ClickHouse release 18.14.18, 2018-12-04

Bug fixes:

- Fixed error in `dictGet...` function for dictionaries of type `range`, if one of the arguments is constant and other is not. [#3751](#)
- Fixed error that caused messages `netlink: '...': attribute type 1 has an invalid length` to be printed in Linux kernel log, that was happening only on fresh enough versions of Linux kernel. [#3749](#)
- Fixed segfault in function `empty` for argument of `FixedString` type. [Daniel](#), [Dao Quang Minh](#)
- Fixed excessive memory allocation when using large value of `max_query_size` setting (a memory chunk of `max_query_size` bytes was preallocated at once). [#3720](#)

Build changes:

- Fixed build with LLVM/Clang libraries of version 7 from the OS packages (these libraries are used for runtime query compilation). [#3582](#)

ClickHouse release 18.14.17, 2018-11-30

Bug fixes:

- Fixed cases when the ODBC bridge process did not terminate with the main server process. [#3642](#)
- Fixed synchronous insertion into the `Distributed` table with a columns list that differs from the column list of the remote table. [#3673](#)
- Fixed a rare race condition that can lead to a crash when dropping a MergeTree table. [#3643](#)
- Fixed a query deadlock in case when query thread creation fails with the `Resource temporarily unavailable` error. [#3643](#)
- Fixed parsing of the `ENGINE` clause when the `CREATE AS table` syntax was used and the `ENGINE` clause was specified before the `AS table` (the error resulted in ignoring the specified engine). [#3692](#)

ClickHouse release 18.14.15, 2018-11-21

Bug fixes:

- The size of memory chunk was overestimated while deserializing the column of type `Array(String)` that leads to "Memory limit exceeded" errors. The issue appeared in version 18.12.13. [#3589](#)

ClickHouse release 18.14.14, 2018-11-20

Bug fixes:

- Fixed `ON CLUSTER` queries when cluster configured as secure (flag `<secure>`). [#3599](#)

Build changes:

- Fixed problems (llvm-7 from system, macos) [#3582](#)

ClickHouse release 18.14.13, 2018-11-08

Bug fixes:

- Fixed the `Block structure mismatch in MergingSorted stream` error. [#3162](#)
- Fixed `ON CLUSTER` queries in case when secure connections were turned on in the cluster config (the `<secure>` flag). [#3465](#)
- Fixed an error in queries that used `SAMPLE`, `PREWHERE` and alias columns. [#3543](#)
- Fixed a rare `unknown compression method` error when the `min_bytes_to_use_direct_io` setting was enabled. [3544](#)

Performance improvements:

- Fixed performance regression of queries with `GROUP BY` of columns of `UInt16` or `Date` type when executing on AMD EPYC processors. [Igor Lapko](#)
- Fixed performance regression of queries that process long strings. [#3530](#)

Build improvements:

- Improvements for simplifying the Arcadia build. [#3475](#), [#3535](#)

ClickHouse release 18.14.12, 2018-11-02

Bug fixes:

- Fixed a crash on joining two unnamed subqueries. [#3505](#)
- Fixed generating incorrect queries (with an empty `WHERE` clause) when querying external databases. [hotid](#)
- Fixed using an incorrect timeout value in ODBC dictionaries. [Marek Vavruša](#)

ClickHouse release 18.14.11, 2018-10-29

Bug fixes:

- Fixed the error `Block structure mismatch in UNION stream: different number of columns` in `LIMIT` queries. [#2156](#)
- Fixed errors when merging data in tables containing arrays inside Nested structures. [#3397](#)
- Fixed incorrect query results if the `merge_tree_uniform_read_distribution` setting is disabled (it is enabled by default). [#3429](#)
- Fixed an error on inserts to a Distributed table in Native format. [#3411](#)

ClickHouse release 18.14.10, 2018-10-23

- The `compile_expressions` setting (JIT compilation of expressions) is disabled by default. [#3410](#)
- The `enable_optimize_predicate_expression` setting is disabled by default.

ClickHouse release 18.14.9, 2018-10-16

New features:

- The `WITH CUBE` modifier for `GROUP BY` (the alternative syntax `GROUP BY CUBE(...)` is also available). [#3172](#)
- Added the `formatDateTime` function. [Alexandr Krasheninnikov](#)
- Added the `JDBC` table engine and `jdbc` table function (requires installing clickhouse-jdbc-bridge). [Alexandr Krasheninnikov](#)
- Added functions for working with the ISO week number: `toISOWeek`, `toISOYear`, `toStartOfISOYear`, and `toDayOfYear`. [#3146](#)
- Now you can use `Nullable` columns for `MySQL` and `ODBC` tables. [#3362](#)
- Nested data structures can be read as nested objects in `JSONEachRow` format. Added the `input_format_import_nested_json` setting. [Veloman Yunkan](#)
- Parallel processing is available for many `MATERIALIZED VIEWS` when inserting data. See the `parallel_view_processing` setting. [Marek Vavruša](#)
- Added the `SYSTEM FLUSH LOGS` query (forced log flushes to system tables such as `query_log`) [#3321](#)
- Now you can use pre-defined `database` and `table` macros when declaring `Replicated` tables. [#3251](#)
- Added the ability to read `Decimal` type values in engineering notation (indicating powers of ten). [#3153](#)

Experimental features:

- Optimization of the `GROUP BY` clause for `LowCardinality` data types. [#3138](#)
- Optimized calculation of expressions for `LowCardinality` data types. [#3200](#)

Improvements:

- Significantly reduced memory consumption for queries with `ORDER BY` and `LIMIT`. See the `max_bytes_before_remerge_sort` setting. [#3205](#)
- In the absence of `JOIN` (`LEFT`, `INNER`, ...), `INNER JOIN` is assumed. [#3147](#)
- Qualified asterisks work correctly in queries with `JOIN`. [Winter Zhang](#)
- The `ODBC` table engine correctly chooses the method for quoting identifiers in the SQL dialect of a remote database. [Alexandr Krasheninnikov](#)
- The `compile_expressions` setting (JIT compilation of expressions) is enabled by default.
- Fixed behavior for simultaneous `DROP DATABASE/TABLE IF EXISTS` and `CREATE DATABASE/TABLE IF NOT EXISTS`. Previously, a `CREATE DATABASE ... IF NOT EXISTS` query could return the error message "File ... already exists", and the `CREATE TABLE ... IF NOT EXISTS` and `DROP TABLE IF EXISTS` queries could return "Table ... is creating or attaching right now". [#3101](#)
- `LIKE` and `IN` expressions with a constant right half are passed to the remote server when querying from `MySQL` or `ODBC` tables. [#3182](#)
- Comparisons with constant expressions in a `WHERE` clause are passed to the remote server when querying from `MySQL` and `ODBC` tables. Previously, only comparisons with constants were passed. [#3182](#)
- Correct calculation of row width in the terminal for `Pretty` formats, including strings with hieroglyphs. [Amos Bird](#).
- `ON CLUSTER` can be specified for `ALTER UPDATE` queries.
- Improved performance for reading data in `JSONEachRow` format. [#3332](#)
- Added synonyms for the `LENGTH` and `CHARACTER_LENGTH` functions for compatibility. The `CONCAT` function is no longer case-sensitive. [#3306](#)
- Added the `TIMESTAMP` synonym for the `DateTime` type. [#3390](#)
- There is always space reserved for `query_id` in the server logs, even if the log line is not related to a query. This makes it easier to parse server text logs with third-party tools.
- Memory consumption by a query is logged when it exceeds the next level of an integer number of gigabytes. [#3205](#)

- Added compatibility mode for the case when the client library that uses the Native protocol sends fewer columns by mistake than the server expects for the INSERT query. This scenario was possible when using the clickhouse-cpp library. Previously, this scenario caused the server to crash. [#3171](#)
- In a user-defined WHERE expression in [clickhouse-copier](#), you can now use a [partition_key](#) alias (for additional filtering by source table partition). This is useful if the partitioning scheme changes during copying, but only changes slightly. [#3166](#)
- The workflow of the [Kafka](#) engine has been moved to a background thread pool in order to automatically reduce the speed of data reading at high loads. [Marek Vavruša](#).
- Support for reading [Tuple](#) and [Nested](#) values of structures like [struct](#) in the [Cap'n'Proto](#) format. [Marek Vavruša](#)
- The list of top-level domains for the [firstSignificantSubdomain](#) function now includes the domain [biz](#). [decaseal](#)
- In the configuration of external dictionaries, [null_value](#) is interpreted as the value of the default data type. [#3330](#)
- Support for the [intDiv](#) and [intDivOrZero](#) functions for [Decimal](#). [b48402e8](#)
- Support for the [Date](#), [DateTime](#), [UUID](#), and [Decimal](#) types as a key for the [sumMap](#) aggregate function. [#3281](#)
- Support for the [Decimal](#) data type in external dictionaries. [#3324](#)
- Support for the [Decimal](#) data type in [SummingMergeTree](#) tables. [#3348](#)
- Added specializations for [UUID](#) in [if](#). [#3366](#)
- Reduced the number of [open](#) and [close](#) system calls when reading from a [MergeTree](#) table. [#3283](#)
- A [TRUNCATE TABLE](#) query can be executed on any replica (the query is passed to the leader replica). [Kirill Shvakov](#)

Bug fixes:

- Fixed an issue with [Dictionary](#) tables for [range_hashed](#) dictionaries. This error occurred in version 18.12.17. [#1702](#)
- Fixed an error when loading [range_hashed](#) dictionaries (the message [Unsupported type Nullable \(...\)](#)). This error occurred in version 18.12.17. [#3362](#)
- Fixed errors in the [pointInPolygon](#) function due to the accumulation of inaccurate calculations for polygons with a large number of vertices located close to each other. [#3331](#) [#3341](#)
- If after merging data parts, the checksum for the resulting part differs from the result of the same merge in another replica, the result of the merge is deleted and the data part is downloaded from the other replica (this is the correct behavior). But after downloading the data part, it couldn't be added to the working set because of an error that the part already exists (because the data part was deleted with some delay after the merge). This led to cyclical attempts to download the same data. [#3194](#)
- Fixed incorrect calculation of total memory consumption by queries (because of incorrect calculation, the [max_memory_usage_for_all_queries](#) setting worked incorrectly and the [MemoryTracking](#) metric had an incorrect value). This error occurred in version 18.12.13. [Marek Vavruša](#)
- Fixed the functionality of [CREATE TABLE ... ON CLUSTER ... AS SELECT ...](#). This error occurred in version 18.12.13. [#3247](#)
- Fixed unnecessary preparation of data structures for [JOINS](#) on the server that initiates the query if the [JOIN](#) is only performed on remote servers. [#3340](#)
- Fixed bugs in the [Kafka](#) engine: deadlocks after exceptions when starting to read data, and locks upon completion [Marek Vavruša](#).
- For [Kafka](#) tables, the optional [schema](#) parameter was not passed (the schema of the [Cap'n'Proto](#) format). [Vojtech Splichal](#)
- If the ensemble of ZooKeeper servers has servers that accept the connection but then immediately close it instead of responding to the handshake, ClickHouse chooses to connect another server. Previously, this produced the error [Cannot read all data. Bytes read: 0. Bytes expected: 4.](#) and the server couldn't start. [8218cf3a](#)
- If the ensemble of ZooKeeper servers contains servers for which the DNS query returns an error, these servers are ignored. [17b8e209](#)
- Fixed type conversion between [Date](#) and [DateTime](#) when inserting data in the [VALUES](#) format (if [input_format_values_interpret_expressions](#) = 1). Previously, the conversion was performed between the numerical value of the number of days in Unix Epoch time and the Unix timestamp, which led to unexpected results. [#3229](#)
- Corrected type conversion between [Decimal](#) and integer numbers. [#3211](#)
- Fixed errors in the [enable_optimize_predicate_expression](#) setting. [Winter Zhang](#)

- Fixed a parsing error in CSV format with floating-point numbers if a non-default CSV separator is used, such as ; [#3155](#)
- Fixed the `arrayCumSumNonNegative` function (it does not accumulate negative values if the accumulator is less than zero). [Aleksey Studnev](#)
- Fixed how `Merge` tables work on top of `Distributed` tables when using `PREWHERE`. [#3165](#)
- Bug fixes in the `ALTER UPDATE` query.
- Fixed bugs in the `odbc` table function that appeared in version 18.12. [#3197](#)
- Fixed the operation of aggregate functions with `StateArray` combinators. [#3188](#)
- Fixed a crash when dividing a `Decimal` value by zero. [69dd6609](#)
- Fixed output of types for operations using `Decimal` and integer arguments. [#3224](#)
- Fixed the segfault during `GROUP BY` on `Decimal128`. [3359ba06](#)
- The `log_query_threads` setting (logging information about each thread of query execution) now takes effect only if the `log_queries` option (logging information about queries) is set to 1. Since the `log_query_threads` option is enabled by default, information about threads was previously logged even if query logging was disabled. [#3241](#)
- Fixed an error in the distributed operation of the quantiles aggregate function (the error message `Not found column quantile...`). [292a8855](#)
- Fixed the compatibility problem when working on a cluster of version 18.12.17 servers and older servers at the same time. For distributed queries with `GROUP BY` keys of both fixed and non-fixed length, if there was a large amount of data to aggregate, the returned data was not always fully aggregated (two different rows contained the same aggregation keys). [#3254](#)
- Fixed handling of substitutions in `clickhouse-performance-test`, if the query contains only part of the substitutions declared in the test. [#3263](#)
- Fixed an error when using `FINAL` with `PREWHERE`. [#3298](#)
- Fixed an error when using `PREWHERE` over columns that were added during `ALTER`. [#3298](#)
- Added a check for the absence of `arrayJoin` for `DEFAULT` and `MATERIALIZED` expressions. Previously, `arrayJoin` led to an error when inserting data. [#3337](#)
- Added a check for the absence of `arrayJoin` in a `PREWHERE` clause. Previously, this led to messages like `Size ... doesn't match` or `Unknown compression method` when executing queries. [#3357](#)
- Fixed segfault that could occur in rare cases after optimization that replaced `AND` chains from equality evaluations with the corresponding `IN` expression. [liuyimin-bytedance](#)
- Minor corrections to `clickhouse-benchmark`: previously, client information was not sent to the server; now the number of queries executed is calculated more accurately when shutting down and for limiting the number of iterations. [#3351](#) [#3352](#)

Backward incompatible changes:

- Removed the `allow_experimental_decimal_type` option. The `Decimal` data type is available for default use. [#3329](#)

ClickHouse release 18.12.17, 2018-09-16

New features:

- `invalidate_query` (the ability to specify a query to check whether an external dictionary needs to be updated) is implemented for the `clickhouse` source. [#3126](#)
- Added the ability to use `UInt*`, `Int*`, and `DateTime` data types (along with the `Date` type) as a `range_hashed` external dictionary key that defines the boundaries of ranges. Now `NULL` can be used to designate an open range. [Vasily Nemkov](#)
- The `Decimal` type now supports `var*` and `stddev*` aggregate functions. [#3129](#)
- The `Decimal` type now supports mathematical functions (`exp`, `sin` and so on.) [#3129](#)
- The `system.part_log` table now has the `partition_id` column. [#3089](#)

Bug fixes:

- `Merge` now works correctly on `Distributed` tables. [Winter Zhang](#)
- Fixed incompatibility (unnecessary dependency on the `glibc` version) that made it impossible to run ClickHouse on `Ubuntu Precise` and older versions. The incompatibility arose in version 18.12.13. [#3130](#)
- Fixed errors in the `enable_optimize_predicate_expression` setting. [Winter Zhang](#)

- Fixed a minor issue with backwards compatibility that appeared when working with a cluster of replicas on versions earlier than 18.12.13 and simultaneously creating a new replica of a table on a server with a newer version (shown in the message `Can not clone replica, because the ... updated to new ClickHouse version`, which is logical, but shouldn't happen). [#3122](#)

Backward incompatible changes:

- The `enable_optimize_predicate_expression` option is enabled by default (which is rather optimistic). If query analysis errors occur that are related to searching for the column names, set `enable_optimize_predicate_expression` to 0. [Winter Zhang](#)

ClickHouse release 18.12.14, 2018-09-13

New features:

- Added support for `ALTER UPDATE` queries. [#3035](#)
- Added the `allow_ddl` option, which restricts the user's access to DDL queries. [#3104](#)
- Added the `min_merge_bytes_to_use_direct_io` option for `MergeTree` engines, which allows you to set a threshold for the total size of the merge (when above the threshold, data part files will be handled using `O_DIRECT`). [#3117](#)
- The `system.merges` system table now contains the `partition_id` column. [#3099](#)

Improvements

- If a data part remains unchanged during mutation, it isn't downloaded by replicas. [#3103](#)
- Autocomplete is available for names of settings when working with `clickhouse-client`. [#3106](#)

Bug fixes:

- Added a check for the sizes of arrays that are elements of `Nested` type fields when inserting. [#3118](#)
- Fixed an error updating external dictionaries with the `ODBC` source and `hashed` storage. This error occurred in version 18.12.13.
- Fixed a crash when creating a temporary table from a query with an `IN` condition. [Winter Zhang](#)
- Fixed an error in aggregate functions for arrays that can have `NULL` elements. [Winter Zhang](#)

ClickHouse release 18.12.13, 2018-09-10

New features:

- Added the `DECIMAL(digits, scale)` data type (`Decimal32(scale)`, `Decimal64(scale)`, `Decimal128(scale)`). To enable it, use the setting `allow_experimental_decimal_type`. [#2846](#) [#2970](#) [#3008](#) [#3047](#)
- New `WITH ROLLUP` modifier for `GROUP BY` (alternative syntax: `GROUP BY ROLLUP(...)`). [#2948](#)
- In queries with `JOIN`, the star character expands to a list of columns in all tables, in compliance with the SQL standard. You can restore the old behavior by setting `asterisk_left_columns_only` to 1 on the user configuration level. [Winter Zhang](#)
- Added support for `JOIN` with table functions. [Winter Zhang](#)
- Autocomplete by pressing Tab in `clickhouse-client`. [Sergey Shcherbin](#)
- Ctrl+C in `clickhouse-client` clears a query that was entered. [#2877](#)
- Added the `join_default_strictness` setting (values: `"`, `'any'`, `'all'`). This allows you to not specify `ANY` or `ALL` for `JOIN`. [#2982](#)
- Each line of the server log related to query processing shows the query ID. [#2482](#)
- Now you can get query execution logs in `clickhouse-client` (use the `send_logs_level` setting). With distributed query processing, logs are cascaded from all the servers. [#2482](#)
- The `system.query_log` and `system.processes` (`SHOW PROCESSLIST`) tables now have information about all changed settings when you run a query (the nested structure of the `Settings` data). Added the `log_query_settings` setting. [#2482](#)
- The `system.query_log` and `system.processes` tables now show information about the number of threads that are participating in query execution (see the `thread_numbers` column). [#2482](#)
- Added `ProfileEvents` counters that measure the time spent on reading and writing over the network and reading and writing to disk, the number of network errors, and the time spent waiting when network bandwidth is limited. [#2482](#)

- Added `ProfileEvents` counters that contain the system metrics from rusage (you can use them to get information about CPU usage in userspace and the kernel, page faults, and context switches), as well as taskstats metrics (use these to obtain information about I/O wait time, CPU wait time, and the amount of data read and recorded, both with and without page cache). [#2482](#)
- The `ProfileEvents` counters are applied globally and for each query, as well as for each query execution thread, which allows you to profile resource consumption by query in detail. [#2482](#)
- Added the `system.query_thread_log` table, which contains information about each query execution thread. Added the `log_query_threads` setting. [#2482](#)
- The `system.metrics` and `system.events` tables now have built-in documentation. [#3016](#)
- Added the `arrayEnumerateDense` function. [Amos Bird](#)
- Added the `arrayCumSumNonNegative` and `arrayDifference` functions. [Aleksey Studnev](#)
- Added the `retention` aggregate function. [Sundy Li](#)
- Now you can add (merge) states of aggregate functions by using the plus operator, and multiply the states of aggregate functions by a nonnegative constant. [#3062](#) [#3034](#)
- Tables in the MergeTree family now have the virtual column `_partition_id`. [#3089](#)

Experimental features:

- Added the `LowCardinality(T)` data type. This data type automatically creates a local dictionary of values and allows data processing without unpacking the dictionary. [#2830](#)
- Added a cache of JIT-compiled functions and a counter for the number of uses before compiling. To JIT compile expressions, enable the `compile_expressions` setting. [#2990](#) [#3077](#)

Improvements:

- Fixed the problem with unlimited accumulation of the replication log when there are abandoned replicas. Added an effective recovery mode for replicas with a long lag.
- Improved performance of `GROUP BY` with multiple aggregation fields when one of them is string and the others are fixed length.
- Improved performance when using `PREWHERE` and with implicit transfer of expressions in `PREWHERE`.
- Improved parsing performance for text formats (`CSV`, `TSV`). [Amos Bird](#) [#2980](#)
- Improved performance of reading strings and arrays in binary formats. [Amos Bird](#)
- Increased performance and reduced memory consumption for queries to `system.tables` and `system.columns` when there is a very large number of tables on a single server. [#2953](#)
- Fixed a performance problem in the case of a large stream of queries that result in an error (the `_dl_addr` function is visible in `perf top`, but the server isn't using much CPU). [#2938](#)
- Conditions are cast into the View (when `enable_optimize_predicate_expression` is enabled). [Winter Zhang](#)
- Improvements to the functionality for the `UUID` data type. [#3074](#) [#2985](#)
- The `UUID` data type is supported in The-Alchemist dictionaries. [#2822](#)
- The `visitParamExtractRaw` function works correctly with nested structures. [Winter Zhang](#)
- When the `input_format_skip_unknown_fields` setting is enabled, object fields in `JSONEachRow` format are skipped correctly. [BlahGeek](#)
- For a `CASE` expression with conditions, you can now omit `ELSE`, which is equivalent to `ELSE NULL`. [#2920](#)
- The operation timeout can now be configured when working with ZooKeeper. [urykhy](#)
- You can specify an offset for `LIMIT n, m` as `LIMIT n OFFSET m`. [#2840](#)
- You can use the `SELECT TOP n` syntax as an alternative for `LIMIT`. [#2840](#)
- Increased the size of the queue to write to system tables, so the `SystemLog parameter queue is full` error doesn't happen as often.
- The `windowFunnel` aggregate function now supports events that meet multiple conditions. [Amos Bird](#)
- Duplicate columns can be used in a `USING` clause for `JOIN`. [#3006](#)
- `Pretty` formats now have a limit on column alignment by width. Use the `output_format_pretty_max_column_pad_width` setting. If a value is wider, it will still be displayed in its entirety, but the other cells in the table will not be too wide. [#3003](#)
- The `odbc` table function now allows you to specify the database/schema name. [Amos Bird](#)
- Added the ability to use a username specified in the `clickhouse-client` config file. [Vladimir Kozbin](#)
- The `ZooKeeperExceptions` counter has been split into three counters: `ZooKeeperUserExceptions`, `ZooKeeperHardwareExceptions`, and `ZooKeeperOtherExceptions`.

- `ALTER DELETE` queries work for materialized views.
- Added randomization when running the cleanup thread periodically for `ReplicatedMergeTree` tables in order to avoid periodic load spikes when there are a very large number of `ReplicatedMergeTree` tables.
- Support for `ATTACH TABLE ... ON CLUSTER` queries. [#3025](#)

Bug fixes:

- Fixed an issue with `Dictionary` tables (throws the `Size of offsets doesn't match size of column` or `Unknown compression method` exception). This bug appeared in version 18.10.3. [#2913](#)
- Fixed a bug when merging `CollapsingMergeTree` tables if one of the data parts is empty (these parts are formed during merge or `ALTER DELETE` if all data was deleted), and the `vertical` algorithm was used for the merge. [#3049](#)
- Fixed a race condition during `DROP` or `TRUNCATE` for `Memory` tables with a simultaneous `SELECT`, which could lead to server crashes. This bug appeared in version 1.1.54388. [#3038](#)
- Fixed the possibility of data loss when inserting in `Replicated` tables if the `Session is expired` error is returned (data loss can be detected by the `ReplicatedDataLoss` metric). This error occurred in version 1.1.54378. [#2939](#) [#2949](#) [#2964](#)
- Fixed a segfault during `JOIN ... ON`. [#3000](#)
- Fixed the error searching column names when the `WHERE` expression consists entirely of a qualified column name, such as `WHERE table.column`. [#2994](#)
- Fixed the "Not found column" error that occurred when executing distributed queries if a single column consisting of an `IN` expression with a subquery is requested from a remote server. [#3087](#)
- Fixed the `Block structure mismatch in UNION stream: different number of columns` error that occurred for distributed queries if one of the shards is local and the other is not, and optimization of the move to `PREWHERE` is triggered. [#2226](#) [#3037](#) [#3055](#) [#3065](#) [#3073](#) [#3090](#) [#3093](#)
- Fixed the `pointInPolygon` function for certain cases of non-convex polygons. [#2910](#)
- Fixed the incorrect result when comparing `nan` with integers. [#3024](#)
- Fixed an error in the `zlib-ng` library that could lead to segfault in rare cases. [#2854](#)
- Fixed a memory leak when inserting into a table with `AggregateFunction` columns, if the state of the aggregate function is not simple (allocates memory separately), and if a single insertion request results in multiple small blocks. [#3084](#)
- Fixed a race condition when creating and deleting the same `Buffer` or `MergeTree` table simultaneously.
- Fixed the possibility of a segfault when comparing tuples made up of certain non-trivial types, such as tuples. [#2989](#)
- Fixed the possibility of a segfault when running certain `ON CLUSTER` queries. [Winter Zhang](#)
- Fixed an error in the `arrayDistinct` function for `Nullable` array elements. [#2845](#) [#2937](#)
- The `enable_optimize_predicate_expression` option now correctly supports cases with `SELECT *`. [Winter Zhang](#)
- Fixed the segfault when re-initializing the ZooKeeper session. [#2917](#)
- Fixed potential blocking when working with ZooKeeper.
- Fixed incorrect code for adding nested data structures in a `SummingMergeTree`.
- When allocating memory for states of aggregate functions, alignment is correctly taken into account, which makes it possible to use operations that require alignment when implementing states of aggregate functions. [chenxing-xc](#)

Security fix:

- Safe use of ODBC data sources. Interaction with ODBC drivers uses a separate `clickhouse-odbc-bridge` process. Errors in third-party ODBC drivers no longer cause problems with server stability or vulnerabilities. [#2828](#) [#2879](#) [#2886](#) [#2893](#) [#2921](#)
- Fixed incorrect validation of the file path in the `catBoostPool` table function. [#2894](#)
- The contents of system tables (`tables`, `databases`, `parts`, `columns`, `parts_columns`, `merges`, `mutations`, `replicas`, and `replication_queue`) are filtered according to the user's configured access to databases (`allow_databases`). [Winter Zhang](#)

Backward incompatible changes:

- In queries with JOIN, the star character expands to a list of columns in all tables, in compliance with the SQL standard. You can restore the old behavior by setting `asterisk_left_columns_only` to 1 on the user configuration level.

Build changes:

- Most integration tests can now be run by commit.
- Code style checks can also be run by commit.
- The `memcpy` implementation is chosen correctly when building on CentOS7/Fedora. [Etienne Champetier](#)
- When using clang to build, some warnings from `-Weverything` have been added, in addition to the regular `-Wall-Wextra-Werror`. [#2957](#)
- Debugging the build uses the `jemalloc` debug option.
- The interface of the library for interacting with ZooKeeper is declared abstract. [#2950](#)

ClickHouse release 18.10.3, 2018-08-13

New features:

- HTTPS can be used for replication. [#2760](#)
- Added the functions `murmurHash2_64`, `murmurHash3_32`, `murmurHash3_64`, and `murmurHash3_128` in addition to the existing `murmurHash2_32`. [#2791](#)
- Support for Nullable types in the ClickHouse ODBC driver (`ODBCDriver2` output format). [#2834](#)
- Support for `UUID` in the key columns.

Improvements:

- Clusters can be removed without restarting the server when they are deleted from the config files. [#2777](#)
- External dictionaries can be removed without restarting the server when they are removed from config files. [#2779](#)
- Added `SETTINGS` support for the `Kafka` table engine. [Alexander Marshalov](#)
- Improvements for the `UUID` data type (not yet complete). [#2618](#)
- Support for empty parts after merges in the `SummingMergeTree`, `CollapsingMergeTree` and `VersionedCollapsingMergeTree` engines. [#2815](#)
- Old records of completed mutations are deleted (`ALTER DELETE`). [#2784](#)
- Added the `system.merge_tree_settings` table. [Kirill Shvakov](#)
- The `system.tables` table now has dependency columns: `dependencies_database` and `dependencies_table`. [Winter Zhang](#)
- Added the `max_partition_size_to_drop` config option. [#2782](#)
- Added the `output_format_json_escape_forward_slashes` option. [Alexander Bocharov](#)
- Added the `max_fetch_partition_retries_count` setting. [#2831](#)
- Added the `prefer_localhost_replica` setting for disabling the preference for a local replica and going to a local replica without inter-process interaction. [#2832](#)
- The `quantileExact` aggregate function returns `nan` in the case of aggregation on an empty `Float32` or `Float64` set. [Sundy Li](#)

Bug fixes:

- Removed unnecessary escaping of the connection string parameters for ODBC, which made it impossible to establish a connection. This error occurred in version 18.6.0.
- Fixed the logic for processing `REPLACE PARTITION` commands in the replication queue. If there are two `REPLACE` commands for the same partition, the incorrect logic could cause one of them to remain in the replication queue and not be executed. [#2814](#)
- Fixed a merge bug when all data parts were empty (parts that were formed from a merge or from `ALTER DELETE` if all data was deleted). This bug appeared in version 18.1.0. [#2930](#)
- Fixed an error for concurrent `Set` or `Join`. [Amos Bird](#)
- Fixed the `Block structure mismatch in UNION stream: different number of columns` error that occurred for `UNION ALL` queries inside a sub-query if one of the `SELECT` queries contains duplicate column names. [Winter Zhang](#)
- Fixed a memory leak if an exception occurred when connecting to a MySQL server.
- Fixed incorrect clickhouse-client response code in case of a query error.

- Fixed incorrect behavior of materialized views containing DISTINCT. [#2795](#)

Backward incompatible changes

- Removed support for CHECK TABLE queries for Distributed tables.

Build changes:

- The allocator has been replaced: [jemalloc](#) is now used instead of [tcmalloc](#). In some scenarios, this increases speed up to 20%. However, there are queries that have slowed by up to 20%. Memory consumption has been reduced by approximately 10% in some scenarios, with improved stability. With highly competitive loads, CPU usage in userspace and in system shows just a slight increase. [#2773](#)
- Use of libressl from a submodule. [#1983](#) [#2807](#)
- Use of unixodbc from a submodule. [#2789](#)
- Use of mariadb-connector-c from a submodule. [#2785](#)
- Added functional test files to the repository that depend on the availability of test data (for the time being, without the test data itself).

ClickHouse release 18.6.0, 2018-08-02

New features:

- Added support for ON expressions for the JOIN ON syntax:
`JOIN ON Expr([table.]column ...) = Expr([table.]column, ...) [AND Expr([table.]column, ...) = Expr([table.]column, ...) ...]`
 The expression must be a chain of equalities joined by the AND operator. Each side of the equality can be an arbitrary expression over the columns of one of the tables. The use of fully qualified column names is supported ([table.name](#), [database.table.name](#), [table_alias.name](#), [subquery_alias.name](#)) for the right table. [#2742](#)
- HTTPS can be enabled for replication. [#2760](#)

Improvements:

- The server passes the patch component of its version to the client. Data about the patch version component is in [system.processes](#) and [query_log](#). [#2646](#)

ClickHouse release 18.5.1, 2018-07-31

New features:

- Added the hash function [murmurHash2_32](#) [#2756](#).

Improvements:

- Now you can use the [from_env](#) [#2741](#) attribute to set values in config files from environment variables.
- Added case-insensitive versions of the [coalesce](#), [ifNull](#), and [nullIf](#) functions [#2752](#).

Bug fixes:

- Fixed a possible bug when starting a replica [#2759](#).

ClickHouse release 18.4.0, 2018-07-28

New features:

- Added system tables: [formats](#), [data_type_families](#), [aggregate_function_combinators](#), [table_functions](#), [table_engines](#), [collations](#) [#2721](#).
- Added the ability to use a table function instead of a table as an argument of a [remote](#) or [cluster table function](#) [#2708](#).
- Support for [HTTP Basic](#) authentication in the replication protocol [#2727](#).
- The [has](#) function now allows searching for a numeric value in an array of [Enum](#) values [Maxim Khrisanfov](#).
- Support for adding arbitrary message separators when reading from [Kafka](#) [Amos Bird](#).

Improvements:

- The `ALTER TABLE t DELETE WHERE` query does not rewrite data parts that were not affected by the WHERE condition (#2694).
- The `use_minimalistic_checksums_in_zookeeper` option for `ReplicatedMergeTree` tables is enabled by default. This setting was added in version 1.1.54378, 2018-04-16. Versions that are older than 1.1.54378 can no longer be installed.
- Support for running `KILL` and `OPTIMIZE` queries that specify `ON CLUSTER` Winter Zhang.

Bug fixes:

- Fixed the error `Column ... is not under an aggregate function and not in GROUP BY` for aggregation with an IN expression. This bug appeared in version 18.1.0. (bddd780b)
- Fixed a bug in the `windowFunnel` aggregate function Winter Zhang.
- Fixed a bug in the `anyHeavy` aggregate function (a2101df2)
- Fixed server crash when using the `countArray()` aggregate function.

Backward incompatible changes:

- Parameters for `Kafka` engine was changed from `Kafka(kafka_broker_list, kafka_topic_list, kafka_group_name, kafka_format[, kafka_schema, kafka_num_consumers])` to `Kafka(kafka_broker_list, kafka_topic_list, kafka_group_name, kafka_format[, kafka_row_delimiter, kafka_schema, kafka_num_consumers])`. If your tables use `kafka_schema` or `kafka_num_consumers` parameters, you have to manually edit the metadata files `path/metadata/database/table.sql` and add `kafka_row_delimiter` parameter with `"` value.

ClickHouse release 18.1.0, 2018-07-23

New features:

- Support for the `ALTER TABLE t DELETE WHERE` query for non-replicated MergeTree tables (#2634).
- Support for arbitrary types for the `uniq*` family of aggregate functions (#2010).
- Support for arbitrary types in comparison operators (#2026).
- The `users.xml` file allows setting a subnet mask in the format `10.0.0.1/255.255.255.0`. This is necessary for using masks for IPv6 networks with zeros in the middle (#2637).
- Added the `arrayDistinct` function (#2670).
- The SummingMergeTree engine can now work with AggregateFunction type columns (Constantin S. Pan).

Improvements:

- Changed the numbering scheme for release versions. Now the first part contains the year of release (A.D., Moscow timezone, minus 2000), the second part contains the number for major changes (increases for most releases), and the third part is the patch version. Releases are still backwards compatible, unless otherwise stated in the changelog.
- Faster conversions of floating-point numbers to a string (Amos Bird).
- If some rows were skipped during an insert due to parsing errors (this is possible with the `input_allow_errors_num` and `input_allow_errors_ratio` settings enabled), the number of skipped rows is now written to the server log (Leonardo Cecchi).

Bug fixes:

- Fixed the TRUNCATE command for temporary tables (Amos Bird).
- Fixed a rare deadlock in the ZooKeeper client library that occurred when there was a network error while reading the response (c315200).
- Fixed an error during a CAST to Nullable types (#1322).
- Fixed the incorrect result of the `maxIntersection()` function when the boundaries of intervals coincided (Michael Furmur).
- Fixed incorrect transformation of the OR expression chain in a function argument (chenxing-xc).
- Fixed performance degradation for queries containing `IN (subquery)` expressions inside another subquery (#2571).
- Fixed incompatibility between servers with different versions in distributed queries that use a `CAST` function that isn't in uppercase letters (fe8c4d6).
- Added missing quoting of identifiers for queries to an external DBMS (#2635).

Backward incompatible changes:

- Converting a string containing the number zero to DateTime does not work. Example: `SELECT toDateTime('0')`. This is also the reason that `DateTime DEFAULT '0'` does not work in tables, as well as `<null_value>0</null_value>` in dictionaries. Solution: replace `0` with `0000-00-00 00:00:00`.

ClickHouse release 1.1.54394, 2018-07-12

New features:

- Added the `histogram` aggregate function (Mikhail Surin).
- Now `OPTIMIZE TABLE ... FINAL` can be used without specifying partitions for `ReplicatedMergeTree` (Amos Bird).

Bug fixes:

- Fixed a problem with a very small timeout for sockets (one second) for reading and writing when sending and downloading replicated data, which made it impossible to download larger parts if there is a load on the network or disk (it resulted in cyclical attempts to download parts). This error occurred in version 1.1.54388.
- Fixed issues when using chroot in ZooKeeper if you inserted duplicate data blocks in the table.
- The `has` function now works correctly for an array with Nullable elements (#2115).
- The `system.tables` table now works correctly when used in distributed queries. The `metadata_modification_time` and `engine_full` columns are now non-virtual. Fixed an error that occurred if only these columns were queried from the table.
- Fixed how an empty `TinyLog` table works after inserting an empty data block (#2563).
- The `system.zookeeper` table works if the value of the node in ZooKeeper is NULL.

ClickHouse release 1.1.54390, 2018-07-06

New features:

- Queries can be sent in `multipart/form-data` format (in the `query` field), which is useful if external data is also sent for query processing (Olga Hvoslikova).
- Added the ability to enable or disable processing single or double quotes when reading data in CSV format. You can configure this in the `format_csv_allow_single_quotes` and `format_csv_allow_double_quotes` settings (Amos Bird).
- Now `OPTIMIZE TABLE ... FINAL` can be used without specifying the partition for non-replicated variants of `MergeTree` (Amos Bird).

Improvements:

- Improved performance, reduced memory consumption, and correct memory consumption tracking with use of the `IN` operator when a table index could be used (#2584).
- Removed redundant checking of checksums when adding a data part. This is important when there are a large number of replicas, because in these cases the total number of checks was equal to N^2 .
- Added support for `Array(Tuple(...))` arguments for the `arrayEnumerateUniq` function (#2573).
- Added `Nullable` support for the `runningDifference` function (#2594).
- Improved query analysis performance when there is a very large number of expressions (#2572).
- Faster selection of data parts for merging in `ReplicatedMergeTree` tables. Faster recovery of the ZooKeeper session (#2597).
- The `format_version.txt` file for `MergeTree` tables is re-created if it is missing, which makes sense if ClickHouse is launched after copying the directory structure without files (Ciprian Hacman).

Bug fixes:

- Fixed a bug when working with ZooKeeper that could make it impossible to recover the session and readonly states of tables before restarting the server.
- Fixed a bug when working with ZooKeeper that could result in old nodes not being deleted if the session is interrupted.
- Fixed an error in the `quantileTDigest` function for Float arguments (this bug was introduced in version 1.1.54388) (Mikhail Surin).

- Fixed a bug in the index for MergeTree tables if the primary key column is located inside the function for converting types between signed and unsigned integers of the same size ([#2603](#)).
- Fixed segfault if `macros` are used but they aren't in the config file ([#2570](#)).
- Fixed switching to the default database when reconnecting the client ([#2583](#)).
- Fixed a bug that occurred when the `use_index_for_in_with_subqueries` setting was disabled.

Security fix:

- Sending files is no longer possible when connected to MySQL (`LOAD DATA LOCAL INFILE`).

ClickHouse release 1.1.54388, 2018-06-28

New features:

- Support for the `ALTER TABLE t DELETE WHERE` query for replicated tables. Added the `system.mutations` table to track progress of this type of queries.
- Support for the `ALTER TABLE t [REPLACE|ATTACH] PARTITION` query for *MergeTree tables.
- Support for the `TRUNCATE TABLE` query ([Winter Zhang](#)).
- Several new `SYSTEM` queries for replicated tables (`RESTART REPLICAS`, `SYNC REPLICA`, `[STOP|START] [MERGES|FETCHES|SENDS REPLICATED|REPLICATION QUEUES]`).
- Added the ability to write to a table with the MySQL engine and the corresponding table function ([sundy-li](#)).
- Added the `url()` table function and the `URL` table engine ([Alexander Sapin](#)).
- Added the `windowFunnel` aggregate function ([sundy-li](#)).
- New `startsWith` and `endsWith` functions for strings ([Vadim Plakhtinsky](#)).
- The `numbers()` table function now allows you to specify the offset ([Winter Zhang](#)).
- The password to `clickhouse-client` can be entered interactively.
- Server logs can now be sent to syslog ([Alexander Krashennikov](#)).
- Support for logging in dictionaries with a shared library source ([Alexander Sapin](#)).
- Support for custom CSV delimiters ([Ivan Zhukov](#)).
- Added the `date_time_input_format` setting. If you switch this setting to `'best_effort'`, DateTime values will be read in a wide range of formats.
- Added the `clickhouse-obfuscator` utility for data obfuscation. Usage example: publishing data used in performance tests.

Experimental features:

- Added the ability to calculate `and` arguments only where they are needed ([Anastasia Tsarkova](#)).
- JIT compilation to native code is now available for some expressions (`pyos`).

Bug fixes:

- Duplicates no longer appear for a query with `DISTINCT` and `ORDER BY`.
- Queries with `ARRAY JOIN` and `arrayFilter` no longer return an incorrect result.
- Fixed an error when reading an array column from a Nested structure ([#2066](#)).
- Fixed an error when analyzing queries with a HAVING clause like `HAVING tuple IN (...)`.
- Fixed an error when analyzing queries with recursive aliases.
- Fixed an error when reading from ReplacingMergeTree with a condition in PREWHERE that filters all rows ([#2525](#)).
- User profile settings were not applied when using sessions in the HTTP interface.
- Fixed how settings are applied from the command line parameters in `clickhouse-local`.
- The ZooKeeper client library now uses the session timeout received from the server.
- Fixed a bug in the ZooKeeper client library when the client waited for the server response longer than the timeout.
- Fixed pruning of parts for queries with conditions on partition key columns ([#2342](#)).
- Merges are now possible after `CLEAR COLUMN IN PARTITION` ([#2315](#)).
- Type mapping in the ODBC table function has been fixed ([sundy-li](#)).
- Type comparisons have been fixed for `DateTime` with and without the time zone ([Alexander Bocharov](#)).
- Fixed syntactic parsing and formatting of the `CAST` operator.
- Fixed insertion into a materialized view for the Distributed table engine ([Babacar Diassé](#)).

- Fixed a race condition when writing data from the [Kafka](#) engine to materialized views ([Yangkuan Liu](#)).
- Fixed SSRF in the `remote()` table function.
- Fixed exit behavior of [clickhouse-client](#) in multiline mode ([#2510](#)).

Improvements:

- Background tasks in replicated tables are now performed in a thread pool instead of in separate threads ([Silviu Caragea](#)).
- Improved LZ4 compression performance.
- Faster analysis for queries with a large number of JOINS and sub-queries.
- The DNS cache is now updated automatically when there are too many network errors.
- Table inserts no longer occur if the insert into one of the materialized views is not possible because it has too many parts.
- Corrected the discrepancy in the event counters [Query](#), [SelectQuery](#), and [InsertQuery](#).
- Expressions like `tuple IN (SELECT tuple)` are allowed if the tuple types match.
- A server with replicated tables can start even if you haven't configured ZooKeeper.
- When calculating the number of available CPU cores, limits on cgroups are now taken into account ([Atri Sharma](#)).
- Added `chown` for config directories in the `systemd` config file ([Mikhail Shiryayev](#)).

Build changes:

- The `gcc8` compiler can be used for builds.
- Added the ability to build `llvm` from submodule.
- The version of the `librdkafka` library has been updated to `v0.11.4`.
- Added the ability to use the system `libcuuid` library. The library version has been updated to `0.4.0`.
- Fixed the build using the `vectorclass` library ([Babacar Diassé](#)).
- `Cmake` now generates files for `ninja` by default (like when using `-G Ninja`).
- Added the ability to use the `libtinfo` library instead of `libtermcap` ([Georgy Kondratiev](#)).
- Fixed a header file conflict in Fedora Rawhide ([#2520](#)).

Backward incompatible changes:

- Removed escaping in [Vertical](#) and [Pretty*](#) formats and deleted the [VerticalRaw](#) format.
- If servers with version `1.1.54388` (or newer) and servers with an older version are used simultaneously in a distributed query and the query has the `cast(x, 'Type')` expression without the `AS` keyword and doesn't have the word `cast` in uppercase, an exception will be thrown with a message like `Not found column cast(0, 'UInt8') in block`.
Solution: Update the server on the entire cluster.

ClickHouse release 1.1.54385, 2018-06-01

Bug fixes:

- Fixed an error that in some cases caused ZooKeeper operations to block.

ClickHouse release 1.1.54383, 2018-05-22

Bug fixes:

- Fixed a slowdown of replication queue if a table has many replicas.

ClickHouse release 1.1.54381, 2018-05-14

Bug fixes:

- Fixed a nodes leak in ZooKeeper when ClickHouse loses connection to ZooKeeper server.

ClickHouse release 1.1.54380, 2018-04-21

New features:

- Added the table function `file(path, format, structure)`. An example reading bytes from `/dev/urandom`: `In -s /dev/urandom /var/lib/clickhouse/user_files/random``clickhouse-client -q "SELECT * FROM file('random', 'RowBinary', 'd UInt8') LIMIT 10"`.

Improvements:

- Subqueries can be wrapped in `()` brackets to enhance query readability. For example: `(SELECT 1) UNION ALL (SELECT 1)`.
- Simple `SELECT` queries from the `system.processes` table are not included in the `max_concurrent_queries` limit.

Bug fixes:

- Fixed incorrect behavior of the `IN` operator when select from `MATERIALIZED VIEW`.
- Fixed incorrect filtering by partition index in expressions like `partition_key_column IN (...)`.
- Fixed inability to execute `OPTIMIZE` query on non-leader replica if `RENAME` was performed on the table.
- Fixed the authorization error when executing `OPTIMIZE` or `ALTER` queries on a non-leader replica.
- Fixed freezing of `KILL QUERY`.
- Fixed an error in ZooKeeper client library which led to loss of watches, freezing of distributed DDL queue, and slowdowns in the replication queue if a non-empty `chroot` prefix is used in the ZooKeeper configuration.

Backward incompatible changes:

- Removed support for expressions like `(a, b) IN (SELECT (a, b))` (you can use the equivalent expression `(a, b) IN (SELECT a, b)`). In previous releases, these expressions led to undetermined `WHERE` filtering or caused errors.

ClickHouse release 1.1.54378, 2018-04-16

New features:

- Logging level can be changed without restarting the server.
- Added the `SHOW CREATE DATABASE` query.
- The `query_id` can be passed to `clickhouse-client` (elBroom).
- New setting: `max_network_bandwidth_for_all_users`.
- Added support for `ALTER TABLE ... PARTITION ...` for `MATERIALIZED VIEW`.
- Added information about the size of data parts in uncompressed form in the system table.
- Server-to-server encryption support for distributed tables (`<secure>1</secure>` in the replica config in `<remote_servers>`).
- Configuration of the table level for the `ReplicatedMergeTree` family in order to minimize the amount of data stored in Zookeeper: `: use_minimalistic_checksums_in_zookeeper = 1`
- Configuration of the `clickhouse-client` prompt. By default, server names are now output to the prompt. The server's display name can be changed. It's also sent in the `X-ClickHouse-Display-Name` HTTP header (Kirill Shvakov).
- Multiple comma-separated `topics` can be specified for the `Kafka` engine (Tobias Adamson)
- When a query is stopped by `KILL QUERY` or `replace_running_query`, the client receives the `Query was cancelled` exception instead of an incomplete result.

Improvements:

- `ALTER TABLE ... DROP/DETACH PARTITION` queries are run at the front of the replication queue.
- `SELECT ... FINAL` and `OPTIMIZE ... FINAL` can be used even when the table has a single data part.
- A `query_log` table is recreated on the fly if it was deleted manually (Kirill Shvakov).
- The `lengthUTF8` function runs faster (zhang2014).
- Improved performance of synchronous inserts in `Distributed` tables (`insert_distributed_sync = 1`) when there is a very large number of shards.
- The server accepts the `send_timeout` and `receive_timeout` settings from the client and applies them when connecting to the client (they are applied in reverse order: the server socket's `send_timeout` is set to the `receive_timeout` value received from the client, and vice versa).
- More robust crash recovery for asynchronous insertion into `Distributed` tables.
- The return type of the `countEqual` function changed from `UInt32` to `UInt64` (谢磊).

Bug fixes:

- Fixed an error with `IN` when the left side of the expression is `Nullable`.
- Correct results are now returned when using tuples with `IN` when some of the tuple components are in the table index.
- The `max_execution_time` limit now works correctly with distributed queries.
- Fixed errors when calculating the size of composite columns in the `system.columns` table.
- Fixed an error when creating a temporary table `CREATE TEMPORARY TABLE IF NOT EXISTS`.
- Fixed errors in `StorageKafka` ([##2075](#))
- Fixed server crashes from invalid arguments of certain aggregate functions.
- Fixed the error that prevented the `DETACH DATABASE` query from stopping background tasks for `ReplicatedMergeTree` tables.
- `Too many parts` state is less likely to happen when inserting into aggregated materialized views ([##2084](#)).
- Corrected recursive handling of substitutions in the config if a substitution must be followed by another substitution on the same level.
- Corrected the syntax in the metadata file when creating a `VIEW` that uses a query with `UNION ALL`.
- `SummingMergeTree` now works correctly for summation of nested data structures with a composite key.
- Fixed the possibility of a race condition when choosing the leader for `ReplicatedMergeTree` tables.

Build changes:

- The build supports `ninja` instead of `make` and uses `ninja` by default for building releases.
- Renamed packages: `clickhouse-server-base` in `clickhouse-common-static`; `clickhouse-server-common` in `clickhouse-server`; `clickhouse-common-dbg` in `clickhouse-common-static-dbg`. To install, use `clickhouse-server` `clickhouse-client`. Packages with the old names will still load in the repositories for backward compatibility.

Backward incompatible changes:

- Removed the special interpretation of an `IN` expression if an array is specified on the left side. Previously, the expression `arr IN (set)` was interpreted as "at least one `arr` element belongs to the `set`". To get the same behavior in the new version, write `arrayExists(x -> x IN (set), arr)`.
- Disabled the incorrect use of the socket option `SO_REUSEPORT`, which was incorrectly enabled by default in the Poco library. Note that on Linux there is no longer any reason to simultaneously specify the addresses `::` and `0.0.0.0` for listen – use just `::`, which allows listening to the connection both over IPv4 and IPv6 (with the default kernel config settings). You can also revert to the behavior from previous versions by specifying `<listen_reuse_port>1</listen_reuse_port>` in the config.

ClickHouse release 1.1.54370, 2018-03-16

New features:

- Added the `system.macros` table and auto updating of macros when the config file is changed.
- Added the `SYSTEM RELOAD CONFIG` query.
- Added the `maxIntersections(left_col, right_col)` aggregate function, which returns the maximum number of simultaneously intersecting intervals `[left; right]`. The `maxIntersectionsPosition(left, right)` function returns the beginning of the "maximum" interval. ([Michael Furmur](#)).

Improvements:

- When inserting data in a `Replicated` table, fewer requests are made to `ZooKeeper` (and most of the user-level errors have disappeared from the `ZooKeeper` log).
- Added the ability to create aliases for data sets. Example: `WITH (1, 2, 3) AS set SELECT number IN set FROM system.numbers LIMIT 10`.

Bug fixes:

- Fixed the `Illegal PREWHERE` error when reading from Merge tables for `Distributed` tables.
- Added fixes that allow you to start clickhouse-server in IPv4-only Docker containers.
- Fixed a race condition when reading from system `system.parts_columns` tables.
- Removed double buffering during a synchronous insert to a `Distributed` table, which could have caused the connection to timeout.

- Fixed a bug that caused excessively long waits for an unavailable replica before beginning a `SELECT` query.
- Fixed incorrect dates in the `system.parts` table.
- Fixed a bug that made it impossible to insert data in a `Replicated` table if `chroot` was non-empty in the configuration of the `ZooKeeper` cluster.
- Fixed the vertical merging algorithm for an empty `ORDER BY` table.
- Restored the ability to use dictionaries in queries to remote tables, even if these dictionaries are not present on the requestor server. This functionality was lost in release 1.1.54362.
- Restored the behavior for queries like `SELECT * FROM remote('server2', default.table) WHERE col IN (SELECT col2 FROM default.table)` when the right side of the `IN` should use a remote `default.table` instead of a local one. This behavior was broken in version 1.1.54358.
- Removed extraneous error-level logging of `Not found column ... in block`.

Clickhouse Release 1.1.54362, 2018-03-11

New features:

- Aggregation without `GROUP BY` for an empty set (such as `SELECT count(*) FROM table WHERE 0`) now returns a result with one row with null values for aggregate functions, in compliance with the SQL standard. To restore the old behavior (return an empty result), set `empty_result_for_aggregation_by_empty_set` to 1.
- Added type conversion for `UNION ALL`. Different alias names are allowed in `SELECT` positions in `UNION ALL`, in compliance with the SQL standard.
- Arbitrary expressions are supported in `LIMIT BY` clauses. Previously, it was only possible to use columns resulting from `SELECT`.
- An index of `MergeTree` tables is used when `IN` is applied to a tuple of expressions from the columns of the primary key. Example: `WHERE (UserID, EventDate) IN ((123, '2000-01-01'), ...)` (Anastasiya Tsarkova).
- Added the `clickhouse-copier` tool for copying between clusters and resharding data (beta).
- Added consistent hashing functions: `yandexConsistentHash`, `jumpConsistentHash`, `sumburConsistentHash`. They can be used as a sharding key in order to reduce the amount of network traffic during subsequent reshardings.
- Added functions: `arrayAny`, `arrayAll`, `hasAny`, `hasAll`, `arrayIntersect`, `arrayResize`.
- Added the `arrayCumSum` function (Javi Santana).
- Added the `parseDateTimeBestEffort`, `parseDateTimeBestEffortOrZero`, and `parseDateTimeBestEffortOrNull` functions to read the `DateTime` from a string containing text in a wide variety of possible formats.
- Data can be partially reloaded from external dictionaries during updating (load just the records in which the value of the specified field greater than in the previous download) (Arsen Hakobyan).
- Added the `cluster` table function. Example: `cluster(cluster_name, db, table)`. The `remote` table function can accept the cluster name as the first argument, if it is specified as an identifier.
- The `remote` and `cluster` table functions can be used in `INSERT` queries.
- Added the `create_table_query` and `engine_full` virtual columns to the `system.tables` table. The `metadata_modification_time` column is virtual.
- Added the `data_path` and `metadata_path` columns to `system.tables` and `system.databases` tables, and added the `path` column to the `system.parts` and `system.parts_columns` tables.
- Added additional information about merges in the `system.part_log` table.
- An arbitrary partitioning key can be used for the `system.query_log` table (Kirill Shvakov).
- The `SHOW TABLES` query now also shows temporary tables. Added temporary tables and the `is_temporary` column to `system.tables` (zhang2014).
- Added `DROP TEMPORARY TABLE` and `EXISTS TEMPORARY TABLE` queries (zhang2014).
- Support for `SHOW CREATE TABLE` for temporary tables (zhang2014).
- Added the `system_profile` configuration parameter for the settings used by internal processes.
- Support for loading `object_id` as an attribute in `MongoDB` dictionaries (Pavel Litvinenko).
- Reading `null` as the default value when loading data for an external dictionary with the `MongoDB` source (Pavel Litvinenko).
- Reading `DateTime` values in the `Values` format from a Unix timestamp without single quotes.
- Failover is supported in `remote` table functions for cases when some of the replicas are missing the requested table.
- Configuration settings can be overridden in the command line when you run `clickhouse-server`. Example: `clickhouse-server -- --logger.level=information`.

- Implemented the `empty` function from a `FixedString` argument: the function returns 1 if the string consists entirely of null bytes (zhang2014).
- Added the `listen_try` configuration parameter for listening to at least one of the listen addresses without quitting, if some of the addresses can't be listened to (useful for systems with disabled support for IPv4 or IPv6).
- Added the `VersionedCollapsingMergeTree` table engine.
- Support for rows and arbitrary numeric types for the `library` dictionary source.
- `MergeTree` tables can be used without a primary key (you need to specify `ORDER BY tuple()`).
- A `Nullable` type can be `CAST` to a non-`Nullable` type if the argument is not `NULL`.
- `RENAME TABLE` can be performed for `VIEW`.
- Added the `throwIf` function.
- Added the `odbc_default_field_size` option, which allows you to extend the maximum size of the value loaded from an ODBC source (by default, it is 1024).
- The `system.processes` table and `SHOW PROCESSLIST` now have the `is_cancelled` and `peak_memory_usage` columns.

Improvements:

- Limits and quotas on the result are no longer applied to intermediate data for `INSERT SELECT` queries or for `SELECT` subqueries.
- Fewer false triggers of `force_restore_data` when checking the status of `Replicated` tables when the server starts.
- Added the `allow_distributed_ddl` option.
- Nondeterministic functions are not allowed in expressions for `MergeTree` table keys.
- Files with substitutions from `config.d` directories are loaded in alphabetical order.
- Improved performance of the `arrayElement` function in the case of a constant multidimensional array with an empty array as one of the elements. Example: `[[1], []][x]`.
- The server starts faster now when using configuration files with very large substitutions (for instance, very large lists of IP networks).
- When running a query, table valued functions run once. Previously, `remote` and `mysql` table valued functions performed the same query twice to retrieve the table structure from a remote server.
- The `MkDocs` documentation generator is used.
- When you try to delete a table column that `DEFAULT/MATERIALIZED` expressions of other columns depend on, an exception is thrown (zhang2014).
- Added the ability to parse an empty line in text formats as the number 0 for `Float` data types. This feature was previously available but was lost in release 1.1.54342.
- `Enum` values can be used in `min`, `max`, `sum` and some other functions. In these cases, it uses the corresponding numeric values. This feature was previously available but was lost in the release 1.1.54337.
- Added `max_expanded_ast_elements` to restrict the size of the AST after recursively expanding aliases.

Bug fixes:

- Fixed cases when unnecessary columns were removed from subqueries in error, or not removed from subqueries containing `UNION ALL`.
- Fixed a bug in merges for `ReplacingMergeTree` tables.
- Fixed synchronous insertions in `Distributed` tables (`insert_distributed_sync = 1`).
- Fixed segfault for certain uses of `FULL` and `RIGHT JOIN` with duplicate columns in subqueries.
- Fixed segfault for certain uses of `replace_running_query` and `KILL QUERY`.
- Fixed the order of the `source` and `last_exception` columns in the `system.dictionaries` table.
- Fixed a bug when the `DROP DATABASE` query did not delete the file with metadata.
- Fixed the `DROP DATABASE` query for `Dictionary` databases.
- Fixed the low precision of `uniqHLL12` and `uniqCombined` functions for cardinalities greater than 100 million items (Alex Bocharov).
- Fixed the calculation of implicit default values when necessary to simultaneously calculate default explicit expressions in `INSERT` queries (zhang2014).
- Fixed a rare case when a query to a `MergeTree` table couldn't finish (chenxing-xc).
- Fixed a crash that occurred when running a `CHECK` query for `Distributed` tables if all shards are local (chenxing.xc).
- Fixed a slight performance regression with functions that use regular expressions.

- Fixed a performance regression when creating multidimensional arrays from complex expressions.
- Fixed a bug that could cause an extra **FORMAT** section to appear in an **.sql** file with metadata.
- Fixed a bug that caused the **max_table_size_to_drop** limit to apply when trying to delete a **MATERIALIZED VIEW** looking at an explicitly specified table.
- Fixed incompatibility with old clients (old clients were sometimes sent data with the **DateTime('timezone')** type, which they do not understand).
- Fixed a bug when reading **Nested** column elements of structures that were added using **ALTER** but that are empty for the old partitions, when the conditions for these columns moved to **PREWHERE**.
- Fixed a bug when filtering tables by virtual **_table** columns in queries to **Merge** tables.
- Fixed a bug when using **ALIAS** columns in **Distributed** tables.
- Fixed a bug that made dynamic compilation impossible for queries with aggregate functions from the **quantile** family.
- Fixed a race condition in the query execution pipeline that occurred in very rare cases when using **Merge** tables with a large number of tables, and when using **GLOBAL** subqueries.
- Fixed a crash when passing arrays of different sizes to an **arrayReduce** function when using aggregate functions from multiple arguments.
- Prohibited the use of queries with **UNION ALL** in a **MATERIALIZED VIEW**.
- Fixed an error during initialization of the **part_log** system table when the server starts (by default, **part_log** is disabled).

Backward incompatible changes:

- Removed the **distributed_ddl_allow_replicated_alter** option. This behavior is enabled by default.
- Removed the **strict_insert_defaults** setting. If you were using this functionality, write to clickhouse-feedback@yandex-team.com.
- Removed the **UnsortedMergeTree** engine.

Clickhouse Release 1.1.54343, 2018-02-05

- Added macros support for defining cluster names in distributed DDL queries and constructors of Distributed tables: **CREATE TABLE distr ON CLUSTER '{cluster}' (...) ENGINE = Distributed('{cluster}', 'db', 'table')**
- Now queries like **SELECT ... FROM table WHERE expr IN (subquery)** are processed using the **table** index.
- Improved processing of duplicates when inserting to Replicated tables, so they no longer slow down execution of the replication queue.

Clickhouse Release 1.1.54342, 2018-01-22

This release contains bug fixes for the previous release 1.1.54337:

- Fixed a regression in 1.1.54337: if the default user has readonly access, then the server refuses to start up with the message **Cannot create database in readonly mode**.
- Fixed a regression in 1.1.54337: on systems with systemd, logs are always written to syslog regardless of the configuration; the watchdog script still uses **init.d**.
- Fixed a regression in 1.1.54337: wrong default configuration in the Docker image.
- Fixed nondeterministic behavior of GraphiteMergeTree (you can see it in log messages **Data after merge is not byte-identical to the data on another replicas**).
- Fixed a bug that may lead to inconsistent merges after **OPTIMIZE** query to Replicated tables (you may see it in log messages **Part ... intersects the previous part**).
- Buffer tables now work correctly when **MATERIALIZED** columns are present in the destination table (by zhang2014).
- Fixed a bug in implementation of **NULL**.

Clickhouse Release 1.1.54337, 2018-01-18

New features:

- Added support for storage of multi-dimensional arrays and tuples (**Tuple** data type) in tables.

- Support for table functions for `DESCRIBE` and `INSERT` queries. Added support for subqueries in `DESCRIBE`. Examples: `DESC TABLE remote('host', default.hits);` `DESC TABLE (SELECT 1);` `INSERT INTO TABLE FUNCTION remote('host', default.hits);`. Support for `INSERT INTO TABLE` in addition to `INSERT INTO`.
- Improved support for time zones. The `DateTime` data type can be annotated with the timezone that is used for parsing and formatting in text formats. Example: `DateTime('Europe/Moscow')`. When timezones are specified in functions for `DateTime` arguments, the return type will track the timezone, and the value will be displayed as expected.
- Added the functions `toTimeZone`, `timeDiff`, `toQuarter`, `toRelativeQuarterNum`. The `toRelativeHour/Minute/Second` functions can take a value of type `Date` as an argument. The `now` function name is case-sensitive.
- Added the `toStartOfFifteenMinutes` function (Kirill Shvakov).
- Added the `clickhouse format` tool for formatting queries.
- Added the `format_schema_path` configuration parameter (Marek Vavruša). It is used for specifying a schema in `Cap'n Proto` format. Schema files can be located only in the specified directory.
- Added support for config substitutions (`incl` and `conf.d`) for configuration of external dictionaries and models (Pavel Yakunin).
- Added a column with documentation for the `system.settings` table (Kirill Shvakov).
- Added the `system.parts_columns` table with information about column sizes in each data part of `MergeTree` tables.
- Added the `system.models` table with information about loaded `CatBoost` machine learning models.
- Added the `mysql` and `odbc` table function and corresponding `MySQL` and `ODBC` table engines for accessing remote databases. This functionality is in the beta stage.
- Added the possibility to pass an argument of type `AggregateFunction` for the `groupArray` aggregate function (so you can create an array of states of some aggregate function).
- Removed restrictions on various combinations of aggregate function combinators. For example, you can use `avgForEachIf` as well as `avgIfForEach` aggregate functions, which have different behaviors.
- The `-ForEach` aggregate function combinator is extended for the case of aggregate functions of multiple arguments.
- Added support for aggregate functions of `Nullable` arguments even for cases when the function returns a non-`Nullable` result (added with the contribution of Silviu Caragea). Example: `groupArray`, `groupUniqArray`, `topK`.
- Added the `max_client_network_bandwidth` for `clickhouse-client` (Kirill Shvakov).
- Users with the `readonly = 2` setting are allowed to work with TEMPORARY tables (CREATE, DROP, INSERT...) (Kirill Shvakov).
- Added support for using multiple consumers with the `Kafka` engine. Extended configuration options for `Kafka` (Marek Vavruša).
- Added the `intExp3` and `intExp4` functions.
- Added the `sumKahan` aggregate function.
- Added the `to * Number Or Null functions`, where `* Number` is a numeric type.
- Added support for `WITH` clauses for an `INSERT SELECT` query (author: zhang2014).
- Added settings: `http_connection_timeout`, `http_send_timeout`, `http_receive_timeout`. In particular, these settings are used for downloading data parts for replication. Changing these settings allows for faster failover if the network is overloaded.
- Added support for `ALTER` for tables of type `Null` (Anastasiya Tsarkova).
- The `reinterpretAsString` function is extended for all data types that are stored contiguously in memory.
- Added the `--silent` option for the `clickhouse-local` tool. It suppresses printing query execution info in stderr.
- Added support for reading values of type `Date` from text in a format where the month and/or day of the month is specified using a single digit instead of two digits (Amos Bird).

Performance optimizations:

- Improved performance of aggregate functions `min`, `max`, `any`, `anyLast`, `anyHeavy`, `argMin`, `argMax` from string arguments.
- Improved performance of the functions `isInfinite`, `isFinite`, `isNaN`, `roundToExp2`.
- Improved performance of parsing and formatting `Date` and `DateTime` type values in text format.
- Improved performance and precision of parsing floating point numbers.
- Lowered memory usage for `JOIN` in the case when the left and right parts have columns with identical names that are not contained in `USING`.

- Improved performance of aggregate functions `varSamp`, `varPop`, `stddevSamp`, `stddevPop`, `covarSamp`, `covarPop`, `corr` by reducing computational stability. The old functions are available under the names `varSampStable`, `varPopStable`, `stddevSampStable`, `stddevPopStable`, `covarSampStable`, `covarPopStable`, `corrStable`.

Bug fixes:

- Fixed data deduplication after running a `DROP` or `DETACH PARTITION` query. In the previous version, dropping a partition and inserting the same data again was not working because inserted blocks were considered duplicates.
- Fixed a bug that could lead to incorrect interpretation of the `WHERE` clause for `CREATE MATERIALIZED VIEW` queries with `POPULATE`.
- Fixed a bug in using the `root_path` parameter in the `zookeeper_servers` configuration.
- Fixed unexpected results of passing the `Date` argument to `toStartOfDay`.
- Fixed the `addMonths` and `subtractMonths` functions and the arithmetic for `INTERVAL n MONTH` in cases when the result has the previous year.
- Added missing support for the `UUID` data type for `DISTINCT`, `JOIN`, and `uniq` aggregate functions and external dictionaries (Evgeniy Ivanov). Support for `UUID` is still incomplete.
- Fixed `SummingMergeTree` behavior in cases when the rows summed to zero.
- Various fixes for the `Kafka` engine (Marek Vavruša).
- Fixed incorrect behavior of the `Join` table engine (Amos Bird).
- Fixed incorrect allocator behavior under FreeBSD and OS X.
- The `extractAll` function now supports empty matches.
- Fixed an error that blocked usage of `libressl` instead of `openssl`.
- Fixed the `CREATE TABLE AS SELECT` query from temporary tables.
- Fixed non-atomicity of updating the replication queue. This could lead to replicas being out of sync until the server restarts.
- Fixed possible overflow in `gcd`, `lcm` and `modulo` (`%` operator) (Maks Skorokhod).
- `-preprocessed` files are now created after changing `umask` (`umask` can be changed in the config).
- Fixed a bug in the background check of parts (`MergeTreePartChecker`) when using a custom partition key.
- Fixed parsing of tuples (values of the `Tuple` data type) in text formats.
- Improved error messages about incompatible types passed to `multif`, `array` and some other functions.
- Redesigned support for `Nullable` types. Fixed bugs that may lead to a server crash. Fixed almost all other bugs related to `NULL` support: incorrect type conversions in `INSERT SELECT`, insufficient support for `Nullable` in `HAVING` and `PREWHERE`, `join_use_nulls` mode, `Nullable` types as arguments of `OR` operator, etc.
- Fixed various bugs related to internal semantics of data types. Examples: unnecessary summing of `Enum` type fields in `SummingMergeTree`; alignment of `Enum` types in `Pretty` formats, etc.
- Stricter checks for allowed combinations of composite columns.
- Fixed the overflow when specifying a very large parameter for the `FixedString` data type.
- Fixed a bug in the `topK` aggregate function in a generic case.
- Added the missing check for equality of array sizes in arguments of n-ary variants of aggregate functions with an `-Array` combinator.
- Fixed a bug in `--pager` for `clickhouse-client` (author: ks1322).
- Fixed the precision of the `exp10` function.
- Fixed the behavior of the `visitParamExtract` function for better compliance with documentation.
- Fixed the crash when incorrect data types are specified.
- Fixed the behavior of `DISTINCT` in the case when all columns are constants.
- Fixed query formatting in the case of using the `tupleElement` function with a complex constant expression as the tuple element index.
- Fixed a bug in `Dictionary` tables for `range_hashed` dictionaries.
- Fixed a bug that leads to excessive rows in the result of `FULL` and `RIGHT JOIN` (Amos Bird).
- Fixed a server crash when creating and removing temporary files in `config.d` directories during config reload.
- Fixed the `SYSTEM DROP DNS CACHE` query: the cache was flushed but addresses of cluster nodes were not updated.
- Fixed the behavior of `MATERIALIZED VIEW` after executing `DETACH TABLE` for the table under the view (Marek Vavruša).

Build improvements:

- The **pbuilder** tool is used for builds. The build process is almost completely independent of the build host environment.
- A single build is used for different OS versions. Packages and binaries have been made compatible with a wide range of Linux systems.
- Added the **clickhouse-test** package. It can be used to run functional tests.
- The source tarball can now be published to the repository. It can be used to reproduce the build without using GitHub.
- Added limited integration with Travis CI. Due to limits on build time in Travis, only the debug build is tested and a limited subset of tests are run.
- Added support for **Cap'n'Proto** in the default build.
- Changed the format of documentation sources from **Restricted Text** to **Markdown**.
- Added support for **systemd** (Vladimir Smirnov). It is disabled by default due to incompatibility with some OS images and can be enabled manually.
- For dynamic code generation, **clang** and **lld** are embedded into the **clickhouse** binary. They can also be invoked as **clickhouse clang** and **clickhouse lld**.
- Removed usage of GNU extensions from the code. Enabled the **-Wextra** option. When building with **clang** the default is **libc++** instead of **libstdc++**.
- Extracted **clickhouse_parsers** and **clickhouse_common_io** libraries to speed up builds of various tools.

Backward incompatible changes:

- The format for marks in **Log** type tables that contain **Nullable** columns was changed in a backward incompatible way. If you have these tables, you should convert them to the **TinyLog** type before starting up the new server version. To do this, replace **ENGINE = Log** with **ENGINE = TinyLog** in the corresponding **.sql** file in the **metadata** directory. If your table doesn't have **Nullable** columns or if the type of your table is not **Log**, then you don't need to do anything.
- Removed the **experimental_allow_extended_storage_definition_syntax** setting. Now this feature is enabled by default.
- The **runningIncome** function was renamed to **runningDifferenceStartingWithFirstvalue** to avoid confusion.
- Removed the **FROM ARRAY JOIN arr** syntax when **ARRAY JOIN** is specified directly after **FROM** with no table (Amos Bird).
- Removed the **BlockTabSeparated** format that was used solely for demonstration purposes.
- Changed the state format for aggregate functions **varSamp**, **varPop**, **stddevSamp**, **stddevPop**, **covarSamp**, **covarPop**, **corr**. If you have stored states of these aggregate functions in tables (using the **AggregateFunction** data type or materialized views with corresponding states), please write to clickhouse-feedback@yandex-team.com.
- In previous server versions there was an undocumented feature: if an aggregate function depends on parameters, you can still specify it without parameters in the **AggregateFunction** data type. Example: **AggregateFunction(quantiles, UInt64)** instead of **AggregateFunction(quantiles(0.5, 0.9), UInt64)**. This feature was lost. Although it was undocumented, we plan to support it again in future releases.
- Enum data types cannot be used in min/max aggregate functions. This ability will be returned in the next release.

Please note when upgrading:

- When doing a rolling update on a cluster, at the point when some of the replicas are running the old version of ClickHouse and some are running the new version, replication is temporarily stopped and the message **unknown parameter 'shard'** appears in the log. Replication will continue after all replicas of the cluster are updated.
- If different versions of ClickHouse are running on the cluster servers, it is possible that distributed queries using the following functions will have incorrect results: **varSamp**, **varPop**, **stddevSamp**, **stddevPop**, **covarSamp**, **covarPop**, **corr**. You should update all cluster nodes.

ClickHouse release 1.1.54327, 2017-12-21

This release contains bug fixes for the previous release 1.1.54318:

- Fixed bug with possible race condition in replication that could lead to data loss. This issue affects versions 1.1.54310 and 1.1.54318. If you use one of these versions with Replicated tables, the update is strongly recommended. This issue shows in logs in Warning messages like `Part ... from own log doesn't exist`. The issue is relevant even if you don't see these messages in logs.

ClickHouse release 1.1.54318, 2017-11-30

This release contains bug fixes for the previous release 1.1.54310:

- Fixed incorrect row deletions during merges in the SummingMergeTree engine
- Fixed a memory leak in unreplicated MergeTree engines
- Fixed performance degradation with frequent inserts in MergeTree engines
- Fixed an issue that was causing the replication queue to stop running
- Fixed rotation and archiving of server logs

ClickHouse release 1.1.54310, 2017-11-01

New features:

- Custom partitioning key for the MergeTree family of table engines.
- **Kafka** table engine.
- Added support for loading **CatBoost** models and applying them to data stored in ClickHouse.
- Added support for time zones with non-integer offsets from UTC.
- Added support for arithmetic operations with time intervals.
- The range of values for the Date and DateTime types is extended to the year 2105.
- Added the **CREATE MATERIALIZED VIEW x TO y** query (specifies an existing table for storing the data of a materialized view).
- Added the **ATTACH TABLE** query without arguments.
- The processing logic for Nested columns with names ending in `-Map` in a SummingMergeTree table was extracted to the `sumMap` aggregate function. You can now specify such columns explicitly.
- Max size of the IP trie dictionary is increased to 128M entries.
- Added the `getsizeofEnumType` function.
- Added the `sumWithOverflow` aggregate function.
- Added support for the Cap'n Proto input format.
- You can now customize compression level when using the `zstd` algorithm.

Backward incompatible changes:

- Creation of temporary tables with an engine other than Memory is not allowed.
- Explicit creation of tables with the View or MaterializedView engine is not allowed.
- During table creation, a new check verifies that the sampling key expression is included in the primary key.

Bug fixes:

- Fixed hangups when synchronously inserting into a Distributed table.
- Fixed nonatomic adding and removing of parts in Replicated tables.
- Data inserted into a materialized view is not subjected to unnecessary deduplication.
- Executing a query to a Distributed table for which the local replica is lagging and remote replicas are unavailable does not result in an error anymore.
- Users don't need access permissions to the **default** database to create temporary tables anymore.
- Fixed crashing when specifying the Array type without arguments.
- Fixed hangups when the disk volume containing server logs is full.
- Fixed an overflow in the `toRelativeWeekNum` function for the first week of the Unix epoch.

Build improvements:

- Several third-party libraries (notably Poco) were updated and converted to git submodules.

ClickHouse release 1.1.54304, 2017-10-19

New features:

- TLS support in the native protocol (to enable, set `tcp_ssl_port` in `config.xml`).

Bug fixes:

- `ALTER` for replicated tables now tries to start running as soon as possible.
- Fixed crashing when reading data with the setting `preferred_block_size_bytes=0`.
- Fixed crashes of `clickhouse-client` when pressing `Page Down`
- Correct interpretation of certain complex queries with `GLOBAL IN` and `UNION ALL`
- `FREEZE PARTITION` always works atomically now.
- Empty POST requests now return a response with code 411.
- Fixed interpretation errors for expressions like `CAST(1 AS Nullable(UInt8))`.
- Fixed an error when reading `Array(Nullable(String))` columns from `MergeTree` tables.
- Fixed crashing when parsing queries like `SELECT dummy AS dummy, dummy AS b`
- Users are updated correctly with invalid `users.xml`
- Correct handling when an executable dictionary returns a non-zero response code.

ClickHouse release 1.1.54292, 2017-09-20

New features:

- Added the `pointInPolygon` function for working with coordinates on a coordinate plane.
- Added the `sumMap` aggregate function for calculating the sum of arrays, similar to `SummingMergeTree`.
- Added the `trunc` function. Improved performance of the rounding functions (`round`, `floor`, `ceil`, `roundToExp2`) and corrected the logic of how they work. Changed the logic of the `roundToExp2` function for fractions and negative numbers.
- The ClickHouse executable file is now less dependent on the libc version. The same ClickHouse executable file can run on a wide variety of Linux systems. There is still a dependency when using compiled queries (with the setting `compile = 1` , which is not used by default).
- Reduced the time needed for dynamic compilation of queries.

Bug fixes:

- Fixed an error that sometimes produced `part ... intersects previous part` messages and weakened replica consistency.
- Fixed an error that caused the server to lock up if ZooKeeper was unavailable during shutdown.
- Removed excessive logging when restoring replicas.
- Fixed an error in the `UNION ALL` implementation.
- Fixed an error in the `concat` function that occurred if the first column in a block has the `Array` type.
- Progress is now displayed correctly in the `system.merges` table.

ClickHouse release 1.1.54289, 2017-09-13

New features:

- `SYSTEM` queries for server administration: `SYSTEM RELOAD DICTIONARY`, `SYSTEM RELOAD DICTIONARIES`, `SYSTEM DROP DNS CACHE`, `SYSTEM SHUTDOWN`, `SYSTEM KILL`.
- Added functions for working with arrays: `concat`, `arraySlice`, `arrayPushBack`, `arrayPushFront`, `arrayPopBack`, `arrayPopFront`.
- Added `root` and `identity` parameters for the ZooKeeper configuration. This allows you to isolate individual users on the same ZooKeeper cluster.
- Added aggregate functions `groupBitAnd`, `groupBitOr`, and `groupBitXor` (for compatibility, they are also available under the names `BIT_AND`, `BIT_OR`, and `BIT_XOR`).
- External dictionaries can be loaded from MySQL by specifying a socket in the filesystem.
- External dictionaries can be loaded from MySQL over SSL (`ssl_cert`, `ssl_key`, `ssl_ca` parameters).
- Added the `max_network_bandwidth_for_user` setting to restrict the overall bandwidth use for queries per user.
- Support for `DROP TABLE` for temporary tables.
- Support for reading `DateTime` values in Unix timestamp format from the `CSV` and `JSONEachRow` formats.
- Lagging replicas in distributed queries are now excluded by default (the default threshold is 5 minutes).
- FIFO locking is used during `ALTER`: an `ALTER` query isn't blocked indefinitely for continuously running queries.
- Option to set `umask` in the config file.

- Improved performance for queries with `DISTINCT`.

Bug fixes:

- Improved the process for deleting old nodes in ZooKeeper. Previously, old nodes sometimes didn't get deleted if there were very frequent inserts, which caused the server to be slow to shut down, among other things.
- Fixed randomization when choosing hosts for the connection to ZooKeeper.
- Fixed the exclusion of lagging replicas in distributed queries if the replica is localhost.
- Fixed an error where a data part in a `ReplicatedMergeTree` table could be broken after running `ALTER MODIFY` on an element in a `Nested` structure.
- Fixed an error that could cause `SELECT` queries to "hang".
- Improvements to distributed DDL queries.
- Fixed the query `CREATE TABLE ... AS <materialized view>`.
- Resolved the deadlock in the `ALTER ... CLEAR COLUMN IN PARTITION` query for `Buffer` tables.
- Fixed the invalid default value for `Enum`s (0 instead of the minimum) when using the `JSONEachRow` and `TSKV` formats.
- Resolved the appearance of zombie processes when using a dictionary with an `executable` source.
- Fixed segfault for the `HEAD` query.

Improved workflow for developing and assembling ClickHouse:

- You can use `pbuilder` to build ClickHouse.
- You can use `libc++` instead of `libstdc++` for builds on Linux.
- Added instructions for using static code analysis tools: `Coverage`, `clang-tidy`, `cppcheck`.

Please note when upgrading:

- There is now a higher default value for the MergeTree setting `max_bytes_to_merge_at_max_space_in_pool` (the maximum total size of data parts to merge, in bytes): it has increased from 100 GiB to 150 GiB. This might result in large merges running after the server upgrade, which could cause an increased load on the disk subsystem. If the free space available on the server is less than twice the total amount of the merges that are running, this will cause all other merges to stop running, including merges of small data parts. As a result, `INSERT` queries will fail with the message "Merges are processing significantly slower than inserts." Use the `SELECT * FROM system.merges` query to monitor the situation. You can also check the `DiskSpaceReservedForMerge` metric in the `system.metrics` table, or in Graphite. You don't need to do anything to fix this, since the issue will resolve itself once the large merges finish. If you find this unacceptable, you can restore the previous value for the `max_bytes_to_merge_at_max_space_in_pool` setting. To do this, go to the section in `config.xml`, set `<merge_tree>`<max_bytes_to_merge_at_max_space_in_pool>107374182400</max_bytes_to_merge_at_max_space_in_pool>` and restart the server.

ClickHouse release 1.1.54284, 2017-08-29

- This is a bugfix release for the previous 1.1.54282 release. It fixes leaks in the parts directory in ZooKeeper.

ClickHouse release 1.1.54282, 2017-08-23

This release contains bug fixes for the previous release 1.1.54276:

- Fixed `DB::Exception: Assertion violation: !_path.empty()` when inserting into a Distributed table.
- Fixed parsing when inserting in RowBinary format if input data starts with `;`.
- Errors during runtime compilation of certain aggregate functions (e.g. `groupArray()`).

Clickhouse Release 1.1.54276, 2017-08-16

New features:

- Added an optional `WITH` section for a `SELECT` query. Example query: `WITH 1+1 AS a SELECT a, a*a`
- `INSERT` can be performed synchronously in a Distributed table: `OK` is returned only after all the data is saved on all the shards. This is activated by the setting `insert_distributed_sync=1`.
- Added the `UUID` data type for working with 16-byte identifiers.

- Added aliases of CHAR, FLOAT and other types for compatibility with the Tableau.
- Added the functions toYYYYMM, toYYYYMMDD, and toYYYYMMDDhhmmss for converting time into numbers.
- You can use IP addresses (together with the hostname) to identify servers for clustered DDL queries.
- Added support for non-constant arguments and negative offsets in the function `substring(str, pos, len)`.
- Added the max_size parameter for the `groupArray(max_size)(column)` aggregate function, and optimized its performance.

Main changes:

- Security improvements: all server files are created with 0640 permissions (can be changed via config parameter).
- Improved error messages for queries with invalid syntax.
- Significantly reduced memory consumption and improved performance when merging large sections of MergeTree data.
- Significantly increased the performance of data merges for the ReplacingMergeTree engine.
- Improved performance for asynchronous inserts from a Distributed table by combining multiple source inserts. To enable this functionality, use the setting `distributed_directory_monitor_batch_inserts=1`.

Backward incompatible changes:

- Changed the binary format of aggregate states of `groupArray(array_column)` functions for arrays.

Complete list of changes:

- Added the `output_format_json_quote_denormals` setting, which enables outputting nan and inf values in JSON format.
- Optimized stream allocation when reading from a Distributed table.
- Settings can be configured in readonly mode if the value doesn't change.
- Added the ability to retrieve non-integer granules of the MergeTree engine in order to meet restrictions on the block size specified in the `preferred_block_size_bytes` setting. The purpose is to reduce the consumption of RAM and increase cache locality when processing queries from tables with large columns.
- Efficient use of indexes that contain expressions like `toStartOfHour(x)` for conditions like `toStartOfHour(x) op constexpr`.
- Added new settings for MergeTree engines (the `merge_tree` section in `config.xml`):
- `replicated_deduplication_window_seconds` sets the number of seconds allowed for deduplicating inserts in Replicated tables.
- `cleanup_delay_period` sets how often to start cleanup to remove outdated data.
- `replicated_can_become_leader` can prevent a replica from becoming the leader (and assigning merges).
- Accelerated cleanup to remove outdated data from ZooKeeper.
- Multiple improvements and fixes for clustered DDL queries. Of particular interest is the new setting `distributed_ddl_task_timeout`, which limits the time to wait for a response from the servers in the cluster.
- Improved display of stack traces in the server logs.
- Added the "none" value for the compression method.
- You can use multiple `dictionaries_config` sections in `config.xml`.
- It is possible to connect to MySQL through a socket in the file system.
- The `system.parts` table has a new column with information about the size of marks, in bytes.

Bug fixes:

- Distributed tables using a Merge table now work correctly for a SELECT query with a condition on the `_table` field.
- Fixed a rare race condition in ReplicatedMergeTree when checking data parts.
- Fixed possible freezing on "leader election" when starting a server.
- The `max_replica_delay_for_distributed_queries` setting was ignored when using a local replica of the data source. This has been fixed.
- Fixed incorrect behavior of `ALTER TABLE CLEAR COLUMN IN PARTITION` when attempting to clean a non-existing column.
- Fixed an exception in the `multIf` function when using empty arrays or strings.
- Fixed excessive memory allocations when deserializing Native format.

- Fixed incorrect auto-update of Trie dictionaries.
- Fixed an exception when running queries with a GROUP BY clause from a Merge table when using SAMPLE.
- Fixed a crash of GROUP BY when using distributed_aggregation_memory_efficient=1.
- Now you can specify the database.table in the right side of IN and JOIN.
- Too many threads were used for parallel aggregation. This has been fixed.
- Fixed how the "if" function works with FixedString arguments.
- SELECT worked incorrectly from a Distributed table for shards with a weight of 0. This has been fixed.
- Running `CREATE VIEW IF EXISTS` no longer causes crashes.
- Fixed incorrect behavior when input_format_skip_unknown_fields=1 is set and there are negative numbers.
- Fixed an infinite loop in the `dictGetHierarchy()` function if there is some invalid data in the dictionary.
- Fixed `Syntax error: unexpected (...)` errors when running distributed queries with subqueries in an IN or JOIN clause and Merge tables.
- Fixed an incorrect interpretation of a SELECT query from Dictionary tables.
- Fixed the "Cannot mmap" error when using arrays in IN and JOIN clauses with more than 2 billion elements.
- Fixed the failover for dictionaries with MySQL as the source.

Improved workflow for developing and assembling ClickHouse:

- Builds can be assembled in Arcadia.
- You can use gcc 7 to compile ClickHouse.
- Parallel builds using ccache+distcc are faster now.

ClickHouse release 1.1.54245, 2017-07-04

New features:

- Distributed DDL (for example, `CREATE TABLE ON CLUSTER`)
- The replicated query `ALTER TABLE CLEAR COLUMN IN PARTITION`.
- The engine for Dictionary tables (access to dictionary data in the form of a table).
- Dictionary database engine (this type of database automatically has Dictionary tables available for all the connected external dictionaries).
- You can check for updates to the dictionary by sending a request to the source.
- Qualified column names
- Quoting identifiers using double quotation marks.
- Sessions in the HTTP interface.
- The OPTIMIZE query for a Replicated table can run not only on the leader.

Backward incompatible changes:

- Removed SET GLOBAL.

Minor changes:

- Now after an alert is triggered, the log prints the full stack trace.
- Relaxed the verification of the number of damaged/extra data parts at startup (there were too many false positives).

Bug fixes:

- Fixed a bad connection "sticking" when inserting into a Distributed table.
- GLOBAL IN now works for a query from a Merge table that looks at a Distributed table.
- The incorrect number of cores was detected on a Google Compute Engine virtual machine. This has been fixed.
- Changes in how an executable source of cached external dictionaries works.
- Fixed the comparison of strings containing null characters.
- Fixed the comparison of Float32 primary key fields with constants.
- Previously, an incorrect estimate of the size of a field could lead to overly large allocations.
- Fixed a crash when querying a Nullable column added to a table using ALTER.
- Fixed a crash when sorting by a Nullable column, if the number of rows is less than LIMIT.
- Fixed an ORDER BY subquery consisting of only constant values.

- Previously, a Replicated table could remain in the invalid state after a failed DROP TABLE.
- Aliases for scalar subqueries with empty results are no longer lost.
- Now a query that used compilation does not fail with an error if the .so file gets damaged.

Fixed in ClickHouse Release 18.12.13, 2018-09-10

CVE-2018-14672

Functions for loading CatBoost models allowed path traversal and reading arbitrary files through error messages.

Credits: Andrey Krasichkov of Yandex Information Security Team

Fixed in ClickHouse Release 18.10.3, 2018-08-13

CVE-2018-14671

unixODBC allowed loading arbitrary shared objects from the file system which led to a Remote Code Execution vulnerability.

Credits: Andrey Krasichkov and Evgeny Sidorov of Yandex Information Security Team

Fixed in ClickHouse Release 1.1.54388, 2018-06-28

CVE-2018-14668

"remote" table function allowed arbitrary symbols in "user", "password" and "default_database" fields which led to Cross Protocol Request Forgery Attacks.

Credits: Andrey Krasichkov of Yandex Information Security Team

Fixed in ClickHouse Release 1.1.54390, 2018-07-06

CVE-2018-14669

ClickHouse MySQL client had "LOAD DATA LOCAL INFILE" functionality enabled that allowed a malicious MySQL database read arbitrary files from the connected ClickHouse server.

Credits: Andrey Krasichkov and Evgeny Sidorov of Yandex Information Security Team

Fixed in ClickHouse Release 1.1.54131, 2017-01-10

CVE-2018-14670

Incorrect configuration in deb package could lead to unauthorized use of the database.

Credits: the UK's National Cyber Security Centre (NCSC)