ClickHouse Documentation

Выпуск

Yandex LLC

авг. 16, 2017

Оглавление

Глава 1

Introduction

What is ClickHouse?

ClickHouse is a columnar DBMS for OLAP.

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

```
5123456789123456789     1       Eurobasket - Greece - Bosnia and Herzegovina - example.com      1 ␣
↪      2011-09-01 01:03:02     6274717   1294101174      11409   612345678912345678      0       ␣
↪33      6       http://www.example.com/basketball/team/123/match/456789.html http://www.example.
↪com/basketball/team/123/match/987654.html       0       1366    768     32      10      3183    ␣
↪ 0       0       13      0\0     1       1       0       0                       2011142 -1      ␣
↪0               0       01321      613     660     2011-09-01 08:01:17     0       0       0       ␣
↪ 0       utf-8   1466    0       0       0       5678901234567890123                277789954   ␣
↪  0       0       0       0       0
5234985259563631958     0       Consulting, Tax assessment, Accounting, Law       1       2011-09-
↪01 01:03:02     6320881   2111222333      213     6458937489576391093     0       3       2       ␣
↪ http://www.example.ru/         0       800     600       16      10      2       153.1   0       ␣
↪ 0       10      63      1       1       0       0                       2111678 000       0       ␣
↪  588     368     240     2011-09-01 01:03:17     4       0       60310   0       windows-1251    ␣
↪ 1466    0       000                778899001       0       0       0       0       0
...
```

In other words, all the values related to a row are stored next to each other. Examples of a row-oriented DBMS are MySQL, Postgres, MS SQL Server, and others.

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

```
WatchID:    5385521489354350662     5385521490329509958     5385521489953706054    ␣
↪5385521490476781638     5385521490583269446     5385521490218868806     5385521491437850694  ␣
↪5385521491090174022      5385521490792669254     5385521490420695110     5385521491532181574   ␣
↪5385521491559694406     5385521491459625030     5385521492275175494   5385521492781318214    ␣
↪5385521492710027334     5385521492955615302     5385521493708759110     5385521494506434630    ␣
↪5385521493104611398
JavaEnable: 1       0       1       0       0       0       1       0       1       1       1       ␣
↪ 1       1       1       0       1       0       0       1       1
```

```
Title:      Yandex  Announcements - Investor Relations - Yandex     Yandex - Contact us - Moscow ␣
↪ Yandex - Mission        Ru      Yandex - History - History of Yandex    Yandex Financial␣
↪Releases - Investor Relations - Yandex Yandex - Locations      Yandex Board of Directors -␣
↪Corporate Governance - Yandex       Yandex - Technologies
GoodEvent:  1       1       1       1       1       1       1       1       1       1       1     ␣
↪ 1       1       1       1       1       1       1       1       1
EventTime:  2016-05-18 05:19:20     2016-05-18 08:10:20     2016-05-18 07:38:00     2016-05-18␣
↪01:13:08     2016-05-18 00:04:06     2016-05-18 04:21:30     2016-05-18 00:34:16     2016-05-18␣
↪07:35:49     2016-05-18 11:41:59     2016-05-18 01:13:32
```

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, Paraccel (Actian Matrix) (Amazon Redshift), Sybase IQ, Exasol, Infobright, InfiniDB, MonetDB (VectorWise) (Actian Vector), LucidDB, SAP HANA, Google Dremel, Google PowerDrill, Druid, kdb+, etc.

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 to the scenario, and the more specific 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 of the scenarios.

We'll say that the following is true for the OLAP (online analytical processing) 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).
- There are no transactions.
- 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. That is, data is filtered or aggregated. 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 Elliptics for analytics, you will get very poor performance compared to OLAP databases.

Columnar-oriented databases are better suited to OLAP scenarios (at least 100 times better in processing speed for most queries), for the following reasons:

1. For I/O.
2. 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 expect a 20-fold reduction in I/O.
3. 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.
4. 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:

```
milovidov@hostname:~$ 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.)
```

```
:)
```

2. For 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, there isn't any "garbage" stored with the values. For example, 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 separate columns separately, but that can't effectively process analytical queries due to their optimization for other scenarios. Example 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.

Also note that ClickHouse is a DBMS, 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 really improves performance.

Disk storage of data

Many column-oriented DBMSs (SAP HANA, and Google PowerDrill) can only work in RAM. But even on thousands of servers, the RAM is too small for storing all the pageviews and sessions in Yandex.Metrica.

Parallel processing on multiple cores

Large queries are parallelized in a natural way.

Distributed processing on multiple servers

Almost none of the columnar DBMSs listed above have support for distributed 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

If you are familiar with standard SQL, we can't really talk about SQL support. NULLs are not supported. All the functions have different names. However, this is a declarative query language based on SQL that can't be differentiated from SQL in many instances. JOINs are supported. Subqueries are supported in FROM, IN, JOIN clauses; and scalar subqueries. Correlated subqueries 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 performance.

Real time data updates

ClickHouse supports primary key tables. 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. There is no locking when adding data.

Indexes

Having a primary key allows, for example, extracting data for specific clients (Metrica counters) for a specific time range, with low latency less than several dozen milliseconds.

Suitable for online queries

This lets us use the system as the back-end for a web interface. Low latency means queries can be processed without delay, while the Yandex.Metrica interface page is loading (in online mode).

Support for approximated calculations

1. The system contains aggregate functions for approximated calculation of the number of various values, medians, and quantiles.
2. Supports 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. Supports 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 support for data integrity on replicas

Uses asynchronous multi-master replication. After being written to any available replica, data is distributed to all the remaining replicas. The system maintains identical data on different replicas. Data is restored automatically after a failure, or using a "button" for complex cases. For more information, see the section "Data replication".

ClickHouse features that can be considered disadvantages

1. No transactions.
2. For aggregation, query results must fit in the RAM on a single server. However, the volume of source data for a query may be indefinitely large.
3. Lack of full-fledged UPDATE/DELETE implementation.

The Yandex.Metrica task

ClickHouse currently powers Yandex.Metrica, world's second largest web analytics platform, with over 13 trillion database records and over 20 billion events a day, generating customized reports on the fly directly from non-aggregated data.

We need to get custom reports based on hits and sessions, with custom segments set by the user. Data for the reports is updated in real-time. Queries must be run immediately (in online mode). We must be able to build reports for any time period. Complex aggregates must be calculated, such as the number of unique visitors. At this time (April 2014), Yandex.Metrica receives approximately 12 billion events (pageviews and mouse clicks) daily. All these events must be stored in order to build custom reports. A single query may require scanning hundreds of millions of rows over a few seconds, or millions of rows in no more than a few hundred milliseconds.

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.

ClickHouse is also used for:

- Storing WebVisor data.
- Processing intermediate data.
- Building global reports with Analytics.
- Running queries for debugging the 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 will not view all the reports we calculate 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.

Possible silly questions

Why not to use systems like MapReduce?

Systems like map-reduce are distributed computing systems, where the reduce phase is performed using distributed sorting. Regarding this aspect, map-reduce is similar to other systems like YAMR, Hadoop, YT.

These systems are not suitable for online queries because of latency, So they can't be used in backend-level for web interface. Systems like this also are not suitable for real-time updates. Distributed sorting is not optimal solution for reduce operations, if the result of the operation and all intermediate results, shall they exist, fit in operational memory of a single server, as usually happens in case of online analytical queries. In this case the optimal way to perform reduce operations is by using a hash-table. A common optimization method for map-reduce tasks is combine operation (partial reduce) which uses hash-tables in memory. This optimization is done by the user manually. Distributed sorting is the main reason for long latencies of simple map-reduce jobs.

Systems similar to map-reduce enable running any code on the cluster. But for OLAP use-cases declarative query languages are better suited as they allow to carry out investigations faster. For example, for Hadoop there are Hive and Pig. There are others: Cloudera Impala, Shark (deprecated) and Spark SQL for Spark, Presto, Apache Drill. However, performance of such tasks is highly sub-optimal compared to the performance of specialized systems and relatively high latency does not allow the use of these systems as a backend for the web interface.

YT allows you to store separate groups of columns. But YT is not a truly columnar storage system, as the system has no fixed length data types (so you can efficiently store a number without "garbage"), and there is no vector engine. Tasks in YT are performed by arbitrary code in streaming mode, so can not be sufficiently optimized (up to hundreds of millions of lines per second per server). In 2014-2016 YT is to develop "dynamic table sorting" functionality using Merge Tree, strongly typed values and SQL-like language support. Dynamically sorted tables are not suited for OLAP tasks, since the data is stored in rows. Query language development in YT is still in incubating phase, which does not allow it to focus on this functionality. YT developers are considering dynamically sorted tables for use in OLTP and Key-Value scenarios.

Performance

According to internal testing results, ClickHouse shows the best performance for comparable operating scenarios among systems of its class that were available for testing. This includes the highest throughput for long queries, and the lowest latency on short queries. Testing results are shown on this page.

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 on data insertion

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 Yandex 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.

Глава 2

Getting started

System requirements

This is not a cross-platform system. It requires Linux Ubuntu Precise (12.04) or newer, x86_64 architecture with SSE 4.2 instruction set. To test for SSE 4.2 support, do:

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

We recommend using Ubuntu Trusty or Ubuntu Xenial or Ubuntu Precise. The terminal must use UTF-8 encoding (the default in Ubuntu).

Installation

For testing and development, the system can be installed on a single server or on a desktop computer.

Installing from packages

In */etc/apt/sources.list* (or in a separate */etc/apt/sources.list.d/clickhouse.list* file), add the repository:

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

For other Ubuntu versions, replace *trusty* to *xenial* or *precise*. If you want to use the most fresh testing version of ClickHouse, replace *stable* to *testing*.

Then run:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv E0C56BD4    # optional
sudo apt-get update
sudo apt-get install clickhouse-client clickhouse-server-common
```

You can also download and install packages manually from here: http://repo.yandex.ru/clickhouse/trusty/pool/main/c/clickhouse/, http://repo.yandex.ru/clickhouse/xenial/pool/main/c/clickhouse/, http://repo.yandex.ru/clickhouse/precise/pool/main/c/clickhouse/.

ClickHouse contains access restriction settings. They are located in the ‘users.xml’ file (next to ‘config.xml’). By default, access is allowed from everywhere for the default user without a password. See ‘user/default/networks’. For more information, see the section “Configuration files”.

Installing from source

To build, follow the instructions in build.md (for Linux) or in build_osx.md (for Mac OS X).

You can compile packages and install them. You can also use programs without installing packages.

```
Client: dbms/src/Client/
Server: dbms/src/Server/
```

For the server, create a catalog with data, such as:

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

(Configured in the server config.) Run ‘chown’ for the desired user.

Note the path to logs in the server config (src/dbms/src/Server/config.xml).

Other methods of installation

The Docker image is located here: https://hub.docker.com/r/yandex/clickhouse-server/

There is Gentoo overlay located here: https://github.com/kmeaw/clickhouse-overlay

Launch

To start the server (as a daemon), run:

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

View the logs in the catalog */var/log/clickhouse-server/*

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

You can also 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’.

You can use the command-line client to connect to the server:

```
clickhouse-client
```

The default parameters indicate connecting with localhost:9000 on behalf of the user 'default' without a password. The client can be used for connecting to a remote server. For example:

```
clickhouse-client --host=example.com
```

For more information, see the section "Command-line client".

Checking the system:

```
milovidov@hostname:~/work/metrica/src/dbms/src/Client$ ./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, it works!

For further experiments you can try playing with one of the following example datasets:

AMPLab Big Data Benchmark

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

Register free account on https://aws.amazon.com - you will need credit card, email and phone number Create new access key on https://console.aws.amazon.com/iam/home?nc2=h_m_sc#security_credential

Execute this in shell:

```
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 ..
```

Execute these 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);
```

Back into shell:

```
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
```

Data retrieval queries:

```
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
```

Criteo Terabyte Click Logs

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

Create log table:

```
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
```

Load 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 table for 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)
```

Parse the data from raw log and put it into second table:

```
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;
```

NYC Taxi Dataset

How to create dataset from raw data

Look at https://github.com/toddwschneider/nyc-taxi-data and http://tech.marksblogg.com/billion-nyc-taxi-rides-redshift.html for description of the dataset and loading instructions.

Data will download to ~227 GB of uncompressed CSV files. It takes about one hour on 1 Gbit connection. (Parallel download from s3.amazonaws.com saturate at least half of one gigabit.) Some files could be downloaded incompletely. Look at suspicious file sizes and repeat downloading of incomplete files.

Some files contain broken rows. To correct them, run:

```
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 data must be preprocessed inside PostgreSQL. It will do point-in-polygon lookups (map points to areas of New York), and finally JOIN all data to single denormalized flat table. You must install PostgreSQL with PostGIS support.

When running `initialize_database.sh` script, be careful and check manually, that all tables get created successfully.

Processing of each month of yellow taxi data in PostgreSQL takes about 20-30 minutes, about 48 hours in total.

Check exact amount of loaded rows:

```
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)

Data in PostgreSQL takes 370 GB (346 GiB).

Export data from PostgreSQL:

```
COPY
(
    SELECT trips.id,
           trips.vendor_id,
           trips.pickup_datetime,
           trips.dropoff_datetime,
           trips.store_and_fwd_flag,
           trips.rate_code_id,
           trips.pickup_longitude,
           trips.pickup_latitude,
           trips.dropoff_longitude,
           trips.dropoff_latitude,
           trips.passenger_count,
           trips.trip_distance,
           trips.fare_amount,
           trips.extra,
           trips.mta_tax,
           trips.tip_amount,
           trips.tolls_amount,
           trips.ehail_fee,
           trips.improvement_surcharge,
           trips.total_amount,
           trips.payment_type,
           trips.trip_type,
           trips.pickup,
           trips.dropoff,

           cab_types.type cab_type,

           weather.precipitation_tenths_of_mm rain,
           weather.snow_depth_mm,
           weather.snowfall_mm,
           weather.max_temperature_tenths_degrees_celsius max_temp,
           weather.min_temperature_tenths_degrees_celsius min_temp,
           weather.average_wind_speed_tenths_of_meters_per_second wind,

           pick_up.gid pickup_nyct2010_gid,
           pick_up.ctlabel pickup_ctlabel,
           pick_up.borocode pickup_borocode,
           pick_up.boroname pickup_boroname,
           pick_up.ct2010 pickup_ct2010,
           pick_up.boroct2010 pickup_boroct2010,
           pick_up.cdeligibil pickup_cdeligibil,
           pick_up.ntacode pickup_ntacode,
           pick_up.ntaname pickup_ntaname,
           pick_up.puma pickup_puma,

           drop_off.gid dropoff_nyct2010_gid,
           drop_off.ctlabel dropoff_ctlabel,
           drop_off.borocode dropoff_borocode,
           drop_off.boroname dropoff_boroname,
           drop_off.ct2010 dropoff_ct2010,
           drop_off.boroct2010 dropoff_boroct2010,
           drop_off.cdeligibil dropoff_cdeligibil,
           drop_off.ntacode dropoff_ntacode,
           drop_off.ntaname dropoff_ntaname,
           drop_off.puma dropoff_puma
    FROM trips
    LEFT JOIN cab_types
        ON trips.cab_type_id = cab_types.id
    LEFT JOIN central_park_weather_observations_raw weather
        ON weather.date = trips.pickup_datetime::date
    LEFT JOIN nyct2010 pick_up
        ON pick_up.gid = trips.pickup_nyct2010_gid
    LEFT JOIN nyct2010 drop_off
        ON drop_off.gid = trips.dropoff_nyct2010_gid
) TO '/opt/milovidov/nyc-taxi-data/trips.tsv';
```

Dump is created at speed about 50 MB/sec. While creating dump, PostgreSQL reads data from disk at about 28 MB/sec. It takes about 5 hours. Resulting tsv file is 590612904969 bytes.

Create 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_boroct2010       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_boroct2010      Nullable(String),
dropoff_cdeligibil      Nullable(String),
dropoff_ntacode         Nullable(String),
dropoff_ntaname         Nullable(String),
dropoff_puma            Nullable(String)
) ENGINE = Log;
```

This table is needed to select better data types for fields, and if possible, get rid of NULLs.

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

real    75m56.214s
```

Data is read at 112-140 MB/sec. Loading data into Log table in single thread took 76 minutes. Data inside Log table takes 142 GB.

(You could also import data directly from Postgres, using `COPY ... TO PROGRAM`.)

Sadly, all weather-related fields (precipitation...average_wind_speed) are NULLs. We will omit them from final dataset.

For the first time, we will create a table on single server. Later we will distribute the table. Create and fill final 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, '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
```

```
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_boroname) AS Enum8('Manhattan' = 1, 'Queens' = 4, 'Brooklyn' = 3, '' =␣
↪0, 'Bronx' = 2, 'Staten Island' = 5)) AS dropoff_boroname,
toFixedString(ifNull(dropoff_ct2010, '000000'), 6) AS dropoff_ct2010,
toFixedString(ifNull(dropoff_boroct2010, '0000000'), 7) AS dropoff_boroct2010,
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,
toUInt16(ifNull(dropoff_puma, '0')) AS dropoff_puma

FROM trips
```

This is done in 3030 sec at about 428000 rows/sec. If you want faster load time, you may create table with `Log` engine instead of `MergeTree`. In that case, load is under 200 sec.

Table has used 126 GiB 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                     |
----------------
```

BTW, you could run OPTIMIZE for MergeTree table. But this is not necessary, everything will be fine.

Results on single server

Q1:

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

0.490 sec.

Q2:

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

1.224 sec.

Q3:

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

2.104 sec.

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 sec.

The server is:

Two-socket Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz, total 16 physical cores, 128 GiB RAM, 8x6 TB HDD in software RAID-5

Query time is best of three runs. Actually, starting from second run, the query reads data from OS page cache. No further caching happens: data is read and processed in each run.

Create table on cluster with 3 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_boroct2010␣
↪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-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
```

On source server:

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

This will distribute the data:

```
INSERT INTO trips_mergetree_x3 SELECT * FROM trips_mergetree
```

This takes 2454 sec.

On three servers:

Q1: 0.212 sec. Q2: 0.438 sec. Q3: 0.733 sec. Q4: 1.241 sec.

No surprise, as queries are linearly scalable.

Also there are results for 140-node cluster:

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

In that case, query execution speed is dominated by latency. We do queries from client located in Yandex datacenter in Mäntsälä (Finland) to cluster somewhere in Russia, that adds at least 20 ms of latency.

Summary

```
nodes   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
```

OnTime

This benchmark was created by Vadim Tkachenko, see:

- https://www.percona.com/blog/2009/10/02/analyzing-air-traffic-performance-with-infobright-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

Download the data:

```
for s in `seq 1987 2017`
do
for m in `seq 1 12`
do
wget http://transtats.bts.gov/PREZIP/On_Time_On_Time_Performance_${s}_${m}.zip
done
done
```

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

Create 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)
```

Load the 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
```

Queries:

Q0.

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

Q1. Count flights per day from 2000 to 2008 years

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

Q2. Count of flights delayed more than 10min per day of week for 2000-2008 years

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

Q3. Count of delays per airport for years 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. Count of delays per Carrier for 2007 year

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

Q5. Percentage of delays for each carrier for 2007 year.

```
SELECT Carrier, c, c2, c*1000/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;
```

More optimal version of same query:

```
SELECT Carrier, avg(DepDelay > 10) * 1000 AS c3 FROM ontime WHERE Year = 2007 GROUP BY Carrier␣
↪ORDER BY Carrier
```

Q6. Let’s try the same query for wide range of years 2000-2008.

```
SELECT Carrier, c, c2, c*1000/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;
```

More optimal version of same query:

```
SELECT Carrier, avg(DepDelay > 10) * 1000 AS c3 FROM ontime WHERE Year >= 2000 AND Year <= 2008␣
↪GROUP BY Carrier ORDER BY Carrier
```

Q7. Percent of delayed (more 10mins) flights per year.

```
SELECT Year, c1/c2
FROM
(
    select
        Year,
        count(*)*1000 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
```

More optimal version of same query:

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

Q8. Most popular destination in sense count of direct connected cities for different range of years.

```
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;
```

Star Schema

Compile dbgen: https://github.com/vadimtk/ssb-dbgen

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

You will see some warnings. It's Ok.

Place dbgen and dists.dss to some place with at least 800 GB free space available.

Generate data:

```
./dbgen -s 1000 -T c
./dbgen -s 1000 -T l
```

Create tables in ClickHouse:

```
CREATE TABLE lineorder (
        LO_ORDERKEY             UInt32,
        LO_LINENUMBER           UInt8,
        LO_CUSTKEY              UInt32,
        LO_PARTKEY              UInt32,
        LO_SUPPKEY              UInt32,
        LO_ORDERDATE            Date,
        LO_ORDERPRIORITY        String,
        LO_SHIPPRIORITY         UInt8,
        LO_QUANTITY             UInt8,
        LO_EXTENDEDPRICE        UInt32,
        LO_ORDTOTALPRICE        UInt32,
        LO_DISCOUNT             UInt8,
        LO_REVENUE              UInt32,
        LO_SUPPLYCOST           UInt32,
        LO_TAX                  UInt8,
        LO_COMMITDATE           Date,
        LO_SHIPMODE             String
)Engine=MergeTree(LO_ORDERDATE,(LO_ORDERKEY,LO_LINENUMBER,LO_ORDERDATE),8192);

CREATE TABLE customer (
        C_CUSTKEY       UInt32,
        C_NAME              String,
        C_ADDRESS       String,
        C_CITY          String,
        C_NATION        String,
        C_REGION        String,
        C_PHONE         String,
        C_MKTSEGMENT    String,
        C_FAKEDATE      Date
)Engine=MergeTree(C_FAKEDATE,(C_CUSTKEY,C_FAKEDATE),8192);

CREATE TABLE part (
        P_PARTKEY       UInt32,
        P_NAME          String,
        P_MFGR          String,
        P_CATEGORY      String,
        P_BRAND         String,
        P_COLOR         String,
        P_TYPE          String,
        P_SIZE          UInt8,
        P_CONTAINER     String,
        P_FAKEDATE      Date
)Engine=MergeTree(P_FAKEDATE,(P_PARTKEY,P_FAKEDATE),8192);

CREATE TABLE lineorderd AS lineorder ENGINE = Distributed(perftest_3shards_1replicas, default,␣
↪lineorder, rand());
CREATE TABLE customerd AS customer ENGINE = Distributed(perftest_3shards_1replicas, default,␣
↪customer, rand());
CREATE TABLE partd AS part ENGINE = Distributed(perftest_3shards_1replicas, default, part, rand());
```

For single-node setup, create just MergeTree tables. For Distributed setup, you must configure cluster `perftest_3shards_1replicas` in configuration file. Then create MergeTree tables on each node and then create Distributed tables.

Load data (change customer to customerd in case of distributed setup):

```
cat customer.tbl | sed 's/$/2000-01-01/' | clickhouse-client --query "INSERT INTO customer FORMAT␣
↪CSV"
cat lineorder.tbl | clickhouse-client --query "INSERT INTO lineorder FORMAT CSV"
```

WikiStat

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

Create table:

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

Load data:

```
for i in {2007..2016}; do for j in {01..12}; do echo $i-$j >&2; curl -sS "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 -1 /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]{2})([0-
↪9]{2})([0-9]{2})\.gz/\1-\2-\3 \4-00-00/')" | clickhouse-client --query="INSERT INTO wikistat␣
↪FORMAT TabSeparated"; done
```

Глава 3

Interfaces

To explore the system's capabilities, download data to tables, or make manual queries, use the clickhouse-client program.

Command-line client

To work for 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.

:) SELECT 1
```

The `clickhouse-client` program accepts the following parameters, which are all optional:

`--host, -h` - server name, by defaul - localhost. You can use either the name or the IPv4 or IPv6 address.

`--port` - The port to connect to, by default - 9000. Note that the HTTP interface and the native interface use different ports.

`--user, -u` - The username, by default - default.

`--password` - The password, by default - empty string.

`--query, -q` - Query to process when using non-interactive mode.

`--database, -d` - Select the current default database, by default - the current DB from the server settings (by default, the 'default' DB).

`--multiline, -m` - If specified, allow multiline queries (do not send request on Enter).

`--multiquery, -n` - If specified, allow processing multiple queries separated by semicolons. 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, -t` - If specified, print the query execution time to 'stderr' in non-interactive mode. `--stacktrace` - If specified, also prints the stack trace if an exception occurs. `--config-file` - Name of the configuration file that has additional settings or changed defaults for the settings listed above. By default, files are searched for in this order:

```
./clickhouse-client.xml
~/./clickhouse-client/config.xml
/etc/clickhouse-client/config.xml
```

Settings are only taken from the first file found.

You can also specify any settings that will be used for processing queries. For example, `clickhouse-client --max_threads=1`. For more information, see the section "Settings".

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 break, and the data in 'stdin'. This is convenient for large INSERT queries.

Examples for insert data via clickhouse-client:

```
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 a 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 break - 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.

You can specify `\G` instead of or after the semicolon. This indicates using 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 even nothing, depending on 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 : “exit”, “quit”, “logout”, “учше”, “йгше”, “дщпщге”, “exit;”, “quit;”, “logout;”, “учшеж”, “йгшеж”, “дщпщгеж”, “q”, “й”, “q”, “Q”, ”:q”, “й”, “Й”, “Жй”

When processing a query, the client shows: #. 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. #. The formatted query after parsing, for debugging. #. The result in the specified format. #. The number of lines in the result, the time passed, and the average speed of query processing.

To cancel a lengthy query, press 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 request processing”.

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 break 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 request in the ‘query’ parameter, and the rest in the POST (we’ll explain later why this is necessary). URL length is limited by 16KB, this limit should be taken into account when sending long queries in the ‘query’ parameter.

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

$ GET '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 not very convenient because spaces have to be URL-escaped. Although wget escapes everything on its own, we don't recommend 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 break 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' | POST 'http://localhost:8123/'
```

Using the familiar INSERT query for data insertion:

```
echo 'INSERT INTO t VALUES (1),(2),(3)' | POST 'http://localhost:8123/'
```

Data can be sent separately from the query:

```
echo '(4),(5),(6)' | POST 'http://localhost:8123/?query=INSERT INTO t VALUES'
```

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)' | POST 'http://localhost:8123/?query=INSERT INTO t FORMAT Values'
```

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

```
echo -ne '10\n11\n12\n' | POST 'http://localhost:8123/?query=INSERT INTO t FORMAT TabSeparated'
```

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

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

Deleting the table.

```
POST 'http://localhost:8123/?query=DROP TABLE t'
```

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

You can use compression when transmitting data. The compressed data has a non-standard format, and you will need to use a special compressor program to work with it (*sudo apt-get install compressor-metrika-yandex*).

If you specified 'compress=1' in the URL, the server will compress the data it sends you. If you specified 'decompress=1' in the URL, the server will decompress the same data that you pass in the POST method.

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

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 @-
```

3. Using ‘X-ClickHouse-User’ and ‘X-ClickHouse-Key’ headers. Example:

```
echo 'SELECT 1' | curl -H "X-ClickHouse-User: user" -H "X-ClickHouse-Key: password"  'http://
↪localhost:8123/' -d @-
```

If the user name is not indicated, the username ‘default’ is used. If the password is not indicated, an 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 section “Settings”.

```
$ 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”.

In contrast to the native interface, the HTTP interface does not support the concept of sessions or session settings, does not allow aborting a query (to be exact, it allows this in only a few cases), and does not show the progress of query processing. 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). It can also be passed as ‘X-ClickHouse-Quota’ header. 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”.

JDBC driver

There is official JDBC driver for ClickHouse. See here .

Native interface (TCP)

The native interface is used in the “clickhouse-client” command-line client for interaction between servers with distributed query processing, and also in C++ programs. We will only cover the command-line client.

Third-party client libraries

There exist third-party client libraries for ClickHouse:

- **Python:**
    - infi.clickhouse_orm
    - sqlalchemy-clickhouse
- **PHP**
    - clickhouse-php-client
    - PhpClickHouseClient
    - phpClickHouse
- **Go**
    - clickhouse (Go)
    - go-clickhouse
- **NodeJs**
    - clickhouse (NodeJs)
    - node-clickhouse
- **Perl**
    - perl-DBD-ClickHouse
    - HTTP-ClickHouse
    - AnyEvent-ClickHouse
- **Ruby**
    - clickhouse (Ruby)
- **R**
    - clickhouse-r
- **.NET**
    - ClickHouse-Net

Libraries was not tested by us. Ordering is arbitrary.

Third-party GUI

There are open source project Tabix company of SMI2, which implements a graphical web interface for ClickHouse.

Tabix key features: - works with ClickHouse from the browser directly, without installing additional software; - query editor that supports highlighting of SQL syntax ClickHouse, auto-completion for all objects, including dictionaries and context-sensitive help for built-in functions. - graphs, charts and geo-referenced for mapping query results; - interactive designer PivotTables (pivot) for query results; - graphical tools for analysis ClickHouse; - two color theme: light and dark.

Tabix documentation

Глава 4

Query language

clickhouse-local

Application `clickhouse-local` can fast processing of local files that store tables without resorting to deployment and configuration clickhouse-server ...

Queries

CREATE DATABASE

Creates the ‘db_name’ database.

```
CREATE DATABASE [IF NOT EXISTS] db_name
```

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 [TEMPORARY] TABLE [IF NOT EXISTS] [db.]name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS 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. For example: `RegionID UInt32`. Expressions can also be defined for default values (see below).

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] [db.]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 [TEMPORARY] TABLE [IF NOT EXISTS] [db.]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.

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.

Temporary tables

In all cases, if TEMPORARY is specified, a temporary table will be created. Temporary tables have the following characteristics: - Temporary tables disappear when the session ends, including if the connection is lost. - A temporary table is created with the Memory engine. The other table engines are not supported. - 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.

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 section)

Queries `CREATE`, `DROP`, `ALTER`, `RENAME` support distributed execution on cluster. For example, the following query creates `Distributed` table `all_hits` for each host of the cluster `cluster`:

```
CREATE TABLE IF NOT EXISTS all_hits ON CLUSTER cluster (p Date, i Int32) ENGINE = Distributed(cluster, default, hits)
```

To correctly execute such queries you need to have equal definitions of the cluster on each host (you can use *ZooKeeper substitutions* to syncronize configs on hosts) and connection to ZooKeeper quorum. Local version of the query will be eventually executed on each host of the cluster, even if some hosts are temporary unavaiable; on each host queries are executed stictly sequentually. At the moment, `ALTER` queries for replicated tables are not supported yet.

CREATE VIEW

`CREATE [MATERIALIZED] VIEW [IF NOT EXISTS] [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 can specify ENGINE - the table engine for storing data. By default, it uses the same engine as for the table that the SELECT query is made from.

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 ... query. 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.

The 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.

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.

ATTACH

The query is exactly the same as CREATE, except - The word ATTACH is used instead of CREATE. - The query doesn't 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.

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).

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 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.

DETACH

Deletes information about the table from the server. The server stops knowing about the table's existence.

```
DETACH TABLE [IF EXISTS] [db.]name
```

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.

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).
```

ALTER

The ALTER query is only supported for *MergeTree type tables, as well as for Merge and Distributed types. The query has several variations.

Column manipulations

Lets you change the table structure.

```
ALTER TABLE [db].name [ON CLUSTER cluster] ADD|DROP|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 name [type] [default_expr] [AFTER name_after]
```

Adds a new column to the table with the specified name, type, and default expression (see the section "Default expressions"). 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.

```
DROP COLUMN name
```

Deletes the column with the name 'name'.

Deletes data from the file system. Since this deletes entire files, the query is completed almost instantly.

```
MODIFY COLUMN name [type] [default_expr]
```

Changes the 'name' column's type to 'type' and/or the default expression to 'default_expr'. When changing the type, values are converted as if the 'toType' function were applied to them.

If only the default expression is changed, the query doesn't do anything complex, and is completed almost instantly.

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 stages of execution: - 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.

There is no support for changing the column type in arrays and nested data structures.

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 of columns in the primary key or the sampling key (columns that are in the ENGINE expression). Changing the type of columns in the primary key is allowed only if such change doesn't entail changing the actual data (e.g. adding the value to an Enum or changing the type from DateTime to UInt32 is allowed).

If the ALTER query is not sufficient for making 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.

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 the SELECT 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 (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.

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.

Manipulations with partitions and parts

Only works for tables in the MergeTree family. The following operations are available:

- `DETACH PARTITION` - Move a partition to the 'detached' directory and forget it.
- `DROP PARTITION` - Delete a partition.
- `ATTACH PART|PARTITION` - Add a new part or partition from the 'detached' directory to the table.

- `FREEZE PARTITION` - Create a backup of a partition.
- `FETCH PARTITION` - Download a partition from another server.

Each type of query is covered separately below.

A partition in a table is data for a single calendar month. This is determined by the values of the date key specified in the table engine parameters. Each month's data is stored separately in order to simplify manipulations with this data.

A "part" in the table is part of the data from a single partition, sorted by the primary key.

You can use the `system.parts` table to view the set of table parts and partitions:

```
SELECT * FROM system.parts WHERE active
```

`active` - Only count active parts. Inactive parts are, for example, source parts remaining after merging to a larger part - these parts are deleted approximately 10 minutes after merging.

Another way to view a set of parts and partitions is to go into the directory with table data. The directory with data is /var/lib/clickhouse/data/database/table/, where /var/lib/clickhouse/ is the path to ClickHouse data, 'database' is the database name, and 'table' is the table name. Example:

```
$ ls -l /var/lib/clickhouse/data/test/visits/
total 48
drwxrwxrwx 2 clickhouse clickhouse 20480 мая   13 02:58 20140317_20140323_2_2_0
drwxrwxrwx 2 clickhouse clickhouse 20480 мая   13 02:58 20140317_20140323_4_4_0
drwxrwxrwx 2 clickhouse clickhouse  4096 мая   13 02:55 detached
-rw-rw-rw- 1 clickhouse clickhouse     2 мая   13 02:58 increment.txt
```

Here `20140317_20140323_2_2_0`, `20140317_20140323_4_4_0` - are directories of parts.

Let's look at the name of the first part: `20140317_20140323_2_2_0`. * `20140317` - minimum date of part data * `20140323` - maximum date of part data * `2` - minimum number of the data block * `2` - maximum number of the data block * `0` - part level, depth of the merge tree that formed it

Each part corresponds to a single partition and contains data for a single month. 201403 - The partition name. A partition is a set of parts for a single month.

On an operating server, you can't manually change the set of parts or their data on the file system, since the server won't know about it. For non-replicated tables, you can do this when the server is stopped, but we don't recommended it. For replicated tables, the set of parts can't be changed in any case.

The 'detached' directory contains parts that are not used by the server - detached from the table using the ALTER ... DETACH query. Parts that are damaged are also moved to this directory, instead of deleting them. You can add, delete, or modify the data in the 'detached' directory at any time - the server won't know about this until you make the ALTER TABLE ... ATTACH query.

```
ALTER TABLE [db.]table DETACH PARTITION 'name'
```

Move all data for partitions named 'name' to the 'detached' directory and forget about them. The partition name is specified in YYYYMM format. It can be indicated in single quotes or without them.

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.

The query is replicated - data will be moved to the 'detached' directory and forgotten on all replicas. The query can only be sent to a leader replica. To find out if a replica is a leader, perform SELECT to the 'system.replicas' system table. Alternatively, it is easier to make a query on all replicas, and all except one will throw an exception.

```
ALTER TABLE [db.]table DROP PARTITION 'name'
```

Similar to the DETACH operation. Deletes data from the table. Data parts will be tagged as inactive and will be completely deleted in approximately 10 minutes. The query is replicated - data will be deleted on all replicas.

```
ALTER TABLE [db.]table ATTACH PARTITION|PART 'name'
```

Adds data to the table from the 'detached' directory.

It is possible to add data for an entire partition or a separate part. For a part, specify the full name of the part in single quotes.

The query is replicated. Each replica checks whether there is data in the 'detached' directory. If there is data, it checks the integrity, verifies that it matches the data on the server that initiated the query, and then adds it if everything is correct. 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 in the 'detached' directory on one replica, and use the ALTER ... ATTACH query to add it to the table on all replicas.

```
ALTER TABLE [db.]table FREEZE PARTITION 'name'
```

Creates a local backup of one or multiple partitions. The name can be the full name of the partition (for example, 201403), or its prefix (for example, 2014) - then the backup will be created for all the corresponding partitions.

The query does the following: for a data snapshot at the time of execution, it creates hardlinks to table data in the directory /var/lib/clickhouse/shadow/N/... /var/lib/clickhouse/ is the working ClickHouse directory from the config. N is the incremental number of the backup.

`/var/lib/clickhouse/` - working directory of ClickHouse from config file. `N` - incremental number of backup.

The same structure of directories is created inside the backup as inside `/var/lib/clickhouse/`. It also performs 'chmod' for all files, forbidding writes to them.

The backup is created almost instantly (but first it waits for current queries to the corresponding table to finish running). At first, the backup doesn't take any space on the disk. As the system works, the backup can take disk space, as data is modified. If the backup is made for old enough data, it won't take space on the disk.

After creating the backup, data from `/var/lib/clickhouse/shadow/` can be copied to the remote server and then deleted on the local server. The entire backup process is performed without stopping the server.

The `ALTER ... FREEZE PARTITION` query is not replicated. A local backup is only created on the local server.

As an alternative, you can manually copy data from the `/var/lib/clickhouse/data/database/table directory`. But if you do this while the server is running, race conditions are possible when copying directories with files being added or changed, and the backup may be inconsistent. You can do this if the server isn't running - then the resulting data will be the same as after the ALTER TABLE t FREEZE PARTITION query.

`ALTER TABLE ... FREEZE PARTITION` only copies data, not table metadata. To make a backup of table metadata, copy the file `/var/lib/clickhouse/metadata/database/table.sql`

To restore from a backup: * Use the CREATE query to create the table if it doesn't exist. The query can be taken from an .sql file (replace ATTACH in it with CREATE). * Copy data from the `data/database/table/` directory inside the backup to the `/var/lib/clickhouse/data/database/table/detached/` directory.

* Run `ALTER TABLE ... ATTACH PARTITION YYYYMM``queries where ``YYYYMM` is the month, for every month.

In this way, data from the backup will be added to the table. Restoring from a backup doesn't require stopping the server.

Backups and replication

Replication provides protection from device failures. If all data disappeared on one of your replicas, follow the instructions in the "Restoration after failure" section to restore it.

For protection from device failures, you must use replication. For more information about replication, see the section "Data replication".

Backups protect against human error (accidentally deleting data, deleting the wrong data or in the wrong cluster, or corrupting data). For high-volume databases, it can be difficult to copy backups to remote servers. In such cases, to protect from human error, you can keep a backup on the same server (it will reside in /var/lib/clickhouse/shadow/).

```
ALTER TABLE [db.]table FETCH PARTITION 'name' FROM 'path-in-zookeeper'
```

This query only works for replicatable tables.

It downloads the specified partition from the shard that has its ZooKeeper path specified in the FROM clause, then puts it in the 'detached' directory for the specified table.

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.

Data is placed in the 'detached' directory. You can use the ALTER TABLE ... ATTACH query to attach the data.

The path to ZooKeeper is specified in the FROM clause. For example, `/clickhouse/tables/01-01/visits`. Before downloading, the system checks that the partition exists and the table structure matches. The most appropriate replica is selected automatically from the healthy replicas.

The ALTER ... FETCH PARTITION query is not replicated. The partition will be downloaded to the 'detached' directory only on the local server. Note that if after this you use the ALTER TABLE ... ATTACH query to add data to the table, the data will be added on all replicas (on one of the replicas it will be added from the 'detached' directory, and on the rest it will be loaded from neighboring replicas).

Synchronicity of ALTER queries

For non-replicatable 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 - wait for own completion (default), 2 - wait for all.

SHOW DATABASES

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

Prints a list of all databases. This query is identical to the query `SELECT name FROM system.databases [INTO OUTFILE filename] [FORMAT format]` See the section "Formats".

SHOW TABLES

```
SHOW TABLES [FROM db] [LIKE 'pattern'] [INTO OUTFILE filename] [FORMAT format]
```

Outputs a list of * 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.

The query is identical to the query SELECT name FROM system.tables WHERE database = 'db' [AND name LIKE 'pattern'] [INTO OUTFILE filename] [FORMAT format] See the section "LIKE operator".

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 is 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 is 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 exactly the same as: SELECT * FROM system.processes [INTO OUTFILE filename] [FORMAT format].

Tip (execute in the console): `watch -n1 "clickhouse-client --query='SHOW PROCESSLIST'"`

SHOW CREATE TABLE

```
SHOW CREATE 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.

DESCRIBE TABLE

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

Returns two String-type columns: 'name' and 'type', which indicate the names and types of columns in the specified table.

Nested data structures are output in "expanded" format. Each column is shown separately, with the name after a dot.

EXISTS

```
EXISTS 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.

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.

SET

```
SET [GLOBAL] param = value
```

Lets you set the 'param' setting 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, settings made using SET are lost. To make settings that persist after a server restart, you can only use the server's config file.

OPTIMIZE

```
OPTIMIZE TABLE [db.]name [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 `PARTITION` is specified, then only specified partition will be optimized. If `FINAL` is specified, then optimization will be performed even if data inside the partition already optimized (i. e. all data is in single part).

INSERT

This query has several variations.

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

Inserts rows with the listed values in the ‘table’ table. This query is exactly the same as:

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

```
INSERT INTO [db.]table [(c1, c2, c3)] FORMAT format ...
```

Inserts data in any specified format. The data itself comes after ‘format’, after all space symbols up to the first line break if there is one and including it, or after all space symbols if there isn’t a line break. We recommend writing data starting from the next line (this is important if the data starts with space characters).

Example:

```
INSERT INTO t FORMAT TabSeparated
11  Hello, world!
22  Qwerty
```

For more information about data formats, see the section “Formats”. The “Interfaces” section describes how to insert data separately from the query when using the command-line client or the HTTP interface.

The query may optionally specify a list of columns for insertion. In this case, the default values are written to the other columns. Default values are calculated from DEFAULT expressions specified in table definitions, or, if the DEFAULT is not explicitly defined, zeros and empty strings are used. If the ‘strict_insert_default’ setting is set to 1, all the columns that do not have explicit DEFAULTS must be specified in the query.

```
INSERT INTO [db.]table [(c1, c2, c3)] SELECT ...
```

Inserts the result of the SELECT query into a table. The names and data types of the SELECT result must exactly match the table structure that data is inserted into, or the specified list of columns. To change column names, use synonyms (AS) in the SELECT query. To change data types, use type conversion functions (see the section “Functions”).

None of the data formats allows using expressions as values. In other words, you can’t write INSERT INTO t VALUES (now(), 1 + 1, DEFAULT).

There is no support for other data part modification queries: UPDATE, DELETE, REPLACE, MERGE, UPSERT, INSERT UPDATE. However, you can delete old data using ALTER TABLE ... DROP PARTITION.

SELECT

His Highness, the SELECT query.

```
SELECT [DISTINCT] expr_list
    [FROM [db.]table | (subquery) | table_function] [FINAL]
    [SAMPLE sample_coeff]
    [ARRAY JOIN ...]
    [GLOBAL] ANY|ALL INNER|LEFT 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]
```

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 may consume too much RAM, if appropriate restrictions are not defined (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. Merge join is not implemented.

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. Approximated query processing is only supported by MergeTree* type tables, and only if the sampling expression was specified during table creation (see the section "MergeTree engine").

SAMPLE has the format `SAMPLE k`, where 'k' is a decimal number from 0 to 1, or `SAMPLE n`, where 'n' is a sufficiently large integer.

In the first case, the query will be executed on 'k' percent of data. For example, `SAMPLE 0.1` runs the query on 10% of data. In the second case, the query will be executed on a sample of no more than 'n' rows. For example, SAMPLE 10000000 runs the query on a maximum of 10,000,000 rows.

Example:

```
SELECT
    Title,
    count() * 10 AS PageViews
FROM hits_distributed
SAMPLE 0.1
WHERE
    CounterID = 34
    AND toDate(EventDate) >= toDate('2013-01-29')
    AND toDate(EventDate) <= toDate('2013-02-04')
    AND NOT DontCountHits
    AND NOT Refresh
    AND Title != ''
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.

When using something like SAMPLE 10000000, there isn't any information about which relative percent of data was processed or what the aggregate functions should be multiplied by, so this method of writing is not always appropriate to the situation.

A sample with a relative coefficient is "consistent": if we look at all possible data that could be in the table, a sample (when using a single sampling expression specified during table creation) with the same coefficient always selects the same subset of possible data. In other words, a sample from different tables on different servers at different times is made the same way.

For example, a sample of user IDs takes rows with the same subset of all the possible user IDs from different tables. This allows using the sample in subqueries in the IN clause, as well as for manually correlating results of different queries with samples.

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.

ARRAY JOIN is essentially INNER JOIN with an array. Example:

```
:) CREATE TABLE arrays_test (s String, arr Array(UInt8)) ENGINE = Memory

CREATE TABLE arrays_test
(
    s String,
    arr Array(UInt8)
) ENGINE = Memory

Ok.

0 rows in set. Elapsed: 0.001 sec.

:) INSERT INTO arrays_test VALUES ('Hello', [1,2]), ('World', [3,4,5]), ('Goodbye', [])

INSERT INTO arrays_test VALUES
```

```

Ok.

3 rows in set. Elapsed: 0.001 sec.

:) SELECT * FROM arrays_test

SELECT *
FROM arrays_test

-s-----arr---
| Hello   | [1,2]   |
| World   | [3,4,5] |
| Goodbye | []      |
----------

3 rows in set. Elapsed: 0.001 sec.

:) SELECT s, arr FROM arrays_test ARRAY JOIN arr

SELECT s, arr
FROM arrays_test
ARRAY JOIN arr

-s----arr-
| Hello |   1 |
| Hello |   2 |
| World |   3 |
| World |   4 |
| World |   5 |
-------

5 rows in set. Elapsed: 0.001 sec.
```

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 by the original name. Example:

```
:) SELECT s, arr, a FROM arrays_test ARRAY JOIN arr AS a

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 |
-----------

5 rows in set. Elapsed: 0.001 sec.
```

Multiple arrays of the same size can be comma-separated in the ARRAY JOIN clause. In this case, JOIN is performed with them simultaneously (the direct sum, not the direct product). 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

SELECT s, arr, a, num, mapped
FROM arrays_test
ARRAY JOIN arr AS a, arrayEnumerate(arr) AS num, arrayMap(lambda(tuple(x), plus(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 |
------------------

5 rows in set. Elapsed: 0.002 sec.

:) SELECT s, arr, a, num, arrayEnumerate(arr) FROM arrays_test ARRAY JOIN arr AS a,␣
↪arrayEnumerate(arr) AS num

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]           |
-------------------------

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

ARRAY JOIN also works with nested data structures. Example:

```
:) CREATE TABLE nested_test (s String, nest Nested(x UInt8, y UInt32)) ENGINE = Memory

CREATE TABLE nested_test
(
    s String,
    nest Nested(
    x UInt8,
    y UInt32)
) ENGINE = Memory

Ok.

0 rows in set. Elapsed: 0.006 sec.

:) INSERT INTO nested_test VALUES ('Hello', [1,2], [10,20]), ('World', [3,4,5], [30,40,50]), (
↪'Goodbye', [], [])

INSERT INTO nested_test VALUES

Ok.
```

```
3 rows in set. Elapsed: 0.001 sec.

:) SELECT * FROM nested_test

SELECT *
FROM nested_test

-s-----nest.x--nest.y---
| Hello   | [1,2]   | [10,20]    |
| World   | [3,4,5] | [30,40,50] |
| Goodbye | []      | []         |
----------------

3 rows in set. Elapsed: 0.001 sec.

:) SELECT s, nest.x, nest.y FROM nested_test ARRAY JOIN nest

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 |
------------

5 rows in set. Elapsed: 0.001 sec.
```

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. Example:

```
:) SELECT s, nest.x, nest.y FROM nested_test ARRAY JOIN nest.x, nest.y

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 |
------------

5 rows in set. Elapsed: 0.001 sec.
```

This variation also makes sense:

```
:) SELECT s, nest.x, nest.y FROM nested_test ARRAY JOIN nest.x

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] |
--------------

5 rows in set. Elapsed: 0.001 sec.
```

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

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] |
---------------------

5 rows in set. Elapsed: 0.001 sec.
```

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

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 |
------------------------

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

The query can only specify a single ARRAY JOIN clause.

The corresponding conversion can be performed before the WHERE/PREWHERE clause (if its result is needed in this clause), or after completing WHERE/PREWHERE (to reduce the volume of calculations).

JOIN clause

The normal JOIN, which is not related to ARRAY JOIN described above.

```
[GLOBAL] ANY|ALL INNER|LEFT [OUTER] JOIN (subquery)|table USING columns_list
```

Performs joins with data from the subquery. At the beginning of query execution, the subquery specified after JOIN is run, and its result is saved in memory. Then it is read from the "left" table specified in the FROM clause, and while it is being read, for each of the read rows from the "left" table, rows are selected from the subquery results table (the "right" table) that meet the condition for matching the values of the columns specified in USING.

The table name can be specified instead of a 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.

All columns that are not needed for the JOIN are deleted from the subquery.

There are several types of JOINs:

INNER or LEFT - the type: If INNER is specified, the result will contain only those rows that have a matching row in the right table. If LEFT is specified, any rows in the left table that don't have matching rows in the right table will be assigned the default value - zeros or empty rows. LEFT OUTER may be written instead of LEFT; the word OUTER does not affect anything.

ANY or ALL - strictness: If ANY is specified and there are multiple matching rows in the right table, only the first one will be joined. If ALL is specified and there are multiple matching rows in the right table, the data will be multiplied by the number of these rows.

Using ALL corresponds to the normal JOIN semantic from standard SQL. Using ANY is optimal. If the right table has only one matching row, the results of ANY and ALL are the same. You must specify either ANY or ALL (neither of them is selected by default).

GLOBAL - distribution:

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 JOINs. For more information, see the section "Distributed subqueries" below.

Any combination of JOINs is possible. For example, `GLOBAL ANY LEFT OUTER JOIN`.

When running JOINs, 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 order of execution, 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.

Only one JOIN can be specified in a query (on a single level). To run multiple JOINs, you can put them in subqueries.

Each time a query is run with the same JOIN, the subquery is run again - 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. For more information, see the section “Table engines, Join”.

In some cases, it is more efficient to use IN instead of JOIN. Among the various types of JOINs, 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”.

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 not suitable for indexes 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 *MergeTree tables.

A query may simultaneously specify PREWHERE and WHERE. In this case, PREWHERE precedes WHERE.

Keep in mind that it does not make much sense for PREWHERE to only specify those columns that have an index, because when using an index, only the data blocks that match the index are read.

If the 'optimize_move_to_prewhere' 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 -- для каждого домена достаём первый попавшийся заголовок страницы
```

```
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()`.

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 is 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.

external memory GROUP BY

It is possible to turn on spilling temporary data to disk to limit memory consumption during the execution of GROUP BY. Value of `max_bytes_before_external_group_by` setting determines the maximum memory consumption before temporary data is dumped to the file system. If it is 0 (the default value), the feature is turned off.

When using max_bytes_before_external_group_by it is advisable to set max_memory_usage to an approximately twice greater value. The reason for this is that aggregation is executed in two stages: reading and generation of intermediate data (1) and merging of intermediate data (2). Spilling data to the filesystem can be performed only on stage 1. If the spilling did not happen, then stage 2 could consume up to the same amount of memory as stage 1.

For example: if max_memory_usage is equal 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. If dumping data to the file system happened at least once during the execution, maximum memory consumption would be just a little bit higher than max_bytes_before_external_group_by.

During distributed query execution external aggregation is performed on the remote servers. If you want the memory consumption on the originating server to be small, set distributed_aggregation_memory_efficient to 1. If distributed_aggregation_memory_efficient is turned on then during merging of the dumped data and also during merging of the query results from the remote servers, total memory consumption is no more than 1/256 * number of threads of the total amount of memory.

If external aggregation is turned on and total memory consumption was less than max_bytes_before_external_group_by (meaning that no spilling took place), the query performance is the same as when external aggregation is turned off. If some data was dumped, then execution time will be several times longer (approximately 3x).

If you have an ORDER BY clause with some small LIMIT after a GROUP BY, then ORDER BY will not consume significant amount of memory. But if no LIMIT is provided, don't forget to turn on external sorting (max_bytes_before_external_sort).

LIMIT N BY modifier

LIMIT N BY COLUMNS allows you to restrict top N rows per each group of COLUMNS. LIMIT N BY is unrelated to LIMIT clause. Key for LIMIT N BY could contain arbitrary number of columns or expressions.

Example:

```
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
```

will select top 5 referrers for each domain - device type pair, total number of rows - 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.

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. In this case, using DISTINCT is much more optimal.
- 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.

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.

'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.

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 `INTO OUTFILE` filename clause (where filename is a string literal) to redirect query output to a file filename. In contrast to MySQL the file is created on a client host. The query will fail if a file with the same filename already exists. INTO OUTFILE is available in the command-line client and clickhouse-local (a query sent via HTTP interface will fail).

Default output format is TabSeparated (the same as in the batch mode of command-line client).

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.

Distributed subqueries

There are two versions of INs with subqueries (and for JOINs): the regular `IN` / `JOIN`, and `GLOBAL IN` / `GLOBAL JOIN`. They differ in how they are run for distributed query processing.

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 JOIN`, first all the subqueries for `GLOBAL IN` / `GLOBAL JOIN` are run, 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** 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 the 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)
```

- calculates the overlap in the audiences of two websites.

This query will be sent to all the 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)
```

Each of the remote servers will start running the subquery. Since the subquery uses a distributed table, each remote server will re-send the subquery 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 execute the subquery

```
SELECT UserID FROM distributed_table WHERE CounterID = 34
```

and the result will be put in a temporary table in RAM. Then a query 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 together 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 in each datacenter, there will be at least one replica of each shard, and there is a fast network between them - for possibility to process query with transferring data only inside datacenter.

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.

KILL QUERY

```
KILL QUERY WHERE <where expression to SELECT FROM system.processes query> [SYNC|ASYNC|TEST]␣
↪[FORMAT format]
```

Tries to finish currently executing queries. Queries to be finished are selected from `system.processes` table according to expression after WHERE term.

Examples:

```
KILL QUERY WHERE query_id='2-857d-4a57-9ee0-327da5d60a90'
```

Finishes all queries with specified query_id.

```
KILL QUERY WHERE user='username' SYNC
```

Synchronously finishes all queries of user `username`.

Readonly users can kill only own queries.

Syntax

There are two types of parsers in the system: a full SQL parser (a recursive descent parser), and a 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. 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 section "Formats".

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 break, 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. C-style comments: 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).

Identifiers

Identifiers (column names, functions, and data types) can be quoted or non-quoted. Non-quoted identifiers start with a Latin letter or underscore, and continue with a Latin letter, underscore, or number. In other words, they must match the regex `^[a-zA-Z_][0-9a-zA-Z_]*$`. Examples: `x`, `_1`, `X_y__Z123_`. Quoted identifiers are placed in reversed quotation marks `` `id` `` (the same as in MySQL), and can indicate any set of bytes (non-empty). In addition, symbols (for example, the reverse quotation mark) inside this type of identifier can be backslash-escaped. Escaping rules are the same as for string literals (see below). We recommend using identifiers that do not need to be quoted.

Literals

There are numeric literals, string literals, and compound literals.

Numeric literals

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 literals

Only string literals in single quotes are supported. The enclosed characters can be backslash-escaped. The following escape sequences have special meanings: `\b`, `\f`, `\r`, `\n`, `\t`, `\0`, `\a`, `\v`, `\xHH`. In all other cases, escape sequences like c, where c is any character, are transformed to c. This means that the sequences `\'` and `\\` can be used. The value will have the String type.

Minimum set of symbols that must be escaped in string literal is `'` and `\`.

Compound literals

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. For more information, see the section "Operators". 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-type tables).

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)`. For more information, see the section "Operators" below.

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”.

Synonyms

In the SELECT query, expressions can specify synonyms using the AS keyword. Any expression is placed to the left of AS. The identifier name for the synonym is placed to the right of AS. As opposed to standard SQL, synonyms are not only declared on the top level of expressions:

```
SELECT (1 AS n) + 2, n
```

In contrast to standard SQL, synonyms can be used in all parts of a query, not just SELECT.

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 a synonym. A list of expressions is one or more expressions separated by commas. Functions and operators, in turn, can have expressions as arguments.

Глава 5

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.
- When reading data, the engine is only required to extract the necessary set of columns. However, in some cases, the query may be partially processed inside the table engine.

Note that for most serious tasks, you should use engines from the MergeTree family.

AggregatingMergeTree

This engine differs from `MergeTree` in that the merge combines the states of aggregate functions stored in the table for rows with the same primary key value.

In order for this to work, it uses the AggregateFunction data type and the -State and -Merge modifiers for aggregate functions. Let's examine it more closely.

There is an AggregateFunction data type, which is a parametric data type. As parameters, the name of the aggregate function is passed, then the types of its arguments. Examples:

```
CREATE TABLE t
(
    column1 AggregateFunction(uniq, UInt64),
    column2 AggregateFunction(anyIf, String, UInt8),
    column3 AggregateFunction(quantiles(0.5, 0.9), UInt64)
) ENGINE = ...
```

This type of column stores the state of an aggregate function.

To get this type of value, use aggregate functions with the ‘State’ suffix. Example: uniqState(UserID), quantilesState(0.5, 0.9)(SendTiming) - in contrast to the corresponding ‘uniq’ and ‘quantiles’ functions, these functions return the state, rather than the prepared value. In other words, they return an AggregateFunction type value.

An AggregateFunction type value can’t be output in Pretty formats. In other formats, these types of values are output as implementation-specific binary data. The AggregateFunction type values are not intended for output or saving in a dump.

The only useful thing you can do with AggregateFunction type values is combine the states and get a result, which essentially means to finish aggregation. Aggregate functions with the ‘Merge’ suffix are used for this purpose. Example: uniqMerge(UserIDState), where UserIDState has the AggregateFunction type.

In other words, an aggregate function with the ‘Merge’ suffix takes a set of states, combines them, and returns the result. As an example, these 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)
```

There is an AggregatingMergeTree engine. Its job during a merge is to combine the states of aggregate functions from different table rows with the same primary key value.

You can’t use a normal INSERT to insert a row in a table containing AggregateFunction columns, because you can’t explicitly define the AggregateFunction value. Instead, use INSERT SELECT with ‘-State’ aggregate functions for inserting data.

With SELECT from an AggregatingMergeTree table, use GROUP BY and aggregate functions with the ‘-Merge’ modifier in order to complete data aggregation.

You can use AggregatingMergeTree tables for incremental data aggregation, including for aggregated materialized views.

Example: Creating a materialized AggregatingMergeTree view that tracks the ‘test.visits’ table:

```
CREATE MATERIALIZED VIEW test.basic
ENGINE = AggregatingMergeTree(StartDate, (CounterID, StartDate), 8192)
AS SELECT
    CounterID,
    StartDate,
    sumState(Sign)    AS Visits,
    uniqState(UserID) AS Users
FROM test.visits
GROUP BY CounterID, StartDate;
```

Inserting data in the ‘test.visits’ table. Data will also be inserted in the view, where it will be aggregated:

```
INSERT INTO test.visits ...
```

Performing SELECT from the view using GROUP BY to finish data aggregation:

```
SELECT
    StartDate,
    sumMerge(Visits) AS Visits,
    uniqMerge(Users) AS Users
FROM test.basic
GROUP BY StartDate
ORDER BY StartDate;
```

You can create a materialized view like this and assign a normal view to it that finishes data aggregation.

Note that in most cases, using AggregatingMergeTree is not justified, since queries can be run efficiently enough on non-aggregated data.

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 - The level of parallelism. Physically, the table will be represented as 'num_layers' of independent buffers. The recommended value is 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, 10000000, 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 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.

PREWHERE, 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. Because of this, 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").

CollapsingMergeTree

This engine differs from MergeTree in that it allows automatic deletion, or "collapsing" certain pairs of rows when merging.

Yandex.Metrica has normal logs (such as hit logs) and change logs. Change logs are used for incrementally calculating statistics on data that is constantly changing. Examples are the log of session changes, or logs of changes to user histories. Sessions are constantly changing in Yandex.Metrica. For example, the number of hits per session increases. We refer to changes in any object as a pair (?old values, ?new values). Old values may be missing if the object was created. New values may be missing if the object was deleted. If the object was changed, but existed previously and was not deleted, both values are present. In the change log, one or two entries are made for each change. Each entry contains all the attributes that the object has, plus a special attribute for differentiating between the old and new values. When objects change, only the new entries are added to the change log, and the existing ones are not touched.

The change log makes it possible to incrementally calculate almost any statistics. To do this, we need to consider "new" rows with a plus sign, and "old" rows with a minus sign. In other words, incremental calculation is possible for all statistics whose algebraic structure contains an operation for taking the inverse of an element. This is true of most statistics. We can also calculate "idempotent" statistics, such as the number of unique visitors, since the unique visitors are not deleted when making changes to sessions.

This is the main concept that allows Yandex.Metrica to work in real time.

CollapsingMergeTree accepts an additional parameter - the name of an Int8-type column that contains the row's "sign". Example:

```
CollapsingMergeTree(EventDate, (CounterID, EventDate, intHash32(UniqID), VisitID), 8192, Sign)
```

Here, 'Sign' is a column containing -1 for "old" values and 1 for "new" values.

When merging, each group of consecutive identical primary key values (columns for sorting data) is reduced to no more than one row with the column value 'sign_column = -1' (the "negative row") and no more than one row with the column value 'sign_column = 1' (the "positive row"). In other words, entries from the change log are collapsed.

If the number of positive and negative rows matches, the first negative row and the last positive row are written. If there is one more positive row than negative rows, only the last positive row is written. If there is one more negative row than positive rows, only the first negative row is written. Otherwise, there will be a logical error and none of the rows will be written. (A logical error can occur if the same section of the log was accidentally inserted more than once. The error is just recorded in the server log, and the merge continues.)

Thus, collapsing should not change the results of calculating statistics. Changes are gradually collapsed so that in the end only the last value of almost every object is left. Compared to MergeTree, the CollapsingMergeTree engine allows a multifold reduction of data volume.

There are several ways to get completely "collapsed" data from a CollapsingMergeTree table:

1. Write a query with GROUP BY and aggregate functions that accounts for the sign. For example, to calculate quantity, write 'sum(Sign)' instead of 'count()'. To calculate the sum of something, write 'sum(Sign * x)' instead of 'sum(x)', and so on, and also add 'HAVING sum(Sign) > 0'. Not all amounts can be calculated this way. For example, the aggregate functions 'min' and 'max' can't be rewritten.
2. If you must 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.

Distributed

The Distributed engine by itself does not store data, 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. By default, 1. -->
            <weight>1</weight>
            <!-- Optional. Whether to write data to just one of the replicas. By default, false -␣
↪write data to all of the 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>
                <port>9000</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).

For each server, there are several parameters: mandatory: `'host'`, `'port'`, and optional: `'user'`, `'password'`.

- `host` - address of remote server. May be specified as domain name or IPv4 or IPv6 address. If you specify domain, server will perform DNS lookup at startup, and result will be cached till server shutdown. If DNS request is failed, server won't start. If you are changing DNS records, restart the server for new records to take effect.
- `port` - TCP-port for interserver communication (tcp_port in configuration file, usually 9000). Don't get confused with http_port.
- `user` - user name to connect to remote server. By default user is 'default'. This user must have access rights to connect to remote server. Access rights are managed in users.xml configuration file. For additional info, consider "Access rights" section.
- `password` - password to log in to remote server, in plaintext. Default is empty string.

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 config are updated on the fly, it does not require server restart. 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 [10, 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.

When using Replicated tables, it is possible to reshard data - look at "Resharding" section. But in many cases, better to do without it. 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.

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 the section. –**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 comma-separated list of column types. For example, `UInt64,String`. Columns will be named `_1`, `_2`, … –**structure** - 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 `name`.

Examples:

```
echo -ne "1\n2\n3\n" | clickhouse-client --query="SELECT count() FROM test.visits WHERE␣
↪TraficSourceID 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' passes 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.

File(InputFormat)

The data source is a file that stores data in one of the supported input formats (TabSeparated, Native, etc.) …

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, and `LEFT``|``INNER` - the type. 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. The only way to retrieve data is to use it as the "right-hand" table for JOIN.

Storing data on the disk is the same as for the Set engine.

Log

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 - 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.

MaterializedView

Used for implementing materialized views (for more information, see `CREATE MATERIALIZED VIEW`). 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").

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 with names that match the regex `'^WatchLog'`.

Instead of the database name, you can use a constant expression that returns a string. For example, `currentDatabase()`.

Regular expressions are re2 (similar to 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 don't do this.

The typical way to use the Merge engine is for working with a large number of TinyLog tables as if with a single table.

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.

A Merge table contains the virtual column **_table** of the String type. (If the table already has a '_table' column, the virtual column is named '_table1', and if it already has '_table1', it is named '_table2', and so on.) It contains the name of the table that data was read from.

If the WHERE or 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.

MergeTree

The MergeTree engine supports an index by primary key and by date, and provides the possibility to update data in real time. This is the most advanced table engine in ClickHouse. Don't confuse it with the Merge engine.

The engine accepts parameters: the name of a Date type column containing the date, a sampling expression (optional), a tuple that defines the table's primary key, and the index granularity. Example:

Example without sampling support:

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

Example with sampling support:

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

A MergeTree type table must have a separate column containing the date. In this example, it is the 'EventDate' column. The type of the date column must be 'Date' (not 'DateTime').

The primary key may be a tuple from any expressions (usually this is just a tuple of columns), or a single expression.

The sampling expression (optional) can be any expression. It must also be present in the primary key. The example uses a hash of user IDs to pseudo-randomly disperse data in the table for each CounterID and EventDate. In other words, when using the SAMPLE clause in a query, you get an evenly pseudo-random sample of data for a subset of users.

The table is implemented as a set of parts. Each part is sorted by the primary key. In addition, each part has the minimum and maximum date assigned. When inserting in the table, a new sorted part is created. The merge process is periodically initiated in the background. When merging, several parts are selected, usually the smallest ones, and then merged into one large sorted part.

In other words, incremental sorting occurs when inserting to the table. Merging is implemented so that the table always consists of a small number of sorted parts, and the merge itself doesn't do too much work.

During insertion, data belonging to different months is separated into different parts. The parts that correspond to different months are never combined. The purpose of this is to provide local data modification (for ease in backups).

Parts are combined up to a certain size threshold, so there aren't any merges that are too long.

For each part, an index file is also written. The index file contains the primary key value for every 'index_granularity' row in the table. In other words, this is an abbreviated index of sorted data.

For columns, "marks" are also written to each 'index_granularity' row so that data can be read in a specific range.

When reading from a table, the SELECT query is analyzed for whether indexes can be used. An index can be used if the WHERE or 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 above columns that are in the primary key or date, or Boolean operators over them.

Thus, it is possible to quickly run queries on one or many ranges of the primary key. In the example given, queries will work quickly for a specific counter, for a specific counter and range of dates, for a specific counter and date, for multiple counters and a range of dates, and so on.

```
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'))
```

All of these cases will use the index by date and by primary key. 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 this example, the index can't be used:

```
SELECT count() FROM table WHERE CounterID = 34 OR URL LIKE '%upyachka%'
```

The index by date only allows reading those parts that contain dates from the desired range. However, a data part may contain data for many dates (up to an entire month), while within a single part the data is ordered by the 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.

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.

The OPTIMIZE query is supported, which calls an extra merge step.

You can use a single large table and continually add data to it in small chunks - this is what MergeTree is intended for.

Data replication is possible for all types of tables in the MergeTree family (see the section "Data 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.

ReplacingMergeTree

This engine differs from `MergeTree` in that it can deduplicate data by primary key while merging.

For ReplacingMergeTree mode, last parameter is optional name of 'version' column. While merging, for all rows with same primary key, only one row is selected: last row, if version column was not specified, or last row with maximum version value, if specified.

Version column must have type of UInt family or `Date` or `DateTime`.

```
ReplacingMergeTree(EventDate, (OrderID, EventDate, BannerID, ...), 8192, ver)
```

Please note, that data is deduplicated only while merging process. Merges are processed in background. Exact time of merge is unspecified and you could not rely on it. Some part of data could be not merged at all. While you could trigger extra merge with OPTIMIZE query, it is not recommended, as OPTIMIZE will read and write vast amount of data.

This table engine is suitable for background removal of duplicate data to save space, but not suitable to guarantee of deduplication.

Developed for special purposes of not Yandex.Metrica department.

Data replication

ReplicatedMergeTree

ReplicatedCollapsingMergeTree

ReplicatedAggregatingMergeTree

ReplicatedSummingMergeTree

Replication is only supported for tables in the MergeTree family. 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.

INSERT and ALTER are replicated (for more information, see ALTER). Compressed data is replicated, not query texts. The CREATE, DROP, ATTACH, DETACH, and RENAME queries are not replicated. In other words, they belong to a single server. 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.

Replication is not related to sharding in any way. Replication works independently on each shard.

Replication is an optional feature. 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 For example, the version in the Ubuntu Precise package is too old.

You can specify any existing ZooKeeper cluster - 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 isn't used for SELECT queries. In other words, replication doesn't affect the productivity of SELECT queries - they work just as fast as for non-replicated tables.

For each INSERT query (more precisely, for each inserted block of data; the INSERT query contains a single block, or per block for every max_insert_block_size = 1048576 rows), approximately ten entries are made in ZooKeeper in several transactions. 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 this server, then sent to the other servers. Because it is asynchronous, recently inserted data appears on the other replicas with some latency. If a part of the replicas is not available, the data on them 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.

There are no quorum writes. You can't write data with confirmation that it was received by more than one replica. If you write a batch of data to one replica and the server with this data ceases to exist before the data has time to get to the other replicas, this data will be lost.

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.

Blocks of data 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. This only works for the last 100 blocks inserted in a table.

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`.

Two parameters are also added in the beginning of the parameters list - the path to the table in ZooKeeper, and the replica name in ZooKeeper.

Example:

```
ReplicatedMergeTree('/clickhouse/tables/{layer}-{shard}/hits', '{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 config 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.

`hits` - 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.

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 everything explicitly instead of using substitutions. This might be convenient for testing and for configuring small clusters, but it is inconvenient when working with large clusters.

Run CREATE TABLE 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 command to recover all replicated tables:

```
sudo -u clickhouse touch /var/lib/clickhouse/flags/force_restore_data
```

Then launch 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, launch the server and make all the ATTACH TABLE queries that should have been in the .sql files in /var/lib/clickhouse/metadata/.)
4. Create the `/path_to_table/replica_name/flags/force_restore_data` node in ZooKeeper with any content or run command to recover all replicated tables: `sudo -u clickhouse touch /var/lib/clickhouse/flags/force_restore_data`

Then launch the server (restart it 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

From here on, we use `MergeTree` to refer to all the table engines in the `MergeTree` family, including `ReplicatedMergeTree`.

If you had a MergeTree table that was manually replicated, you can convert it to a replicatable 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 PART on one of the replicas to add these data parts to the working set.

If exactly the same parts exist on the other replicas, they are added to the working set on them. If not, the parts are downloaded from the replica that has them.

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 you lost ZooKeeper, you can save data by moving it to an unreplicated table as described above.

Resharding

```
ALTER TABLE t RESHARD [COPY] [PARTITION partition] TO cluster description USING sharding key
```

Query works only for Replicated tables and for Distributed tables that are looking at Replicated tables.

When executing, query first checks correctness of query, sufficiency of free space on nodes and writes to ZooKeeper at some path a task to to. Next work is done asynchronously.

For using resharding, you must specify path in ZooKeeper for task queue in configuration file:

```
<resharding>
      <task_queue_path>/clickhouse/task_queue</task_queue_path>
</resharding>
```

When running `ALTER TABLE t RESHARD` query, node in ZooKeeper is created if not exists.

Cluster description is list of shards with weights to distribute the data. Shard is specified as address of table in ZooKeeper. Example: /clickhouse/tables/01-03/hits Relative weight of shard (optional, default is 1) could be specified after WEIGHT keyword. Example:

```
ALTER TABLE merge.hits
RESHARD PARTITION 201501
TO
      '/clickhouse/tables/01-01/hits' WEIGHT 1,
      '/clickhouse/tables/01-02/hits' WEIGHT 2,
      '/clickhouse/tables/01-03/hits' WEIGHT 1,
      '/clickhouse/tables/01-04/hits' WEIGHT 1
USING UserID
```

Sharding key (`UserID` in example) has same semantic as for Distributed tables. You could specify `rand()` as sharding key for random distribution of data.

When query is run, it checks:

- identity of table structure on all shards.
- availability of free space on local node in amount of partition size in bytes, with additional 10% reserve.

- availability of free space on all replicas of all specified shards, except local replica, if exists, in amount of patition size times ratio of shard weight to total weight of all shards, with additional 10% reserve.

Next, asynchronous processing of query is of following steps:

1. **Split patition to parts on local node.** It merges all parts forming a partition and in the same time, splits them to several, according sharding key. Result is placed to /reshard directory in table data directory. Source parts doesn't modified and all process doesn't intervent table working data set.
2. Copying all parts to remote nodes (to each replica of corresponding shard).
3. **Execution of queries `ALTER TABLE t DROP PARTITION` on local node and `ALTER TABLE t ATTACH PARTITI`** Note: this operation is not atomic. There are time point when user could see absence of data.

   When `COPY` keyword is specified, source data is not removed. It is suitable for copying data from one cluster to another with changing sharding scheme in same time.
4. Removing temporary data from local node.

When having multiple resharding queries, their tasks will be done sequentially.

Query in example is to reshard single partition. If you don't specify partition in query, then tasks to reshard all partitions will be created. Example:

```
ALTER TABLE merge.hits
RESHARD
TO ...
```

When resharding Distributed tables, each shard will be resharded (corresponding query is sent to each shard).

You could reshard Distributed table to itself or to another table.

Resharding is intended for "old" data: in case when during job, resharded partition was modified, task for that partition will be cancelled.

On each server, resharding is done in single thread. It is doing that way to not disturb normal query processing.

As of June 2016, resharding is in "beta" state: it was tested only for small data sets - up to 5 TB.

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.

SummingMergeTree

This engine differs from `MergeTree` in that it totals data while merging.

```
SummingMergeTree(EventDate, (OrderID, EventDate, BannerID, ...), 8192)
```

The columns to total are implicit. When merging, all rows with the same primary key value (in the example, OrderId, EventDate, BannerID, ...) have their values totaled in numeric columns that are not part of the primary key.

```
SummingMergeTree(EventDate, (OrderID, EventDate, BannerID, ...), 8192, (Shows, Clicks, Cost, ...))
```

The columns to total are set explicitly (the last parameter - Shows, Clicks, Cost, ...). When merging, all rows with the same primary key value have their values totaled in the specified columns. The specified columns also must be numeric and must not be part of the primary key.

If the values were null in all of these columns, the row is deleted. (The exception is cases when the data part would not have any rows left in it.)

For the other rows that are not part of the primary key, the first value that occurs is selected when merging.

Summation is not performed for a read operation. If it is necessary, write the appropriate GROUP BY.

In addition, a table can have nested data structures that are processed in a special way. If the name of a nested table ends in 'Map' and it contains at least two columns that meet the following criteria: * for the first table, numeric ((U)IntN, Date, DateTime), we'll refer to it as 'key' * for other tables, arithmetic ((U)IntN, Float32/64), we'll refer to it as '(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)]
```

For nested data structures, you don't need to specify the columns as a list of columns for totaling.

This table engine is not particularly useful. Remember that when saving just pre-aggregated data, you lose some of the system's advantages.

TinyLog

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.

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).

Глава 6

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. System tables are located in the 'system' database.

system.asynchronous_metrics

Like system.events, but show info about currently executing events or consuming resources. Example: The number of currently executing SELECT queries; memory consumption of the system.

system.clusters

Contains information about clusters available in the config file and the servers in them. Columns:

```
cluster String      - Cluster name.
shard_num UInt32    - Number of a shard in the cluster, starting from 1.
shard_weight UInt32 - Relative weight of a shard when writing data.
replica_num UInt32  - Number of a replica in the shard, starting from 1.
host_name String    - Host name as specified in the config.
host_address String - Host's IP address obtained from DNS.
port UInt16         - The port used to access the server.
user String         - The username to use for connecting to the server.
```

system.columns

Contains information about the columns in all tables. You can use this table to get information similar to `DESCRIBE TABLE`, but for multiple tables at once.

```
database String           - Name of the database the table is located in.
table String              - Table name.
name String               - Column name.
type String               - Column type.
default_type 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.
```

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.dictionaries

Contains information about external dictionaries.

Columns:

```
name String                   - Dictionary name.
type String                   - Dictionary type: Flat, Hashed, Cache.
origin String                 - Path to the config file where the dictionary is described.
attribute.names Array(String) - Array of attribute names provided by the dictionary.
attribute.types Array(String) - Corresponding array of attribute types provided by the dictionary.
has_hierarchy UInt8           - Whether the dictionary is hierarchical.
bytes_allocated UInt64        - The amount of RAM used by the dictionary.
hit_rate Float64              - For cache dictionaries, the percent of usage for which the value␣
↪was in the cache.
element_count UInt64          - The number of items stored in the dictionary.
load_factor Float64           - The filled percentage of the dictionary (for a hashed dictionary,␣
↪it is the filled percentage of the hash table).
creation_time DateTime        - Time spent for the creation or last successful reload of the␣
↪dictionary.
last_exception String         - Text of an error that occurred 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           - Function name.
is_aggregate UInt8    - Whether it is an aggregate function.
```

system.merges

Contains information about merges currently in process for tables in the MergeTree family.

Columns:

```
database String                    - Name of the database the table is located in.
table String                       - Name of the table.
elapsed Float64                    - Time in seconds since the merge started.
progress Float64                   - Percent of progress made, from 0 to 1.
num_parts UInt64                   - Number of parts to merge.
result_part_name String            - Name of the part that will be formed as the result of the␣
↪merge.
total_size_bytes_compressed UInt64 - Total size of compressed data in the parts being merged.
total_size_marks UInt64            - Total number of marks in the parts being merged.
bytes_read_uncompressed UInt64     - Amount of bytes read, decompressed.
rows_read UInt64                   - Number of rows read.
bytes_written_uncompressed UInt64  - Amount of bytes written, uncompressed.
rows_written UInt64                - Number of 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 a table in the MergeTree family.

Columns:

```
database String            - Name of the database where the table that this part belongs to is␣
↪located.
table String               - Name of the table that this part belongs to.
engine String              - Name of the table engine, without parameters.
partition String           - Name of the partition, in the format YYYYMM.
name String                - Name of the part.
replicated UInt8           - Whether the part belongs to replicated data.
active UInt8               - Whether the part is used in a table, or is no longer needed and will␣
↪be deleted soon. Inactive parts remain after merging.
marks UInt64               - Number of marks - multiply by the index granularity (usually 8192) to␣
↪get the approximate number of rows in the part.
bytes UInt64               - Number of bytes when compressed.
modification_time DateTime - Time the directory with the part was modified. Usually corresponds to␣
↪the part's creation time.
remove_time DateTime       - For inactive parts only - the time when the part became inactive.
refcount UInt32            - The number of places where the part is used. A value greater than 2␣
↪indicates that this part participates in queries or merges.
```

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.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.
Only one replica can be the leader at a time. 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.

is_readonly:        Whether the replica is in read-only mode.
This mode is turned on if the config doesn't have sections with ZK, if an unknown error occurred␣
↪when reinitializing sessions in ZK, and during session reinitialization in ZK.

is_session_expired: Whether the session with ZK has expired.
Basically the same as 'is_readonly'.

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.
A part is put in the verification queue if there is suspicion that it might be damaged.
```

```
zookeeper_path:     Path to table data in ZK.
replica_name:       Replica name in ZK. Different replicas of the same table have different names.
replica_path:      Path to replica data in ZK. The same as concatenating 'zookeeper_path/replicas/
↪replica_path'.

columns_version:    Version number of the table structure. Indicates how many times ALTER was␣
↪performed. If replicas have different versions, it means some replicas haven't made all of the␣
↪ALTERs yet.

queue_size:         Size of the queue for operations waiting to be performed. Operations include␣
↪inserting blocks of data, merges, and certain other actions. It usually coincides with 'future_
↪parts'.

inserts_in_queue:   Number of inserts of blocks of data that need to be made. Insertions are␣
↪usually replicated fairly quickly. If this number is large, it means something is wrong.

merges_in_queue:    The number of merges waiting to be made. Sometimes merges are lengthy, so this␣
↪value may be greater than one for a long time.

The next 4 columns have a non-zero value only where there is an active session with ZK.

log_max_index:      Maximum entry number in the log of general activity.
log_pointer:        Maximum entry number from the log of general activity that the replica copied␣
↪to its queue for execution, plus one.
If log_pointer is much smaller than log_max_index, something is wrong.

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 ZK (i.e., the␣
↪number of functioning replicas).к
```

If you request all the columns, the table may work a bit slowly, since several reads from ZK 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

This table contains the String columns 'database', 'name', and 'engine' and DateTime column metadata_modification_time. Each table that the server knows about is entered in the 'system.tables' table. There is an issue: table engines are specified without parameters. This system table is used for implementing SHOW TABLES queries.

system.zookeeper

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.

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          - Name of the node.
path String          - Path to the node.
value String         - Value of the node.
dataLength Int32     - Size of the value.
numChildren Int32    - Number of children.
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 added or removed children.
ctime DateTime       - Time of node creation.
mtime DateTime       - Time of the last node modification.
version Int32        - Node version - the number of times the node was changed.
cversion Int32       - Number of added or removed children.
aversion Int32       - Number of changes to 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
```

Глава 7

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.

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.

remote

`remote('addresses_expr', db, table[, 'user'[, 'password']])`

or

`remote('addresses_expr', db.table[, 'user'[, 'password']])`

- Allows accessing a remote server without creating a Distributed table.

`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 the 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).

Note: As an exception, when specifying an IPv6 address, the port is required.

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, the query goes to all the specified addresses (like to shards with different data) and uses distributed processing.

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 fragments 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. The replicas are evaluated 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.

The username can be omitted. In this case, the 'default' username is used. The password can be omitted. In this case, an empty password is used.

Глава 8

Formats

The format determines how data is given (written by server as output) to you after SELECTs, and how it is accepted (read by server as input) for INSERTs.

BlockTabSeparated

Data is not written by row, but by column and block. Each block consists of parts of columns, each of which is written on a separate line. The values are tab-separated. The last value in a column part is followed by a line break instead of a tab. Blocks are separated by a double line break. The rest of the rules are the same as in the TabSeparated format. This format is only appropriate for outputting a query result, not for parsing.

CSV

Comma separated values (RFC).

String values are output in double quotes. Double quote inside a string is output as two consecutive double quotes. That's all escaping rules. Date and DateTime values are output in double quotes. Numbers are output without quotes. Fields are delimited by commas. Rows are delimited by unix newlines (LF). Arrays are output in following way: first, array are serialized to String (as in TabSeparated or Values formats), and then the String value are output in double quotes. Tuples are narrowed and serialized as separate columns.

During parsing, values could be enclosed or not enclosed in quotes. Supported both single and double quotes. In particular, Strings could be represented without quotes - in that case, they are parsed up to comma or newline (CR or LF). Contrary to RFC, in case of parsing strings without quotes, leading and trailing spaces and tabs are ignored. As line delimiter, both Unix (LF), Windows (CR LF) or Mac OS Classic (LF CR) variants are supported.

CSV format supports output of totals and extremes similar to TabSeparated format.

CSVWithNames

Also contains header, 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"
                }
        ],

        "data":
        [
                {
                        "SearchPhrase": "",
                        "c": "8267016"
                },
                {
                        "SearchPhrase": "интерьер ванной комнаты",
                        "c": "2166"
                },
                {
                        "SearchPhrase": "яндекс",
                        "c": "1655"
                },
                {
                        "SearchPhrase": "весна 2014 мода",
                        "c": "1549"
                },
                {
                        "SearchPhrase": "фриформ фото",
                        "c": "1480"
                }
        ],

        "totals":
        {
                "SearchPhrase": "",
                "c": "8873898"
        },
        "extremes":
        {
                "min":
                {
                        "SearchPhrase": "",
                        "c": "1480"
                },
                "max":
                {
                        "SearchPhrase": "",
                        "c": "8267016"
                }
        },

        "rows": 5,

        "rows_before_limit_at_least": 141137
}
```

JSON is compatible with JavaScript. For this purpose, certain symbols are additionally escaped: the forward slash `/` is escaped as `\/`; alternative line breaks `U+2028` and `U+2029`, which don't work in some browsers, are escaped as uXXXX-sequences. ASCII control characters are escaped: backspace, form feed, line feed, carriage return, and horizontal tab as `\b`, `\f`, `\n`, `\r`, and `\t` respectively, along with the rest of the bytes from the range 00-1F using uXXXX-sequences. Invalid UTF-8 sequences are changed to the replacement character and, thus, the output text will consist of valid UTF-8 sequences. UInt64 and Int64 numbers are output in double quotes for compatibility with JavaScript.

`rows` - The total number of output rows.

`rows_before_limit_at_least` - The minimal number of rows there would have been without a 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, not for parsing. See JSONEachRow format for INSERT queries.

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"],
                ["bath interiors", "2166"],
                ["yandex", "1655"],
                ["spring 2014 fashion", "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, not for parsing. See `JSONEachRow` format for INSERT queries.

JSONEachRow

If put in SELECT query, displays data in newline delimited JSON (JSON objects separated by \n character) format. If put in INSERT query, expects this kind of data as input.

```
{"SearchPhrase":"","count()":"8267016"}
{"SearchPhrase":"bathroom interior","count()":"2166"}
{"SearchPhrase":"yandex","count()":"1655"}
{"SearchPhrase":"spring 2014 fashion","count()":"1549"}
{"SearchPhrase":"free-form photo","count()":"1480"}
{"SearchPhrase":"Angelina Jolie","count()":"1245"}
{"SearchPhrase":"omsk","count()":"1112"}
{"SearchPhrase":"photos of dog breeds","count()":"1091"}
{"SearchPhrase":"curtain design","count()":"1064"}
{"SearchPhrase":"baku","count()":"1000"}
```

Unlike JSON format, there are no replacements of invalid UTF-8 sequences. There can be arbitrary amount of bytes in a line. This is done in order to avoid data loss during formatting. Values are displayed analogous to JSON format.

In INSERT queries JSON data can be supplied with arbitrary order of columns (JSON key-value pairs). It is also possible to omit values in which case the default value of the column is inserted. N.B. when using JSONEachRow format, complex default values are not supported, so when omitting a column its value will be zeros or empty string depending on its type.

Space characters between JSON objects are skipped. Between objects there can be a comma which is ignored. Newline character is not a mandatory separator for objects.

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 outputting a query result, not for parsing.

Pretty

Writes 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). 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, not for parsing.

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 * FROM system.events FORMAT PrettyCompactNoEscapes'"
```

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

PrettyCompactNoEscapes

The same.

PrettySpaceNoEscapes

The same.

PrettySpace

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

RowBinary

Writes 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.

Numbers is written in little endian, fixed width. For example, UInt64 takes 8 bytes. DateTime is written as UInt32 with unix timestamp value. Date is written as UInt16 with number of days since 1970-01-01 in value. String is written as length in varint (unsigned LEB128) format and then bytes of string. FixedString is written as just its bytes. Array is written as length in varint (unsigned LEB128) format and then all elements, contiguously

TabSeparated

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

Numbers are written in decimal form. Numbers may contain an extra "+" symbol at the beginning (but it is not recorded during an output). Non-negative numbers can't contain the negative sign. When parsing, 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 formatted 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, a result is not necessarily the nearest machine-representable number.

Dates are formatted in YYYY-MM-DD format and parsed in the same format, but with any characters as separators. DateTimes are formatted 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 DateTimes, 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 parsing 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 DateTime 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 parsed and formatted with backslash-escaped special characters. The following escape sequences are used while formatting: `\b`, `\f`, `\r`, `\n`, `\t`, `\0`, `\'`, and `\\`. For parsing, also supported a, v and xHH (hex escape sequence) and any sequences of the type c where c is any character (these sequences are converted to c). This means that parsing supports formats where a line break can be written as n or as and a line break. For example, the string 'Hello world' with a line break between the words instead of a space can be retrieved in any of the following variations:

```
Hello\nworld

Hello\
world
```

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

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

Minimum set of symbols that you must escape in TabSeparated format is tab, newline (LF) and backslash.

Arrays are formatted 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 formatted in single quotes with the same escaping rules as above.

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
```

It's also available as `TSV`.

TabSeparatedRaw

Differs from the `TabSeparated` format in that the rows are formatted without escaping. This format is only appropriate for outputting a query result, but not for parsing data to insert into a table.

It's also available as `TSVRaw`.

TabSeparatedWithNames

Differs from the TabSeparated format in that the column names are output in the first row. For parsing, the first row is completely ignored. You can't use column names to determine their position or to check their correctness. (Support for using header while parsing could be added in future.)

It's also available as `TSVWithNames`.

TabSeparatedWithNamesAndTypes

Differs from the `TabSeparated` format in that the column names are output to the first row, while the column types are in the second row. For parsing, the first and second rows are completely ignored.

It's also available as `TSVWithNamesAndTypes`.

TSKV

Similar to TabSeparated, but displays data in name=value format. Names are displayed just as in TabSeparated. Additionally, a = symbol is displayed.

```
SearchPhrase=   count()=8267016
SearchPhrase=bathroom interior    count()=2166
SearchPhrase=yandex     count()=1655
SearchPhrase=spring 2014 fashion    count()=1549
SearchPhrase=free-form photo       count()=1480
SearchPhrase=Angelina Jolie    count()=1245
SearchPhrase=omsk       count()=1112
SearchPhrase=photos of dog breeds    count()=1091
SearchPhrase=curtain design        count()=1064
SearchPhrase=baku       count()=1000
```

In case of many small columns this format is obviously not effective and there usually is no reason to use it. This format is supported because it is used for some cases in Yandex.

Format is supported both for input and output. In INSERT queries data can be supplied with arbitrary order of columns. It is also possible to omit values in which case the default value of the column is inserted. N.B. when using TSKV format, complex default values are not supported, so when omitting a column its value will be zeros or empty string depending on its type.

Values

Prints every row in parentheses. Rows are separated by commas. There is no comma after the last row. The values inside the parentheses 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 same as in 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).

Minimum set of symbols that you must escape in Values format is single quote and backslash.

This is the format that is used in `INSERT INTO t VALUES ...` But you can also use it for query result.

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. This format is only appropriate for outputting a query result, not for parsing.

XML

XML format is supported only for displaying data, not for INSERTS. 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</SearchPhrase>
                        <field>2166</field>
                </row>
                <row>
                        <SearchPhrase>yandex>
                        <field>1655</field>
                </row>
                <row>
                        <SearchPhrase>spring 2014 fashion</SearchPhrase>
                        <field>1549</field>
                </row>
                <row>
                        <SearchPhrase>free-form photo</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 design</SearchPhrase>
                        <field>1064</field>
                </row>
                <row>
                        <SearchPhrase>baku</SearchPhrase>
                        <field>1000</field>
                </row>
        </data>
        <rows>10</rows>
        <rows_before_limit_at_least>141137</rows_before_limit_at_least>
</result>
```

If name of a column contains some unacceptable character, field is used as a name. In other aspects XML uses JSON structure. As in case of JSON, invalid UTF-8 sequences are replaced by replacement character so displayed text will only contain valid UTF-8 sequences.

In string values `<` and `&` are displayed as `&lt;` and `&amp;`.

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

Глава 9

Data types

Array(T)

Array of T-type items. The T type can be any type, including an array.

We don't recommend using multidimensional arrays, because they are not well supported (for example, you can't store multidimensional arrays in tables with engines from MergeTree family).

Boolean

There is no separate type for boolean values. For them, the type UInt8 is used, in which only the values 0 and 1 are used.

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 2038, but it may be expanded to 2106). 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.

Note that by default the client adopts the server time zone at the beginning of the session. You can change this behaviour with the –use_client_time_zone command line switch.

Supports only those time zones that never had the time differ from UTC for a partial number of hours (without leap seconds) over the entire time range you will be working with.

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.

Enum

Enum8 or Enum16. A set of enumerated string values that are stored as Int8 or Int16.

Example:

```
Enum8('hello' = 1, 'world' = 2)
```

This data type has two possible values - 'hello' and 'world'.

The numeric values must be within -128..127 for `Enum8` and -32768..32767 for `Enum16`. Every member of the enum must also have different numbers. The empty string is a valid value. The numbers do not need to be sequential and can be in any order. The order does not matter.

In memory, the data is stored in the same way as the numeric types `Int8` and `Int16`. 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 format, the stored number is looked up and the corresponding string is written out. An exception will be thrown if the number does not correspond to a known value. In binary format, the information is saved in the same way as `Int8` and `Int16`. The implicit default value for an Enum is the value having the smallest numeric value.

In ORDER BY, GROUP BY, IN, DISTINCT, etc. Enums behave like the numeric value. e.g. they will be sorted by the numeric value in an `ORDER BY`. Equality and comparison operators behave like they do on the underlying numeric value.

Enum values cannot be compared to numbers, they must be compared to a 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.

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 toString function can be used to convert the Enum to 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.

It is possible to add new members to the `Enum` using `ALTER`. If the only change is to the set of values, the operation will be almost instant. It is also possible to remove members of the Enum using ALTER. Removing members is only safe 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`.

FixedString(N)

A fixed-length string of N bytes (not characters or code points). N must be a strictly positive natural number. When server reads a string (as an input passed in INSERT query, for example) that contains fewer bytes, the string is padded to N bytes by appending null bytes at the right. When server reads a string that contains more bytes, an error message is returned. When server writes a string (as an output of SELECT query, for example), null bytes are not trimmed off of the end of the string, but are output. Note that this behavior differs from MySQL behavior for the CHAR type (where strings are padded with spaces, and the spaces are removed for output).

Fewer functions can work with the FixedString(N) type than with String, so it is less convenient to use.

Float32, Float64

Floating-point numbers are just like 'float' and 'double' in the C language. In contrast to standard SQL, floating-point numbers support 'inf', '-inf', and even 'nan's. See the notes on sorting nans in "ORDER BY clause". We do not recommend storing floating-point numbers in tables.

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

Fixed-length integers, with or without a sign.

Int ranges

Тип	From	To
Int8	-128	127
Int16	-32768	32767
Int32	-2147483648	2147483647
Int64	-9223372036854775808	9223372036854775807

Uint ranges

Тип	From	To
UInt8	0	255
UInt16	0	65535
UInt32	0	4294967295
UInt64	0	18446744073709551615

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.

Tuple(T1, T2, ...)

Tuples can't be written to 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 "IN operators" and "Higher order functions".

Tuples can be output as the result of running 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).

Nested data structures

AggregateFunction(name, types_of_arguments...)

The intermediate state of an aggregate function. To get it, use aggregate functions with the '-State' suffix. For more information, see "AggregatingMergeTree".

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. Nested structure columns with array type are equivalent to multidimensional arrays and thus their support is limited (storing such columns in tables with engines from MergeTree family is not supported).

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.

Глава 10

Operators

All operators are transformed to the corresponding functions at the query parsing stage, in accordance with their precedence and associativity.

Access operators

`a[N]` - Access to an array element, arrayElement(a, N) function.

`a.N` - Access to a tuple element, tupleElement(a, N) function.

Numeric negation operator

`-a` - negate(a) function

Multiplication and division operators

`a * b` - multiply(a, b) function

`a / b` - divide(a, b) function

`a % b` - modulo(a, b) function

Addition and subtraction operators

`a + b` - plus(a, b) function

`a - b` - minus(a, b) function

Comparison operators

`a = b` - equals(a, b) function

`a == b` - equals(a, b) function

`a != b` - notEquals(a, b) function

`a <> b` - notEquals(a, b) function

`a <= b` - lessOrEquals(a, b) function

`a >= b` - greaterOrEquals(a, b) function

`a < b` - less(a, b) function

`a > b` - greater(a, b) function

`a LIKE s` - like(a, b) function

`a NOT LIKE s` - notLike(a, b) function

`a BETWEEN b AND c` - equivalent to a >= b AND a <= c

Operators for working with data sets

See the section "IN operators".

`a IN ...` - in(a, b) function

`a NOT IN ...` - notIn(a, b) function

`a GLOBAL IN ...` - globalIn(a, b) function

`a GLOBAL NOT IN ...` - globalNotIn(a, b) function

Logical negation operator

`NOT a` - `not(a)` function

Logical "AND" operator

`a AND b` - function `and(a, b)`

Logical "OR" operator

`a OR b` - function `or(a, b)`

Conditional operator

`a ? b : c` - function `if(a, b, c)`

Conditional expression

```
CASE [x]
    WHEN a THEN b
    [WHEN ... THEN ...]
    ELSE c
END
```

If x is given - transform(x, [a, ...], [b, ...], c). Otherwise, multiIf(a, b, ..., c).

String concatenation operator

`s1 || s2` - concat(s1, s2) function

Lambda creation operator

`x -> expr` - lambda(x, expr) function

The following operators do not have a priority, since they are brackets:

Array creation operator

`[x1, ...]` - array(x1, ...) function

Tuple creation operator

`(x1, x2, ...)` - 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 > 3 > 2'` results 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.

Глава 11

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.

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 whole numbers with a date or date and time. In the case of a date, adding a whole number 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 whole 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). When dividing by zero or when dividing a minimal negative number by minus one, an exception is thrown.

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.

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 as argument and returns an array of one element 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.

arrayElement(arr, n), operator arr[n]

Get the element with the index 'n' from the array 'arr'. 'n' should 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 goes beyond the array bounds, a default value is returned (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. 'elem' must be a constant.

indexOf(arr, x)

Returns the index of the 'x' element (starting from 1) if it is in the array, or 0 if it is not.

countEqual(arr, x)

Returns the number of elements in the array equal to 'x'. Equivalent to `arrayCount(elem -> elem = x, arr)`.

arrayEnumerate(arr)

Returns the array `[1, 2, 3, ..., length(arr)]`

This function is normally used together 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.

arrayUniq(arr, …)

If a single array is passed, returns a number of unique elements in that array. If multiple arrays of the same size are passed as arguments to the function, returns a number of unique tuples of elements in the same positions in all the arrays.

If you need an array of the unique elements, you can use `arrayReduce('groupUniqArray', arr)`.

arrayJoin(arr)

A special function. See the section "arrayJoin 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

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] |
--------------
```

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)

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: before version 1.1.54134 signed and unsigned numbers were compared the same way as in C++. That is, you could got an incorrect result in such cases: SELECT 9223372036854775807 > -1. From version 1.1.54134, the behavior has changed and numbers are compared mathematically correct.

equals, a = b and a == b operator

notEquals, a != b and a <> b operator

less, < operator

greater, > operator

lessOrEquals, <= operator

greaterOrEquals, >= operator

Conditional functions

if(cond, then, else), ternary operator cond ? then : else

Returns ‘then’ if ‘cond != 0’, or ‘else’ if ‘cond = 0’. ‘cond’ must be UInt 8, and ‘then’ and ‘else’ must be a type that has the smallest common type.

Functions for working with dates and times

Time Zone Support

All functions for working with the date and time for which this makes sense, can take a second, optional argument - the time zone name. Example: Asia / Yekaterinburg. In this case, they do not use the local time zone (the default), but the specified 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 are supported, different from UTC for an integer number of hours.

toYear

Converts a date or date with time to a UInt16 number containing the year number (AD).

toMonth

Converts a date or date with time to a UInt8 number containing the month number (1-12).

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 once 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.

toStartOfDay

Rounds down a date with time to the start of the day.

toMonday

Rounds down a date or date with time to the nearest Monday. Returns the date.

toStartOfMonth

Rounds down a date or date with time to the first day of the month. 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.

toStartOfYear

Rounds down a date or date with time to the first day of the year. Returns the date.

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 5 minute (00:00, 00:05, 00:10...).

toStartOfHour

Rounds down a date with time to the start of the hour.

toTime

Converts a date with time to some 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.

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.

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 counter shows a single user's consecutive pageviews that differ in time by strictly more than this amount. This means that tuples (the counter number, user ID, and time slot) can be used to search for pageviews that are included in the corresponding session.

timeSlots(StartTime, Duration)

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 half hour. For example, `timeSlots(toDateTime('2012-01-01 12:20:00'), toUInt32(600)) = [toDateTime('2012-01-01 12:00:00'), toDateTime('2012-01-01 12:30:00')]`. This is necessary for searching for page views in the corresponding session.

Encoding functions

hex

Accepts a string, number, date, or date with time. Returns a string containing the argument's hexadecimal representation. Uses uppercase letters A-F. Doesn't 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'`. Dates are encoded as the number of days since the beginning of the Unix Epoch. Dates with times are 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 doesn't 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 functions 'reverse' and 'reinterpretAsType'

UUIDStringToNum(str)

Accepts a string containing the UUID in the text format (`123e4567-e89b-12d3-a456-426655440000`). Returns a binary representation of the UUID in `FixedString(16)`.

UUIDNumToString(str)

Accepts a FixedString(16) value containing the UUID in the binary format. Returns a readable string containing the UUID in the 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 external dictionaries

For more information, see the section "External dictionaries".

dictGetUInt8, dictGetUInt16, dictGetUInt32, dictGetUInt64

dictGetInt8, dictGetInt16, dictGetInt32, dictGetInt64

dictGetFloat32, dictGetFloat64

dictGetDate, dictGetDateTime

dictGetString

`dictGetT('dict_name', 'attr_name', id)` Gets the value of the 'attr_name' attribute from the 'dict_name' dictionary by the 'id' key. 'dict_name' and 'attr_name' are constant strings. 'id' must be UInt64. If the 'id' key is not in the dictionary, it returns the default value set in the dictionary definition.

dictGetTOrDefault

`dictGetT('dict_name', 'attr_name', id, default)` Similar to the functions dictGetT, but the default value is taken from the last argument of the function.

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 the presence of a key in the dictionary. Returns a value of type UInt8, equal to 0, if there is no key and 1 if there is a key.

Hash functions

Hash functions can be used for deterministic pseudo-random shuffling of elements.

halfMD5

Calculates the MD5 from a string. Then it takes the first 8 bytes of the hash and interprets them as UInt64 in big endian. Accepts a String-type argument. Returns UInt64. This function works fairly slowly (5 million short strings per second per processor core). If you don't need MD5 in particular, use the 'sipHash64' function instead.

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 need the same result as gives 'md5sum' utility, write `lower(hex(MD5(s)))`.

sipHash64

Calculates SipHash from a string. Accepts a String-type argument. Returns UInt64. SipHash is a cryptographic hash function. It works at least three times faster than MD5. For more information, see https://131002.net/siphash/

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 bits.

cityHash64

Calculates CityHash64 from a string or a similar hash function for any number of any type of arguments. For String-type arguments, CityHash is used. This is a fast non-cryptographic hash function for strings with decent quality. For other types of arguments, a decent implementation-specific fast non-cryptographic hash function is used. If multiple arguments are passed, the function is calculated using the same rules and chain combinations using the CityHash combinator. For example, you can compute the checksum of an 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.

`URL Hash(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.

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 - 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 'arr' for which 'func' returns something other than 0. If 'func' is not specified, it returns the number of non-zero items 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.

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 columns from a tuple. 'N' is the column index, starting from 1. 'N' must be a constant. 'N' must be a strict postive integer no greater than the size of the tuple. There is no cost to execute the function.

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: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.

Functions for working with JSON.

In Yandex.Metrica, JSON is passed by users as session parameters. There are several 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 not to do anything.

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. 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') = ''
```

Currently, there is no support for code points not from the basic multilingual plane written in the format `\uXXXX\uYYYY` (they are converted to CESU-8 instead of UTF-8).

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

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()

Accepts zero arguments and returns a Float64 number close to the e number.

pi()

Accepts zero arguments and returns a Float64 number close to π.

exp(x)

Accepts a numeric argument and returns a Float64 number close to the exponent of the argument.

log(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 2x.

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 10x.

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 `erf(x / σ2) -` 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 - 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)

x to the power y.

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.

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.

toTypeName(x)

Gets the type name. Returns a string containing the type name of the passed argument.

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(...)

A function that accepts any arguments and always returns 0. However, the argument is still calculated. 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.

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 String columns - database name, table name and column name. Returns constant UInt8 value, equal to 1 if column exists, otherwise 0. If hostname is specified, the check will proceed on remote host. If table doesn't exist than exception is thrown. For elements of nested data structure function checks existence of column. For nested data structure 0 is returned.

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. `min, max` - Integer constants. The value must fit in Int64. `width` - Constant, positive number, may be a fraction.

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
```

```
-h----c--bar--------
|  0 | 292907 |          |
|  1 | 180563 |              |
|  2 | 114861 |                |
|  3 |  85069 |                 |
|  4 |  68543 |                 |
|  5 |  78116 |                 |
|  6 | 113474 |                |
|  7 | 170678 |              |
|  8 | 278380 |          |
|  9 | 391053 |       |
| 10 | 457681 |    |
| 11 | 493667 |   |
| 12 | 509641 |   |
| 13 | 522947 |  |
| 14 | 539954 |  |
| 15 | 528460 |  |
| 16 | 539201 |  |
| 17 | 523539 |  |
| 18 | 506467 |   |
| 19 | 520915 |  |
| 20 | 521665 |  |
| 21 | 542078 |  |
| 22 | 493642 |   |
| 23 | 400397 |      |
----------------
```

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` - Constant. Which value to use if 'x' is not equal to one of the values in 'from' `'array_from'` and `'array_to'` are 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], ['Яндекс', 'Google'], 'Остальные') AS title,
    count() AS c
FROM test.hits
WHERE SearchEngineID != 0
GROUP BY title
ORDER BY c DESC
```

```
-title------c-
| Яндекс    | 498635 |
| Google    | 229872 |
| Остальные | 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', 'ввв.яндекс.рф
↪', '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 |
| ввв.яндекс.рф  |  105668 |
| .ru      |  100355 |
| .ru   |   65040 |
| news.yandex.ru |   64515 |
| .net      |   59141 |
| example.com    |   57316 |
-------------
```

formatReadableSize(x)

Gets a size (number of bytes). Returns a string that contains rounded size with the suffix (KiB, MiB etc.).

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 least element of a and b.

greatest(a, b)

Returns the greatest element of a and b

uptime()

Returns server's uptime in seconds.

version()

Returns server's version as a string.

rowNumberInAllBlocks()

Returns an incremental row number within all blocks that were processed by this function.

runningDifference(x)

Calculates the difference between consecutive values in the data block. Result of the function depends on the order of the data in the blocks.

It works only inside of the each processed block of data. Data splitting in the blocks is not explicitly controlled by the user. If you specify `ORDER BY` in subquery and call runningDifference outside of it, you could get an expected result.

Example:

```sql
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 |
--------------------
```

MACNumToString(num)

Takes a UInt64 number. Interprets it as an 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 reverse function of MACNumToString. If the MAC address has an invalid format, it returns 0.

MACStringToOUI(s)

Takes MAC address in the format AA:BB:CC:DD:EE:FF (colon-separated numbers in hexadecimal form). Returns first three octets as UInt64 number. If the MAC address has an invalid format, it returns 0.

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.

Rounding functions

floor(x[, N])

Returns a rounder number that is less than or equal to 'x'. A round number is a multiple of 1 / 10N, or the nearest number of the appropriate data type `if 1 / 10N` 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])

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])

Returns the round number nearest to 'num', which may be less than, greater than, or equal to 'x'. If 'x' is exactly in the middle between the nearest round numbers, one of them is returned (implementation-specific). The number '-0.' may or may not be considered round (implementation-specific). In every other way, this function is the same as 'floor' and 'ceil' described above.

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 numbers from the set: 18, 25, 35, 45, 55. This function is specific to Yandex.Metrica and used for implementing the report on user age.

Functions for splitting and merging strings and arrays

splitByChar(separator, s)

Splits a string into substrings, using 'separator' as the 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 strings from the array elements, using 'separator' as the separator. 'separator' is a string constant, an optional parameter. By default it is an empty string. Returns a string.

alphaTokens(s)

Selects substrings of consecutive bytes from the range a-z and A-Z. Returns an array of selected substrings.

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.

lower

Converts ASCII Latin symbols in a string to lowercase.

upper

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 length of UTF-8 sequence is different for upper and lower case of code point, then result for that code point could be incorrect. If value contains invalid UTF-8, the behavior is unspecified.

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 length of UTF-8 sequence is different for upper and lower case of code point, then result for that code point could be incorrect. If value contains invalid UTF-8, the behavior is unspecified.

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).

concat(s1, s2, …)

Concatenates strings from the function arguments, without a separator.

substring(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 a string with the data s (encoded as from charset) that was converted to the to charset.

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)

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 0-9. The substitution 0 includes the entire regular expression. The substitutions 1-9 include the subpattern corresponding to the number. In order to specify the symbol in a pattern, you must use a symbol to escape it. 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. Copy the 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!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,,  WWoorrlldd!! |
--------------
```

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! |
-----------
```

Functions for searching strings

The search is case-sensitive in all these functions. The search substring or regular expression must be a constant in all these functions.

position(haystack, needle)

Searches for the 'needle' substring in the 'haystack' string. Returns the position (in bytes) of the found substring, starting from 1, or returns 0 if the substring was not found. There's also positionCaseInsensitive function.

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). There's also positionCaseInsensitiveUTF8 function.

match(haystack, pattern)

Checks whether the string matches the 'pattern' regular expression. The regular expression is re2. 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.

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), оператор haystack LIKE pattern

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), оператор haystack NOT LIKE pattern

The same thing as 'like', but negative.

Type conversion functions

toUInt8, toUInt16, toUInt32, toUInt64

toInt8, toInt16, toInt32, toInt64

toFloat32, toFloat64

toUInt8OrZero, toUInt16OrZero, toUInt32OrZero, toUInt64OrZero, toInt8OrZero, toInt16OrZero, toInt32OrZero, toInt64OrZero, toFloat32OrZero, toFloat64OrZero

toDate, toDateTime

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.

Formats of date and date with time for 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 type numbers 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++.

To do transformations on DateTime in given time zone, pass second argument with time zone name:

```sql
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 |
----------------------
```

To format DateTime in given time zone:

```
toString(now(), 'Asia/Yekaterinburg')
```

To get unix timestamp for string with datetime in specified time zone:

```
toUnixTimestamp('2000-01-01 00:00:00', 'Asia/Yekaterinburg')
```

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 a String that is cut to a first null byte occurrence.

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

reinterpretAsInt8, reinterpretAsInt16, reinterpretAsInt32, reinterpretAsInt64

reinterpretAsFloat32, reinterpretAsFloat64

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.

CAST(x, t)

Casts 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 |
-----------------------------------------------------
```

Casting to FixedString(N) works only for String and FixedString(N).

Functions for working with URLs

All these functions don't follow the RFC. They are maximally simplified for improved performance.

Functions tat extract part of the URL

If there isn't anything similar in a URL, an empty string is returned.

protocol

Selects the protocol. Examples: http, ftp, mailto, magnet...

domain

Selects the domain.

domainWithoutWWW

Selects the domain and removes no more than one 'www.' from the beginning of it, if present.

topLevelDomain

Selects the top-level domain. Example: .ru.

firstSignificantSubdomain

Selects 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

Selects 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

Selects 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

Selects the query-string. Example: page=1&lr=213. query-string does not include the first question mark, or # and everything that comes after #.

fragment

Selects the fragment identifier. fragment does not include the first number sign (#).

queryStringAndFragment

Selects the query-string and fragment identifier. Example: page=1#29390.

extractURLParameter(URL, name)

Selects the value of the 'name' parameter in the URL, if present. Otherwise, selects 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 in exactly the same way as in the argument passed.

extractURLParameters(URL)

Gets an array of name=value strings corresponding to the URL parameters. The values are not decoded in any way.

extractURLParameterNames(URL)

Gets an array of name=value strings corresponding to the names of URL parameters. The values are not decoded in any way.

URLHierarchy(URL)

Gets an array containing the URL trimmed to the `/`, `?` characters 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 thing, but without the protocol and host in the result. The / element (root) is not included. Example: This function is used for implementing tree-view reports by URL in Yandex.Metrica.

```
URLPathHierarchy('https://example.com/browse/CONV-6788') =
[
    '/browse/',
    '/browse/CONV-6788'
]
```

decodeURLComponent(URL)

Returns a URL-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 the 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 URL parameter named 'name', 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 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 indicated 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'), 'en')
FROM system.numbers
LIMIT 15
```

```
-regionToName(regionToArea(toUInt32(number), \'ua\'), \'en\')-
|                                                            |
| Moscow and Moscow region                                   |
| Saint-Petersburg and Leningradskaya oblast                 |
| Belgorod District                                          |
| Ivanovo district                                           |
| Kaluga District                                            |
| Kostroma District                                          |
| Kursk District                                             |
| Lipetsk District                                           |
| Orel District                                              |
| Ryazhan District                                           |
| Smolensk District                                          |
| Tambov District                                            |
| Tver District                                              |
| Tula District                                              |
-------------------------------
```

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'), 'en')
FROM system.numbers
LIMIT 15
```

```
-regionToName(regionToDistrict(toUInt32(number), \'ua\'), \'en\')-
|                                                                |
| Central                                                        |
| Northwest                                                      |
| South                                                          |
| North Kavkaz                                                   |
| Volga Region                                                   |
| Ural                                                           |
| Siberian                                                       |
| Far East                                                       |
| 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` mean the same thing - Ukrainian.

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 return 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.

Immutability

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 for the operators `&&`, `||`, `?:`. 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 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 distributed_table has only one shard, all the functions `f`, `g`, and `h` 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.

Глава 12

Aggregate functions

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 for example MySQL) requires that all expressions in the `SELECT`, `HAVING` and `ORDER BY` clauses be calculated from keys or from aggregate functions. That is, 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.

anyLast(x)

Selects the last value encountered. The result is just as indeterminate as for the ‘any’ function.

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.

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.

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, and dates with times.

Uses an adaptive sampling algorithm: for the calculation state, it uses a sample of element hash values with a size up to 65535. Compared with the widely known HyperLogLog algorithm, this algorithm is less effective in terms of accuracy and memory consumption (even up to proportionality), but it is adaptive. This means that with fairly high accuracy, it consumes less memory during simultaneous computation of cardinality for a large number of data sets whose cardinality has power law distribution (i.e. in cases when most of the data sets are small). This algorithm is also very accurate for data sets with small 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).

There is no compensation for the bias of an estimate, so for large data sets the results are systematically deflated. This function is normally used for computing the number of unique visitors in Yandex.Metrica, so this bias does not play a role.

The result is deterministic (it does not depend on the order of query execution).

uniqCombined(x)

Approximately computes the number of different values of the argument. Works for numbers, strings, dates, date-with-time, for several arguments and arguments-tuples.

A combination of three algorithms is used: an array, a hash table and HyperLogLog with an error correction table. The memory consumption is several times smaller than the `uniq` function, and the accuracy is several times higher. The speed of operation is slightly lower than that of the `uniq` function, but sometimes it can be even higher - in the case of distributed requests, in which a large number of aggregation states are transmitted over the network. The maximum state size is 96 KiB (HyperLogLog of 217 6-bit cells).

The result is deterministic (it does not depend on the order of query execution).

The `uniqCombined` function is a good default choice for calculating the number of different values.

uniqHLL12(x)

Uses the HyperLogLog algorithm to approximate the number of different values of the argument. It uses 212 5-bit cells. The size of the state is slightly more than 2.5 KB.

The result is deterministic (it does not depend on the order of query execution).

In most cases, use the 'uniq' function. You should only use this function if you understand its advantages well.

uniqExact(x)

Calculates the number of different values of the argument, exactly. There is no reason to fear approximations, so it's better to use the `uniq` function. You should 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)

Creates an array of argument values. Values can be added to the array in any (indeterminate) order.

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`.

groupUniqArray(x)

Creates an array from different argument values. Memory consumption is the same as for the `uniqExact` function.

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.

The algorithm is the same as for the `median` function. Actually, `quantile` and `median` are internally the same function. You can use the `quantile` function without parameters - in this case, it calculates the median, and you can use the `median` function with parameters - in this case, it calculates the quantile of the set level.

When using multiple ``quantile` and ``median`` 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` function.

quantileDeterministic(level)(x, determinator)

Calculates the quantile of 'level' using the same algorithm as the `medianDeterministic` function.

quantileTiming(level)(x)

Calculates the quantile of 'level' using the same algorithm as the `medianTiming` function.

quantileTimingWeighted(level)(x, weight)

Calculates the quantile of 'level' using the same algorithm as the `medianTimingWeighted` function.

quantileExact(level)(x)

Computes the level quantile exactly. To do this, all transferred values are added to an array, which is then partially sorted. Therefore, the function consumes O(n) memory, where n is the number of transferred values. However, for a small number of values, the function is very effective.

quantileExactWeighted(level)(x, weight)

Computes the level quantile exactly. In this case, each value is taken into account with the weight weight - as if it is present weight once. The arguments of the function can be considered as histograms, where the value "x" corresponds to the "column" of the histogram of the height weight, and the function itself can be considered as the summation of histograms.

The algorithm is a hash table. Because of this, in case the transmitted values are often repeated, the function consumes less RAM than the quantileExact. You can use this function instead of quantileExact, specifying the number 1 as the weight.

quantileTDigest(level)(x)

Computes the level quantile approximately, using the t-digest algorithm. The maximum error is 1%. The memory consumption per state is proportional to the logarithm of the number of transmitted values.

The performance of the function is below quantile, quantileTiming. By the ratio of state size and accuracy, the function is significantly better than quantile.

The result depends on the order in which the query is executed, and is nondeterministic.

median

Approximates the median. Also see the similar `quantile` function. Works for numbers, dates, and dates with times. For numbers it returns Float64, for dates - a date, and 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 proved to be more practical than another well-known algorithm - QDigest.

The result depends on the order of running the query, and is nondeterministic.

quantiles(level1, level2, ...)(x)

Approximates quantiles of all specified levels. The result is an array containing the corresponding number of values.

varSamp(x)

Calculates the amount `Σ((x - x)2) / (n - 1)`, where 'n' is the sample size and '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. If n $<=$ 1, it returns $+\infty$.

varPop(x)

Calculates the amount `Σ((x - x)2) / n`, where 'n' is the sample size and '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)`.

covarSamp(x, y)

Calculates the value of `Σ((x - x)(y - y)) / (n - 1)`.

Returns Float64. If n <= 1, it returns +∞.

covarPop(x, y)

Calculates the value of `Σ((x - x)(y - y)) / n`.

corr(x, y)

Calculates the Pearson correlation coefficient: `Σ((x - x)(y - y)) / sqrt(Σ((x - x)2) * Σ((y - y)2))`.

Глава 13

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 event time of the DateTime type. 'cond1, cond2 ...' are from one to 32 arguments of the UInt8 type that indicate whether an event condition was met.

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 pages with the address in company were visited earlier than pages with the address in cart.

This is a simple 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)` - Reference to a condition (any number in place of 1). `.*` - Any number of events. `(?t>=1800)` - Time condition. Any quantity of any type of events is allowed over the specified time. The operators <, >, <= may be used instead of >=. Any number may be specified in place of 1800.

Events that occur during the same second may be put in the chain in any order. This may affect the result of the function.

sequenceCount(pattern)(time, cond1, cond2, ...)

Similar to the sequenceMatch function, but it does not return the fact that there is a chain of events, and UInt64 is the number of strings found. Chains are searched without overlapping. That is, the following chain can start only after the end of the previous one.

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 N value 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.

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 query `GROUP BY SearchPhrase HAVING uniqUpTo(4)(UserID) >= 5`

Глава 14

Aggregate function combinators

The name of an aggregate function can have a suffix appended to it. This changes the way the aggregate function works. There are `If` and `Array` combinators. See the sections below.

If combinator. Conditional aggregate functions

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.

You can use aggregate functions to calculate aggregates for multiple conditions at once, without using subqueries and JOINs. For example, in Yandex.Metrica, we use conditional aggregate functions for implementing segment comparison functionality.

Array combinator. Aggregate functions for array arguments

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.

The `-If` and `-Array` combinators can be used together. 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 combinator

If this combinator is used, the aggregate function returns a non-completed/non-finished value (for example, in the case of the `uniq` function, the number of unique values), and the intermediate aggregation state (for example, in the case of the `uniq` function, a hash table for calculating the number of unique values), which has type of `AggregateFunction(...)` and can be used for further processing or can be saved to a table for subsequent pre-aggregation - see the sections "AggregatingMergeTree" and "functions for working with intermediate aggregation states".

Merge combinator

In the case of using this combinator, the aggregate function will take as an argument the intermediate state of an aggregation, pre-aggregate (combine together) these states, and return the finished/complete value.

MergeState combinator

Merges the intermediate aggregation states, similar to the -Merge combinator, but returns a non-complete value, but an intermediate aggregation state, similar to the -State combinator.

Глава 15

Dictionaries

A dictionary is a mapping (key -> attributes) that can be used in a query as functions. You can think of this as a more convenient and efficient type of JOIN with dimension tables.

There are built-in (internal) and add-on (external) dictionaries.

External dictionaries

It is possible to add your own dictionaries from various data sources. The data source for a dictionary can be a file in the local file system, the ClickHouse server, or a MySQL server. A dictionary can be stored completely in RAM and updated regularly, or it can be partially cached in RAM and dynamically load missing values.

The configuration of external dictionaries is in a separate file or files specified in the `dictionaries_config` configuration parameter. This parameter contains the absolute or relative path to the file with the dictionary configuration. A relative path is relative to the directory with the server config file. The path can contain wildcards `*` and `?`, in which case all matching files are found. Example: `dictionaries/*.xml`.

The dictionary configuration, as well as the set of files with the configuration, can be updated without restarting the server. The server checks updates every 5 seconds. This means that dictionaries can be enabled dynamically.

Dictionaries can be created when starting the server, or at first use. This is defined by the `dictionaries_lazy_load` parameter in the main server config file. This parameter is optional, 'true' by default. If set to 'true', each dictionary is created at 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 dictionary config file has the following format:

```
<dictionaries>
    <comment>Optional element with any content; completely ignored.</comment>

    <!-- You can set any number of different dictionaries. -->
    <dictionary>
        <!-- Dictionary name. The dictionary will be accessed for use by this name. -->
        <name>os</name>

        <!-- Data source. -->
        <source>
            <!-- Source is a file in the local file system. -->
            <file>
                <!-- Path on the local file system. -->
                <path>/opt/dictionaries/os.tsv</path>
                <!-- Which format to use for reading the file. -->
                <format>TabSeparated</format>
            </file>

            <!-- or the source is a table on a MySQL server.
            <mysql>
                <!- - These parameters can be specified outside (common for all replicas) or inside a specific replica - ->
                <port>3306</port>
                <user>clickhouse</user>
                <password>qwerty</password>
                <!- - Specify from one to any number of replicas for fault tolerance. - ->
                <replica>
                    <host>example01-1</host>
                    <priority>1</priority> <!- - The lower the value, the higher the priority. - - >
                </replica>
                <replica>
                    <host>example01-2</host>
                    <priority>1</priority>
                </replica>
                <db>conv_main</db>
                <table>counters</table>
            </mysql>
            -->

            <!-- or the source is a table on the ClickHouse server.
            <clickhouse>
                <host>example01-01-1</host>
                <port>9000</port>
                <user>default</user>
                <password></password>
                <db>default</db>
                <table>counters</table>
            </clickhouse>
            <!- - If the address is similar to localhost, the request is made without network interaction. For fault tolerance, you can create a Distributed table on localhost and enter it. - - >
            -->

            <!-- or the source is a executable. If layout.complex_key_cache - list of needed keys will be written in STDIN of program -->
            <executable>
                <!-- Path on the local file system or name located in one of env PATH dirs. -->
                <command>cat /opt/dictionaries/os.tsv</command>
                <!-- Which format to use for reading/writing stream. -->
                <format>TabSeparated</format>
            </executable>
            <!-- or the source is a http server. If layout.complex_key_cache - list of needed␣
↪keys will be sent as POST -->
            <http>
                <!-- Host. -->
                <url>http://[::1]/os.tsv</url>
                <!-- Which format to use for reading answer and making POST. -->
                <format>TabSeparated</format>
            </http>

        </source>

        <!-- Update interval for fully loaded dictionaries. 0 - never update. -->
        <lifetime>
            <min>300</min>
            <max>360</max>
            <!-- The update interval is selected uniformly randomly between min and max, in order␣
↪to spread out the load when updating dictionaries on a large number of servers. -->
        </lifetime>

        <!-- or <!- - The update interval for fully loaded dictionaries or invalidation time for␣
↪cached dictionaries. 0 - never update. - ->
        <lifetime>300</lifetime>
        -->

        <layout> <!-- Method for storing in memory. -->
            <flat />
            <!-- or <hashed />
            or
            <cache>
                <!- - Cache size in number of cells; rounded up to a degree of two. - ->
                <size_in_cells>1000000000</size_in_cells>
            </cache>
            or
            <ip_trie />
            -->
        </layout>

        <!-- Structure. -->
        <structure>
            <!-- Description of the column that serves as the dictionary identifier (key). -->
            <id>
                <!-- Column name with ID. -->
                <name>Id</name>
            </id>

            <attribute>
                <!-- Column name. -->
                <name>Name</name>
                <!-- Column type. (How the column is understood when loading. For MySQL, a table␣
↪can have TEXT, VARCHAR, and BLOB, but these are all loaded as String) -->
                 <type>String</type>
                <!-- Value to use for a non-existing element. In the example, an empty string. -->
                <null_value></null_value>
            </attribute>
            <!-- Any number of attributes can be specified. -->
            <attribute>
                <name>ParentID</name>
                <type>UInt64</type>
                <null_value>0</null_value>
                <!-- Whether it defines a hierarchy - mapping to the parent ID (by default,␣
↪false). -->
                <hierarchical>true</hierarchical>
                <!-- The mapping id -> attribute can be considered injective, in order to␣
↪optimize GROUP BY. (by default, false) -->
                <injective>true</injective>
            </attribute>
        </structure>
    </dictionary>
</dictionaries>
```

The dictionary identifier (key attribute) should be a number that fits into UInt64. Also, you can use arbitrary tuples as keys (see section "Dictionaries with complex keys"). Note: you can use complex keys consisting of just one element. This allows using e.g. Strings as dictionary keys.

There are six ways to store dictionaries in memory.

flat

This is the most effective method. It works if all keys are smaller than `500,000`. If a larger key is discovered when creating the dictionary, an exception is thrown and the dictionary is not created. The dictionary is loaded to RAM in its entirety. The dictionary uses the amount of memory proportional to maximum key value. With the limit of 500,000, memory consumption is not likely to be high. All types of sources are supported. When updating, data (from a file or from a table) is read in its entirety.

hashed

This method is slightly less effective than the first one. The dictionary is also loaded to RAM in its entirety, and can contain any number of items with any identifiers. In practice, it makes sense to use up to tens of millions of items, while there is enough RAM. All types of sources are supported. When updating, data (from a file or from a table) is read in its entirety.

cache

This is the least effective method. It is appropriate if the dictionary doesn't fit in RAM. It is a cache of a fixed number of cells, where frequently-used data can be located. MySQL, ClickHouse, executable, http sources are supported, but file sources are not supported. When searching a dictionary, the cache is searched first. For each data block, all keys not found in the cache (or expired keys) are collected in a package, which is sent to the source with the query `SELECT attrs... FROM db.table WHERE id IN (k1, k2, ...)`. The received data is then written to the cache.

range_hashed

The table lists some data for date ranges, for each key. To give the possibility to extract this data for a given key, for a given date.

Example: in the table there are discounts for each advertiser in the form:

```
advertiser id    discount start date    end date    value
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
```

Adding layout = range_hashed. When using such a layout, the structure should have the elements range_min, range_max.

Example:

```
<structure>
    <id>
        <name>Id</name>
    </id>
    <range_min>
        <name>first</name>
    </range_min>
    <range_max>
        <name>last</name>
    </range_max>
    ...
```

These columns must be of type Date. Other types are not yet supported. The columns indicate a closed date range.

To work with such dictionaries, dictGetT functions must take one more argument - the date:

`dictGetT('dict_name', 'attr_name', id, date)`

The function takes out the value for this id and for the date range, which includes the transmitted date. If no id is found or the range found is not found for the found id, the default value for the dictionary is returned.

If there are overlapping ranges, then any suitable one can be used.

If the range boundary is NULL or is an incorrect date (1900-01-01, 2039-01-01), then the range should be considered open. The range can be open on both sides.

In the RAM, the data is presented as a hash table with a value in the form of an ordered array of ranges and their corresponding values.

Example of a dictionary by ranges:

```
<dictionaries>
        <dictionary>
                <name>xxx</name>
                <source>
                        <mysql>
                                <password>xxx</password>
                                <port>3306</port>
                                <user>xxx</user>
                                <replica>
                                        <host>xxx</host>
                                        <priority>1</priority>
                                </replica>
                                <db>dicts</db>
                                <table>xxx</table>
                        </mysql>
                </source>
                <lifetime>
                        <min>300</min>
                        <max>360</max>
                </lifetime>
                <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>
</dictionaries>
```

ip_trie

The table stores IP prefixes for each key (IP address), which makes it possible to map IP addresses to metadata such as ASN or threat score.

Example: in the table there are prefixes matches to AS number and country:

```
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 such a layout, the structure should have the “key” element.

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>
    ...
```

These key must have only one attribute of type String, containing a valid IP prefix. Other types are not yet supported.

For querying, same functions (dictGetT with tuple) as for complex key dictionaries have to be used:

`dictGetT('dict_name', 'attr_name', tuple(ip))`

The function accepts either UInt32 for IPv4 address or FixedString(16) for IPv6 address in wire format:

`dictGetString('prefix', 'asn', tuple(IPv6StringToNum('2001:db8::1')))`

No other type is supported. The function returns attribute for a prefix matching the given IP address. If there are overlapping prefixes, the most specific one is returned.

The data is stored currently in a bitwise trie, it has to fit in memory.

complex_key_hashed

The same as `hashed`, but for complex keys.

complex_key_cache

The same as `cache`, but for complex keys.

Notes

We recommend using the `flat` method when possible, or `hashed`. The speed of the dictionaries is impeccable with this type of memory storage.

Use the cache method only in cases when it is unavoidable. The speed of the cache depends strongly on correct settings and the usage scenario. A cache type dictionary only works normally for high enough hit rates (recommended 99% and higher). You can view the average hit rate in the system.dictionaries table. Set a large enough cache size. You will need to experiment to find the right number of cells - select a value, use a query to get the cache completely full, look at the memory consumption (this information is in the system.dictionaries table), then proportionally increase the number of cells so that a reasonable amount of memory is consumed. We recommend MySQL as the source for the cache, because ClickHouse doesn't handle requests with random reads very well.

In all cases, performance is better if you call the function for working with a dictionary after `GROUP BY`, and if the attribute being fetched is marked as injective. For a dictionary cache, performance improves if you call the function after LIMIT. To do this, you can use a subquery with LIMIT, and call the function with the dictionary from the outside.

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`.

When updating dictionaries from a file, first the file modification time is checked, and it is loaded only if the file has changed. When updating from MySQL, for flat and hashed dictionaries, first a `SHOW TABLE STATUS` query is made, and the table update time is checked. If it is not NULL, it is compared to the stored time. This works for MyISAM tables, but for InnoDB tables the update time is unknown, so loading from InnoDB is performed on each update.

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.

If a dictionary couldn't be loaded even once, an attempt to use it throws an exception. If an error occurred during a request to a cached source, an exception is thrown. 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.

You can view the list of external dictionaries and their status in the system.dictionaries table.

To use external dictionaries, see the section "Functions for working with external dictionaries".

Note that you can convert values for a small dictionary by specifying all the contents of the dictionary directly in a `SELECT` query (see the section "transform function"). This functionality is not related to external dictionaries.

Dictionaries with complex keys

You can use tuples consisting of fields of arbitrary types as keys. Configure your dictionary with `complex_key_hashed` or `complex_key_cache` layout in this case.

Key structure is configured not in the `<id>` element but in the `<key>` element. Fields of the key tuple are configured analogously to dictionary attributes. Example:

```
<structure>
    <key>
        <attribute>
            <name>field1</name>
            <type>String</type>
        </attribute>
        <attribute>
            <name>field2</name>
            <type>UInt32</type>
        </attribute>
        ...
    </key>
...
```

When using such dictionary, use a Tuple of field values as a key in dictGet* functions. Example: `dictGetString('dict_name', 'attr_name', tuple('field1_value', 123))`.

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 config file.

The geobase is loaded from text files. If you are Yandex employee, to create them, use the following instructions: https://github.yandex-team.ru/raw/Metrika/ClickHouse_private/master/doc/create_embedded_geobase_dictionaries.txt

Put the regions_hierarchy*.txt files in 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 breaks, 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 the server restart. 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, while 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.

Глава 16

Operations

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. Let's look at part of the `users.xml` file:

```
<!-- Users and ACL. -->
<users>
    <!-- If the username is not specified, the default user is used. -->
    <default>
        <!-- Password (in plaintext). May be empty. -->
        <password></password>

        <!-- List of networks that access is allowed from. Each list item has one of the following forms:
            <ip> IP address or subnet mask. For example, 222.111.222.3 or 10.0.0.1/8 or 2a02:6b8::3 or 2a02:6b8::3/64.
            <host> Host name. Example: example01. A DNS query is made for verification, and all received address are compared to the client address.
            <host_regexp> Regex for host names. For example, ^example\d\d-\d\d-\d\.yandex\.ru$
                A DNS PTR query is made to verify the client address and the regex is applied to the result.
                Then another DNS query is made for the result of the PTR query, and all received address are compared to the client address.
                We strongly recommend that the regex ends with \.yandex\.ru$. If you are installing ClickHouse independently, here you should specify:
            <networks>
                <ip>::/0</ip>
            </networks> -->

        <networks incl="networks" />
        <!-- Settings profile for the user. -->
        <profile>default</profile>
        <!-- Quota for the user. -->
        <quota>default</quota>
    </default>

    <!-- For queries from the user interface. -->
    <web>
        <password></password>
        <networks incl="networks" />
        <profile>web</profile>
        <quota>default</quota>
    </web>
```

Here we can see that two users are declared: `default` and `web`. We added the `web` user ourselves. The `default` user is chosen in cases when the username is not passed, so this user must be present in the config file. The `default` user is also used for distributed query processing - the system accesses remote servers under this username. So the `default` user must have an empty password and must not have substantial restrictions or quotas - otherwise, distributed queries will fail.

The password is specified in plain text directly in the config. In this regard, you should not consider these passwords as providing security against potential malicious attacks. Rather, they are necessary for protection from Yandex 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>93.111.222.128/26</ip>
        <ip>2a02:6b8:0:111::/64</ip>
        ...
    </networks>
</yandex>
```

We could have defined 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 `host_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` set to `1`, which provides read-only access.

After this, the quota is defined (see the section "Quotas"). You can specify the default quota, `default`. It is set in the config by default so that it only counts resource usage, but does not restrict 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.

Configuration files

The main server config file is `config.xml`. It resides in the `/etc/clickhouse-server/` directory.

Certain settings can be overridden in the `*.xml` and `*.conf` files from the `conf.d` and `config.d` directories next to the config.

The `replace` and `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 config in the `include_from` element. The substitution values are specified in `/yandex/substitution_name` elements of this file.

You can also perform substitutions from ZooKeeper nodes. To do that add the `from_zk="/path/to/node"` attribute to a config element. Element contents will be substituted with the contents of the /path/to/node ZooKeeper node. The ZooKeeper node can contain a whole XML subtree, and it will be inserted as a child of the substituted node.

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`. For `users_config`, overrides and substitutions may also exist in files from the `users_config.d` directory (for example, `users.d`).

For each config file, the server also generates `file-preprocessed.xml` files on launch. These files contain all the completed substitutions and overrides, and they are intended for informational use. If ZooKeeper substitutions were used in a config file and the ZooKeeper is unavailable during server startup, the configuration is loaded from the respective preprocessed file.

The server tracks changes to config files and files and ZooKeeper nodes that were used for substitutions and overrides and reloads users and clusters configurations in runtime. That is, you can add or change users, clusters and their settings without relaunching 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 multiple time intervals with various␣
↪restrictions. -->
        <interval>
            <!-- Length of time. -->
            <duration>3600</duration>

            <!-- No restrictions. 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.

```
<statbox>
    <!-- Restrictions for a time period. You can set multiple time intervals with various␣
↪restrictions. -->
    <interval>
        <!-- Length of time. -->
        <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.

Let's examine the amounts that can be restricted:

`queries` - The overall number of queries.

`errors` - The number of queries that threw exceptions.

`result_rows` - The total number of rows output in results.

`read_rows` - The total number of source rows retrieved from tables for running a query, on all remote servers.

`execution_time` - The total time of query execution, 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 report builder. -->
<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 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.

Operations Tips

CPU

SSE 4.2 instruction set support is required. Most recent (since 2008) CPUs have this instruction set.

When choosing between CPU with more cores and slightly less frequency and CPU with less cores and more frequency, choose first. For example, 16 cores with 2600 MHz is better than 8 cores with 3600 MHz.

Hyper-Threading

Don't disable hyper-threading. Some queries will benefit from hyper-threading and some will not.

Turbo-Boost

Don't disable turbo-boost. It will do significant performance gain on typical load. Use `turbostat` tool to show real CPU frequency under load.

CPU scaling governor

Always use `performance` scaling governor. `ondemand` scaling governor performs much worse even on constantly high demand.

> sudo echo 'performance' | tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor

CPU throttling

Your CPU could be overheated. Use `dmesg` to show if it was thermal throttled. Your CPU could be power capped in datacenter. Use `turbostat` tool under load to monitor that.

RAM

For small amount of data (up to ~200 GB compressed) prefer to use as much RAM as data volume. For larger amount of data, if you run interactive (online) queries, use reasonable amount of RAM (128 GB or more) to hot data fit in page cache. Even for data volumes of ~50 TB per server, using 128 GB of RAM is much better for query performance than 64 GB.

Swap

Disable swap. The only possible reason to not disable swap is when you are running ClickHouse on your personal laptop/desktop.

Huge pages

Disable transparent huge pages. It interferes badly with memory allocators, leading to major performance degradation.

```
echo ‘never’ | sudo tee /sys/kernel/mm/transparent_hugepage/enabled
```

Use `perf top` to monitor time spent in kernel on doing memory management. Don’t allocate permanent huge pages.

Storage subsystem

If you could afford SSD, use SSD. Otherwise use HDD. SATA HDDs 7200 RPM are Ok.

Prefer more servers with in place storage to less servers with huge disk shelves. Of course you could use huge disk shelves for archive storage with rare queries.

RAID

When using HDDs, you could use RAID-10, RAID-5, RAID-6 or RAID-50. Use Linux software RAID (`mdadm`). Better to not use LVM. When creating RAID-10, choose `far` layout. Prefer RAID-10 if you could afford it.

Don’t use RAID-5 on more than 4 HDDs - use RAID-6 or RAID-50. RAID-6 is better. When using RAID-5, RAID-6, or RAID-50, always increase stripe_cache_size, because default setting is awful.

```
echo 4096 | sudo tee /sys/block/md2/md/stripe_cache_size
```

Exact number is calculated from number of devices and chunk size: `2 * num_devices * chunk_size_in_bytes / 4096`.

Chunk size 1024K is Ok for all RAID configurations. Never use too small or too large chunk sizes.

On SSDs, you could use RAID-0. Regardless to RAID, always use replication for data safety.

Enable NCQ. Use high queue depth. Use CFQ scheduler for HDDs and noop for SSDs. Don’t lower readahead setting. Enable write cache on HDDs.

Filesystem

Ext4 is Ok. Mount with `noatime,nobarrier`. XFS is Ok too, but less tested with ClickHouse. Most other filesystems should work fine. Filesystems with delayed allocation are better.

Linux kernel

Don't use too old Linux kernel. For example, on 2015, 3.18.19 is Ok. You could use Yandex kernel: https://github.com/yandex/smart which gives at least 5% performance increase.

Network

When using IPv6, you must increase route cache. Linux kernels before 3.2 has awful bugs in IPv6 implementation.

Prefer at least 10 Gbit network. 1 Gbit will also work, but much worse for repairing replicas with tens of terabytes of data and for processing huge distributed queries with much intermediate data.

ZooKeeper

Probably you already have ZooKeeper for other purposes. It's Ok to use existing ZooKeeper installation if it is not overloaded.

Use recent version of ZooKeeper. At least 3.5 is Ok. Version in your Linux package repository might be outdated.

With default settings, ZooKeeper have time bomb:

> A ZooKeeper server will not remove old snapshots and log files when using the default configuration (see autopurge below), this is the responsibility of the operator.

You need to defuse the bomb.

Below is ZooKeeper (3.5.1) configuration used by Yandex.Metrica in production as of 2017-05-20.

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 thes 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'] }}
ZOOCFGDIR=/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="$ZOOCFGDIR:/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"

ZOOCFG="$ZOOCFGDIR/zoo.cfg"
ZOO_LOG_DIR=/var/log/$NAME
```

```
USER=zookeeper
GROUP=zookeeper
PIDDIR=/var/run/$NAME
PIDFILE=$PIDDIR/$NAME.pid
SCRIPTNAME=/etc/init.d/$NAME
JAVA=/usr/bin/java
ZOOMAIN="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"
```

Инициализация через Salt:

```
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
```

Settings

In this section, we review settings that you can make using a SET query or in a config file. Remember that these settings can be set for a session or globally. Settings that can only be made in the server config file are not covered here.

There are three ways to setup settings (sorted by priority):

- Settings in server configuration files.

  They are set in user profiles.

- Session-level settings.

  Use `SET setting=value` command in ClickHouse console client running in interactive mode. Also sessions are supported in HTTP interface (you need to pass `session_id` HTTP parameter)

- Query-level settings.

  Use `--setting=value` command line parameters in non-iteractive mode of ClickHouse console client. Use HTTP parameters (`URL?setting_1=value&setting_2=value...`) with HTTP ClickHouse interface.

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 SELECTs. For distributed query processing, restrictions are applied on each server separately.

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.

readonly

If set to 0, allows to run any queries. If set to 1, allows to run only queries that don't change data or settings (e.g. SELECT or SHOW). INSERT and SET are forbidden. If set to 2, allows to run queries that don't change data (SELECT, SHOW) and allows to change settings (SET).

After you set the read-only mode, you won't be able to disable it in the current session.

When using the GET method in the HTTP interface, 'readonly = 1' is set automatically. 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.

max_memory_usage

The maximum amount of memory consumption when running a query on a single server. By default, 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.

Certain cases of memory consumption are not tracked:

- Large constants (for example, a very long string constant).
- The states of 'groupArray' aggregate functions, and also 'quantile' (it is tracked for 'quantileTiming').

Memory consumption is not fully considered for aggregate function states `min`, `max`, `any`, `anyLast`, `argMin`, and `argMax` from String and Array arguments.

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.

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, 10,000`.

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_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. 'max_block_size' 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, so that too much memory isn't consumed when extracting a large number of columns in multiple threads, and so that at least some cache locality is preserved.

By default, it is 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.

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 in the same blocks that are formed after SELECT.

By default, it is 1,048,576.

This 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_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 execution pipeline in parallel. For example, if reading from a table, evaluating expressions with functions, filtering with WHERE and pre-aggregating for GROUP BY can all be done in parallel using at least `max_threads` number of threads, then 'max_threads' are used.

By default, 8.

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`, 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) and 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 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.

By default, 256 KiB.

interactive_delay

The interval in microseconds for checking whether request execution has been canceled and sending the progress.

By default, 100,000 (check for canceling and send progress ten times per second).

connect_timeout

receive_timeout

send_timeout

Timeouts in seconds on the socket used for communicating with the client.

By default, 10, 300, 300.

poll_interval

Lock in a wait loop for the specified number of seconds.

By default, 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.

By default, 100.

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.

By default, 128.

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.

By default, 50.

connections_with_failover_max_tries

The maximum number of connection attempts with each replica, for the Distributed table engine.

By default, 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). The uncompressed cache (only for tables in the MergeTree family) allows significantly reducing latency and increasing 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; 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. So 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.

load_balancing

Which replicas (among healthy replicas) to preferably send a query to (on the first attempt) for distributed processing.

random (by default)

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

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).

As an example, 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 a little stupid, but it doesn't use 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

Replicas are accessed in the same order as they are specified. The number of errors does not matter. This method is appropriate when you know exactly which replica is preferable.

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".

default_sample

A floating-point number from 0 to 1. By default, 1. Allows setting a default sampling coefficient for all SELECT queries. (For tables that don't support sampling, an exception will be thrown.) If set to 1, default sampling is not performed.

max_parallel_replicas

The maximum number of replicas of each shard used when the query is executed. For consistency (to get different parts of the same partition), this option only works for the specified sampling key. The lag of the replicas is not controlled.

compile

Enable query compilation. The default is 0 (disabled).

Compilation is provided for only part of the request processing pipeline - for the first aggregation step (GROUP BY). In the event that this part of the pipeline was compiled, the query can work faster, by deploying short loops and inlining the aggregate function calls. The maximum performance increase (up to four times in rare cases) is achieved on queries with several simple aggregate functions. Typically, the performance gain is negligible. In very rare cases, the request may be slowed down.

min_count_to_compile

After how many times, when the compiled piece of code could come in handy, perform its compilation. The default is 3. In case the value is zero, the compilation is executed synchronously, and the request will wait for the compilation process to finish before continuing. This can be used for testing, otherwise use values starting with 1. Typically, compilation takes about 5-10 seconds. If the value is 1 or more, the compilation is performed asynchronously, in a separate thread. If the result is ready, it will be immediately used, including those already running at the moment requests.

The compiled code is required for each different combination of aggregate functions used in the query and the type of keys in GROUP BY. The compilation results are saved in the build directory as .so files. The number of compilation results is unlimited, since they do not take up much space. When the server is restarted, the old results will be used, except for the server update - then the old results are deleted.

input_format_skip_unknown_fields

If the parameter is true, INSERT operation will skip columns with unknown names from input. Otherwise, an exception will be generated, it is default behavior. The parameter works only for JSONEachRow and TSKV input formats.

output_format_json_quote_64bit_integers

If the parameter is true (default value), UInt64 and Int64 numbers are printed as quoted strings in all JSON output formats. Such behavior is compatible with most JavaScript interpreters that stores all numbers as double-precision floating point numbers. Otherwise, they are printed as regular numbers.

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 ‘profile’. Example:

```
SET profile = 'web'
```

Load the ‘web’ profile. That is, set all the options belonging to the ‘web’ profile.

Settings profiles are declared in the user config file. This is normally ‘users.xml’.

Example:

```
<!-- Settings profiles. -->
<profiles>
    <!-- Default settings -->
    <default>
        <!-- Maximum number of threads for executing 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>100000000</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>
```

In the example, two profiles are set: `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.

Глава 17

ClickHouse Development

Overview of ClickHouse architecture

ClickHouse is a true column-oriented DBMS. Data is stored by columns, and during query execution data is processed by 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 dispatch cost relative to 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 widely 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 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 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 mainly uses vectorized query execution and has limited initial support for runtime code generation (only the inner loop of the first stage of GROUP BY can be compiled).

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 and creates a new filtered column. 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, 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 `Block` s. 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 nice 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++ `iostream`'s. Don't worry: every mature C++ project is using something other than `iostream`'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 `TableFunction``s. 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 arbitary 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 synchronise 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 on Linux

Build should work on Linux Ubuntu 12.04, 14.04 or newer. With appropriate changes, build should work on any other Linux distribution. Build is not intended to work on Mac OS X. 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
```

Detect number of threads

```
export THREADS=$(grep -c ^processor /proc/cpuinfo)
```

Install GCC 6

There are several ways to do it.

Install from 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-6 g++-6
```

Install from sources

Example:

```
# Download gcc from https://gcc.gnu.org/mirrors.html
wget ftp://ftp.fu-berlin.de/unix/languages/gcc/releases/gcc-6.2.0/gcc-6.2.0.tar.bz2
tar xf gcc-6.2.0.tar.bz2
cd gcc-6.2.0
./contrib/download_prerequisites
cd ..
mkdir gcc-build
cd gcc-build
../gcc-6.2.0/configure --enable-languages=c,c++
make -j $THREADS
sudo make install
hash gcc g++
gcc --version
sudo ln -s /usr/local/bin/gcc /usr/local/bin/gcc-6
sudo ln -s /usr/local/bin/g++ /usr/local/bin/g++-6
sudo ln -s /usr/local/bin/gcc /usr/local/bin/cc
sudo ln -s /usr/local/bin/g++ /usr/local/bin/c++
# /usr/local/bin/ should be in $PATH
```

Use GCC 6 for builds

```
export CC=gcc-6
export CXX=g++-6
```

Install required libraries from packages

```
sudo apt-get install libicu-dev libreadline-dev libmysqlclient-dev libssl-dev unixodbc-dev
```

Checkout ClickHouse sources

To get latest stable version:

```
git clone -b stable git@github.com:yandex/ClickHouse.git
# or: git clone -b stable https://github.com/yandex/ClickHouse.git

cd ClickHouse
```

For development, switch to the `master` branch. For latest release candidate, switch to the `testing` branch.

Build ClickHouse

There are two variants of build.

Build release package

Install prerequisites to build debian packages.

```
sudo apt-get install devscripts dupload fakeroot debhelper
```

Install recent version of clang.

Clang is embedded into ClickHouse package and used at runtime. Minimum version is 3.8.0. It is optional.

You can build clang from sources:

```
cd ..
sudo apt-get install subversion
mkdir llvm
cd llvm
svn co http://llvm.org/svn/llvm-project/llvm/tags/RELEASE_400/final llvm
cd llvm/tools
svn co http://llvm.org/svn/llvm-project/cfe/tags/RELEASE_400/final clang
cd ..
cd projects/
svn co http://llvm.org/svn/llvm-project/compiler-rt/tags/RELEASE_400/final compiler-rt
cd ../..
mkdir build
cd build/
cmake -D CMAKE_BUILD_TYPE:STRING=Release ../llvm
make -j $THREADS
sudo make install
hash clang
```

Or install it from packages. On Ubuntu 16.04 or newer:

```
sudo apt-get install clang
```

You may also build ClickHouse with clang for development purposes. For production releases, GCC is used.

Run release script:

```
rm -f ../clickhouse*.deb
./release
```

You will find built packages in parent directory:

```
ls -l ../clickhouse*.deb
```

Note that usage of debian packages is not required. ClickHouse has no runtime dependencies except libc, so it could work on almost any Linux.

Installing just built packages on development server:

```
sudo dpkg -i ../clickhouse*.deb
sudo service clickhouse-server start
```

Build to work with code

```
mkdir build
cd build
cmake ..
make -j $THREADS
cd ..
```

To create an executable, run `make clickhouse`. This will create the `dbms/src/Server/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. If you're using earlier version, you can try to build ClickHouse using Gentoo Prefix and clang sl in this instruction. With appropriate changes, build should work on any other OS X distribution.

Install Homebrew

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Install required compilers, tools, libraries

```
brew install cmake gcc icu4c mysql openssl unixodbc libtool gettext homebrew/dupes/zlib readline boost --cc=gcc-6
```

Checkout ClickHouse sources

To get the latest stable version:

```
git clone -b stable --recursive --depth=10 git@github.com:yandex/ClickHouse.git
# or: git clone -b stable --recursive --depth=10 https://github.com/yandex/ClickHouse.git

cd ClickHouse
```

For development, switch to the `master` branch. For the latest release candidate, switch to the `testing` branch.

Build ClickHouse

```
mkdir build
cd build
cmake .. -DCMAKE_CXX_COMPILER=`which g++-6` -DCMAKE_C_COMPILER=`which gcc-6`
make -j `sysctl -n hw.ncpu`
cd ..
```

Caveats

If you intend to run clickhouse-server, make sure to increase system's maxfiles variable. See MacOS.md for more details.

How to write C++ code

General

1. This text should be considered as recommendations.
2. If you edit some code, it makes sense to keep style in changes consistent with the rest of it.
3. Style is needed to keep code consistent. Consistency is required to make it easier (more convenient) to read the code. And also for code navigation.
4. Many rules do not have any logical explanation and just come from existing practice.

Formatting

1. Most of formatting is done automatically by `clang-format`.
2. Indent is 4 spaces wide. Configure your IDE to insert 4 spaces on pressing Tab.
3. Curly braces on separate lines.

```
inline void readBoolText(bool & x, ReadBuffer & buf)
{
    char tmp = '0';
    readChar(tmp, buf);
    x = tmp != '0';
}
```

4. But if function body is short enough (one statement) you can put the whole thing on one line. In this case put spaces near curly braces, except for the last one in the end.

```
inline size_t mask() const                { return buf_size() - 1; }
inline size_t place(HashValue x) const    { return x & mask(); }
```

5. For functions there are no spaces near brackets.

```
void reinsert(const Value & x)
```

```
memcpy(&buf[place_value], &x, sizeof(x));
```

6. When using expressions if, for, while, ... (in contrast to function calls) there should be space before opening bracket.

```
for (size_t i = 0; i < rows; i += storage.index_granularity)
```

7. There should be spaces around binary operators (+, -, *, /, %, ...) and 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 there’s a line break, operator is written on new line and it has additional indent.

```
if (elapsed_ns)
    message << " ("
        << rows_read_on_server * 1000000000 / elapsed_ns << " rows/s., "
        << bytes_read_on_server * 1000.0 / elapsed_ns << " MB/s.) ";
```

   (a) It is ok to insert additional spaces to align the code.

   ```
   dst.ClickLogID         = click.LogID;
   dst.ClickEventID       = click.EventID;
   dst.ClickGoodEvent     = click.GoodEvent;
   ```

9. **No spaces around `.`, `->` operators.** If necessary these operators can be moved to next line with additional indent.

10. Unary operators (`--`, `++`, `*`, `&`, ...) are not delimited from argument.

11. Space is put after comma or semicolon, not before.

12. Operator `[]` is not delimited with spaces.

13. In `template <...>`, put space between `template` and `<`; after `<` and before `>` - do not.

```
template <typename TKey, typename TValue>
struct AggregatedStatElement
```

14. In classes and structs keywords public, private, protected are written on same indention level as class/struct, while other contents - deeper.

```
template <typename T, typename Ptr = std::shared_ptr<T>>
class MultiVersion
{
public:
    /// Конкретная версия объекта для использования. shared_ptr определяет время жизни версии.
    using Version = Ptr;
```

15. If there’s only one namespace in a file and there’s nothing else significant - no need to indent the namespace.

16. If `if`, `for`, `while...` block consists of only one statement, it’s not required to wrap it in curly braces. Instead you can put the statement on separate line. This statements can also be a `if`, `for`, `while...` block. But if internal statement contains curly braces or else, this option should not be used.

```
/// Если файлы не открыты, то открываем их.
if (streams.empty())
    for (const auto & name : column_names)
        streams.emplace(name, std::make_unique<Stream>(
            storage.files[name].data_file.path(),
            storage.files[name].marks[mark_number].offset));
```

17. No spaces before end of line.

18. Source code should be in UTF-8 encoding.

19. It’s ok to have non-ASCII characters in string literals.

```
<< ", " << (timer.elapsed() / chunks_stats.hits) << " μsec/hit.";
```

20. Don't put multiple statements on single line.
21. Inside functions do not delimit logical blocks by more than one empty line.
22. Functions, classes and similar constructs are delimited by one or two empty lines.
23. const (related to value) is written before type name.

    ```
    const char * pos
    ```

    ```
    const std::string & s
    ```

    char const * pos

24. When declaring pointer or reference symbols * and & should be surrounded by spaces.

    ```
    const char * pos
    ```

    const char* pos const char *pos

25. Alias template types with `using` keyword (except the most simple cases). It can be declared even locally, for example inside functions.

    ```
    using FileStreams = std::map<std::string, std::shared_ptr<Stream>>;
    FileStreams streams;
    ```

    std::map<std::string, std::shared_ptr<Stream>> streams;

26. Do not declare several variables of different types in one statements.

    int x, *y;

27. C-style casts should be avoided.

    std::cerr << (int)c << std::endl;

    ```
    std::cerr << static_cast<int>(c) << std::endl;
    ```

28. In classes and structs group members and functions separately inside each visibility scope.
29. **For small classes and structs, it is not necessary to split method declaration and implementation.** The same for small methods. For templated classes and structs it is better not to split declaration and implementations (because anyway they should be defined in the same translation unit).
30. Lines should be wrapped at 140 symbols, not 80.
31. Always use prefix increment/decrement if postfix is not required.

    ```
    for (Names::const_iterator it = column_names.begin(); it != column_names.end();␣
    ↪++it)
    ```

Comments

1. **You shoud write comments in all not trivial places.** It is very important. While writing comment you could even understand that code does the wrong thing or is completely unnecessary.

```
/** 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. Comments are written before the relevant code. In rare cases - after 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 only in english

5. When writing a library, put it’s detailed description in it’s main header file.

6. You shouldn’t write comments not providing additional information. For instance, you *CAN’T* write empty comments like this one:

```
/*
 * 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:
 */
```

(example is borrowed from here: http://home.tamk.fi/~jaalto/course/coding-style/doc/unmaintainable-code/)

7. You shouldn’t write garbage comments (author, creation date...) in the beginning of each file.

8. **One line comments should start with three slashes: ///, multiline - with /**. This comments are consic**

   Note: such comments could be used to generate docs using Doxygen. But in reality Doxygen is not used because it is way more convenient to use IDE for code navigation.

9. In beginning and end of multiline comments there should be no empty lines (except the one where the comment ends).
10. For commented out code use simple, not "documenting" comments. Delete commented out code before commits.
11. Do not use profanity in comments or code.
12. **Do not use too many question signs, exclamation points or capital letters.** /// WHAT THE FAIL???
13. **Do not make delimeters from comments.** /**/
14. **Do not create discussions in comments.** /// Why you did this?
15. **Do not comment end of block describing what kind of block it was.** } /// for

Names

1. Names of variables and class members — in lowercase with underscores.

```
size_t max_block_size;
```

2. Names of functions (methids) - in camelCase starting with lowercase letter.

```
std::string getName() const override { return "Memory"; }
```

3. Names of classes (structs) - CamelCase starting with uppercase letter. Prefixes are not used, except I for interfaces.

```
class StorageMemory : public IStorage
```

4. Names of `using`'s - same as classes and can have _t suffix.
5. **Names of template type arguments: in simple cases - T; T, U; T1, T2.** In complex cases - like class names or can have T prefix.

```
template <typename TKey, typename TValue>
struct AggregatedStatElement
```

6. Names of template constant arguments: same as variable names or N in simple cases.

```
template <bool without_www>
struct ExtractDomain
```

7. For abstract classes (interfaces) you can add I to the start of name.

```
class IBlockInputStream
```

8. **If variable is used pretty locally, you can use short name.** In other cases - use descriptive name.

```
bool info_successfully_loaded = false;
```

9. `define`'s should be in ALL_CAPS with underlines. Global constants - too.

```
#define MAX_SRC_TABLE_NAMES_TO_STORE 1000
```

10. **Names of files should match it's contents.** If file contains one class - name it like class in CamelCase. If file contains one function - name it like function in camelCase.

11. **If name contains an abbreviation:**

    - **for variables names it should be all lowercase;** `mysql_connection` mySQL_connection

    - **for class and function names it should be all uppercase;** `MySQLConnection` MySqlConnection

12. Constructor arguments used just to initialize the class members, should have the matching name, but with underscore suffix.

    ```
    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 argument is not used in constructor body.

13. **Naming of local variables and class members do not have any differences (no prefixes required).** `timer` m_timer

14. Constants in enums - CamelCase starting with uppercase letter. ALL_CAPS is also ok. If enum is not local, use enum class.

    ```
    enum class CompressionMethod
    {
        QuickLZ = 0,
        LZ4     = 1,
    };
    ```

15. **All names - in English. Transliteration from Russian is not allowed.** Stroka

16. Abbreviations are fine only if they are well known (when you can find what it means in wikipedia or with web search query).

    `AST` `SQL` NVDH (some random letters)

    Using incomplete words is ok if it is commonly used. Also you can put the whole word in comments.

17. **C++ source code extensions should be .cpp. Header files - only .h.** .hpp .cc .C .inl `.inl.h` is ok, but not .h.inl:strike:

How to write code

1. **Memory management.** Manual memory deallocation (delete) is ok only in destructors in library code. In application code memory should be freed by some object that owns it. Examples: * you can put object on stack or make it a member of another class. * use containers for many small objects. * for automatic deallocation of not that many objects residing in heap, use shared_ptr/unique_ptr.

2. **Resource management.** Use RAII and see abovee.

3. **Error handling.** Use exceptions. In most cases you should only throw exception, but not catch (because of RAII). In offline data processing applications it's often ok not to catch exceptions. In server code serving user requests usually you should catch exceptions only on top level of connection handler. In thread functions you should catch and keep all exceptions to rethrow it in main thread after join.

   ```
   /// If there were no other calculations yet - lets do it synchronously
   if (!started)
   {
       calculate();
       started = true;
   }
   else    /// If the calculations are already in progress - lets wait
       pool.wait();

   if (exception)
       exception->rethrow();
   ```

   Never hide exceptions without handling. Never just blindly put all exceptions to log. 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 error codes - always check it and throw exception in case of error.

   ```
   if (0 != close(fd))
       throwFromErrno("Cannot close file " + file_name, ErrorCodes::CANNOT_CLOSE_FILE);
   ```

   Asserts are not used.

4. **Exception types.** No need to use complex exception hierarchy in application code. Exception code should be understandable by operations engineer.

5. **Throwing exception from destructors.** Not recommended, but allowed. Use the following options: * Create function (done() or finalize()) that will in advance do all the work that might lead to exception. If that function was called, later there should be no exceptions in destructor. * Too complex work (for example, sending messages via network) can be put in separate method that class user will have to call before destruction. * If nevertheless there's an exception in destructor it's better to log it that to hide it. * In simple applications it is ok to rely on std::terminate (in case of noexcept by default in C++11) to handle exception.

6. **Anonymous code blocks.** It is ok to declare anonymous code block to make some variables local to it and make them be destroyed earlier than they otherwise would.

   ```
   Block block = data.in->read();

   {
       std::lock_guard<std::mutex> lock(mutex);
       data.ready = true;
       data.block = block;
   }

   ready_any.set();
   ```

7. **Multithreading.** In case of offline data processing applications: * Try to make code as fast as possible on single core. * Make it parallel only if single core performance appeared to be not enough. In server application: * use thread pool for request handling; * for now there were no tasks where userspace context switching was really necessary. Fork is not used to parallelize code.

8. **Synchronizing threads.** Often it is possible to make different threads use different memory cells (better - different cache lines) and do not use any synchronization (except joinAll). If synchronization is necessary in most cases mutex under lock_guard is enough. In other cases use system synchronization primitives. Do not use busy wait. Atomic operations should be used only in the most simple cases. Do not try to implement lock-free data structures unless it is your primary area of expertise.

9. **Pointers vs reference.** Prefer references.

10. **const.** Use constant references, pointers to constants, const_iterator, const methods. Consider const to be default and use non-const only when necessary. When passing variable by value using const usually do not make sense.

11. **unsigned.** unsinged is ok if necessary.

12. **Numeric types.** Use types UInt8, UInt16, UInt32, UInt64, Int8, Int16, Int32, Int64, as well as size_t, ssize_t, ptrdiff_t. Do not use типы signed/unsigned long, long long, short; signed char, unsigned char, and char.

13. **Passing arguments.** Pass complex values by reference (including std::string). If functions captures object ownership created in heap, make an argument to be shared_ptr or unique_ptr.

14. **Returning values.** In most cases just use return. Do not write return std::move(res). If function allocates an object on heap and returns it, use shared_ptr or unique_ptr. In rare cases you might need to return value via argument, in this cases the argument should be a reference.

```
using AggregateFunctionPtr = std::shared_ptr<IAggregateFunction>;

/** Creates aggregate function by it's name
  */
class AggregateFunctionFactory
{
public:
    AggregateFunctionFactory();
    AggregateFunctionPtr get(const String & name, const DataTypes & argument_types)␣
↪const;
```

15. **namespace.** No need to use separate namespace for application code or small libraries. For medium to large libraries - put everything in namespace. You can use additional detail namespace in .h file to hide implementation details. In .cpp you can use static or anonymous namespace to hide symbols. You can also use namespace for enums to prevent it's names to pollute outer namespace, but it's better to use enum class.

16. **Delayed initialization.** If arguments are required for initialization then do not write default constructor. If later you'll need to delay initialization you can add default constructor creating invalid object. For small number of object you could use shared_ptr/unique_ptr.

```
Loader(DB::Connection * connection_, const std::string & query, size_t max_block_size_);

/// For delayed initialization
Loader() {}
```

17. **Virtual methods.** Do not mark methods or destructor as virtual if class is not intended for polymorph usage.
18. **Encoding.** Always use UTF-8. Use `std::string`, `char *`. Do not use `std::wstring`, `wchar_t`.
19. **Logging.** See examples in code. Remove debug logging before commit. Logging in cycles should be avoided, even on Trace level. Logs should be readable even with most detailed settings. Log mostly in application code. Log messages should be in English and understandable by operations engineers.
20. **I/O.** In internal cycle (critical for application performance) you can't use iostreams (especially stringstream). Instead use DB/IO library.
21. **Date and time.** See DateLUT library.
22. **include.** Always use `#pragma once` instead of include guards.
23. **using.** Don't use `using namespace`. `using` something specific is fine, try to put it as locally as possible.
24. **Do not use trailing return unless necessary.** auto f() -> void;
25. **Do not delcare and init variables like this:** auto s = std::string{"Hello"}; Do it like this instead: `std::string s = "Hello"; std::string s{"Hello"};`
26. For virtual functions write virtual in base class and don't forget to write override in descendant classes.

Unused C++ features

1. Virtual inheritance.
2. Exception specifiers from C++03.
3. Function try block, except main function in tests.

Platform

1. We write code for specific platform. But other things equal cross platform and portable code is preferred.
2. Language is C++17. GNU extensions can be used if necessary.
3. **Compiler is gcc. As of April 2017 version 6.3 is used. It is also compilable with clang 4.** Standard library from gcc is used.
4. OS - Linux Ubuntu, not older than Precise.
5. **Code is written for x86_64 CPU architecture.** CPU instruction set is SSE4.2 is currently required.
6. `-Wall -Werror` compilation flags are used.
7. Static linking is used by default. See ldd output for list of exceptions from this rule.
8. Code is developed and debugged with release settings.

Tools

1. Good IDE - KDevelop.
2. For debugging gdb, valgrind (memcheck), strace, -fsanitize=..., tcmalloc_minimal_debug are used.
3. For profiling - Linux Perf, valgrind (callgrind), strace -cf.

4. Source code is in Git.
5. Compilation is managed by CMake.
6. Releases are in deb packages.
7. **Commits to master should not break build.** Though only selected revisions are considered workable.
8. **Commit as often as possible, even if code is not quite ready yet.** Use branches for this. If your code is not buildable yet, exclude it from build before pushing to master; you'll have few days to fix or delete it from master after that.
9. For non-trivial changes use branches and publish them on server.
10. Unused code is removed from repository.

Libraries

1. C++14 standard library is used (experimental extensions are fine), as well as boost and Poco frameworks.
2. **If necessary you can use any well known library available in OS with packages.** If there's a good ready solution it is used even if it requires to install one more dependency. (Buy be prepared to remove bad libraries from code.)
3. It is ok to use library not from packages if it is not available or too old.
4. If the library is small and does not have it's own complex build system, you should put it's sources in contrib folder.
5. Already used libraries are preferred.

General

1. Write as short code as possible.
2. Try the most simple solution.
3. Do not write code if you do not know how it will work yet.
4. In the most simple cases prefer using over classes and structs.
5. Write copy constructors, assignment operators, destructor (except virtual), mpve-constructor and move assignment operators only if there's no other option. You can use `default`.
6. It is encouraged to simplify code.

Additional

1. **Explicit std:: for types from stddef.h.** Not recommended, but allowed.
2. **Explicit std:: for functions from C standard library.** Not recommended. For example, write memcpy instead of std::memcpy. Sometimes there are non standard functions with similar names, like memmem. It will look weird to have memcpy with std:: prefix near memmem without. Though specifying std:: is not prohibited.
3. **Usage of C functions when there are alternatives in C++ standard library.** Allowed if they are more effective. For example, use memcpy instead of std::copy for copying large chunks of memory.

4. **Multiline function arguments.** All of the following 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)
```

How to run ClickHouse tests

The `clickhouse-test` utility that is used for functional testing is written using Python 2.x. It also requires you to have some third-party packages:

```
$ pip install lxml termcolor
```

In a nutshell:

- Put the `clickhouse` program to `/usr/bin` (or `PATH`)
- Create a `clickhouse-client` symlink in `/usr/bin` pointing to `clickhouse`
- Start the `clickhouse` server
- `cd dbms/tests/`
- Run `./clickhouse-test`

Example usage

Run `./clickhouse-test --help` to see available options.

To run tests without having to create a symlink or mess with `PATH`:

```
./clickhouse-test -c "../../build/dbms/src/Server/clickhouse --client"
```

To run a single test, i.e. `00395_nullable`:

```
./clickhouse-test 00395
```

Глава 18

Roadmap

Q3 2017

- SYSTEM queries
- Limit on parallel replica downloads
- Finalize NULL support
- SELECT db.table.column

Q4 2017

- Arbitrary partitioning key for MergeTree engine family
- Better compliance of JOIN syntax with SQL standard
- Resource pools for queries (CPU, disk I/O, network bandwidth)

Q1 2018

- Basic support for UPDATE and DELETE