什么是ClickHouse?

ClickHouse是一个用于联机分析(OLAP)的列式数据库管理系统(DBMS)。

在传统的行式数据库系统中，数据按如下顺序存储：

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

处于同一行中的数据总是被物理的存储在一起。

常见的行式数据库系统有： MySQL、Postgres和MS SQL Server。

在列式数据库系统中，数据按如下的顺序存储：

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

该示例中只展示了数据在列式数据库中数据的排列顺序。 对于存储而言，列式数据库总是将同一列的数据存储在一起，不同列的数据也总是分开存储。

常见的列式数据库有： Vertica、 Paraccel (Actian Matrix，Amazon Redshift)、 Sybase IQ、 Exasol、 Infobright、 InfiniDB、 MonetDB (VectorWise， Actian Vector)、 LucidDB、 SAP HANA、 Google Dremel、 Google PowerDrill、 Druid、 kdb+。

不同的存储方式适合不同的场景，这里的查询场景包括： 进行了哪些查询，多久查询一次以及各类查询的比例； 每种查询读取多少数据————行、列和字节；读取数据和写入数据之间的关系；使用的数据集大小以及如何使用本地的数据集；是否使用事务,以及它们是如何进行隔离的；数据的复制机制与数据的完整性要求；每种类型的查询要求的延迟与吞吐量等等。

系统负载越高，根据使用场景进行定制化就越重要，并且定制将会变的越精细。没有一个系统同样适用于明显不同的场景。如果系统适用于广泛的场景，在负载高的情况下，所有的场景可以会被公平但低效处理，或者高效处理一小部分场景。

OLAP场景的关键特征

- 大多数是读请求

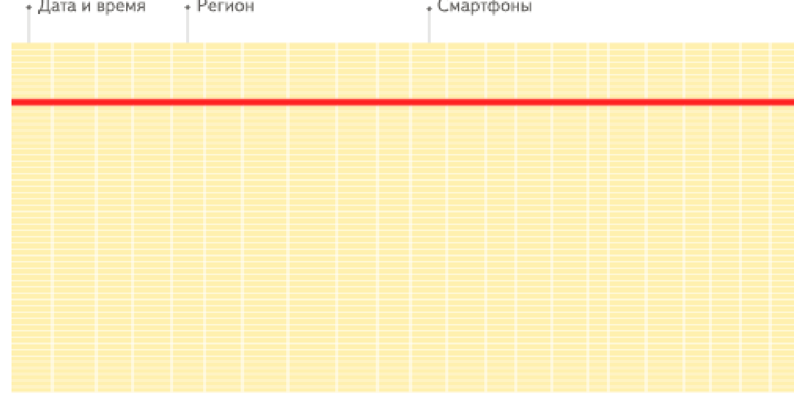
- 数据总是以相当大的批(> 1000 rows)进行写入
- 不修改已添加的数据
- 每次查询都从数据库中读取大量的行，但是同时又仅需要少量的列
- 宽表，即每个表包含着大量的列
- 较少的查询(通常每台服务器每秒数百个查询或更少)
- 对于简单查询，允许延迟大约50毫秒
- 列中的数据相对较小： 数字和短字符串(例如，每个URL 60个字节)
- 处理单个查询时需要高吞吐量（每个服务器每秒高达数十亿行）
- 事务不是必须的
- 对数据一致性要求低
- 每一个查询除了一个大表外都很小
- 查询结果明显小于源数据，换句话说，数据被过滤或聚合后能够被盛放在单台服务器的内存中

很容易可以看出，OLAP场景与其他流行场景(例如,OLTP或K/V)有很大的不同， 因此想要使用OLTP或Key-Value数据库去高效的处理分析查询是没有意义的，例如，使用OLAP数据库去处理分析请求通常要优于使用MongoDB或Redis去处理分析请求。

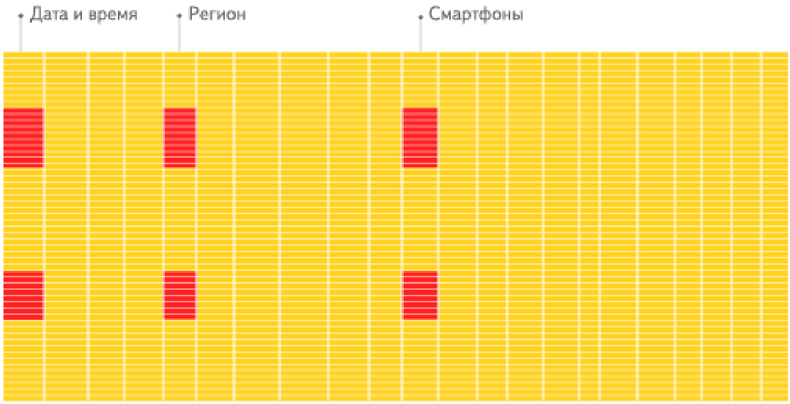
列式数据库更适合OLAP场景的原因

列式数据库更适合于OLAP场景(对于大多数查询而言，处理速度至少提高了100倍)，下面详细解释了原因(通过图片更有利于直观理解)：

行式



列式

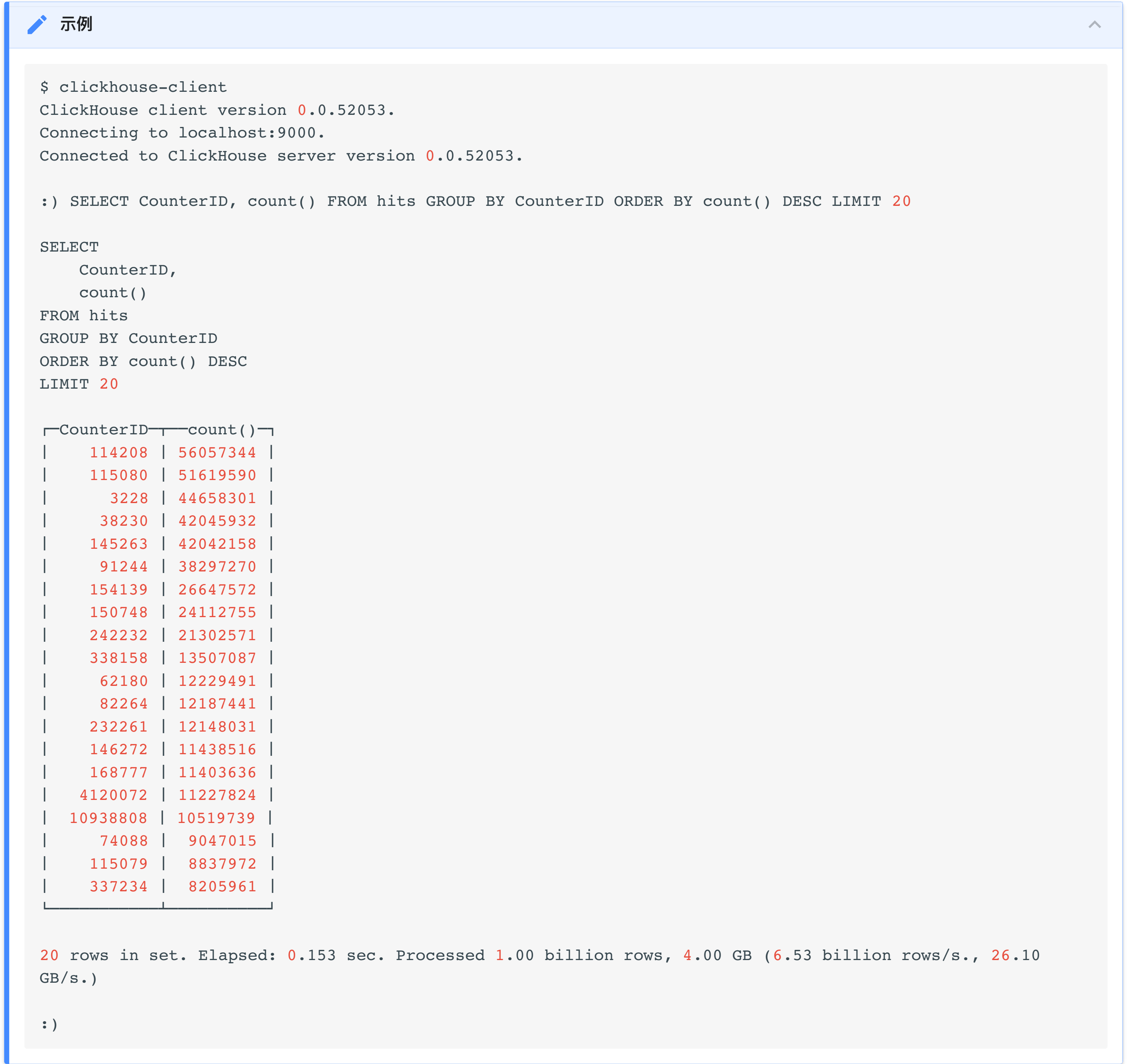


看到差别了么？下面将详细介绍为什么会发生这种情况。

Input/output

1. 针对分析类查询，通常只需要读取表的一小部分列。在列式数据库中你可以只读取你需要的数据。例如，如果只需要读取100列中的5列，这将帮助你最少减少20倍的I/O消耗。
2. 由于数据总是打包成批量读取的，所以压缩是非常容易的。同时数据按列分别存储这也更容易压缩。这进一步降低了I/O的体积。
3. 由于I/O的降低，这将帮助更多的数据被系统缓存。

例如，查询“统计每个广告平台的记录数量”需要读取“广告平台ID”这一列，它在未压缩的情况下需要1个字节进行存储。如果大部分流量不是来自广告平台，那么这一列至少可以以十倍的压缩率被压缩。当采用快速压缩算法，它的解压速度最少在十亿字节(未压缩数据)每秒。换句话说，这个查询可以在单个服务器上以每秒大约几十亿行的速度进行处理。这实际上是当前实现的速度。



示例

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

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

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

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

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

:)
```

CPU

由于执行一个查询需要处理大量的行，因此在整个向量上执行所有操作将比在每一行上执行所有操作更加高效。同时这将有助于实现一个几乎没有调用成本的查询引擎。如果你不这样做，使用任何一个机械硬盘，查询引擎都不可避免的停止CPU进行等待。所以，在数据按列存储并且按列执行是很有意义的。

有两种方法可以做到这一点：

1. 向量引擎：所有的操作都是为向量而不是为单个值编写的。这意味着多个操作之间的不再需要频繁的调用，并且调用的成

本基本可以忽略不计。操作代码包含一个优化的内部循环。

2. 代码生成：生成一段代码，包含查询中的所有操作。

这是不应该在一个通用数据库中实现的，因为这在运行简单查询时是没有意义的。但是也有例外，例如，MemSQL使用代码生成来减少处理SQL查询的延迟(只是为了比较，分析型数据库通常需要优化的是吞吐而不是延迟)。

请注意，为了提高CPU效率，查询语言必须是声明型的(SQL或MDX)， 或者至少一个向量(J，K)。 查询应该只包含隐式循环，允许进行优化。

Distinctive Features of ClickHouse

True Column-Oriented DBMS

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

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

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

Data Compression

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

Disk Storage of Data

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

Parallel Processing on Multiple Cores

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

Distributed Processing on Multiple Servers

Almost none of the columnar DBMSs mentioned above have support for distributed query processing. In ClickHouse, data can reside on different shards. Each shard can be a group of replicas that are used for fault tolerance. The query is processed on all the shards in parallel. This is transparent for the user.

SQL Support

ClickHouse supports a declarative query language based on SQL that is identical to the SQL standard in many cases. Supported queries include GROUP BY, ORDER BY, subqueries in FROM, IN, and JOIN clauses, and scalar subqueries. Dependent subqueries and window functions are not supported.

Vector Engine

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

Real-time Data Updates

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

Index

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

Suitable for Online Queries

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

Support for Approximated Calculations

ClickHouse provides various ways to trade accuracy for performance:

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

Data replication and data integrity support

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

For more information, see the section Data replication [#table_engines-replication].

ClickHouse可以考虑缺点的功能

1. 没有完整的交易。
2. 缺乏以高速率和低延迟修改或删除已插入数据的能力。 有批次删除和更新可用于清理或修改数据，例如符合GDPR [https://gdpr-info.eu]。
3. 稀疏索引使得ClickHouse不适合通过其键检索单行的点查询。

Performance

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

This has also been confirmed by numerous independent benchmarks. They are not difficult to find using an internet search, or you can see our small collection of related links[https://clickhouse.yandex/#independent-bookmarks].

Throughput for a Single Large Query

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

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

Latency When Processing Short Queries

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

Throughput When Processing a Large Quantity of Short Queries

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

Performance When Inserting Data

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

Yandex.Metrica Use Case

ClickHouse was originally developed to power Yandex.Metrica [https://metrica.yandex.com/], the second largest web analytics platform in the world [http://w3techs.com/technologies/overview/traffic_analysis/all], and continues to be the core component of this system. With more than 13 trillion records in the database and more than 20 billion events daily, ClickHouse allows generating custom reports on the fly directly from non-aggregated data. This article briefly covers the goals of ClickHouse in the early stages of its development.

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

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

Usage in Yandex.Metrica and Other Yandex Services

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

ClickHouse is also used for:

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

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

Aggregated and Non-aggregated Data

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

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

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

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

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

Yandex.Metrica has a specialized system for aggregating data called Metrage, which is used for the majority of reports. Starting in 2009, Yandex.Metrica also used a specialized OLAP database for non-aggregated data called OLAPServer, which was previously used for the report builder. OLAPServer worked well for non-aggregated data, but it had many restrictions that did not allow it to be used for all reports as desired. These included the lack of support for data types (only numbers), and the inability to incrementally update data in real-time (it could only be done by rewriting data daily).

OLAPServer is not a DBMS, but a specialized DB.

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

入门指南

系统要求

如果从官方仓库安装，需要确保您使用的是x86_64处理器构架的Linux并且支持SSE 4.2指令集

检查是否支持SSE 4.2：

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

我们推荐使用Ubuntu或者Debian。终端必须使用UTF-8编码。

基于rpm的系统,你可以使用第三方的安装包：https://packagecloud.io/altinity/clickhouse 或者直接安装debian安装包。

ClickHouse还可以在FreeBSD与Mac OS X上工作。同时它可以在不支持SSE 4.2的x86_64构架和AArch64 CPUs上编译。

安装

为了测试和开发，系统可以安装在单个服务器或普通PC机上。

为Debian/Ubuntu安装

在 `/etc/apt/sources.list` (或创建 `/etc/apt/sources.list.d/clickhouse.list` 文件)中添加仓库：

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

如果你想使用最新的测试版本，请使用'testing'替换'stable'。

然后运行：

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

你也可以从这里手动下载安装包：https://repo.yandex.ru/clickhouse/deb/stable/main/ [https://repo.yandex.ru/clickhouse/deb/stable/main/]。

ClickHouse包含访问控制配置，它们位于 `users.xml` 文件中(与'config.xml'同目录)。 默认情况下，允许从任何地方使用默认的‘default’用户无密码的访问ClickHouse。参考‘user/default/networks’。 有关更多信息，请参考"Configuration files"部分。

使用源码安装

具体编译方式可以参考build.md。

你可以编译并安装它们。 你也可以直接使用而不进行安装。

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

在服务器中为数据创建如下目录：

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

(它们可以在server config中配置。) 为需要的用户运行‘chown’

日志的路径可以在server config (src/dbms/programs/server/config.xml)中配置。

其他的安装方法

Docker image：https://hub.docker.com/r/yandex/clickhouse-server/ [https://hub.docker.com/r/yandex/clickhouse-server/]

CentOS或RHEL安装包：https://github.com/Altinity/clickhouse-rpm-install [https://github.com/Altinity/clickhouse-rpm-install]

Gentoo： `emerge clickhouse`

启动

可以运行如下命令在后台启动服务：

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

可以在 `/var/log/clickhouse-server/` 目录中查看日志。

如果服务没有启动，请检查配置文件 `/etc/clickhouse-server/config.xml` 。

你也可以在控制台中直接启动服务：

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

在这种情况下，日志将被打印到控制台中，这在开发过程中很方便。 如果配置文件在当前目录中，你可以不指定‘--config-file’参数。它默认使用‘./config.xml’。

你可以使用命令行客户端连接到服务：

```
clickhouse-client
```

默认情况下它使用‘default’用户无密码的与localhost:9000服务建立连接。 客户端也可以用于连接远程服务，例如：

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

有关更多信息，请参考"Command-line client"部分。

检查系统是否工作：

```
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.

:)
```

恭喜，系统已经工作了!

为了继续进行实验，你可以尝试下载测试数据集。

[]

航班飞行数据

下载数据：

```
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
```

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

创建表结构：

```
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)
```

加载数据：

```
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
```

查询：

Q0.

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

Q1. 查询从2000年到2008年每天的航班数

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

Q2. 查询从2000年到2008年每周延误超过10分钟的航班数。

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

Q3. 查询2000年到2008年每个机场延误超过10分钟以上的次数

```
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. 查询2007年各航空公司延误超过10分钟以上的次数

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

Q5. 查询2007年各航空公司延误超过10分钟以上的百分比

```
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;
```

更好的查询版本:

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

Q6. 同上一个查询一致,只是查询范围扩大到2000年到2008年

```
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;
```

更好的查询版本：

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

Q7. 每年航班延误超过10分钟的百分比

```
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
```

更好的查询版本：

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

Q8. 每年更受人们喜爱的目的地

```
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;
```

这个性能测试由Vadim Tkachenko提供。参考：

- https://www.percona.com/blog/2009/10/02/analyzing-air-traffic-performance-with-infobright-and-monetdb/ [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/2009/10/26/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/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/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/ [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 [http://nickmakos.blogspot.ru/2012/08/analyzing-air-traffic-performance-with.html]

纽约市出租车数据

怎样导入原始数据

可以参考https://github.com/toddwschneider/nyc-taxi-data [https://github.com/toddwschneider/nyc-taxi-data]和http://tech.marksblogg.com/billion-nyc-taxi-rides-redshift.html [http://tech.marksblogg.com/billion-nyc-taxi-rides-redshift.html]中的关于数据集结构描述与数据下载指令说明。

数据集包含227GB的CSV文件。这大约需要一个小时的下载时间(1Gbit带宽下，并行下载大概是一半时间)。 下载时注意损坏的文件。可以检查文件大小并重新下载损坏的文件。

有些文件中包含一些无效的行，您可以使用如下语句修复他们：

```
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
```

然后您必须在PostgreSQL中预处理这些数据。这将创建多边形中的点（以匹配在地图中纽约市中范围），然后通过使用JOIN查询将数据关联组合到一个规范的表中。为了完成这部分操作，您需要安装PostgreSQL的同时安装PostGIS插件。

运行 `initialize_database.sh` 时要小心，并手动重新检查是否正确创建了所有表。

在PostgreSQL中处理每个月的数据大约需要20-30分钟，总共大约需要48小时。

您可以按如下方式检查下载的行数：

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

real    7m9.164s
```

(根据Mark Litwintschik的系列博客报道数据略多余11亿行)

PostgreSQL处理这些数据大概需要370GB的磁盘空间。

从PostgreSQL中导出数据：

```
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';
```

数据快照的创建速度约为每秒50 MB。 在创建快照时，PostgreSQL以每秒约28 MB的速度从磁盘读取数据。 这大约需要5个小时。 最终生成的TSV文件为590612904969 bytes。

在ClickHouse中创建临时表：

```sql
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;
```

接下来,需要将字段转换为更正确的数据类型，并且在可能的情况下，消除NULL。

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

real    75m56.214s
```

数据的读取速度为112-140 Mb/秒。 通过这种方式将数据加载到Log表中需要76分钟。 这个表中的数据需要使用142 GB的磁盘空间.

（也可以直接使用 `COPY ... TO PROGRAM` 从Postgres中导入数据）

由于数据中与天气相关的所有数据（precipitation……average_wind_speed）都填充了NULL。 所以，我们将从最终数据集中删除它们

首先，我们使用单台服务器创建表，后面我们将在多台节点上创建这些表。

创建表结构并写入数据：

```
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' = 195)) AS pickup_ntaname,

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

assumeNotNull(dropoff_nyct2010_gid) AS dropoff_nyct2010_gid,
toFloat32(ifNull(dropoff_ctlabel, '0')) AS dropoff_ctlabel,
assumeNotNull(dropoff_borocode) AS dropoff_borocode,
CAST(assumeNotNull(dropoff_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
```

```
Riverdale' = 120, 'North Side-South Side' = 121, 'Norwood' = 122, 'Oakland Gardens' = 123, 'Oakwood-Oakwood
Beach' = 124, 'Ocean Hill' = 125, 'Ocean Parkway South' = 126, 'Old Astoria' = 127, 'Old Town-Dongan Hills-
South Beach' = 128, 'Ozone Park' = 129, 'Park Slope-Gowanus' = 130, 'Parkchester' = 131, 'Pelham Bay-Country
Club-City Island' = 132, 'Pelham Parkway' = 133, 'Pomonok-Flushing Heights-Hillcrest' = 134, 'Port Richmond' =
135, 'Prospect Heights' = 136, 'Prospect Lefferts Gardens-Wingate' = 137, 'Queens Village' = 138, 'Queensboro
Hill' = 139, 'Queensbridge-Ravenswood-Long Island City' = 140, 'Rego Park' = 141, 'Richmond Hill' = 142,
'Ridgewood' = 143, 'Rikers Island' = 144, 'Rosedale' = 145, 'Rossville-Woodrow' = 146, 'Rugby-Remsen Village'
= 147, 'Schuylerville-Throgs Neck-Edgewater Park' = 148, 'Seagate-Coney Island' = 149, 'Sheepshead Bay-
Gerritsen Beach-Manhattan Beach' = 150, 'SoHo-TriBeCa-Civic Center-Little Italy' = 151, 'Soundview-Bruckner'
= 152, 'Soundview-Castle Hill-Clason Point-Harding Park' = 153, 'South Jamaica' = 154, 'South Ozone Park' =
155, 'Springfield Gardens North' = 156, 'Springfield Gardens South-Brookville' = 157, 'Spuyten Duyvil-
Kingsbridge' = 158, 'St. Albans' = 159, 'Stapleton-Rosebank' = 160, 'Starrett City' = 161, 'Steinway' = 162,
'Stuyvesant Heights' = 163, 'Stuyvesant Town-Cooper Village' = 164, 'Sunset Park East' = 165, 'Sunset Park
West' = 166, 'Todt Hill-Emerson Hill-Heartland Village-Lighthouse Hill' = 167, 'Turtle Bay-East Midtown' =
168, 'University Heights-Morris Heights' = 169, 'Upper East Side-Carnegie Hill' = 170, 'Upper West Side' =
171, 'Van Cortlandt Village' = 172, 'Van Nest-Morris Park-Westchester Square' = 173, 'Washington Heights
North' = 174, 'Washington Heights South' = 175, 'West Brighton' = 176, 'West Concourse' = 177, 'West Farms-
Bronx River' = 178, 'West New Brighton-New Brighton-St. George' = 179, 'West Village' = 180, 'Westchester-
Unionport' = 181, 'Westerleigh' = 182, 'Whitestone' = 183, 'Williamsbridge-Olinville' = 184, 'Williamsburg' =
185, 'Windsor Terrace' = 186, 'Woodhaven' = 187, 'Woodlawn-Wakefield' = 188, 'Woodside' = 189, 'Yorkville' =
190, 'park-cemetery-etc-Bronx' = 191, 'park-cemetery-etc-Brooklyn' = 192, 'park-cemetery-etc-Manhattan' =
193, 'park-cemetery-etc-Queens' = 194, 'park-cemetery-etc-Staten Island' = 195)) AS dropoff_ntaname,

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

FROM trips
```

这需要3030秒，速度约为每秒428,000行。 要加快速度，可以使用 `Log` 引擎替换'MergeTree`引擎来创建表。 在这种情况下，下载速度超过200秒。

这个表需要使用126GB的磁盘空间。

```
:) 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                     │
└────────────────────────────────┘
```

除此之外，你还可以在MergeTree上运行OPTIMIZE查询来进行优化。但这不是必须的，因为即使在没有进行优化的情况下它的表现依然是很好的。

单台服务器运行结果

Q1:

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

0.490 seconds.

Q2:

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

1.224 seconds.

Q3:

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

2.104 seconds.

Q4:

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

3.593 seconds.

我们使用的是如下配置的服务器：

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

执行时间是取三次运行中最好的值，但是从第二次查询开始，查询就讲从文件系统的缓存中读取数据。同时在每次读取和处理后不在进行缓存。

在三台服务器中创建表结构：

在每台服务器中运行：

```
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 Gardens North' = 156, 'Springfield
Gardens South-Brookville' = 157, 'Spuyten Duyvil-Kingsbridge' = 158, 'St. Albans' = 159, 'Stapleton-Rosebank'
= 160, 'Starrett City' = 161, 'Steinway' = 162, 'Stuyvesant Heights' = 163, 'Stuyvesant Town-Cooper Village'
= 164, 'Sunset Park East' = 165, 'Sunset Park West' = 166, 'Todt Hill-Emerson Hill-Heartland Village-
Lighthouse Hill' = 167, 'Turtle Bay-East Midtown' = 168, 'University Heights-Morris Heights' = 169, 'Upper
East Side-Carnegie Hill' = 170, 'Upper West Side' = 171, 'Van Cortlandt Village' = 172, 'Van Nest-Morris
Park-Westchester Square' = 173, 'Washington Heights North' = 174, 'Washington Heights South' = 175, 'West
Brighton' = 176, 'West Concourse' = 177, 'West Farms-Bronx River' = 178, 'West New Brighton-New Brighton-St.
George' = 179, 'West Village' = 180, 'Westchester-Unionport' = 181, 'Westerleigh' = 182, 'Whitestone' = 183,
'Williamsbridge-Olinville' = 184, 'Williamsburg' = 185, 'Windsor Terrace' = 186, 'Woodhaven' = 187,
'Woodlawn-Wakefield' = 188, 'Woodside' = 189, 'Yorkville' = 190, 'park-cemetery-etc-Bronx' = 191, 'park-
cemetery-etc-Brooklyn' = 192, 'park-cemetery-etc-Manhattan' = 193, 'park-cemetery-etc-Queens' = 194, 'park-
cemetery-etc-Staten Island' = 195),  pickup_puma UInt16,  dropoff_nyct2010_gid UInt8,  dropoff_ctlabel
Float32,  dropoff_borocode UInt8,  dropoff_boroname Enum8('' = 0, 'Manhattan' = 1, 'Bronx' = 2, 'Brooklyn' =
3, 'Queens' = 4, 'Staten Island' = 5),  dropoff_ct2010 FixedString(6),  dropoff_boroct2010 FixedString(7),
dropoff_cdeligibil Enum8(' ' = 0, 'E' = 1, 'I' = 2),  dropoff_ntacode FixedString(4),  dropoff_ntaname
Enum16('' = 0, 'Airport' = 1, 'Allerton-Pelham Gardens' = 2, 'Annadale-Huguenot-Prince\'s Bay-Eltingville' =
3, 'Arden Heights' = 4, 'Astoria' = 5, 'Auburndale' = 6, 'Baisley Park' = 7, 'Bath Beach' = 8, 'Battery Park
City-Lower Manhattan' = 9, 'Bay Ridge' = 10, 'Bayside-Bayside Hills' = 11, 'Bedford' = 12, 'Bedford Park-
Fordham North' = 13, 'Bellerose' = 14, 'Belmont' = 15, 'Bensonhurst East' = 16, 'Bensonhurst West' = 17,
'Borough Park' = 18, 'Breezy Point-Belle Harbor-Rockaway Park-Broad Channel' = 19, 'Briarwood-Jamaica Hills'
= 20, 'Brighton Beach' = 21, 'Bronxdale' = 22, 'Brooklyn Heights-Cobble Hill' = 23, 'Brownsville' = 24,
'Bushwick North' = 25, 'Bushwick South' = 26, 'Cambria Heights' = 27, 'Canarsie' = 28, 'Carroll Gardens-
Columbia Street-Red Hook' = 29, 'Central Harlem North-Polo Grounds' = 30, 'Central Harlem South' = 31,
'Charleston-Richmond Valley-Tottenville' = 32, 'Chinatown' = 33, 'Claremont-Bathgate' = 34, 'Clinton' = 35,
'Clinton Hill' = 36, 'Co-op City' = 37, 'College Point' = 38, 'Corona' = 39, 'Crotona Park East' = 40, 'Crown
Heights North' = 41, 'Crown Heights South' = 42, 'Cypress Hills-City Line' = 43, 'DUMBO-Vinegar Hill-Downtown
Brooklyn-Boerum Hill' = 44, 'Douglas Manor-Douglaston-Little Neck' = 45, 'Dyker Heights' = 46, 'East
Concourse-Concourse Village' = 47, 'East Elmhurst' = 48, 'East Flatbush-Farragut' = 49, 'East Flushing' = 50,
'East Harlem North' = 51, 'East Harlem South' = 52, 'East New York' = 53, 'East New York (Pennsylvania Ave)'
= 54, 'East Tremont' = 55, 'East Village' = 56, 'East Williamsburg' = 57, 'Eastchester-Edenwald-Baychester' =
58, 'Elmhurst' = 59, 'Elmhurst-Maspeth' = 60, 'Erasmus' = 61, 'Far Rockaway-Bayswater' = 62, 'Flatbush' = 63,
'Flatlands' = 64, 'Flushing' = 65, 'Fordham South' = 66, 'Forest Hills' = 67, 'Fort Greene' = 68, 'Fresh
Meadows-Utopia' = 69, 'Ft. Totten-Bay Terrace-Clearview' = 70, 'Georgetown-Marine Park-Bergen Beach-Mill
Basin' = 71, 'Glen Oaks-Floral Park-New Hyde Park' = 72, 'Glendale' = 73, 'Gramercy' = 74, 'Grasmere-
Arrochar-Ft. Wadsworth' = 75, 'Gravesend' = 76, 'Great Kills' = 77, 'Greenpoint' = 78, 'Grymes Hill-Clifton-
Fox Hills' = 79, 'Hamilton Heights' = 80, 'Hammels-Arverne-Edgemere' = 81, 'Highbridge' = 82, 'Hollis' = 83,
'Homecrest' = 84, 'Hudson Yards-Chelsea-Flatiron-Union Square' = 85, 'Hunters Point-Sunnyside-West Maspeth' =
86, 'Hunts Point' = 87, 'Jackson Heights' = 88, 'Jamaica' = 89, 'Jamaica Estates-Holliswood' = 90,
'Kensington-Ocean Parkway' = 91, 'Kew Gardens' = 92, 'Kew Gardens Hills' = 93, 'Kingsbridge Heights' = 94,
'Laurelton' = 95, 'Lenox Hill-Roosevelt Island' = 96, 'Lincoln Square' = 97, 'Lindenwood-Howard Beach' = 98,
'Longwood' = 99, 'Lower East Side' = 100, 'Madison' = 101, 'Manhattanville' = 102, 'Marble Hill-Inwood' =
103, 'Mariner\'s Harbor-Arlington-Port Ivory-Graniteville' = 104, 'Maspeth' = 105, 'Melrose South-Mott Haven
North' = 106, 'Middle Village' = 107, 'Midtown-Midtown South' = 108, 'Midwood' = 109, 'Morningside Heights' =
110, 'Morrisania-Melrose' = 111, 'Mott Haven-Port Morris' = 112, 'Mount Hope' = 113, 'Murray Hill' = 114,
'Murray Hill-Kips Bay' = 115, 'New Brighton-Silver Lake' = 116, 'New Dorp-Midland Beach' = 117, 'New
Springville-Bloomfield-Travis' = 118, 'North Corona' = 119, 'North Riverdale-Fieldston-Riverdale' = 120,
'North Side-South Side' = 121, 'Norwood' = 122, 'Oakland Gardens' = 123, 'Oakwood-Oakwood Beach' = 124,
'Ocean Hill' = 125, 'Ocean Parkway South' = 126, 'Old Astoria' = 127, 'Old Town-Dongan Hills-South Beach' =
128, 'Ozone Park' = 129, 'Park Slope-Gowanus' = 130, 'Parkchester' = 131, 'Pelham Bay-Country Club-City
Island' = 132, 'Pelham Parkway' = 133, 'Pomonok-Flushing Heights-Hillcrest' = 134, 'Port Richmond' = 135,
'Prospect Heights' = 136, 'Prospect Lefferts Gardens-Wingate' = 137, 'Queens Village' = 138, 'Queensboro Hill'
= 139, 'Queensbridge-Ravenswood-Long Island City' = 140, 'Rego Park' = 141, 'Richmond Hill' = 142,
'Ridgewood' = 143, 'Rikers Island' = 144, 'Rosedale' = 145, 'Rossville-Woodrow' = 146, 'Rugby-Remsen Village'
= 147, 'Schuylerville-Throgs Neck-Edgewater Park' = 148, 'Seagate-Coney Island' = 149, 'Sheepshead Bay-
Gerritsen Beach-Manhattan Beach' = 150, 'SoHo-TriBeCa-Civic Center-Little Italy' = 151, 'Soundview-Bruckner'
= 152, 'Soundview-Castle Hill-Clason Point-Harding Park' = 153, 'South Jamaica' = 154, 'South Ozone Park' =
155, 'Springfield Gardens North' = 156, 'Springfield Gardens South-Brookville' = 157, 'Spuyten Duyvil-
Kingsbridge' = 158, 'St. Albans' = 159, 'Stapleton-Rosebank' = 160, 'Starrett City' = 161, 'Steinway' = 162,
'Stuyvesant Heights' = 163, 'Stuyvesant Town-Cooper Village' = 164, 'Sunset Park East' = 165, 'Sunset Park
West' = 166, 'Todt Hill-Emerson Hill-Heartland Village-Lighthouse Hill' = 167, 'Turtle Bay-East Midtown' =
168, 'University Heights-Morris Heights' = 169, 'Upper East Side-Carnegie Hill' = 170, 'Upper West Side' =
171, 'Van Cortlandt Village' = 172, 'Van Nest-Morris Park-Westchester Square' = 173, 'Washington Heights
North' = 174, 'Washington Heights South' = 175, 'West Brighton' = 176, 'West Concourse' = 177, 'West Farms-
Bronx River' = 178, 'West New Brighton-New Brighton-St. George' = 179, 'West Village' = 180, 'Westchester-
Unionport' = 181, 'Westerleigh' = 182, 'Whitestone' = 183, 'Williamsbridge-Olinville' = 184, 'Williamsburg' =
185, 'Windsor Terrace' = 186, 'Woodhaven' = 187, 'Woodlawn-Wakefield' = 188, 'Woodside' = 189, 'Yorkville' =
190, 'park-cemetery-etc-Bronx' = 191, 'park-cemetery-etc-Brooklyn' = 192, 'park-cemetery-etc-Manhattan' =
193, 'park-cemetery-etc-Queens' = 194, 'park-cemetery-etc-Staten Island' = 195),  dropoff_puma UInt16) ENGINE
= MergeTree(pickup_date, pickup_datetime, 8192)
```

在之前的服务器中运行：

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

运行如下查询重新分布数据：

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

这个查询需要运行2454秒。

在三台服务器集群中运行的结果：

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

不出意料，查询是线性扩展的。

我们同时在140台服务器的集群中运行的结果：

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

在这种情况下，查询处理时间首先由网络延迟确定。 我们使用位于芬兰的Yandex数据中心中的客户端去位于俄罗斯的集群上运行查询，这增加了大约20毫秒的延迟。

总结

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

AMPLab 大数据基准测试

参考 https://amplab.cs.berkeley.edu/benchmark/ [https://amplab.cs.berkeley.edu/benchmark/]

需要您在https://aws.amazon.com [https://aws.amazon.com]注册一个免费的账号。注册时需要您提供信用卡、邮箱、电话等信息。之后可以在https://console.aws.amazon.com/iam/home?nc2=h_m_sc#security_credential [https://console.aws.amazon.com/iam/home?nc2=h_m_sc#security_credential]获取新的访问密钥

在控制台运行以下命令：

```
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 ..
```

在ClickHouse运行如下查询：

```
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);
```

回到控制台运行如下命令：

```
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
```

简单的查询示例:

```
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
```

维基访问数据

参考: http://dumps.wikimedia.org/other/pagecounts-raw/ [http://dumps.wikimedia.org/other/pagecounts-raw/]

创建表结构:

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

加载数据:

```
for i in {2007..2016}; do for j in {01..12}; do echo $i-$j >&2; curl -sSL "http://dumps.wikimedia.org/other/pagecounts-raw/$i/$i-$j/" | grep -oE 'pagecounts-[0-9]+-[0-9]+\.gz'; done; done | sort | uniq | tee links.txt
cat links.txt | while read link; do wget http://dumps.wikimedia.org/other/pagecounts-raw/$(echo $link | sed -r 's/pagecounts-([0-9]{4})([0-9]{2})[0-9]{2}-[0-9]+\.gz/\1/')/$(echo $link | sed -r 's/pagecounts-([0-9]{4})([0-9]{2})[0-9]{2}-[0-9]+\.gz/\1-\2/')/$link; done
ls -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
```

Criteo TB级别点击日志

可以从http://labs.criteo.com/downloads/download-terabyte-click-logs/ [http://labs.criteo.com/downloads/download-terabyte-click-logs/]上下载数据

创建原始数据对应的表结构：

```
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
```

下载数据：

```
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
```

创建转换后的数据对应的表结构：

```
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)
```

将第一张表中的原始数据转化写入到第二张表中去：

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

DROP TABLE criteo_log;
```

Star Schema 基准测试

编译 dbgen: https://github.com/vadimtk/ssb-dbgen [https://github.com/vadimtk/ssb-dbgen]

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

在编译过程中可能会有一些警告，这是正常的。

将 `dbgen` 和 `dists.dss` 放在一个可用容量大于800GB的磁盘中。

开始生成数据:

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

在ClickHouse中创建表结构:

```sql
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());
```

如果是在单节点中进行的测试，那么只需要创建对应的MergeTree表。 如果是在多节点中进行的测试，您需要在配置文件中配置 `perftest_3shards_1replicas` 集群的信息。 然后在每个节点中同时创建MergeTree表和Distributed表。

下载数据（如果您是分布式测试的话将'customer'更改为'customerd'）：

```bash
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"
```

[]

客户端

ClickHouse提供了两个网络接口（两者都可以选择包装在TLS中以提高安全性）：

- HTTP [#http]，记录在案，易于使用.
- 本地人TCP [#tcp]，这有较少的开销.

在大多数情况下，建议使用适当的工具或库，而不是直接与这些工具或库进行交互。 Yandex的官方支持如下： *命令行客户端 [#cli]* JDBC驱动程序 [#jdbc] * ODBC驱动程序 [#odbc]

还有许多第三方库可供使用ClickHouse： *客户端库 [#third-party/client_libraries]* 集成 [#third-party/integrations] * 可视界面 [#third-party/gui]

命令行客户端

通过命令行来访问 ClickHouse，您可以使用 `clickhouse-client`

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

该客户端支持命令行参数以及配置文件。查看更多，请看 "配置 [#interfaces_cli_configuration]"

使用方式

这个客户端可以选择使用交互式与非交互式（批量）两种模式。使用批量模式，要指定 `query` 参数，或者发送数据到 `stdin` （它会检查 `stdin` 是否是 Terminal），或者两种同时使用。 它与 HTTP 接口很相似，当使用 `query` 参数发送数据到 `stdin` 时，客户端请求就是一行一行的 `stdin` 输入作为 `query` 的参数。这种方式在大规模的插入请求中非常方便。

使用这个客户端插入数据的示例：

```
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";
```

在批量模式中，默认的数据格式是 `TabSeparated` 分隔的。您可以根据查询来灵活设置 FORMAT 格式。

默认情况下，在批量模式中只能执行单个查询。为了从一个 Script 中执行多个查询，可以使用 `--multiquery` 参数。除了 INSERT 请求外，这种方式在任何地方都有用。查询的结果会连续且不含分隔符地输出。 同样的，为了执行大规模的查询，您可以为每个查询执行一次 `clickhouse-client` 。但注意到每次启动 `clickhouse-client` 程序都需要消耗几十毫秒时间。

在交互模式下，每条查询过后，你可以直接输入下一条查询命令。

如果 `multiline` 没有指定（默认没指定）：为了执行查询，按下 Enter 即可。查询语句不是必须使用分号结尾。如果需要写一个多行的查询语句，可以在换行之前输入一个反斜杠 `\` ，然后在您按下 Enter 键后，您就可以输入当前语句的下一行查询了。

如果 `multiline` 指定了：为了执行查询，需要以分号结尾并且按下 Enter 键。如果行末没有分号，将认为当前语句并没有输入完而要求继续输入下一行。

若只运行单个查询，分号后面的所有内容都会被忽略。

您可以指定 `\G` 来替代分号或者在分号后面，这表示 `Vertical` 的格式。在这种格式下，每一个值都会打印在不同的行中，这种方式对于宽表来说很方便。这个不常见的特性是为了兼容 MySQL 命令而加的。

命令行客户端是基于 `readline` 库（`history` 库或者 `libedit` 库, 或不基于其他库, 这取决于客户端是如何编译的）。换句话说，它可以使用我们熟悉的快捷键方式来操作以及保留历史命令。 历史命令会写入在 `~/.clickhouse-client-history` 中。

默认情况下，输出的格式是 `PrettyCompact` 。您可以通过 FORMAT 设置根据不同查询来修改格式，或者通过在查询末尾指定

`\G` 字符，或通过在命令行中使用 `--format` or `--vertical` 参数，或使用客户端的配置文件。

若要退出客户端，使用 Ctrl+D （或 Ctrl+C），或者输入以下其中一个命令： `exit` , `quit` , `logout` , `учше` , `йгше` , `дщпщгеу` , `exit;` , `quit;` , `logout;` , `учшеж` , `йгшеж` , `дщпщгеж` , `q` , `й` , `q` , `Q` , `:q` , `й` , `Й` , `Жй`

当执行一个查询的时候，客户端会显示：

1. 进度, 进度会每秒更新十次 （默认情况下） 。 对于很快的查询，进度可能没有时间显示。
2. 为了调试会显示解析且格式化后的查询语句。
3. 指定格式的输出结果。
4. 输出结果的行数的行数，经过的时间，以及查询处理的速度。

您可以通过 Ctrl+C 来取消一个长时间的查询。然而，您依然需要等待服务端来中止请求。在某个阶段去取消查询是不可能的。如果您不等待并再次按下 Ctrl + C，客户端将会退出。

命令行客户端允许通过外部数据 （外部临时表） 来查询。更多相关信息，请参考 "外部数据查询处理 [#external-data]".

[]

配置

您可以通过以下方式传入参数到 `clickhouse-client` 中 （所有的参数都有默认值） ：

- 通过命令行

  命令行参数会覆盖默认值和配置文件的配置。

- 配置文件

  配置文件的配置会覆盖默认值

命令行参数

- `--host, -h` -– 服务端的 host 名称, 默认是 'localhost'。 您可以选择使用 host 名称或者 IPv4 或 IPv6 地址。
- `--port` – 连接的端口，默认值： 9000。注意 HTTP 接口以及 TCP 原生接口是使用不同端口的。
- `--user, -u` – 用户名。 默认值： default。
- `--password` – 密码。 默认值： 空字符串。
- `--query, -q` – 非交互模式下的查询语句.
- `--database, -d` – 默认当前操作的数据库. 默认值： 服务端默认的配置 （默认是 `default` ）。
- `--multiline, -m` – 如果指定，允许多行语句查询（Enter 仅代表换行，不代表查询语句完结）。
- `--multiquery, -n` – 如果指定, 允许处理用逗号分隔的多个查询，只在非交互模式下生效。
- `--format, -f` – 使用指定的默认格式输出结果。
- `--vertical, -E` – 如果指定，默认情况下使用垂直格式输出结果。这与 '--format=Vertical' 相同。在这种格式中，每个值都在单独的行上打印，这种方式对显示宽表很有帮助。
- `--time, -t` – 如果指定，非交互模式下会打印查询执行的时间到 'stderr' 中。
- `--stacktrace` – 如果指定，如果出现异常，会打印堆栈跟踪信息。
- `-config-file` – 配置文件的名称。

配置文件

`clickhouse-client` 使用一下第一个存在的文件：

- 通过 `-config-file` 参数指定的文件.
- `./clickhouse-client.xml`

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

配置文件示例:

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

原生客户端接口（TCP）

本机协议用于命令行客户端 [#cli]，用于分布式查询处理期间的服务器间通信，以及其他C ++程序。 不幸的是，本机ClickHouse协议还没有正式的规范，但它可以从ClickHouse源代码进行逆向工程从这里开始 [https://github.com/yandex/ClickHouse/tree/master/dbms/src/Client]）和/或拦截和分析TCP流量。

HTTP 客户端

HTTP 接口可以让你通过任何平台和编程语言来使用 ClickHouse。我们用 Java 和 Perl 以及 shell 脚本来访问它。在其他的部门中，HTTP 接口会用在 Perl，Python 以及 Go 中。HTTP 接口比 TCP 原生接口更为局限，但是却有更好的兼容性。

默认情况下，clickhouse-server 会在端口 8123 上监控 HTTP 请求（这可以在配置中修改）。 如果你发送了一个不带参数的 GET 请求，它会返回一个字符串 "Ok"（结尾有换行）。可以将它用在健康检查脚本中。

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

通过 URL 中的 `query` 参数来发送请求，或者发送 POST 请求，或者将查询的开头部分放在 URL 的 `query` 参数中，其他部分放在 POST 中（我们会在后面解释为什么这样做是有必要的）。URL 的大小会限制在 16 KB，所以发送大型查询时要时刻记住这点。

如果请求成功，将会收到 200 的响应状态码和响应主体中的结果。 如果发生了某个异常，将会收到 500 的响应状态码和响应主体中的异常描述信息。

当使用 GET 方法请求时， `readonly` 会被设置。换句话说，若要作修改数据的查询，只能发送 POST 方法的请求。可以将查询通过 POST 主体发送，也可以通过 URL 参数发送。

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
```

可以看到，curl 命令由于空格需要 URL 转义，所以不是很方便。尽管 wget 命令对url做了 URL 转义，但我们并不推荐使用他，因为在 HTTP 1.1 协议下使用 keep-alive 和 Transfer-Encoding: chunked 头部设置它并不能很好的工作。

```
$ 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
```

如果一部分请求是通过参数发送的，另外一部分通过 POST 主体发送，两部分查询之间会一行空行插入。错误示例：

```
$ 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
```

默认情况下，返回的数据是 TabSeparated 格式的，更多信息，见 "[数据格式]" 部分。 可以使用 FORMAT 设置查询来请求不同格式。

```
$ echo 'SELECT 1 FORMAT Pretty' | curl 'http://localhost:8123/?' --data-binary @-
┏━━━┓
┃ 1 ┃
┡━━━┩
│ 1 │
└───┘
```

INSERT 必须通过 POST 方法来插入数据。这种情况下，你可以将查询的开头部分放在 URL 参数中，然后用 POST 主体传入插入的数据。插入的数据可以是，举个例子，从 MySQL 导出的以 tab 分割的数据。在这种方式中，INSERT 查询取代了 LOAD DATA LOCAL INFILE from MySQL。

示例: 创建一个表:

```
echo 'CREATE TABLE t (a UInt8) ENGINE = Memory' | POST 'http://localhost:8123/'
```

使用类似 INSERT 的查询来插入数据：

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

数据可以从查询中单独发送：

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

可以指定任何数据格式。值的格式和写入表 `t` 的值的格式相同：

```
echo '(7),(8),(9)' | POST 'http://localhost:8123/?query=INSERT INTO t FORMAT Values'
```

若要插入 tab 分割的数据，需要指定对应的格式：

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

从表中读取内容。由于查询处理是并行的，数据以随机顺序输出。

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

删除表。

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

成功请求后并不会返回数据，返回一个空的响应体。

可以通过压缩来传输数据。压缩的数据没有一个标准的格式，但你需要指定一个压缩程序来使用它(sudo apt-get install compressor-metrika-yandex）。

如果在 URL 中指定了 `compress=1` ，服务会返回压缩的数据。 如果在 URL 中指定了 `decompress=1` ，服务会解压通过 POST 方法发送的数据。

可以通过为每份数据进行立即压缩来减少大规模数据传输中的网络压力。

可以指定 'database' 参数来指定默认的数据库。

```
$ 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
```

默认情况下，默认数据库会在服务的配置中注册，默认是 `default` 。或者，也可以在表名之前使用一个点来指定数据库。

用户名密码可以通过以下两种方式指定：

1. 通过 HTTP Basic Authentication。示例：

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

1. 通过 URL 参数 中的 'user' 和 'password'。示例：

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

如果用户名没有指定，默认的用户是 `default` 。如果密码没有指定，默认会使用空密码。 可以使用 URL 参数指定配置或者设置整个配置文件来处理单个查询。示例： `http://localhost:8123/?profile=web&max_rows_to_read=1000000000&query=SELECT+1`

更多信息，参见 "设置 [#settings]" 部分。

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

更多关于其他参数的信息，参见 "设置 [#settings]" 部分。

相比起 TCP 原生接口，HTTP 接口不支持会话和会话设置的概念，不允许中止查询（准确地说，只在少数情况下允许），不显示查询处理的进展。执行解析和数据格式化都是在服务端处理，网络上会比 TCP 原生接口更低效。

可选的 `query_id` 参数可能当做 query ID 传入（或者任何字符串）。更多信息，参见 "设置 replace_running_query [#replace-running-query]" 部分。

可选的 `quota_key` 参数可能当做 quota key 传入（或者任何字符串）。更多信息，参见 "配额 [#quotas]" 部分。

HTTP 接口允许传入额外的数据（外部临时表）来查询。更多信息，参见 "外部数据查询处理 [#external-data]" 部分。

响应缓冲

可以在服务器端启用响应缓冲。提供了 `buffer_size` 和 `wait_end_of_query` 两个URL 参数来达此目的。

`buffer_size` 决定了查询结果要在服务内存中缓冲多少个字节数据. 如果响应体比这个阈值大，缓冲区会写入到 HTTP 管道，剩下的数据也直接发到 HTTP 管道中。

为了确保整个响应体被缓冲，可以设置 `wait_end_of_query=1` 。这种情况下，存入内存的数据会被缓冲到服务端的一个临时文件中。

示例:

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

查询请求响应状态码和 HTTP 头被发送到客户端后，若发生查询处理出错，使用缓冲区可以避免这种情况的发生。在这种情况下，响应主体的结尾会写入一条错误消息，而在客户端，只能在解析阶段检测到该错误。

[]

输入输出格式

ClickHouse 可以接受多种数据格式，可以在 (`INSERT`) 以及 (`SELECT`) 请求中使用。

下列表格列出了支持的数据格式以及在 (`INSERT`) 以及 (`SELECT`) 请求中使用它们的方式。

格式	INSERT	SELECT
TabSeparated [#tabseparated]	✔	✔
TabSeparatedRaw [#tabseparatedraw]	✗	✔
TabSeparatedWithNames [#tabseparatedwithnames]	✔	✔
TabSeparatedWithNamesAndTypes [#tabseparatedwithnamesandtypes]	✔	✔
CSV [#csv]	✔	✔
CSVWithNames [#csvwithnames]	✔	✔
Values [#values]	✔	✔
Vertical [#vertical]	✗	✔
VerticalRaw [#verticalraw]	✗	✔
JSON [#json]	✗	✔
JSONCompact [#jsoncompact]	✗	✔
JSONEachRow [#jsoneachrow]	✔	✔
TSKV [#tskv]	✔	✔
Pretty [#pretty]	✗	✔
PrettyCompact [#prettycompact]	✗	✔
PrettyCompactMonoBlock [#prettycompactmonoblock]	✗	✔
PrettyNoEscapes [#prettynoescapes]	✗	✔
PrettySpace [#prettyspace]	✗	✔
RowBinary [#rowbinary]	✔	✔
Native [#native]	✔	✔
Null [#null]	✗	✔
XML [#xml]	✗	✔
CapnProto [#capnproto]	✔	✔

[]

TabSeparated

在 TabSeparated 格式中，数据按行写入。每行包含由制表符分隔的值。除了行中的最后一个值（后面紧跟换行符）之外，每个值都跟随一个制表符。 在任何地方都可以使用严格的 Unix 命令行。最后一行还必须在最后包含换行符。值以文本格式编写，不包含引号，并且要转义特殊字符。

这种格式也可以用 `TSV` 来表示。

TabSeparated 格式非常方便用于自定义程序或脚本处理数据。HTTP 客户端接口默认会用这种格式，命令行客户端批量模式下也会用这种格式。这种格式允许在不同数据库之间传输数据。例如，从 MYSQL 中导出数据然后导入到 ClickHouse 中，反之亦然。

TabSeparated 格式支持输出数据总值（当使用 WITH TOTALS） 以及极值（当 'extremes' 设置是1）。这种情况下，总值和极值输出在主数据的后面。主要的数据，总值，极值会以一个空行隔开，例如：

```
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
```

数据解析方式

整数以十进制形式写入。数字在开头可以包含额外的 `+` 字符（解析时忽略，格式化时不记录）。非负数不能包含负号。 读取时，允许将空字符串解析为零，或者（对于带符号的类型）将仅包含负号的字符串解析为零。 不符合相应数据类型的数字可能会被解析为不同的数字，而不会显示错误消息。

浮点数以十进制形式写入。点号用作小数点分隔符。支持指数等符号，如'inf'，'+ inf'，'-inf'和'nan'。 浮点数的输入可以以小数点开始或结束。 格式化的时候，浮点数的精确度可能会丢失。 解析的时候，没有严格需要去读取与机器可以表示的最接近的数值。

日期会以 YYYY-MM-DD 格式写入和解析，但会以任何字符作为分隔符。 带时间的日期会以 YYYY-MM-DD hh:mm:ss 格式写入和解析，但会以任何字符作为分隔符。 这一切都发生在客户端或服务器启动时的系统时区（取决于哪一种格式的数据）。对于具有时间的日期，夏时制时间未指定。 因此，如果转储在夏令时中有时间，则转储不会明确地匹配数据，解析将选择两者之一。 在读取操作期间，不正确的日期和具有时间的日期可以使用自然溢出或空日期和时间进行分析，而不会出现错误消息。

有个例外情况，Unix 时间戳格式（10个十进制数字）也支持使用时间解析日期。结果不是时区相关的。格式 YYYY-MM-DD hh:mm:ss和 NNNNNNNNNN 会自动区分。

字符串以反斜线转义的特殊字符输出。 以下转义序列用于输出： `\b`， `\f`， `\r`， `\n`， `\t`， `\0`， `\'`， `\\`。 解析还支持 `\a`， `\v` 和 `\xHH` （十六进制转义字符）和任何 `\c` 字符，其中 `c` 是任何字符（这些序列被转换为 `c`）。 因此，读取数据支持可以将换行符写为 `\n` 或 `\` 的格式，或者换行。例如，字符串 `Hello world` 在单词之间换行而不是空格可以解析为以下任何形式：

```
Hello\nworld

Hello\
world
```

第二种形式是支持的，因为 MySQL 读取 tab-separated 格式数据集的时候也会使用它。

在 TabSeparated 格式中传递数据时需要转义的最小字符集为：Tab，换行符（LF）和反斜杠。

只有一小组符号会被转义。你可以轻易地找到一个字符串值，但这不会正常在你的终端显示。

数组写在方括号内的逗号分隔值列表中。 通常情况下，数组中的数字项目会被拼凑，但日期，带时间的日期以及字符串将使用与上面相同的转义规则用单引号引起来。

NULL [#null-literal] 将输出为 `\N` 。

[]

TabSeparatedRaw

与 `TabSeparated` 格式不一样的是，行数据是不会被转义的。 该格式仅适用于输出查询结果，但不适用于解析输入（将数据插入到表中） 。

这种格式也可以使用名称 `TSVRaw` 来表示。[]

TabSeparatedWithNames

与 `TabSeparated` 格式不一样的是，第一行会显示列的名称。 在解析过程中，第一行完全被忽略。您不能使用列名来确定其位置或检查其正确性。 （未来可能会加入解析头行的功能）

这种格式也可以使用名称 `TSVWithNames` 来表示。[]

TabSeparatedWithNamesAndTypes

与 `TabSeparated` 格式不一样的是，第一行会显示列的名称，第二行会显示列的类型。 在解析过程中，第一行和第二行完全被忽略。

这种格式也可以使用名称 `TSVWithNamesAndTypes` 来表示。[]

TSKV

与 `TabSeparated` 格式类似，但它输出的是 `name=value` 的格式。名称会和 `TabSeparated` 格式一样被转义， `=` 字符也会被转义。

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

NULL [#null-literal] 输出为 `\N` 。

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

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

当有大量的小列时，这种格式是低效的，通常没有理由使用它。它被用于 Yandex 公司的一些部门。

数据的输出和解析都支持这种格式。对于解析，任何顺序都支持不同列的值。可以省略某些值，用 `-` 表示， 它们被视为等于它们的默认值。在这种情况下，零和空行被用作默认值。作为默认值，不支持表中指定的复杂值。

对于不带等号或值，可以用附加字段 `tskv` 来表示，这种在解析上是被允许的。这样的话该字段被忽略。[]

CSV

按逗号分隔的数据格式(RFC [https://tools.ietf.org/html/rfc4180])。

格式化的时候，行是用双引号括起来的。字符串中的双引号会以两个双引号输出，除此之外没有其他规则来做字符转义了。日期和时间也会以双引号包括。数字的输出不带引号。值由一个单独的字符隔开，这个字符默认是 `,`。行使用 Unix 换行符（LF）分隔。 数组序列化成 CSV 规则如下：首先将数组序列化为 TabSeparated 格式的字符串，然后将结果字符串用双引号包括输出到 CSV。CSV 格式的元组被序列化为单独的列（即它们在元组中的嵌套关系会丢失）。

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

*默认情况下间隔符是 `,` ，在format_csv_delimiter [#format_csv_delimiter]中可以了解更多间隔符配置。

解析的时候，可以使用或不使用引号来解析所有值。支持双引号和单引号。行也可以不用引号排列。 在这种情况下，它们被解析为逗号或换行符（CR 或 LF）。在解析不带引号的行时，若违反 RFC 规则，会忽略前导和尾随的空格和制表符。 对于换行，全部支持 Unix（LF），Windows（CR LF）和 Mac OS Classic（CR LF）。

`NULL` 将输出为 `\N` 。

CSV 格式是和 TabSeparated 一样的方式输出总数和极值。

CSVWithNames

会输出带头部行，和 `TabSeparatedWithNames` 一样。[]

JSON

以 JSON 格式输出数据。除了数据表之外，它还输出列名称和类型以及一些附加信息：输出行的总数以及在没有 LIMIT 时可以输出的行数。 例：

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

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

	"data":
	[
		{
			"SearchPhrase": "",
			"c": "8267016"
		},
		{
			"SearchPhrase": "bathroom interior design",
			"c": "2166"
		},
		{
			"SearchPhrase": "yandex",
			"c": "1655"
		},
		{
			"SearchPhrase": "spring 2014 fashion",
			"c": "1549"
		},
		{
			"SearchPhrase": "freeform photos",
			"c": "1480"
		}
	],

	"totals":
	{
		"SearchPhrase": "",
		"c": "8873898"
	},

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

	"rows": 5,

	"rows_before_limit_at_least": 141137
}
```

JSON 与 JavaScript 兼容。为了确保这一点，一些字符被另外转义：斜线 `/` 被转义为 `\/`；替代的换行符 `U+2028` 和 `U+2029` 会打断一些浏览器解析，它们会被转义为 `\uXXXX` 。 ASCII 控制字符被转义：退格，换页，换行，回车和水平制表符被替换为 `\b`， `\f`， `\n`， `\r`， `\t` 作为使用 `\uXXXX` 序列的00-1F范围内的剩余字节。 无效的 UTF-8 序列更改为替换字符 ，因此输出文本将包含有效的 UTF-8 序列。 为了与 JavaScript 兼容，默认情况下，Int64 和 UInt64 整数用双引号引起来。要除去引号，可以将配置参数 output_format_json_quote_64bit_integers 设置为0。

`rows` – 结果输出的行数。

`rows_before_limit_at_least` 去掉 LIMIT 过滤后的最小行总数。 只会在查询包含 LIMIT 条件时输出。 若查询包含 GROUP BY，rows_before_limit_at_least 就是去掉 LIMIT 后过滤后的准确行数。

`totals` – 总值 （当使用 TOTALS 条件时）。

`extremes` – 极值 （当 extremes 设置为 1时）。

该格式仅适用于输出查询结果，但不适用于解析输入（将数据插入到表中）。

ClickHouse 支持 NULL [#null-literal], 在 JSON 格式中以 `null` 输出来表示.

参考 JSONEachRow 格式。

[]

JSONCompact

与 JSON 格式不同的是它以数组的方式输出结果，而不是以结构体。

示例：

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

        "data":
        [
                ["", "8267016"],
                ["bathroom interior design", "2166"],
                ["yandex", "1655"],
                ["fashion trends spring 2014", "1549"],
                ["freeform photo", "1480"]
        ],

        "totals": ["","8873898"],

        "extremes":
        {
                "min": ["","1480"],
                "max": ["","8267016"]
        },

        "rows": 5,

        "rows_before_limit_at_least": 141137
}
```

这种格式仅仅适用于输出结果集，而不适用于解析（将数据插入到表中）。参考 `JSONEachRow` 格式。[]

JSONEachRow

将数据结果每一行以 JSON 结构体输出（换行分割 JSON 结构体）。

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

与 JSON 格式不同的是，没有替换无效的UTF-8序列。任何一组字节都可以在行中输出。这是必要的，因为这样数据可以被格式化而不会丢失任何信息。值的转义方式与JSON相同。

对于解析，任何顺序都支持不同列的值。可以省略某些值 - 它们被视为等于它们的默认值。在这种情况下，零和空行被用作默认值。 作为默认值，不支持表中指定的复杂值。元素之间的空白字符被忽略。如果在对象之后放置逗号，它将被忽略。对象不一定必须用新行分隔。[]

Native

最高性能的格式。 据通过二进制格式的块进行写入和读取。对于每个块，该块中的行数，列数，列名称和类型以及列的部分将被相继记录。 换句话说，这种格式是 “列式”的 - 它不会将列转换为行。 这是用于在服务器之间进行交互的本地界面中使用的格式，用于使用命令行客户端和 C++ 客户端。

您可以使用此格式快速生成只能由 ClickHouse DBMS 读取的格式。但自己处理这种格式是没有意义的。[]

Null

没有输出。但是，查询已处理完毕，并且在使用命令行客户端时，数据将传输到客户端。这仅用于测试，包括生产力测试。显然，这种格式只适用于输出，不适用于解析。[]

Pretty

将数据以表格形式输出，也可以使用 ANSI 转义字符在终端中设置颜色。 它会绘制一个完整的表格，每行数据在终端中占用两行。 每一个结果块都会以单独的表格输出。这是很有必要的，以便结果块不用缓冲结果输出（缓冲在可以预见结果集宽度的时候是很有必要的）。

NULL [#null-literal] 输出为 `NULL` 。

```
SELECT * FROM t_null
```

```
┌─x─┬────y─┐
│ 1 │ NULL │
└───┴──────┘
```

为避免将太多数据传输到终端，只打印前10,000行。 如果行数大于或等于10,000，则会显示消息“Showed first 10 000”。 该格式仅适用于输出查询结果，但不适用于解析输入（将数据插入到表中）。

Pretty格式支持输出总值（当使用 WITH TOTALS 时）和极值（当 `extremes` 设置为1时）。 在这些情况下，总数值和极值在主数据之后以单独的表格形式输出。 示例（以 PrettyCompact 格式显示）：

```
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

与 `Pretty` 格式不一样的是， `PrettyCompact` 去掉了行之间的表格分割线，这样使得结果更加紧凑。这种格式会在交互命令行客户端下默认使用。[]

PrettyCompactMonoBlock

与 `PrettyCompact` 格式不一样的是，它支持 10,000 行数据缓冲，然后输出在一个表格中，不会按照块来区分 []

PrettyNoEscapes

与 `Pretty` 格式不一样的是，它不使用 ANSI 字符转义， 这在浏览器显示数据以及在使用 `watch` 命令行工具是有必要的。

示例：

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

您可以使用 HTTP 接口来获取数据，显示在浏览器中。

PrettyCompactNoEscapes

用法类似上述。

PrettySpaceNoEscapes

用法类似上述。[]

PrettySpace

与 `PrettyCompact` (#prettycompact) 格式不一样的是，它使用空格来代替网格来显示数据。[]

RowBinary

以二进制格式逐行格式化和解析数据。行和值连续列出，没有分隔符。 这种格式比 Native 格式效率低，因为它是基于行的。

整数使用固定长度的小端表示法。 例如，UInt64 使用8个字节。 DateTime 被表示为 UInt32 类型的Unix 时间戳值。 Date 被表示为 UInt16 对象，它的值为 1970-01-01以来的天数。 字符串表示为 varint 长度（无符号LEB128

[https://en.wikipedia.org/wiki/LEB128]），后跟字符串的字节数。 FixedString 被简单地表示为一个字节序列。

数组表示为 varint 长度（无符号LEB128 [https://en.wikipedia.org/wiki/LEB128]），后跟有序的数组元素。

对于 NULL [#null-literal] 的支持， 一个为 1 或 0 的字节会加在每个 Nullable [#data_type-nullable] 值前面。如果为 1, 那么该值就是 `NULL` 。 如果为 0，则不为 `NULL` 。

Values

在括号中打印每一行。行由逗号分隔。最后一行之后没有逗号。括号内的值也用逗号分隔。数字以十进制格式输出，不含引号。 数组以方括号输出。带有时间的字符串，日期和时间用引号包围输出。转义字符的解析规则与 TabSeparated [#tabseparated] 格式类似。 在格式化过程中，不插入额外的空格，但在解析过程中，空格是被允许并跳过的（除了数组值之外的空格，这是不允许的）。 NULL [#null-literal] 为 `NULL` 。

以 Values 格式传递数据时需要转义的最小字符集是：单引号和反斜线。

这是 `INSERT INTO t VALUES ...` 中可以使用的格式，但您也可以将其用于查询结果。

[]

Vertical

使用指定的列名在单独的行上打印每个值。如果每行都包含大量列，则此格式便于打印一行或几行。

NULL [#null-literal] 输出为 `NULL` 。

示例:

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

```
Row 1:
──────
x: 1
y: NULL
```

该格式仅适用于输出查询结果，但不适用于解析输入（将数据插入到表中）。

[]

VerticalRaw

和 `Vertical` 格式不同点在于，行是不会被转义的。 这种格式仅仅适用于输出，但不适用于解析输入（将数据插入到表中）。

示例:

```
:) SHOW CREATE TABLE geonames FORMAT VerticalRaw;
Row 1:
──────
statement: CREATE TABLE default.geonames ( geonameid UInt32, date Date DEFAULT CAST('2017-12-08' AS Date))
ENGINE = MergeTree(date, geonameid, 8192)

:) SELECT 'string with \'quotes\' and \t with some special \n characters' AS test FORMAT VerticalRaw;
Row 1:
──────
test: string with 'quotes' and 	 with some special
 characters
```

和 Vertical 格式相比：

```
:) SELECT 'string with \'quotes\' and \t with some special \n characters' AS test FORMAT Vertical;
Row 1:
──────
test: string with \'quotes\' and \t with some special \n characters
```

[]

XML

该格式仅适用于输出查询结果，但不适用于解析输入，示例：

```
<?xml version='1.0' encoding='UTF-8' ?>
<result>
        <meta>
                <columns>
                        <column>
                                <name>SearchPhrase</name>
                                <type>String</type>
                        </column>
                        <column>
                                <name>count()</name>
                                <type>UInt64</type>
                        </column>
                </columns>
        </meta>
        <data>
                <row>
                        <SearchPhrase></SearchPhrase>
                        <field>8267016</field>
                </row>
                <row>
                        <SearchPhrase>bathroom interior design</SearchPhrase>
                        <field>2166</field>
                </row>
                <row>
                        <SearchPhrase>yandex</SearchPhrase>
                        <field>1655</field>
                </row>
                <row>
                        <SearchPhrase>2014 spring fashion</SearchPhrase>
                        <field>1549</field>
                </row>
                <row>
                        <SearchPhrase>freeform photos</SearchPhrase>
                        <field>1480</field>
                </row>
                <row>
                        <SearchPhrase>angelina jolie</SearchPhrase>
                        <field>1245</field>
                </row>
                <row>
                        <SearchPhrase>omsk</SearchPhrase>
                        <field>1112</field>
                </row>
                <row>
                        <SearchPhrase>photos of dog breeds</SearchPhrase>
                        <field>1091</field>
                </row>
                <row>
                        <SearchPhrase>curtain designs</SearchPhrase>
                        <field>1064</field>
                </row>
                <row>
                        <SearchPhrase>baku</SearchPhrase>
                        <field>1000</field>
                </row>
        </data>
        <rows>10</rows>
        <rows_before_limit_at_least>141137</rows_before_limit_at_least>
</result>
```

如果列名称没有可接受的格式，则仅使用 field 作为元素名称。 通常，XML 结构遵循 JSON 结构。 就像JSON一样，将无效的 UTF-8 字符都作替换，以便输出文本将包含有效的 UTF-8 字符序列。

在字符串值中，字符 < 和 & 被转义为 < 和 & 。

数组输出为 <array> <elem> Hello </ elem> <elem> World </ elem> ... </ array>，元组输出为 <tuple> <elem> Hello </ elem> <elem> World </ ELEM> ... </tuple> 。

[]

CapnProto

Cap'n Proto 是一种二进制消息格式，类似 Protocol Buffers 和 Thriftis，但与 JSON 或 MessagePack 格式不一样。

Cap'n Proto 消息格式是严格类型的，而不是自我描述，这意味着它们不需要外部的描述。这种格式可以实时地应用，并针对每个查询进行缓存。

```
SELECT SearchPhrase, count() AS c FROM test.hits
       GROUP BY SearchPhrase FORMAT CapnProto SETTINGS schema = 'schema:Message'
```

其中 schema.capnp 描述如下：

```
struct Message {
  SearchPhrase @0 :Text;
  c @1 :Uint64;
}
```

格式文件存储的目录可以在服务配置中的 format_schema_path [#server_settings-format_schema_path] 指定。

Cap'n Proto 反序列化是很高效的，通常不会增加系统的负载。

JDBC 驱动

- ClickHouse官方有 JDBC 的驱动。 见这里 [https://github.com/yandex/clickhouse-jdbc]。
- 三方提供的 JDBC 驱动 ClickHouse-Native-JDBC [https://github.com/housepower/ClickHouse-Native-JDBC].

ODBC 驱动

- ClickHouse官方有 JDBC 的驱动。 见这里 [https://github.com/yandex/clickhouse-jdbc]。

第三方开发的库

放弃

Yandex不维护下面列出的库，也没有进行任何广泛的测试以确保其质量。

- Python
  - infi.clickhouse_orm [https://github.com/Infinidat/infi.clickhouse_orm]
  - clickhouse-driver [https://github.com/mymarilyn/clickhouse-driver]
  - clickhouse-client [https://github.com/yurial/clickhouse-client]
- PHP
  - phpClickHouse [https://github.com/smi2/phpClickHouse]

    - clickhouse-php-client [https://github.com/8bitov/clickhouse-php-client]
    - clickhouse-client [https://github.com/bozerkins/clickhouse-client]
    - PhpClickHouseClient [https://github.com/SevaCode/PhpClickHouseClient]
- Go
    - clickhouse [https://github.com/kshvakov/clickhouse/]
    - go-clickhouse [https://github.com/roistat/go-clickhouse]
    - mailrugo-clickhouse [https://github.com/mailru/go-clickhouse]
    - golang-clickhouse [https://github.com/leprosus/golang-clickhouse]
- NodeJs
    - clickhouse (NodeJs) [https://github.com/TimonKK/clickhouse]
    - node-clickhouse [https://github.com/apla/node-clickhouse]
- Perl
    - perl-DBD-ClickHouse [https://github.com/elcamlost/perl-DBD-ClickHouse]
    - HTTP-ClickHouse [https://metacpan.org/release/HTTP-ClickHouse]
    - AnyEvent-ClickHouse [https://metacpan.org/release/AnyEvent-ClickHouse]
- Ruby
    - clickhouse (Ruby) [https://github.com/archan937/clickhouse]
- R
    - clickhouse-r [https://github.com/hannesmuehleisen/clickhouse-r]
    - RClickhouse [https://github.com/IMSMWU/RClickhouse]
- Java
    - clickhouse-client-java [https://github.com/VirtusAI/clickhouse-client-java]
- Scala
    - clickhouse-scala-client [https://github.com/crobox/clickhouse-scala-client]
- Kotlin
    - AORM [https://github.com/TanVD/AORM]
- C#
    - ClickHouse.Ado [https://github.com/killwort/ClickHouse-Net]
    - ClickHouse.Net [https://github.com/ilyabreev/ClickHouse.Net]
- C++
    - clickhouse-cpp [https://github.com/artpaul/clickhouse-cpp/]
- Elixir
    - clickhousex [https://github.com/appodeal/clickhousex/]
- Nim
    - nim-clickhouse [https://github.com/leonardoce/nim-clickhouse]

第三方集成库

放弃

Yandex不维护下面列出的库，也没有进行任何广泛的测试以确保其质量。

- Python
    - SQLAlchemy [https://www.sqlalchemy.org]
        - sqlalchemy-clickhouse [https://github.com/cloudflare/sqlalchemy-clickhouse] (uses infi.clickhouse_orm [https://github.com/Infinidat/infi.clickhouse_orm])
- Java
    - Hadoop [http://hadoop.apache.org]
        - clickhouse-hdfs-loader [https://github.com/jaykelin/clickhouse-hdfs-loader] (uses JDBC [#jdbc])
- Scala
    - Akka [https://akka.io]
        - clickhouse-scala-client [https://github.com/crobox/clickhouse-scala-client]
- C#
    - ADO.NET [https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview]
        - ClickHouse.Ado [https://github.com/killwort/ClickHouse-Net]
        - ClickHouse.Net [https://github.com/ilyabreev/ClickHouse.Net]
        - ClickHouse.Net.Migrations [https://github.com/ilyabreev/ClickHouse.Net.Migrations]
- Elixir
    - Ecto [https://github.com/elixir-ecto/ecto]
        - clickhouse_ecto [https://github.com/appodeal/clickhouse_ecto]

第三方开发的可视化界面

Tabix

ClickHouse Web 界面 Tabix [https://github.com/tabixio/tabix].

主要功能：

- 浏览器直接连接 ClickHouse，不需要安装其他软件。
- 高亮语法的编辑器。
- 自动命令补全。
- 查询命令执行的图形分析工具。
- 配色方案选项。

Tabix 文档 [https://tabix.io/doc/].

HouseOps

HouseOps [https://github.com/HouseOps/HouseOps] 是一个交互式 UI/IDE 工具，可以运行在 OSX, Linux and Windows 平台中。

主要功能：

- 查询高亮语法提示，可以以表格或 JSON 格式查看数据。
- 支持导出 CSV 或 JSON 格式数据。
- 支持查看查询执行的详情，支持 KILL 查询。
- 图形化显示，支持显示数据库中所有的表和列的详细信息。
- 快速查看列占用的空间。

- 服务配置。

以下功能正在计划开发：- 数据库管理 - 用户管理 - 实时数据分析 - 集群监控 - 集群管理 - 监控副本情况以及 Kafka 引擎表

[]

数据类型

ClickHouse 可以在数据表中存储多种数据类型。

本节描述 ClickHouse 支持的数据类型，以及使用或者实现它们时（如果有的话）的注意事项。

[]

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

固定长度的整型，包括有符号整型或无符号整型。

整型范围

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

无符号整型范围

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

Float32, Float64

浮点数 [https://en.wikipedia.org/wiki/IEEE_754]。

类型与以下 C 语言中类型是相同的：

- `Float32` - `float`
- `Float64` - `double`

我们建议您尽可能以整数形式存储数据。例如，将固定精度的数字转换为整数值，例如货币数量或页面加载时间用毫秒为单位表示

使用浮点数

- 对浮点数进行计算可能引起四舍五入的误差。

```
SELECT 1 - 0.9
```

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

- 计算的结果取决于计算方法（计算机系统的处理器类型和体系结构）
- 浮点计算结果可能是诸如无穷大（`INF`）和"非数字"（`NaN`）。对浮点数计算的时候应该考虑到这点。
- 当一行行阅读浮点数的时候，浮点数的结果可能不是机器最近显示的数值。

NaN and Inf

与标准SQL相比，ClickHouse 支持以下类别的浮点数：

- `Inf` – 正无穷

```
SELECT 0.5 / 0
```

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

- `-Inf` – 负无穷

```
SELECT -0.5 / 0
```

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

- `NaN` – 非数字

```
SELECT 0 / 0
```

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

可以在ORDER BY 子句 [#query_language-queries-order_by] 查看更多关于 `NaN` 排序的规则。

[]

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

有符号的定点数，可在加、减和乘法运算过程中保持精度。对于除法，最低有效数字会被丢弃（不舍入）。

参数

- P - 精度。有效范围：[1:38]，决定可以有多少个十进制数字（包括分数）。
- S - 规模。有效范围：[0：P]，决定数字的小数部分中包含的小数位数。

对于不同的 P 参数值 Decimal 表示，以下例子都是同义的： - P from [1 : 9] - for Decimal32(S) - P from [10 : 18] - for Decimal64(S) - P from [19 : 38] - for Decimal128(S)

十进制值范围

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

例如，Decimal32(4) 可以表示 -99999.9999 至 99999.9999 的数值，步长为0.0001。

内部表示方式

数据采用与自身位宽相同的有符号整数存储。这个数在内存中实际范围会高于上述范围，从 String 转换到十进制数的时候会做对应的检查。

由于现代CPU不支持128位数字，因此 Decimal128 上的操作由软件模拟。所以 Decimal128 的运算速度明显慢于 Decimal32/Decimal64。

运算和结果类型

对Decimal的二进制运算导致更宽的结果类型（无论参数的顺序如何）。

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

精度变化的规则：

- 加法，减法：S = max(S1, S2)。
- 乘法：S = S1 + S2。
- 除法：S = S1。

对于 Decimal 和整数之间的类似操作，结果是与参数大小相同的十进制。

未定义Decimal和Float32/Float64之间的函数。要执行此类操作，您可以使用：toDecimal32、toDecimal64、toDecimal128 或 toFloat32，toFloat64，需要显式地转换其中一个参数。注意，结果将失去精度，类型转换是昂贵的操作。

Decimal上的一些函数返回结果为Float64（例如，var或stddev）。对于其中一些，中间计算发生在Decimal中。对于此类函数，尽管结果类型相同，但Float64和Decimal中相同数据的结果可能不同。

溢出检查

在对 Decimal 类型执行操作时，数值可能会发生溢出。分数中的过多数字被丢弃（不是舍入的）。整数中的过多数字将导致异常。

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

```
┌──────x─┬─divide(toDecimal32(2, 4), 3)─┐
│ 2.0000 │                       0.6666 │
└────────┴──────────────────────────────┘
```

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

```
DB::Exception: Scale is out of bounds.
```

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

```
DB::Exception: Decimal math overflow.
```

检查溢出会导致计算变慢。如果已知溢出不可能，则可以通过设置 `decimal_check_overflow` 来禁用溢出检查，在这种情况下，溢出将导致结果不正确：

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

```
┌──────────x─┬─multiply(6, toDecimal32(4.2, 8))─┐
│ 4.20000000 │                     -17.74967296 │
└────────────┴──────────────────────────────────┘
```

溢出检查不仅发生在算术运算上，还发生在比较运算上：

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

```
DB::Exception: Can't compare.
```

Boolean Values

没有单独的类型来存储布尔值。可以使用 UInt8 类型，取值限制为 0 或 1。

[]

String

字符串可以任意长度的。它可以包含任意的字节集，包含空字节。因此，字符串类型可以代替其他 DBMSs 中的 VARCHAR、BLOB、CLOB 等类型。

编码

ClickHouse 没有编码的概念。字符串可以是任意的字节集，按它们原本的方式进行存储和输出。若需存储文本，我们建议使用 UTF-8 编码。至少，如果你的终端使用UTF-8（推荐），这样读写就不需要进行任何的转换了。同样，对不同的编码文本 ClickHouse 会有不同处理字符串的函数。比如， `length` 函数可以计算字符串包含的字节数组的长度，然而 `lengthUTF8` 函数是假设字符串以 UTF-8 编码，计算的是字符串包含的 Unicode 字符的长度。

FixedString(N)

固定长度 N 的字符串。N 必须是严格的正自然数。 当服务端读取长度小于 N 的字符串时候（例如解析 INSERT 数据时），通过在字符串末尾添加空字节来达到 N 字节长度。 当服务端读取长度大于 N 的字符串时候，将返回错误消息。 当服务器写入一个字符串（例如，当输出 SELECT 查询的结果）时，NULL字节不会从字符串的末尾被移除，而是被输出。 注意这种方式与 MYSQL 的 CHAR 类型是不一样的（MYSQL 的字符串会以空格填充，然后输出的时候空格会被修剪）。

与 `String` 类型相比，极少的函数会使用 `FixedString(N)`，因此使用起来不太方便。

Date

日期类型，用两个字节存储，表示从 1970-01-01 (无符号) 到当前的日期值。允许存储从 Unix 纪元开始到编译阶段定义的上限阈值常量（目前上限是2106年，但最终完全支持的年份为2105）。最小值输出为0000-00-00。

日期中没有存储时区信息。

[]

DateTime

时间戳类型。用四个字节（无符号的）存储 Unix 时间戳）。允许存储与日期类型相同的范围内的值。最小值为 0000-00-00 00:00:00。时间戳类型值精确到秒（没有闰秒）。

时区

使用启动客户端或服务器时的系统时区，时间戳是从文本（分解为组件）转换为二进制并返回。在文本格式中，有关夏令时的信息会丢失。

默认情况下，客户端连接到服务的时候会使用服务端时区。您可以通过启用客户端命令行选项 `--use_client_time_zone` 来设置使用客户端时间。

因此，在处理文本日期时（例如，在保存文本转储时），请记住在夏令时更改期间可能存在歧义，如果时区发生更改，则可能存在匹配数据的问题。

[]

Enum8, Enum16

包括 `Enum8` 和 `Enum16` 类型。`Enum` 保存 `'string'= integer` 的对应关系。在 ClickHouse 中，尽管用户使用的是字符串常量，但所有含有 `Enum` 数据类型的操作都是按照包含整数的值来执行。这在性能方面比使用 `String` 数据类型更有效。

- `Enum8` 用 `'String'= Int8` 对描述。
- `Enum16` 用 `'String'= Int16` 对描述。

用法示例

创建一个带有一个枚举 `Enum8('hello' = 1, 'world' = 2)` 类型的列：

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

这个 `x` 列只能存储类型定义中列出的值：`'hello'` 或 `'world'` 。如果您尝试保存任何其他值，ClickHouse 抛出异常。

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

INSERT INTO t_enum VALUES

Ok.

3 rows in set. Elapsed: 0.002 sec.

:) insert into t_enum values('a')

INSERT INTO t_enum VALUES


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

当您从表中查询数据时，ClickHouse 从 `Enum` 中输出字符串值。

```
SELECT * FROM t_enum

┌─x─────┐
│ hello │
│ world │
│ hello │
└───────┘
```

如果需要看到对应行的数值，则必须将 `Enum` 值转换为整数类型。

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

┌─CAST(x, 'Int8')─┐
│               1 │
│               2 │
│               1 │
└─────────────────┘
```

在查询中创建枚举值，您还需要使用 `CAST` 。

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

┌─toTypeName(CAST('a', 'Enum8(\'a\' = 1, \'b\' = 2)'))─┐
│ Enum8('a' = 1, 'b' = 2)                              │
└──────────────────────────────────────────────────────┘
```

规则及用法

`Enum8` 类型的每个值范围是 `-128 ... 127` ， `Enum16` 类型的每个值范围是 `-32768 ... 32767` 。所有的字符串或者数字都必须是不一样的。允许存在空字符串。如果某个 Enum 类型被指定了（在表定义的时候），数字可以是任意顺序。然而，顺序并不重要。

`Enum` 中的字符串和数值都不能是 NULL [#null-literal]。

`Enum` 包含在 Nullable [#data_type-nullable] 类型中。因此，如果您使用此查询创建一个表

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

不仅可以存储 `'hello'` 和 `'world'` ，还可以存储 `NULL` 。

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

在内存中， `Enum` 列的存储方式与相应数值的 `Int8` 或 `Int16` 相同。

当以文本方式读取的时候，ClickHouse 将值解析成字符串然后去枚举值的集合中搜索对应字符串。如果没有找到，会抛出异常。当读取文本格式的时候，会根据读取到的字符串去找对应的数值。如果没有找到，会抛出异常。

当以文本形式写入时，ClickHouse 将值解析成字符串写入。如果列数据包含垃圾数据（不是来自有效集合的数字），则抛出异常。Enum 类型以二进制读取和写入的方式与 `Int8` 和 `Int16` 类型一样的。

隐式默认值是数值最小的值。

在 `ORDER BY` ， `GROUP BY` ， `IN` ， `DISTINCT` 等等中，Enum 的行为与相应的数字相同。例如，按数字排序。对于等式运算符和比较运算符，Enum 的工作机制与它们在底层数值上的工作机制相同。

枚举值不能与数字进行比较。枚举可以与常量字符串进行比较。如果与之比较的字符串不是有效Enum值，则将引发异常。可以使用 IN 运算符来判断一个 Enum 是否存在于某个 Enum 集合中，其中集合中的 Enum 需要用字符串表示。

大多数具有数字和字符串的运算并不适用于Enums；例如，Enum 类型不能和一个数值相加。但是，Enum有一个原生的 `toString` 函数，它返回它的字符串值。

Enum 值使用 `toT` 函数可以转换成数值类型，其中 T 是一个数值类型。若 `T` 恰好对应 Enum 的底层数值类型，这个转换是零消耗的。

Enum 类型可以被 `ALTER` 无成本地修改对应集合的值。可以通过 `ALTER` 操作来增加或删除 Enum 的成员（只要表没有用到该值，删除都是安全的）。作为安全保障，改变之前使用过的 Enum 成员将抛出异常。

通过 `ALTER` 操作，可以将 `Enum8` 转成 `Enum16`，反之亦然，就像 `Int8` 转 `Int16` 一样。

[]

Array(T)

由 `T` 类型元素组成的数组。

`T` 可以是任意类型，包含数组类型。 但不推荐使用多维数组，ClickHouse 对多维数组的支持有限。例如，不能存储在 `MergeTree` 表中存储多维数组。

创建数组

您可以使用array函数来创建数组：

```
array(T)
```

您也可以使用方括号：

```
[]
```

创建数组示例：

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

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

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

1 rows in set. Elapsed: 0.002 sec.

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

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

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

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

使用数据类型

ClickHouse会自动检测数组元素,并根据元素计算出存储这些元素最小的数据类型。如果在元素中存在NULL [#null-literal]或存在Nullable [#data_type-nullable]类型元素，那么数组的元素类型将会变成Nullable [#data_type-nullable]。

如果 ClickHouse 无法确定数据类型，它将产生异常。当尝试同时创建一个包含字符串和数字的数组时会发生这种情况 (`SELECT array(1, 'a')`)。

自动数据类型检测示例：

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

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

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

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

如果您尝试创建不兼容的数据类型数组，ClickHouse 将引发异常：

```
:) SELECT array(1, 'a')

SELECT [1, 'a']

Received exception from server (version 1.1.54388):
Code: 386. DB::Exception: Received from localhost:9000, 127.0.0.1. DB::Exception: There is no supertype for
types UInt8, String because some of them are String/FixedString and some of them are not.

0 rows in set. Elapsed: 0.246 sec.
```

AggregateFunction(name, types_of_arguments…)

表示聚合函数中的中间状态。可以在聚合函数中通过 '-State' 后缀来访问它。更多信息，参考 "AggregatingMergeTree"。

[]

Tuple(T1, T2, ...)

元组，其中每个元素都有单独的 类型 [#data_types]。

不能在表中存储元组（除了内存表）。它们可以用于临时列分组。在查询中，IN 表达式和带特定参数的 lambda 函数可以来对临时列进行分组。更多信息，请参阅 IN 操作符 [#in_operators] and Higher order functions [#higher_order_functions]。

元组可以是查询的结果。在这种情况下，对于JSON以外的文本格式，括号中的值是逗号分隔的。在JSON格式中，元组作为数组输出（在方括号中）。

创建元组

可以使用函数来创建元组：

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

创建元组的示例：

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

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

┌─x───────┬─toTypeName(tuple(1, 'a'))─┐
│ (1,'a') │ Tuple(UInt8, String)      │
└─────────┴───────────────────────────┘

1 rows in set. Elapsed: 0.021 sec.
```

元组中的数据类型

在动态创建元组时，ClickHouse 会自动为元组的每一个参数赋予最小可表达的类型。如果参数为 NULL [#null-literal]，那这个元组对应元素是 Nullable [#data_type-nullable]。

自动数据类型检测示例：

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

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

┌─x────────┬─toTypeName(tuple(1, NULL))──────┐
│ (1,NULL) │ Tuple(UInt8, Nullable(Nothing)) │
└──────────┴─────────────────────────────────┘

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

[]

Nullable(TypeName)

允许用特殊标记 (NULL [#null-literal]) 表示"缺失值"，可以与 `TypeName` 的正常值存放一起。例如， `Nullable(Int8)` 类型的列可以存储 `Int8` 类型值，而没有值的行将存储 `NULL` 。

对于 `TypeName` ， 不能使用复合数据类型 Array [#data_type is array] 和 Tuple [#data_type-tuple]。复合数据类型可以包含 `Nullable` 类型值，例如 `Array(Nullable(Int8))` 。

`Nullable` 类型字段不能包含在表索引中。

除非在 ClickHouse 服务器配置中另有说明，否则 `NULL` 是任何 `Nullable` 类型的默认值。

存储特性

要在表的列中存储 `Nullable` 类型值，ClickHouse 除了使用带有值的普通文件外，还使用带有 `NULL` 掩码的单独文件。 掩码文件中的条目允许 ClickHouse 区分每个表行的 `NULL` 和相应数据类型的默认值。 由于附加了新文件， `Nullable` 列与类似的普通文件相比消耗额外的存储空间。

!!! 注意点 使用 `Nullable` 几乎总是对性能产生负面影响，在设计数据库时请记住这一点

掩码文件中的条目允许ClickHouse区分每个表行的对应数据类型的"NULL"和默认值由于有额外的文件，"Nullable"列比普通列消耗更多的存储空间

用法示例

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

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

Ok.

0 rows in set. Elapsed: 0.012 sec.

:) INSERT INTO t_null VALUES (1, NULL)

INSERT INTO t_null VALUES

Ok.

1 rows in set. Elapsed: 0.007 sec.

:) SELECT x + y FROM t_null

SELECT x + y
FROM t_null

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

2 rows in set. Elapsed: 0.144 sec.
```

嵌套数据结构

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

嵌套数据结构类似于嵌套表。嵌套数据结构的参数（列名和类型）与 CREATE 查询类似。每个表可以包含任意多行嵌套数据结构。

示例:

```
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)
```

上述示例声明了 `Goals` 这种嵌套数据结构，它包含访客转化相关的数据（访客达到的目标）。在 'visits' 表中每一行都可以对应零个或者任意个转化数据。

只支持一级嵌套。嵌套结构的列中，若列的类型是数组类型，那么该列其实和多维数组是相同的，所以目前嵌套层级的支持很局限（MergeTree 引擎中不支持存储这样的列）

大多数情况下，处理嵌套数据结构时，会指定一个单独的列。为了这样实现，列的名称会与点号连接起来。这些列构成了一组匹配类型。在同一条嵌套数据中，所有的列都具有相同的长度。

示例:

```
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'] │
└────────────────────────────────┴───────────────────────────────────────────────────────────────────────────────────────────┘
```

所以可以简单地把嵌套数据结构当做是所有列都是相同长度的多列数组。

SELECT 查询只有在使用 ARRAY JOIN 的时候才可以指定整个嵌套数据结构的名称。更多信息，参考 "ARRAY JOIN 子句"。示例：

```
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 │
└─────────┴─────────────────────┘
```

不能对整个嵌套数据结构执行 SELECT。只能明确列出属于它一部分列。

对于 INSERT 查询，可以单独地传入所有嵌套数据结构中的列数组（假如它们是单独的列数组）。在插入过程中，系统会检查它们是否有相同的长度。

对于 DESCRIBE 查询，嵌套数据结构中的列会以相同的方式分别列出来。

ALTER 查询对嵌套数据结构的操作非常有限。

Special Data Types

特殊数据类型的值既不能存在表中也不能在结果中输出，但可用于查询的中间结果。

Expression

用于表示高阶函数中的Lambd表达式。

Set

可以用在 IN 表达式的右半部分。

[]

Nothing

此数据类型的唯一目的是表示不是期望值的情况。 所以不能创建一个 `Nothing` 类型的值。

例如，文本 NULL [#null-literal] 的类型为 `Nullable(Nothing)` 。详情请见 Nullable [#data_type-nullable]。

`Nothing` 类型也可以用来表示空数组：

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

SELECT toTypeName([])

┌─toTypeName(array())─┐
│ Array(Nothing)      │
└─────────────────────┘

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

SQL Reference

- SELECT [#select]
- INSERT INTO [#queries-insert]
- CREATE [#create-database]
- ALTER [#query_language_queries_alter]
- Other types of queries [#miscellanous-queries]

SELECT Queries Syntax

`SELECT` performs data retrieval.

```
SELECT [DISTINCT] expr_list
    [FROM [db.]table | (subquery) | table_function] [FINAL]
    [SAMPLE sample_coeff]
    [ARRAY JOIN ...]
    [GLOBAL] ANY|ALL INNER|LEFT JOIN (subquery)|table USING columns_list
    [PREWHERE expr]
    [WHERE expr]
    [GROUP BY expr_list] [WITH TOTALS]
    [HAVING expr]
    [ORDER BY expr_list]
    [LIMIT [n, ]m]
    [UNION ALL ...]
    [INTO OUTFILE filename]
    [FORMAT format]
    [LIMIT n BY columns]
```

All the clauses are optional, except for the required list of expressions immediately after SELECT. The clauses below are described in almost the same order as in the query execution conveyor.

If the query omits the `DISTINCT`, `GROUP BY` and `ORDER BY` clauses and the `IN` and `JOIN` subqueries, the query will be completely stream processed, using O(1) amount of RAM. Otherwise, the query might consume a lot of RAM if the appropriate restrictions are not specified: `max_memory_usage`, `max_rows_to_group_by`, `max_rows_to_sort`, `max_rows_in_distinct`, `max_bytes_in_distinct`, `max_rows_in_set`, `max_bytes_in_set`, `max_rows_in_join`, `max_bytes_in_join`, `max_bytes_before_external_sort`, `max_bytes_before_external_group_by`. For more information, see the section "Settings". It is possible to use external sorting (saving temporary tables to a disk) and external aggregation. `The system does not have "merge join"`.

FROM Clause

If the FROM clause is omitted, data will be read from the `system.one` table. The 'system.one' table contains exactly one row (this table fulfills the same purpose as the DUAL table found in other DBMSs).

The FROM clause specifies the table to read data from, or a subquery, or a table function; ARRAY JOIN and the regular JOIN may also be included (see below).

Instead of a table, the SELECT subquery may be specified in brackets. In this case, the subquery processing pipeline will be built into the processing pipeline of an external query. In contrast to standard SQL, a synonym does not need to be specified after a subquery. For compatibility, it is possible to write 'AS name' after a subquery, but the specified name isn't used anywhere.

A table function may be specified instead of a table. For more information, see the section "Table functions".

To execute a query, all the columns listed in the query are extracted from the appropriate table. Any columns not needed for the external query are thrown out of the subqueries. If a query does not list any columns (for example, SELECT count() FROM t), some column is extracted from the table anyway (the smallest one is preferred), in order to calculate the number of rows.

The FINAL modifier can be used only for a SELECT from a CollapsingMergeTree table. When you specify FINAL, data is selected fully "collapsed". Keep in mind that using FINAL leads to a selection that includes columns related to the primary key, in addition to the columns specified in the SELECT. Additionally, the query will be executed in a single stream, and data will be merged during query execution. This means that when using FINAL, the query is processed more slowly. In most cases, you should avoid using FINAL. For more information, see the section "CollapsingMergeTree engine".

SAMPLE Clause

The SAMPLE clause allows for approximated query processing. 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 processing, 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` 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` stringency:If `ANY` is specified and the right table has several matching rows, only the first one found is joined. If `ALL` is specified and the right table has several matching rows, 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".

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

When running a JOIN, there is no optimization of the order of execution in relation to other stages of the query. The join (a search in the right table) is run before filtering in WHERE and before aggregation. In order to explicitly set the processing order, we recommend running a JOIN subquery with a subquery.

Example:

```
SELECT
    CounterID,
    hits,
    visits
FROM
(
    SELECT
        CounterID,
        count() AS hits
    FROM test.hits
    GROUP BY CounterID
) ANY LEFT JOIN
(
    SELECT
        CounterID,
        sum(Sign) AS visits
    FROM test.visits
    GROUP BY CounterID
) USING CounterID
ORDER BY hits DESC
LIMIT 10
```

```
┌─CounterID─┬───hits─┬─visits─┐
│   1143050 │ 523264 │  13665 │
│    731962 │ 475698 │ 102716 │
│    722545 │ 337212 │ 108187 │
│    722889 │ 252197 │  10547 │
│   2237260 │ 196036 │   9522 │
│  23057320 │ 147211 │   7689 │
│    722818 │  90109 │  17847 │
│     48221 │  85379 │   4652 │
│  19762435 │  77807 │   7026 │
│    722884 │  77492 │  11056 │
└───────────┴────────┴────────┘
```

Subqueries don't allow you to set names or use them for referencing a column from a specific subquery. The columns specified in USING must have the same names in both subqueries, and the other columns must be named differently. You can use aliases to change the names of columns in subqueries (the example uses the aliases 'hits' and 'visits').

The USING clause specifies one or more columns to join, which establishes the equality of these columns. The list of columns is set without brackets. More complex join conditions are not supported.

The right table (the subquery result) resides in RAM. If there isn't enough memory, you can't run a JOIN.

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

The JOIN behavior is affected by the join_use_nulls [#settings-join_use_nulls] setting. With `join_use_nulls=1,` `JOIN` works like in standard SQL.

If the JOIN keys are Nullable [#data_types-nullable] fields, the rows where at least one of the keys has the value NULL [#null-literal] are not joined.

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 tables from the `*MergeTree` family.

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 -- getting the first occurred page header for each domain.
FROM hits
GROUP BY domain
```

For every different key value encountered, GROUP BY calculates a set of aggregate function values.

GROUP BY is not supported for array columns.

A constant can't be specified as arguments for aggregate functions. Example: sum(1). Instead of this, you can get rid of the constant. Example: `count()` .

NULL PROCESSING

For grouping, ClickHouse interprets NULL [#null-literal] as a value, and `NULL=NULL` .

Here's an example to show what this means.

Assume you have this table:

```
┌─x─┬────y─┐
│ 1 │    2 │
│ 2 │ ᴺᵁᴸᴸ │
│ 3 │    2 │
│ 3 │    3 │
│ 3 │ ᴺᵁᴸᴸ │
└───┴──────┘
```

The query `SELECT sum(x), y FROM t_null_big GROUP BY y` results in:

```
┌─sum(x)─┬────y─┐
│      4 │    2 │
│      3 │    3 │
│      5 │ ᴺᵁᴸᴸ │
└────────┴──────┘
```

You can see that `GROUP BY` for `y = NULL` summed up `x`, as if `NULL` is this value.

If you pass several keys to `GROUP BY`, the result will give you all the combinations of the selection, as if `NULL` were a specific value.

WITH TOTALS MODIFIER

If the WITH TOTALS modifier is specified, another row will be calculated. This row will have key columns containing default values (zeros or empty lines), and columns of aggregate functions with the values calculated across all the rows (the "total" values).

This extra row is output in JSON*, TabSeparated*, and Pretty* formats, separately from the other rows. In the other formats, this row is not output.

In JSON* formats, this row is output as a separate 'totals' field. In TabSeparated* formats, the row comes after the main result, preceded by an empty row (after the other data). In Pretty* formats, the row is output as a separate table after the main result.

`WITH TOTALS` can be run in different ways when HAVING is present. The behavior depends on the 'totals_mode' setting. By default, `totals_mode = 'before_having'`. In this case, 'totals' is calculated across all rows, including the ones that don't pass through HAVING and 'max_rows_to_group_by'.

The other alternatives include only the rows that pass through HAVING in 'totals', and behave differently with the setting `max_rows_to_group_by` and `group_by_overflow_mode = 'any'`.

`after_having_exclusive` – Don't include rows that didn't pass through `max_rows_to_group_by`. In other words, 'totals' will have less than or the same number of rows as it would if `max_rows_to_group_by` were omitted.

`after_having_inclusive` – Include all the rows that didn't pass through 'max_rows_to_group_by' in 'totals'. In other words, 'totals' will have more than or the same number of rows as it would if `max_rows_to_group_by` were omitted.

`after_having_auto` – Count the number of rows that passed through HAVING. If it is more than a certain amount (by default, 50%), include all the rows that didn't pass through 'max_rows_to_group_by' in 'totals'. Otherwise, do not include them.

`totals_auto_threshold` – By default, 0.5. The coefficient for `after_having_auto`.

If `max_rows_to_group_by` and `group_by_overflow_mode = 'any'` are not used, all variations of `after_having` are the same, and you can use any of them (for example, `after_having_auto`).

You can use WITH TOTALS in subqueries, including subqueries in the JOIN clause (in this case, the respective total values

are combined).

GROUP BY IN EXTERNAL MEMORY

You can enable dumping temporary data to the disk to restrict memory usage during GROUP BY. The `max_bytes_before_external_group_by` setting determines the threshold RAM consumption for dumping GROUP BY temporary data to the file system. If set to 0 (the default), it is disabled.

When using `max_bytes_before_external_group_by`, we recommend that you set max_memory_usage about twice as high. This is necessary because there are two stages to aggregation: reading the date and forming intermediate data (1) and merging the intermediate data (2). Dumping data to the file system can only occur during stage 1. If the temporary data wasn't dumped, then stage 2 might require up to the same amount of memory as in stage 1.

For example, if `max_memory_usage` was set to 10000000000 and you want to use external aggregation, it makes sense to set `max_bytes_before_external_group_by` to 10000000000, and max_memory_usage to 20000000000. When external aggregation is triggered (if there was at least one dump of temporary data), maximum consumption of RAM is only slightly more than `max_bytes_before_external_group_by`.

With distributed query processing, external aggregation is performed on remote servers. In order for the requestor server to use only a small amount of RAM, set `distributed_aggregation_memory_efficient` to 1.

When merging data flushed to the disk, as well as when merging results from remote servers when the `distributed_aggregation_memory_efficient` setting is enabled, consumes up to 1/256 * the number of threads from the total amount of RAM.

When external aggregation is enabled, if there was less than `max_bytes_before_external_group_by` of data (i.e. data was not flushed), the query runs just as fast as without external aggregation. If any temporary data was flushed, the run time will be several times longer (approximately three times).

If you have an ORDER BY with a small LIMIT after GROUP BY, then the ORDER BY CLAUSE will not use significant amounts of RAM. But if the ORDER BY doesn't have LIMIT, don't forget to enable external sorting (`max_bytes_before_external_sort`).

LIMIT N BY Clause

LIMIT N BY COLUMNS selects the top N rows for each group of COLUMNS. LIMIT N BY is not related to LIMIT; they can both be used in the same query. The key for LIMIT N BY can contain any 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
```

The query will select the top 5 referrers for each `domain, device_type` pair, but not more than 100 rows (`LIMIT n BY + LIMIT`).

HAVING Clause

Allows filtering the result received after GROUP BY, similar to the WHERE clause. WHERE and HAVING differ in that WHERE is performed before aggregation (GROUP BY), while HAVING is performed after it. If aggregation is not performed, HAVING can't be used.

[]

ORDER BY Clause

The ORDER BY clause contains a list of expressions, which can each be assigned DESC or ASC (the sorting direction). If the direction is not specified, ASC is assumed. ASC is sorted in ascending order, and DESC in descending order. The sorting direction applies to a single expression, not to the entire list. Example: `ORDER BY Visits DESC, SearchPhrase`

For sorting by String values, you can specify collation (comparison). Example: `ORDER BY SearchPhrase COLLATE 'tr'` - for sorting by keyword in ascending order, using the Turkish alphabet, case insensitive, assuming that strings are UTF-8 encoded. COLLATE can be specified or not for each expression in ORDER BY independently. If ASC or DESC is specified, COLLATE is specified after it. When using COLLATE, sorting is always case-insensitive.

We only recommend using COLLATE for final sorting of a small number of rows, since sorting with COLLATE is less efficient than normal sorting by bytes.

Rows that have identical values for the list of sorting expressions are output in an arbitrary order, which can also be nondeterministic (different each time). If the ORDER BY clause is omitted, the order of the rows is also undefined, and may be nondeterministic as well.

`NaN` and `NULL` sorting order:

- With the modifier `NULLS FIRST` — First `NULL` , then `NaN` , then other values.
- With the modifier `NULLS LAST` — First the values, then `NaN` , then `NULL` .
- Default — The same as with the `NULLS LAST` modifier.

Example:

For the table

```
┌─x─┬────y─┐
│ 1 │ ᴺᵁᴸᴸ │
│ 2 │    2 │
│ 1 │  nan │
│ 2 │    2 │
│ 3 │    4 │
│ 5 │    6 │
│ 6 │  nan │
│ 7 │ ᴺᵁᴸᴸ │
│ 6 │    7 │
│ 8 │    9 │
└───┴──────┘
```

Run the query `SELECT * FROM t_null_nan ORDER BY y NULLS FIRST` to get:

```
┌─x─┬────y─┐
│ 1 │ ᴺᵁᴸᴸ │
│ 7 │ ᴺᵁᴸᴸ │
│ 1 │  nan │
│ 6 │  nan │
│ 2 │    2 │
│ 2 │    2 │
│ 3 │    4 │
│ 5 │    6 │
│ 6 │    7 │
│ 8 │    9 │
└───┴──────┘
```

When floating point numbers are sorted, NaNs are separate from the other values. Regardless of the sorting order, NaNs

come at the end. In other words, for ascending sorting they are placed as if they are larger than all the other numbers, while for descending sorting they are placed as if they are smaller than the rest.

Less RAM is used if a small enough LIMIT is specified in addition to ORDER BY. Otherwise, the amount of memory spent is proportional to the volume of data for sorting. For distributed query processing, if GROUP BY is omitted, sorting is partially done on remote servers, and the results are merged on the requestor server. This means that for distributed sorting, the volume of data to sort can be greater than the amount of memory on a single server.

If there is not enough RAM, it is possible to perform sorting in external memory (creating temporary files on a disk). Use the setting `max_bytes_before_external_sort` for this purpose. If it is set to 0 (the default), external sorting is disabled. If it is enabled, when the volume of data to sort reaches the specified number of bytes, the collected data is sorted and dumped into a temporary file. After all data is read, all the sorted files are merged and the results are output. Files are written to the /var/lib/clickhouse/tmp/ directory in the config (by default, but you can use the 'tmp_path' parameter to change this setting).

Running a query may use more memory than 'max_bytes_before_external_sort'. For this reason, this setting must have a value significantly smaller than 'max_memory_usage'. As an example, if your server has 128 GB of RAM and you need to run a single query, set 'max_memory_usage' to 100 GB, and 'max_bytes_before_external_sort' to 80 GB.

External sorting works much less effectively than sorting in RAM.

SELECT Clause

The expressions specified in the SELECT clause are analyzed after the calculations for all the clauses listed above are completed. More specifically, expressions are analyzed that are above the aggregate functions, if there are any aggregate functions. The aggregate functions and everything below them are calculated during aggregation (GROUP BY). These expressions work as if they are applied to separate rows in the result.

DISTINCT Clause

If DISTINCT is specified, only a single row will remain out of all the sets of fully matching rows in the result. The result will be the same as if GROUP BY were specified across all the fields specified in SELECT without aggregate functions. But there are several differences from GROUP BY:

- DISTINCT can be applied together with GROUP BY.
- When ORDER BY is omitted and LIMIT is defined, the query stops running immediately after the required number of different rows has been read.
- Data blocks are output as they are processed, without waiting for the entire query to finish running.

DISTINCT is not supported if SELECT has at least one array column.

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.

`DISTINCT` works with NULL [#null-literal] as if `NULL` were a specific value, and `NULL=NULL` . In other words, in the `DISTINCT` results, different combinations with `NULL` only occur once.

UNION ALL Clause

You can use UNION ALL to combine any number of queries. Example:

```
SELECT CounterID, 1 AS table, toInt64(count()) AS c
    FROM test.hits
    GROUP BY CounterID

UNION ALL

SELECT CounterID, 2 AS table, sum(Sign) AS c
    FROM test.visits
    GROUP BY CounterID
    HAVING c > 0
```

Only UNION ALL is supported. The regular UNION (UNION DISTINCT) is not supported. If you need UNION DISTINCT, you can write SELECT DISTINCT from a subquery containing UNION ALL.

Queries that are parts of UNION ALL can be run simultaneously, and their results can be mixed together.

The structure of results (the number and type of columns) must match for the queries. But the column names can differ. In this case, the column names for the final result will be taken from the first query. Type casting is performed for unions. For example, if two queries being combined have the same field with non-`Nullable` and `Nullable` types from a compatible type, the resulting `UNION ALL` has a `Nullable` type field.

Queries that are parts of UNION ALL can't be enclosed in brackets. ORDER BY and LIMIT are applied to separate queries, not to the final result. If you need to apply a conversion to the final result, you can put all the queries with UNION ALL in a subquery in the FROM clause.

INTO OUTFILE Clause

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

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

FORMAT Clause

Specify 'FORMAT format' to get data in any specified format. You can use this for convenience, or for creating dumps. For more information, see the section "Formats". If the FORMAT clause is omitted, the default format is used, which depends on both the settings and the interface used for accessing the DB. For the HTTP interface and the command-line client in batch mode, the default format is TabSeparated. For the command-line client in interactive mode, the default format is PrettyCompact (it has attractive and compact tables).

When using the command-line client, data is passed to the client in an internal efficient format. The client independently interprets the FORMAT clause of the query and formats the data itself (thus relieving the network and the server from the load).

[]

IN Operators

The `IN`, `NOT IN`, `GLOBAL IN`, and `GLOBAL NOT IN` operators are covered separately, since their functionality is quite rich.

The left side of the operator is either a single column or a tuple.

Examples:

```
SELECT UserID IN (123, 456) FROM ...
SELECT (CounterID, UserID) IN ((34, 123), (101500, 456)) FROM ...
```

If the left side is a single column that is in the index, and the right side is a set of constants, the system uses the index for processing the query.

Don't list too many values explicitly (i.e. millions). If a data set is large, put it in a temporary table (for example, see the section "External data for query processing"), then use a subquery.

The right side of the operator can be a set of constant expressions, a set of tuples with constant expressions (shown in the examples above), or the name of a database table or SELECT subquery in brackets.

If the right side of the operator is the name of a table (for example, `UserID IN users`), this is equivalent to the subquery `UserID IN (SELECT * FROM users)`. Use this when working with external data that is sent along with the query. For example, the query can be sent together with a set of user IDs loaded to the 'users' temporary table, which should be filtered.

If the right side of the operator is a table name that has the Set engine (a prepared data set that is always in RAM), the data set will not be created over again for each query.

The subquery may specify more than one column for filtering tuples. Example:

```
SELECT (CounterID, UserID) IN (SELECT CounterID, UserID FROM ...) FROM ...
```

The columns to the left and right of the IN operator should have the same type.

The IN operator and subquery may occur in any part of the query, including in aggregate functions and lambda functions. Example:

```
SELECT
    EventDate,
    avg(UserID IN
    (
        SELECT UserID
        FROM test.hits
        WHERE EventDate = toDate('2014-03-17')
    )) AS ratio
FROM test.hits
GROUP BY EventDate
ORDER BY EventDate ASC
```

```
┌──EventDate─┬────ratio─┐
│ 2014-03-17 │        1 │
│ 2014-03-18 │ 0.807696 │
│ 2014-03-19 │ 0.755406 │
│ 2014-03-20 │ 0.723218 │
│ 2014-03-21 │ 0.697021 │
│ 2014-03-22 │ 0.647851 │
│ 2014-03-23 │ 0.648416 │
└────────────┴──────────┘
```

For each day after March 17th, count the percentage of pageviews made by users who visited the site on March 17th. A subquery in the IN clause is always run just one time on a single server. There are no dependent subqueries.

NULL PROCESSING

During request processing, the IN operator assumes that the result of an operation withNULL [#null-literal] is always equal to `0`, regardless of whether `NULL` is on the right or left side of the operator. `NULL` values are not included in any dataset, do not correspond to each other and cannot be compared.

Here is an example with the `t_null` table:

```
┌─x─┬────y─┐
│ 1 │ NULL │
│ 2 │    3 │
└───┴──────┘
```

Running the query `SELECT x FROM t_null WHERE y IN (NULL,3)` gives you the following result:

```
┌─x─┐
│ 2 │
└───┘
```

You can see that the row in which `y = NULL` is thrown out of the query results. This is because ClickHouse can't decide whether `NULL` is included in the `(NULL,3)` set, returns `0` as the result of the operation, and `SELECT` excludes this row from the final output.

``` SELECT y IN (NULL, 3) FROM t_null

┌─in(y, tuple(NULL, 3))─┐ │ 0 │ │ 1 │ └───────────────────────┘ ```

[]

DISTRIBUTED SUBQUERIES

There are two options for IN-s with subqueries (similar to JOINs): normal `IN` / `OIN` and `IN GLOBAL` / `GLOBAL JOIN`. They differ in how they are run for distributed query processing.

**Attention**

Remember that the algorithms described below may work differently depending on the settings[#settings-distributed_product_mode] `distributed_product_mode` setting.

When using the regular IN, the query is sent to remote servers, and each of them runs the subqueries in the `IN` or `JOIN` clause.

When using `GLOBAL IN` / `GLOBAL JOINs`, first all the subqueries are run for `GLOBAL IN` / `GLOBAL JOINs`, and the results are collected in temporary tables. Then the temporary tables are sent to each remote server, where the queries are run using this temporary data.

For a non-distributed query, use the regular `IN` / `JOIN`.

Be careful when using subqueries in the `IN` / `JOIN` clauses for distributed query processing.

Let's look at some examples. Assume that each server in the cluster has a normal**local_table**. Each server also has a **distributed_table** table with the **Distributed** type, which looks at all the servers in the cluster.

For a query to the**distributed_table**, the query will be sent to all the remote servers and run on them using the**local_table**.

For example, the query

```
SELECT uniq(UserID) FROM distributed_table
```

will be sent to all remote servers as

```
SELECT uniq(UserID) FROM local_table
```

and run on each of them in parallel, until it reaches the stage where intermediate results can be combined. Then the intermediate results will be returned to the requestor server and merged on it, and the final result will be sent to the client.

Now let's examine a query with IN:

```
SELECT uniq(UserID) FROM distributed_table WHERE CounterID = 101500 AND UserID IN (SELECT UserID FROM
local_table WHERE CounterID = 34)
```

- Calculation of the intersection of audiences of two sites.

This query will be sent to all remote servers as
```

```sql
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:

```sql
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

```sql
SELECT uniq(UserID) FROM local_table WHERE CounterID = 101500 AND UserID IN (SELECT UserID FROM distributed_table WHERE CounterID = 34)
```

The subquery will begin running on each remote server. Since the subquery uses a distributed table, the subquery that is on each remote server will be resent to every remote server as

```sql
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

```sql
SELECT uniq(UserID) FROM distributed_table WHERE CounterID = 101500 AND UserID GLOBAL IN (SELECT UserID FROM distributed_table WHERE CounterID = 34)
```

The requestor server will run the subquery

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

and the result will be put in a temporary table in RAM. Then the request will be sent to each remote server as

```sql
SELECT uniq(UserID) FROM local_table WHERE CounterID = 101500 AND UserID GLOBAL IN _data1
```

and the temporary table `_data1` will be sent to every remote server with the query (the name of the temporary table is implementation-defined).

This is more optimal than using the normal IN. However, keep the following points in mind:

1. When creating a temporary table, data is not made unique. To reduce the volume of data transmitted over the network, specify DISTINCT in the subquery. (You don't need to do this for a normal IN.)
2. The temporary table will be sent to all the remote servers. Transmission does not account for network topology. For example, if 10 remote servers reside in a datacenter that is very remote in relation to the requestor server, the data will be sent 10 times over the channel to the remote datacenter. Try to avoid large data sets when using GLOBAL IN.
3. When transmitting data to remote servers, restrictions on network bandwidth are not configurable. You might overload the network.
4. Try to distribute data across servers so that you don't need to use GLOBAL IN on a regular basis.
5. If you need to use GLOBAL IN often, plan the location of the ClickHouse cluster so that a single group of replicas

resides in no more than one data center with a fast network between them, so that a query can be processed entirely within a single data center.

It also makes sense to specify a local table in the `GLOBAL IN` clause, in case this local table is only available on the requestor server and you want to use data from it on remote servers.

Extreme Values

In addition to results, you can also get minimum and maximum values for the results columns. To do this, set the**extremes** setting to 1. Minimums and maximums are calculated for numeric types, dates, and dates with times. For other columns, the default values are output.

An extra two rows are calculated – the minimums and maximums, respectively. These extra two rows are output in JSON*, TabSeparated*, and Pretty* formats, separate from the other rows. They are not output for other formats.

In JSON* formats, the extreme values are output in a separate 'extremes' field. In TabSeparated* formats, the row comes after the main result, and after 'totals' if present. It is preceded by an empty row (after the other data). In Pretty* formats, the row is output as a separate table after the main result, and after 'totals' if present.

Extreme values are calculated for rows that have passed through LIMIT. However, when using 'LIMIT offset, size', the rows before 'offset' are included in 'extremes'. In stream requests, the result may also include a small number of rows that passed through LIMIT.

Notes

The `GROUP BY` and `ORDER BY` clauses do not support positional arguments. This contradicts MySQL, but conforms to standard SQL. For example, `GROUP BY 1, 2` will be interpreted as grouping by constants (i.e. aggregation of all rows into one).

You can use synonyms (`AS` aliases) in any part of a query.

You can put an asterisk in any part of a query instead of an expression. When the query is analyzed, the asterisk is expanded to a list of all table columns (excluding the `MATERIALIZED` and `ALIAS` columns). There are only a few cases when using an asterisk is justified:

- When creating a table dump.
- For tables containing just a few columns, such as system tables.
- For getting information about what columns are in a table. In this case, set `LIMIT 1`. But it is better to use the `DESC TABLE` query.
- When there is strong filtration on a small number of columns using `PREWHERE`.
- In subqueries (since columns that aren't needed for the external query are excluded from subqueries).

In all other cases, we don't recommend using the asterisk, since it only gives you the drawbacks of a columnar DBMS instead of the advantages. In other words using the asterisk is not recommended.

[]

INSERT

正在添加数据。

基本查询格式:

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

此查询能够指定字段的列表来插入 `[(c1, c2, c3)]` 。在这种情况下，剩下的字段用如下来填充:

- 从表定义中指定的 `DEFAULT` 表达式中计算出值。
- 空字符串， 如果 `DEFAULT` 表达式没有定义。

如果 strict_insert_defaults=1 [#settings-strict_insert_defaults]， 没有 `DEFAULT` 定义的字段必须在查询中列出.

在任何ClickHouse所支持的格式上 format [#formats] 数据被传入到 INSERT中. 此格式必须被显式地指定在查询中:

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

例如， 如下的查询格式与基本的 `INSERT ... VALUES` 版本相同:

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

ClickHouse 在数据之前， 删除所有空格和换行(如果有)。 当形成一个查询时， 我们推荐在查询操作符之后将数据放入新行(如果数据以空格开始， 这是重要的)。

示例:

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

你能够单独从查询中插入数据，通过命令行或 HTTP 接口. 进一步信息， 参见 "Interfaces [#interfaces]".

Inserting The Results of `SELECT`

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

在 SELECT语句中， 根据字段的位置来映射。 然而， 在SELECT表达式中的名称和表名可能不同。 如果必要， 可以进行类型转换。

除了值以外没有其他数据类型允许设置值到表达式中， 例如 `now()` ， `1 + 2` ， 等。 值格式允许使用有限制的表达式， 但是它并不推荐， 因为在这种情况下， 执行了低效的代码。

不支持修改数据分区的查询如下: `UPDATE` ， `DELETE` ， `REPLACE` ， `MERGE` ， `UPSERT` ， `INSERT UPDATE` 。 然而， 你能够使用 `ALTER TABLE ... DROP PARTITION` 来删除旧数据。

Performance Considerations

`INSERT` 通过主键来排序数据， 并通过月份来拆分数据到每个分区中。 如果插入的数据有混合的月份， 会显著降低 `INSERT` 插入的性能。 应该避免此类操作:

- 大批量地添加数据， 如每次 100，000 行。
- 在上传数据之前， 通过月份分组数据。

下面操作性能不会下降:

- 数据实时插入。
- 上传的数据通过时间来排序。

创建数据库

创建 `db_name` 数据库。

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

数据库是一个包含多个表的目录，如果在CREATE DATABASE语句中包含 `IF NOT EXISTS` ，则在数据库已经存在的情况下查询也不会返回错误。

[]

创建表

`CREATE TABLE` 语句有几种形式.

```
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
```

如果 `db` 没有设置， 在数据库 `db` 中或者当前数据库中， 创建一个表名为 `name` 的表， 在括号和 `engine` 引擎中指定结构。 表的结构是一个列描述的列表。 如果引擎支持索引， 则他们将是表引擎的参数。

表结构是一个列描述的列表。 如果引擎支持索引， 他们以表引擎的参数表示。

在最简单的情况， 一个列描述是'命名类型'。 例如: RegionID UInt32。 对于默认值， 表达式也能够被定义。

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] [db.]name AS [db2.]name2 [ENGINE = engine]
```

创建一个表， 其结构与另一个表相同。 你能够为此表指定一个不同的引擎。 如果引擎没有被指定， 相同的引擎将被用于 `db2。name2` 表上。

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] [db.]name ENGINE = engine AS SELECT ...
```

创建一个表，其结构类似于 SELECT 查询后的结果， 带有 `engine` 引擎， 从 SELECT查询数据填充它。

在所有情况下，如果 `IF NOT EXISTS` 被指定， 如果表已经存在， 查询并不返回一个错误。 在这种情况下， 查询并不做任何事情。

默认值

列描述能够为默认值指定一个表达式， 其中一个方法是:DEFAULT expr， MATERIALIZED expr， ALIAS expr。 例如: URLDomain String DEFAULT domain(URL)。

如果默认值的一个表达式没有定义， 如果字段是数字类型， 默认值是将设置为0， 如果是字符类型， 则设置为空字符串， 日期类型则设置为 0000-00-00 或者 0000-00-00 00:00:00(时间戳)。 NULLs 则不支持。

如果默认表达式被定义， 字段类型是可选的。 如果没有明确的定义类型， 则将使用默认表达式。 例如: EventDate DEFAULT toDate(EventTime) – `Date` 类型将用于 `EventDate` 字段。

如果数据类型和默认表达式被明确定义， 此表达式将使用函数被转换为特定的类型。 例如: Hits UInt32 DEFAULT 0 与 Hits UInt32 DEFAULT toUInt32(0)是等价的。

默认表达是可能被定义为一个任意的表达式，如表的常量和字段。 当创建和更改表结构时， 它将检查表达式是否包含循环。 对于 INSERT操作来说， 它将检查表达式是否可解析 – 所有的字段通过传参后进行计算。

`DEFAULT expr`

正常的默认值。 如果 INSERT 查询并没有指定对应的字段， 它将通过计算对应的表达式来填充。

`物化表达式`

物化表达式。 此类型字段并没有指定插入操作， 因为它经常执行计算任务。 对一个插入操作， 无字段列表， 那么这些字段将不考虑。 另外， 当在一个SELECT查询语句中使用星号时， 此字段并不被替换。 这将保证INSERT INTO SELECT * FROM 的不可变性。

`别名表达式`

别名。 此字段不存储在表中。 此列的值不插入到表中， 当在一个SELECT查询语句中使用星号时，此字段并不被替换。 它能够用在 SELECTs中，如果别名在查询解析时被扩展。

当使用更新查询添加一个新的字段， 这些列的旧值不被写入。 相反， 新字段没有值，当读取旧值时， 表达式将被计算。 然而，如果运行表达式需要不同的字段， 这些字段将被读取 ， 但是仅读取相关的数据块。

如果你添加一个新的字段到表中， 然后改变它的默认表达式， 对于使用的旧值将更改(对于此数据， 值不保存在磁盘上)。 当运行背景线程时， 缺少合并数据块的字段数据写入到合并数据块中。

在嵌套数据结构中设置默认值是不允许的。

临时表

在任何情况下， 如果临时表被指定， 一个临时表将被创建。 临时表有如下的特性:

- 当会话结束后， 临时表将删除，或者连接丢失。
- 一个临时表使用内存表引擎创建。 其他的表引擎不支持临时表。
- 数据库不能为一个临时表指定。 它将创建在数据库之外。
- 如果一个临时表与另外的表有相同的名称 ， 一个查询指定了表名并没有指定数据库， 将使用临时表。
- 对于分布式查询处理， 查询中的临时表将被传递给远程服务器。

在大多数情况下， 临时表并不能手工创建， 但当查询外部数据或使用分布式全局(GLOBAL)IN时，可以创建临时表。

分布式 DDL 查询 (ON CLUSTER clause)

`CREATE`， `DROP`， `ALTER`， 和 `RENAME` 查询支持在集群上分布式执行。例如， 如下的查询在集群中的每个机器节点上创建了 all_hits Distributed 表:

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

为了正确执行这些语句，每个节点必须有相同的集群设置(为了简化同步配置，可以使用 zookeeper 来替换)。 这些节点也可以连接到ZooKeeper 服务器。 查询语句会在每个节点上执行， 而 `ALTER` 查询目前暂不支持在同步表(replicated table)上执行。

CREATE VIEW

```
CREATE [MATERIALIZED] VIEW [IF NOT EXISTS] [db.]name [TO[db.]name] [ENGINE = engine] [POPULATE] AS SELECT ...
```

创建一个视图。 有两种类型的视图: 正常视图和物化(MATERIALIZED)视图。

当创建一个物化视图时， 你必须指定表引擎 – 此表引擎用于存储数据

一个物化视图工作流程如下所示: 当插入数据到SELECT 查询指定的表中时， 插入数据部分通过SELECT查询部分来转换， 结果插入到视图中。

正常视图不保存任何数据， 但是可以从任意表中读取数据。 换句话说，正常视图可以看作是查询结果的一个结果缓存。 当从一个视图中读取数据时， 此查询可以看做是 FROM语句的子查询。

例如， 假设你已经创建了一个视图:

```
CREATE VIEW view AS SELECT ...
```

写了一个查询语句:

```
SELECT a, b, c FROM view
```

此查询完全等价于子查询:

```
SELECT a, b, c FROM (SELECT ...)
```

物化视图保存由SELECT语句查询转换的数据。

当创建一个物化视图时，你必须指定一个引擎 – 存储数据的目标引擎。

一个物化视图使用流程如下: 当插入数据到 SELECT 指定的表时， 插入数据部分通过SELECT 来转换， 同时结果被插入到视图中。

如果你指定了 POPULATE， 当创建时， 现有的表数据被插入到了视图中， 类似于 `CREATE TABLE ... AS SELECT ...` . 否则， 在创建视图之后，查询仅包含表中插入的数据. 我们不建议使用 POPULATE， 在视图创建过程中，插入到表中的数据不插入到其中.

一个 `SELECT` 查询可以包含 `DISTINCT`， `GROUP BY`， `ORDER BY`， `LIMIT` 。。。 对应的转换在每个数据块上独立执行。 例如， 如果 GROUP BY 被设置， 数据将在插入过程中进行聚合， 但仅是在一个插入数据包中。数据不再进一步聚合。 当使用一个引擎时， 如SummingMergeTree，它将独立执行数据聚合。

视图看起来和正常表相同。 例如， 你可以使用 SHOW TABLES来列出视图表的相关信息。

物化视图的 `ALTER` 查询执行还没有完全开发出来， 因此使用上可能不方便。 如果物化视图使用 `TO [db。]name`， 你能够 `DETACH` 视图， 在目标表运行 `ALTER`， 然后 `ATTACH` 之前的 `DETACH` 视图。

视图看起来和正常表相同。 例如， 你可以使用 `SHOW TABLES` 来列出视图表的相关信息。

因此并没有一个单独的SQL语句来删除视图。 为了删除一个视图， 可以使用 `DROP TABLE` 。

[]

ALTER

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

Column Manipulations

Changing the table structure.

```
ALTER TABLE [db].name [ON CLUSTER cluster] ADD|DROP|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_expr` (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). 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 processing stages:

- Preparing temporary (new) files with modified data.
- Renaming old files.
- Renaming the temporary (new) files to the old names.
- Deleting the old files.

Only the first stage takes time. If there is a failure at this stage, the data is not changed. If there is a failure during one of the successive stages, data can be restored manually. The exception is if the old files were deleted from the file system but the data for the new files did not get written to the disk and was lost.

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 columns in the primary key or the sampling key (columns that are in the `ENGINE` expression). Changing the type for columns that are included in the primary key is only possible if this change does not cause the data to be modified (for example, it is allowed to add values to an Enum or change a type with `DateTime` to `UInt32`).

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 it to complete. At the same time, all new queries to the same table will wait while this `ALTER` is running.

For tables that don't store data themselves (such as `Merge` and `Distributed`), `ALTER` just changes the table structure, and does not change the structure of subordinate tables. For example, when running ALTER for a `Distributed` table, you will also need to run `ALTER` for the tables on all remote servers.

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

It 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. Data directory: `/var/lib/clickhouse/data/database/table/` ,where `/var/lib/clickhouse/` is the path to the 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 May  5 02:58 20140317_20140323_2_2_0
drwxrwxrwx 2 clickhouse clickhouse 20480 May  5 02:58 20140317_20140323_4_4_0
drwxrwxrwx 2 clickhouse clickhouse  4096 May  5 02:55 detached
-rw-rw-rw- 1 clickhouse clickhouse     2 May  5 02:58 increment.txt
```

Here, `20140317_20140323_2_2_0` and `20140317_20140323_4_4_0` are the directories of data parts.

Let's break down the name of the first part: `20140317_20140323_2_2_0` .

- `20140317` is the minimum date of the data in the chunk.
- `20140323` is the maximum date of the data in the chunk.
- `2` is the minimum number of the data block.
- `2` is the maximum number of the data block.
- `0` is the chunk level (the depth of the merge tree it is formed from).

Each piece relates to a single partition and contains data for just one month. `201403` is the name of the partition. 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'
```

The same as 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.

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 the 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 `FROM` clause specifies the path in `ZooKeeper`. 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` – only wait for own execution (default); `2` – wait for all.

[]

Mutations

Mutations are an ALTER query variant that allows changing or deleting rows in a table. In contrast to standard `UPDATE` and `DELETE` queries that are intended for point data changes, mutations are intended for heavy operations that change a lot of rows in a table.

The functionality is in beta stage and is available starting with the 1.1.54388 version. Currently *MergeTree table engines are supported (both replicated and unreplicated).

Existing tables are ready for mutations as-is (no conversion necessary), but after the first mutation is applied to a table, its metadata format becomes incompatible with previous server versions and falling back to a previous version becomes impossible.

Currently available commands:

```
ALTER TABLE [db.]table DELETE WHERE filter_expr
```

The `filter_expr` must be of type UInt8. The query deletes rows in the table for which this expression takes a non-zero value.

```
ALTER TABLE [db.]table UPDATE column1 = expr1 [, ...] WHERE filter_expr
```

The command is available starting with the 18.12.14 version. The `filter_expr` must be of type UInt8. This query updates values of specified columns to the values of corresponding expressions in rows for which the `filter_expr` takes a non-zero value. Values are casted to the column type using the `CAST` operator. Updating columns that are used in the calculation of the primary or the partition key is not supported.

One query can contain several commands separated by commas.

For *MergeTree tables mutations execute by rewriting whole data parts. There is no atomicity - parts are substituted for mutated parts as soon as they are ready and a `SELECT` query that started executing during a mutation will see data from parts that have already been mutated along with data from parts that have not been mutated yet.

Mutations are totally ordered by their creation order and are applied to each part in that order. Mutations are also partially ordered with INSERTs - data that was inserted into the table before the mutation was submitted will be mutated and data that was inserted after that will not be mutated. Note that mutations do not block INSERTs in any way.

A mutation query returns immediately after the mutation entry is added (in case of replicated tables to ZooKeeper, for nonreplicated tables - to the filesystem). The mutation itself executes asynchronously using the system profile settings. To track the progress of mutations you can use the `system.mutations` table. A mutation that was successfully submitted will continue to execute even if ClickHouse servers are restarted. There is no way to roll back the mutation once it is submitted.

Entries for finished mutations are not deleted right away (the number of preserved entries is determined by the `finished_mutations_to_keep` storage engine parameter). Older mutation entries are deleted.

SYSTEM.MUTATIONS TABLE

The table contains information about mutations of MergeTree tables and their progress. Each mutation command is represented by a single row. The table has the following columns:

database, **table** - The name of the database and table to which the mutation was applied.

mutation_id - The ID of the mutation. For replicated tables these IDs correspond to znode names in the `<table_path_in_zookeeper>/mutations/` directory in ZooKeeper. For unreplicated tables the IDs correspond to file names in the data directory of the table.

command - The mutation command string (the part of the query after `ALTER TABLE [db.]table`).

create_time - When this mutation command was submitted for execution.

block_numbers.partition_id, **block_numbers.number** - A Nested column. For mutations of replicated tables contains one record for each partition: the partition ID and the block number that was acquired by the mutation (in each partition only parts that contain blocks with numbers less than the block number acquired by the mutation in that partition will be mutated). Because in non-replicated tables blocks numbers in all partitions form a single sequence, for mutatations of non-replicated tables the column will contain one record with a single block number acquired by the mutation.

parts_to_do - The number of data parts that need to be mutated for the mutation to finish.

is_done - Is the mutation done? Note that even if `parts_to_do = 0` it is possible that a mutation of a replicated table is not done yet because of a long-running INSERT that will create a new data part that will need to be mutated.

Miscellaneous Queries

ATTACH

This query is exactly the same as `CREATE` , but

- instead of the word `CREATE` it uses the word `ATTACH` .
- 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.

If the table was previously detached (`DETACH`), meaning that its structure is known, you can use shorthand without defining the structure.

```
ATTACH TABLE [IF NOT EXISTS] [db.]name [ON CLUSTER cluster]
```

This query is used when starting the server. The server stores table metadata as files with `ATTACH` queries, which it simply runs at launch (with the exception of system tables, which are explicitly created on the server).

DROP

This query has two types: `DROP DATABASE` and `DROP TABLE` .

```
DROP DATABASE [IF EXISTS] db [ON CLUSTER cluster]
```

Deletes all tables inside the 'db' database, then deletes the 'db' database itself. If `IF EXISTS` is specified, it doesn't return an error if the database doesn't exist.

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

Deletes the table. If `IF EXISTS` is specified, it doesn't return an error if the table doesn't exist or the database doesn't exist.

DETACH

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

```
DETACH TABLE [IF EXISTS] [db.]name [ON CLUSTER cluster]
```

This does not delete the table's data or metadata. On the next server launch, the server will read the metadata and find out about the table again. Similarly, a "detached" table can be re-attached using the `ATTACH` query (with the exception of system tables, which do not have metadata stored for them).

There is no `DETACH DATABASE` query.

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).

SHOW DATABASES

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

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

See also the section "Formats".

SHOW TABLES

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

Displays a list of tables

- tables from the current database, or from the 'db' database if "FROM db" is specified.
- all tables, or tables whose name matches the pattern, if "LIKE 'pattern'" is specified.

This query is identical to: `SELECT name FROM system.tables WHERE database = 'db' [AND name LIKE 'pattern'] [INTO OUTFILE filename] [FORMAT format]`.

See also the section "LIKE operator".

SHOW PROCESSLIST

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

Outputs a list of queries currently being processed, other than `SHOW PROCESSLIST` queries.

Prints a table containing the columns:

user – The user who made the query. Keep in mind that for distributed processing, queries are sent to remote servers under the 'default' user. SHOW PROCESSLIST shows the username for a specific query, not for a query that this query initiated.

address – The name of the host that the query was sent from. For distributed processing, on remote servers, this is the name of the query requestor host. To track where a distributed query was originally made from, look at SHOW PROCESSLIST on the query requestor server.

elapsed – The execution time, in seconds. Queries are output in order of decreasing execution time.

rows_read, **bytes_read** – How many rows and bytes of uncompressed data were read when processing the query. For distributed processing, data is totaled from all the remote servers. This is the data used for restrictions and quotas.

memory_usage – Current RAM usage in bytes. See the setting 'max_memory_usage'.

query – The query itself. In INSERT queries, the data for insertion is not output.

query_id – The query identifier. Non-empty only if it was explicitly defined by the user. For distributed processing, the query ID is not passed to remote servers.

This query is identical to: `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 [TEMPORARY] TABLE [db.]table [INTO OUTFILE filename] [FORMAT format]
```

Returns a single `String` -type 'statement' column, which contains a single value – the `CREATE` query used for creating the specified table.

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 [TEMPORARY] TABLE [db.]name [INTO OUTFILE filename] [FORMAT format]
```

Returns a single `UInt8` -type column, which contains the single value `0` if the table or database doesn't exist, or `1` if the table exists in the specified database.

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 param = value
```

Allows you to set `param` to `value` . You can also make all the settings from the specified settings profile in a single query. To do this, specify 'profile' as the setting name. For more information, see the section "Settings". The setting is made for the session, or for the server (globally) if `GLOBAL` is specified. When making a global setting, the setting is not applied to sessions already running, including the current session. It will only be used for new sessions.

When the server is restarted, global settings made using `SET` are lost. To make settings that persist after a server restart, you can only use the server's config file.

OPTIMIZE

```
OPTIMIZE TABLE [db.]name [ON CLUSTER cluster] [PARTITION partition] [FINAL]
```

Asks the table engine to do something for optimization. Supported only by `*MergeTree` engines, in which this query initializes a non-scheduled merge of data parts. If you specify a `PARTITION` , only the specified partition will be optimized. If you specify `FINAL` , optimization will be performed even when all the data is already in one part.

Warning

OPTIMIZE can't fix the "Too many parts" error.

KILL QUERY

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

Attempts to forcibly terminate the currently running queries. The queries to terminate are selected from the system.processes table using the criteria defined in the `WHERE` clause of the `KILL` query.

Examples:

```
-- Forcibly terminates all queries with the specified query_id:
KILL QUERY WHERE query_id='2-857d-4a57-9ee0-327da5d60a90'

-- Synchronously terminates all queries run by 'username':
KILL QUERY WHERE user='username' SYNC
```

Read-only users can only stop their own queries.

By default, the asynchronous version of queries is used (`ASYNC`), which doesn't wait for confirmation that queries have stopped.

The synchronous version (`SYNC`) waits for all queries to stop and displays information about each process as it stops. The response contains the `kill_status` column, which can take the following values:

1. 'finished' – The query was terminated successfully.
2. 'waiting' – Waiting for the query to end after sending it a signal to terminate.
3. The other values explain why the query can't be stopped.

A test query (`TEST`) only checks the user's rights and displays a list of queries to stop.

Functions

There are at least* two types of functions - regular functions (they are just called "functions") and aggregate functions. These are completely different concepts. Regular functions work as if they are applied to each row separately (for each row, the result of the function doesn't depend on the other rows). Aggregate functions accumulate a set of values from various rows (i.e. they depend on the entire set of rows).

In this section we discuss regular functions. For aggregate functions, see the section "Aggregate functions".

* - There is a third type of function that the 'arrayJoin' function belongs to; table functions can also be mentioned separately.*

Strong typing

In contrast to standard SQL, ClickHouse has strong typing. In other words, it doesn't make implicit conversions between types. Each function works for a specific set of types. This means that sometimes you need to use type conversion functions.

Common subexpression elimination

All expressions in a query that have the same AST (the same record or same result of syntactic parsing) are considered to have identical values. Such expressions are concatenated and executed once. Identical subqueries are also eliminated this way.

Types of results

All functions return a single 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.

NULL processing

Functions have the following behaviors:

- If at least one of the arguments of the function is `NULL` , the function result is also `NULL` .
- Special behavior that is specified individually in the description of each function. In the ClickHouse source code, these functions have `UseDefaultImplementationForNulls=false` .

Constancy

Functions can't change the values of their arguments – any changes are returned as the result. Thus, the result of calculating separate functions does not depend on the order in which the functions are written in the query.

Error handling

Some functions might throw an exception if the data is invalid. In this case, the query is canceled and an error text is returned to the client. For distributed processing, when an exception occurs on one of the servers, the other servers also attempt to abort the query.

Evaluation of argument expressions

In almost all programming languages, one of the arguments might not be evaluated for certain operators. This is usually the operators `&&`, `||`, and `?:`. But in ClickHouse, arguments of functions (operators) are always evaluated. This is because entire parts of columns are evaluated at once, instead of calculating each row separately.

Performing functions for distributed query processing

For distributed query processing, as many stages of query processing as possible are performed on remote servers, and the rest of the stages (merging intermediate results and everything after that) are performed on the requestor server.

This means that functions can be performed on different servers. For example, in the query `SELECT f(sum(g(x))) FROM distributed_table GROUP BY h(y),`

- if a `distributed_table` has at least two shards, the functions 'g' and 'h' are performed on remote servers, and the function 'f' is performed on the requestor server.
- if a `distributed_table` has only one shard, all the 'f', 'g', and 'h' functions are performed on this shard's server.

The result of a function usually doesn't depend on which server it is performed on. However, sometimes this is important. For example, functions that work with dictionaries use the dictionary that exists on the server they are running on. Another example is the `hostName` function, which returns the name of the server it is running on in order to make `GROUP BY` by servers in a `SELECT` query.

If a function in a query is performed on the requestor server, but you need to perform it on remote servers, you can wrap it in an 'any' aggregate function or add it to a key in `GROUP BY`.

Arithmetic functions

For all arithmetic functions, the result type is calculated as the smallest number type that the result fits in, if there is such a type. The minimum is taken simultaneously based on the number of bits, whether it is signed, and whether it floats. If there are not enough bits, the highest bit type is taken.

Example:

```
SELECT toTypeName(0), toTypeName(0 + 0), toTypeName(0 + 0 + 0), toTypeName(0 + 0 + 0 + 0)
```

```
┌─toTypeName(0)─┬─toTypeName(plus(0, 0))─┬─toTypeName(plus(plus(0, 0), 0))─┬─toTypeName(plus(plus(plus(0, 0), 0), 0))─┐
│ UInt8         │ UInt16                 │ UInt32                          │ UInt64                                   │
└───────────────┴────────────────────────┴─────────────────────────────────┴──────────────────────────────────────────┘
```

Arithmetic functions work for any pair of types from UInt8, UInt16, UInt32, UInt64, Int8, Int16, Int32, Int64, Float32, or Float64.

Overflow is produced the same way as in C++.

plus(a, b), a + b operator

Calculates the sum of the numbers. You can also add integer numbers with a date or date and time. In the case of a date, adding an integer means adding the corresponding number of days. For a date with time, it means adding the corresponding number of seconds.

minus(a, b), a - b operator

Calculates the difference. The result is always signed.

You can also calculate integer numbers from a date or date with time. The idea is the same – see above for 'plus'.

multiply(a, b), a * b operator

Calculates the product of the numbers.

divide(a, b), a / b operator

Calculates the quotient of the numbers. The result type is always a floating-point type. It is not integer division. For integer division, use the 'intDiv' function. When dividing by zero you get 'inf', '-inf', or 'nan'.

intDiv(a, b)

Calculates the quotient of the numbers. Divides into integers, rounding down (by the absolute value). An exception is thrown when dividing by zero or when dividing a minimal negative number by minus one.

intDivOrZero(a, b)

Differs from 'intDiv' in that it returns zero when dividing by zero or when dividing a minimal negative number by minus one.

modulo(a, b), a % b operator

Calculates the remainder after division. If arguments are floating-point numbers, they are pre-converted to integers by dropping the decimal portion. The remainder is taken in the same sense as in C++. Truncated division is used for negative numbers. An exception is thrown when dividing by zero or when dividing a minimal negative number by minus one.

negate(a), -a operator

Calculates a number with the reverse sign. The result is always signed.

abs(a)

Calculates the absolute value of the number (a). That is, if a < 0, it returns -a. For unsigned types it doesn't do anything. For signed integer types, it returns an unsigned number.

gcd(a, b)

Returns the greatest common divisor of the numbers. An exception is thrown when dividing by zero or when dividing a minimal negative number by minus one.

lcm(a, b)

Returns the least common multiple of the numbers. An exception is thrown when dividing by zero or when dividing a minimal negative number by minus one.

Comparison functions

Comparison functions always return 0 or 1 (Uint8).

The following types can be compared:

- numbers
- strings and fixed strings
- dates
- dates with times

within each group, but not between different groups.

For example, you can't compare a date with a string. You have to use a function to convert the string to a date, or vice versa.

Strings are compared by bytes. A shorter string is smaller than all strings that start with it and that contain at least one more character.

Note. Up until version 1.1.54134, signed and unsigned numbers were compared the same way as in C++. In other words, you could get an incorrect result in cases like SELECT 9223372036854775807 > -1. This behavior changed in version 1.1.54134 and is now mathematically correct.

equals, a = b and a == b operator

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

less, `< operator`

greater, `> operator`

lessOrEquals, `<= operator`

greaterOrEquals, `>= operator`

Logical functions

Logical functions accept any numeric types, but return a UInt8 number equal to 0 or 1.

Zero as an argument is considered "false," while any non-zero value is considered "true".

and, AND operator

or, OR operator

not, NOT operator

xor

[]

Type conversion functions

toUInt8, toUInt16, toUInt32, toUInt64

toInt8, toInt16, toInt32, toInt64

toFloat32, toFloat64

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

toDate, toDateTime

toDecimal32(value, S), toDecimal64(value, S), toDecimal128(value, S)

Converts `value` to Decimal [#data_type-decimal] of precision `S`. The `value` can be a number or a string. The `S` (scale) parameter specifies the number of decimal places.

toString

Functions for converting between numbers, strings (but not fixed strings), dates, and dates with times. All these functions accept one argument.

When converting to or from a string, the value is formatted or parsed using the same rules as for the TabSeparated format (and almost all other text formats). If the string can't be parsed, an exception is thrown and the request is canceled.

When converting dates to numbers or vice versa, the date corresponds to the number of days since the beginning of the Unix epoch. When converting dates with times to numbers or vice versa, the date with time corresponds to the number of seconds since the beginning of the Unix epoch.

The date and date-with-time formats for the toDate/toDateTime functions are defined as follows:

```
YYYY-MM-DD
YYYY-MM-DD hh:mm:ss
```

As an exception, if converting from UInt32, Int32, UInt64, or Int64 numeric types to Date, and if the number is greater than or equal to 65536, the number is interpreted as a Unix timestamp (and not as the number of days) and is rounded to the date. This allows support for the common occurrence of writing 'toDate(unix_timestamp)', which otherwise would be an error and would require writing the more cumbersome 'toDate(toDateTime(unix_timestamp))'.

Conversion between a date and date with time is performed the natural way: by adding a null time or dropping the time.

Conversion between numeric types uses the same rules as assignments between different numeric types in C++.

Additionally, the toString function of the DateTime argument can take a second String argument containing the name of the time zone. Example: `Asia/Yekaterinburg` In this case, the time is formatted according to the specified time zone.

```
SELECT
    now() AS now_local,
    toString(now(), 'Asia/Yekaterinburg') AS now_yekat
```

```
┌───────────now_local─┬─now_yekat───────────┐
│ 2016-06-15 00:11:21 │ 2016-06-15 02:11:21 │
└─────────────────────┴─────────────────────┘
```

Also see the `toUnixTimestamp` function.

toFixedString(s, N)

Converts a String type argument to a FixedString(N) type (a string with fixed length N). N must be a constant. If the string has fewer bytes than N, it is passed with null bytes to the right. If the string has more bytes than N, an exception is thrown.

toStringCutToZero(s)

Accepts a String or FixedString argument. Returns the String with the content truncated at the first zero byte found.

Example:

```
SELECT toFixedString('foo', 8) AS s, toStringCutToZero(s) AS s_cut
```

```
┌─s─────────────┬─s_cut─┐
│ foo\0\0\0\0\0 │ foo   │
└───────────────┴───────┘
```

```
SELECT toFixedString('foo\0bar', 8) AS s, toStringCutToZero(s) AS s_cut
```

```
┌─s──────────┬─s_cut─┐
│ foo\0bar\0 │ foo   │
└────────────┴───────┘
```

reinterpretAsUInt8, reinterpretAsUInt16, reinterpretAsUInt32, reinterpretAsUInt64

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)

Converts 'x' to the 't' data type. The syntax CAST(x AS t) is also supported.

Example:

```
SELECT
    '2016-06-15 23:00:00' AS timestamp,
    CAST(timestamp AS DateTime) AS datetime,
    CAST(timestamp AS Date) AS date,
    CAST(timestamp, 'String') AS string,
    CAST(timestamp, 'FixedString(22)') AS fixed_string
```

```
┌─timestamp───────────┬────────────datetime─┬───────date─┬─string──────────────┬─fixed_string──────────────┐
│ 2016-06-15 23:00:00 │ 2016-06-15 23:00:00 │ 2016-06-15 │ 2016-06-15 23:00:00 │ 2016-06-15 23:00:00\0\0\0 │
└─────────────────────┴─────────────────────┴────────────┴─────────────────────┴───────────────────────────┘
```

Conversion to FixedString(N) only works for arguments of type String or FixedString(N).

Type conversion to Nullable [#data_type-nullable] and back is supported. Example:

```
SELECT toTypeName(x) FROM t_null

┌─toTypeName(x)─┐
│ Int8          │
│ Int8          │
└───────────────┘

SELECT toTypeName(CAST(x, 'Nullable(UInt16)')) FROM t_null

┌─toTypeName(CAST(x, 'Nullable(UInt16)'))─┐
│ Nullable(UInt16)                        │
│ Nullable(UInt16)                        │
└─────────────────────────────────────────┘
```

Functions for working with dates and times

Support for time zones

All functions for working with the date and time that have a logical use for the time zone can accept a second optional time zone argument. Example: Asia/Yekaterinburg. In this case, they use the specified time zone instead of the local (default) one.

```
SELECT
    toDateTime('2016-06-15 23:00:00') AS time,
    toDate(time) AS date_local,
    toDate(time, 'Asia/Yekaterinburg') AS date_yekat,
    toString(time, 'US/Samoa') AS time_samoa
```

```
┌────────────────time─┬─date_local─┬─date_yekat─┬─time_samoa──────────┐
│ 2016-06-15 23:00:00 │ 2016-06-15 │ 2016-06-16 │ 2016-06-15 09:00:00 │
└─────────────────────┴────────────┴────────────┴─────────────────────┘
```

Only time zones that differ from UTC by a whole number of hours are supported.

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 twice changed at a different time).

toMinute

Converts a date with time to a UInt8 number containing the number of the minute of the hour (0-59).

toSecond

Converts a date with time to a UInt8 number containing the number of the second in the minute (0-59). Leap seconds are not accounted for.

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.

Attention

The behavior of parsing incorrect dates is implementation specific. ClickHouse may return zero date, throw an exception or do "natural" overflow.

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 hour.

toStartOfFifteenMinutes

Rounds down the date with time to the start of the fifteen-minute interval.

Note: If you need to round a date with time to any other number of seconds, minutes, or hours, you can convert it into a number by using the toUInt32 function, then round the number using intDiv and multiplication, and convert it back using the toDateTime function.

toStartOfHour

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

toStartOfDay

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

toTime

Converts a date with time to a certain fixed date, while preserving the time.

toRelativeYearNum

Converts a date with time or date to the number of the year, starting from a certain fixed point in the past.

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

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'), 600) = [toDateTime('2012-01-01 12:00:00'), toDateTime('2012-01-01 12:30:00')]` . This is necessary for searching for pageviews in the corresponding session.

formatDateTime(Time, Format[, Timezone])

Function formats a Time according given Format string. N.B.: Format is a constant expression, e.g. you can not have multiple formats for single result column.

Supported modifiers for Format: ("Example" column shows formatting result for time `2018-01-02 22:33:44`)

Modifier	Description	Example
%C	year divided by 100 and truncated to integer (00-99)	20
%d	day of the month, zero-padded (01-31)	02
%D	Short MM/DD/YY date, equivalent to %m/%d/%y	01/02/2018
%e	day of the month, space-padded (1-31)	2
%F	short YYYY-MM-DD date, equivalent to %Y-%m-%d	2018-01-02
%H	hour in 24h format (00-23)	22
%I	hour in 12h format (01-12)	10
%j	day of the year (001-366)	002
%m	month as a decimal number (01-12)	01
%M	minute (00-59)	33
%n	new-line character ('\n')	
%p	AM or PM designation	PM
%R	24-hour HH:MM time, equivalent to %H:%M	22:33
%S	second (00-59)	44
%t	horizontal-tab character ('\t')	
%T	ISO 8601 time format (HH:MM:SS), equivalent to %H:%M:%S	22:33:44
%u	ISO 8601 weekday as number with Monday as 1 (1-7)	2
%V	ISO 8601 week number (01-53)	01
%w	weekday as a decimal number with Sunday as 0 (0-6)	2
%y	Year, last two digits (00-99)	18
%Y	Year	2018
%%	a % sign	%

Functions for working with strings

empty

Returns 1 for an empty string or 0 for a non-empty string. The result type is UInt8. A string is considered non-empty if it contains at least one byte, even if this is a space or a null byte. The function also works for arrays.

notEmpty

Returns 0 for an empty string or 1 for a non-empty string. The result type is UInt8. The function also works for arrays.

length

Returns the length of a string in bytes (not in characters, and not in code points). The result type is UInt64. The function also works for arrays.

lengthUTF8

Returns the length of a string in Unicode code points (not in characters), assuming that the string contains a set of bytes that make up UTF-8 encoded text. If this assumption is not met, it returns some result (it doesn't throw an exception). The result type is UInt64.

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 the length of the UTF-8 byte sequence is different for upper and lower case of a code point, the result may be incorrect for this code point. If the string contains a set of bytes that is not UTF-8, then the behavior is undefined.

upperUTF8

Converts a string to uppercase, assuming the string contains a set of bytes that make up a UTF-8 encoded text. It doesn't detect the language. So for Turkish the result might not be exactly correct. If the length of the UTF-8 byte sequence is different for upper and lower case of a code point, the result may be incorrect for this code point. If the string contains a set of bytes that is not UTF-8, then the behavior is undefined.

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 the strings listed in the 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 the string 's' that was converted from the encoding in 'from' to the encoding in 'to'.

base64Encode(s)

Encodes 's' string into base64

base64Decode(s)

Decode base64-encoded string 's' into original string. In case of failure raises an exception.

tryBase64Decode(s)

Similar to base64Decode, but in case of error an empty string would be returned.

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)

Search for the substring `needle` in the string `haystack`. Returns the position (in bytes) of the found substring, starting from 1, or returns 0 if the substring was not found.

For a case-insensitive search, use the function `positionCaseInsensitive`.

positionUTF8(haystack, needle)

The same as `position`, but the position is returned in Unicode code points. Works under the assumption that the string contains a set of bytes representing a UTF-8 encoded text. If this assumption is not met, it returns some result (it doesn't throw an exception).

For a case-insensitive search, use the function `positionCaseInsensitiveUTF8`.

match(haystack, pattern)

Checks whether the string matches the `pattern` regular expression. A `re2` regular expression. The syntax [https://github.com/google/re2/wiki/Syntax] of the `re2` regular expressions is more limited than the syntax of the Perl regular expressions.

Returns 0 if it doesn't match, or 1 if it matches.

Note that the backslash symbol (`\`) is used for escaping in the regular expression. The same symbol is used for escaping in string literals. So in order to escape the symbol in a regular expression, you must write two backslashes (\) in a string literal.

The regular expression works with the string as if it is a set of bytes. The regular expression can't contain null bytes. For patterns to search for substrings in a string, it is better to use LIKE or 'position', since they work much faster.

extract(haystack, pattern)

Extracts a fragment of a string using a regular expression. If 'haystack' doesn't match the 'pattern' regex, an empty string is returned. If the regex doesn't contain subpatterns, it takes the fragment that matches the entire regex. Otherwise, it takes the fragment that matches the first subpattern.

extractAll(haystack, pattern)

Extracts all the fragments of a string using a regular expression. If 'haystack' doesn't match the 'pattern' regex, an empty string is returned. Returns an array of strings consisting of all matches to the regex. In general, the behavior is the same as the 'extract' function (it takes the first subpattern, or the entire expression if there isn't a subpattern).

like(haystack, pattern), haystack LIKE pattern operator

Checks whether a string matches a simple regular expression. The regular expression can contain the metasymbols `%` and `_` .

``% indicates any quantity of any bytes (including zero characters).

`_` indicates any one byte.

Use the backslash (`\`) for escaping metasymbols. See the note on escaping in the description of the 'match' function.

For regular expressions like `%needle%` , the code is more optimal and works as fast as the `position` function. For other regular expressions, the code is the same as for the 'match' function.

notLike(haystack, pattern), haystack NOT LIKE pattern operator

The same thing as 'like', but negative.

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. Substitutions `\1-\9` correspond to the subpattern numbers.To use the `\` character in a template, escape it using `\` . Also keep in mind that a string literal requires an extra escape.

Example 1. Converting the date to American format:

```
SELECT DISTINCT
    EventDate,
    replaceRegexpOne(toString(EventDate), '(\\d{4})-(\\d{2})-(\\d{2})', '\\2/\\3/\\1') AS res
FROM test.hits
LIMIT 7
FORMAT TabSeparated
```

```
2014-03-17      03/17/2014
2014-03-18      03/18/2014
2014-03-19      03/19/2014
2014-03-20      03/20/2014
2014-03-21      03/21/2014
2014-03-22      03/22/2014
2014-03-23      03/23/2014
```

Example 2. Copying a string ten times:

```
SELECT replaceRegexpOne('Hello, World!', '.*', '\\0\\0\\0\\0\\0\\0\\0\\0\\0\\0') AS res
```

```
┌─res────────────────────────────────────────────────────────────────────────────────────────────────────────

│ Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello,
World!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! │
└─────────────────────┘
```

Conditional functions

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

Returns `then` if `cond != 0`, or `else` if `cond = 0`. `cond` must be of type `UInt8`, and `then` and `else` must have the lowest common type.

`then` and `else` can be `NULL`

multiIf

Allows you to write the CASE [#operator_case] operator more compactly in the query.

```
multiIf(cond_1, then_1, cond_2, then_2...else)
```

Parameters:

- `cond_N` — The condition for the function to return `then_N`.
- `then_N` — The result of the function when executed.
- `else` — The result of the function if none of the conditions is met.

The function accepts `2N+1` parameters.

Returned values

The function returns one of the values `then_N` or `else`, depending on the conditions `cond_N`.

Example

Take the table

```
┌─x─┬────y─┐
│ 1 │ NULL │
│ 2 │    3 │
└───┴──────┘
```

Run the query `SELECT multiIf(isNull(y) x, y < 3, y, NULL) FROM t_null`. Result:

```
┌─multiIf(isNull(y), x, less(y, 3), y, NULL)─┐
│                                          1 │
│                                       NULL │
└────────────────────────────────────────────┘
```

Mathematical functions

All the functions return a Float64 number. The accuracy of the result is close to the maximum precision possible, but the result might not coincide with the machine representable number nearest to the corresponding real number.

e()

Returns a Float64 number that is close to the number e.

pi()

Returns a Float64 number that is close to the number π.

exp(x)

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

log(x)

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

exp2(x)

Accepts a numeric argument and returns a Float64 number close to 2 to the power of x.

log2(x)

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

exp10(x)

Accepts a numeric argument and returns a Float64 number close to 10 to the power of x.

log10(x)

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

sqrt(x)

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

cbrt(x)

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

erf(x)

If 'x' is non-negative, then erf(x / σ√2) is the probability that a random variable having a normal distribution with standard deviation 'σ' takes the value that is separated from the expected value by more than 'x'.

Example (three sigma rule):

```
SELECT erf(3 / sqrt(2))
```

```
┌─erf(divide(3, sqrt(2)))─┐
│      0.9973002039367398 │
└─────────────────────────┘
```

erfc(x)

Accepts a numeric argument and returns a Float64 number close to 1 - 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)

Takes two numeric arguments x and y. Returns a Float64 number close to x to the power of y.

Rounding functions

floor(x[, N])

Returns the largest round number that is less than or equal to x. A round number is a multiple of 1/10N, 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])

Implements banker's rounding [https://en.wikipedia.org/wiki/Rounding#Round_half_to_even], i.e., rounding to the nearest even integer.

Function arguments:

- `x` — the number to be rounded. Type [#data_types] — any number.
- `N` — the position of the number after the decimal point to round the number to.

Returned value:

The rounded number of the same type as the input number `x`

Example:

```
SELECT
    number / 2 AS x,
    round(x)
FROM system.numbers
LIMIT 10
```

```
┌───x─┬─round(divide(number, 2))─┐
│   0 │                        0 │
│ 0.5 │                        0 │
│   1 │                        1 │
│ 1.5 │                        2 │
│   2 │                        2 │
│ 2.5 │                        2 │
│   3 │                        3 │
│ 3.5 │                        4 │
│   4 │                        4 │
│ 4.5 │                        4 │
└─────┴──────────────────────────┘
```

roundToExp2(num)

Accepts a number. If the number is less than one, it returns 0. Otherwise, it rounds the number down to the nearest (whole non-negative) degree of two.

roundDuration(num)

Accepts a number. If the number is less than one, it returns 0. Otherwise, it rounds the number down to numbers from the set: 1, 10, 30, 60, 120, 180, 240, 300, 600, 1200, 1800, 3600, 7200, 18000, 36000. This function is specific to Yandex.Metrica and used for implementing the report on session length

roundAge(num)

Accepts a number. If the number is less than 18, it returns 0. Otherwise, it rounds the number down to a number from the set: 18, 25, 35, 45, 55. This function is specific to Yandex.Metrica and used for implementing the report on user age.

Functions for working with arrays

empty

Returns 1 for an empty array, or 0 for a non-empty array. The result type is UInt8. The function also works for strings.

notEmpty

Returns 0 for an empty array, or 1 for a non-empty array. The result type is UInt8. The function also works for strings.

length

Returns the number of items in the array. The result type is UInt64. The function also works for strings.

emptyArrayUInt8, emptyArrayUInt16, emptyArrayUInt32, emptyArrayUInt64

emptyArrayInt8, emptyArrayInt16, emptyArrayInt32, emptyArrayInt64

emptyArrayFloat32, emptyArrayFloat64

emptyArrayDate, emptyArrayDateTime

emptyArrayString

Accepts zero arguments and returns an empty array of the appropriate type.

emptyArrayToSingle

Accepts an empty array and returns a one-element array that is equal to the default value.

range(N)

Returns an array of numbers from 0 to N-1. Just in case, an exception is thrown if arrays with a total length of more than 100,000,000 elements are created in a data block.

array(x1, ...), operator [x1, ...]

Creates an array from the function arguments. The arguments must be constants and have types that have the smallest common type. At least one argument must be passed, because otherwise it isn't clear which type of array to create. That is, you can't use this function to create an empty array (to do that, use the 'emptyArray*' function described above). Returns an 'Array(T)' type result, where 'T' is the smallest common type out of the passed arguments.

arrayConcat

Combines arrays passed as arguments.

```
arrayConcat(arrays)
```

Parameters

- `arrays` – Arbitrary number of arguments of [Array][../../data_types/array.md#data_type-array] type.

Example

```sql
SELECT arrayConcat([1, 2], [3, 4], [5, 6]) AS res
```

```
┌─res───────────┐
│ [1,2,3,4,5,6] │
└───────────────┘
```

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

Get the element with the index `n` from the array `arr`. `n` must be any integer type. Indexes in an array begin from one. Negative indexes are supported. In this case, it selects the corresponding element numbered from the end. For example, `arr[-1]` is the last item in the array.

If the index falls outside of the bounds of an array, it returns some default value (0 for numbers, an empty string for strings, etc.).

has(arr, elem)

Checks whether the 'arr' array has the 'elem' element. Returns 0 if the the element is not in the array, or 1 if it is.

`NULL` is processed as a value.

```
SELECT has([1, 2, NULL], NULL)

┌─has([1, 2, NULL], NULL)─┐
│                       1 │
└─────────────────────────┘
```

hasAll

Checks whether one array is a subset of another.

```
hasAll(set, subset)
```

Parameters

- `set` – Array of any type with a set of elements.
- `subset` – Array of any type with elements that should be tested to be a subset of `set` .

Return values

- `1` , if `set` contains all of the elements from `subset` .
- `0` , otherwise.

Peculiar properties

- An empty array is a subset of any array.
- `Null` processed as a value.
- Order of values in both of arrays doesn't matter.

Examples

`SELECT hasAll([], [])` returns 1.

`SELECT hasAll([1, Null], [Null])` returns 1.

`SELECT hasAll([1.0, 2, 3, 4], [1, 3])` returns 1.

`SELECT hasAll(['a', 'b'], ['a'])` returns 1.

`SELECT hasAll([1], ['a'])` returns 0.

`SELECT hasAll([[1, 2], [3, 4]], [[1, 2], [3, 5]])` returns 0.

hasAny

Checks whether two arrays have intersection by some elements.

```
hasAny(array1, array2)
```

Parameters

- `array1` – Array of any type with a set of elements.

- `array2` – Array of any type with a set of elements.

Return values

- `1`, if `array1` and `array2` have one similar element at least.
- `0`, otherwise.

Peculiar properties

- `Null` processed as a value.
- Order of values in both of arrays doesn't matter.

Examples

`SELECT hasAny([1], [])` returns `0`.

`SELECT hasAny([Null], [Null, 1])` returns `1`.

`SELECT hasAny([-128, 1., 512], [1])` returns `1`.

`SELECT hasAny([[1, 2], [3, 4]], ['a', 'c'])` returns `0`.

`SELECT hasAll([[1, 2], [3, 4]], [[1, 2], [1, 2]])` returns `1`.

indexOf(arr, x)

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

Example:

```
:) SELECT indexOf([1,3,NULL,NULL],NULL)

SELECT indexOf([1, 3, NULL, NULL], NULL)

┌─indexOf([1, 3, NULL, NULL], NULL)─┐
│                                 3 │
└───────────────────────────────────┘
```

Elements set to `NULL` are handled as normal values.

countEqual(arr, x)

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

`NULL` elements are handled as separate values.

Example:

```
SELECT countEqual([1, 2, NULL, NULL], NULL)

┌─countEqual([1, 2, NULL, NULL], NULL)─┐
│                                    2 │
└──────────────────────────────────────┘
```

arrayEnumerate(arr)

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

This function is normally used with ARRAY JOIN. It allows counting something just once for each array after applying ARRAY JOIN. Example:

```sql
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:

```sql
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:

```sql
SELECT
    Goals.ID AS GoalID,
    sum(Sign) AS Reaches,
    sumIf(Sign, num = 1) AS Visits
FROM test.visits
ARRAY JOIN
    Goals,
    arrayEnumerateUniq(Goals.ID) AS num
WHERE CounterID = 160656
GROUP BY GoalID
ORDER BY Reaches DESC
LIMIT 10
```

```
┌──GoalID─┬─Reaches─┬─Visits─┐
│   53225 │    3214 │   1097 │
│ 2825062 │    3188 │   1097 │
│   56600 │    2803 │    488 │
│ 1989037 │    2401 │    365 │
│ 2830064 │    2396 │    910 │
│ 1113562 │    2372 │    373 │
│ 3270895 │    2262 │    812 │
│ 1084657 │    2262 │    345 │
│   56599 │    2260 │    799 │
│ 3271094 │    2256 │    812 │
└─────────┴─────────┴────────┘
```

In this example, each goal ID has a calculation of the number of conversions (each element in the Goals nested data

structure is a goal that was reached, which we refer to as a conversion) and the number of sessions. Without ARRAY JOIN, we would have counted the number of sessions as sum(Sign). But in this particular case, the rows were multiplied by the nested Goals structure, so in order to count each session one time after this, we apply a condition to the value of the arrayEnumerateUniq(Goals.ID) function.

The arrayEnumerateUniq function can take multiple arrays of the same size as arguments. In this case, uniqueness is considered for tuples of elements in the same positions in all the arrays.

```
SELECT arrayEnumerateUniq([1, 1, 1, 2, 2, 2], [1, 1, 2, 1, 1, 2]) AS res
```

```
┌─res───────────┐
│ [1,2,1,1,2,1] │
└───────────────┘
```

This is necessary when using ARRAY JOIN with a nested data structure and further aggregation across multiple elements in this structure.

arrayPopBack

Removes the last item from the array.

```
arrayPopBack(array)
```

Parameters

- `array` – Array.

Example

```
SELECT arrayPopBack([1, 2, 3]) AS res
```

```
┌─res───┐
│ [1,2] │
└───────┘
```

arrayPopFront

Removes the first item from the array.

```
arrayPopFront(array)
```

Parameters

- `array` – Array.

Example

```
SELECT arrayPopFront([1, 2, 3]) AS res
```

```
┌─res───┐
│ [2,3] │
└───────┘
```

arrayPushBack

Adds one item to the end of the array.

```
arrayPushBack(array, single_value)
```

Parameters

- `array` – Array.
- `single_value` – A single value. Only numbers can be added to an array with numbers, and only strings can be added to an array of strings. When adding numbers, ClickHouse automatically sets the `single_value` type for the data type of the array. For more information about the types of data in ClickHouse, see "Data types [#data_types]". Can be `NULL`. The function adds a `NULL` element to an array, and the type of array elements converts to `Nullable`.

Example

```
SELECT arrayPushBack(['a'], 'b') AS res
```

```
┌─res───────┐
│ ['a','b'] │
└───────────┘
```

arrayPushFront

Adds one element to the beginning of the array.

```
arrayPushFront(array, single_value)
```

Parameters

- `array` – Array.
- `single_value` – A single value. Only numbers can be added to an array with numbers, and only strings can be added to an array of strings. When adding numbers, ClickHouse automatically sets the `single_value` type for the data type of the array. For more information about the types of data in ClickHouse, see "Data types [#data_types]". Can be `NULL`. The function adds a `NULL` element to an array, and the type of array elements converts to `Nullable`.

Example

```
SELECT arrayPushBack(['b'], 'a') AS res
```

```
┌─res───────┐
│ ['a','b'] │
└───────────┘
```

arrayResize

Changes the length of the array.

```
arrayResize(array, size[, extender])
```

Parameters:

- `array` — Array.
- `size` — Required length of the array.
  - If `size` is less than the original size of the array, the array is truncated from the right.
- If `size` is larger than the initial size of the array, the array is extended to the right with `extender` values or default values for the data type of the array items.

- `extender` — Value for extending an array. Can be `NULL` .

Returned value:

An array of length `size` .

Examples of calls

```
SELECT arrayResize([1], 3)

┌─arrayResize([1], 3)─┐
│ [1,0,0]             │
└─────────────────────┘
```

```
SELECT arrayResize([1], 3, NULL)

┌─arrayResize([1], 3, NULL)─┐
│ [1,NULL,NULL]             │
└───────────────────────────┘
```

arraySlice

Returns a slice of the array.

```
arraySlice(array, offset[, length])
```

Parameters

- `array` – Array of data.
- `offset` – Indent from the edge of the array. A positive value indicates an offset on the left, and a negative value is an indent on the right. Numbering of the array items begins with 1.
- `length` - The length of the required slice. If you specify a negative value, the function returns an open slice `[offset, array_length - length)` . If you omit the value, the function returns the slice `[offset, the_end_of_array]` .

Example

```
SELECT arraySlice([1, 2, NULL, 4, 5], 2, 3) AS res
```

```
┌─res────────┐
│ [2,NULL,4] │
└────────────┘
```

Array elements set to `NULL` are handled as normal values.

arrayUniq(arr, ...)

If one argument is passed, it counts the number of different elements in the array. If multiple arguments are passed, it counts the number of different tuples of elements at corresponding positions in multiple arrays.

If you want to get a list of unique items in an array, you can use arrayReduce('groupUniqArray', arr).

arrayJoin(arr)

A special function. See the section "ArrayJoin function" [#functions_arrayjoin].

Functions for splitting and merging strings and arrays

splitByChar(separator, s)

Splits a string into substrings separated by 'separator'.'separator' must be a string constant consisting of exactly one character. Returns an array of selected substrings. Empty substrings may be selected if the separator occurs at the beginning or end of the string, or if there are multiple consecutive separators.

splitByString(separator, s)

The same as above, but it uses a string of multiple characters as the separator. The string must be non-empty.

arrayStringConcat(arr[, separator])

Concatenates the strings listed in the array with the separator.'separator' is an optional parameter: a constant string, set to an empty string by default. Returns the string.

alphaTokens(s)

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

Example:

```
SELECT alphaTokens('abca1abc')

┌─alphaTokens('abca1abc')─┐
│ ['abca','abc']          │
└─────────────────────────┘
```

Bit functions

Bit functions work for any pair of types from UInt8, UInt16, UInt32, UInt64, Int8, Int16, Int32, Int64, Float32, or Float64.

The result type is an integer with bits equal to the maximum bits of its arguments. If at least one of the arguments is signed, the result is a signed number. If an argument is a floating-point number, it is cast to Int64.

bitAnd(a, b)

bitOr(a, b)

bitXor(a, b)

bitNot(a)

bitShiftLeft(a, b)

bitShiftRight(a, b)

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 want to get the same result as output by the md5sum utility, use 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 the link: https://131002.net/siphash/ [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 bytes.

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. `URLHash(s, N)` – Calculates a hash from a string up to the N level in the URL hierarchy, without one of the trailing symbols `/`, `?` or `#` at the end, if present. Levels are the same as in URLHierarchy. This function is specific to Yandex.Metrica.

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.

Encoding functions

hex

Accepts arguments of types: `String`, `unsigned integer`, `Date`, or `DateTime`. Returns a string containing the argument's hexadecimal representation. Uses uppercase letters `A-F`. Does not use `0x` prefixes or `h` suffixes. For strings, all bytes are simply encoded as two hexadecimal numbers. Numbers are converted to big endian ("human readable") format. For numbers, older zeros are trimmed, but only by entire bytes. For example, `hex (1) = '01'`. `Date` is encoded as the number of days since the beginning of the Unix epoch. `DateTime` is encoded as the number of seconds since the beginning of the Unix epoch.

unhex(str)

Accepts a string containing any number of hexadecimal digits, and returns a string containing the corresponding bytes. Supports both uppercase and lowercase letters A-F. The number of hexadecimal digits does not have to be even. If it is odd, the last digit is interpreted as the younger half of the 00-0F byte. If the argument string contains anything other than hexadecimal digits, some implementation-defined result is returned (an exception isn't thrown). If you want to convert the result to a number, you can use the 'reverse' and 'reinterpretAsType' functions.

UUIDStringToNum(str)

Accepts a string containing 36 characters in the format `123e4567-e89b-12d3-a456-426655440000`, and returns it as a set of bytes in a FixedString(16).

UUIDNumToString(str)

Accepts a FixedString(16) value. Returns a string containing 36 characters in text format.

bitmaskToList(num)

Accepts an integer. Returns a string containing the list of powers of two that total the source number when summed. They are comma-separated without spaces in text format, in ascending order.

bitmaskToArray(num)

Accepts an integer. Returns an array of UInt64 numbers containing the list of powers of two that total the source number when summed. Numbers in the array are in ascending order.

Functions for working with URLs

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

Functions that extract part of a URL

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

protocol

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

domain

Gets the domain.

domainWithoutWWW

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

topLevelDomain

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

firstSignificantSubdomain

Returns the "first significant subdomain". This is a non-standard concept specific to Yandex.Metrica. The first significant subdomain is a second-level domain if it is 'com', 'net', 'org', or 'co'. Otherwise, it is a third-level domain. For example, firstSignificantSubdomain ('https://news.yandex.ru/ [https://news.yandex.ru/]') = 'yandex ', firstSignificantSubdomain ('https://news.yandex.com.tr/ [https://news.yandex.com.tr/]') = 'yandex '. The list of "insignificant" second-level domains and other implementation details may change in the future.

cutToFirstSignificantSubdomain

Returns the part of the domain that includes top-level subdomains up to the "first significant subdomain" (see the explanation above).

For example, `cutToFirstSignificantSubdomain('https://news.yandex.com.tr/') = 'yandex.com.tr'`.

path

Returns the path. Example: `/top/news.html` The path does not include the query string.

pathFull

The same as above, but including query string and fragment. Example: /top/news.html?page=2#comments

queryString

Returns the query string. Example: page=1&lr=213. query-string does not include the initial question mark, as well as # and everything after #.

fragment

Returns the fragment identifier. fragment does not include the initial hash symbol.

queryStringAndFragment

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

extractURLParameter(URL, name)

Returns the value of the 'name' parameter in the URL, if present. Otherwise, an empty string. If there are many parameters with this name, it returns the first occurrence. This function works under the assumption that the parameter name is encoded in the URL exactly the same way as in the passed argument.

extractURLParameters(URL)

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

extractURLParameterNames(URL)

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

URLHierarchy(URL)

Returns an array containing the URL, truncated at the end by the symbols /,? in the path and query-string. Consecutive separator characters are counted as one. The cut is made in the position after all the consecutive separator characters. Example:

URLPathHierarchy(URL)

The same as above, but without the protocol and host in the result. The / element (root) is not included. Example: the function is used to implement tree reports the URL in Yandex. Metric.

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

decodeURLComponent(URL)

Returns the decoded URL. Example:

```
SELECT decodeURLComponent('http://127.0.0.1:8123/?query=SELECT%201%3B') AS DecodedURL;
```

```
┌─DecodedURL─────────────────────────────┐
│ http://127.0.0.1:8123/?query=SELECT 1; │
└────────────────────────────────────────┘
```

Functions that remove part of a URL.

If the URL doesn't have anything similar, the URL remains unchanged.

cutWWW

Removes no more than one 'www.' from the beginning of the URL's domain, if present.

cutQueryString

Removes query string. The question mark is also removed.

cutFragment

Removes the fragment identifier. The number sign is also removed.

cutQueryStringAndFragment

Removes the query string and fragment identifier. The question mark and number sign are also removed.

cutURLParameter(URL, name)

Removes the 'name' URL parameter, if present. This function works under the assumption that the parameter name is encoded in the URL exactly the same way as in the passed argument.

Functions for working with IP addresses

IPv4NumToString(num)

Takes a UInt32 number. Interprets it as an IPv4 address in big endian. Returns a string containing the corresponding IPv4 address in the format A.B.C.d (dot-separated numbers in decimal form).

IPv4StringToNum(s)

The reverse function of IPv4NumToString. If the IPv4 address has an invalid format, it returns 0.

IPv4NumToStringClassC(num)

Similar to IPv4NumToString, but using xxx instead of the last octet.

Example:

```
SELECT
    IPv4NumToStringClassC(ClientIP) AS k,
    count() AS c
FROM test.hits
GROUP BY k
ORDER BY c DESC
LIMIT 10
```

```
┌─k──────────────┬─────c─┐
│ 83.149.9.xxx   │ 26238 │
│ 217.118.81.xxx │ 26074 │
│ 213.87.129.xxx │ 25481 │
│ 83.149.8.xxx   │ 24984 │
│ 217.118.83.xxx │ 22797 │
│ 78.25.120.xxx  │ 22354 │
│ 213.87.131.xxx │ 21285 │
│ 78.25.121.xxx  │ 20887 │
│ 188.162.65.xxx │ 19694 │
│ 83.149.48.xxx  │ 17406 │
└────────────────┴───────┘
```

Since using 'xxx' is highly unusual, this may be changed in the future. We recommend that you don't rely on the exact format of this fragment.

IPv6NumToString(x)

Accepts a FixedString(16) value containing the IPv6 address in binary format. Returns a string containing this address in text format. IPv6-mapped IPv4 addresses are output in the format ::ffff:111.222.33.44. Examples:

```
SELECT IPv6NumToString(toFixedString(unhex('2A0206B8000000000000000000000011'), 16)) AS addr
```

```
┌─addr─────────┐
│ 2a02:6b8::11 │
└──────────────┘
```

```
SELECT
    IPv6NumToString(ClientIP6 AS k),
    count() AS c
FROM hits_all
WHERE EventDate = today() AND substring(ClientIP6, 1, 12) != unhex('00000000000000000000FFFF')
GROUP BY k
ORDER BY c DESC
LIMIT 10
```

```
┌─IPv6NumToString(ClientIP6)──────────────┬─────c─┐
│ 2a02:2168:aaa:bbbb::2                   │ 24695 │
│ 2a02:2698:abcd:abcd:abcd: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 transmitted by users as session parameters. There are some special functions for working with this JSON. (Although in most of the cases, the JSONs are additionally pre-processed, and the resulting values are put in

separate columns in their processed format.) All these functions are based on strong assumptions about what the JSON can be, but they try to do as little as possible to get the job done.

The following assumptions are made:

1. The field name (function argument) must be a constant.
2. The field name is somehow canonically encoded in JSON. For example: `visitParamHas('{"abc":"def"}', 'abc') = 1`, but `visitParamHas('{"\\u0061\\u0062\\u0063":"def"}', 'abc') = 0`
3. Fields are searched for on any nesting level, indiscriminately. If there are multiple matching fields, the first occurrence is used.
4. The JSON doesn't have space characters outside of string literals.

visitParamHas(params, name)

Checks whether there is a field with the 'name' name.

visitParamExtractUInt(params, name)

Parses UInt64 from the value of the field named 'name'. If this is a string field, it tries to parse a number from the beginning of the string. If the field doesn't exist, or it exists but doesn't contain a number, it returns 0.

visitParamExtractInt(params, name)

The same as for Int64.

visitParamExtractFloat(params, name)

The same as for Float64.

visitParamExtractBool(params, name)

Parses a true/false value. The result is UInt8.

visitParamExtractRaw(params, name)

Returns the value of a field, including separators.

Examples:

```
visitParamExtractRaw('{"abc":"\\n\\u0000"}', 'abc') = '"\\n\\u0000"'
visitParamExtractRaw('{"abc":{"def":[1,2,3]}}', 'abc') = '{"def":[1,2,3]}'
```

visitParamExtractString(params, name)

Parses the string in double quotes. The value is unescaped. If unescaping failed, it returns an empty string.

Examples:

```
visitParamExtractString('{"abc":"\\n\\u0000"}', 'abc') = '\n\0'
visitParamExtractString('{"abc":"\\u263a"}', 'abc') = '☺'
visitParamExtractString('{"abc":"\\u263"}', 'abc') = ''
visitParamExtractString('{"abc":"hello}', 'abc') = ''
```

There is currently no support for code points in the format `\uXXXX\uYYYY` that are not from the basic multilingual plane

(they are converted to CESU-8 instead of UTF-8).

Higher-order functions

-> operator, lambda(params, expr) function

Allows describing a lambda function for passing to a higher-order function. The left side of the arrow has a formal parameter, which is any ID, or multiple formal parameters – any IDs in a tuple. The right side of the arrow has an expression that can use these formal parameters, as well as any table columns.

Examples: `x -> 2 * x, str -> str != Referer.`

Higher-order functions can only accept lambda functions as their functional argument.

A lambda function that accepts multiple arguments can be passed to a higher-order function. In this case, the higher-order function is passed several arrays of identical length that these arguments will correspond to.

For all functions other than 'arrayMap' and 'arrayFilter', the first argument (the lambda function) can be omitted. In this case, identical mapping is assumed.

arrayMap(func, arr1, ...)

Returns an array obtained from the original application of the 'func' function to each element in the 'arr' array.

arrayFilter(func, arr1, ...)

Returns an array containing only the elements in 'arr1' for which 'func' returns something other than 0.

Examples:

```
SELECT arrayFilter(x -> x LIKE '%World%', ['Hello', 'abc World']) AS res
```

```
┌─res───────────┐
│ ['abc World'] │
└───────────────┘
```

```
SELECT
    arrayFilter(
        (i, x) -> x LIKE '%World%',
        arrayEnumerate(arr),
        ['Hello', 'abc World'] AS arr)
    AS res
```

```
┌─res─┐
│ [2] │
└─────┘
```

arrayCount([func,] arr1, ...)

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

arrayExists([func,] arr1, ...)

Returns 1 if there is at least one element in 'arr' for which 'func' returns something other than 0. Otherwise, it returns 0.

arrayAll([func,] arr1, ...)

Returns 1 if 'func' returns something other than 0 for all the elements in 'arr'. Otherwise, it returns 0.

arraySum([func,] arr1, ...)

Returns the sum of the 'func' values. If the function is omitted, it just returns the sum of the array elements.

arrayFirst(func, arr1, ...)

Returns the first element in the 'arr1' array for which 'func' returns something other than 0.

arrayFirstIndex(func, arr1, ...)

Returns the index of the first element in the 'arr1' array for which 'func' returns something other than 0.

arrayCumSum([func,] arr1, ...)

Returns an array of partial sums of elements in the source array (a running sum). If the `func` function is specified, then the values of the array elements are converted by this function before summing.

Example:

```
SELECT arrayCumSum([1, 1, 1, 1]) AS res
```

```
┌─res──────────┐
│ [1, 2, 3, 4] │
└──────────────┘
```

arraySort([func,] arr1, ...)

Returns an array as result of sorting the elements of `arr1` in ascending order. If the `func` function is specified, sorting order is determined by the result of the function `func` applied to the elements of array (arrays)

The Schwartzian transform [https://en.wikipedia.org/wiki/Schwartzian_transform] is used to impove sorting efficiency.

Example:

```
SELECT arraySort((x, y) -> y, ['hello', 'world'], [2, 1]);
```

```
┌─res────────────────┐
│ ['world', 'hello'] │
└────────────────────┘
```

arrayReverseSort([func,] arr1, ...)

Returns an array as result of sorting the elements of `arr1` in descending order. If the `func` function is specified, sorting order is determined by the result of the function `func` applied to the elements of array (arrays)

[]

Functions for working with external dictionaries

For information on connecting and configuring external dictionaries, see "External dictionaries [#dicts-external_dicts]".

dictGetUInt8, dictGetUInt16, dictGetUInt32, dictGetUInt64

dictGetInt8, dictGetInt16, dictGetInt32, dictGetInt64

dictGetFloat32, dictGetFloat64

dictGetDate, dictGetDateTime

dictGetUUID

dictGetString

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

- Get the value of the attr_name attribute from the dict_name dictionary using the 'id' key. `dict_name` and `attr_name` are constant strings. `id` must be UInt64. If there is no `id` key in the dictionary, it returns the default value specified in the dictionary description.

dictGetTOrDefault

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

The same as the `dictGetT` functions, but the default value is taken from the function's last argument.

dictIsIn

`dictIsIn ('dict_name', child_id, ancestor_id)`

- For the 'dict_name' hierarchical dictionary, finds out whether the 'child_id' key is located inside 'ancestor_id' (or matches 'ancestor_id'). Returns UInt8.

dictGetHierarchy

`dictGetHierarchy('dict_name', id)`

- For the 'dict_name' hierarchical dictionary, returns an array of dictionary keys starting from 'id' and continuing along the chain of parent elements. Returns Array(UInt64).

dictHas

`dictHas('dict_name', id)`

- Check whether the dictionary has the key. Returns a UInt8 value equal to 0 if there is no key and 1 if there is a key.

Functions for working with Yandex.Metrica dictionaries

In order for the functions below to work, the server config must specify the paths and addresses for getting all the Yandex.Metrica dictionaries. The dictionaries are loaded at the first call of any of these functions. If the reference lists can't be loaded, an exception is thrown.

For information about creating reference lists, see the section "Dictionaries".

Multiple geobases

ClickHouse supports working with multiple alternative geobases (regional hierarchies) simultaneously, in order to support various perspectives on which countries certain regions belong to.

The 'clickhouse-server' config specifies the file with the regional hierarchy:: `<path_to_regions_hierarchy_file>/opt/geo/regions_hierarchy.txt</path_to_regions_hierarchy_file>`

Besides this file, it also searches for files nearby that have the _ symbol and any suffix appended to the name (before the file extension). For example, it will also find the file `/opt/geo/regions_hierarchy_ua.txt` , if present.

`ua` is called the dictionary key. For a dictionary without a suffix, the key is an empty string.

All the dictionaries are re-loaded in runtime (once every certain number of seconds, as defined in the builtin_dictionaries_reload_interval config parameter, or once an hour by default). However, the list of available dictionaries is defined one time, when the server starts.

All functions for working with regions have an optional argument at the end – the dictionary key. It is referred to as the geobase. Example:

```
regionToCountry(RegionID) – Uses the default dictionary: /opt/geo/regions_hierarchy.txt
regionToCountry(RegionID, '') – Uses the default dictionary: /opt/geo/regions_hierarchy.txt
regionToCountry(RegionID, 'ua') – Uses the dictionary for the 'ua' key: /opt/geo/regions_hierarchy_ua.txt
```

regionToCity(id[, geobase])

Accepts a UInt32 number – the region ID from the Yandex geobase. If this region is a city or part of a city, it returns the region ID for the appropriate city. Otherwise, returns 0.

regionToArea(id[, geobase])

Converts a region to an area (type 5 in the geobase). In every other way, this function is the same as 'regionToCity'.

```
SELECT DISTINCT regionToName(regionToArea(toUInt32(number), 'ua'))
FROM system.numbers
LIMIT 15
```

```
┌─regionToName(regionToArea(toUInt32(number), \'ua\'))─┐
│                                                      │
│ Moscow and Moscow region                             │
│ St. Petersburg and Leningrad region                  │
│ Belgorod region                                      │
│ Ivanovsk region                                      │
│ Kaluga region                                        │
│ Kostroma region                                      │
│ Kursk region                                         │
│ Lipetsk region                                       │
│ Orlov region                                         │
│ Ryazan region                                        │
│ Smolensk region                                      │
│ Tambov region                                        │
│ Tver region                                          │
│ Tula region                                          │
└──────────────────────────────────────────────────────┘
```

regionToDistrict(id[, geobase])

Converts a region to a federal district (type 4 in the geobase). In every other way, this function is the same as 'regionToCity'.

```
SELECT DISTINCT regionToName(regionToDistrict(toUInt32(number), 'ua'))
FROM system.numbers
LIMIT 15
```

```
┌─regionToName(regionToDistrict(toUInt32(number), \'ua\'))─┐
│                                                          │
│ Central federal district                                 │
│ Northwest federal district                               │
│ South federal district                                   │
│ North Caucases federal district                          │
│ Privolga federal district                                │
│ Ural federal district                                    │
│ Siberian federal district                                │
│ Far East federal district                                │
│ Scotland                                                 │
│ Faroe Islands                                            │
│ Flemish region                                           │
│ Brussels capital region                                  │
│ Wallonia                                                 │
│ Federation of Bosnia and Herzegovina                     │
└──────────────────────────────────────────────────────────┘
```

regionToCountry(id[, geobase])

Converts a region to a country. In every other way, this function is the same as 'regionToCity'. Example: `regionToCountry(toUInt32(213)) = 225` converts Moscow (213) to Russia (225).

regionToContinent(id[, geobase])

Converts a region to a continent. In every other way, this function is the same as 'regionToCity'. Example: `regionToContinent(toUInt32(213)) = 10001` converts Moscow (213) to Eurasia (10001).

regionToPopulation(id[, geobase])

Gets the population for a region. The population can be recorded in files with the geobase. See the section "External dictionaries". If the population is not recorded for the region, it returns 0. In the Yandex geobase, the population might be recorded for child regions, but not for parent regions.

regionIn(lhs, rhs[, geobase])

Checks whether a 'lhs' region belongs to a 'rhs' region. Returns a UInt8 number equal to 1 if it belongs, or 0 if it doesn't belong. The relationship is reflexive – any region also belongs to itself.

regionHierarchy(id[, geobase])

Accepts a UInt32 number – the region ID from the Yandex geobase. Returns an array of region IDs consisting of the passed region and all parents along the chain. Example: `regionHierarchy(toUInt32(213)) = [213,1,3,225,10001,10000]`.

regionToName(id[, lang])

Accepts a UInt32 number – the region ID from the Yandex geobase. A string with the name of the language can be passed as a second argument. Supported languages are: ru, en, ua, uk, by, kz, tr. If the second argument is omitted, the language 'ru' is used. If the language is not supported, an exception is thrown. Returns a string – the name of the region in the corresponding language. If the region with the specified ID doesn't exist, an empty string is returned.

`ua` and `uk` both mean Ukrainian.

Functions for implementing the IN operator

in, notIn, globalIn, globalNotIn

See the section "IN operators".

tuple(x, y, ...), operator (x, y, ...)

A function that allows grouping multiple columns. For columns with the types T1, T2, ..., it returns a Tuple(T1, T2, ...) type tuple containing these columns. There is no cost to execute the function. Tuples are normally used as intermediate values for an argument of IN operators, or for creating a list of formal parameters of lambda functions. Tuples can't be written to a table.

tupleElement(tuple, n), operator x.N

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

[]

arrayJoin function

This is a very unusual function.

Normal functions don't change a set of rows, but just change the values in each row (map). Aggregate functions compress a set of rows (fold or reduce). The 'arrayJoin' function takes each row and generates a set of rows (unfold).

This function takes an array as an argument, and propagates the source row to multiple rows for the number of elements in the array. All the values in columns are simply copied, except the values in the column where this function is applied; it is replaced with the corresponding array value.

A query can use multiple `arrayJoin` functions. In this case, the transformation is performed multiple times.

Note the ARRAY JOIN syntax in the SELECT query, which provides broader possibilities.

Example:

```
SELECT arrayJoin([1, 2, 3] AS src) AS dst, 'Hello', src
```

```
┌─dst─┬─\'Hello\'─┬─src─────┐
│   1 │ Hello     │ [1,2,3] │
│   2 │ Hello     │ [1,2,3] │
│   3 │ Hello     │ [1,2,3] │
└─────┴───────────┴─────────┘
```

Functions for working with geographical coordinates

greatCircleDistance

Calculate the distance between two points on the Earth's surface using the great-circle formula [https://en.wikipedia.org/wiki/Great-circle_distance].

```
greatCircleDistance(lon1Deg, lat1Deg, lon2Deg, lat2Deg)
```

Input parameters

- `lon1Deg` — Longitude of the first point in degrees. Range: `[-180°, 180°]`.
- `lat1Deg` — Latitude of the first point in degrees. Range: `[-90°, 90°]`.
- `lon2Deg` — Longitude of the second point in degrees. Range: `[-180°, 180°]`.
- `lat2Deg` — Latitude of the second point in degrees. Range: `[-90°, 90°]`.

Positive values correspond to North latitude and East longitude, and negative values correspond to South latitude and

West longitude.

Returned value

The distance between two points on the Earth's surface, in meters.

Generates an exception when the input parameter values fall outside of the range.

Example

```
SELECT greatCircleDistance(55.755831, 37.617673, -55.755831, -37.617673)
```

```
┌─greatCircleDistance(55.755831, 37.617673, -55.755831, -37.617673)─┐
│                                                 14132374.194975413 │
└───────────────────────────────────────────────────────────────────┘
```

pointInEllipses

Checks whether the point belongs to at least one of the ellipses.

```
pointInEllipses(x, y, x₀, y₀, a₀, b₀,...,x?, y?, a?, b?)
```

Input parameters

- `x, y` — Coordinates of a point on the plane.
- `xᵢ, yᵢ` — Coordinates of the center of the `i` -th ellipsis.
- `aᵢ, bᵢ` — Axes of the `i` -th ellipsis in meters.

The input parameters must be `2+4⋅n` , where `n` is the number of ellipses.

Returned values

`1` if the point is inside at least one of the ellipses; `0` if it is not.

Example

```
SELECT pointInEllipses(55.755831, 37.617673, 55.755831, 37.617673, 1.0, 2.0)
```

```
┌─pointInEllipses(55.755831, 37.617673, 55.755831, 37.617673, 1., 2.)─┐
│                                                                   1 │
└─────────────────────────────────────────────────────────────────────┘
```

pointInPolygon

Checks whether the point belongs to the polygon on the plane.

```
pointInPolygon((x, y), [(a, b), (c, d) ...], ...)
```

Input values

- `(x, y)` — Coordinates of a point on the plane. Data type — Tuple [#data_type-tuple] — A tuple of two numbers.
- `[(a, b), (c, d) ...]` — Polygon vertices. Data type — Array [#data_type-array]. Each vertex is represented by a pair of coordinates `(a, b)` . Vertices should be specified in a clockwise or counterclockwise order. The minimum number of vertices is 3. The polygon must be constant.
- The function also supports polygons with holes (cut out sections). In this case, add polygons that define the cut out sections using additional arguments of the function. The function does not support non-simply-connected polygons.

Returned values

`1` if the point is inside the polygon, `0` if it is not. If the point is on the polygon boundary, the function may return either 0 or 1.

Example

```
SELECT pointInPolygon((3., 3.), [(6, 0), (8, 4), (5, 8), (0, 2)]) AS res
```

```
┌─res─┐
│   1 │
└─────┘
```

Functions for working with Nullable aggregates

isNull

Checks whether the argument is NULL [#null-literal].

```
isNull(x)
```

Parameters

- `x` — A value with a non-compound data type.

Returned value

- `1` if `x` is `NULL`.
- `0` if `x` is not `NULL`.

Example

Input table

```
┌─x─┬────y─┐
│ 1 │ NULL │
│ 2 │    3 │
└───┴──────┘
```

Query

```
:) SELECT x FROM t_null WHERE isNull(y)

SELECT x
FROM t_null
WHERE isNull(y)

┌─x─┐
│ 1 │
└───┘

1 rows in set. Elapsed: 0.010 sec.
```

isNotNull

Checks whether the argument is NULL [#null-literal].

```
isNotNull(x)
```

Parameters:

- `x` — A value with a non-compound data type.

Returned value

- `0` if `x` is `NULL`.
- `1` if `x` is not `NULL`.

Example

Input table

```
┌─x─┬────y─┐
│ 1 │ NULL │
│ 2 │    3 │
└───┴──────┘
```

Query

```
:) SELECT x FROM t_null WHERE isNotNull(y)

SELECT x
FROM t_null
WHERE isNotNull(y)

┌─x─┐
│ 2 │
└───┘

1 rows in set. Elapsed: 0.010 sec.
```

coalesce

Checks from left to right whether `NULL` arguments were passed and returns the first non-`NULL` argument.

```
coalesce(x,...)
```

Parameters:

- Any number of parameters of a non-compound type. All parameters must be compatible by data type.

Returned values

- The first non-`NULL` argument.
- `NULL`, if all arguments are `NULL`.

Example

Consider a list of contacts that may specify multiple ways to contact a customer.

```
┌─name─────┬─mail─┬─phone─────┬──icq─┐
│ client 1 │ NULL │ 123-45-67 │  123 │
│ client 2 │ NULL │ NULL      │ NULL │
└──────────┴──────┴───────────┴──────┘
```

The `mail` and `phone` fields are of type String, but the `icq` field is `UInt32`, so it needs to be converted to `String`.

Get the first available contact method for the customer from the contact list:

```
:) SELECT coalesce(mail, phone, CAST(icq,'Nullable(String)')) FROM aBook

SELECT coalesce(mail, phone, CAST(icq, 'Nullable(String)'))
FROM aBook

┌─name─────┬─coalesce(mail, phone, CAST(icq, 'Nullable(String)'))─┐
│ client 1 │ 123-45-67                                            │
│ client 2 │ NULL                                                 │
└──────────┴──────────────────────────────────────────────────────┘

2 rows in set. Elapsed: 0.006 sec.
```

ifNull

Returns an alternative value if the main argument is `NULL`.

```
ifNull(x,alt)
```

Parameters:

- `x` — The value to check for `NULL`.
- `alt` — The value that the function returns if `x` is `NULL`.

Returned values

- The value `x`, if `x` is not `NULL`.
- The value `alt`, if `x` is `NULL`.

Example

```
SELECT ifNull('a', 'b')

┌─ifNull('a', 'b')─┐
│ a                │
└──────────────────┘
```

```
SELECT ifNull(NULL, 'b')

┌─ifNull(NULL, 'b')─┐
│ b                 │
└───────────────────┘
```

nullIf

Returns `NULL` if the arguments are equal.

```
nullIf(x, y)
```

Parameters:

`x`, `y` — Values for comparison. They must be compatible types, or ClickHouse will generate an exception.

Returned values

- `NULL`, if the arguments are equal.
- The `x` value, if the arguments are not equal.

Example

```
SELECT nullIf(1, 1)

┌─nullIf(1, 1)─┐
│         NULL │
└──────────────┘
```

```
SELECT nullIf(1, 2)

┌─nullIf(1, 2)─┐
│            1 │
└──────────────┘
```

assumeNotNull

Results in a value of type Nullable [#data_type-nullable] for a non- `Nullable` , if the value is not `NULL` .

```
assumeNotNull(x)
```

Parameters:

- `x` — The original value.

Returned values

- The original value from the non-`Nullable` type, if it is not `NULL` .
- The default value for the non-`Nullable` type if the original value was `NULL` .

Example

Consider the `t_null` table.

```
SHOW CREATE TABLE t_null

┌─statement─────────────────────────────────────────────────────────────────┐
│ CREATE TABLE default.t_null ( x Int8,  y Nullable(Int8)) ENGINE = TinyLog │
└───────────────────────────────────────────────────────────────────────────┘
```

```
┌─x─┬────y─┐
│ 1 │ NULL │
│ 2 │    3 │
└───┴──────┘
```

Apply the `resumenotnull` function to the `y` column.

```
SELECT assumeNotNull(y) FROM t_null

┌─assumeNotNull(y)─┐
│                0 │
│                3 │
└──────────────────┘
```

```
SELECT toTypeName(assumeNotNull(y)) FROM t_null

┌─toTypeName(assumeNotNull(y))─┐
│ Int8                         │
│ Int8                         │
└──────────────────────────────┘
```

toNullable

Converts the argument type to `Nullable`.

```
toNullable(x)
```

Parameters:

- `x` — The value of any non-compound type.

Returned value

- The input value with a non-`Nullable` type.

Example

```
SELECT toTypeName(10)

┌─toTypeName(10)─┐
│ UInt8          │
└────────────────┘

SELECT toTypeName(toNullable(10))

┌─toTypeName(toNullable(10))─┐
│ Nullable(UInt8)            │
└────────────────────────────┘
```

Other functions

hostName()

Returns a string with the name of the host that this function was performed on. For distributed processing, this is the name of the remote server host, if the function is performed on a remote server.

visibleWidth(x)

Calculates the approximate width when outputting values to the console in text format (tab-separated). This function is used by the system for implementing Pretty formats.

`NULL` is represented as a string corresponding to `NULL` in `Pretty` formats.

```
SELECT visibleWidth(NULL)

┌─visibleWidth(NULL)─┐
│                  4 │
└────────────────────┘
```

toTypeName(x)

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

If `NULL` is passed to the function as input, then it returns the `Nullable(Nothing)` type, which corresponds to an internal `NULL` representation in ClickHouse.

blockSize()

Gets the size of the block. In ClickHouse, queries are always run on blocks (sets of column parts). This function allows getting the size of the block that you called it for.

materialize(x)

Turns a constant into a full column containing just one value. In ClickHouse, full columns and constants are represented differently in memory. Functions work differently for constant arguments and normal arguments (different code is executed), although the result is almost always the same. This function is for debugging this behavior.

ignore(...)

Accepts any arguments, including `NULL` . Always returns 0. However, the argument is still evaluated. This can be used for benchmarks.

sleep(seconds)

Sleeps 'seconds' seconds on each data block. You can specify an integer or a floating-point number.

currentDatabase()

Returns the name of the current database. You can use this function in table engine parameters in a CREATE TABLE query where you need to specify the database.

isFinite(x)

Accepts Float32 and Float64 and returns UInt8 equal to 1 if the argument is not infinite and not a NaN, otherwise 0.

isInfinite(x)

Accepts Float32 and Float64 and returns UInt8 equal to 1 if the argument is infinite, otherwise 0. Note that 0 is returned for a NaN.

isNaN(x)

Accepts Float32 and Float64 and returns UInt8 equal to 1 if the argument is a NaN, otherwise 0.

hasColumnInTable(['hostname'[, 'username'[, 'password']],] 'database', 'table', 'column')

Accepts constant strings: database name, table name, and column name. Returns a UInt8 constant expression equal to 1 if there is a column, otherwise 0. If the hostname parameter is set, the test will run on a remote server. The function throws an exception if the table does not exist. For elements in a nested data structure, the function checks for the existence of a column. For the nested data structure itself, the function returns 0.

bar

Allows building a unicode-art diagram.

`bar(x, min, max, width)` draws a band with a width proportional to `(x - min)` and equal to `width` characters when `x = max` .

Parameters:

- `x` — Size to display.
- `min, max` — Integer constants. The value must fit in `Int64` .
- `width` — Constant, positive integer, can be fractional.

The band is drawn with accuracy to one eighth of a symbol.

Example:

```
SELECT
    toHour(EventTime) AS h,
    count() AS c,
    bar(c, 0, 600000, 20) AS bar
FROM test.hits
GROUP BY h
ORDER BY h ASC
```

```
┌──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` – Which value to use if 'x' is not equal to any of the values in 'from'.

`array_from` and `array_to` – Arrays of the same size.

Types:

`transform(T, Array(T), Array(U), U) -> U`

`T` and `U` can be numeric, string, or Date or DateTime types. Where the same letter is indicated (T or U), for numeric types these might not be matching types, but types that have a common type. For example, the first argument can have the Int64 type, while the second has the Array(Uint16) type.

If the 'x' value is equal to one of the elements in the 'array_from' array, it returns the existing element (that is numbered the same) from the 'array_to' array. Otherwise, it returns 'default'. If there are multiple matching elements in 'array_from', it returns one of the matches.

Example:

```
SELECT
    transform(SearchEngineID, [2, 3], ['Yandex', 'Google'], 'Other') AS title,
    count() AS c
FROM test.hits
WHERE SearchEngineID != 0
GROUP BY title
ORDER BY c DESC
```

```
┌─title─────┬──────c─┐
│ Yandex    │ 498635 │
│ Google    │ 229872 │
│ Other     │ 104472 │
└───────────┴────────┘
```

1. `transform(x, array_from, array_to)`

Differs from the first variation in that the 'default' argument is omitted. If the 'x' value is equal to one of the elements in the 'array_from' array, it returns the matching element (that is numbered the same) from the 'array_to' array. Otherwise, it returns 'x'.

Types:

`transform(T, Array(T), Array(T)) -> T`

Example:

```
SELECT
    transform(domain(Referer), ['yandex.ru', 'google.ru', 'vk.com'], ['www.yandex', 'example.com']) AS s,
    count() AS c
FROM test.hits
GROUP BY domain(Referer)
ORDER BY count() DESC
LIMIT 10
```

```
┌─s──────────────┬───────c─┐
│                │ 2906259 │
│ www.yandex     │  867767 │
│ ███████.ru     │  313599 │
│ mail.yandex.ru │  107147 │
│ ██████.ru      │  100355 │
│ █████████.ru   │   65040 │
│ news.yandex.ru │   64515 │
│ ██████.net     │   59141 │
│ example.com    │   57316 │
└────────────────┴─────────┘
```

formatReadableSize(x)

Accepts the size (number of bytes). Returns a rounded size with a suffix (KiB, MiB, etc.) as a string.

Example:

```
SELECT
    arrayJoin([1, 1024, 1024*1024, 192851925]) AS filesize_bytes,
    formatReadableSize(filesize_bytes) AS filesize
```

```
┌─filesize_bytes─┬─filesize───┐
│              1 │ 1.00 B     │
│           1024 │ 1.00 KiB   │
│        1048576 │ 1.00 MiB   │
│      192851925 │ 183.92 MiB │
└────────────────┴────────────┘
```

least(a, b)

Returns the smallest value from a and b.

greatest(a, b)

Returns the largest value of a and b.

uptime()

Returns the server's uptime in seconds.

version()

Returns the version of the server as a string.

rowNumberInAllBlocks()

Returns the ordinal number of the row in the data block. This function only considers the affected data blocks.

runningDifference(x)

Calculates the difference between successive row values in the data block. Returns 0 for the first row and the difference from the previous row for each subsequent row.

The result of the function depends on the affected data blocks and the order of data in the block. If you make a subquery with ORDER BY and call the function from outside the subquery, you can get the expected result.

Example:

```
SELECT
    EventID,
    EventTime,
    runningDifference(EventTime) AS delta
FROM
(
    SELECT
        EventID,
        EventTime
    FROM events
    WHERE EventDate = '2016-11-24'
    ORDER BY EventTime ASC
    LIMIT 5
)
```

```
┌─EventID─┬───────────EventTime─┬─delta─┐
│    1106 │ 2016-11-24 00:00:04 │     0 │
│    1107 │ 2016-11-24 00:00:05 │     1 │
│    1108 │ 2016-11-24 00:00:05 │     0 │
│    1109 │ 2016-11-24 00:00:09 │     4 │
│    1110 │ 2016-11-24 00:00:10 │     1 │
└─────────┴─────────────────────┴───────┘
```

MACNumToString(num)

Accepts a UInt64 number. Interprets it as a MAC address in big endian. Returns a string containing the corresponding MAC address in the format AA:BB:CC:DD:EE:FF (colon-separated numbers in hexadecimal form).

MACStringToNum(s)

The inverse function of MACNumToString. If the MAC address has an invalid format, it returns 0.

MACStringToOUI(s)

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

getSizeOfEnumType

Returns the number of fields in Enum [#data_type-enum].

```
getSizeOfEnumType(value)
```

Parameters:

- `value` — Value of type `Enum`.

Returned values

- The number of fields with `Enum` input values.
- An exception is thrown if the type is not `Enum`.

Example

```
SELECT getSizeOfEnumType( CAST('a' AS Enum8('a' = 1, 'b' = 2) ) ) AS x

┌─x─┐
│ 2 │
└───┘
```

toColumnTypeName

Returns the name of the class that represents the data type of the column in RAM.

```
toColumnTypeName(value)
```

Parameters:

- `value` — Any type of value.

Returned values

- A string with the name of the class that is used for representing the `value` data type in RAM.

Example of the difference between `toTypeName ' and ' toColumnTypeName`

```
:) select toTypeName(cast('2018-01-01 01:02:03' AS DateTime))

SELECT toTypeName(CAST('2018-01-01 01:02:03', 'DateTime'))

┌─toTypeName(CAST('2018-01-01 01:02:03', 'DateTime'))─┐
│ DateTime                                            │
└─────────────────────────────────────────────────────┘

1 rows in set. Elapsed: 0.008 sec.

:) select toColumnTypeName(cast('2018-01-01 01:02:03' AS DateTime))

SELECT toColumnTypeName(CAST('2018-01-01 01:02:03', 'DateTime'))

┌─toColumnTypeName(CAST('2018-01-01 01:02:03', 'DateTime'))─┐
│ Const(UInt32)                                             │
└───────────────────────────────────────────────────────────┘
```

The example shows that the `DateTime` data type is stored in memory as `Const(UInt32)` .

dumpColumnStructure

Outputs a detailed description of data structures in RAM

```
dumpColumnStructure(value)
```

Parameters:

- `value` — Any type of value.

Returned values

- A string describing the structure that is used for representing the `value` data type in RAM.

Example

```
SELECT dumpColumnStructure(CAST('2018-01-01 01:02:03', 'DateTime'))

┌─dumpColumnStructure(CAST('2018-01-01 01:02:03', 'DateTime'))─┐
│ DateTime, Const(size = 1, UInt32(size = 1))                  │
└──────────────────────────────────────────────────────────────┘
```

defaultValueOfArgumentType

Outputs the default value for the data type.

Does not include default values for custom columns set by the user.

```
defaultValueOfArgumentType(expression)
```

Parameters:

- `expression` — Arbitrary type of value or an expression that results in a value of an arbitrary type.

Returned values

- `0` for numbers.
- Empty string for strings.
- `NULL` for Nullable [#data_type-nullable].

Example

```
:) SELECT defaultValueOfArgumentType( CAST(1 AS Int8) )

SELECT defaultValueOfArgumentType(CAST(1, 'Int8'))

┌─defaultValueOfArgumentType(CAST(1, 'Int8'))─┐
│                                           0 │
└─────────────────────────────────────────────┘

1 rows in set. Elapsed: 0.002 sec.

:) SELECT defaultValueOfArgumentType( CAST(1 AS Nullable(Int8) ) )

SELECT defaultValueOfArgumentType(CAST(1, 'Nullable(Int8)'))

┌─defaultValueOfArgumentType(CAST(1, 'Nullable(Int8)'))─┐
│                                                  NULL │
└───────────────────────────────────────────────────────┘

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

indexHint

Outputs data in the range selected by the index without filtering by the expression specified as an argument.

The expression passed to the function is not calculated, but ClickHouse applies the index to this expression in the same way as if the expression was in the query without `indexHint`.

Returned value

- a.

Example

Here is a table with the test data for ontime [#example_datasets-ontime].

```
SELECT count() FROM ontime

┌─count()─┐
│ 4276457 │
└─────────┘
```

The table has indexes for the fields `(FlightDate, (Year, FlightDate))`.

Create a selection by date like this:

```
:) SELECT FlightDate AS k, count() FROM ontime GROUP BY k ORDER BY k

SELECT
    FlightDate AS k,
    count()
FROM ontime
GROUP BY k
ORDER BY k ASC

┌──────────k─┬─count()─┐
│ 2017-01-01 │   13970 │
│ 2017-01-02 │   15882 │
........................
│ 2017-09-28 │   16411 │
│ 2017-09-29 │   16384 │
│ 2017-09-30 │   12520 │
└────────────┴─────────┘

273 rows in set. Elapsed: 0.072 sec. Processed 4.28 million rows, 8.55 MB (59.00 million rows/s., 118.01
MB/s.)
```

In this selection, the index is not used and ClickHouse processed the entire table (`Processed 4.28 million rows`). To apply the index, select a specific date and run the following query:

```
:) SELECT FlightDate AS k, count() FROM ontime WHERE k = '2017-09-15' GROUP BY k ORDER BY k

SELECT
    FlightDate AS k,
    count()
FROM ontime
WHERE k = '2017-09-15'
GROUP BY k
ORDER BY k ASC

┌──────────k─┬─count()─┐
│ 2017-09-15 │   16428 │
└────────────┴─────────┘

1 rows in set. Elapsed: 0.014 sec. Processed 32.74 thousand rows, 65.49 KB (2.31 million rows/s., 4.63 MB/s.)
```

The last line of output shows that by using the index, ClickHouse processed a significantly smaller number of rows (`Processed 32.74 thousand rows`).

Now pass the expression `k = '2017-09-15'` to the `indexHint` function:

```
:) SELECT FlightDate AS k, count() FROM ontime WHERE indexHint(k = '2017-09-15') GROUP BY k ORDER BY k

SELECT
    FlightDate AS k,
    count()
FROM ontime
WHERE indexHint(k = '2017-09-15')
GROUP BY k
ORDER BY k ASC

┌──────────k─┬─count()─┐
│ 2017-09-14 │    7071 │
│ 2017-09-15 │   16428 │
│ 2017-09-16 │    1077 │
│ 2017-09-30 │    8167 │
└────────────┴─────────┘

4 rows in set. Elapsed: 0.004 sec. Processed 32.74 thousand rows, 65.49 KB (8.97 million rows/s., 17.94 MB/s.)
```

The response to the request shows that ClickHouse applied the index in the same way as the previous time (`Processed 32.74 thousand rows`). However, the resulting set of rows shows that the expression `k = '2017-09-15'` was not used when generating the result.

Because the index is sparse in ClickHouse, "extra" data ends up in the response when reading a range (in this case, the adjacent dates). Use the `indexHint` function to see it.

replicate

Creates an array with a single value.

Used for internal implementation of arrayJoin [#functions_arrayjoin].

```
replicate(x, arr)
```

Parameters:

- `arr` — Original array. ClickHouse creates a new array of the same length as the original and fills it with the value `x` .
- `x` — The value that the resulting array will be filled with.

Output value

- An array filled with the value `x`.

Example

```
SELECT replicate(1, ['a', 'b', 'c'])

┌─replicate(1, ['a', 'b', 'c'])─┐
│ [1,1,1]                       │
└───────────────────────────────┘
```

[]

Aggregate functions

Aggregate functions work in the normal [http://www.sql-tutorial.com/sql-aggregate-functions-sql-tutorial] way as expected by database experts.

ClickHouse also supports:

- Parametric aggregate functions [#aggregate_functions_parametric], which accept other parameters in addition to columns.
- Combinators [#aggregate_functions_combinators], which change the behavior of aggregate functions.

NULL processing

During aggregation, all `NULL` s are skipped.

Examples:

Consider this table:

```
┌─x─┬────y─┐
│ 1 │    2 │
│ 2 │ NULL │
│ 3 │    2 │
│ 3 │    3 │
│ 3 │ NULL │
└───┴──────┘
```

Let's say you need to total the values in the `y` column:

```
:) SELECT sum(y) FROM t_null_big

SELECT sum(y)
FROM t_null_big

┌─sum(y)─┐
│      7 │
└────────┘

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

The `sum` function interprets `NULL` as `0`. In particular, this means that if the function receives input of a selection where all the values are `NULL`, then the result will be `0`, not `NULL`.

Now you can use the `groupArray` function to create an array from the `y` column:

```
:) SELECT groupArray(y) FROM t_null_big

SELECT groupArray(y)
FROM t_null_big

┌─groupArray(y)─┐
│ [2,2,3]       │
└───────────────┘

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

`groupArray` does not include `NULL` in the resulting array.

[]

Function reference

count()

Counts the number of rows. Accepts zero arguments and returns UInt64. The syntax `COUNT(DISTINCT x)` is not supported. The separate `uniq` aggregate function exists for this purpose.

A `SELECT count() FROM table` query is not optimized, because the number of entries in the table is not stored separately. It will select some small column from the table and count the number of values in it.

[]

any(x)

Selects the first encountered value. The query can be executed in any order and even in a different order each time, so the result of this function is indeterminate. To get a determinate result, you can use the 'min' or 'max' function instead of 'any'.

In some cases, you can rely on the order of execution. This applies to cases when SELECT comes from a subquery that uses ORDER BY.

When a `SELECT` query has the `GROUP BY` clause or at least one aggregate function, ClickHouse (in contrast to MySQL) requires that all expressions in the `SELECT`, `HAVING`, and `ORDER BY` clauses be calculated from keys or from aggregate functions. In other words, each column selected from the table must be used either in keys or inside aggregate functions. To get behavior like in MySQL, you can put the other columns in the `any` aggregate function.

anyHeavy(x)

Selects a frequently occurring value using the heavy hitters [http://www.cs.umd.edu/~samir/498/karp.pdf] algorithm. If there is a value that occurs more than in half the cases in each of the query's execution threads, this value is returned. Normally, the result is nondeterministic.

```
anyHeavy(column)
```

Arguments

- `column` – The column name.

Example

Take the OnTime [#example_datasets-ontime] data set and select any frequently occurring value in the `AirlineID` column.

```
SELECT anyHeavy(AirlineID) AS res
FROM ontime
```

```
┌───res─┐
│ 19690 │
└───────┘
```

anyLast(x)

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

groupBitAnd

Applies bitwise `AND` for series of numbers.

```
groupBitAnd(expr)
```

Parameters

`expr` – An expression that results in `UInt*` type.

Return value

Value of the `UInt*` type.

Example

Test data:

```
binary     decimal
00101100 = 44
00011100 = 28
00001101 = 13
01010101 = 85
```

Query:

```
SELECT groupBitAnd(num) FROM t
```

Where `num` is the column with the test data.

Result:

```
binary     decimal
00000100 = 4
```

groupBitOr

Applies bitwise `OR` for series of numbers.

```
groupBitOr(expr)
```

Parameters

`expr` – An expression that results in `UInt*` type.

Return value

Value of the `UInt*` type.

Example

Test data:

```
binary     decimal
00101100 = 44
00011100 = 28
00001101 = 13
01010101 = 85
```

Query:

```
SELECT groupBitOr(num) FROM t
```

Where `num` is the column with the test data.

Result:

```
binary     decimal
01111101 = 125
```

groupBitXor

Applies bitwise `XOR` for series of numbers.

```
groupBitXor(expr)
```

Parameters

`expr` – An expression that results in `UInt*` type.

Return value

Value of the `UInt*` type.

Example

Test data:

```
binary     decimal
00101100 = 44
00011100 = 28
00001101 = 13
01010101 = 85
```

Query:

```
SELECT groupBitXor(num) FROM t
```

Where `num` is the column with the test data.

Result:

```
binary     decimal
01101000 = 104
```

min(x)

Calculates the minimum.

max(x)

Calculates the maximum.

argMin(arg, val)

Calculates the 'arg' value for a minimal 'val' value. If there are several different values of 'arg' for minimal values of 'val', the first of these values encountered is output.

Example:

```
┌─user─────┬─salary─┐
│ director │   5000 │
│ manager  │   3000 │
│ worker   │   1000 │
└──────────┴────────┘

SELECT argMin(user, salary) FROM salary

┌─argMin(user, salary)─┐
│ worker               │
└──────────────────────┘
```

argMax(arg, val)

Calculates the 'arg' value for a maximum 'val' value. If there are several different values of 'arg' for maximum values of 'val', the first of these values encountered is output.

[]

sum(x)

Calculates the sum. Only works for numbers.

sumWithOverflow(x)

Computes the sum of the numbers, using the same data type for the result as for the input parameters. If the sum exceeds the maximum value for this data type, the function returns an error.

Only works for numbers.

[]

sumMap(key, value)

Totals the 'value' array according to the keys specified in the 'key' array. The number of elements in 'key' and 'value' must be the same for each row that is totaled. Returns a tuple of two arrays: keys in sorted order, and values summed for the corresponding keys.

Example:

```
CREATE TABLE sum_map(
    date Date,
    timeslot DateTime,
    statusMap Nested(
        status UInt16,
        requests UInt64
    )
) ENGINE = Log;
INSERT INTO sum_map VALUES
    ('2000-01-01', '2000-01-01 00:00:00', [1, 2, 3], [10, 10, 10]),
    ('2000-01-01', '2000-01-01 00:00:00', [3, 4, 5], [10, 10, 10]),
    ('2000-01-01', '2000-01-01 00:01:00', [4, 5, 6], [10, 10, 10]),
    ('2000-01-01', '2000-01-01 00:01:00', [6, 7, 8], [10, 10, 10]);
SELECT
    timeslot,
    sumMap(statusMap.status, statusMap.requests)
FROM sum_map
GROUP BY timeslot
```

```
┌────────────timeslot─┬─sumMap(statusMap.status, statusMap.requests)─┐
│ 2000-01-01 00:00:00 │ ([1,2,3,4,5],[10,10,20,10,10])               │
│ 2000-01-01 00:01:00 │ ([4,5,6,7,8],[10,10,20,10,10])               │
└─────────────────────┴──────────────────────────────────────────────┘
```

avg(x)

Calculates the average. Only works for numbers. The result is always Float64.

[]

uniq(x)

Calculates the approximate number of different values of the argument. Works for numbers, strings, dates, date-with-time, and for multiple arguments and tuple arguments.

Uses an adaptive sampling algorithm: for the calculation state, it uses a sample of element hash values with a size up to 65536. This algorithm is also very accurate for data sets with low cardinality (up to 65536) and very efficient on CPU (when computing not too many of these functions, using `uniq` is almost as fast as using other aggregate functions).

The result is determinate (it doesn't depend on the order of query processing).

This function provides excellent accuracy even for data sets with extremely high cardinality (over 10 billion elements). It is recommended for default use.

uniqCombined(HLL_precision)(x)

Calculates the approximate number of different values of the argument. Works for numbers, strings, dates, date-with-time, and for multiple arguments and tuple arguments.

A combination of three algorithms is used: array, hash table and HyperLogLog [https://en.wikipedia.org/wiki/HyperLogLog] with an error correction table. For small number of distinct elements, the array is used. When the set size becomes larger the hash table is used, while it is smaller than HyperLogLog data structure. For larger number of elements, the HyperLogLog is used, and it will occupy fixed amount of memory.

The parameter "HLL_precision" is the base-2 logarithm of the number of cells in HyperLogLog. You can omit the parameter (omit first parens). The default value is 17, that is effectively 96 KiB of space (2^17 cells of 6 bits each). The memory consumption is several times smaller than for the `uniq` function, and the accuracy is several times higher. Performance is slightly lower than for the `uniq` function, but sometimes it can be even higher than it, such as with distributed queries that transmit a large number of aggregation states over the network.

The result is deterministic (it doesn't depend on the order of query processing).

The `uniqCombined` function is a good default choice for calculating the number of different values, but keep in mind that the estimation error for large sets (200 million elements and more) will become larger than theoretical value due to poor choice of hash function.

uniqHLL12(x)

Uses the HyperLogLog [https://en.wikipedia.org/wiki/HyperLogLog] algorithm to approximate the number of different values of the argument. 212 5-bit cells are used. The size of the state is slightly more than 2.5 KB. The result is not very accurate (up to ~10% error) for small data sets (<10K elements). However, the result is fairly accurate for high-cardinality data sets (10K-100M), with a maximum error of ~1.6%. Starting from 100M, the estimation error increases, and the function will return very inaccurate results for data sets with extremely high cardinality (1B+ elements).

The result is determinate (it doesn't depend on the order of query processing).

We don't recommend using this function. In most cases, use the `uniq` or `uniqCombined` function.

uniqExact(x)

Calculates the number of different values of the argument, exactly. There is no reason to fear approximations. It's better to use the `uniq` function. Use the `uniqExact` function if you definitely need an exact result.

The `uniqExact` function uses more memory than the `uniq` function, because the size of the state has unbounded growth as the number of different values increases.

groupArray(x), groupArray(max_size)(x)

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

The second version (with the `max_size` parameter) limits the size of the resulting array to `max_size` elements. For example, `groupArray (1) (x)` is equivalent to `[any (x)]`.

In some cases, you can still rely on the order of execution. This applies to cases when `SELECT` comes from a subquery that uses `ORDER BY`.

[]

groupArrayInsertAt(x)

Inserts a value into the array in the specified position.

Accepts the value and position as input. If several values are inserted into the same position, any of them might end up in the resulting array (the first one will be used in the case of single-threaded execution). If no value is inserted into a position, the position is assigned the default value.

Optional parameters:

- The default value for substituting in empty positions.
- The length of the resulting array. This allows you to receive arrays of the same size for all the aggregate keys. When using this parameter, the default value must be specified.

groupUniqArray(x)

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.

In this function, as well as in all functions for calculating quantiles, the `level` parameter can be omitted. In this case, it is assumed to be equal to 0.5 (in other words, the function will calculate the median).

Works for numbers, dates, and dates with times. Returns: for numbers – `Float64` ; for dates – a date; for dates with times – a date with time.

Uses reservoir sampling [https://en.wikipedia.org/wiki/Reservoir_sampling] with a reservoir size up to 8192. If necessary, the result is output with linear approximation from the two neighboring values. This algorithm provides very low accuracy. See also: `quantileTiming` , `quantileTDigest` , `quantileExact` .

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

When using multiple `quantile` (and similar) functions with different levels in a query, the internal states are not combined (that is, the query works less efficiently than it could). In this case, use the `quantiles` (and similar) functions.

quantileDeterministic(level)(x, determinator)

Works the same way as the `quantile` function, but the result is deterministic and does not depend on the order of query execution.

To achieve this, the function takes a second argument – the "determinator". This is a number whose hash is used instead of a random number generator in the reservoir sampling algorithm. For the function to work correctly, the same determinator value should not occur too often. For the determinator, you can use an event ID, user ID, and so on.

Don't use this function for calculating timings. There is a more suitable function for this purpose: `quantileTiming` .

quantileTiming(level)(x)

Computes the quantile of 'level' with a fixed precision. Works for numbers. Intended for calculating quantiles of page loading time in milliseconds.

If the value is greater than 30,000 (a page loading time of more than 30 seconds), the result is equated to 30,000.

If the total value is not more than about 5670, then the calculation is accurate.

Otherwise:

- if the time is less than 1024 ms, then the calculation is accurate.
- otherwise the calculation is rounded to a multiple of 16 ms.

When passing negative values to the function, the behavior is undefined.

The returned value has the Float32 type. If no values were passed to the function (when using `quantileTimingIf`), 'nan' is returned. The purpose of this is to differentiate these instances from zeros. See the note on sorting NaNs in "ORDER BY clause".

The result is determinate (it doesn't depend on the order of query processing).

For its purpose (calculating quantiles of page loading times), using this function is more effective and the result is more accurate than for the `quantile` function.

quantileTimingWeighted(level)(x, weight)

Differs from the `quantileTiming` function in that it has a second argument, "weights". Weight is a non-negative integer. The result is calculated as if the `x` value were passed `weight` number of times to the `quantileTiming` function.

quantileExact(level)(x)

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

quantileExactWeighted(level)(x, weight)

Computes the quantile of 'level' exactly. In addition, each value is counted with its weight, as if it is present 'weight' times. The arguments of the function can be considered as histograms, where the value 'x' corresponds to a histogram "column" of the height 'weight', and the function itself can be considered as a summation of histograms.

A hash table is used as the algorithm. Because of this, if the passed values are frequently repeated, the function consumes less RAM than `quantileExact`. You can use this function instead of `quantileExact` and specify the weight as 1.

quantileTDigest(level)(x)

Approximates the quantile level using the t-digest [https://github.com/tdunning/t-digest/blob/master/docs/t-digest-paper/histo.pdf] algorithm. The maximum error is 1%. Memory consumption by State is proportional to the logarithm of the number of passed values.

The performance of the function is lower than for `quantile` or `quantileTiming`. In terms of the ratio of State size to precision, this function is much better than `quantile`.

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

median(x)

All the quantile functions have corresponding median functions: `median`, `medianDeterministic`, `medianTiming`, `medianTimingWeighted`, `medianExact`, `medianExactWeighted`, `medianTDigest`. They are synonyms and their behavior is identical.

quantiles(level1, level2, …)(x)

All the quantile functions also have corresponding quantiles functions: `quantiles`, `quantilesDeterministic`, `quantilesTiming`, `quantilesTimingWeighted`, `quantilesExact`, `quantilesExactWeighted`, `quantilesTDigest`. These functions calculate all the quantiles of the listed levels in one pass, and return an array of the resulting values.

varSamp(x)

Calculates the amount `Σ((x - x̅)^2) / (n - 1)`, where `n` is the sample size and `x̅` is the average value of `x`.

It represents an unbiased estimate of the variance of a random variable, if the values passed to the function are a sample of this random amount.

Returns `Float64`. When `n <= 1`, returns `+∞`.

varPop(x)

Calculates the amount `Σ((x - x̅)^2) / (n - 1)`, where `n` is the sample size and `x̅` is the average value of `x`.

In other words, dispersion for a set of values. Returns `Float64`.

stddevSamp(x)

The result is equal to the square root of `varSamp(x)`.

stddevPop(x)

The result is equal to the square root of `varPop(x)`.

topK(N)(column)

Returns an array of the most frequent values in the specified column. The resulting array is sorted in descending order of frequency of values (not by the values themselves).

Implements the Filtered Space-Saving [http://www.l2f.inesc-id.pt/~fmmb/wiki/uploads/Work/misnis.ref0a.pdf] algorithm for analyzing TopK, based on the reduce-and-combine algorithm from Parallel Space Saving [https://arxiv.org/pdf/1401.0702.pdf].

```
topK(N)(column)
```

This function doesn't provide a guaranteed result. In certain situations, errors might occur and it might return frequent values that aren't the most frequent values.

We recommend using the `N < 10` value; performance is reduced with large `N` values. Maximum value of `N = 65536`.

Arguments

- 'N' is the number of values.
- ' x ' – The column.

Example

Take the OnTime [#example_datasets-ontime] data set and select the three most frequently occurring values in the `AirlineID` column.

```
SELECT topK(3)(AirlineID) AS res
FROM ontime
```

```
┌─res─────────────────┐
│ [19393,19790,19805] │
└─────────────────────┘
```

covarSamp(x, y)

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

Returns Float64. When `n <= 1`, returns +∞.

covarPop(x, y)

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

corr(x, y)

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

[]

Aggregate function combinators

The name of an aggregate function can have a suffix appended to it. This changes the way the aggregate function works.

[]

-If

The suffix -If can be appended to the name of any aggregate function. In this case, the aggregate function accepts an extra argument – a condition (Uint8 type). The aggregate function processes only the rows that trigger the condition. If the condition was not triggered even once, it returns a default value (usually zeros or empty strings).

Examples: `sumIf(column, cond)`, `countIf(cond)`, `avgIf(x, cond)`, `quantilesTimingIf(level1, level2)(x, cond)`, `argMinIf(arg, val, cond)` and so on.

With conditional aggregate functions, you can calculate aggregates for several conditions at once, without using subqueries and `JOIN`s. For example, in Yandex.Metrica, conditional aggregate functions are used to implement the segment comparison functionality.

-Array

The -Array suffix can be appended to any aggregate function. In this case, the aggregate function takes arguments of the 'Array(T)' type (arrays) instead of 'T' type arguments. If the aggregate function accepts multiple arguments, this must be arrays of equal lengths. When processing arrays, the aggregate function works like the original aggregate function across all array elements.

Example 1: `sumArray(arr)` - Totals all the elements of all 'arr' arrays. In this example, it could have been written more simply: `sum(arraySum(arr))`.

Example 2: `uniqArray(arr)` – Count the number of unique elements in all 'arr' arrays. This could be done an easier way: `uniq(arrayJoin(arr))`, but it's not always possible to add 'arrayJoin' to a query.

-If and -Array can be combined. However, 'Array' must come first, then 'If'. Examples: `uniqArrayIf(arr, cond)`, `quantilesTimingArrayIf(level1, level2)(arr, cond)`. Due to this order, the 'cond' argument can't be an array.

-State

If you apply this combinator, the aggregate function doesn't return the resulting value (such as the number of unique values for the `uniq` function), but an intermediate state of the aggregation (for `uniq`, this is the hash table for calculating the number of unique values). This is an AggregateFunction(...) that can be used for further processing or stored in a table to finish aggregating later. See the sections "AggregatingMergeTree" and "Functions for working with intermediate aggregation states".

-Merge

If you apply this combinator, the aggregate function takes the intermediate aggregation state as an argument, combines the states to finish aggregation, and returns the resulting value.

-MergeState.

Merges the intermediate aggregation states in the same way as the -Merge combinator. However, it doesn't return the resulting value, but an intermediate aggregation state, similar to the -State combinator.

-ForEach

Converts an aggregate function for tables into an aggregate function for arrays that aggregates the corresponding array items and returns an array of results. For example, `sumForEach` for the arrays `[1, 2]`, `[3, 4, 5]` and `[6, 7]` returns the result `[10, 13, 5]` after adding together the corresponding array items.

[]

Parametric aggregate functions

Some aggregate functions can accept not only argument columns (used for compression), but a set of parameters – constants for initialization. The syntax is two pairs of brackets instead of one. The first is for parameters, and the second is for arguments.

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

Pattern matching for event chains.

`pattern` is a string containing a pattern to match. The pattern is similar to a regular expression.

`time` is the time of the event with the DateTime type.

`cond1`, `cond2` ... is from one to 32 arguments of type UInt8 that indicate whether a certain condition was met for the event.

The function collects a sequence of events in RAM. Then it checks whether this sequence matches the pattern. It returns UInt8: 0 if the pattern isn't matched, or 1 if it matches.

Example: `sequenceMatch ('(?1).*(?2)')(EventTime, URL LIKE '%company%', URL LIKE '%cart%')`

- whether there was a chain of events in which a pageview with 'company' in the address occurred earlier than a pageview with 'cart' in the address.

This is a singular example. You could write it using other aggregate functions:

```
minIf(EventTime, URL LIKE '%company%') < maxIf(EventTime, URL LIKE '%cart%').
```

However, there is no such solution for more complex situations.

Pattern syntax:

`(?1)` refers to the condition (any number can be used in place of 1).

`.*` is any number of any events.

`(?t>=1800)` is a time condition.

Any quantity of any type of events is allowed over the specified time.

Instead of `>=`, the following operators can be used: `<`, `>`, `<=`.

Any number may be specified in place of 1800.

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

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

Works the same way as the sequenceMatch function, but instead of returning whether there is an event chain, it returns UInt64 with the number of event chains found. Chains are searched for without overlapping. In other words, the next chain can start only after the end of the previous one.

windowFunnel(window)(timestamp, cond1, cond2, cond3, ...)

Searches for event chains in a sliding time window and calculates the maximum number of events that occurred from the chain.

```
windowFunnel(window)(timestamp, cond1, cond2, cond3, ...)
```

Parameters:

- `window` — Length of the sliding window in seconds.
- `timestamp` — Name of the column containing the timestamp. Data type: DateTime [#data_type-datetime] or UInt32 [#data_type-int].
- `cond1`, `cond2` ... — Conditions or data describing the chain of events. Data type: `UInt8`. Values can be 0 or 1.

Algorithm

- The function searches for data that triggers the first condition in the chain and sets the event counter to 1. This is the moment when the sliding window starts.
- If events from the chain occur sequentially within the window, the counter is incremented. If the sequence of events is disrupted, the counter isn't incremented.
- If the data has multiple event chains at varying points of completion, the function will only output the size of the longest chain.

Returned value

- Integer. The maximum number of consecutive triggered conditions from the chain within the sliding time window. All the chains in the selection are analyzed.

Example

Determine if one hour is enough for the user to select a phone and purchase it in the online store.

Set the following chain of events:

1. The user logged in to their account on the store (`eventID=1001`).
2. The user searches for a phone (`eventID = 1003, product = 'phone'`).
3. The user placed an order (`eventID = 1009`).

To find out how far the user `user_id` could get through the chain in an hour in January of 2017, make the query:

```
SELECT
    level,
    count() AS c
FROM
(
    SELECT
        user_id,
        windowFunnel(3600)(timestamp, eventID = 1001, eventID = 1003 AND product = 'phone', eventID = 1009)
AS level
    FROM trend_event
    WHERE (event_date >= '2017-01-01') AND (event_date <= '2017-01-31')
    GROUP BY user_id
)
GROUP BY level
ORDER BY level
```

Simply, the level value could only be 0, 1, 2, 3, it means the maxium event action stage that one user could reach.

retention(cond1, cond2, ...)

Retention refers to the ability of a company or product to retain its customers over some specified periods.

`cond1` , `cond2` ... is from one to 32 arguments of type UInt8 that indicate whether a certain condition was met for the event

Example:

Consider you are doing a website analytics, intend to calculate the retention of customers

This could be easily calculate by `retention`

```
SELECT
    sum(r[1]) AS r1,
    sum(r[2]) AS r2,
    sum(r[3]) AS r3
FROM
(
    SELECT
        uid,
        retention(date = '2018-08-10', date = '2018-08-11', date = '2018-08-12') AS r
    FROM events
    WHERE date IN ('2018-08-10', '2018-08-11', '2018-08-12')
    GROUP BY uid
)
```

Simply, `r1` means the number of unique visitors who met the `cond1` condition, `r2` means the number of unique visitors who met `cond1` and `cond2` conditions, `r3` means the number of unique visitors who met `cond1` and `cond3` conditions.

uniqUpTo(N)(x)

Calculates the number of different argument values if it is less than or equal to N. If the number of different argument values is greater than N, it returns N + 1.

Recommended for use with small Ns, up to 10. The maximum value of N is 100.

For the state of an aggregate function, it uses the amount of memory equal to 1 + N * the size of one value of bytes. For strings, it stores a non-cryptographic hash of 8 bytes. That is, the calculation is approximated for strings.

The function also works for several arguments.

It works as fast as possible, except for cases when a large N value is used and the number of unique values is slightly less than N.

Usage example:

```
Problem: Generate a report that shows only keywords that produced at least 5 unique users.
Solution: Write in the GROUP BY query SearchPhrase HAVING uniqUpTo(4)(UserID) >= 5
```

Table functions

Table functions can be specified in the FROM clause instead of the database and table names. Table functions can only be used if 'readonly' is not set. Table functions aren't related to other functions.

[]

file

Creates a table from a file.

```
file(path, format, structure)
```

Input parameters

- `path` — The relative path to the file from user_files_path [#user_files_path].
- `format` — The format [#formats] of the file.
- `structure` — Structure of the table. Format `'colunmn1_name column1_ype, column2_name column2_type, ...'`.

Returned value

A table with the specified structure for reading or writing data in the specified file.

Example

Setting `user_files_path` and the contents of the file `test.csv`:

```
$ grep user_files_path /etc/clickhouse-server/config.xml
    <user_files_path>/var/lib/clickhouse/user_files/</user_files_path>

$ cat /var/lib/clickhouse/user_files/test.csv
    1,2,3
    3,2,1
    78,43,45
```

Table from `test.csv` and selection of the first two rows from it:

```
SELECT *
FROM file('test.csv', 'CSV', 'column1 UInt32, column2 UInt32, column3 UInt32')
LIMIT 2
```

```
┌─column1─┬─column2─┬─column3─┐
│       1 │       2 │       3 │
│       3 │       2 │       1 │
└─────────┴─────────┴─────────┘
```

```
-- getting the first 10 lines of a table that contains 3 columns of UInt32 type from a CSV file
SELECT * FROM file('test.csv', 'CSV', 'column1 UInt32, column2 UInt32, column3 UInt32') LIMIT 10
```

merge

`merge(db_name, 'tables_regexp')` – Creates a temporary Merge table. For more information, see the section "Table engines, Merge".

The table structure is taken from the first table encountered that matches the regular expression.

numbers

`numbers(N)` – Returns a table with the single 'number' column (UInt64) that contains integers from 0 to N-1. `numbers(N, M)` - Returns a table with the single 'number' column (UInt64) that contains integers from N to (N + M - 1).

Similar to the `system.numbers` table, it can be used for testing and generating successive values, `numbers(N, M)` more efficient than `system.numbers`.

The following queries are equivalent:

```
SELECT * FROM numbers(10);
SELECT * FROM numbers(0, 10);
SELECT * FROM system.numbers LIMIT 10;
```

Examples:

```
-- Generate a sequence of dates from 2010-01-01 to 2010-12-31
select toDate('2010-01-01') + number as d FROM numbers(365);
```

[]

remote

Allows you to access remote servers without creating a `Distributed` table.

Signatures:

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

`addresses_expr` – An expression that generates addresses of remote servers. This may be just one server address. The server address is `host:port`, or just `host`. The host can be specified as the server name, or as the IPv4 or IPv6 address. An IPv6 address is specified in square brackets. The port is the TCP port on the remote server. If the port is omitted, it uses `tcp_port` from the server's config file (by default, 9000).

Important

The port is required for an IPv6 address.

Examples:

```
example01-01-1
example01-01-1:9000
localhost
127.0.0.1
[::]:9000
[2a02:6b8:0:1111::11]:9000
```

Multiple addresses can be comma-separated. In this case, ClickHouse will use distributed processing, so it will send the query to all specified addresses (like to shards with different data).

Example:

```
example01-01-1,example01-02-1
```

Part of the expression can be specified in curly brackets. The previous example can be written as follows:

```
example01-0{1,2}-1
```

Curly brackets can contain a range of numbers separated by two dots (non-negative integers). In this case, the range is expanded to a set of values that generate shard addresses. If the first number starts with zero, the values are formed with the same zero alignment. The previous example can be written as follows:

```
example01-{01..02}-1
```

If you have multiple pairs of curly brackets, it generates the direct product of the corresponding sets.

Addresses and parts of addresses in curly brackets can be separated by the pipe symbol (|). In this case, the corresponding sets of addresses are interpreted as replicas, and the query will be sent to the first healthy replica. However, the replicas are iterated in the order currently set in the load_balancing [#settings-load_balancing] setting.

Example:

```
example01-{01..02}-{1|2}
```

This example specifies two shards that each have two replicas.

The number of addresses generated is limited by a constant. Right now this is 1000 addresses.

Using the `remote` table function is less optimal than creating a `Distributed` table, because in this case, the server connection is re-established for every request. In addition, if host names are set, the names are resolved, and errors are not counted when working with various replicas. When processing a large number of queries, always create the `Distributed` table ahead of time, and don't use the `remote` table function.

The `remote` table function can be useful in the following cases:

- Accessing a specific server for data comparison, debugging, and testing.
- Queries between various ClickHouse clusters for research purposes.
- Infrequent distributed requests that are made manually.
- Distributed requests where the set of servers is re-defined each time.

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

[]

url

`url(URL, format, structure)` - returns a table created from the `URL` with given `format` and `structure`.

URL - HTTP or HTTPS server address, which can accept `GET` and/or `POST` requests.

format - format [#formats] of the data.

structure - table structure in `'UserID UInt64, Name String'` format. Determines column names and types.

Example

```
-- getting the first 3 lines of a table that contains columns of String and UInt32 type from HTTP-server
which answers in CSV format.
SELECT * FROM url('http://127.0.0.1:12345/', CSV, 'column1 String, column2 UInt32') LIMIT 3
```

[]

jdbc

`jdbc(jdbc_connection_uri, schema, table)` - returns table that is connected via JDBC driver.

This table function requires separate `clickhouse-jdbc-bridge` program to be running. It supports Nullable types (based on DDL of remote table that is queried).

Examples

```
SELECT * FROM jdbc('jdbc:mysql://localhost:3306/?user=root&password=root', 'schema', 'table')
```

```
SELECT * FROM jdbc('mysql://localhost:3306/?user=root&password=root', 'schema', 'table')
```

```
SELECT * FROM jdbc('datasource://mysql-local', 'schema', 'table')
```

Dictionaries

A dictionary is a mapping (`key -> attributes`) that is convenient for various types of reference lists.

ClickHouse supports special functions for working with dictionaries that can be used in queries. It is easier and more efficient to use dictionaries with functions than a `JOIN` with reference tables.

NULL [#null-literal] values can't be stored in a dictionary.

ClickHouse supports:

- Built-in dictionaries [#internal_dicts] with a specific set of functions [#ym_dict_functions].
- Plug-in (external) dictionaries [#dicts-external_dicts] with a set of functions [#ext_dict_functions].

[]

External Dictionaries

You can add your own dictionaries from various data sources. The data source for a dictionary can be a local text or executable file, an HTTP(s) resource, or another DBMS. For more information, see "Sources for external dictionaries [#dicts-external_dicts_dict_sources]".

ClickHouse:

> - Fully or partially stores dictionaries in RAM.
> - Periodically updates dictionaries and dynamically loads missing values. In other words, dictionaries can be loaded dynamically.

The configuration of external dictionaries is located in one or more files. The path to the configuration is specified in the dictionaries_config [#server_settings-dictionaries_config] parameter.

Dictionaries can be loaded at server startup or at first use, depending on the dictionaries_lazy_load [#server_settings-dictionaries_lazy_load] setting.

The dictionary config file has the following format:

```
<yandex>
    <comment>An optional element with any content. Ignored by the ClickHouse server.</comment>

    <!--Optional element. File name with substitutions-->
    <include_from>/etc/metrika.xml</include_from>


    <dictionary>
        <!-- Dictionary configuration -->
    </dictionary>

    ...

    <dictionary>
        <!-- Dictionary configuration -->
    </dictionary>
</yandex>
```

You can configure [#dicts-external_dicts_dict] any number of dictionaries in the same file. The file format is preserved even if there is only one dictionary (i.e. <yandex><dictionary> <!--configuration -> </dictionary></yandex>).

See also "Functions for working with external dictionaries [#ext_dict_functions]".

Attention

You can convert values for a small dictionary by describing it in a `SELECT` query (see the transform [#other_functions-transform] function). This functionality is not related to external dictionaries.

[]

Configuring an External Dictionary

The dictionary configuration has the following structure:

```
<dictionary>
    <name>dict_name</name>

    <source>
      <!-- Source configuration -->
    </source>

    <layout>
      <!-- Memory layout configuration -->
    </layout>

    <structure>
      <!-- Complex key configuration -->
    </structure>

    <lifetime>
      <!-- Lifetime of dictionary in memory -->
    </lifetime>
</dictionary>
```

- name – The identifier that can be used to access the dictionary. Use the characters `[a-zA-Z0-9_\-]` .
- source [#dicts-external_dicts_dict_sources] — Source of the dictionary.
- layout [#dicts-external_dicts_dict_layout] — Dictionary layout in memory.
- structure [#dicts-external_dicts_dict_structure] — Structure of the dictionary . A key and attributes that can be retrieved by this key.
- lifetime [#dicts-external_dicts_dict_lifetime] — Frequency of dictionary updates.

[]

Storing Dictionaries in Memory

There are a variety of ways [#dicts-external_dicts_dict_layout-manner] to store dictionaries in memory.

We recommend flat [#dicts-external_dicts_dict_layout-flat], hashed [#dicts-external_dicts_dict_layout-hashed]andcomplex_key_hashed [#dicts-external_dicts_dict_layout-complex_key_hashed]. which provide optimal processing speed.

Caching is not recommended because of potentially poor performance and difficulties in selecting optimal parameters. Read more in the section "cache [#dicts-external_dicts_dict_layout-cache]".

There are several ways to improve dictionary performance:

- Call the function for working with the dictionary after `GROUP BY`.
- Mark attributes to extract as injective. An attribute is called injective if different attribute values correspond to different keys. So when `GROUP BY` uses a function that fetches an attribute value by the key, this function is automatically taken out of `GROUP BY`.

ClickHouse generates an exception for errors with dictionaries. Examples of errors:

- The dictionary being accessed could not be loaded.
- Error querying a `cached` dictionary.

You can view the list of external dictionaries and their statuses in the `system.dictionaries` table.

The configuration looks like this:

```
<yandex>
    <dictionary>
        ...
        <layout>
            <layout_type>
                <!-- layout settings -->
            </layout_type>
        </layout>
        ...
    </dictionary>
</yandex>
```

[]

Ways to Store Dictionaries in Memory

- flat [#dicts-external_dicts_dict_layout-flat]
- hashed [#dicts-external_dicts_dict_layout-hashed]
- cache [#dicts-external_dicts_dict_layout-cache]
- range_hashed [#dicts-external_dicts_dict_layout-range_hashed]
- complex_key_hashed [#dicts-external_dicts_dict_layout-complex_key_hashed]
- complex_key_cache [#dicts-external_dicts_dict_layout-complex_key_cache]
- ip_trie [#dicts-external_dicts_dict_layout-ip_trie]

[]

flat

The dictionary is completely stored in memory in the form of flat arrays. How much memory does the dictionary use? The amount is proportional to the size of the largest key (in space used).

The dictionary key has the `UInt64` type and the value is limited to 500,000. If a larger key is discovered when creating the dictionary, ClickHouse throws an exception and does not create the dictionary.

All types of sources are supported. When updating, data (from a file or from a table) is read in its entirety.

This method provides the best performance among all available methods of storing the dictionary.

Configuration example:

```
<layout>
  <flat />
</layout>
```

[]

hashed

The dictionary is completely stored in memory in the form of a hash table. The dictionary can contain any number of elements with any identifiers In practice, the number of keys can reach tens of millions of items.

All types of sources are supported. When updating, data (from a file or from a table) is read in its entirety.

Configuration example:

```
<layout>
  <hashed />
</layout>
```

[]

complex_key_hashed

This type of storage is for use with composite keys [#dicts-external_dicts_dict_structure]. Similar to `hashed`.

Configuration example:

```
<layout>
  <complex_key_hashed />
</layout>
```

[]

range_hashed

The dictionary is stored in memory in the form of a hash table with an ordered array of ranges and their corresponding values.

This storage method works the same way as hashed and allows using date/time ranges in addition to the key, if they appear in the dictionary.

Example: The table contains discounts for each advertiser in the format:

```
+---------------+---------------------+-------------------+--------+
| advertiser id | discount start date | discount end date | amount |
+===============+=====================+===================+========+
| 123           | 2015-01-01          | 2015-01-15        | 0.15   |
+---------------+---------------------+-------------------+--------+
| 123           | 2015-01-16          | 2015-01-31        | 0.25   |
+---------------+---------------------+-------------------+--------+
| 456           | 2015-01-01          | 2015-01-15        | 0.05   |
+---------------+---------------------+-------------------+--------+
```

To use a sample for date ranges, define the `range_min` and `range_max` elements in the structure [#dicts-

external_dicts_dict_structure].

Example:

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

To work with these dictionaries, you need to pass an additional date argument to the `dictGetT` function:

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

This function returns the value for the specified `id` s and the date range that includes the passed date.

Details of the algorithm:

- If the `id` is not found or a range is not found for the `id` , it returns the default value for the dictionary.
- If there are overlapping ranges, you can use any.
- If the range delimiter is `NULL` or an invalid date (such as 1900-01-01 or 2039-01-01), the range is left open. The range can be open on both sides.

Configuration example:

```
<yandex>
        <dictionary>

                ...

                <layout>
                        <range_hashed />
                </layout>

                <structure>
                        <id>
                                <name>Abcdef</name>
                        </id>
                        <range_min>
                                <name>StartDate</name>
                        </range_min>
                        <range_max>
                                <name>EndDate</name>
                        </range_max>
                        <attribute>
                                <name>XXXType</name>
                                <type>String</type>
                                <null_value />
                        </attribute>
                </structure>

        </dictionary>
</yandex>
```

[]

cache

The dictionary is stored in a cache that has a fixed number of cells. These cells contain frequently used elements.

When searching for a dictionary, the cache is searched first. For each block of data, all keys that are not found in the cache or are outdated are requested from the source using `SELECT attrs... FROM db.table WHERE id IN (k1, k2, ...)`. The received data is then written to the cache.

For cache dictionaries, the expiration lifetime [#dicts-external_dicts_dict_lifetime] of data in the cache can be set. If more time than `lifetime` has passed since loading the data in a cell, the cell's value is not used, and it is re-requested the next time it needs to be used.

This is the least effective of all the ways to store dictionaries. The speed of the cache depends strongly on correct settings and the usage scenario. A cache type dictionary performs well only when the hit rates are high enough (recommended 99% and higher). You can view the average hit rate in the `system.dictionaries` table.

To improve cache performance, use a subquery with `LIMIT`, and call the function with the dictionary externally.

Supported sources [#dicts-external_dicts_dict_sources]: MySQL, ClickHouse, executable, HTTP.

Example of settings:

```
<layout>
    <cache>
        <!-- The size of the cache, in number of cells. Rounded up to a power of two. -->
        <size_in_cells>1000000000</size_in_cells>
    </cache>
</layout>
```

Set a large enough cache size. You need to experiment to select the number of cells:

1. Set some value.
2. Run queries until the cache is completely full.
3. Assess memory consumption using the `system.dictionaries` table.
4. Increase or decrease the number of cells until the required memory consumption is reached.

Warning

Do not use ClickHouse as a source, because it is slow to process queries with random reads.

[]

complex_key_cache

This type of storage is for use with composite keys [#dicts-external_dicts_dict_structure]. Similar to `cache`.

[]

ip_trie

This type of storage is for mapping network prefixes (IP addresses) to metadata such as ASN.

Example: The table contains network prefixes and their corresponding AS number and country code:

```
  +-----------------+-------+--------+
  | prefix          | asn   | cca2   |
  +=================+=======+========+
  | 202.79.32.0/20  | 17501 | NP     |
  +-----------------+-------+--------+
  | 2620:0:870::/48 | 3856  | US     |
  +-----------------+-------+--------+
  | 2a02:6b8:1::/48 | 13238 | RU     |
  +-----------------+-------+--------+
  | 2001:db8::/32   | 65536 | ZZ     |
  +-----------------+-------+--------+
```

When using this type of layout, the structure must have a composite key.

Example:

```
<structure>
    <key>
        <attribute>
            <name>prefix</name>
            <type>String</type>
        </attribute>
    </key>
    <attribute>
            <name>asn</name>
            <type>UInt32</type>
            <null_value />
    </attribute>
    <attribute>
            <name>cca2</name>
            <type>String</type>
            <null_value>??</null_value>
    </attribute>
    ...
```

The key must have only one String type attribute that contains an allowed IP prefix. Other types are not supported yet.

For queries, you must use the same functions (`dictGetT` with a tuple) as for dictionaries with composite keys:

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

The function takes either `UInt32` for IPv4, or `FixedString(16)` for IPv6:

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

Other types are not supported yet. The function returns the attribute for the prefix that corresponds to this IP address. If there are overlapping prefixes, the most specific one is returned.

Data is stored in a `trie` . It must completely fit into RAM.

[]

Dictionary Updates

ClickHouse periodically updates the dictionaries. The update interval for fully downloaded dictionaries and the invalidation interval for cached dictionaries are defined in the `<lifetime>` tag in seconds.

Dictionary updates (other than loading for first use) do not block queries. During updates, the old version of a dictionary is used. If an error occurs during an update, the error is written to the server log, and queries continue using the old version of dictionaries.

Example of settings:

```
<dictionary>
    ...
    <lifetime>300</lifetime>
    ...
</dictionary>
```

Setting `<lifetime> 0</lifetime>` prevents updating dictionaries.

You can set a time interval for upgrades, and ClickHouse will choose a uniformly random time within this range. This is necessary in order to distribute the load on the dictionary source when upgrading on a large number of servers.

Example of settings:

```
<dictionary>
    ...
    <lifetime>
        <min>300</min>
        <max>360</max>
    </lifetime>
    ...
</dictionary>
```

When upgrading the dictionaries, the ClickHouse server applies different logic depending on the type of source [#dicts-external_dicts_dict_sources]:

- For a text file, it checks the time of modification. If the time differs from the previously recorded time, the dictionary is updated.
- For MyISAM tables, the time of modification is checked using a `SHOW TABLE STATUS` query.
- Dictionaries from other sources are updated every time by default.

For MySQL (InnoDB), ODBC and ClickHouse sources, you can set up a query that will update the dictionaries only if they really changed, rather than each time. To do this, follow these steps:

- The dictionary table must have a field that always changes when the source data is updated.
- The settings of the source must specify a query that retrieves the changing field. The ClickHouse server interprets the query result as a row, and if this row has changed relative to its previous state, the dictionary is updated. Specify the query in the `<invalidate_query>` field in the settings for the source [#dicts-external_dicts_dict_sources].

Example of settings:

```
<dictionary>
    ...
    <odbc>
      ...
      <invalidate_query>SELECT update_time FROM dictionary_source where id = 1</invalidate_query>
    </odbc>
    ...
</dictionary>
```

[]

Sources of External Dictionaries

An external dictionary can be connected from many different sources.

The configuration looks like this:

```
<yandex>
  <dictionary>
    ...
    <source>
      <source_type>
        <!-- Source configuration -->
      </source_type>
    </source>
    ...
  </dictionary>
  ...
</yandex>
```

The source is configured in the `source` section.

Types of sources (`source_type`):

- Local file [#dicts-external_dicts_dict_sources-local_file]
- Executable file [#dicts-external_dicts_dict_sources-executable]
- HTTP(s) [#dicts-external_dicts_dict_sources-http]
- DBMS
  - MySQL [#dicts-external_dicts_dict_sources-mysql]
  - ClickHouse [#dicts-external_dicts_dict_sources-clickhouse]
  - MongoDB [#dicts-external_dicts_dict_sources-mongodb]
  - ODBC [#dicts-external_dicts_dict_sources-odbc]

[]

Local File

Example of settings:

```
<source>
  <file>
    <path>/opt/dictionaries/os.tsv</path>
    <format>TabSeparated</format>
  </file>
</source>
```

Setting fields:

- `path` – The absolute path to the file.
- `format` – The file format. All the formats described in "Formats [#formats]" are supported.

[]

Executable File

Working with executable files depends on how the dictionary is stored in memory [#dicts-external_dicts_dict_layout]. If the dictionary is stored using `cache` and `complex_key_cache` , ClickHouse requests the necessary keys by sending a request to the executable file's `STDIN` .

Example of settings:

```
<source>
    <executable>
        <command>cat /opt/dictionaries/os.tsv</command>
        <format>TabSeparated</format>
    </executable>
</source>
```

Setting fields:

- `command` – The absolute path to the executable file, or the file name (if the program directory is written to `PATH`).
- `format` – The file format. All the formats described in "Formats [#formats]" are supported.

[]

HTTP(s)

Working with an HTTP(s) server depends on how the dictionary is stored in memory [#dicts-external_dicts_dict_layout]. If

the dictionary is stored using `cache` and `complex_key_cache`, ClickHouse requests the necessary keys by sending a request via the `POST` method.

Example of settings:

```
<source>
    <http>
        <url>http://[::1]/os.tsv</url>
        <format>TabSeparated</format>
    </http>
</source>
```

In order for ClickHouse to access an HTTPS resource, you must configure openSSL [#server_settings-openSSL] in the server configuration.

Setting fields:

- `url` – The source URL.
- `format` – The file format. All the formats described in "Formats [#formats]" are supported.

[]

ODBC

You can use this method to connect any database that has an ODBC driver.

Example of settings:

```
<odbc>
    <db>DatabaseName</db>
    <table>ShemaName.TableName</table>
    <connection_string>DSN=some_parameters</connection_string>
    <invalidate_query>SQL_QUERY</invalidate_query>
</odbc>
```

Setting fields:

- `db` – Name of the database. Omit it if the database name is set in the `<connection_string>` parameters.
- `table` – Name of the table and schema if exists.
- `connection_string` – Connection string.
- `invalidate_query` – Query for checking the dictionary status. Optional parameter. Read more in the section Updating dictionaries [#dicts-external_dicts_dict_lifetime].

ClickHouse receives quoting symbols from ODBC-driver and quote all settings in queries to driver, so it's necessary to set table name accordingly to table name case in database.

Known vulnerability of the ODBC dictionary functionality

Attention

When connecting to the database through the ODBC driver connection parameter `Servername` can be substituted. In this case values of `USERNAME` and `PASSWORD` from `odbc.ini` are sent to the remote server and can be compromised.

Example of insecure use

Let's configure unixODBC for PostgreSQL. Content of `/etc/odbc.ini`:

```
[gregtest]
Driver = /usr/lib/psqlodbca.so
Servername = localhost
PORT = 5432
DATABASE = test_db
##OPTION = 3
USERNAME = test
PASSWORD = test
```

If you then make a query such as

```
SELECT * FROM odbc('DSN=gregtest;Servername=some-server.com', 'test_db');
```

ODBC driver will send values of `USERNAME` and `PASSWORD` from `odbc.ini` to `some-server.com`.

Example of Connecting PostgreSQL

Ubuntu OS.

Installing unixODBC and the ODBC driver for PostgreSQL:

```
sudo apt-get install -y unixodbc odbcinst odbc-postgresql
```

Configuring `/etc/odbc.ini` (or `~/.odbc.ini`):

```
    [DEFAULT]
    Driver = myconnection

    [myconnection]
    Description         = PostgreSQL connection to my_db
    Driver              = PostgreSQL Unicode
    Database            = my_db
    Servername          = 127.0.0.1
    UserName            = username
    Password            = password
    Port                = 5432
    Protocol            = 9.3
    ReadOnly            = No
    RowVersioning       = No
    ShowSystemTables    = No
    ConnSettings        =
```

The dictionary configuration in ClickHouse:

```
<yandex>
    <dictionary>
        <name>table_name</name>
        <source>
            <odbc>
                <!-- You can specify the following parameters in connection_string: -->
                <!-- DSN=myconnection;UID=username;PWD=password;HOST=127.0.0.1;PORT=5432;DATABASE=my_db -->
                <connection_string>DSN=myconnection</connection_string>
                <table>postgresql_table</table>
            </odbc>
        </source>
        <lifetime>
            <min>300</min>
            <max>360</max>
        </lifetime>
        <layout>
            <hashed/>
        </layout>
        <structure>
            <id>
                <name>id</name>
            </id>
            <attribute>
                <name>some_column</name>
                <type>UInt64</type>
                <null_value>0</null_value>
            </attribute>
        </structure>
    </dictionary>
</yandex>
```

You may need to edit `odbc.ini` to specify the full path to the library with the driver `DRIVER=/usr/local/lib/psqlodbcw.so`.

Example of Connecting MS SQL Server

Ubuntu OS.

Installing the driver: :

```
    sudo apt-get install tdsodbc freetds-bin sqsh
```

Configuring the driver: :

```
$ cat /etc/freetds/freetds.conf
...

[MSSQL]
host = 192.168.56.101
port = 1433
tds version = 7.0
client charset = UTF-8

$ cat /etc/odbcinst.ini
...

[FreeTDS]
Description     = FreeTDS
Driver          = /usr/lib/x86_64-linux-gnu/odbc/libtdsodbc.so
Setup           = /usr/lib/x86_64-linux-gnu/odbc/libtdsS.so
FileUsage       = 1
UsageCount      = 5

$ cat ~/.odbc.ini
...

[MSSQL]
Description     = FreeTDS
Driver          = FreeTDS
Servername      = MSSQL
Database        = test
UID             = test
PWD             = test
Port            = 1433
```

Configuring the dictionary in ClickHouse:

```
<yandex>
    <dictionary>
        <name>test</name>
        <source>
            <odbc>
                <table>dict</table>
                <connection_string>DSN=MSSQL;UID=test;PWD=test</connection_string>
            </odbc>
        </source>

        <lifetime>
            <min>300</min>
            <max>360</max>
        </lifetime>

        <layout>
            <flat />
        </layout>

        <structure>
            <id>
                <name>k</name>
            </id>
            <attribute>
                <name>s</name>
                <type>String</type>
                <null_value></null_value>
            </attribute>
        </structure>
    </dictionary>
</yandex>
```

DBMS

[]

MySQL

Example of settings:

```
<source>
  <mysql>
      <port>3306</port>
      <user>clickhouse</user>
      <password>qwerty</password>
      <replica>
          <host>example01-1</host>
          <priority>1</priority>
      </replica>
      <replica>
          <host>example01-2</host>
          <priority>1</priority>
      </replica>
      <db>db_name</db>
      <table>table_name</table>
      <where>id=10</where>
      <invalidate_query>SQL_QUERY</invalidate_query>
  </mysql>
</source>
```

Setting fields:

- `port` – The port on the MySQL server. You can specify it for all replicas, or for each one individually (inside `<replica>`).
- `user` – Name of the MySQL user. You can specify it for all replicas, or for each one individually (inside `<replica>`).
- `password` – Password of the MySQL user. You can specify it for all replicas, or for each one individually (inside `<replica>`).
- `replica` – Section of replica configurations. There can be multiple sections.
  - `replica/host` – The MySQL host.

  * `replica/priority` – The replica priority. When attempting to connect, ClickHouse traverses the replicas in order of priority. The lower the number, the higher the priority.
- `db` – Name of the database.
- `table` – Name of the table.
- `where` – The selection criteria. Optional parameter.
- `invalidate_query` – Query for checking the dictionary status. Optional parameter. Read more in the sectionUpdating dictionaries [#dicts-external_dicts_dict_lifetime].

MySQL can be connected on a local host via sockets. To do this, set `host` and `socket`.

Example of settings:

```
<source>
  <mysql>
      <host>localhost</host>
      <socket>/path/to/socket/file.sock</socket>
      <user>clickhouse</user>
      <password>qwerty</password>
      <db>db_name</db>
      <table>table_name</table>
      <where>id=10</where>
      <invalidate_query>SQL_QUERY</invalidate_query>
  </mysql>
</source>
```

[]

ClickHouse

Example of settings:

```
<source>
    <clickhouse>
        <host>example01-01-1</host>
        <port>9000</port>
        <user>default</user>
        <password></password>
        <db>default</db>
        <table>ids</table>
        <where>id=10</where>
    </clickhouse>
</source>
```

Setting fields:

- `host` – The ClickHouse host. If it is a local host, the query is processed without any network activity. To improve fault tolerance, you can create a Distributed [#table_engines-distributed] table and enter it in subsequent configurations.
- `port` – The port on the ClickHouse server.
- `user` – Name of the ClickHouse user.
- `password` – Password of the ClickHouse user.
- `db` – Name of the database.
- `table` – Name of the table.
- `where` – The selection criteria. May be omitted.
- `invalidate_query` – Query for checking the dictionary status. Optional parameter. Read more in the sectionUpdating dictionaries [#dicts-external_dicts_dict_lifetime].

[]

MongoDB

Example of settings:

```
<source>
    <mongodb>
        <host>localhost</host>
        <port>27017</port>
        <user></user>
        <password></password>
        <db>test</db>
        <collection>dictionary_source</collection>
    </mongodb>
</source>
```

Setting fields:

- `host` – The MongoDB host.
- `port` – The port on the MongoDB server.
- `user` – Name of the MongoDB user.
- `password` – Password of the MongoDB user.
- `db` – Name of the database.
- `collection` – Name of the collection.

[]

Dictionary Key and Fields

The `<structure>` clause describes the dictionary key and fields available for queries.

Overall structure:

```
<dictionary>
    <structure>
        <id>
            <name>Id</name>
        </id>

        <attribute>
            <!-- Attribute parameters -->
        </attribute>

        ...

    </structure>
</dictionary>
```

Columns are described in the structure:

- `<id>` - key column [#dicts-external_dicts_dict_structure-key].
- `<attribute>` - data column [#dicts-external_dicts_dict_structure-attributes]. There can be a large number of columns.

Key

ClickHouse supports the following types of keys:

- Numeric key. UInt64. Defined in the tag `<id>` .
- Composite key. Set of values of different types. Defined in the tag `<key>` .

A structure can contain either `<id>` or `<key>` .

> **Warning**
>
> The key doesn't need to be defined separately in attributes.

Numeric Key

Format: `UInt64` .

Configuration example:

```
<id>
    <name>Id</name>
</id>
```

Configuration fields:

- name – The name of the column with keys.

Composite Key

The key can be a `tuple` from any types of fields. The layout [#dicts-external_dicts_dict_layout] in this case must be `complex_key_hashed` or `complex_key_cache` .

Tip

A composite key can consist of a single element. This makes it possible to use a string as the key, for instance.

The key structure is set in the element `<key>`. Key fields are specified in the same format as the dictionaryattributes [#dicts-external_dicts_dict_structure-attributes]. Example:

```
<structure>
    <key>
        <attribute>
            <name>field1</name>
            <type>String</type>
        </attribute>
        <attribute>
            <name>field2</name>
            <type>UInt32</type>
        </attribute>
        ...
    </key>
...
```

For a query to the `dictGet*` function, a tuple is passed as the key. Example: `dictGetString('dict_name', 'attr_name', tuple('string for field1', num_for_field2))`.

[]

Attributes

Configuration example:

```
<structure>
    ...
    <attribute>
        <name>Name</name>
        <type>Type</type>
        <null_value></null_value>
        <expression>rand64()</expression>
        <hierarchical>true</hierarchical>
        <injective>true</injective>
        <is_object_id>true</is_object_id>
    </attribute>
</structure>
```

Configuration fields:

- `name` – The column name.
- `type` – The column type. Sets the method for interpreting data in the source. For example, for MySQL, the field might be `TEXT`, `VARCHAR`, or `BLOB` in the source table, but it can be uploaded as `String`.
- `null_value` – The default value for a non-existing element. In the example, it is an empty string.
- `expression` – The attribute can be an expression. The tag is not required.
- `hierarchical` – Hierarchical support. Mirrored to the parent identifier. By default, `false`.
- `injective` – Whether the `id -> attribute` image is injective. If `true`, then you can optimize the `GROUP BY` clause. By default, `false`.
- `is_object_id` – Whether the query is executed for a MongoDB document by `ObjectID`.

[]

Internal dictionaries

ClickHouse contains a built-in feature for working with a geobase.

This allows you to:

- Use a region's ID to get its name in the desired language.
- Use a region's ID to get the ID of a city, area, federal district, country, or continent.
- Check whether a region is part of another region.
- Get a chain of parent regions.

All the functions support "translocality," the ability to simultaneously use different perspectives on region ownership. For more information, see the section "Functions for working with Yandex.Metrica dictionaries".

The internal dictionaries are disabled in the default package. To enable them, uncomment the parameters `path_to_regions_hierarchy_file` and `path_to_regions_names_files` in the server configuration file.

The geobase is loaded from text files.

Place the `regions_hierarchy*.txt` files into the `path_to_regions_hierarchy_file` directory. This configuration parameter must contain the path to the `regions_hierarchy.txt` file (the default regional hierarchy), and the other files (`regions_hierarchy_ua.txt`) must be located in the same directory.

Put the `regions_names_*.txt` files in the `path_to_regions_names_files` directory.

You can also create these files yourself. The file format is as follows:

`regions_hierarchy*.txt` : TabSeparated (no header), columns:

- region ID (`UInt32`)
- parent region ID (`UInt32`)
- region type (`UInt8`): 1 - continent, 3 - country, 4 - federal district, 5 - region, 6 - city; other types don't have values
- population (`UInt32`) — optional column

`regions_names_*.txt` : TabSeparated (no header), columns:

- region ID (`UInt32`)
- region name (`String`) — Can't contain tabs or line feeds, even escaped ones.

A flat array is used for storing in RAM. For this reason, IDs shouldn't be more than a million.

Dictionaries can be updated without restarting the server. However, the set of available dictionaries is not updated. For updates, the file modification times are checked. If a file has changed, the dictionary is updated. The interval to check for changes is configured in the `builtin_dictionaries_reload_interval` parameter. Dictionary updates (other than loading at first use) do not block queries. During updates, queries use the old versions of dictionaries. If an error occurs during an update, the error is written to the server log, and queries continue using the old version of dictionaries.

We recommend periodically updating the dictionaries with the geobase. During an update, generate new files and write them to a separate location. When everything is ready, rename them to the files used by the server.

There are also functions for working with OS identifiers and Yandex.Metrica search engines, but they shouldn't be used.

Operators

All operators are transformed to the corresponding functions at the query parsing stage, in accordance with their precedence and associativity. Groups of operators are listed in order of priority (the higher it is in the list, the earlier the operator is connected to its arguments).

Access Operators

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

`a.N` – Access to a tuble element; `tupleElement(a, N)` function.

Numeric Negation Operator

`-a` – The `negate (a)` function.

Multiplication and Division Operators

`a * b` – The `multiply (a, b) function.`

`a / b` – The `divide(a, b) function.`

`a % b` – The `modulo(a, b) function.`

Addition and Subtraction Operators

`a + b` – The `plus(a, b) function.`

`a - b` – The `minus(a, b) function.`

Comparison Operators

`a = b` – The `equals(a, b) function.`

`a == b` – The `equals(a, b) function.`

`a != b` – The `notEquals(a, b) function.`

`a <> b` – The `notEquals(a, b) function.`

`a <= b` – The `lessOrEquals(a, b) function.`

`a >= b` – The `greaterOrEquals(a, b) function.`

`a < b` – The `less(a, b) function.`

`a > b` – The `greater(a, b) function.`

`a LIKE s` – The `like(a, b) function.`

`a NOT LIKE s` – The `notLike(a, b) function.`

`a BETWEEN b AND c` – The same as `a >= b AND a <= c.`

Operators for Working With Data Sets

See the section "IN operators".

`a IN ...` – The `in(a, b) function`

`a NOT IN ...` – The `notIn(a, b) function.`

`a GLOBAL IN ...` – The `globalIn(a, b) function.`

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

Logical Negation Operator

`NOT a` The `not(a) function.`

Logical AND Operator

`a AND b` – The `and(a, b) function.`

Logical OR Operator

`a OR b` – The `or(a, b) function.`

Conditional Operator

`a ? b : c` – The `if(a, b, c) function.`

Note:

The conditional operator calculates the values of b and c, then checks whether condition a is met, and then returns the corresponding value. If `b` or `c` is an arrayJoin() [#functions_arrayjoin] function, each row will be replicated regardless of the "a" condition.

[] []

Conditional Expression

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

If `x` is specified, then `transform(x, [a, ...], [b, ...], c)` function is used. Otherwise – `multiIf(a, b, ..., c)` .

If there is no `ELSE c` clause in the expression, the default value is `NULL` .

The `transform` function does not work with `NULL` .

Concatenation Operator

`s1 || s2` – The `concat(s1, s2) function.`

Lambda Creation Operator

`x -> expr` – The `lambda(x, expr) function.`

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

Array Creation Operator

`[x1, ...]` – The `array(x1, ...) function.`

Tuple Creation Operator

`(x1, x2, ...)` – The `tuple(x2, x2, ...) function.`

Associativity

All binary operators have left associativity. For example, `1 + 2 + 3` is transformed to `plus(plus(1, 2), 3)`. Sometimes this doesn't work the way you expect. For example, `SELECT 4 > 2 > 3` will result in 0.

For efficiency, the `and` and `or` functions accept any number of arguments. The corresponding chains of `AND` and `OR` operators are transformed to a single call of these functions.

Checking for `NULL`

ClickHouse supports the `IS NULL` and `IS NOT NULL` operators.

[]

[] []

IS NULL

- For Nullable [#data_type-nullable] type values, the `IS NULL` operator returns:
  - `1`, if the value is `NULL`.
  - `0` otherwise.
- For other values, the `IS NULL` operator always returns `0`.

```
:) SELECT x+100 FROM t_null WHERE y IS NULL

SELECT x + 100
FROM t_null
WHERE isNull(y)

┌─plus(x, 100)─┐
│          101 │
└──────────────┘

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

[]

IS NOT NULL

- For Nullable [#data_type-nullable] type values, the `IS NOT NULL` operator returns:
  - `0`, if the value is `NULL`.
  - `1` otherwise.
- For other values, the `IS NOT NULL` operator always returns `1`.

```
:) SELECT * FROM t_null WHERE y IS NOT NULL

SELECT *
FROM t_null
WHERE isNotNull(y)

┌─x─┬─y─┐
│ 2 │ 3 │
└───┴───┘

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

Syntax

There are two types of parsers in the system: the full SQL parser (a recursive descent parser), and the data format parser (a fast stream parser). In all cases except the INSERT query, only the full SQL parser is used. The INSERT query uses both parsers:

```
INSERT INTO t VALUES (1, 'Hello, world'), (2, 'abc'), (3, 'def')
```

The `INSERT INTO t VALUES` fragment is parsed by the full parser, and the data `(1, 'Hello, world'), (2, 'abc'), (3, 'def')` is parsed by the fast stream parser. 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 feed, CR, and form feed.

Comments

SQL-style and C-style comments are supported. SQL-style comments: from `--` to the end of the line. The space after `--` can be omitted. Comments in C-style: from `/*` to `*/`. These comments can be multiline. Spaces are not required here, either.

Keywords

Keywords (such as `SELECT`) are not case-sensitive. Everything else (column names, functions, and so on), in contrast to standard SQL, is case-sensitive. Keywords are not reserved (they are just parsed as keywords in the corresponding context).

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 a corresponding special value: `\b` , `\f` , `\r` , `\n` , `\t` , `\0` , `\a` , `\v` , `\xHH` . In all other cases, escape sequences in the format `\c` , where "c" is any character, are converted to "c". This means that you can use the sequences `\'` and `\\` . The value will have the String type.

The minimum set of characters that you need to escape in string literals: `'` and `\` .

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 "Operators2". 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).

[]

NULL Literal

Indicates that the value is missing.

In order to store `NULL` in a table field, it must be of theNullable [#data_type-nullable] type.

Depending on the data format (input or output), `NULL` may have a different representation. For more information, see the documentation for data formats [#formats].

There are many nuances to processing `NULL` . For example, if at least one of the arguments of a comparison operation is `NULL` , the result of this operation will also be `NULL` . The same is true for multiplication, addition, and other operations. For more information, read the documentation for each operation.

In queries, you can check `NULL` using the IS NULL [#operator-is-null] and IS NOT NULL [#operator-is-not-null] operators and the related functions `isNull` and `isNotNull` .

Functions

Functions are written like an identifier with a list of arguments (possibly empty) in brackets. In contrast to standard SQL, the brackets are required, even for an empty arguments list. Example: `now()` . There are regular and aggregate functions (see the section "Aggregate functions"). Some aggregate functions can contain two lists of arguments in brackets. Example: `quantile (0.9) (x)` . These aggregate functions are called "parametric" functions, and the arguments in the first list are called "parameters". The syntax of aggregate functions without parameters is the same as for regular functions.

Operators

Operators are converted to their corresponding functions during query parsing, taking their priority and associativity into account. For example, the expression `1 + 2 * 3 + 4` is transformed to `plus(plus(1, multiply(2, 3)), 4)` . 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.

Operations

表引擎

表引擎（即表的类型）决定了：

- 数据的存储方式和位置，写到哪里以及从哪里读取数据
- 支持哪些查询以及如何支持。
- 并发数据访问。
- 索引的使用（如果存在）。
- 是否可以执行多线程请求。
- 数据复制参数。

在读取时，引擎只需要输出所请求的列，但在某些情况下，引擎可以在响应请求时部分处理数据。

对于大多数正式的任务，应该使用MergeTree族中的引擎。

[]

MergeTree

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

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

method is much more efficient than constantly rewriting data in the storage at the insert.

Main features:

- Stores data sorted by primary key.

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

- This allows you to use partitions if the partitioning key [#table_engines-custom_partitioning_key] is specified.

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

- Data replication support.

  The family of `ReplicatedMergeTree` tables is used for this. For more information, see the Data replication [#table_engines-replication] section.

- Data sampling support.

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

Info

The Merge [#table_engine-merge] engine does not belong to the `*MergeTree` family.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = MergeTree()
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

For a description of request parameters, seerequest description [#query_language-queries-create_table].

Query clauses

- `ENGINE` - Name and parameters of the engine. `ENGINE = MergeTree()`. `MergeTree` engine does not have parameters.

- `ORDER BY` — Primary key.

  A tuple of columns or arbitrary expressions. Example: `ORDER BY (CounterID, EventDate)`. If a sampling expression is used, the primary key must contain it. Example: `ORDER BY (CounterID, EventDate, intHash32(UserID))`.

- `PARTITION BY` — The partitioning key [#table_engines-custom_partitioning_key].

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

- `SAMPLE BY` — An expression for sampling. Example: `intHash32(UserID))`.

- `SETTINGS` — Additional parameters that control the behavior of the `MergeTree`:

  - `index_granularity` — The granularity of an index. The number of data rows between the "marks" of an index. By default, 8192.

Example of sections setting

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

In the example, we set partitioning by month.

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

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

Deprecated Method for Creating a Table

Attention

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

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

MergeTree() parameters

- `date-column` — The name of a column of the type Date [#data_type-date]. ClickHouse automatically creates partitions by month on the basis of this column. The partition names are in the `"YYYYMM"` format.
- `sampling_expression` — an expression for sampling.
- `(primary, key)` — primary key. Type — Tuple() [#data_type-tuple]. It may consist of arbitrary expressions, but it typically is a tuple of columns. It must include an expression for sampling if it is set. It must not include a column with a `date-column` date.
- `index_granularity` — The granularity of an index. The number of data rows between the "marks" of an index. The value 8192 is appropriate for most tasks.

Example

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

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

Data Storage

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

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

Data belonging to different partitions are separated into different parts. In the background, ClickHouse merges data parts for more efficient storage. Parts belonging to different partitions are not merged. The merge mechanism does not guarantee that all rows with the same primary key will be in the same data part.

For each data part, ClickHouse creates an index file that contains the primary key value for each index row ("mark"). Index row numbers are defined as `n * index_granularity`. The maximum value `n` is equal to the integer part of dividing the total number of rows by the `index_granularity`. For each column, the "marks" are also written for the same index rows as the primary key. These "marks" allow you to find the data directly in the columns.

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

Primary Keys and Indexes in Queries

Let's take the `(CounterID, Date)` primary key. In this case, the sorting and index can be illustrated as follows:

```
Whole data:     [-------------------------------------------------------------------------]
CounterID:      [aaaaaaaaaaaaaaaaaabbbbcdeeeeeeeeeeeeefgggggggghhhhhhhhhiiiiiiiiikllllllll]
Date:           [1111111222222233331233211111222222333211111112122222223111112223311122333]
Marks:           |      |      |      |      |      |      |      |      |      |      |
                a,1    a,2    a,3    b,3    e,2    e,3    g,1    h,2    i,1    i,3    l,3
Marks numbers:   0      1      2      3      4      5      6      7      8      9      10
```

If the data query specifies:

- `CounterID in ('a', 'h')`, the server reads the data in the ranges of marks `[0, 3)` and `[6, 8)`.
- `CounterID IN ('a', 'h') AND Date = 3`, the server reads the data in the ranges of marks `[1, 3)` and `[7, 8)`.
- `Date = 3`, the server reads the data in the range of marks `[1, 10]`.

The examples above show that it is always more effective to use an index than a full scan.

A sparse index allows extra strings to be read. When reading a single range of the primary key, up to `index_granularity * 2` extra rows in each data block can be read. In most cases, ClickHouse performance does not degrade when `index_granularity = 8192`.

Sparse indexes allow you to work with a very large number of table rows, because such indexes are always stored in the computer's RAM.

ClickHouse does not require a unique primary key. You can insert multiple rows with the same primary key.

Selecting the Primary Key

The number of columns in the primary key is not explicitly limited. Depending on the data structure, you can include more or fewer columns in the primary key. This may:

- Improve the performance of an index.

  If the primary key is `(a, b)`, then adding another column `c` will improve the performance if the following conditions are met: - There are queries with a condition on column `c`. - Long data ranges (several times longer than the `index_granularity`) with identical values for `(a, b)` are common. In other words, when adding another column allows you to skip quite long data ranges.

- Improve data compression.

  ClickHouse sorts data by primary key, so the higher the consistency, the better the compression.

- Provide additional logic when data parts merging in the CollapsingMergeTree [#table_engine-collapsingmergetree] and SummingMergeTree [#table_engine-summingmergetree] engines.

  You may need many fields in the primary key even if they are not necessary for the previous steps.

A long primary key will negatively affect the insert performance and memory consumption, but extra columns in the primary key do not affect ClickHouse performance during `SELECT` queries.

Use of Indexes and Partitions in Queries

For `SELECT` queries, ClickHouse analyzes whether an index can be used. An index can be used if the `WHERE/PREWHERE` clause has an expression (as one of the conjunction elements, or entirely) that represents an equality or inequality comparison operation, or if it has `IN` or `LIKE` with a fixed prefix on columns or expressions that are in the primary key or partitioning key, or on certain partially repetitive functions of these columns, or logical relationships of these expressions.

Thus, it is possible to quickly run queries on one or many ranges of the primary key. In this example, queries will be fast when run for a specific tracking tag; for a specific tag and date range; for a specific tag and date; for multiple tags with a date range, and so on.

Let's look at the engine configured as follows:

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

In this case, in queries:

```
SELECT count() FROM table WHERE EventDate = toDate(now()) AND CounterID = 34
SELECT count() FROM table WHERE EventDate = toDate(now()) AND (CounterID = 34 OR CounterID = 42)
SELECT count() FROM table WHERE ((EventDate >= toDate('2014-01-01') AND EventDate <= toDate('2014-01-31')) OR
EventDate = toDate('2014-05-01')) AND CounterID IN (101500, 731962, 160656) AND (CounterID = 101500 OR
EventDate != toDate('2014-05-01'))
```

ClickHouse will use the primary key index to trim improper data and the monthly partitioning key to trim partitions that are in improper date ranges.

The queries above show that the index is used even for complex expressions. Reading from the table is organized so that using the index can't be slower than a full scan.

In the example below, the index can't be used.

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

To check whether ClickHouse can use the index when running a query, use the settings force_index_by_date [#settings-settings-force_index_by_date] and force_primary_key [#settings-settings-force_primary_key].

The key for partitioning by month allows reading only those data blocks which contain dates from the proper range. In this case, the data block may contain data for many dates (up to an entire month). Within a block, data is sorted by primary key, which might not contain the date as the first column. Because of this, using a query with only a date condition that does not specify the primary key prefix will cause more data to be read than for a single date.

Concurrent Data Access

For concurrent table access, we use multi-versioning. In other words, when a table is simultaneously read and updated, data is read from a set of parts that is current at the time of the query. There are no lengthy locks. Inserts do not get in the way of read operations.

Reading from a table is automatically parallelized.

[]

Data Replication

Replication is only supported for tables in the MergeTree family:

- ReplicatedMergeTree
- ReplicatedSummingMergeTree
- ReplicatedReplacingMergeTree
- ReplicatedAggregatingMergeTree
- ReplicatedCollapsingMergeTree
- ReplicatedGraphiteMergeTree

Replication works at the level of an individual table, not the entire server. A server can store both replicated and non-replicated tables at the same time.

Replication does not depend on sharding. Each shard has its own independent replication.

Compressed data for `INSERT` and `ALTER` queries is replicated (for more information, see the documentation for ALTER [#query_language_queries_alter]).

`CREATE`, `DROP`, `ATTACH`, `DETACH` and `RENAME` queries are executed on a single server and are not replicated:

- `The CREATE TABLE` query creates a new replicatable table on the server where the query is run. If this table already exists on other servers, it adds a new replica.
- `The DROP TABLE` query deletes the replica located on the server where the query is run.
- `The RENAME` query renames the table on one of the replicas. In other words, replicated tables can have different names on different replicas.

To use replication, set the addresses of the ZooKeeper cluster in the config file. Example:

```
<zookeeper>
    <node index="1">
        <host>example1</host>
        <port>2181</port>
    </node>
    <node index="2">
        <host>example2</host>
        <port>2181</port>
    </node>
    <node index="3">
        <host>example3</host>
        <port>2181</port>
    </node>
</zookeeper>
```

Use ZooKeeper version 3.4.5 or later.

You can specify any existing ZooKeeper cluster and the system will use a directory on it for its own data (the directory is specified when creating a replicatable table).

If ZooKeeper isn't set in the config file, you can't create replicated tables, and any existing replicated tables will be read-only.

ZooKeeper is not used in `SELECT` queries because replication does not affect the performance of `SELECT` and queries run just as fast as they do for non-replicated tables. When querying distributed replicated tables, ClickHouse behavior is controlled by the settings max_replica_delay_for_distributed_queries [#settings_settings_max_replica_delay_for_distributed_queries] and fallback_to_stale_replicas_for_distributed_queries [#settings-settings-fallback_to_stale_replicas_for_distributed_queries].

For each `INSERT` query, approximately ten entries are added to ZooKeeper through several transactions. (To be more precise, this is for each inserted block of data; an INSERT query contains one block or one block per `max_insert_block_size = 1048576` rows.) This leads to slightly longer latencies for `INSERT` compared to non-replicated tables. But if you follow the recommendations to insert data in batches of no more than one `INSERT` per second, it doesn't create any problems. The entire ClickHouse cluster used for coordinating one ZooKeeper cluster has a total of several hundred `INSERTs` per second. The throughput on data inserts (the number of rows per second) is just as high as for non-replicated data.

For very large clusters, you can use different ZooKeeper clusters for different shards. However, this hasn't proven necessary on the Yandex.Metrica cluster (approximately 300 servers).

Replication is asynchronous and multi-master. `INSERT` queries (as well as `ALTER`) can be sent to any available server. Data is inserted on the server where the query is run, and then it is copied to the other servers. Because it is asynchronous, recently inserted data appears on the other replicas with some latency. If part of the replicas are not available, the data is written when they become available. If a replica is available, the latency is the amount of time it takes to transfer the block of compressed data over the network.

By default, an INSERT query waits for confirmation of writing the data from only one replica. If the data was successfully written to only one replica and the server with this replica ceases to exist, the stored data will be lost. Tp enable getting confirmation of data writes from multiple replicas, use the `insert_quorum` option.

Each block of data is written atomically. The INSERT query is divided into blocks up to `max_insert_block_size = 1048576` rows. In other words, if the `INSERT` query has less than 1048576 rows, it is made atomically.

Data blocks are deduplicated. For multiple writes of the same data block (data blocks of the same size containing the same rows in the same order), the block is only written once. The reason for this is in case of network failures when the client application doesn't know if the data was written to the DB, so the `INSERT` query can simply be repeated. It doesn't matter which replica INSERTs were sent to with identical data. `INSERTs` are idempotent. Deduplication parameters are controlled by merge_tree [#server_settings-merge_tree] server settings.

During replication, only the source data to insert is transferred over the network. Further data transformation (merging) is coordinated and performed on all the replicas in the same way. This minimizes network usage, which means that replication works well when replicas reside in different datacenters. (Note that duplicating data in different datacenters is the main goal of replication.)

You can have any number of replicas of the same data. Yandex.Metrica uses double replication in production. Each server uses RAID-5 or RAID-6, and RAID-10 in some cases. This is a relatively reliable and convenient solution.

The system monitors data synchronicity on replicas and is able to recover after a failure. Failover is automatic (for small differences in data) or semi-automatic (when data differs too much, which may indicate a configuration error).

[]

Creating Replicated Tables

The `Replicated` prefix is added to the table engine name. For example: `ReplicatedMergeTree`.

Replicated*MergeTree parameters

- `zoo_path` — The path to the table in ZooKeeper.
- `replica_name` — The replica name in ZooKeeper.

Example:

```
CREATE TABLE table_name
(
    EventDate DateTime,
    CounterID UInt32,
    UserID UInt32
) ENGINE = ReplicatedMergeTree('/clickhouse/tables/{layer}-{shard}/hits', '{replica}')
PARTITION BY toYYYYMM(EventDate)
ORDER BY (CounterID, EventDate, intHash32(UserID))
SAMPLE BY intHash32(UserID)
```

Example in deprecated syntax:

```
CREATE TABLE table_name
(
    EventDate DateTime,
    CounterID UInt32,
    UserID UInt32
) ENGINE = ReplicatedMergeTree('/clickhouse/tables/{layer}-{shard}/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 configuration file. Example:

```
<macros>
    <layer>05</layer>
    <shard>02</shard>
    <replica>example05-02-1.yandex.ru</replica>
</macros>
```

The path to the table in ZooKeeper should be unique for each replicated table. Tables on different shards should have different paths. In this case, the path consists of the following parts:

`/clickhouse/tables/` is the common prefix. We recommend using exactly this one.

`{layer}-{shard}` is the shard identifier. In this example it consists of two parts, since the Yandex.Metrica cluster uses bi-level sharding. For most tasks, you can leave just the {shard} substitution, which will be expanded to the shard identifier.

`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 the parameters explicitly instead of using substitutions. This might be convenient for testing and for configuring small clusters. However, you can't use distributed DDL queries (`ON CLUSTER`) in this case.

When working with large clusters, we recommend using substitutions because they reduce the probability of error.

Run the `CREATE TABLE` query on each replica. This query creates a new replicated table, or adds a new replica to an existing one.

If you add a new replica after the table already contains some data on other replicas, the data will be copied from the other replicas to the new one after running the query. In other words, the new replica syncs itself with the others.

To delete a replica, run `DROP TABLE` . However, only one replica is deleted – the one that resides on the server where you run the query.

Recovery After Failures

If ZooKeeper is unavailable when a server starts, replicated tables switch to read-only mode. The system periodically attempts to connect to ZooKeeper.

If ZooKeeper is unavailable during an `INSERT` , or an error occurs when interacting with ZooKeeper, an exception is thrown.

After connecting to ZooKeeper, the system checks whether the set of data in the local file system matches the expected set of data (ZooKeeper stores this information). If there are minor inconsistencies, the system resolves them by syncing data with the replicas.

If the system detects broken data parts (with the wrong size of files) or unrecognized parts (parts written to the file system but not recorded in ZooKeeper), it moves them to the 'detached' subdirectory (they are not deleted). Any missing parts are copied from the replicas.

Note that ClickHouse does not perform any destructive actions such as automatically deleting a large amount of data.

When the server starts (or establishes a new session with ZooKeeper), it only checks the quantity and sizes of all files. If the file sizes match but bytes have been changed somewhere in the middle, this is not detected immediately, but only when attempting to read the data for a `SELECT` query. The query throws an exception about a non-matching checksum or size of a compressed block. In this case, data parts are added to the verification queue and copied from the replicas if necessary.

If the local set of data differs too much from the expected one, a safety mechanism is triggered. The server enters this in the log and refuses to launch. The reason for this is that this case may indicate a configuration error, such as if a replica on a shard was accidentally configured like a replica on a different shard. However, the thresholds for this mechanism are set

fairly low, and this situation might occur during normal failure recovery. In this case, data is restored semi-automatically - by "pushing a button".

To start recovery, create the node `/path_to_table/replica_name/flags/force_restore_data` in ZooKeeper with any content, or run the command to restore all replicated tables:

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

Then restart the server. On start, the server deletes these flags and starts recovery.

Recovery After Complete Data Loss

If all data and metadata disappeared from one of the servers, follow these steps for recovery:

1. Install ClickHouse on the server. Define substitutions correctly in the config file that contains the shard identifier and replicas, if you use them.
2. If you had unreplicated tables that must be manually duplicated on the servers, copy their data from a replica (in the directory `/var/lib/clickhouse/data/db_name/table_name/`).
3. Copy table definitions located in `/var/lib/clickhouse/metadata/` from a replica. If a shard or replica identifier is defined explicitly in the table definitions, correct it so that it corresponds to this replica. (Alternatively, start the server and make all the `ATTACH TABLE` queries that should have been in the .sql files in `/var/lib/clickhouse/metadata/`.)
4. To start recovery, create the ZooKeeper node `/path_to_table/replica_name/flags/force_restore_data` with any content, or run the command to restore all replicated tables: `sudo -u clickhouse touch /var/lib/clickhouse/flags/force_restore_data`

Then start the server (restart, if it is already running). Data will be downloaded from replicas.

An alternative recovery option is to delete information about the lost replica from ZooKeeper (`/path_to_table/replica_name`), then create the replica again as described in "Creating replicatable tables [#table_engines-replication-creation_of_rep_tables]".

There is no restriction on network bandwidth during recovery. Keep this in mind if you are restoring many replicas at once.

[]

Converting from MergeTree to ReplicatedMergeTree

We use the term `MergeTree` to refer to all table engines in the `MergeTree family`, the same as for `ReplicatedMergeTree`.

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

Converting from ReplicatedMergeTree to MergeTree

Create a MergeTree table with a different name. Move all the data from the directory with the `ReplicatedMergeTree` table data to the new table's data directory. Then delete the `ReplicatedMergeTree` table and restart the server.

If you want to get rid of a `ReplicatedMergeTree` table without launching the server:

- Delete the corresponding `.sql` file in the metadata directory (`/var/lib/clickhouse/metadata/`).
- Delete the corresponding path in ZooKeeper (`/path_to_table/replica_name`).

After this, you can launch the server, create a `MergeTree` table, move the data to its directory, and then restart the server.

Recovery When Metadata in The ZooKeeper Cluster is Lost or Damaged

If the data in ZooKeeper was lost or damaged, you can save data by moving it to an unreplicated table as described above.

[]

Custom Partitioning Key

The partition key can be an expression from the table columns, or a tuple of such expressions (similar to the primary key). The partition key can be omitted. When creating a table, specify the partition key in the ENGINE description with the new syntax:

```
ENGINE [=] Name(...) [PARTITION BY expr] [ORDER BY expr] [SAMPLE BY expr] [SETTINGS name=value, ...]
```

For MergeTree tables, the partition expression is specified after `PARTITION BY` , the primary key after `ORDER BY` , the sampling key after `SAMPLE BY` , and `SETTINGS` can specify `index_granularity` (optional; the default value is 8192), as well as other settings from MergeTreeSettings.h [https://github.com/yandex/ClickHouse/blob/master/dbms/src/Storages/MergeTree/MergeTreeSettings.h]. The other engine parameters are specified in parentheses after the engine name, as previously. Example:

```
ENGINE = ReplicatedCollapsingMergeTree('/clickhouse/tables/name', 'replica1', Sign)
    PARTITION BY (toMonday(StartDate), EventType)
    ORDER BY (CounterID, StartDate, intHash32(UserID))
    SAMPLE BY intHash32(UserID)
```

The traditional partitioning by month is expressed as `toYYYYMM(date_column)` .

You can't convert an old-style table to a table with custom partitions (only via INSERT SELECT).

After this table is created, merge will only work for data parts that have the same value for the partitioning expression. Note: This means that you shouldn't make overly granular partitions (more than about a thousand partitions), or SELECT will perform poorly.

To specify a partition in ALTER PARTITION commands, specify the value of the partition expression (or a tuple). Constants and constant expressions are supported. Example:

```
ALTER TABLE table DROP PARTITION (toMonday(today()), 1)
```

Deletes the partition for the current week with event type 1. The same is true for the OPTIMIZE query. To specify the only partition in a non-partitioned table, specify `PARTITION tuple()` .

Note: For old-style tables, the partition can be specified either as a number `201710` or a string `'201710'` . The syntax for the new style of tables is stricter with types (similar to the parser for the VALUES input format). In addition, ALTER TABLE FREEZE PARTITION uses exact match for new-style tables (not prefix match).

In the `system.parts` table, the `partition` column specifies the value of the partition expression to use in ALTER queries (if quotas are removed). The `name` column should specify the name of the data part that has a new format.

Old: `20140317_20140323_2_2_0` (minimum date - maximum date - minimum block number - maximum block number - level).

Now: `201403_2_2_0` (partition ID - minimum block number - maximum block number - level).

The partition ID is its string identifier (human-readable, if possible) that is used for the names of data parts in the file system and in ZooKeeper. You can specify it in ALTER queries in place of the partition key. Example: Partition key `toYYYYMM(EventDate)`; ALTER can specify either `PARTITION 201710` or `PARTITION ID '201710'`.

For more examples, see the tests `00502_custom_partitioning_local` [https://github.com/yandex/ClickHouse/blob/master/dbms/tests/queries/0_stateless/00502_custom_partitioning_local.sql] and `00502_custom_partitioning_replicated_zookeeper` [https://github.com/yandex/ClickHouse/blob/master/dbms/tests/queries/0_stateless/00502_custom_partitioning_replicated_zookeeper.sql].

ReplacingMergeTree

The engine differs from MergeTree [#table_engines-mergetree] in that it removes duplicate entries with the same primary key value.

Data deduplication occurs only during a merge. Merging occurs in the background at an unknown time, so you can't plan for it. Some of the data may remain unprocessed. Although you can run an unscheduled merge using the `OPTIMIZE` query, don't count on using it, because the `OPTIMIZE` query will read and write a large amount of data.

Thus, `ReplacingMergeTree` is suitable for clearing out duplicate data in the background in order to save space, but it doesn't guarantee the absence of duplicates.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = ReplacingMergeTree([ver])
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

For a description of request parameters, seerequest description [#query_language-queries-create_table].

ReplacingMergeTree Parameters

- `ver` — column with version. Type `UInt*`, `Date` or `DateTime`. Optional parameter.

  When merging, `ReplacingMergeTree` from all the rows with the same primary key leaves only one: - Last in the selection, if `ver` not set. - With the maximum version, if `ver` specified.

Query clauses

When creating a `ReplacingMergeTree` table the same clauses [#table_engines-mergetree-configuring] are required, as when creating a `MergeTree` table.

Deprecated Method for Creating a Table

Attention

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

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

All of the parameters excepting `ver` have the same meaning as in `MergeTree`.

- `ver` - column with the version. Optional parameter. For a description, see the text above.

[]

SummingMergeTree

The engine inherits from MergeTree [#table_engines-mergetree]. The difference is that when merging data parts for `SummingMergeTree` tables ClickHouse replaces all the rows with the same primary key with one row which contains summarized values for the columns with the numeric data type. If the primary key is composed in a way that a single key value corresponds to large number of rows, this significantly reduces storage volume and speeds up data selection.

We recommend to use the engine together with `MergeTree`. Store complete data in `MergeTree` table, and use `SummingMergeTree` for aggregated data storing, for example, when preparing reports. Such an approach will prevent you from losing valuable data due to an incorrectly composed primary key.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = MergeTree()
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

For a description of request parameters, seerequest description [#query_language-queries-create_table].

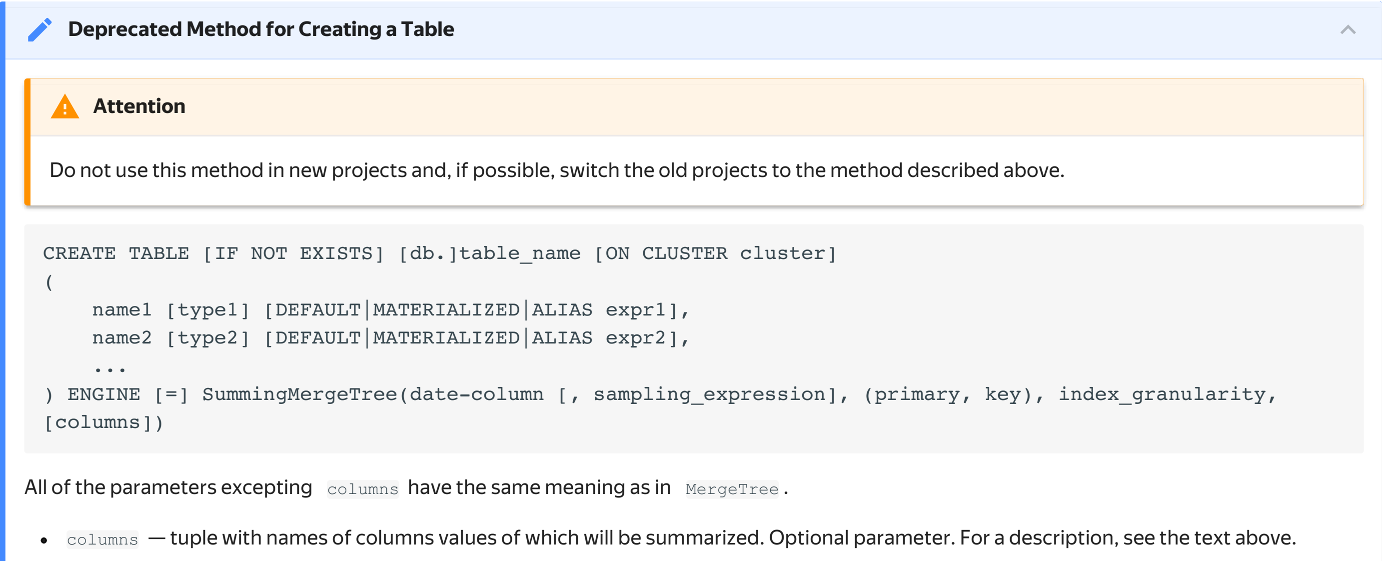
Parameters of SummingMergeTree

- `columns` - a tuple with the names of columns where values will be summarized. Optional parameter. The columns must be of a numeric type and must not be in the primary key.

  If `columns` not specified, ClickHouse summarizes the values in all columns with a numeric data type that are not in the primary key.

Query clauses

When creating a `SummingMergeTree` table the same clauses [#table_engines-mergetree-configuring] are required, as when creating a `MergeTree` table.



Deprecated Method for Creating a Table

Attention

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

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

All of the parameters excepting `columns` have the same meaning as in `MergeTree`.

- `columns` — tuple with names of columns values of which will be summarized. Optional parameter. For a description, see the text above.

Usage Example

Consider the following table:

```
CREATE TABLE summtt
(
    key UInt32,
    value UInt32
)
ENGINE = SummingMergeTree()
ORDER BY key
```

Insert data to it:

```
:) INSERT INTO summtt Values(1,1),(1,2),(2,1)
```

ClickHouse may sum all the rows not completely (see below [#summary-data-processing]), so we use an aggregate function `sum` and `GROUP BY` clause in the query.

```
SELECT key, sum(value) FROM summtt GROUP BY key
```

```
┌─key─┬─sum(value)─┐
│   2 │          1 │
│   1 │          3 │
└─────┴────────────┘
```

[]

Data Processing

When data are inserted into a table, they are saved as-is. Clickhouse merges the inserted parts of data periodically and this is when rows with the same primary key are summed and replaced with one for each resulting part of data.

ClickHouse can merge the data parts so that different resulting parts of data cat consist rows with the same primary key, i.e. the summation will be incomplete. Therefore (`SELECT`) an aggregate function sum() [#agg_function-sum] and `GROUP BY` clause should be used in a query as described in the example above.

Common rules for summation

The values in the columns with the numeric data type are summarized. The set of columns is defined by the parameter `columns`.

If the values were 0 in all of the columns for summation, the row is deleted.

If column is not in the primary key and is not summarized, an arbitrary value is selected from the existing ones.

The values are not summarized for columns in the primary key.

The Summation in the AggregateFunction Columns

For columns of AggregateFunction type [#data_type-aggregatefunction] ClickHouse behaves as AggregatingMergeTree [#table_engine-aggregatingmergetree] engine aggregating according to the function.

Nested Structures

Table can have nested data structures that are processed in a special way.

If the name of a nested table ends with `Map` and it contains at least two columns that meet the following criteria:

- the first column is numeric `(*Int*, Date, DateTime)` , let's call it `key` ,
- the other columns are arithmetic `(*Int*, Float32/64)` , let's call it `(values...)` ,

then this nested table is interpreted as a mapping of `key => (values...)` , and when merging its rows, the elements of two data sets are merged by `key` with a summation of the corresponding `(values...)` .

Examples:

```
[(1, 100)] + [(2, 150)] -> [(1, 100), (2, 150)]
[(1, 100)] + [(1, 150)] -> [(1, 250)]
[(1, 100)] + [(1, 150), (2, 150)] -> [(1, 250), (2, 150)]
[(1, 100), (2, 150)] + [(1, -100)] -> [(2, 150)]
```

When requesting data, use the sumMap(key, value) [#agg_function-summary] function for aggregation of `Map` .

For nested data structure, you do not need to specify its columns in the tuple of columns for summation.

[]

AggregatingMergeTree

The engine inherits from MergeTree [#table_engines-mergetree], altering the logic for data parts merging. ClickHouse replaces all rows with the same primary key with a single row (within a one data part) that stores a combination of states of aggregate functions.

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

The engine processes all columns with AggregateFunction [#data_type-aggregatefunction] type.

It is appropriate to use `AggregatingMergeTree` if it reduces the number of rows by orders.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = AggregatingMergeTree()
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

For a description of request parameters, seerequest description [#query_language-queries-create_table].

Query clauses

When creating a `ReplacingMergeTree` table the same clauses [#table_engines-mergetree-configuring] are required, as when creating a `MergeTree` table.

Deprecated Method for Creating a Table

Attention

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

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

All of the parameters have the same meaning as in `MergeTree`.

SELECT and INSERT

To insert data, use INSERT SELECT [#queries-insert-select] query with aggregate `-State` - functions.

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

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

Example of an Aggregated Materialized View

`AggregatingMergeTree` materialized view that watches the `test.visits` table:

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

Inserting of data into the `test.visits` table.

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

The data are inserted in both the table and view `test.basic` that will perform the aggregation.

To get the aggregated data, we need to execute a query such as `SELECT ... GROUP BY ...` from the view `test.basic`:

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

[]

CollapsingMergeTree

The engine inherits from MergeTree [#table_engines-mergetree] and adds the logic of rows collapsing to data parts merge algorithm.

`CollapsingMergeTree` asynchronously deletes (collapses) pairs of rows if all of the fields in a row are equivalent excepting the particular field `Sign` which can have `1` and `-1` values. Rows without a pair are kept. For more details see the Collapsing [#collapsingmergetree-collapsing] section of the document.

The engine may significantly reduce the volume of storage and increase efficiency of `SELECT` query as a consequence.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = CollapsingMergeTree(sign)
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

For a description of request parameters, seerequest description [#query_language-queries-create_table].

CollapsingMergeTree Parameters

- `sign` — Name of the column with the type of row: `1` is a "state" row, `-1` is a "cancel" row.

  Column data type — `Int8` .

Query clauses

When creating a `CollapsingMergeTree` table, the same clauses [#table_engines-mergetree-configuring] are required, as when creating a `MergeTree` table.

Deprecated Method for Creating a Table

Attention

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

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

All of the parameters excepting `sign` have the same meaning as in `MergeTree`.

- `sign` — Name of the column with the type of row: `1` — "state" row, `-1` — "cancel" row.

  Column Data Type — `Int8`.

[]

Collapsing

Data

Consider the situation where you need to save continually changing data for some object. It sounds logical to have one row for an object and update it at any change, but update operation is expensive and slow for DBMS because it requires rewriting of the data in the storage. If you need to write data quickly, update not acceptable, but you can write the changes of an object sequentially as follows.

Use the particular column `Sign` when writing row. If `Sign = 1` it means that the row is a state of an object, let's call it "state" row. If `Sign = -1` it means the cancellation of the state of an object with the same attributes, let's call it "cancel" row.

For example, we want to calculate how much pages users checked at some site and how long they were there. At some moment of time we write the following row with the state of user activity:

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │         5 │      146 │    1 │
└─────────────────────┴───────────┴──────────┴──────┘
```

At some moment later we register the change of user activity and write it with the following two rows.

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │         5 │      146 │   -1 │
│ 4324182021466249494 │         6 │      185 │    1 │
└─────────────────────┴───────────┴──────────┴──────┘
```

The first row cancels the previous state of the object (user). It should copy all of the fields of the canceled state excepting `Sign`.

The second row contains the current state.

As we need only the last state of user activity, the rows

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │         5 │      146 │    1 │
│ 4324182021466249494 │         5 │      146 │   -1 │
└─────────────────────┴───────────┴──────────┴──────┘
```

can be deleted collapsing the invalid (old) state of an object. `CollapsingMergeTree` does this while merging of the data parts.

Why we need 2 rows for each change read in the "Algorithm" paragraph.

Peculiar properties of such approach

1. The program that writes the data should remember the state of an object to be able to cancel it. "Cancel" string should be the copy of "state" string with the opposite `Sign`. It increases the initial size of storage but allows to write the data quickly.
2. Long growing arrays in columns reduce the efficiency of the engine due to load for writing. The more straightforward data, the higher efficiency.
3. `SELECT` results depend strongly on the consistency of object changes history. Be accurate when preparing data for inserting. You can get unpredictable results in inconsistent data, for example, negative values for non-negative metrics such as session depth.

Algorithm

When ClickHouse merges data parts, each group of consecutive rows with the same primary key is reduced to not more than two rows, one with `Sign = 1` ("state" row) and another with `Sign = -1` ("cancel" row). In other words, entries collapse.

For each resulting data part ClickHouse saves:

1. The first "cancel" and the last "state" rows, if the number of "state" and "cancel" rows matches.
2. The last "state" row, if there is one more "state" row than "cancel" rows.
3. The first "cancel" row, if there is one more "cancel" row than "state" rows.
4. None of the rows, in all other cases.

   The merge continues, but ClickHouse treats this situation as a logical error and records it in the server log. This error can occur if the same data were inserted more than once.

Thus, collapsing should not change the results of calculating statistics. Changes gradually collapsed so that in the end only the last state of almost every object left.

The `Sign` is required because the merging algorithm doesn't guarantee that all of the rows with the same primary key will be in the same resulting data part and even on the same physical server. ClickHouse process `SELECT` queries with multiple threads, and it can not predict the order of rows in the result. The aggregation is required if there is a need to get completely "collapsed" data from `CollapsingMergeTree` table.

To finalize collapsing write a query with `GROUP BY` clause and aggregate functions that account for the sign. For example, to calculate quantity, use `sum(Sign)` instead of `count()`. To calculate the sum of something, use `sum(Sign * x)` instead of `sum(x)`, and so on, and also add `HAVING sum(Sign) > 0`.

The aggregates `count`, `sum` and `avg` could be calculated this way. The aggregate `uniq` could be calculated if an object has at list one state not collapsed. The aggregates `min` and `max` could not be calculated because `CollapsingMergeTree` does not save values history of the collapsed states.

If you need to extract data without aggregation (for example, to check whether rows are present whose newest values match certain conditions), you can use the `FINAL` modifier for the `FROM` clause. This approach is significantly less efficient.

Example of use

Example data:

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │         5 │      146 │    1 │
│ 4324182021466249494 │         5 │      146 │   -1 │
│ 4324182021466249494 │         6 │      185 │    1 │
└─────────────────────┴───────────┴──────────┴──────┘
```

Creation of the table:

```
CREATE TABLE UAct
(
    UserID UInt64,
    PageViews UInt8,
    Duration UInt8,
    Sign Int8
)
ENGINE = CollapsingMergeTree(Sign)
ORDER BY UserID
```

Insertion of the data:

```
INSERT INTO UAct VALUES (4324182021466249494, 5, 146, 1)
```

```
INSERT INTO UAct VALUES (4324182021466249494, 5, 146, -1),(4324182021466249494, 6, 185, 1)
```

We use two `INSERT` queries to create two different data parts. If we insert the data with one query ClickHouse creates one data part and will not perform any merge ever.

Getting the data:

```
SELECT * FROM UAct
```

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │         5 │      146 │   -1 │
│ 4324182021466249494 │         6 │      185 │    1 │
└─────────────────────┴───────────┴──────────┴──────┘
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │         5 │      146 │    1 │
└─────────────────────┴───────────┴──────────┴──────┘
```

What do we see and where is collapsing? With two `INSERT` queries, we created 2 data parts. The `SELECT` query was performed in 2 threads, and we got a random order of rows. Collapsing not occurred because there was no merge of the data parts yet. ClickHouse merges data part in an unknown moment of time which we can not predict.

Thus we need aggregation:

```
SELECT
    UserID,
    sum(PageViews * Sign) AS PageViews,
    sum(Duration * Sign) AS Duration
FROM UAct
GROUP BY UserID
HAVING sum(Sign) > 0
```

```
┌──────────────UserID─┬─PageViews─┬─Duration─┐
│ 4324182021466249494 │         6 │      185 │
└─────────────────────┴───────────┴──────────┘
```

If we do not need aggregation and want to force collapsing, we can use `FINAL` modifier for `FROM` clause.

```
SELECT * FROM UAct FINAL
```

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │         6 │      185 │    1 │
└─────────────────────┴───────────┴──────────┴──────┘
```

This way of selecting the data is very inefficient. Don't use it for big tables.

[]

GraphiteMergeTree

This engine is designed for rollup (thinning and aggregating/averaging) Graphite [http://graphite.readthedocs.io/en/latest/index.html] data. It may be helpful to developers who want to use ClickHouse as a data store for Graphite.

You can use any ClickHouse table engine to store the Graphite data if you don't need rollup, but if you need a rollup use `GraphiteMergeTree`. The engine reduces the volume of storage and increases the efficiency of queries from Graphite.

The engine inherits properties from MergeTree [#table_engines-mergetree].

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    Path String,
    Time DateTime,
    Value <Numeric_type>,
    Version <Numeric_type>
    ...
) ENGINE = GraphiteMergeTree(config_section)
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

For a description of request parameters, seerequest description [#query_language-queries-create_table].

A table for the Graphite date should have the following columns:

- Column with the metric name (Graphite sensor). Data type: `String`.
- Column with the time for measuring the metric. Data type: `DateTime`.
- Column with the value of the metric. Data type: any numeric.
- Column with the version of the metric with the same name and time of measurement. Data type: any numeric.

  ClickHouse saves the rows with the highest version or the last written if versions are the same. Other rows are deleted during the merge of data parts.

The names of these columns should be set in the rollup configuration.

GraphiteMergeTree parameters

- `config_section` — Name of the section in the configuration file, where are the rules of rollup set.

Query clauses

When creating a `GraphiteMergeTree` table, the same clauses [#table_engines-mergetree-configuring] are required, as when creating a `MergeTree` table.

Deprecated Method for Creating a Table

Attention

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

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    EventDate Date,
    Path String,
    Time DateTime,
    Value <Numeric_type>,
    Version <Numeric_type>
    ...
) ENGINE [=] GraphiteMergeTree(date-column [, sampling_expression], (primary, key), index_granularity,
config_section)
```

All of the parameters excepting `config_section` have the same meaning as in `MergeTree`.

- `config_section` — Name of the section in the configuration file, where are the rules of rollup set.

Rollup configuration

The settings for rollup are defined by thegraphite_rollup [#server_settings-graphite_rollup] parameter in the server configuration. The name of the parameter could be any. You can create several configurations and use them for different tables.

Rollup configuration structure:

```
required-columns
pattern
    regexp
    function
    age + precision
    ...
pattern
    ...
default
    function
    age + precision
    ...
```

When processing a row, ClickHouse checks the rules in the `pattern` section. If the metric name matches the `regexp`, the rules from the `pattern` section are applied; otherwise, the rules from the `default` section are used.

The rules are defined with fields `function` and `age + precision`.

Fields for `pattern` and `default` sections:

- `regexp` – A pattern for the metric name.
- `age` – The minimum age of the data in seconds.
- `precision` – How precisely to define the age of the data in seconds.
- `function` – The name of the aggregating function to apply to data whose age falls within the range `[age, age + precision]`.

The `required-columns`:

- `path_column_name` — Column with the metric name (Graphite sensor).
- `time_column_name` — Column with the time for measuring the metric.
- `value_column_name` — Column with the value of the metric at the time set in `time_column_name`.

- `version_column_name` — Column with the version timestamp of the metric with the same name and time remains in the database.

Example of settings:

```
<graphite_rollup>
    <path_column_name>Path</path_column_name>
    <time_column_name>Time</time_column_name>
    <value_column_name>Value</value_column_name>
    <version_column_name>Version</version_column_name>
    <pattern>
        <regexp>click_cost</regexp>
        <function>any</function>
        <retention>
            <age>0</age>
            <precision>5</precision>
        </retention>
        <retention>
            <age>86400</age>
            <precision>60</precision>
        </retention>
    </pattern>
    <default>
        <function>max</function>
        <retention>
            <age>0</age>
            <precision>60</precision>
        </retention>
        <retention>
            <age>3600</age>
            <precision>300</precision>
        </retention>
        <retention>
            <age>86400</age>
            <precision>3600</precision>
        </retention>
    </default>
</graphite_rollup>
```

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.

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 that indicate where to start reading the file in order to skip the specified number of rows. This makes it possible to read table data in multiple threads. For concurrent data access, the read operations can be performed simultaneously, while write operations block reads and each other. The Log engine does not support indexes. Similarly, if writing to a table failed, the table is broken, and reading from it returns an error. The Log engine is appropriate for temporary data, write-once tables, and for testing or demonstration purposes.

Memory

The Memory engine stores data in RAM, in uncompressed form. Data is stored in exactly the same form as it is received when read. In other words, reading from this table is completely free. Concurrent data access is synchronized. Locks are short: read and write operations don't block each other. Indexes are not supported. Reading is parallelized. Maximal productivity (over 10 GB/sec) is reached on simple queries, because there is no reading from the disk, decompressing, or deserializing data. (We should note that in many cases, the productivity of the MergeTree engine is almost as high.) When restarting a server, data disappears from the table and the table becomes empty. Normally, using this table engine is not justified. However, it can be used for tests, and for tasks where maximum speed is required on a relatively small number of rows (up to approximately 100,000,000).

The Memory engine is used by the system for temporary tables with external query data (see the section "External data for processing a query"), and for implementing GLOBAL IN (see the section "IN operators").

Buffer

Buffers the data to write in RAM, periodically flushing it to another table. During the read operation, data is read from the buffer and the other table simultaneously.

```
Buffer(database, table, num_layers, min_time, max_time, min_rows, max_rows, min_bytes, max_bytes)
```

Engine parameters:database, table – The table to flush data to. Instead of the database name, you can use a constant expression that returns a string.num_layers – Parallelism layer. Physically, the table will be represented as 'num_layers' of independent buffers. Recommended value: 16.min_time, max_time, min_rows, max_rows, min_bytes, and max_bytes are conditions for flushing data from the buffer.

Data is flushed from the buffer and written to the destination table if all the 'min' conditions or at least one 'max' condition are met.min_time, max_time – Condition for the time in seconds from the moment of the first write to the buffer.min_rows, max_rows – Condition for the number of rows in the buffer.min_bytes, max_bytes – Condition for the number of bytes in the buffer.

During the write operation, data is inserted to a 'num_layers' number of random buffers. Or, if the data part to insert is large enough (greater than 'max_rows' or 'max_bytes'), it is written directly to the destination table, omitting the buffer.

The conditions for flushing the data are calculated separately for each of the 'num_layers' buffers. For example, if num_layers = 16 and max_bytes = 100000000, the maximum RAM consumption is 1.6 GB.

Example:

```
CREATE TABLE merge.hits_buffer AS merge.hits ENGINE = Buffer(merge, hits, 16, 10, 100, 10000, 1000000,
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 that it supports will be used.)

If the set of columns in the Buffer table doesn't match the set of columns in a subordinate table, a subset of columns that exist in both tables is inserted.

If the types don't match for one of the columns in the Buffer table and a subordinate table, an error message is entered in the server log and the buffer is cleared. The same thing happens if the subordinate table doesn't exist when the buffer is flushed.

If you need to run ALTER for a subordinate table and the Buffer table, we recommend first deleting the Buffer table, running ALTER for the subordinate table, then creating the Buffer table again.

If the server is restarted abnormally, the data in the buffer is lost.

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").

[]

External Data for Query Processing

ClickHouse allows sending a server the data that is needed for processing a query, together with a SELECT query. This data is put in a temporary table (see the section "Temporary tables") and can be used in the query (for example, in IN operators).

For example, if you have a text file with important user identifiers, you can upload it to the server along with a query that uses filtration by this list.

If you need to run more than one query with a large volume of external data, don't use this feature. It is better to upload the data to the DB ahead of time.

External data can be uploaded using the command-line client (in non-interactive mode), or using the HTTP interface.

In the command-line client, you can specify a parameters section in the format

```
--external --file=... [--name=...] [--format=...] [--types=...|--structure=...]
```

You may have multiple sections like this, for the number of tables being transmitted.

--external – Marks the beginning of a clause. **--file** – Path to the file with the table dump, or -, which refers to stdin. Only a single table can be retrieved from stdin.

The following parameters are optional: **--name**– Name of the table. If omitted, _data is used. **--format** – Data format in the file. If omitted, TabSeparated is used.

One of the following parameters is required:**--types** – A list of comma-separated column types. For example: `UInt64,String` . The columns will be named _1, _2, ...**--structure**– The table structure in the format `UserID UInt64` , `URL String` . Defines the column names and types.

The files specified in 'file' will be parsed by the format specified in 'format', using the data types specified in 'types' or 'structure'. The table will be uploaded to the server and accessible there as a temporary table with the name in 'name'.

Examples:

```
echo -ne "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' is passed the parameters 'name_format', 'name_types', and 'name_structure', where 'name' is the name of the table that these parameters correspond to. The meaning of the parameters is the same as when using the command-line client.

Example:

```
cat /etc/passwd | sed 's/:/\t/g' > passwd.tsv

curl -F 'passwd=@passwd.tsv;' 'http://localhost:8123/?
query=SELECT+shell,+count()+AS+c+FROM+passwd+GROUP+BY+shell+ORDER+BY+c+DESC&passwd_structure=login+String,+unuse

/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.

[]

Distributed

The Distributed engine does not store data itself, but allows distributed query processing on multiple servers. Reading is automatically parallelized. During a read, the table indexes on remote servers are used, if there are any. The Distributed

engine accepts parameters: the cluster name in the server's config file, the name of a remote database, the name of a remote table, and (optionally) a sharding key. Example:

```
Distributed(logs, default, hits[, sharding_key])
```

Data will be read from all servers in the 'logs' cluster, from the default.hits table located on every server in the cluster. Data is not only read, but is partially processed on the remote servers (to the extent that this is possible). For example, for a query with GROUP BY, data will be aggregated on remote servers, and the intermediate states of aggregate functions will be sent to the requestor server. Then data will be further aggregated.

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

logs – The cluster name in the server's config file.

Clusters are set like this:

```
<remote_servers>
    <logs>
        <shard>
            <!-- Optional. Shard weight when writing data. Default: 1. -->
            <weight>1</weight>
            <!-- Optional. Whether to write data to just one of the replicas. Default: false (write data to
all replicas). -->
            <internal_replication>false</internal_replication>
            <replica>
                <host>example01-01-1</host>
                <port>9000</port>
            </replica>
            <replica>
                <host>example01-01-2</host>
                <port>9000</port>
            </replica>
        </shard>
        <shard>
            <weight>2</weight>
            <internal_replication>false</internal_replication>
            <replica>
                <host>example01-02-1</host>
                <port>9000</port>
            </replica>
            <replica>
                <host>example01-02-2</host>
                <secure>1</secure>
                <port>9440</port>
            </replica>
        </shard>
    </logs>
</remote_servers>
```

Here a cluster is defined with the name 'logs' that consists of two shards, each of which contains two replicas. Shards refer to the servers that contain different parts of the data (in order to read all the data, you must access all the shards). Replicas are duplicating servers (in order to read all the data, you can access the data on any one of the replicas).

Cluster names must not contain dots.

The parameters `host` , `port` , and optionally `user` , `password` , `secure` , `compression` are specified for each server:

- `host` – The address of the remote server. You can use either the domain or the IPv4 or IPv6 address. If you specify the domain, the server makes a DNS request when it starts, and the result is stored as long as the server is running. If the DNS request fails, the server doesn't start. If you change the DNS record, restart the server.
- `port` – The TCP port for messenger activity ('tcp_port' in the config, usually set to 9000). Do not confuse it with http_port.
- `user` – Name of the user for connecting to a remote server. Default value: default. This user must have access to connect to the specified server. Access is configured in the users.xml file. For more information, see the section

"Access rights".

- `password` – The password for connecting to a remote server (not masked). Default value: empty string.
- `secure` - Use ssl for connection, usually you also should define `port` = 9440. Server should listen on 9440 and have correct certificates.
- `compression` - Use data compression. Default value: true.

When specifying replicas, one of the available replicas will be selected for each of the shards when reading. You can configure the algorithm for load balancing (the preference for which replica to access) – see the 'load_balancing' setting. If the connection with the server is not established, there will be an attempt to connect with a short timeout. If the connection failed, the next replica will be selected, and so on for all the replicas. If the connection attempt failed for all the replicas, the attempt will be repeated the same way, several times. This works in favor of resiliency, but does not provide complete fault tolerance: a remote server might accept the connection, but might not work, or work poorly.

You can specify just one of the shards (in this case, query processing should be called remote, rather than distributed) or up to any number of shards. In each shard, you can specify from one to any number of replicas. You can specify a different number of replicas for each shard.

You can specify as many clusters as you wish in the configuration.

To view your clusters, use the 'system.clusters' table.

The Distributed engine allows working with a cluster like a local server. However, the cluster is inextensible: you must write its configuration in the server config file (even better, for all the cluster's servers).

There is no support for Distributed tables that look at other Distributed tables (except in cases when a Distributed table only has one shard). As an alternative, make the Distributed table look at the "final" tables.

The Distributed engine requires writing clusters to the config file. Clusters from the config file are updated on the fly, without restarting the server. If you need to send a query to an unknown set of shards and replicas each time, you don't need to create a Distributed table – use the 'remote' table function instead. See the section "Table functions".

There are two methods for writing data to a cluster:

First, you can define which servers to write which data to, and perform the write directly on each shard. In other words, perform INSERT in the tables that the distributed table "looks at". This is the most flexible solution – you can use any sharding scheme, which could be non-trivial due to the requirements of the subject area. This is also the most optimal solution, since data can be written to different shards completely independently.

Second, you can perform INSERT in a Distributed table. In this case, the table will distribute the inserted data across servers itself. In order to write to a Distributed table, it must have a sharding key set (the last parameter). In addition, if there is only one shard, the write operation works without specifying the sharding key, since it doesn't have any meaning in this case.

Each shard can have a weight defined in the config file. By default, the weight is equal to one. Data is distributed across shards in the amount proportional to the shard weight. For example, if there are two shards and the first has a weight of 9 while the second has a weight of 10, the first will be sent 9 / 19 parts of the rows, and the second will be sent 10 / 19.

Each shard can have the 'internal_replication' parameter defined in the config file.

If this parameter is set to 'true', the write operation selects the first healthy replica and writes data to it. Use this alternative if the Distributed table "looks at" replicated tables. In other words, if the table where data will be written is going to replicate them itself.

If it is set to 'false' (the default), data is written to all replicas. In essence, this means that the Distributed table replicates data itself. This is worse than using replicated tables, because the consistency of replicas is not checked, and over time they will contain slightly different data.

To select the shard that a row of data is sent to, the sharding expression is analyzed, and its remainder is taken from

dividing it by the total weight of the shards. The row is sent to the shard that corresponds to the half-interval of the remainders from 'prev_weight' to 'prev_weights + weight', where 'prev_weights' is the total weight of the shards with the smallest number, and 'weight' is the weight of this shard. For example, if there are two shards, and the first has a weight of 9 while the second has a weight of 10, the row will be sent to the first shard for the remainders from the range [0, 9), and to the second for the remainders from the range [9, 19).

The sharding expression can be any expression from constants and table columns that returns an integer. For example, you can use the expression 'rand()' for random distribution of data, or 'UserID' for distribution by the remainder from dividing the user's ID (then the data of a single user will reside on a single shard, which simplifies running IN and JOIN by users). If one of the columns is not distributed evenly enough, you can wrap it in a hash function: intHash64(UserID).

A simple remainder from division is a limited solution for sharding and isn't always appropriate. It works for medium and large volumes of data (dozens of servers), but not for very large volumes of data (hundreds of servers or more). In the latter case, use the sharding scheme required by the subject area, rather than using entries in Distributed tables.

SELECT queries are sent to all the shards, and work regardless of how data is distributed across the shards (they can be distributed completely randomly). When you add a new shard, you don't have to transfer the old data to it. You can write new data with a heavier weight – the data will be distributed slightly unevenly, but queries will work correctly and efficiently.

You should be concerned about the sharding scheme in the following cases:

- Queries are used that require joining data (IN or JOIN) by a specific key. If data is sharded by this key, you can use local IN or JOIN instead of GLOBAL IN or GLOBAL JOIN, which is much more efficient.
- A large number of servers is used (hundreds or more) with a large number of small queries (queries of individual clients - websites, advertisers, or partners). In order for the small queries to not affect the entire cluster, it makes sense to locate data for a single client on a single shard. Alternatively, as we've done in Yandex.Metrica, you can set up bi-level sharding: divide the entire cluster into "layers", where a layer may consist of multiple shards. Data for a single client is located on a single layer, but shards can be added to a layer as necessary, and data is randomly distributed within them. Distributed tables are created for each layer, and a single shared distributed table is created for global queries.

Data is written asynchronously. For an INSERT to a Distributed table, the data block is just written to the local file system. The data is sent to the remote servers in the background as soon as possible. You should check whether data is sent successfully by checking the list of files (data waiting to be sent) in the table directory: /var/lib/clickhouse/data/database/table/.

If the server ceased to exist or had a rough restart (for example, after a device failure) after an INSERT to a Distributed table, the inserted data might be lost. If a damaged data part is detected in the table directory, it is transferred to the 'broken' subdirectory and no longer used.

When the max_parallel_replicas option is enabled, query processing is parallelized across all replicas within a single shard. For more information, see the section "Settings, max_parallel_replicas".

[]

Dictionary

The `Dictionary` engine displays the dictionary [#dicts-external_dicts] data as a ClickHouse table.

As an example, consider a dictionary of `products` with the following configuration:

```
<dictionaries>
<dictionary>
        <name>products</name>
        <source>
            <odbc>
                <table>products</table>
                <connection_string>DSN=some-db-server</connection_string>
            </odbc>
        </source>
        <lifetime>
            <min>300</min>
            <max>360</max>
        </lifetime>
        <layout>
            <flat/>
        </layout>
        <structure>
            <id>
                <name>product_id</name>
            </id>
            <attribute>
                <name>title</name>
                <type>String</type>
                <null_value></null_value>
            </attribute>
        </structure>
</dictionary>
</dictionaries>
```

Query the dictionary data:

```
select name, type, key, attribute.names, attribute.types, bytes_allocated, element_count,source from
system.dictionaries where name = 'products';

SELECT
    name,
    type,
    key,
    attribute.names,
    attribute.types,
    bytes_allocated,
    element_count,
    source
FROM system.dictionaries
WHERE name = 'products'
```

```
┌─name─────┬─type─┬─key────┬─attribute.names─┬─attribute.types─┬─bytes_allocated─┬─element_count─┬─source──────
│ products │ Flat │ UInt64 │ ['title']       │ ['String']      │        23065376 │        175032 │ ODBC:
.products │
└──────────┴──────┴────────┴─────────────────┴─────────────────┴─────────────────┴───────────────┴─────────────
```

You can use the dictGet* [#ext_dict_functions] function to get the dictionary data in this format.

This view isn't helpful when you need to get raw data, or when performing a `JOIN` operation. For these cases, you can use the `Dictionary` engine, which displays the dictionary data in a table.

Syntax:

```
CREATE TABLE %table_name% (%fields%) engine = Dictionary(%dictionary_name%)`
```

Usage example:

```
create table products (product_id UInt64, title String) Engine = Dictionary(products);

CREATE TABLE products
(
    product_id UInt64,
    title String,
)
ENGINE = Dictionary(products)
```

```
Ok.

0 rows in set. Elapsed: 0.004 sec.
```

Take a look at what's in the table.

```
select * from products limit 1;

SELECT *
FROM products
LIMIT 1
```

```
┌────product_id─┬─title───────────┐
│        152689 │ Some item       │
└───────────────┴─────────────────┘

1 rows in set. Elapsed: 0.006 sec.
```

Merge

The `Merge` engine (not to be confused with `MergeTree`) does not store data itself, but allows reading from any number of other tables simultaneously. Reading is automatically parallelized. Writing to a table is not supported. When reading, the indexes of tables that are actually being read are used, if they exist. The `Merge` engine accepts parameters: the database name and a regular expression for tables.

Example:

```
Merge(hits, '^WatchLog')
```

Data will be read from the tables in the `hits` database that have names that match the regular expression '`^WatchLog`'.

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

Regular expressions — re2 [https://github.com/google/re2] (supports a subset of PCRE), case-sensitive. See the notes about escaping symbols in regular expressions in the "match" section.

When selecting tables to read, the `Merge` table itself will not be selected, even if it matches the regex. This is to avoid loops. It is possible to create two `Merge` tables that will endlessly try to read each others' data, but this is not a good idea.

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

Example 2:

Let's say you have a old table (WatchLog_old) and decided to change partitioning without moving data to a new table (WatchLog_new) and you need to see data from both tables.

```
CREATE TABLE WatchLog_old(date Date, UserId Int64, EventType String, Cnt UInt64)
ENGINE=MergeTree(date, (UserId, EventType), 8192);
INSERT INTO WatchLog_old VALUES ('2018-01-01', 1, 'hit', 3);

CREATE TABLE WatchLog_new(date Date, UserId Int64, EventType String, Cnt UInt64)
ENGINE=MergeTree PARTITION BY date ORDER BY (UserId, EventType) SETTINGS index_granularity=8192;
INSERT INTO WatchLog_new VALUES ('2018-01-02', 2, 'hit', 3);

CREATE TABLE WatchLog as WatchLog_old ENGINE=Merge(currentDatabase(), '^WatchLog');

SELECT *
FROM WatchLog

┌───────date─┬─UserId─┬─EventType─┬─Cnt─┐
│ 2018-01-01 │      1 │ hit       │   3 │
└────────────┴────────┴───────────┴─────┘
┌───────date─┬─UserId─┬─EventType─┬─Cnt─┐
│ 2018-01-02 │      2 │ hit       │   3 │
└────────────┴────────┴───────────┴─────┘
```

Virtual Columns

Virtual columns are columns that are provided by the table engine, regardless of the table definition. In other words, these columns are not specified in `CREATE TABLE`, but they are accessible for `SELECT`.

Virtual columns differ from normal columns in the following ways:

- They are not specified in table definitions.
- Data can't be added to them with `INSERT`.
- When using `INSERT` without specifying the list of columns, virtual columns are ignored.
- They are not selected when using the asterisk (`SELECT *`).
- Virtual columns are not shown in `SHOW CREATE TABLE` and `DESC TABLE` queries.

The `Merge` type table contains a virtual `_table` column of the `String` type. (If the table already has a `_table` column, the virtual column is called `_table1`; if you already have `_table1`, it's called `_table2`, and so on.) It contains the name of the table that data was read from.

If the `WHERE/PREWHERE` clause contains conditions for the `_table` column that do not depend on other table columns (as one of the conjunction elements, or as an entire expression), these conditions are used as an index. The conditions are performed on a data set of table names to read data from, and the read operation will be performed from only those tables that the condition was triggered on.

[]

File(InputFormat)

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

Usage examples:

- Data export from ClickHouse to file.
- Convert data from one format to another.
- Updating data in ClickHouse via editing a file on a disk.

Usage in ClickHouse Server

```
File(Format)
```

`Format` should be supported for either `INSERT` and `SELECT`. For the full list of supported formats see Formats [#formats].

ClickHouse does not allow to specify filesystem path for `File`. It will use folder defined by path [#server_settings-path] setting in server configuration.

When creating table using `File(Format)` it creates empty subdirectory in that folder. When data is written to that table, it's put into `data.Format` file in that subdirectory.

You may manually create this subfolder and file in server filesystem and then ATTACH [#queries-attach] it to table information with matching name, so you can query data from that file.

Warning

Be careful with this funcionality, because ClickHouse does not keep track of external changes to such files. The result of simultaneous writes via ClickHouse and outside of ClickHouse is undefined.

Example:

1. Set up the `file_engine_table` table:

```
CREATE TABLE file_engine_table (name String, value UInt32) ENGINE=File(TabSeparated)
```

By default ClickHouse will create folder `/var/lib/clickhouse/data/default/file_engine_table`.

2. Manually create `/var/lib/clickhouse/data/default/file_engine_table/data.TabSeparated` containing:

```
$ cat data.TabSeparated
one 1
two 2
```

3. Query the data:

```
SELECT * FROM file_engine_table
```

```
┌─name─┬─value─┐
│ one  │     1 │
│ two  │     2 │
└──────┴───────┘
```

Usage in Clickhouse-local

In clickhouse-local [#utils-clickhouse-local] File engine accepts file path in addition to `Format`. Default input/output streams can be specified using numeric or human-readable names like `0` or `stdin`, `1` or `stdout`.

Example:

```
$ echo -e "1,2\n3,4" | clickhouse-local -q "CREATE TABLE table (a Int64, b Int64) ENGINE = File(CSV, stdin);
SELECT a, b FROM table; DROP TABLE table"
```

Details of Implementation

- Reads can be parallel, but not writes
- Not supported:
- `ALTER`
- `SELECT ... SAMPLE`
- Indices

- Replication

Null

When writing to a Null table, data is ignored. When reading from a Null table, the response is empty.

However, you can create a materialized view on a Null table. So the data written to the table will end up in the view.

Set

A data set that is always in RAM. It is intended for use on the right side of the IN operator (see the section "IN operators").

You can use INSERT to insert data in the table. New elements will be added to the data set, while duplicates will be ignored. But you can't perform SELECT from the table. The only way to retrieve data is by using it in the right half of the IN operator.

Data is always located in RAM. For INSERT, the blocks of inserted data are also written to the directory of tables on the disk. When starting the server, this data is loaded to RAM. In other words, after restarting, the data remains in place.

For a rough server restart, the block of data on the disk might be lost or damaged. In the latter case, you may need to manually delete the file with damaged data.

Join

A prepared data structure for JOIN that is always located in RAM.

```
Join(ANY|ALL, LEFT|INNER, k1[, k2, ...])
```

Engine parameters: `ANY|ALL` – strictness; `LEFT|INNER` – type. 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.

[]

URL(URL, Format)

Manages data on a remote HTTP/HTTPS server. This engine is similar to the `File` [#] engine.

Using the engine in the ClickHouse server

`The format` must be one that ClickHouse can use in `SELECT` queries and, if necessary, in `INSERTs`. For the full list of supported formats, see Formats [#formats].

`The URL` must conform to the structure of a Uniform Resource Locator. The specified URL must point to a server that uses HTTP or HTTPS. This does not require any additional headers for getting a response from the server.

`INSERT` and `SELECT` queries are transformed to `POST` and `GET` requests, respectively. For processing `POST` requests, the remote server must support Chunked transfer encoding [https://en.wikipedia.org/wiki/Chunked_transfer_encoding].

Example:

1. Create a `url_engine_table` table on the server :

```
CREATE TABLE url_engine_table (word String, value UInt64)
ENGINE=URL('http://127.0.0.1:12345/', CSV)
```

2. Create a basic HTTP server using the standard Python 3 tools and start it:

```
from http.server import BaseHTTPRequestHandler, HTTPServer

class CSVHTTPServer(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/csv')
        self.end_headers()

        self.wfile.write(bytes('Hello,1\nWorld,2\n', "utf-8"))

if __name__ == "__main__":
    server_address = ('127.0.0.1', 12345)
    HTTPServer(server_address, CSVHTTPServer).serve_forever()
```

```
python3 server.py
```

3. Request data:

```
SELECT * FROM url_engine_table
```

```
┌─word──┬─value─┐
│ Hello │     1 │
│ World │     2 │
└───────┴───────┘
```

Details of Implementation

- Reads and writes can be parallel
- Not supported:
  - `ALTER` and `SELECT...SAMPLE` operations.
  - Indexes.
  - Replication.

View

Used for implementing views (for more information, see the `CREATE VIEW query`). It does not store data, but only stores the specified `SELECT` query. When reading from a table, it runs this query (and deletes all unnecessary columns from the query).

MaterializedView

Used for implementing materialized views (for more information, see CREATE TABLE [#query_language-queries-create_table]). For storing data, it uses a different engine that was specified when creating the view. When reading from a table, it just uses this engine.

Kafka

This engine works with Apache Kafka [http://kafka.apache.org/].

Kafka lets you:

- Publish or subscribe to data flows.
- Organize fault-tolerant storage.
- Process streams as they become available.

Old format:

```
Kafka(kafka_broker_list, kafka_topic_list, kafka_group_name, kafka_format
      [, kafka_row_delimiter, kafka_schema, kafka_num_consumers])
```

New format:

```
Kafka SETTINGS
  kafka_broker_list = 'localhost:9092',
  kafka_topic_list = 'topic1,topic2',
  kafka_group_name = 'group1',
  kafka_format = 'JSONEachRow',
  kafka_row_delimiter = '\n'
  kafka_schema = '',
  kafka_num_consumers = 2
```

Required parameters:

- `kafka_broker_list` – A comma-separated list of brokers (`localhost:9092`).
- `kafka_topic_list` – A list of Kafka topics (`my_topic`).
- `kafka_group_name` – A group of Kafka consumers (`group1`). Reading margins are tracked for each group separately. If you don't want messages to be duplicated in the cluster, use the same group name everywhere.
- `kafka_format` – Message format. Uses the same notation as the SQL `FORMAT` function, such as `JSONEachRow` . For more information, see the "Formats" section.

Optional parameters:

- `kafka_row_delimiter` - Character-delimiter of records (rows), which ends the message.
- `kafka_schema` – An optional parameter that must be used if the format requires a schema definition. For example, Cap'n Proto [https://capnproto.org/] requires the path to the schema file and the name of the root `schema.capnp:Message` object.
- `kafka_num_consumers` – The number of consumers per table. Default: `1` . Specify more consumers if the throughput of one consumer is insufficient. The total number of consumers should not exceed the number of partitions in the topic, since only one consumer can be assigned per partition.

Examples:

```
CREATE TABLE queue (
  timestamp UInt64,
  level String,
  message String
) ENGINE = Kafka('localhost:9092', 'topic', 'group1', 'JSONEachRow');

SELECT * FROM queue LIMIT 5;

CREATE TABLE queue2 (
  timestamp UInt64,
  level String,
  message String
) ENGINE = Kafka SETTINGS kafka_broker_list = 'localhost:9092',
                          kafka_topic_list = 'topic',
                          kafka_group_name = 'group1',
                          kafka_format = 'JSONEachRow',
                          kafka_num_consumers = 4;

CREATE TABLE queue2 (
  timestamp UInt64,
  level String,
  message String
) ENGINE = Kafka('localhost:9092', 'topic', 'group1')
            SETTINGS kafka_format = 'JSONEachRow',
                     kafka_num_consumers = 4;
```

The delivered messages are tracked automatically, so each message in a group is only counted once. If you want to get the data twice, then create a copy of the table with another group name.

Groups are flexible and synced on the cluster. For instance, if you have 10 topics and 5 copies of a table in a cluster, then each copy gets 2 topics. If the number of copies changes, the topics are redistributed across the copies automatically. Read more about this at http://kafka.apache.org/intro [http://kafka.apache.org/intro].

`SELECT` is not particularly useful for reading messages (except for debugging), because each message can be read only once. It is more practical to create real-time threads using materialized views. To do this:

1. Use the engine to create a Kafka consumer and consider it a data stream.
2. Create a table with the desired structure.
3. Create a materialized view that converts data from the engine and puts it into a previously created table.

When the `MATERIALIZED VIEW` joins the engine, it starts collecting data in the background. This allows you to continually receive messages from Kafka and convert them to the required format using `SELECT`

Example:

```
CREATE TABLE queue (
  timestamp UInt64,
  level String,
  message String
) ENGINE = Kafka('localhost:9092', 'topic', 'group1', 'JSONEachRow');

CREATE TABLE daily (
  day Date,
  level String,
  total UInt64
) ENGINE = SummingMergeTree(day, (day, level), 8192);

CREATE MATERIALIZED VIEW consumer TO daily
  AS SELECT toDate(toDateTime(timestamp)) AS day, level, count() as total
  FROM queue GROUP BY day, level;

SELECT level, sum(total) FROM daily GROUP BY level;
```

To improve performance, received messages are grouped into blocks the size of max_insert_block_size [#settings-settings-max_insert_block_size]. If the block wasn't formed within stream_flush_interval_ms [#settings-

settings_stream_flush_interval_ms] milliseconds, the data will be flushed to the table regardless of the completeness of the block.

To stop receiving topic data or to change the conversion logic, detach the materialized view:

```
DETACH TABLE consumer;
ATTACH MATERIALIZED VIEW consumer;
```

If you want to change the target table by using `ALTER`, we recommend disabling the material view to avoid discrepancies between the target table and the data from the view.

Configuration

Similar to GraphiteMergeTree, the Kafka engine supports extended configuration using the ClickHouse config file. There are two configuration keys that you can use: global (`kafka`) and topic-level (`kafka_*`). The global configuration is applied first, and then the topic-level configuration is applied (if it exists).

```
<!--  Global configuration options for all tables of Kafka engine type -->
<kafka>
  <debug>cgrp</debug>
  <auto_offset_reset>smallest</auto_offset_reset>
</kafka>

<!-- Configuration specific for topic "logs" -->
<kafka_logs>
  <retry_backoff_ms>250</retry_backoff_ms>
  <fetch_min_bytes>100000</fetch_min_bytes>
</kafka_logs>
```

For a list of possible configuration options, see the librdkafka configuration reference [https://github.com/edenhill/librdkafka/blob/master/CONFIGURATION.md]. Use the underscore (`_`) instead of a dot in the ClickHouse configuration. For example, `check.crcs=true` will be `<check_crcs>true</check_crcs>`.

[]

MySQL

The MySQL engine allows you to perform `SELECT` queries on data that is stored on a remote MySQL server.

Call format:

```
MySQL('host:port', 'database', 'table', 'user', 'password'[, replace_query, 'on_duplicate_clause']);
```

Call parameters

- `host:port` — Address of the MySQL server.
- `database` — Database name on the MySQL server.
- `table` — Name of the table.
- `user` — The MySQL User.
- `password` — User password.
- `replace_query` — Flag that sets query substitution `INSERT INTO` to `REPLACE INTO`. If `replace_query=1`, the query is replaced.
- `on_duplicate_clause` — Adds the `ON DUPLICATE KEY on_duplicate_clause` expression to the `INSERT` query.

  Example: `INSERT INTO t (c1,c2) VALUES ('a', 2) ON DUPLICATE KEY UPDATE c2 = c2 + 1`, where `on_duplicate_clause` is `UPDATE c2 = c2 + 1`. See MySQL documentation to find which `on_duplicate_clause` you

can use with `ON DUPLICATE KEY` clause.

To specify `on_duplicate_clause` you need to pass `0` to the `replace_query` parameter. If you simultaneously pass `replace_query = 1` and `on_duplicate_clause`, ClickHouse generates an exception.

At this time, simple `WHERE` clauses such as `=, !=, >, >=, <, <=` are executed on the MySQL server.

The rest of the conditions and the `LIMIT` sampling constraint are executed in ClickHouse only after the query to MySQL finishes.

The `MySQL` engine does not support the Nullable [#data_type-nullable] data type, so when reading data from MySQL tables, `NULL` is converted to default values for the specified column type (usually 0 or an empty string).

Access Rights

Users and access rights are set up in the user config. This is usually `users.xml`.

Users are recorded in the `users` section. Here is a fragment of the `users.xml` file:

```
<!-- Users and ACL. -->
<users>
    <!-- If the user name is not specified, the 'default' user is used. -->
    <default>
        <!-- Password could be specified in plaintext or in SHA256 (in hex format).

             If you want to specify password in plaintext (not recommended), place it in 'password' element.
             Example: <password>qwerty</password>.
             Password could be empty.

             If you want to specify SHA256, place it in 'password_sha256_hex' element.
             Example:
<password_sha256_hex>65e84be33532fb784c48129675f9eff3a682b27168c0ea744b2cf58ee02337c5</password_sha256_hex>

             How to generate decent password:
             Execute: PASSWORD=$(base64 < /dev/urandom | head -c8); echo "$PASSWORD"; echo -n "$PASSWORD" |
sha256sum | tr -d '-'
             In first line will be password and in second - corresponding SHA256.
        -->
        <password></password>

        <!-- A list of networks that access is allowed from.
            Each list item has one of the following forms:
            <ip> The IP address or subnet mask. For example: 198.51.100.0/24 or 2001:DB8::/32.
            <host> Host name. For example: example01. A DNS query is made for verification, and all addresses
obtained are compared with the address of the customer.
            <host_regexp> Regular expression for host names. For example, ^example\d\d-\d\d-\d\.yandex\.ru$
                To check it, a DNS PTR request is made for the client's address and a regular expression 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 yourself, specify here:
                <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 requests from the Yandex.Metrica user interface via the API for data on specific counters. -->
    <web>
        <password></password>
        <networks incl="networks" />
        <profile>web</profile>
        <quota>default</quota>
        <allow_databases>
           <database>test</database>
        </allow_databases>
    </web>
```

You can see a declaration from two users: `default` and `web`. We added the `web` user separately.

The `default` user is chosen in cases when the username is not passed. The `default` user is also used for distributed query processing, if the configuration of the server or cluster doesn't specify the `user` and `password` (see the section on the Distributed [#table_engines-distributed] engine).

The user that is used for exchanging information between servers combined in a cluster must not have substantial restrictions or quotas – otherwise, distributed queries will fail.

The password is specified in clear text (not recommended) or in SHA-256. The hash isn't salted. In this regard, you should not consider these passwords as providing security against potential malicious attacks. Rather, they are necessary for protection from employees.

A list of networks is specified that access is allowed from. In this example, the list of networks for both users is loaded from a separate file (`/etc/metrika.xml`) containing the `networks` substitution. Here is a fragment of it:

```
<yandex>
    ...
    <networks>
        <ip>::/64</ip>
        <ip>203.0.113.0/24</ip>
        <ip>2001:DB8::/32</ip>
        ...
    </networks>
</yandex>
```

You could define this list of networks directly in `users.xml` , or in a file in the `users.d` directory (for more information, see the section "Configuration files [#configuration_files]").

The config includes comments explaining how to open access from everywhere.

For use in production, only specify `ip` elements (IP addresses and their masks), since using `host` and `hoost_regexp` might cause extra latency.

Next the user settings profile is specified (see the section "Settings profiles [#settings_profiles]"). You can specify the default profile, `default'` . The profile can have any name. You can specify the same profile for different users. The most important thing you can write in the settings profile is `readonly=1` , which ensures read-only access.

Then specify the quota to be used (see the section "Quotas [#quotas]"). You can specify the default quota: `default` . It is set in the config by default to only count resource usage, without restricting it. The quota can have any name. You can specify the same quota for different users – in this case, resource usage is calculated for each user individually.

In the optional `<allow_databases>` section, you can also specify a list of databases that the user can access. By default, all databases are available to the user. You can specify the `default` database. In this case, the user will receive access to the database by default.

Access to the `system` database is always allowed (since this database is used for processing queries).

The user can get a list of all databases and tables in them by using `SHOW` queries or system tables, even if access to individual databases isn't allowed.

Database access is not related to the readonly [#query_complexity_readonly] setting. You can't grant full access to one database and `readonly` access to another one.

[]

Configuration Files

The main server config file is `config.xml` . It resides in the `/etc/clickhouse-server/` directory.

Individual settings can be overridden in the `*.xml` and `*.conf` files in the `conf.d` and `config.d` directories next to the config file.

The `replace` or `remove` attributes can be specified for the elements of these config files.

If neither is specified, it combines the contents of elements recursively, replacing values of duplicate children.

If `replace` is specified, it replaces the entire element with the specified one.

If `remove` is specified, it deletes the element.

The config can also define "substitutions". If an element has the `incl` attribute, the corresponding substitution from the

file will be used as the value. By default, the path to the file with substitutions is `/etc/metrika.xml`. This can be changed in the include_from [#server_settings-include_from] element in the server config. The substitution values are specified in `/yandex/substitution_name` elements in this file. If a substitution specified in `incl` does not exist, it is recorded in the log. To prevent ClickHouse from logging missing substitutions, specify the `optional="true"` attribute (for example, settings for macros [#server_settings-macros]).

Substitutions can also be performed from ZooKeeper. To do this, specify the attribute `from_zk = "/path/to/node"`. The element value is replaced with the contents of the node at `/path/to/node` in ZooKeeper. You can also put an entire XML subtree on the ZooKeeper node and it will be fully inserted into the source element.

The `config.xml` file can specify a separate config with user settings, profiles, and quotas. The relative path to this config is set in the 'users_config' element. By default, it is `users.xml`. If `users_config` is omitted, the user settings, profiles, and quotas are specified directly in `config.xml`.

In addition, `users_config` may have overrides in files from the `users_config.d` directory (for example, `users.d`) and substitutions. For example, you can have separate config file for each user like this:

```
$ cat /etc/clickhouse-server/users.d/alice.xml
<yandex>
    <users>
      <alice>
          <profile>analytics</profile>
            <networks>
                  <ip>::/0</ip>
            </networks>
          <password_sha256_hex>...</password_sha256_hex>
          <quota>analytics</quota>
      </alice>
    </users>
</yandex>
```

For each config file, the server also generates `file-preprocessed.xml` files when starting. These files contain all the completed substitutions and overrides, and they are intended for informational use. If ZooKeeper substitutions were used in the config files but ZooKeeper is not available on the server start, the server loads the configuration from the preprocessed file.

The server tracks changes in config files, as well as files and ZooKeeper nodes that were used when performing substitutions and overrides, and reloads the settings for users and clusters on the fly. This means that you can modify the cluster, users, and their settings without restarting the server.

[]

Quotas

Quotas allow you to limit resource usage over a period of time, or simply track the use of resources. Quotas are set up in the user config. This is usually 'users.xml'.

The system also has a feature for limiting the complexity of a single query. See the section "Restrictions on query complexity").

In contrast to query complexity restrictions, quotas:

- Place restrictions on a set of queries that can be run over a period of time, instead of limiting a single query.
- Account for resources spent on all remote servers for distributed query processing.

Let's look at the section of the 'users.xml' file that defines quotas.

```xml
<!-- Quotas -->
<quotas>
    <!-- Quota name. -->
    <default>
        <!-- Restrictions for a time period. You can set many intervals with different restrictions. -->
        <interval>
            <!-- Length of the interval. -->
            <duration>3600</duration>

            <!-- Unlimited. Just collect data for the specified time interval. -->
            <queries>0</queries>
            <errors>0</errors>
            <result_rows>0</result_rows>
            <read_rows>0</read_rows>
            <execution_time>0</execution_time>
        </interval>
    </default>
```

By default, the quota just tracks resource consumption for each hour, without limiting usage. The resource consumption calculated for each interval is output to the server log after each request.

```xml
<statbox>
    <!-- Restrictions for a time period. You can set many intervals with different restrictions. -->
    <interval>
        <!-- Length of the interval. -->
        <duration>3600</duration>

        <queries>1000</queries>
        <errors>100</errors>
        <result_rows>1000000000</result_rows>
        <read_rows>100000000000</read_rows>
        <execution_time>900</execution_time>
    </interval>

    <interval>
        <duration>86400</duration>

        <queries>10000</queries>
        <errors>1000</errors>
        <result_rows>5000000000</result_rows>
        <read_rows>500000000000</read_rows>
        <execution_time>7200</execution_time>
    </interval>
</statbox>
```

For the 'statbox' quota, restrictions are set for every hour and for every 24 hours (86,400 seconds). The time interval is counted starting from an implementation-defined fixed moment in time. In other words, the 24-hour interval doesn't necessarily begin at midnight.

When the interval ends, all collected values are cleared. For the next hour, the quota calculation starts over.

Here are the amounts that can be restricted:

`queries` – The total number of requests.

`errors` – The number of queries that threw an exception.

`result_rows` – The total number of rows given as the result.

`read_rows` – The total number of source rows read from tables for running the query, on all remote servers.

`execution_time` – The total query execution time, in seconds (wall time).

If the limit is exceeded for at least one time interval, an exception is thrown with a text about which restriction was exceeded, for which interval, and when the new interval begins (when queries can be sent again).

Quotas can use the "quota key" feature in order to report on resources for multiple keys independently. Here is an example

of this:

```
<!-- For the global reports designer. -->
<web_global>
    <!-- keyed – The quota_key "key" is passed in the query parameter,
            and the quota is tracked separately for each key value.
        For example, you can pass a Yandex.Metrica username as the key,
            so the quota will be counted separately for each username.
        Using keys makes sense only if quota_key is transmitted by the program, not by a user.

        You can also write <keyed_by_ip /> so the IP address is used as the quota key.
        (But keep in mind that users can change the IPv6 address fairly easily.)
    -->
    <keyed />
```

The quota is assigned to users in the 'users' section of the config. See the section "Access rights".

For distributed query processing, the accumulated amounts are stored on the requestor server. So if the user goes to another server, the quota there will "start over".

When the server is restarted, quotas are reset.

System tables

System tables are used for implementing part of the system's functionality, and for providing access to information about how the system is working. You can't delete a system table (but you can perform DETACH). System tables don't have files with data on the disk or files with metadata. The server creates all the system tables when it starts. System tables are read-only. They are located in the 'system' database. []

system.asynchronous_metrics

Contain metrics used for profiling and monitoring. They usually reflect the number of events currently in the system, or the total resources consumed by the system. Example: The number of SELECT queries currently running; the amount of memory in use. `system.asynchronous_metrics` and `system.metrics` differ in their sets of metrics and how they are calculated.

system.clusters

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

```
cluster String      – The cluster name.
shard_num UInt32 – The shard number in the cluster, starting from 1.
shard_weight UInt32 – The relative weight of the shard when writing data.
replica_num UInt32 – The replica number in the shard, starting from 1.
host_name String – The host name, as specified in the config.
String host_address – The host IP address obtained from DNS.
port UInt16 – The port to use for connecting to the server.
user String – The name of the user for connecting to the server.
```

system.columns

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

```
database String — The name of the database the table is 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 configuration file that describes the dictionary.
- `attribute.names Array(String)` — Array of attribute names provided by the dictionary.
- `attribute.types Array(String)` — Corresponding array of attribute types that are provided by the dictionary.
- `has_hierarchy UInt8` — Whether the dictionary is hierarchical.
- `bytes_allocated UInt64` — The amount of RAM the dictionary uses.
- `hit_rate Float64` — For cache dictionaries, the percentage of uses for which the value was in the cache.
- `element_count UInt64` — The number of items stored in the dictionary.
- `load_factor Float64` — The percentage full of the dictionary (for a hashed dictionary, the percentage filled in the hash table).
- `creation_time DateTime` — The time when the dictionary was created or last successfully reloaded.
- `last_exception String` — Text of the error that occurs when creating or reloading the dictionary if the dictionary couldn't be created.
- `source String` — Text describing the data source for the dictionary.

Note that the amount of memory used by the dictionary is not proportional to the number of items stored in it. So for flat and cached dictionaries, all the memory cells are pre-assigned, regardless of how full the dictionary actually is. []

system.events

Contains information about the number of events that have occurred in the system. This is used for profiling and monitoring purposes. Example: The number of processed SELECT queries. Columns: 'event String' – the event name, and 'value UInt64' – the quantity.

system.functions

Contains information about normal and aggregate functions.

Columns:

- `name` (`String`) – The name of the function.

- `is_aggregate` (`UInt8`) — Whether the function is aggregate.

system.merges

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

Columns:

- `database String` — The name of the database the table is in.
- `table String` — Table name.
- `elapsed Float64` — The time elapsed (in seconds) since the merge started.
- `progress Float64` — The percentage of completed work from 0 to 1.
- `num_parts UInt64` — The number of pieces to be merged.
- `result_part_name String` — The name of the part that will be formed as the result of merging.
- `total_size_bytes_compressed UInt64` — The total size of the compressed data in the merged chunks.
- `total_size_marks UInt64` — The total number of marks in the merged partss.
- `bytes_read_uncompressed UInt64` — Number of bytes read, uncompressed.
- `rows_read UInt64` — Number of rows read.
- `bytes_written_uncompressed UInt64` — Number of bytes written, uncompressed.
- `rows_written UInt64` — Number of lines rows written. []

system.metrics

system.numbers

This table contains a single UInt64 column named 'number' that contains almost all the natural numbers starting from zero. You can use this table for tests, or if you need to do a brute force search. Reads from this table are not parallelized.

system.numbers_mt

The same as 'system.numbers' but reads are parallelized. The numbers can be returned in any order. Used for tests.

system.one

This table contains a single row with a single 'dummy' UInt8 column containing the value 0. This table is used if a SELECT query doesn't specify the FROM clause. This is similar to the DUAL table found in other DBMSs.

system.parts

Contains information about parts of MergeTree [#table_engines-mergetree] tables.

Each row describes one part of the data.

Columns:

- partition (String) – The partition name. To learn what a partition is, see the description of theALTER [#query_language_queries_alter] query.

Formats: - `YYYYMM` for automatic partitioning by month. - `any_string` when partitioning manually.

- name (String) – Name of the data part.

- active (UInt8) – Indicates whether the part is active. If a part is active, it is used in a table; otherwise, it will be deleted. Inactive data parts remain after merging.
- marks (UInt64) – The number of marks. To get the approximate number of rows in a data part, multiply `marks` by the index granularity (usually 8192).
- marks_size (UInt64) – The size of the file with marks.
- rows (UInt64) – The number of rows.
- bytes (UInt64) – The number of bytes when compressed.
- modification_time (DateTime) – The modification time of the directory with the data part. This usually corresponds to the time of data part creation.|
- remove_time (DateTime) – The time when the data part became inactive.
- refcount (UInt32) – The number of places where the data part is used. A value greater than 2 indicates that the data part is used in queries or merges.
- min_date (Date) – The minimum value of the date key in the data part.
- max_date (Date) – The maximum value of the date key in the data part.
- min_block_number (UInt64) – The minimum number of data parts that make up the current part after merging.
- max_block_number (UInt64) – The maximum number of data parts that make up the current part after merging.
- level (UInt32) – Depth of the merge tree. If a merge was not performed, `level=0` .
- primary_key_bytes_in_memory (UInt64) – The amount of memory (in bytes) used by primary key values.
- primary_key_bytes_in_memory_allocated (UInt64) – The amount of memory (in bytes) reserved for primary key values.
- database (String) – Name of the database.
- table (String) – Name of the table.
- engine (String) – Name of the table engine without parameters.

system.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 at a time can be the leader. The leader is responsible for selecting background merges to
perform.
Note that writes can be performed to any replica that is available and has a session in ZK, regardless of
whether it is a leader.

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

is_session_expired: Whether the session with ZooKeeper 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 ZooKeeper.
replica_name:       Replica name in ZooKeeper. Different replicas of the same table have different names.
replica_path:      Path to replica data in ZooKeeper. The same as concatenating
'zookeeper_path/replicas/replica_path'.

columns_version:    Version number of the table structure.
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 zero 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 in the log of general activity that the replica copied to its
execution queue, 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 ZooKeeper (i.e., the number
of functioning replicas).
```

If you request all the columns, the table may work a bit slowly, since several reads from ZooKeeper are made for each row.
If you don't request the last 4 columns (log_max_index, log_pointer, total_replicas, active_replicas), the table works quickly.

For example, you can check that everything is working correctly like this:

```
SELECT
    database,
    table,
    is_leader,
    is_readonly,
    is_session_expired,
    future_parts,
    parts_to_check,
    columns_version,
    queue_size,
    inserts_in_queue,
    merges_in_queue,
    log_max_index,
    log_pointer,
    total_replicas,
    active_replicas
FROM system.replicas
WHERE
       is_readonly
    OR is_session_expired
    OR future_parts > 20
    OR parts_to_check > 10
    OR queue_size > 20
    OR inserts_in_queue > 10
    OR log_max_index - log_pointer > 10
    OR total_replicas < 2
    OR active_replicas < total_replicas
```

If this query doesn't return anything, it means that everything is fine.

system.settings

Contains information about settings that are currently in use. I.e. used for executing the query you are using to read from the system.settings table.

Columns:

```
name String  — Setting name.
value String  — Setting value.
changed UInt8 — Whether the setting was explicitly defined in the config or explicitly changed.
```

Example:

```
SELECT *
FROM system.settings
WHERE changed
```

```
┌─name───────────────────┬─value───────┬─changed─┐
│ max_threads            │ 8           │       1 │
│ use_uncompressed_cache │ 0           │       1 │
│ load_balancing         │ random      │       1 │
│ max_memory_usage       │ 10000000000 │       1 │
└────────────────────────┴─────────────┴─────────┘
```

system.tables

This table contains the String columns 'database', 'name', and 'engine'. The table also contains three virtual columns: metadata_modification_time (DateTime type), create_table_query, and engine_full (String type). Each table that the server knows about is entered in the 'system.tables' table. This system table is used for implementing SHOW TABLES queries.

system.zookeeper

The table does not exist if ZooKeeper is not configured. Allows reading data from the ZooKeeper cluster defined in the

config. The query must have a 'path' equality condition in the WHERE clause. This is the path in ZooKeeper for the children that you want to get data for.

The query `SELECT * FROM system.zookeeper WHERE path = '/clickhouse'` outputs data for all children on the `/clickhouse` node. To output data for all root nodes, write path = '/'. If the path specified in 'path' doesn't exist, an exception will be thrown.

Columns:

- `name String` — The name of the node.
- `path String` — The path to the node.
- `value String` — Node value.
- `dataLength Int32` — Size of the value.
- `numChildren Int32` — Number of descendants.
- `czxid Int64` — ID of the transaction that created the node.
- `mzxid Int64` — ID of the transaction that last changed the node.
- `pzxid Int64` — ID of the transaction that last deleted or added descendants.
- `ctime DateTime` — Time of node creation.
- `mtime DateTime` — Time of the last modification of the node.
- `version Int32` — Node version: the number of times the node was changed.
- `cversion Int32` — Number of added or removed descendants.
- `aversion Int32` — Number of changes to the ACL.
- `ephemeralOwner Int64` — For ephemeral nodes, the ID of hte 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
```

Usage Recommendations

CPU

The SSE 4.2 instruction set must be supported. Modern processors (since 2008) support it.

When choosing a processor, prefer a large number of cores and slightly slower clock rate over fewer cores and a higher clock rate. For example, 16 cores with 2600 MHz is better than 8 cores with 3600 MHz.

Hyper-threading

Don't disable hyper-threading. It helps for some queries, but not for others.

Turbo Boost

Turbo Boost is highly recommended. It significantly improves performance with a typical load. You can use `turbostat` to view the CPU's actual clock rate under a load.

CPU Scaling Governor

Always use the `performance` scaling governor. The `on-demand` scaling governor works much worse with constantly high demand.

```
sudo echo 'performance' | tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
```

CPU Limitations

Processors can overheat. Use `dmesg` to see if the CPU's clock rate was limited due to overheating. The restriction can also be set externally at the datacenter level. You can use `turbostat` to monitor it under a load.

RAM

For small amounts of data (up to \~200 GB compressed), it is best to use as much memory as the volume of data. For large amounts of data and when processing interactive (online) queries, you should use a reasonable amount of RAM (128 GB or more) so the hot data subset will fit in the cache of pages. Even for data volumes of \~50 TB per server, using 128 GB of RAM significantly improves query performance compared to 64 GB.

Do not disable overcommit. The value `cat /proc/sys/vm/overcommit_memory` should be 0 or 1. Run

```
echo 0 | sudo tee /proc/sys/vm/overcommit_memory
```

Swap File

Always disable the swap file. The only reason for not doing this is if you are using ClickHouse on your personal laptop.

Huge Pages

Always disable transparent huge pages. It interferes with memory allocators, which leads to significant performance degradation.

```
echo 'never' | sudo tee /sys/kernel/mm/transparent_hugepage/enabled
```

Use `perf top` to watch the time spent in the kernel for memory management. Permanent huge pages also do not need to be allocated.

Storage Subsystem

If your budget allows you to use SSD, use SSD. If not, use HDD. SATA HDDs 7200 RPM will do.

Give preference to a lot of servers with local hard drives over a smaller number of servers with attached disk shelves. But for storing archives with rare queries, shelves will work.

RAID

When using HDD, you can combine their RAID-10, RAID-5, RAID-6 or RAID-50. For Linux, software RAID is better (with `mdadm`). We don't recommend using LVM. When creating RAID-10, select the `far` layout. If your budget allows, choose RAID-10.

If you have more than 4 disks, use RAID-6 (preferred) or RAID-50, instead of RAID-5. When using RAID-5, RAID-6 or RAID-50, always increase stripe_cache_size, since the default value is usually not the best choice.

```
echo 4096 | sudo tee /sys/block/md2/md/stripe_cache_size
```

Calculate the exact number from the number of devices and the block size, using the formula: `2 * num_devices * chunk_size_in_bytes / 4096`.

A block size of 1024 KB is sufficient for all RAID configurations. Never set the block size too small or too large.

You can use RAID-0 on SSD. Regardless of RAID use, always use replication for data security.

Enable NCQ with a long queue. For HDD, choose the CFQ scheduler, and for SSD, choose noop. Don't reduce the 'readahead' setting. For HDD, enable the write cache.

File System

Ext4 is the most reliable option. Set the mount options `noatime, nobarrier`. XFS is also suitable, but it hasn't been as thoroughly tested with ClickHouse. Most other file systems should also work fine. File systems with delayed allocation work better.

Linux Kernel

Don't use an outdated Linux kernel.

Network

If you are using IPv6, increase the size of the route cache. The Linux kernel prior to 3.2 had a multitude of problems with IPv6 implementation.

Use at least a 10 GB network, if possible. 1 Gb will also work, but it will be much worse for patching replicas with tens of terabytes of data, or for processing distributed queries with a large amount of intermediate data.

ZooKeeper

You are probably already using ZooKeeper for other purposes. You can use the same installation of ZooKeeper, if it isn't already overloaded.

It's best to use a fresh version of ZooKeeper – 3.4.9 or later. The version in stable Linux distributions may be outdated.

You should never use manually written scripts to transfer data between different ZooKeeper clusters, because the result will be incorrect for sequential nodes. Never use the "zkcopy" utility for the same reason: https://github.com/ksprojects/zkcopy/issues/15

If you want to divide an existing ZooKeeper cluster into two, the correct way is to increase the number of its replicas and then reconfigure it as two independent clusters.

Do not run ZooKeeper on the same servers as ClickHouse. Because ZooKeeper is very sensitive for latency and ClickHouse may utilize all available system resources.

With the default settings, ZooKeeper is a time bomb:

> The ZooKeeper server won't delete files from old snapshots and logs when using the default configuration (see autopurge), and this is the responsibility of the operator.

This bomb must be defused.

The ZooKeeper (3.5.1) configuration below is used in the Yandex.Metrica production environment as of May 20, 2017:

zoo.cfg:

```
## http://hadoop.apache.org/zookeeper/docs/current/zookeeperAdmin.html

## The number of milliseconds of each tick
tickTime=2000
## The number of ticks that the initial
## synchronization phase can take
initLimit=30000
## The number of ticks that can pass between
## sending a request and getting an acknowledgement
syncLimit=10

maxClientCnxns=2000

maxSessionTimeout=60000000
## the directory where the snapshot is stored.
dataDir=/opt/zookeeper/{{ cluster['name'] }}/data
## Place the dataLogDir to a separate physical disc for better performance
dataLogDir=/opt/zookeeper/{{ cluster['name'] }}/logs

autopurge.snapRetainCount=10
autopurge.purgeInterval=1


## To avoid seeks ZooKeeper allocates space in the transaction log file in
## blocks of preAllocSize kilobytes. The default block size is 64M. One reason
## for changing the size of the blocks is to reduce the block size if snapshots
## are taken more often. (Also, see snapCount).
preAllocSize=131072

## Clients can submit requests faster than ZooKeeper can process them,
## especially if there are a lot of clients. To prevent ZooKeeper from running
## out of memory due to queued requests, ZooKeeper will throttle clients so that
## there is no more than globalOutstandingLimit outstanding requests in the
## system. The default limit is 1,000.ZooKeeper logs transactions to a
## transaction log. After snapCount transactions are written to a log file a
## snapshot is started and a new transaction log file is started. The default
## snapCount is 10,000.
snapCount=3000000

## If this option is defined, requests will be will logged to a trace file named
## traceFile.year.month.day.
##traceFile=

## Leader accepts client connections. Default value is "yes". The leader machine
## coordinates updates. For higher update throughput at 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"
```

Salt init:

```
description "zookeeper-{{ cluster['name'] }} centralized coordination service"

start on runlevel [2345]
stop on runlevel [!2345]

respawn

limit nofile 8192 8192

pre-start script
    [ -r "/etc/zookeeper-{{ cluster['name'] }}/conf/environment" ] || exit 0
    . /etc/zookeeper-{{ cluster['name'] }}/conf/environment
    [ -d $ZOO_LOG_DIR ] || mkdir -p $ZOO_LOG_DIR
    chown $USER:$GROUP $ZOO_LOG_DIR
end script

script
    . /etc/zookeeper-{{ cluster['name'] }}/conf/environment
    [ -r /etc/default/zookeeper ] && . /etc/default/zookeeper
    if [ -z "$JMXDISABLE" ]; then
        JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote -
Dcom.sun.management.jmxremote.local.only=$JMXLOCALONLY"
    fi
    exec start-stop-daemon --start -c $USER --exec $JAVA --name zookeeper-{{ cluster['name'] }} \
        -- -cp $CLASSPATH $JAVA_OPTS -Dzookeeper.log.dir=${ZOO_LOG_DIR} \
        -Dzookeeper.root.logger=${ZOO_LOG4J_PROP} $ZOOMAIN $ZOOCFG
end script
```

[]

Server configuration parameters

This section contains descriptions of server settings that cannot be changed at the session or query level.

These settings are stored in the `config.xml` file on the ClickHouse server.

Other settings are described in the "Settings [#settings]" section.

Before studying the settings, read the Configuration files [#configuration_files] section and note the use of substitutions (the `incl` and `optional` attributes).

Server settings

[]

builtin_dictionaries_reload_interval

The interval in seconds before reloading built-in dictionaries.

ClickHouse reloads built-in dictionaries every x seconds. This makes it possible to edit dictionaries "on the fly" without restarting the server.

Default value: 3600.

Example

```
<builtin_dictionaries_reload_interval>3600</builtin_dictionaries_reload_interval>
```

[]

compression

Data compression settings.

Warning

Don't use it if you have just started using ClickHouse.

The configuration looks like this:

```
<compression>
    <case>
      <parameters/>
    </case>
    ...
</compression>
```

You can configure multiple sections `<case>`.

Block field `<case>`:

- `min_part_size` – The minimum size of a table part.
- `min_part_size_ratio` – The ratio of the minimum size of a table part to the full size of the table.
- `method` – Compression method. Acceptable values : `lz4` or `zstd` (experimental).

ClickHouse checks `min_part_size` and `min_part_size_ratio` and processes the `case` blocks that match these conditions. If none of the `<case>` matches, ClickHouse applies the `lz4` compression algorithm.

Example

```
<compression incl="clickhouse_compression">
    <case>
        <min_part_size>10000000000</min_part_size>
        <min_part_size_ratio>0.01</min_part_size_ratio>
        <method>zstd</method>
    </case>
</compression>
```

[]

default_database

The default database.

To get a list of databases, use the SHOW DATABASES [#query_language_queries_show_databases] query.

Example

```
<default_database>default</default_database>
```

[]

default_profile

Default settings profile.

Settings profiles are located in the file specified in the parameteruser_config [#server_settings-users_config].

Example

```
<default_profile>default</default_profile>
```

[]

dictionaries_config

The path to the config file for external dictionaries.

Path:

- Specify the absolute path or the path relative to the server config file.
- The path can contain wildcards * and ?.

See also "External dictionaries [#dicts-external_dicts]".

Example

```
<dictionaries_config>*_dictionary.xml</dictionaries_config>
```

[]

dictionaries_lazy_load

Lazy loading of dictionaries.

If `true` , then each dictionary is created on first use. If dictionary creation failed, the function that was using the dictionary throws an exception.

If `false` , all dictionaries are created when the server starts, and if there is an error, the server shuts down.

The default is `true` .

Example

```
<dictionaries_lazy_load>true</dictionaries_lazy_load>
```

[]

format_schema_path

The path to the directory with the schemes for the input data, such as schemas for theCapnProto [#format_capnproto] format.

Example

```
  <!-- Directory containing schema files for various input formats. -->
  <format_schema_path>format_schemas/</format_schema_path>
```

[]

graphite

Sending data to Graphite [https://github.com/graphite-project].

Settings:

- host – The Graphite server.

- port – The port on the Graphite server.
- interval – The interval for sending, in seconds.
- timeout – The timeout for sending data, in seconds.
- root_path – Prefix for keys.
- metrics – Sending data from a :ref: `system_tables-system.metrics` table.
- events – Sending data from a :ref: `system_tables-system.events` table.
- asynchronous_metrics – Sending data from a :ref: `system_tables-system.asynchronous_metrics` table.

You can configure multiple `<graphite>` clauses. For instance, you can use this for sending different data at different intervals.

Example

```
<graphite>
    <host>localhost</host>
    <port>42000</port>
    <timeout>0.1</timeout>
    <interval>60</interval>
    <root_path>one_min</root_path>
    <metrics>true</metrics>
    <events>true</events>
    <asynchronous_metrics>true</asynchronous_metrics>
</graphite>
```

[]

graphite_rollup

Settings for thinning data for Graphite.

For more details, see GraphiteMergeTree [#table_engines-graphitemergetree].

Example

```
<graphite_rollup_example>
    <default>
        <function>max</function>
        <retention>
            <age>0</age>
            <precision>60</precision>
        </retention>
        <retention>
            <age>3600</age>
            <precision>300</precision>
        </retention>
        <retention>
            <age>86400</age>
            <precision>3600</precision>
        </retention>
    </default>
</graphite_rollup_example>
```

[]

http_port/https_port

The port for connecting to the server over HTTP(s).

If `https_port` is specified, openSSL [#server_settings-openSSL] must be configured.

If `http_port` is specified, the openSSL configuration is ignored even if it is set.

Example

```
<https>0000</https>
```

[]

http_server_default_response

The page that is shown by default when you access the ClickHouse HTTP(s) server.

Example

Opens `https://tabix.io/` when accessing `http://localhost: http_port`.

```
<http_server_default_response>
  <![CDATA[<html ng-app="SMI2"><head><base href="http://ui.tabix.io/"></head><body><div ui-view=""
class="content-ui"></div><script src="http://loader.tabix.io/master.js"></script></body></html>]]>
</http_server_default_response>
```

[]

include_from

The path to the file with substitutions.

For more information, see the section "Configuration files [#configuration_files]".

Example

```
<include_from>/etc/metrica.xml</include_from>
```

[]

interserver_http_port

Port for exchanging data between ClickHouse servers.

Example

```
<interserver_http_port>9009</interserver_http_port>
```

[]

interserver_http_host

The host name that can be used by other servers to access this server.

If omitted, it is defined in the same way as the `hostname-f` command.

Useful for breaking away from a specific network interface.

Example

```
<interserver_http_host>example.yandex.ru</interserver_http_host>
```

[]

keep_alive_timeout

The number of seconds that ClickHouse waits for incoming requests before closing the connection. Defaults to 10 seconds

Example

```
<keep_alive_timeout>10</keep_alive_timeout>
```

[]

listen_host

Restriction on hosts that requests can come from. If you want the server to answer all of them, specify `::` .

Examples:

```
<listen_host>::1</listen_host>
<listen_host>127.0.0.1</listen_host>
```

[]

logger

Logging settings.

Keys:

- level – Logging level. Acceptable values: `trace` , `debug` , `information` , `warning` , `error` .
- log – The log file. Contains all the entries according to `level` .
- errorlog – Error log file.
- size – Size of the file. Applies to `log` and `errorlog` . Once the file reaches `size` , ClickHouse archives and renames it, and creates a new log file in its place.
- count – The number of archived log files that ClickHouse stores.

Example

```
<logger>
    <level>trace</level>
    <log>/var/log/clickhouse-server/clickhouse-server.log</log>
    <errorlog>/var/log/clickhouse-server/clickhouse-server.err.log</errorlog>
    <size>1000M</size>
    <count>10</count>
</logger>
```

Writing to the syslog is also supported. Config example:

```
<logger>
    <use_syslog>1</use_syslog>
    <syslog>
        <address>syslog.remote:10514</address>
        <hostname>myhost.local</hostname>
        <facility>LOG_LOCAL6</facility>
        <format>syslog</format>
    </syslog>
</logger>
```

Keys:

- user_syslog — Required setting if you want to write to the syslog.
- address — The host[:порт] of syslogd. If omitted, the local daemon is used.
- hostname — Optional. The name of the host that logs are sent from.
- facility — The syslog facility keyword [https://en.wikipedia.org/wiki/Syslog#Facility] in uppercase letters with the "LOG_" prefix: (`LOG_USER` , `LOG_DAEMON` , `LOG_LOCAL3` , and so on). Default value: `LOG_USER` if `address` is specified, `LOG_DAEMON otherwise.`
- format – Message format. Possible values: `bsd` and `syslog.`

[]

macros

Parameter substitutions for replicated tables.

Can be omitted if replicated tables are not used.

For more information, see the section "Creating replicated tables [#table_engines-replication-creation_of_rep_tables]".

Example

```
<macros incl="macros" optional="true" />
```

[]

mark_cache_size

Approximate size (in bytes) of the cache of "marks" used by MergeTree [#table_engines-mergetree].

The cache is shared for the server and memory is allocated as needed. The cache size must be at least 5368709120.

Example

```
<mark_cache_size>5368709120</mark_cache_size>
```

[]

max_concurrent_queries

The maximum number of simultaneously processed requests.

Example

```
<max_concurrent_queries>100</max_concurrent_queries>
```

[]

max_connections

The maximum number of inbound connections.

Example

```
<max_connections>4096</max_connections>
```

[]

max_open_files

The maximum number of open files.

By default: `maximum`.

We recommend using this option in Mac OS X, since the `getrlimit()` function returns an incorrect value.

Example

```
<max_open_files>262144</max_open_files>
```

[]

max_table_size_to_drop

Restriction on deleting tables.

If the size of a MergeTree [#table_engines-mergetree] table exceeds `max_table_size_to_drop` (in bytes), you can't delete it using a DROP query.

If you still need to delete the table without restarting the ClickHouse server, create the `<clickhouse-path>/flags/force_drop_table` file and run the DROP query.

Default value: 50 GB.

The value 0 means that you can delete all tables without any restrictions.

Example

```
<max_table_size_to_drop>0</max_table_size_to_drop>
```

[]

merge_tree

Fine tuning for tables in the MergeTree [#table_engines-mergetree].

For more information, see the MergeTreeSettings.h header file.

Example

```
<merge_tree>
    <max_suspicious_broken_parts>5</max_suspicious_broken_parts>
</merge_tree>
```

[]

openSSL

SSL client/server configuration.

Support for SSL is provided by the `libpoco` library. The interface is described in the fileSSLManager.h [https://github.com/ClickHouse-Extras/poco/blob/master/NetSSL_OpenSSL/include/Poco/Net/SSLManager.h]

Keys for server/client settings:

- privateKeyFile – The path to the file with the secret key of the PEM certificate. The file may contain a key and certificate

at the same time.

- certificateFile – The path to the client/server certificate file in PEM format. You can omit it if `privateKeyFile` contains the certificate.
- caConfig – The path to the file or directory that contains trusted root certificates.
- verificationMode – The method for checking the node's certificates. Details are in the description of the Context [https://github.com/ClickHouse-Extras/poco/blob/master/NetSSL_OpenSSL/include/Poco/Net/Context.h] class. Possible values: `none` , `relaxed` , `strict` , `once` .
- verificationDepth – The maximum length of the verification chain. Verification will fail if the certificate chain length exceeds the set value.
- loadDefaultCAFile – Indicates that built-in CA certificates for OpenSSL will be used. Acceptable values: `true` , `false` . |
- cipherList – Supported OpenSSL encryptions. For example: `ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH` .
- cacheSessions – Enables or disables caching sessions. Must be used in combination with `sessionIdContext` . Acceptable values: `true` , `false` .
- sessionIdContext – A unique set of random characters that the server appends to each generated identifier. The length of the string must not exceed `SSL_MAX_SSL_SESSION_ID_LENGTH` . This parameter is always recommended, since it helps avoid problems both if the server caches the session and if the client requested caching. Default value: `${application.name}` .
- sessionCacheSize – The maximum number of sessions that the server caches. Default value: 1024*20. 0 – Unlimited sessions.
- sessionTimeout – Time for caching the session on the server.
- extendedVerification – Automatically extended verification of certificates after the session ends. Acceptable values: `true` , `false` .
- requireTLSv1 – Require a TLSv1 connection. Acceptable values: `true` , `false` .
- requireTLSv1_1 – Require a TLSv1.1 connection. Acceptable values: `true` , `false` .
- requireTLSv1 – Require a TLSv1.2 connection. Acceptable values: `true` , `false` .
- fips – Activates OpenSSL FIPS mode. Supported if the library's OpenSSL version supports FIPS.
- privateKeyPassphraseHandler – Class (PrivateKeyPassphraseHandler subclass) that requests the passphrase for accessing the private key. For example: `<privateKeyPassphraseHandler>` , `<name>KeyFileHandler</name>` , `<options><password>test</password></options>` , `</privateKeyPassphraseHandler>` .
- invalidCertificateHandler – Class (subclass of CertificateHandler) for verifying invalid certificates. For example: `<invalidCertificateHandler> <name>ConsoleCertificateHandler</name> </invalidCertificateHandler>` .
- disableProtocols – Protocols that are not allowed to use.
- preferServerCiphers – Preferred server ciphers on the client.

Example of settings:

```
<openSSL>
    <server>
        <!-- openssl req -subj "/CN=localhost" -new -newkey rsa:2048 -days 365 -nodes -x509 -keyout
/etc/clickhouse-server/server.key -out /etc/clickhouse-server/server.crt -->
        <certificateFile>/etc/clickhouse-server/server.crt</certificateFile>
        <privateKeyFile>/etc/clickhouse-server/server.key</privateKeyFile>
        <!-- openssl dhparam -out /etc/clickhouse-server/dhparam.pem 4096 -->
        <dhParamsFile>/etc/clickhouse-server/dhparam.pem</dhParamsFile>
        <verificationMode>none</verificationMode>
        <loadDefaultCAFile>true</loadDefaultCAFile>
        <cacheSessions>true</cacheSessions>
        <disableProtocols>sslv2,sslv3</disableProtocols>
        <preferServerCiphers>true</preferServerCiphers>
    </server>
    <client>
        <loadDefaultCAFile>true</loadDefaultCAFile>
        <cacheSessions>true</cacheSessions>
        <disableProtocols>sslv2,sslv3</disableProtocols>
        <preferServerCiphers>true</preferServerCiphers>
        <!-- Use for self-signed: <verificationMode>none</verificationMode> -->
        <invalidCertificateHandler>
            <!-- Use for self-signed: <name>AcceptCertificateHandler</name> -->
            <name>RejectCertificateHandler</name>
        </invalidCertificateHandler>
    </client>
</openSSL>
```

[]

part_log

Logging events that are associated with MergeTree [#table_engines-mergetree]. For instance, adding or merging data. You can use the log to simulate merge algorithms and compare their characteristics. You can visualize the merge process.

Queries are logged in the ClickHouse table, not in a separate file.

Columns in the log:

- event_time – Date of the event.
- duration_ms – Duration of the event.
- event_type – Type of event. 1 – new data part; 2 – merge result; 3 – data part downloaded from replica; 4 – data part deleted.
- database_name – The name of the database.
- table_name – Name of the table.
- part_name – Name of the data part.
- partition_id – The identifier of the partition.
- size_in_bytes – Size of the data part in bytes.
- merged_from – An array of names of data parts that make up the merge (also used when downloading a merged part).
- merge_time_ms – Time spent on the merge.

Use the following parameters to configure logging:

- database – Name of the database.
- table – Name of the table.
- partition_by – Sets a custom partitioning key [#custom-partitioning-key].
- flush_interval_milliseconds – Interval for flushing data from the buffer in memory to the table.

Example

```
<part_log>
    <database>system</database>
    <table>part_log</table>
    <partition_by>toMonday(event_date)</partition_by>
    <flush_interval_milliseconds>7500</flush_interval_milliseconds>
</part_log>
```

[]

path

The path to the directory containing data.

Note

The trailing slash is mandatory.

Example

```
<path>/var/lib/clickhouse/</path>
```

[]

query_log

Setting for logging queries received with the log_queries=1 [#settings_settings-log_queries] setting.

Queries are logged in the ClickHouse table, not in a separate file.

Use the following parameters to configure logging:

- database – Name of the database.
- table – Name of the table.
- partition_by – Sets a custom partitioning key [#custom-partitioning-key].
- flush_interval_milliseconds – Interval for flushing data from the buffer in memory to the table.

If the table doesn't exist, ClickHouse will create it. If the structure of the query log changed when the ClickHouse server was updated, the table with the old structure is renamed, and a new table is created automatically.

Example

```
<query_log>
    <database>system</database>
    <table>query_log</table>
    <partition_by>toMonday(event_date)</partition_by>
    <flush_interval_milliseconds>7500</flush_interval_milliseconds>
</query_log>
```

[]

remote_servers

Configuration of clusters used by the Distributed table engine.

For more information, see the section "Table engines/Distributed [#table_engines-distributed]".

Example

```
<remote_servers incl="clickhouse_remote_servers" />
```

For the value of the `incl` attribute, see the section "Configuration files [#configuration_files]".

[]

timezone

The server's time zone.

Specified as an IANA identifier for the UTC time zone or geographic location (for example, Africa/Abidjan).

The time zone is necessary for conversions between String and DateTime formats when DateTime fields are output to text format (printed on the screen or in a file), and when getting DateTime from a string. In addition, the time zone is used in functions that work with the time and date if they didn't receive the time zone in the input parameters.

Example

```
<timezone>Europe/Moscow</timezone>
```

[]

tcp_port

Port for communicating with clients over the TCP protocol.

Example

```
<tcp_port>9000</tcp_port>
```

[]

tmp_path

Path to temporary data for processing large queries.

> **Note**
>
> The trailing slash is mandatory.

Example

```
<tmp_path>/var/lib/clickhouse/tmp/</tmp_path>
```

[]

uncompressed_cache_size

Cache size (in bytes) for uncompressed data used by table engines from the MergeTree [#table_engines-mergetree].

There is one shared cache for the server. Memory is allocated on demand. The cache is used if the option use_uncompressed_cache [#settings-use_uncompressed_cache] is enabled.

The uncompressed cache is advantageous for very short queries in individual cases.

Example

```
<uncompressed_cache_size>8589934592</uncompressed_cache_size>
```

user_files_path

The directory with user files. Used in the table function file() [#table_functions-file].

Example

```
<user_files_path>/var/lib/clickhouse/user_files/</user_files_path>
```

[]

users_config

Path to the file that contains:

- User configurations.
- Access rights.
- Settings profiles.
- Quota settings.

Example

```
<users_config>users.xml</users_config>
```

[]

zookeeper

Configuration of ZooKeeper servers.

ClickHouse uses ZooKeeper for storing replica metadata when using replicated tables.

This parameter can be omitted if replicated tables are not used.

For more information, see the section "Replication [#table_engines-replication]".

Example

```
<zookeeper incl="zookeeper-servers" optional="true" />
```

[]

Settings

There are multiple ways to make all the settings described below. Settings are configured in layers, so each subsequent layer redefines the previous settings.

Ways to configure settings, in order of priority:

- Settings in the `users.xml` server configuration file.

  Set in the element `<profiles>`.
- Session settings.

Send `SET setting=value` from the ClickHouse console client in interactive mode. Similarly, you can use ClickHouse sessions in the HTTP protocol. To do this, you need to specify the `session_id` HTTP parameter.

- Query settings.
  - When starting the ClickHouse console client in non-interactive mode, set the startup parameter `--setting=value`.
  - When using the HTTP API, pass CGI parameters (`URL?setting_1=value&setting_2=value...`).

Settings that can only be made in the server config file are not covered in this section.

[]

Permissions for queries

Queries in ClickHouse can be divided into several groups:

1. Read data queries: `SELECT`, `SHOW`, `DESCRIBE`, `EXISTS`.
2. Write data queries: `INSERT`, `OPTIMIZE`.
3. Change settings queries: `SET`, `USE`.
4. DDL [https://en.wikipedia.org/wiki/Data_definition_language] queries: `CREATE`, `ALTER`, `RENAME`, `ATTACH`, `DETACH`, `DROP` `TRUNCATE`.
5. Particular queries: `KILL QUERY`.

The following settings regulate user permissions for the groups of queries:

- readonly [#settings_readonly] — Restricts permissions for all groups of queries excepting DDL.
- allow_ddl [#settings_allow_ddl] — Restricts permissions for DDL queries.

`KILL QUERY` performs with any settings.

[]

readonly

Restricts permissions for read data, write data and change settings queries.

See above [#permissions_for_queries] for the division of queries into groups.

Possible values

- 0 — All queries are allowed. Default value.
- 1 — Read data queries only are allowed.
- 2 — Read data and change settings queries are allowed.

After setting `readonly = 1`, a user can't change `readonly` and `allow_ddl` settings in the current session.

When using the `GET` method in the HTTP interface [#http_interface], `readonly = 1` is set automatically. To modify data use the `POST` method.

[]

allow_ddl

Allows/denies DDL [https://en.wikipedia.org/wiki/Data_definition_language] queries.

See above [#permissions_for_queries] for the division of queries into groups.

Possible values

- 0 — DDL queries are not allowed.
- 1 — DDL queries are allowed. Default value.

You can not execute `SET allow_ddl = 1` if `allow_ddl = 0` for current session.

Restrictions on query complexity

Restrictions on query complexity are part of the settings. They are used in order to provide safer execution from the user interface. Almost all the restrictions only apply to `SELECT`. For distributed query processing, restrictions are applied on each server separately.

ClickHouse checks the restrictions for data parts, not for each row. It means that you can exceed the value of restriction with a size of the data part.

Restrictions on the "maximum amount of something" can take the value 0, which means "unrestricted". Most restrictions also have an 'overflow_mode' setting, meaning what to do when the limit is exceeded. It can take one of two values: `throw` or `break`. Restrictions on aggregation (group_by_overflow_mode) also have the value `any`.

`throw` – Throw an exception (default).

`break` – Stop executing the query and return the partial result, as if the source data ran out.

`any (only for group_by_overflow_mode)` – Continuing aggregation for the keys that got into the set, but don't add new keys to the set.

[]

max_memory_usage

The maximum amount of RAM to use for running a query on a single server.

In the default configuration file, the maximum is 10 GB.

The setting doesn't consider the volume of available memory or the total volume of memory on the machine. The restriction applies to a single query within a single server. You can use `SHOW PROCESSLIST` to see the current memory consumption for each query. In addition, the peak memory consumption is tracked for each query and written to the log.

Memory usage is not monitored for the states of certain aggregate functions.

Memory usage is not fully tracked for states of the aggregate functions `min`, `max`, `any`, `anyLast`, `argMin`, `argMax` from `String` and `Array` arguments.

Memory consumption is also restricted by the parameters `max_memory_usage_for_user` and `max_memory_usage_for_all_queries`.

max_memory_usage_for_user

The maximum amount of RAM to use for running a user's queries on a single server.

Default values are defined in Settings.h [https://github.com/yandex/ClickHouse/blob/master/dbms/src/Interpreters/Settings.h#L244]. By default, the amount is not restricted (`max_memory_usage_for_user = 0`).

See also the description of max_memory_usage [#settings_max_memory_usage].

max_memory_usage_for_all_queries

The maximum amount of RAM to use for running all queries on a single server.

Default values are defined in Settings.h [https://github.com/yandex/ClickHouse/blob/master/dbms/src/Interpreters/Settings.h#L245]. By default, the amount is not restricted (`max_memory_usage_for_all_queries = 0`).

See also the description of max_memory_usage [#settings_max_memory_usage].

max_rows_to_read

The following restrictions can be checked on each block (instead of on each row). That is, the restrictions can be broken a little. When running a query in multiple threads, the following restrictions apply to each thread separately.

Maximum number of rows that can be read from a table when running a query.

max_bytes_to_read

Maximum number of bytes (uncompressed data) that can be read from a table when running a query.

read_overflow_mode

What to do when the volume of data read exceeds one of the limits: 'throw' or 'break'. By default, throw.

max_rows_to_group_by

Maximum number of unique keys received from aggregation. This setting lets you limit memory consumption when aggregating.

group_by_overflow_mode

What to do when the number of unique keys for aggregation exceeds the limit: 'throw', 'break', or 'any'. By default, throw. Using the 'any' value lets you run an approximation of GROUP BY. The quality of this approximation depends on the statistical nature of the data.

max_rows_to_sort

Maximum number of rows before sorting. This allows you to limit memory consumption when sorting.

max_bytes_to_sort

Maximum number of bytes before sorting.

sort_overflow_mode

What to do if the number of rows received before sorting exceeds one of the limits: 'throw' or 'break'. By default, throw.

max_result_rows

Limit on the number of rows in the result. Also checked for subqueries, and on remote servers when running parts of a distributed query.

max_result_bytes

Limit on the number of bytes in the result. The same as the previous setting.

result_overflow_mode

What to do if the volume of the result exceeds one of the limits: 'throw' or 'break'. By default, throw. Using 'break' is similar to using LIMIT.

max_execution_time

Maximum query execution time in seconds. At this time, it is not checked for one of the sorting stages, or when merging and finalizing aggregate functions.

timeout_overflow_mode

What to do if the query is run longer than 'max_execution_time': 'throw' or 'break'. By default, throw.

min_execution_speed

Minimal execution speed in rows per second. Checked on every data block when 'timeout_before_checking_execution_speed' expires. If the execution speed is lower, an exception is thrown.

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.

Settings

[]

distributed_product_mode

Changes the behavior of distributed subqueries [#queries-distributed-subrequests].

ClickHouse applies this setting when the query contains the product of distributed tables, i.e. when the query for a distributed table contains a non-GLOBAL subquery for the distributed table.

Restrictions:

- Only applied for IN and JOIN subqueries.
- Only if the FROM section uses a distributed table containing more than one shard.
- If the subquery concerns a distributed table containing more than one shard,
- Not used for a table-valued remote [#table_functions-remote] function.

The possible values are:

- `deny` — Default value. Prohibits using these types of subqueries (returns the "Double-distributed in/JOIN subqueries is denied" exception).
- `local` — Replaces the database and table in the subquery with local ones for the destination server (shard), leaving the normal `IN` / `JOIN.`
- `global` — Replaces the `IN` / `JOIN` query with `GLOBAL IN` / `GLOBAL JOIN.`
- `allow` — Allows the use of these types of subqueries.

[]

fallback_to_stale_replicas_for_distributed_queries

Forces a query to an out-of-date replica if updated data is not available. See "Replication [#table_engines-replication]".

ClickHouse selects the most relevant from the outdated replicas of the table.

Used when performing `SELECT` from a distributed table that points to replicated tables.

By default, 1 (enabled).

[]

force_index_by_date

Disables query execution if the index can't be used by date.

Works with tables in the MergeTree family.

If `force_index_by_date=1` , ClickHouse checks whether the query has a date key condition that can be used for restricting data ranges. If there is no suitable condition, it throws an exception. However, it does not check whether the condition actually reduces the amount of data to read. For example, the condition `Date != ' 2000-01-01 '` is acceptable even when it matches all the data in the table (i.e., running the query requires a full scan). For more information about ranges of data in MergeTree tables, see "MergeTree [#table_engines-mergetree]".

[]

force_primary_key

Disables query execution if indexing by the primary key is not possible.

Works with tables in the MergeTree family.

If `force_primary_key=1`, ClickHouse checks to see if the query has a primary key condition that can be used for restricting data ranges. If there is no suitable condition, it throws an exception. However, it does not check whether the condition actually reduces the amount of data to read. For more information about data ranges in MergeTree tables, see "MergeTree [#table_engines-mergetree]".

[]

fsync_metadata

Enable or disable fsync when writing .sql files. Enabled by default.

It makes sense to disable it if the server has millions of tiny table chunks that are constantly being created and destroyed.

input_format_allow_errors_num

Sets the maximum number of acceptable errors when reading from text formats (CSV, TSV, etc.).

The default value is 0.

Always pair it with `input_format_allow_errors_ratio`. To skip errors, both settings must be greater than 0.

If an error occurred while reading rows but the error counter is still less than `input_format_allow_errors_num`, ClickHouse ignores the row and moves on to the next one.

If `input_format_allow_errors_num` is exceeded, ClickHouse throws an exception.

input_format_allow_errors_ratio

Sets the maximum percentage of errors allowed when reading from text formats (CSV, TSV, etc.). The percentage of errors is set as a floating-point number between 0 and 1.

The default value is 0.

Always pair it with `input_format_allow_errors_num`. To skip errors, both settings must be greater than 0.

If an error occurred while reading rows but the error counter is still less than `input_format_allow_errors_ratio`, ClickHouse ignores the row and moves on to the next one.

If `input_format_allow_errors_ratio` is exceeded, ClickHouse throws an exception.

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, 65,536.

Blocks the size of `max_block_size` are not always loaded from the table. If it is obvious that less data needs to be retrieved, a smaller block is processed.

preferred_block_size_bytes

Used for the same purpose as `max_block_size`, but it sets the recommended block size in bytes by adapting it to the number of rows in the block. However, the block size cannot be more than `max_block_size` rows. Disabled by default (set to 0). It only works when reading from MergeTree engines.

[]

log_queries

Setting up query logging.

Queries sent to ClickHouse with this setup are logged according to the rules in thequery_log [#server_settings-query_log] server configuration parameter.

Example:

```
log_queries=1
```

[]

max_insert_block_size

The size of blocks to form for insertion into a table. This setting only applies in cases when the server forms the blocks. For example, for an INSERT via the HTTP interface, the server parses the data format and forms blocks of the specified size. But when using clickhouse-client, the client parses the data itself, and the 'max_insert_block_size' setting on the server doesn't affect the size of the inserted blocks. The setting also doesn't have a purpose when using INSERT SELECT, since data is inserted using the same blocks that are formed after SELECT.

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_replica_delay_for_distributed_queries

Disables lagging replicas for distributed queries. See "Replication [#table_engines-replication]".

Sets the time in seconds. If a replica lags more than the set value, this replica is not used.

Default value: 0 (off).

Used when performing `SELECT` from a distributed table that points to replicated tables.

max_threads

The maximum number of query processing threads

- excluding threads for retrieving data from remote servers (see the 'max_distributed_connections' parameter).

This parameter applies to threads that perform the same stages of the query processing pipeline in parallel. For example, 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 [#table_engines-mergetree]" tables. In order to reduce latency when processing queries, a block is compressed when writing the next mark if its size is at least 'min_compress_block_size'. By default, 65,536.

The actual size of the block, if the uncompressed data is less than 'max_compress_block_size', is no less than this value and no less than the volume of data for one mark.

Let's look at an example. Assume that 'index_granularity' was set to 8192 during table creation.

We are writing a UInt32-type column (4 bytes per value). When writing 8192 rows, the total will be 32 KB of data. Since min_compress_block_size = 65,536, a compressed block will be formed for every two marks.

We are writing a URL column with the String type (average size of 60 bytes per value). When writing 8192 rows, the average will be slightly less than 500 KB of data. Since this is more than 65,536, a compressed block will be formed for each mark. In this case, when reading data from the disk in the range of a single mark, extra data won't be decompressed.

There usually isn't any reason to change this setting.

max_query_size

The maximum part of a query that can be taken to RAM for parsing with the SQL parser. The INSERT query also contains data for INSERT that is processed by a separate stream parser (that consumes O(1) RAM), which is not included in this restriction.

The default is 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.

schema

This parameter is useful when you are using formats that require a schema definition, such asCap'n Proto [https://capnproto.org/]. The value depends on the format.

[]

stream_flush_interval_ms

Works for tables with streaming in the case of a timeout, or when a thread generatesmax_insert_block_size [#settings-settings-max_insert_block_size] rows.

The default value is 7500.

The smaller the value, the more often data is flushed into the table. Setting the value too low leads to poor performance.

[]

load_balancing

Which replicas (among healthy replicas) to preferably send a query to (on the first attempt) for distributed processing.

random (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).

For instance, example01-01-1 and example01-01-2.yandex.ru are different in one position, while example01-01-1 and

example01-02-2 differ in two places. This method might seem 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

Floating-point number from 0 to 1. By default, 1. Allows you to set the default sampling ratio for all SELECT queries. (For tables that do not support sampling, it throws an exception.) If set to 1, sampling is not performed by default.

max_parallel_replicas

The maximum number of replicas for each shard when executing a query. For consistency (to get different parts of the same data split), this option only works when the sampling key is set. Replica lag is not controlled.

compile

Enable compilation of queries. By default, 0 (disabled).

Compilation is only used for part of the query-processing pipeline: for the first stage of aggregation (GROUP BY). If this portion of the pipeline was compiled, the query may run faster due to deployment of short cycles and inlining aggregate function calls. The maximum performance improvement (up to four times faster in rare cases) is seen for queries with multiple simple aggregate functions. Typically, the performance gain is insignificant. In very rare cases, it may slow down query execution.

min_count_to_compile

How many times to potentially use a compiled chunk of code before running compilation. By default, 3. If the value is zero, then compilation runs synchronously and the query waits for the end of the compilation process before continuing execution. This can be used for testing; otherwise, use values starting with 1. Compilation normally takes about 5-10 seconds. If the value is 1 or more, compilation occurs asynchronously in a separate thread. The result will be used as soon as it is ready, including by queries that are currently running.

Compiled code is required for each different combination of aggregate functions used in the query and the type of keys in the GROUP BY clause. The results of compilation are saved in the build directory in the form of .so files. There is no restriction on the number of compilation results, since they don't use very much space. Old results will be used after server restarts, except in the case of a server upgrade – in this case, the old results are deleted.

input_format_skip_unknown_fields

If the value is true, running INSERT skips input data from columns with unknown names. Otherwise, this situation will generate an exception. It works for JSONEachRow and TSKV formats.

output_format_json_quote_64bit_integers

If the value is true, integers appear in quotes when using JSON* Int64 and UInt64 formats (for compatibility with most JavaScript implementations); otherwise, integers are output without the quotes.

[]

format_csv_delimiter

The character interpreted as a delimiter in the CSV data. By default, the delimiter is `,` .

[]

join_use_nulls

Affects the behavior of JOIN [#query_language-join].

With `join_use_nulls=1,` `JOIN` behaves like in standard SQL, i.e. if empty cells appear when merging, the type of the corresponding field is converted to Nullable [#data_type-nullable], and empty cells are filled with NULL [#null-literal].

[]

insert_quorum

Enables quorum writes.

- If `insert_quorum < 2` , the quorum writes are disabled.
- If `insert_quorum >= 2` , the quorum writes are enabled.

The default value is 0.

Quorum writes

`INSERT` succeeds only when ClickHouse manages to correctly write data to the `insert_quorum` of replicas during the `insert_quorum_timeout` . If for any reason the number of replicas with successful writes does not reach the `insert_quorum` , the write is considered failed and ClickHouse will delete the inserted block from all the replicas where data has already been written.

All the replicas in the quorum are consistent, i.e., they contain data from all previous `INSERT` queries. The `INSERT` sequence is linearized.

When reading the data written from the `insert_quorum` , you can use theselect_sequential_consistency [#setting-select_sequential_consistency] option.

ClickHouse generates an exception

- If the number of available replicas at the time of the query is less than the `insert_quorum` .
- At an attempt to write data when the previous block has not yet been inserted in the `insert_quorum` of replicas. This situation may occur if the user tries to perform an `INSERT` before the previous one with the `insert_quorum` is completed.

See also the following parameters:

- insert_quorum_timeout [#setting-insert_quorum_timeout]
- select_sequential_consistency [#setting-select_sequential_consistency]

[]

insert_quorum_timeout

Quorum write timeout in seconds. If the timeout has passed and no write has taken place yet, ClickHouse will generate an exception and the client must repeat the query to write the same block to the same or any other replica.

By default, 60 seconds.

See also the following parameters:

- insert_quorum [#setting-insert_quorum]
- select_sequential_consistency [#setting-select_sequential_consistency]

[]

select_sequential_consistency

Enables/disables sequential consistency for `SELECT` queries:

- 0 — disabled. The default value is 0.
- 1 — enabled.

When sequential consistency is enabled, ClickHouse allows the client to execute the `SELECT` query only for those replicas that contain data from all previous `INSERT` queries executed with `insert_quorum` . If the client refers to a partial replica, ClickHouse will generate an exception. The SELECT query will not include data that has not yet been written to the quorum of replicas.

See also the following parameters:

- insert_quorum [#setting-insert_quorum]
- insert_quorum_timeout [#setting-insert_quorum_timeout]

[]

Settings profiles

A settings profile is a collection of settings grouped under the same name. Each ClickHouse user has a profile. To apply all the settings in a profile, set the `profile` setting.

Example:

Install the `web` profile.

```
SET profile = 'web'
```

Settings profiles are declared in the user config file. This is usually `users.xml` .

Example:

```xml
<!-- Settings profiles -->
<profiles>
    <!-- Default settings -->
    <default>
        <!-- The maximum number of threads when running a single query. -->
        <max_threads>8</max_threads>
    </default>

    <!-- Settings for quries 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>
```

The example specifies two profiles: `default` and `web`. The `default` profile has a special purpose: it must always be present and is applied when starting the server. In other words, the `default` profile contains default settings. The `web` profile is a regular profile that can be set using the `SET` query or using a URL parameter in an HTTP query.

Settings profiles can inherit from each other. To use inheritance, indicate the `profile` setting before the other settings that are listed in the profile.

ClickHouse Utility

- clickhouse-local [#utils-clickhouse-local] — Allows running SQL queries on data without stopping the ClickHouse server, similar to how `awk` does this.
- clickhouse-copier [#utils-clickhouse-copier] — Copies (and reshards) data from one cluster to another cluster.

[]

clickhouse-copier

Copies data from the tables in one cluster to tables in another (or the same) cluster.

You can run multiple `clickhouse-copier` instances on different servers to perform the same job. ZooKeeper is used for syncing the processes.

After starting, `clickhouse-copier`:

- Connects to ZooKeeper and receives:
  - Copying jobs.
  - The state of the copying jobs.
- It performs the jobs.

  Each running process chooses the "closest" shard of the source cluster and copies the data into the destination cluster, resharding the data if necessary.

`clickhouse-copier` tracks the changes in ZooKeeper and applies them on the fly.

To reduce network traffic, we recommend running `clickhouse-copier` on the same server where the source data is located.

Running clickhouse-copier

The utility should be run manually:

```
clickhouse-copier copier --daemon --config zookeeper.xml --task-path /task/path --base-dir /path/to/dir
```

Parameters:

- `daemon` — Starts `clickhouse-copier` in daemon mode.
- `config` — The path to the `zookeeper.xml` file with the parameters for the connection to ZooKeeper.
- `task-path` — The path to the ZooKeeper node. This node is used for syncing `clickhouse-copier` processes and storing tasks. Tasks are stored in `$task-path/description`.
- `base-dir` — The path to logs and auxiliary files. When it starts, `clickhouse-copier` creates `clickhouse-copier_YYYYMMHHSS_<PID>` subdirectories in `$base-dir`. If this parameter is omitted, the directories are created in the directory where `clickhouse-copier` was launched.

Format of zookeeper.xml

```xml
<yandex>
    <zookeeper>
        <node index="1">
            <host>127.0.0.1</host>
            <port>2181</port>
        </node>
    </zookeeper>
</yandex>
```

Configuration of copying tasks

```xml
<yandex>
    <!-- Configuration of clusters as in an ordinary server config -->
    <remote_servers>
        <source_cluster>
            <shard>
                <internal_replication>false</internal_replication>
                    <replica>
                        <host>127.0.0.1</host>
                        <port>9000</port>
                    </replica>
            </shard>
            ...
        </source_cluster>

        <destination_cluster>
        ...
        </destination_cluster>
```

```
    </remote_servers>

    <!-- How many simultaneously active workers are possible. If you run more workers superfluous workers will
sleep. -->
    <max_workers>2</max_workers>

    <!-- Setting used to fetch (pull) data from source cluster tables -->
    <settings_pull>
        <readonly>1</readonly>
    </settings_pull>

    <!-- Setting used to insert (push) data to destination cluster tables -->
    <settings_push>
        <readonly>0</readonly>
    </settings_push>

    <!-- Common setting for fetch (pull) and insert (push) operations. Also, copier process context uses it.
         They are overlaid by <settings_pull/> and <settings_push/> respectively. -->
    <settings>
        <connect_timeout>3</connect_timeout>
        <!-- Sync insert is set forcibly, leave it here just in case. -->
        <insert_distributed_sync>1</insert_distributed_sync>
    </settings>

    <!-- Copying tasks description.
         You could specify several table task in the same task description (in the same ZooKeeper node), they
will be performed
         sequentially.
    -->
    <tables>
        <!-- A table task, copies one table. -->
        <table_hits>
            <!-- Source cluster name (from <remote_servers/> section) and tables in it that should be copied
-->
            <cluster_pull>source_cluster</cluster_pull>
            <database_pull>test</database_pull>
            <table_pull>hits</table_pull>

            <!-- Destination cluster name and tables in which the data should be inserted -->
            <cluster_push>destination_cluster</cluster_push>
            <database_push>test</database_push>
            <table_push>hits2</table_push>

            <!-- Engine of destination tables.
                 If destination tables have not be created, workers create them using columns definition from
source tables and engine
                 definition from here.

                 NOTE: If the first worker starts insert data and detects that destination partition is not
empty then the partition will
                 be dropped and refilled, take it into account if you already have some data in destination
tables. You could directly
                 specify partitions that should be copied in <enabled_partitions/>, they should be in quoted
format like partition column of
                 system.parts table.
            -->
            <engine>
            ENGINE=ReplicatedMergeTree('/clickhouse/tables/{cluster}/{shard}/hits2', '{replica}')
            PARTITION BY toMonday(date)
            ORDER BY (CounterID, EventDate)
            </engine>

            <!-- Sharding key used to insert data to destination cluster -->
            <sharding_key>jumpConsistentHash(intHash64(UserID), 2)</sharding_key>

            <!-- Optional expression that filter data while pull them from source servers -->
            <where_condition>CounterID != 0</where_condition>

            <!-- This section specifies partitions that should be copied, other partition will be ignored.
                 Partition names should have the same format as
                 partition column of system.parts table (i.e. a quoted text).
                 Since partition key of source and destination cluster could be different,
                 these partition names specify destination partitions.
```

```
                NOTE: In spite of this section is optional (if it is not specified, all partitions will be copied),
                it is strictly recommended to specify them explicitly.
                If you already have some ready paritions on destination cluster they
                will be removed at the start of the copying since they will be interpeted
                as unfinished data from the previous copying!!!
            -->
            <enabled_partitions>
                <partition>'2018-02-26'</partition>
                <partition>'2018-03-05'</partition>
                ...
            </enabled_partitions>
        </table_hits>

        <!-- Next table to copy. It is not copied until previous table is copying. -->
        </table_visits>
        ...
        </table_visits>
        ...
    </tables>
</yandex>
```

`clickhouse-copier` tracks the changes in `/task/path/description` and applies them on the fly. For instance, if you change the value of `max_workers` , the number of processes running tasks will also change.

[]

clickhouse-local

The `clickhouse-local` program enables you to perform fast processing on local files, without having to deploy and configure the ClickHouse server.

Accepts data that represent tables and queries them using ClickHouse SQL dialect [#queries].

`clickhouse-local` uses the same core as ClickHouse server, so it supports most of the features and the same set of formats and table engines.

By default `clickhouse-local` does not have access to data on the same host, but it supports loading server configuration using `--config-file` argument.

Warning

It is not recommended to load production server configuration into `clickhouse-local` because data can be damaged in case of human error.

Usage

Basic usage:

```
clickhouse-local --structure "table_structure" --input-format "format_of_incoming_data" -q "query"
```

Arguments:

- `-S` , `--structure` — table structure for input data.
- `-if` , `--input-format` — input format, `TSV` by default.
- `-f` , `--file` — path to data, `stdin` by default.
- `-q` `--query` — queries to execute with `;` as delimeter.
- `-N` , `--table` — table name where to put output data, `table` by default.
- `-of` , `--format` , `--output-format` — output format, `TSV` by default.

- `--stacktrace` — whether to dump debug output in case of exception.
- `--verbose` — more details on query execution.
- `-s` — disables `stderr` logging.
- `--config-file` — path to configuration file in same format as for ClickHouse server, by default the configuration empty.
- `--help` — arguments references for `clickhouse-local`.

Also there are arguments for each ClickHouse configuration variable which are more commonly used instead of `--config-file`.

Examples

```
echo -e "1,2\n3,4" | clickhouse-local -S "a Int64, b Int64" -if "CSV" -q "SELECT * FROM table"
Read 2 rows, 32.00 B in 0.000 sec., 5182 rows/sec., 80.97 KiB/sec.
1   2
3   4
```

Previous example is the same as:

```
$ echo -e "1,2\n3,4" | clickhouse-local -q "CREATE TABLE table (a Int64, b Int64) ENGINE = File(CSV, stdin);
SELECT a, b FROM table; DROP TABLE table"
Read 2 rows, 32.00 B in 0.000 sec., 4987 rows/sec., 77.93 KiB/sec.
1   2
3   4
```

Now let's output memory user for each Unix user:

```
$ ps aux | tail -n +2 | awk '{ printf("%s\t%s\n", $1, $4) }' | clickhouse-local -S "user String, mem Float64"
-q "SELECT user, round(sum(mem), 2) as memTotal FROM table GROUP BY user ORDER BY memTotal DESC FORMAT Pretty"
Read 186 rows, 4.15 KiB in 0.035 sec., 5302 rows/sec., 118.34 KiB/sec.
┏━━━━━━━━━━┳━━━━━━━━━━┓
┃ user     ┃ memTotal ┃
┡━━━━━━━━━━╇━━━━━━━━━━┩
│ bayonet  │    113.5 │
├──────────┼──────────┤
│ root     │      8.8 │
├──────────┼──────────┤
...
```

常见问题

为什么不使用MapReduce之类的产品呢?

我们可以将MapReduce这类的系统称为分布式计算系统，其reduce操作基于分布式排序。其中最常见的开源解决方案是Apache Hadoop [http://hadoop.apache.org]。 Yandex使用他们的内部解决方案YT。

这些系统不适合在线查询，因为它们的延迟高。换句话说，它们不能用作Web接口的后端服务。这些系统对于实时数据更新是没有用的。如果操作的结果和所有中间结果（如果有的话）位于单个服务器的内存中，则分布式排序不是执行reduce操作的最佳方式，但这通常是在线查询的情况。在这种情况下，哈希表是执行reduce操作的最佳方式。优化map-reduce任务的常用方法是使用内存中的哈希表进行预聚合（部分reduce），用户手动执行此优化操作。分布式排序是运行简单map-reduce任务时性能降低的主要原因之一。

大多数MapReduce系统允许您在集群上执行任意代码。但是，声明性查询语言更适合OLAP，以便快速运行实验。例如，Hadoop包含Hive和Pig，Cloudera Impala或Shark（过时）for Spark，以及Spark SQL、Presto和Apache Drill。与专业系统相比，运行此类任务时的性能非常不理想，所以将这些系统用作Web接口的后端服务是不现实的，因为延迟相对较高。

ClickHouse Development

Overview of ClickHouse Architecture

ClickHouse is a true column-oriented DBMS. Data is stored by columns, and during the execution of arrays (vectors or chunks of columns). Whenever possible, operations are dispatched on arrays, rather than on individual values. This is called "vectorized query execution," and it helps lower the cost of actual data processing.

> This idea is nothing new. It dates back to the `APL` programming language and its descendants: `A +`, `J`, `K`, and `Q`. Array programming is used in scientific data processing. Neither is this idea something new in relational databases: for example, it is used in the `Vectorwise` system.

There are two different approaches for speeding up the query processing: vectorized query execution and runtime code generation. In the latter, the code is generated for every kind of query on the fly, removing all indirection and dynamic dispatch. Neither of these approaches is strictly better than the other. Runtime code generation can be better when it's fuses many operations together, thus fully utilizing CPU execution units and the pipeline. Vectorized query execution can be less practical, because it involves the temporary vectors that must be written to the cache and read back. If the temporary data does not fit in the L2 cache, this becomes an issue. But vectorized query execution more easily utilizes the SIMD capabilities of the CPU. A research paper [http://15721.courses.cs.cmu.edu/spring2016/papers/p5-sompolski.pdf] written by our friends shows that it is better to combine both approaches. ClickHouse uses vectorized query execution and has limited initial support for runtime code.

Columns

To represent columns in memory (actually, chunks of columns), the `IColumn` interface is used. This interface provides helper methods for implementation of various relational operators. Almost all operations are immutable: they do not modify the original column, but create a new modified one. For example, the `IColumn :: filter` method accepts a filter byte mask. It is used for the `WHERE` and `HAVING` relational operators. Additional examples: the `IColumn :: permute` method to support `ORDER BY`, the `IColumn :: cut` method to support `LIMIT`, and so on.

Various `IColumn` implementations (`ColumnUInt8`, `ColumnString` and so on) are responsible for the memory layout of columns. Memory layout is usually a contiguous array. For the integer type of columns it is just one contiguous array, like `std :: vector`. For `String` and `Array` columns, it is two vectors: one for all array elements, placed contiguously, and a second one for offsets to the beginning of each array. There is also `ColumnConst` that stores just one value in memory, but looks like a column.

Field

Nevertheless, it is possible to work with individual values as well. To represent an individual value, the `Field` is used. `Field` is just a discriminated union of `UInt64`, `Int64`, `Float64`, `String` and `Array`. `IColumn` has the `operator[]` method to get the n-th value as a `Field`, and the `insert` method to append a `Field` to the end of a column. These methods are not very efficient, because they require dealing with temporary `Field` objects representing an individual value. There are more efficient methods, such as `insertFrom`, `insertRangeFrom`, and so on.

`Field` doesn't have enough information about a specific data type for a table. For example, `UInt8`, `UInt16`, `UInt32`, and `UInt64` are all represented as `UInt64` in a `Field`.

Leaky Abstractions

`IColumn` has methods for common relational transformations of data, but they don't meet all needs. For example, `ColumnUInt64` doesn't have a method to calculate the sum of two columns, and `ColumnString` doesn't have a method to run a substring search. These countless routines are implemented outside of `IColumn`.

Various functions on columns can be implemented in a generic, non-efficient way using `IColumn` methods to extract `Field` values, or in a specialized way using knowledge of inner memory layout of data in a specific `IColumn` implementation. To do this, functions are cast to a specific `IColumn` type and deal with internal representation directly. For example, `ColumnUInt64` has the `getData` method that returns a reference to an internal array, then a separate routine reads or fills that array directly. In fact, we have "leaky abstractions" to allow efficient specializations of various routines.

Data Types

`IDataType` is responsible for serialization and deserialization: for reading and writing chunks of columns or individual values in binary or text form. `IDataType` directly corresponds to data types in tables. For example, there are `DataTypeUInt32`, `DataTypeDateTime`, `DataTypeString` and so on.

`IDataType` and `IColumn` are only loosely related to each other. Different data types can be represented in memory by the same `IColumn` implementations. For example, `DataTypeUInt32` and `DataTypeDateTime` are both represented by `ColumnUInt32` or `ColumnConstUInt32`. In addition, the same data type can be represented by different `IColumn` implementations. For example, `DataTypeUInt8` can be represented by `ColumnUInt8` or `ColumnConstUInt8`.

`IDataType` only stores metadata. For instance, `DataTypeUInt8` doesn't store anything at all (except vptr) and `DataTypeFixedString` stores just `N` (the size of fixed-size strings).

`IDataType` has helper methods for various data formats. Examples are methods to serialize a value with possible quoting, to serialize a value for JSON, and to serialize a value as part of XML format. There is no direct correspondence to data formats. For example, the different data formats `Pretty` and `TabSeparated` can use the same `serializeTextEscaped` helper method from the `IDataType` interface.

Block

A `Block` is a container that represents a subset (chunk) of a table in memory. It is just a set of triples: `(IColumn, IDataType, column name)`. During query execution, data is processed by `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 article [http://journal.stuffwithstuff.com/2013/01/13/iteration-inside-and-out/] 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 synchronize their data and maintain consistency. Replicas can be added and removed dynamically simply by creating or dropping a table.

Replication uses an asynchronous multi-master scheme. You can insert data into any replica that has a session with `ZooKeeper` , and data is replicated to all other replicas asynchronously. Because ClickHouse doesn't support UPDATEs, replication is conflict-free. As there is no quorum acknowledgment of inserts, just-inserted data might be lost if one node fails.

Metadata for replication is stored in ZooKeeper. There is a replication log that lists what actions to do. Actions are: get part; merge parts; drop partition, etc. Each replica copies the replication log to its queue and then executes the actions from the queue. For example, on insertion, the "get part" action is created in the log, and every replica downloads that part. Merges are coordinated between replicas to get byte-identical results. All parts are merged in the same way on all replicas. To achieve this, one replica is elected as the leader, and that replica initiates merges and writes "merge parts" actions to the log.

Replication is physical: only compressed parts are transferred between nodes, not queries. To lower the network cost (to avoid network amplification), merges are processed on each replica independently in most cases. Large merged parts are sent over the network only in cases of significant replication lag.

In addition, each replica stores its state in ZooKeeper as the set of parts and its checksums. When the state on the local filesystem diverges from the reference state in ZooKeeper, the replica restores its consistency by downloading missing and broken parts from other replicas. When there is some unexpected or broken data in the local filesystem, ClickHouse does not remove it, but moves it to a separate directory and forgets it.

> The ClickHouse cluster consists of independent shards, and each shard consists of replicas. The cluster is not elastic, so after adding a new shard, data is not rebalanced between shards automatically. Instead, the cluster load will be uneven. This implementation gives you more control, and it is fine for relatively small clusters such as tens of nodes. But for clusters with hundreds of nodes that we are using in production, this approach becomes a significant drawback. We should implement a table engine that will span its data across the cluster with dynamically replicated regions that could be split and balanced between clusters automatically.

How to Build ClickHouse Release Package

Install Git and Pbuilder

```
sudo apt-get update
sudo apt-get install git pbuilder debhelper lsb-release fakeroot sudo debian-archive-keyring debian-keyring
```

Checkout ClickHouse Sources

```
git clone --recursive --branch stable https://github.com/yandex/ClickHouse.git
cd ClickHouse
```

Run Release Script

```
./release
```

How to Build ClickHouse for Development

Build should work on Ubuntu Linux. With appropriate changes, it should also work on any other Linux distribution. The build process 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 ninja-build
```

Or cmake3 instead of cmake on older systems.

Install GCC 7

There are several ways to do this.

Install from a PPA Package

```
sudo apt-get install software-properties-common
sudo apt-add-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
sudo apt-get install gcc-7 g++-7
```

Install from Sources

Look at ci/build-gcc-from-sources.sh [https://github.com/yandex/ClickHouse/blob/master/ci/build-gcc-from-sources.sh]

Use GCC 7 for Builds

```
export CC=gcc-7
export CXX=g++-7
```

Install Required Libraries from Packages

```
sudo apt-get install libicu-dev libreadline-dev
```

Checkout ClickHouse Sources

```
git clone --recursive git@github.com:yandex/ClickHouse.git
## or: git clone --recursive https://github.com/yandex/ClickHouse.git

cd ClickHouse
```

For the latest stable version, switch to the `stable` branch.

Build ClickHouse

```
mkdir build
cd build
cmake ..
ninja
cd ..
```

To create an executable, run `ninja clickhouse`. This will create the `dbms/programs/clickhouse` executable, which can be used with `client` or `server` arguments.

How to Build ClickHouse on Mac OS X

Build should work on Mac OS X 10.12. If you're using earlier version, you can try to build ClickHouse using Gentoo Prefix and clang sl in this instruction. With appropriate changes, it should also work on any other Linux distribution.

Install Homebrew

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Install Required Compilers, Tools, and Libraries

```
brew install cmake ninja gcc icu4c mariadb-connector-c openssl libtool gettext readline
```

Checkout ClickHouse Sources

```
git clone --recursive git@github.com:yandex/ClickHouse.git
## or: git clone --recursive https://github.com/yandex/ClickHouse.git

cd ClickHouse
```

For the latest stable version, switch to the `stable` branch.

Build ClickHouse

```
mkdir build
cd build
cmake .. -DCMAKE_CXX_COMPILER=`which g++-8` -DCMAKE_C_COMPILER=`which gcc-8`
ninja
cd ..
```

Caveats

If you intend to run clickhouse-server, make sure to increase the system's maxfiles variable.

Note

You'll need to use sudo.

To do so, create the following file:

/Library/LaunchDaemons/limit.maxfiles.plist:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
        "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>Label</key>
    <string>limit.maxfiles</string>
    <key>ProgramArguments</key>
    <array>
      <string>launchctl</string>
      <string>limit</string>
      <string>maxfiles</string>
      <string>524288</string>
      <string>524288</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
    <key>ServiceIPC</key>
    <false/>
  </dict>
</plist>
```

Execute the following command:

```
$ sudo chown root:wheel /Library/LaunchDaemons/limit.maxfiles.plist
```

Reboot.

To check if it's working, you can use `ulimit -n` command.

How to Write C++ Code

General Recommendations

1. The following are recommendations, not requirements.

2. If you are editing code, it makes sense to follow the formatting of the existing code.

3. Code style is needed for consistency. Consistency makes it easier to read the code, and it also makes it easier to search the code.

4. Many of the rules do not have logical reasons; they are dictated by established practices.

Formatting

1. Most of the formatting will be done automatically by `clang-format`.

2. Indents are 4 spaces. Configure your development environment so that a tab adds four spaces.

3. Opening and closing curly brackets must be on a separate line.

```
inline void readBoolText(bool & x, ReadBuffer & buf)
{
    char tmp = '0';
    readChar(tmp, buf);
    x = tmp != '0';
}
```

4. If the entire function body is a single `statement`, it can be placed on a single line. Place spaces around curly braces (besides the space at the end of the line).

```
inline size_t mask() const                { return buf_size() - 1; }
inline size_t place(HashValue x) const    { return x & mask(); }
```

5. For functions. Don't put spaces around brackets.

```
void reinsert(const Value & x)
```

```
memcpy(&buf[place_value], &x, sizeof(x));
```

6. In `if`, `for`, `while` and other expressions, a space is inserted in front of the opening bracket (as opposed to function calls).

```
for (size_t i = 0; i < rows; i += storage.index_granularity)
```

7. Add spaces around binary operators (`+`, `-`, `*`, `/`, `%`, ...) and the ternary operator `?:`.

```
UInt16 year = (s[0] - '0') * 1000 + (s[1] - '0') * 100 + (s[2] - '0') * 10 + (s[3] - '0');
UInt8 month = (s[5] - '0') * 10 + (s[6] - '0');
UInt8 day = (s[8] - '0') * 10 + (s[9] - '0');
```

8. If a line feed is entered, put the operator on a new line and increase the indent before it.

```
if (elapsed_ns)
    message << " ("
        << rows_read_on_server * 1000000000 / elapsed_ns << " rows/s., "
        << bytes_read_on_server * 1000.0 / elapsed_ns << " MB/s.) ";
```

9. You can use spaces for alignment within a line, if desired.

```
dst.ClickLogID         = click.LogID;
dst.ClickEventID       = click.EventID;
dst.ClickGoodEvent     = click.GoodEvent;
```

10. Don't use spaces around the operators `.` , `->` .

If necessary, the operator can be wrapped to the next line. In this case, the offset in front of it is increased.

11. Do not use a space to separate unary operators (`--` , `++` , `*` , `&` , ...) from the argument.

12. Put a space after a comma, but not before it. The same rule goes for a semicolon inside a `for` expression.

13. Do not use spaces to separate the `[]` operator.

14. In a `template <...>` expression, use a space between `template` and `<` ; no spaces after `<` or before `>` .

```
template <typename TKey, typename TValue>
struct AggregatedStatElement
{}
```

15. In classes and structures, write `public` , `private` , and `protected` on the same level as `class/struct` , and indent the rest of the code.

```
template <typename T>
class MultiVersion
{
public:
    /// Version of object for usage. shared_ptr manage lifetime of version.
    using Version = std::shared_ptr<const T>;
    ...
}
```

16. If the same `namespace` is used for the entire file, and there isn't anything else significant, an offset is not necessary inside `namespace` .

17. If the block for an `if` , `for` , `while` , or other expression consists of a single `statement` , the curly brackets are optional. Place the `statement` on a separate line, instead. This rule is also valid for nested `if` , `for` , `while` , ...

But if the inner `statement` contains curly brackets or `else` , the external block should be written in curly brackets.

```
/// Finish write.
for (auto & stream : streams)
    stream.second->finalize();
```

18. There shouldn't be any spaces at the ends of lines.

19. Source files are UTF-8 encoded.

20. Non-ASCII characters can be used in string literals.

```
<< ", " << (timer.elapsed() / chunks_stats.hits) << " μsec/hit.";
```

21 Do not write multiple expressions in a single line.

22. Group sections of code inside functions and separate them with no more than one empty line.

23. Separate functions, classes, and so on with one or two empty lines.

24. `A const` (related to a value) must be written before the type name.

```
//correct
const char * pos
const std::string & s
//incorrect
char const * pos
```

25. When declaring a pointer or reference, the `*` and `&` symbols should be separated by spaces on both sides.

```
//correct
const char * pos
//incorrect
const char* pos
const char *pos
```

26. When using template types, alias them with the `using` keyword (except in the simplest cases).

In other words, the template parameters are specified only in `using` and aren't repeated in the code.

`using` can be declared locally, such as inside a function.

```
//correct
using FileStreams = std::map<std::string, std::shared_ptr<Stream>>;
FileStreams streams;
//incorrect
std::map<std::string, std::shared_ptr<Stream>> streams;
```

27. Do not declare several variables of different types in one statement.

```
//incorrect
int x, *y;
```

28. Do not use C-style casts.

```
//incorrect
std::cerr << (int)c <<; std::endl;
//correct
std::cerr << static_cast<int>(c) << std::endl;
```

29. In classes and structs, group members and functions separately inside each visibility scope.

30. For small classes and structs, it is not necessary to separate the method declaration from the implementation.

The same is true for small methods in any classes or structs.

For templated classes and structs, don't separate the method declarations from the implementation (because otherwise they must be defined in the same translation unit).

31. You can wrap lines at 140 characters, instead of 80.

32. Always use the prefix increment/decrement operators if postfix is not required.

```
for (Names::const_iterator it = column_names.begin(); it != column_names.end(); ++it)
```

Comments

1. Be sure to add comments for all non-trivial parts of code.

This is very important. Writing the comment might help you realize that the code isn't necessary, or that it is designed wrong.

```
/** Part of piece of memory, that can be used.
  * For example, if internal_buffer is 1MB, and there was only 10 bytes loaded to buffer from file for
reading,
  * then working_buffer will have size of only 10 bytes
  * (working_buffer.end() will point to position right after those 10 bytes available for read).
  */
```

2. Comments can be as detailed as necessary.

3. Place comments before the code they describe. In rare cases, comments can come after the code, on the same line.

```
/** Parses and executes the query.
*/
void executeQuery(
    ReadBuffer & istr, /// Where to read the query from (and data for INSERT, if applicable)
    WriteBuffer & ostr, /// Where to write the result
    Context & context, /// DB, tables, data types, engines, functions, aggregate functions...
    BlockInputStreamPtr & query_plan, /// Here could be written the description on how query was executed
    QueryProcessingStage::Enum stage = QueryProcessingStage::Complete /// Up to which stage process the
SELECT query
    )
```

4. Comments should be written in English only.

5. If you are writing a library, include detailed comments explaining it in the main header file.

6. Do not add comments that do not provide additional information. In particular, do not leave empty comments like this:

```
/*
* Procedure Name:
* Original procedure name:
* Author:
* Date of creation:
* Dates of modification:
* Modification authors:
* Original file name:
* Purpose:
* Intent:
* Designation:
* Classes used:
* Constants:
* Local variables:
* Parameters:
* Date of creation:
* Purpose:
*/
```

The example is borrowed from the resource http://home.tamk.fi/~jaalto/course/coding-style/doc/unmaintainable-code/ [http://home.tamk.fi/~jaalto/course/coding-style/doc/unmaintainable-code/].

7. Do not write garbage comments (author, creation date ..) at the beginning of each file.

8. Single-line comments begin with three slashes: `///` and multi-line comments begin with `/**`. These comments are considered "documentation".

Note: You can use Doxygen to generate documentation from these comments. But Doxygen is not generally used because it is more convenient to navigate the code in the IDE.

9. Multi-line comments must not have empty lines at the beginning and end (except the line that closes a multi-line comment).

10. For commenting out code, use basic comments, not “documenting” comments.

11. Delete the commented out parts of the code before committing.

12. Do not use profanity in comments or code.

13. Do not use uppercase letters. Do not use excessive punctuation.

```
/// WHAT THE FAIL???
```

14. Do not use comments to make delimeters.

```
///******************************************************
```

15. Do not start discussions in comments.

```
/// Why did you do this stuff?
```

16. There's no need to write a comment at the end of a block describing what it was about.

```
/// for
```

Names

1. Use lowercase letters with underscores in the names of variables and class members.

```
size_t max_block_size;
```

2. For the names of functions (methods), use camelCase beginning with a lowercase letter.

````
```cpp
std::string getName() const override { return "Memory"; }
```
````

3. For the names of classes (structs), use CamelCase beginning with an uppercase letter. Prefixes other than I are not used for interfaces.

````
```cpp
class StorageMemory : public IStorage
```
````

4. `using` are named the same way as classes, or with `_t` on the end.

5. Names of template type arguments: in simple cases, use `T`; `T`, `U`; `T1`, `T2`.

For more complex cases, either follow the rules for class names, or add the prefix `T`.

```
template <typename TKey, typename TValue>
struct AggregatedStatElement
```

6. Names of template constant arguments: either follow the rules for variable names, or use `N` in simple cases.

```
template <bool without_www>
struct ExtractDomain
```

7. For abstract classes (interfaces) you can add the `I` prefix.

```
class IBlockInputStream
```

8. If you use a variable locally, you can use the short name.

In all other cases, use a name that describes the meaning.

```
bool info_successfully_loaded = false;
```

9. Names of `define` s and global constants use ALL_CAPS with underscores.

```
##define MAX_SRC_TABLE_NAMES_TO_STORE 1000
```

10. File names should use the same style as their contents.

If a file contains a single class, name the file the same way as the class (CamelCase).

If the file contains a single function, name the file the same way as the function (camelCase).

11. If the name contains an abbreviation, then:

- For variable names, the abbreviation should use lowercase letters `mysql_connection` (not `mySQL_connection`).
- For names of classes and functions, keep the uppercase letters in the abbreviation `MySQLConnection` (not `MySqlConnection`).

12. Constructor arguments that are used just to initialize the class members should be named the same way as the class members, but with an underscore at the end.

```
FileQueueProcessor(
    const std::string & path_,
    const std::string & prefix_,
    std::shared_ptr<FileHandler> handler_)
    : path(path_),
    prefix(prefix_),
    handler(handler_),
    log(&Logger::get("FileQueueProcessor"))
{
}
```

The underscore suffix can be omitted if the argument is not used in the constructor body.

13. There is no difference in the names of local variables and class members (no prefixes required).

```
timer (not m_timer)
```

14. For the constants in an `enum` , use CamelCase with a capital letter. ALL_CAPS is also acceptable. If the `enum` is non-local, use an `enum class` .

```
enum class CompressionMethod
{
    QuickLZ = 0,
    LZ4     = 1,
};
```

15. All names must be in English. Transliteration of Russian words is not allowed.

```
not Stroka
```

16. Abbreviations are acceptable if they are well known (when you can easily find the meaning of the abbreviation in Wikipedia or in a search engine).

```
`AST`, `SQL`.

Not `NVDH` (some random letters)
```

Incomplete words are acceptable if the shortened version is common use.

You can also use an abbreviation if the full name is included next to it in the comments.

17. File names with C++ source code must have the `.cpp` extension. Header files must have the `.h` extension.

How to Write Code

1. Memory management.

Manual memory deallocation (`delete`) can only be used in library code.

In library code, the `delete` operator can only be used in destructors.

In application code, memory must be freed by the object that owns it.

Examples:

- The easiest way is to place an object on the stack, or make it a member of another class.
- For a large number of small objects, use containers.
- For automatic deallocation of a small number of objects that reside in the heap, use `shared_ptr/unique_ptr` .

2. Resource management.

Use `RAII` and see above.

3. Error handling.

Use exceptions. In most cases, you only need to throw an exception, and don't need to catch it (because of `RAII`).

In offline data processing applications, it's often acceptable to not catch exceptions.

In servers that handle user requests, it's usually enough to catch exceptions at the top level of the connection handler.

In thread functions, you should catch and keep all exceptions to rethrow them in the main thread after `join` .

```
/// If there weren't any calculations yet, calculate the first block synchronously
if (!started)
{
    calculate();
    started = true;
}
else /// If calculations are already in progress, wait for the result
    pool.wait();

if (exception)
    exception->rethrow();
```

Never hide exceptions without handling. Never just blindly put all exceptions to log.

```
//Not correct
catch (...) {}
```

If you need to ignore some exceptions, do so only for specific ones and rethrow the rest.

```
catch (const DB::Exception & e)
{
    if (e.code() == ErrorCodes::UNKNOWN_AGGREGATE_FUNCTION)
        return nullptr;
    else
        throw;
}
```

When using functions with response codes or `errno`, always check the result and throw an exception in case of error.

```
if (0 != close(fd))
    throwFromErrno("Cannot close file " + file_name, ErrorCodes::CANNOT_CLOSE_FILE);
```

`Do not use assert`.

4. Exception types.

There is no need to use complex exception hierarchy in application code. The exception text should be understandable to a system administrator.

5. Throwing exceptions from destructors.

This is not recommended, but it is allowed.

Use the following options:

- Create a function (`done()` or `finalize()`) that will do all the work in advance that might lead to an exception. If that function was called, there should be no exceptions in the destructor later.
- Tasks that are too complex (such as sending messages over the network) can be put in separate method that the class user will have to call before destruction.
- If there is an exception in the destructor, it's better to log it than to hide it (if the logger is available).
- In simple applications, it is acceptable to rely on `std::terminate` (for cases of `noexcept` by default in C++11) to handle exceptions.

6. Anonymous code blocks.

You can create a separate code block inside a single function in order to make certain variables local, so that the destructors are called when exiting the block.

```
Block block = data.in->read();

{
    std::lock_guard<std::mutex> lock(mutex);
    data.ready = true;
    data.block = block;
}

ready_any.set();
```

7. Multithreading.

In offline data processing programs:

- Try to get the best possible performance on a single CPU core. You can then parallelize your code if necessary.

In server applications:

- Use the thread pool to process requests. At this point, we haven't had any tasks that required userspace context switching.

Fork is not used for parallelization.

8. Syncing threads.

Often it is possible to make different threads use different memory cells (even better: different cache lines,) and to not use any thread synchronization (except `joinAll`).

If synchronization is required, in most cases, it is sufficient to use mutex under `lock_guard`.

In other cases use system synchronization primitives. Do not use busy wait.

Atomic operations should be used only in the simplest cases.

Do not try to implement lock-free data structures unless it is your primary area of expertise.

9. Pointers vs references.

In most cases, prefer references.

10. const.

Use constant references, pointers to constants, `const_iterator`, and const methods.

Consider `const` to be default and use non-`const` only when necessary.

When passing variables by value, using `const` usually does not make sense.

11. unsigned.

Use `unsigned` if necessary.

12. Numeric types.

Use the types `UInt8`, `UInt16`, `UInt32`, `UInt64`, `Int8`, `Int16`, `Int32`, and `Int64`, as well as `size_t`, `ssize_t`, and `ptrdiff_t`.

Don't use these types for numbers: `signed/unsigned long`, `long long`, `short`, `signed/unsigned char`, `char`.

13. Passing arguments.

Pass complex values by reference (including `std::string`).

If a function captures ownership of an object created in the heap, make the argument type `shared_ptr` or `unique_ptr`.

14. Return values.

In most cases, just use `return`. Do not write `[return std::move(res)]{.strike}`.

If the function allocates an object on heap and returns it, use `shared_ptr` or `unique_ptr`.

In rare cases you might need to return the value via an argument. In this case, the argument should be a reference.

```
using AggregateFunctionPtr = std::shared_ptr<IAggregateFunction>;

/** Allows creating an aggregate function by its name.
  */
class AggregateFunctionFactory
{
public:
    AggregateFunctionFactory();
    AggregateFunctionPtr get(const String & name, const DataTypes & argument_types) const;
```

15. namespace.

There is no need to use a separate `namespace` for application code.

Small libraries don't need this, either.

For medium to large libraries, put everything in a `namespace` .

In the library's `.h` file, you can use `namespace detail` to hide implementation details not needed for the application code.

In a `.cpp` file, you can use a `static` or anonymous namespace to hide symbols.

Also, a `namespace` can be used for an `enum` to prevent the corresponding names from falling into an external `namespace` (but it's better to use an `enum class`).

16. Deferred initialization.

If arguments are required for initialization, then you normally shouldn't write a default constructor.

If later you'll need to delay initialization, you can add a default constructor that will create an invalid object. Or, for a small number of objects, you can use `shared_ptr/unique_ptr` .

```
Loader(DB::Connection * connection_, const std::string & query, size_t max_block_size_);

/// For deferred initialization
Loader() {}
```

17. Virtual functions.

If the class is not intended for polymorphic use, you do not need to make functions virtual. This also applies to the destructor.

18. Encodings.

Use UTF-8 everywhere. Use `std::string` and `char *` . Do not use `std::wstring` and `wchar_t` .

19. Logging.

See the examples everywhere in the code.

Before committing, delete all meaningless and debug logging, and any other types of debug output.

Logging in cycles should be avoided, even on the Trace level.

Logs must be readable at any logging level.

Logging should only be used in application code, for the most part.

Log messages must be written in English.

The log should preferably be understandable for the system administrator.

Do not use profanity in the log.

Use UTF-8 encoding in the log. In rare cases you can use non-ASCII characters in the log.

20. Input-output.

Don't use `iostreams` in internal cycles that are critical for application performance (and never use `stringstream`).

Use the `DB/IO` library instead.

21. Date and time.

See the `DateLUT` library.

22. include.

Always use `#pragma once` instead of include guards.

23. using.

`using namespace` is not used. You can use `using` with something specific. But make it local inside a class or function.

24. Do not use `trailing return type` for functions unless necessary.

```
[auto f() -&gt; void;]{.strike}
```

25. Declaration and initialization of variables.

```
//right way
std::string s = "Hello";
std::string s{"Hello"};

//wrong way
auto s = std::string{"Hello"};
```

26. For virtual functions, write `virtual` in the base class, but write `override` instead of `virtual` in descendent classes.

Unused Features of C++

1. Virtual inheritance is not used.

2. Exception specifiers from C++03 are not used.

Platform

1. We write code for a specific platform.

But other things being equal, cross-platform or portable code is preferred.

2. Language: C++17.

3. Compiler: `gcc`. At this time (December 2017), the code is compiled using version 7.2. (It can also be compiled using `clang 4`.)

The standard library is used (`libstdc++` or `libc++`).

4.OS: Linux Ubuntu, not older than Precise.

5.Code is written for x86_64 CPU architecture.

The CPU instruction set is the minimum supported set among our servers. Currently, it is SSE 4.2.

6. Use `-Wall -Wextra -Werror` compilation flags.

7. Use static linking with all libraries except those that are difficult to connect to statically (see the output of the `ldd` command).

8. Code is developed and debugged with release settings.

Tools

1. KDevelop is a good IDE.

2. For debugging, use `gdb`, `valgrind` (`memcheck`), `strace`, `-fsanitize=...`, or `tcmalloc_minimal_debug`.

3. For profiling, use `Linux Perf`, `valgrind` (`callgrind`), or `strace -cf`.

4. Sources are in Git.

5. Assembly uses `CMake` .

6. Programs are released using `deb` packages.

7. Commits to master must not break the build.

Though only selected revisions are considered workable.

8. Make commits as often as possible, even if the code is only partially ready.

Use branches for this purpose.

If your code in the `master` branch is not buildable yet, exclude it from the build before the `push` . You'll need to finish it or remove it within a few days.

9. For non-trivial changes, use branches and publish them on the server.

10. Unused code is removed from the repository.

Libraries

1. The C++14 standard library is used (experimental extensions are allowed), as well as `boost` and `Poco` frameworks.

2. If necessary, you can use any well-known libraries available in the OS package.

If there is a good solution already available, then use it, even if it means you have to install another library.

(But be prepared to remove bad libraries from code.)

3. You can install a library that isn't in the packages, if the packages don't have what you need or have an outdated version or the wrong type of compilation.

4. If the library is small and doesn't have its own complex build system, put the source files in the `contrib` folder.

5. Preference is always given to libraries that are already in use.

General Recommendations

1. Write as little code as possible.

2. Try the simplest solution.

3. Don't write code until you know how it's going to work and how the inner loop will function.

4. In the simplest cases, use `using` instead of classes or structs.

5. If possible, do not write copy constructors, assignment operators, destructors (other than a virtual one, if the class contains at least one virtual function), move constructors or move assignment operators. In other words, the compiler-generated functions must work correctly. You can use `default` .

6. Code simplification is encouraged. Reduce the size of your code where possible.

Additional Recommendations

1. Explicitly specifying `std::` for types from `stddef.h`

is not recommended. In other words, we recommend writing `size_t` instead `std::size_t` , because it's shorter.

It is acceptable to add `std::` .

2. Explicitly specifying `std::` for functions from the standard C library

is not recommended. In other words, write `memcpy` instead of `std::memcpy`.

The reason is that there are similar non-standard functions, such as `memmem`. We do use these functions on occasion. These functions do not exist in `namespace std`.

If you write `std::memcpy` instead of `memcpy` everywhere, then `memmem` without `std::` will look strange.

Nevertheless, you can still use `std::` if you prefer it.

3. Using functions from C when the same ones are available in the standard C++ library.

This is acceptable if it is more efficient.

For example, use `memcpy` instead of `std::copy` for copying large chunks of memory.

4. Multiline function arguments.

Any of the following wrapping styles are allowed:

```
function(
  T1 x1,
  T2 x2)
```

```
function(
  size_t left, size_t right,
  const & RangesInDataParts ranges,
  size_t limit)
```

```
function(size_t left, size_t right,
  const & RangesInDataParts ranges,
  size_t limit)
```

```
function(size_t left, size_t right,
      const & RangesInDataParts ranges,
      size_t limit)
```

```
function(
      size_t left,
      size_t right,
      const & RangesInDataParts ranges,
      size_t limit)
```

ClickHouse Testing

Functional Tests

Functional tests are the most simple and convenient to use. Most of ClickHouse features can be tested with functional tests and they are mandatory to use for every change in ClickHouse code that can be tested that way.

Each functional test sends one or multiple queries to the running ClickHouse server and compares the result with reference.

Tests are located in `dbms/src/tests/queries` directory. There are two subdirectories: `stateless` and `stateful`. Stateless tests run queries without any preloaded test data - they often create small synthetic datasets on the fly, within the test itself. Stateful tests require preloaded test data from Yandex.Metrica and not available to general public. We tend to use only `stateless` tests and avoid adding new `stateful` tests.

Each test can be one of two types: `.sql` and `.sh`. `.sql` test is the simple SQL script that is piped to `clickhouse-client --multiquery`. `.sh` test is a script that is run by itself.

To run all tests, use `dbms/tests/clickhouse-test` tool. Look `--help` for the list of possible options. You can simply run all tests or run subset of tests filtered by substring in test name: `./clickhouse-test substring`.

The most simple way to invoke functional tests is to copy `clickhouse-client` to `/usr/bin/`, run `clickhouse-server` and then run `./clickhouse-test` from its own directory.

To add new test, create a `.sql` or `.sh` file in `dbms/src/tests/queries/0_stateless` directory, check it manually and then generate `.reference` file in the following way: `clickhouse-client -n < 00000_test.sql > 00000_test.reference` or `./00000_test.sh > ./00000_test.reference`.

Tests should use (create, drop, etc) only tables in `test` database that is assumed to be created beforehand; also tests can use temporary tables.

If you want to use distributed queries in functional tests, you can leverage `remote` table function with `127.0.0.{1..2}` addresses for the server to query itself; or you can use predefined test clusters in server configuration file like `test_shard_localhost`.

Some tests are marked with `zookeeper`, `shard` or `long` in their names. `zookeeper` is for tests that are using ZooKeeper; `shard` is for tests that requires server to listen `127.0.0.*`; `long` is for tests that run slightly longer that one second.

Integration Tests

Integration tests allow to test ClickHouse in clustered configuration and ClickHouse interaction with other servers like MySQL, Postgres, MongoDB. They are useful to emulate network splits, packet drops, etc. These tests are run under Docker and create multiple containers with various software.

See `dbms/tests/integration/README.md` on how to run these tests.

Note that integration of ClickHouse with third-party drivers is not tested. Also we currently don't have integration tests with our JDBC and ODBC drivers.

Unit Tests

Unit tests are useful when you want to test not the ClickHouse as a whole, but a single isolated library or class. You can enable or disable build of tests with `ENABLE_TESTS` CMake option. Unit tests (and other test programs) are located in `tests` subdirectories across the code. To run unit tests, type `ninja test`. Some tests use `gtest`, but some are just programs that return non-zero exit code on test failure.

It's not necessarily to have unit tests if the code is already covered by functional tests (and functional tests are usually much more simple to use).

Performance Tests

Performance tests allow to measure and compare performance of some isolated part of ClickHouse on synthetic queries. Tests are located at `dbms/tests/performance`. Each test is represented by `.xml` file with description of test case. Tests are run with `clickhouse performance-test` tool (that is embedded in `clickhouse` binary). See `--help` for invocation.

Each test run one or miltiple queries (possibly with combinations of parameters) in a loop with some conditions for stop (like "maximum execution speed is not changing in three seconds") and measure some metrics about query performance (like "maximum execution speed"). Some tests can contain preconditions on preloaded test dataset.

If you want to improve performance of ClickHouse in some scenario, and if improvements can be observed on simple queries, it is highly recommended to write a performance test. It always makes sense to use `perf top` or other perf tools during your tests.

Performance tests are not run on per-commit basis. Results of performance tests are not collected and we compare them manually.

Test Tools And Scripts

Some programs in `tests` directory are not prepared tests, but are test tools. For example, for `Lexer` there is a tool `dbms/src/Parsers/tests/lexer` that just do tokenization of stdin and writes colorized result to stdout. You can use these kind of tools as a code examples and for exploration and manual testing.

You can also place pair of files `.sh` and `.reference` along with the tool to run it on some predefined input - then script result can be compared to `.reference` file. There kind of tests are not automated.

Miscellanous Tests

There are tests for external dictionaries located at `dbms/tests/external_dictionaries` and for machine learned models in `dbms/tests/external_models`. These tests are not updated and must be transferred to integration tests.

There is separate test for quorum inserts. This test run ClickHouse cluster on separate servers and emulate various failure cases: network split, packet drop (between ClickHouse nodes, between ClickHouse and ZooKeeper, between ClickHouse server and client, etc.), `kill -9`, `kill -STOP` and `kill -CONT`, like Jepsen [https://aphyr.com/tags/Jepsen]. Then the test checks that all acknowledged inserts was written and all rejected inserts was not.

Quorum test was written by separate team before ClickHouse was open-sourced. This team no longer work with ClickHouse. Test was accidentially written in Java. For these reasons, quorum test must be rewritten and moved to integration tests.

Manual Testing

When you develop a new feature, it is reasonable to also test it manually. You can do it with the following steps:

Build ClickHouse. Run ClickHouse from the terminal: change directory to `dbms/src/programs/clickhouse-server` and run it with `./clickhouse-server`. It will use configuration (`config.xml`, `users.xml` and files within `config.d` and `users.d` directories) from the current directory by default. To connect to ClickHouse server, run `dbms/src/programs/clickhouse-client/clickhouse-client`.

Note that all clickhouse tools (server, client, etc) are just symlinks to a single binary named `clickhouse`. You can find this binary at `dbms/src/programs/clickhouse`. All tools can also be invoked as `clickhouse tool` instead of `clickhouse-tool`.

Alternatively you can install ClickHouse package: either stable release from Yandex repository or you can build package for yourself with `./release` in ClickHouse sources root. Then start the server with `sudo service clickhouse-server start` (or stop to stop the server). Look for logs at `/etc/clickhouse-server/clickhouse-server.log`.

When ClickHouse is already installed on your system, you can build a new `clickhouse` binary and replace the existing binary:

```
sudo service clickhouse-server stop
sudo cp ./clickhouse /usr/bin/
sudo service clickhouse-server start
```

Also you can stop system clickhouse-server and run your own with the same configuration but with logging to terminal:

```
sudo service clickhouse-server stop
sudo -u clickhouse /usr/bin/clickhouse server --config-file /etc/clickhouse-server/config.xml
```

Example with gdb:

```
sudo -u clickhouse gdb --args /usr/bin/clickhouse server --config-file /etc/clickhouse-server/config.xml
```

If the system clickhouse-server is already running and you don't want to stop it, you can change port numbers in your `config.xml` (or override them in a file in `config.d` directory), provide appropriate data path, and run it.

`clickhouse` binary has almost no dependencies and works across wide range of Linux distributions. To quick and dirty test your changes on a server, you can simply `scp` your fresh built `clickhouse` binary to your server and then run it as in examples above.

Testing Environment

Before publishing release as stable we deploy it on testing environment. Testing environment is a cluster that process 1/39 part of Yandex.Metrica [https://metrica.yandex.com/] data. We share our testing environment with Yandex.Metrica team. ClickHouse is upgraded without downtime on top of existing data. We look at first that data is processed successfully without lagging from realtime, the replication continue to work and there is no issues visible to Yandex.Metrica team. First check can be done in the following way:

```
SELECT hostName() AS h, any(version()), any(uptime()), max(UTCEventTime), count() FROM remote('example01-01-{1..3}t', merge, hits) WHERE EventDate >= today() - 2 GROUP BY h ORDER BY h;
```

In some cases we also deploy to testing environment of our friend teams in Yandex: Market, Cloud, etc. Also we have some hardware servers that are used for development purposes.

Load Testing

After deploying to testing environment we run load testing with queries from production cluster. This is done manually.

Make sure you have enabled `query_log` on your production cluster.

Collect query log for a day or more:

```
clickhouse-client --query="SELECT DISTINCT query FROM system.query_log WHERE event_date = today() AND query LIKE '%ym:%' AND query NOT LIKE '%system.query_log%' AND type = 2 AND is_initial_query" > queries.tsv
```

This is a way complicated example. `type = 2` will filter queries that are executed successfully. `query LIKE '%ym:%'` is to select relevant queries from Yandex.Metrica. `is_initial_query` is to select only queries that are initiated by client, not by ClickHouse itself (as parts of distributed query processing).

`scp` this log to your testing cluster and run it as following:

```
clickhouse benchmark --concurrency 16 < queries.tsv
```

(probably you also want to specify a `--user`)

Then leave it for a night or weekend and go take a rest.

You should check that `clickhouse-server` doesn't crash, memory footprint is bounded and performance not degrading over time.

Precise query execution timings are not recorded and not compared due to high variability of queries and environment.

Build Tests

Build tests allow to check that build is not broken on various alternative configurations and on some foreign systems. Tests are located at `ci` directory. They run build from source inside Docker, Vagrant, and sometimes with `qemu-user-static` inside Docker. These tests are under development and test runs are not automated.

Motivation:

Normally we release and run all tests on a single variant of ClickHouse build. But there are alternative build variants that are not thoroughly tested. Examples:

- build on FreeBSD;
- build on Debian with libraries from system packages;
- build with shared linking of libraries;
- build on AArch64 platform.

For example, build with system packages is bad practice, because we cannot guarantee what exact version of packages a system will have. But this is really needed by Debian maintainers. For this reason we at least have to support this variant of build. Another example: shared linking is a common source of trouble, but it is needed for some enthusiasts.

Though we cannot run all tests on all variant of builds, we want to check at least that various build variants are not broken. For this purpose we use build tests.

Testing For Protocol Compatibility

When we extend ClickHouse network protocol, we test manually that old clickhouse-client works with new clickhouse-server and new clickhouse-client works with old clickhouse-server (simply by running binaries from corresponding packages).

Help From The Compiler

Main ClickHouse code (that is located in `dbms` directory) is built with `-Wall -Wextra -Werror` and with some additional enabled warnings. Although these options are not enabled for third-party libraries.

Clang has even more useful warnings - you can look for them with `-Weverything` and pick something to default build.

For production builds, gcc is used (it still generates slightly more efficient code than clang). For development, clang is usually more convenient to use. You can build on your own machine with debug mode (to save battery of your laptop), but please note that compiler is able to generate more warnings with `-O3` due to better control flow and inter-procedure analysis. When building with clang, `libc++` is used instead of `libstdc++` and when building with debug mode, debug version of `libc++` is used that allows to catch more errors at runtime.

Sanitizers

Address sanitizer. We run functional tests under ASan on per-commit basis.

Valgrind (Memcheck). We run functional tests under Valgrind overnight. It takes multiple hours. Currently there is one known false positive in `re2` library, see this article [https://research.swtch.com/sparse].

Thread sanitizer. We run functional tests under TSan. ClickHouse must pass all tests. Run under TSan is not automated and performed only occasionally.

Memory sanitizer. Currently we still don't use MSan.

Undefined behaviour sanitizer. We still don't use UBSan. The only thing to fix is unaligned placement of structs in Arena during aggregation. This is totally fine, we only have to force alignment under UBSan.

Debug allocator. You can enable debug version of `tcmalloc` with `DEBUG_TCMALLOC` CMake option. We run tests with debug allocator on per-commit basis.

You will find some additional details in `dbms/tests/instructions/sanitizers.txt`.

Fuzzing

As of July 2018 we don't use fuzzing.

Security Audit

People from Yandex Cloud department do some basic overview of ClickHouse capabilities from the security standpoint.

Static Analyzers

We use static analyzers only occasionally. We have evaluated `clang-tidy` , `Coverity` , `cppcheck` , `PVS-Studio` , `tscancode` . You will find instructions for usage in `dbms/tests/instructions/` directory. Also you can read the article in russian [https://habr.com/company/yandex/blog/342018/].

If you use `CLion` as an IDE, you can leverage some `clang-tidy` checks out of the box.

Hardening

`FORTIFY_SOURCE` is used by default. It is almost useless, but still makes sense in rare cases and we don't disable it.

Code Style

Code style rules are described here [https://clickhouse.yandex/docs/en/development/style/].

To check for some common style violations, you can use `utils/check-style` script.

To force proper style of your code, you can use `clang-format` . File `.clang-format` is located at the sources root. It mostly corresponding with our actual code style. But it's not recommended to apply `clang-format` to existing files because it makes formatting worse. You can use `clang-format-diff` tool that you can find in clang source repository.

Alternatively you can try `uncrustify` tool to reformat your code. Configuration is in `uncrustify.cfg` in the sources root. It is less tested than `clang-format` .

`CLion` has its own code formatter that has to be tuned for our code style.

Metrica B2B Tests

Each ClickHouse release is tested with Yandex Metrica and AppMetrica engines. Testing and stable versions of ClickHouse are deployed on VMs and run with a small copy of Metrica engine that is processing fixed sample of input data. Then results of two instances of Metrica engine are compared together.

These tests are automated by separate team. Due to high number of moving parts, tests are fail most of the time by completely unrelated reasons, that are very difficult to figure out. Most likely these tests have negative value for us. Nevertheless these tests was proved to be useful in about one or two times out of hundreds.

Test Coverage

As of July 2018 we don't track test coverage.

Test Automation

We run tests with Travis CI (available for general public) and Jenkins (available inside Yandex).

In Travis CI due to limit on time and computational power we can afford only subset of functional tests that are run with limited build of ClickHouse (debug version with cut off most of libraries). In about half of runs it still fails to finish in 50

minutes timeout. The only advantage - test results are visible for all external contributors.

In Jenkins we run functional tests for each commit and for each pull request from trusted users; the same under ASan; we also run quorum tests, dictionary tests, Metrica B2B tests. We use Jenkins to prepare and publish releases. Worth to note that we are not happy with Jenkins at all.

One of our goals is to provide reliable testing infrastructure that will be available to community.

Roadmap

Q4 2018

- JOIN syntax compatible with SQL standard:
  - Mutliple `JOIN` s in single `SELECT`
- Protobuf and Parquet input and output formats

Q1 2019

- Import/export from HDFS and S3
- Lower metadata size in ZooKeeper
- Adaptive index granularity for MergeTree engine family

Q2 2019

- JOIN execution improvements:
  - Distributed join not limited by memory
- Resource pools for more precise distribution of cluster capacity between users

Q3 2019

- Fine-grained authorization
- Integration with external authentication services

ClickHouse release 18.14.15, 2018-11-21

Bug fixes:

- The size of memory chunk was overestimated while deserializing the column of type `Array(String)` that leads to "Memory limit exceeded" errors. The issue appeared in version 18.12.13. #3589 [https://github.com/yandex/ClickHouse/issues/3589]

ClickHouse release 18.14.14, 2018-11-20

Bug fixes:

- Fixed `ON CLUSTER` queries when cluster configured as secure (flag `<secure>`). #3599 [https://github.com/yandex/ClickHouse/pull/3599]

Build changes:

- Fixed problems (llvm-7 from system, macos) #3582 [https://github.com/yandex/ClickHouse/pull/3582]

ClickHouse release 18.14.11, 2018-10-29

Bug fixes:

- Fixed the error `Block structure mismatch in UNION stream: different number of columns` in LIMIT queries. #2156 [https://github.com/yandex/ClickHouse/issues/2156]
- Fixed errors when merging data in tables containing arrays inside Nested structures. #3397 [https://github.com/yandex/ClickHouse/pull/3397]
- Fixed incorrect query results if the `merge_tree_uniform_read_distribution` setting is disabled (it is enabled by default). #3429 [https://github.com/yandex/ClickHouse/pull/3429]
- Fixed an error on inserts to a Distributed table in Native format. #3411 [https://github.com/yandex/ClickHouse/issues/3411]

ClickHouse release 18.14.10, 2018-10-23

- The `compile_expressions` setting (JIT compilation of expressions) is disabled by default. #3410 [https://github.com/yandex/ClickHouse/pull/3410]
- The `enable_optimize_predicate_expression` setting is disabled by default.

ClickHouse release 18.14.9, 2018-10-16

New features:

- The `WITH CUBE` modifier for `GROUP BY` (the alternative syntax `GROUP BY CUBE(...)` is also available). #3172 [https://github.com/yandex/ClickHouse/pull/3172]
- Added the `formatDateTime` function. Alexandr Krasheninnikov [https://github.com/yandex/ClickHouse/pull/2770]
- Added the `JDBC` table engine and `jdbc` table function (requires installing clickhouse-jdbc-bridge). Alexandr Krasheninnikov [https://github.com/yandex/ClickHouse/pull/3210]
- Added functions for working with the ISO week number: `toISOWeek`, `toISOYear`, `toStartOfISOYear`, and `toDayOfYear`. #3146 [https://github.com/yandex/ClickHouse/pull/3146]
- Now you can use `Nullable` columns for `MySQL` and `ODBC` tables. #3362 [https://github.com/yandex/ClickHouse/pull/3362]
- Nested data structures can be read as nested objects in `JSONEachRow` format. Added the `input_format_import_nested_json` setting. Veloman Yunkan [https://github.com/yandex/ClickHouse/pull/3144]
- Parallel processing is available for many `MATERIALIZED VIEW` s when inserting data. See the `parallel_view_processing` setting. Marek Vavruša [https://github.com/yandex/ClickHouse/pull/3208]
- Added the `SYSTEM FLUSH LOGS` query (forced log flushes to system tables such as `query_log`) #3321 [https://github.com/yandex/ClickHouse/pull/3321]
- Now you can use pre-defined `database` and `table` macros when declaring `Replicated` tables. #3251 [https://github.com/yandex/ClickHouse/pull/3251]
- Added the ability to read `Decimal` type values in engineering notation (indicating powers of ten). #3153 [https://github.com/yandex/ClickHouse/pull/3153]

Experimental features:

- Optimization of the GROUP BY clause for `LowCardinality data types.` #3138 [https://github.com/yandex/ClickHouse/pull/3138]
- Optimized calculation of expressions for `LowCardinality data types.` #3200 [https://github.com/yandex/ClickHouse/pull/3200]

Improvements:

- Significantly reduced memory consumption for requests with `ORDER BY` and `LIMIT` . See the `max_bytes_before_remerge_sort` setting. #3205 [https://github.com/yandex/ClickHouse/pull/3205]
- In the absence of `JOIN` (`LEFT` , `INNER` , ...), `INNER JOIN` is assumed. #3147 [https://github.com/yandex/ClickHouse/pull/3147]
- Qualified asterisks work correctly in queries with `JOIN` . Winter Zhang [https://github.com/yandex/ClickHouse/pull/3202]
- The `ODBC` table engine correctly chooses the method for quoting identifiers in the SQL dialect of a remote database. Alexandr Krasheninnikov [https://github.com/yandex/ClickHouse/pull/3210]
- The `compile_expressions` setting (JIT compilation of expressions) is enabled by default.
- Fixed behavior for simultaneous DROP DATABASE/TABLE IF EXISTS and CREATE DATABASE/TABLE IF NOT EXISTS. Previously, a `CREATE DATABASE ... IF NOT EXISTS` query could return the error message "File ... already exists", and the `CREATE TABLE ... IF NOT EXISTS` and `DROP TABLE IF EXISTS` queries could return `Table ... is creating or attaching right now` . #3101 [https://github.com/yandex/ClickHouse/pull/3101]
- LIKE and IN expressions with a constant right half are passed to the remote server when querying from MySQL or ODBC tables. #3182 [https://github.com/yandex/ClickHouse/pull/3182]
- Comparisons with constant expressions in a WHERE clause are passed to the remote server when querying from MySQL and ODBC tables. Previously, only comparisons with constants were passed. #3182 [https://github.com/yandex/ClickHouse/pull/3182]
- Correct calculation of row width in the terminal for `Pretty` formats, including strings with hieroglyphs. Amos Bird [https://github.com/yandex/ClickHouse/pull/3257].
- `ON CLUSTER` can be specified for `ALTER UPDATE` queries.
- Improved performance for reading data in `JSONEachRow` format. #3332 [https://github.com/yandex/ClickHouse/pull/3332]
- Added synonyms for the `LENGTH` and `CHARACTER_LENGTH` functions for compatibility. The `CONCAT` function is no longer case-sensitive. #3306 [https://github.com/yandex/ClickHouse/pull/3306]
- Added the `TIMESTAMP` synonym for the `DateTime` type. #3390 [https://github.com/yandex/ClickHouse/pull/3390]
- There is always space reserved for query_id in the server logs, even if the log line is not related to a query. This makes it easier to parse server text logs with third-party tools.
- Memory consumption by a query is logged when it exceeds the next level of an integer number of gigabytes.#3205 [https://github.com/yandex/ClickHouse/pull/3205]
- Added compatibility mode for the case when the client library that uses the Native protocol sends fewer columns by mistake than the server expects for the INSERT query. This scenario was possible when using the clickhouse-cpp library. Previously, this scenario caused the server to crash. #3171 [https://github.com/yandex/ClickHouse/pull/3171]
- In a user-defined WHERE expression in `clickhouse-copier` , you can now use a `partition_key` alias (for additional filtering by source table partition). This is useful if the partitioning scheme changes during copying, but only changes slightly. #3166 [https://github.com/yandex/ClickHouse/pull/3166]
- The workflow of the `Kafka` engine has been moved to a background thread pool in order to automatically reduce the speed of data reading at high loads. Marek Vavruša [https://github.com/yandex/ClickHouse/pull/3215].
- Support for reading `Tuple` and `Nested` values of structures like `struct` in the `Cap'n'Proto format` . Marek Vavruša [https://github.com/yandex/ClickHouse/pull/3216]
- The list of top-level domains for the `firstSignificantSubdomain` function now includes the domain `biz` . decaseal [https://github.com/yandex/ClickHouse/pull/3219]
- In the configuration of external dictionaries, `null_value` is interpreted as the value of the default data type. #3330 [https://github.com/yandex/ClickHouse/pull/3330]
- Support for the `intDiv` and `intDivOrZero` functions for `Decimal` . b48402e8

[https://github.com/yandex/ClickHouse/commit/b48402e8712e2b9b151e0eef8193811d433a1264]

- Support for the `Date`, `DateTime`, `UUID`, and `Decimal` types as a key for the `sumMap` aggregate function. #3281 [https://github.com/yandex/ClickHouse/pull/3281]
- Support for the `Decimal` data type in external dictionaries. #3324 [https://github.com/yandex/ClickHouse/pull/3324]
- Support for the `Decimal` data type in `SummingMergeTree` tables. #3348 [https://github.com/yandex/ClickHouse/pull/3348]
- Added specializations for `UUID` in `if`. #3366 [https://github.com/yandex/ClickHouse/pull/3366]
- Reduced the number of `open` and `close` system calls when reading from a `MergeTree table`. #3283 [https://github.com/yandex/ClickHouse/pull/3283]
- A `TRUNCATE TABLE` query can be executed on any replica (the query is passed to the leader replica).Kirill Shvakov [https://github.com/yandex/ClickHouse/pull/3375]

Bug fixes:

- Fixed an issue with `Dictionary` tables for `range_hashed` dictionaries. This error occurred in version 18.12.17. #1702 [https://github.com/yandex/ClickHouse/pull/1702]
- Fixed an error when loading `range_hashed` dictionaries (the message `Unsupported type Nullable (...)`). This error occurred in version 18.12.17. #3362 [https://github.com/yandex/ClickHouse/pull/3362]
- Fixed errors in the `pointInPolygon` function due to the accumulation of inaccurate calculations for polygons with a large number of vertices located close to each other. #3331 [https://github.com/yandex/ClickHouse/pull/3331] #3341 [https://github.com/yandex/ClickHouse/pull/3341]
- If after merging data parts, the checksum for the resulting part differs from the result of the same merge in another replica, the result of the merge is deleted and the data part is downloaded from the other replica (this is the correct behavior). But after downloading the data part, it couldn't be added to the working set because of an error that the part already exists (because the data part was deleted with some delay after the merge). This led to cyclical attempts to download the same data. #3194 [https://github.com/yandex/ClickHouse/pull/3194]
- Fixed incorrect calculation of total memory consumption by queries (because of incorrect calculation, the `max_memory_usage_for_all_queries` setting worked incorrectly and the `MemoryTracking` metric had an incorrect value). This error occurred in version 18.12.13. Marek Vavruša [https://github.com/yandex/ClickHouse/pull/3344]
- Fixed the functionality of `CREATE TABLE ... ON CLUSTER ... AS SELECT ...` This error occurred in version 18.12.13. #3247 [https://github.com/yandex/ClickHouse/pull/3247]
- Fixed unnecessary preparation of data structures for `JOIN`s on the server that initiates the request if the `JOIN` is only performed on remote servers. #3340 [https://github.com/yandex/ClickHouse/pull/3340]
- Fixed bugs in the `Kafka` engine: deadlocks after exceptions when starting to read data, and locks upon completion Marek Vavruša [https://github.com/yandex/ClickHouse/pull/3215].
- For `Kafka` tables, the optional `schema` parameter was not passed (the schema of the `Cap'n'Proto` format). Vojtech Splichal [https://github.com/yandex/ClickHouse/pull/3150]
- If the ensemble of ZooKeeper servers has servers that accept the connection but then immediately close it instead of responding to the handshake, ClickHouse chooses to connect another server. Previously, this produced the error `Cannot read all data. Bytes read: 0. Bytes expected: 4.` and the server couldn't start. 8218cf3a [https://github.com/yandex/ClickHouse/commit/8218cf3a5f39a43401953769d6d12a0bb8d29da9]
- If the ensemble of ZooKeeper servers contains servers for which the DNS query returns an error, these servers are ignored. 17b8e209 [https://github.com/yandex/ClickHouse/commit/17b8e209221061325ad7ba0539f03c6e65f87f29]
- Fixed type conversion between `Date` and `DateTime` when inserting data in the `VALUES` format (if `input_format_values_interpret_expressions = 1`). Previously, the conversion was performed between the numerical value of the number of days in Unix Epoch time and the Unix timestamp, which led to unexpected results. #3229 [https://github.com/yandex/ClickHouse/pull/3229]
- Corrected type conversion between `Decimal` and integer numbers. #3211

[https://github.com/yandex/ClickHouse/pull/3211]

- Fixed errors in the `enable_optimize_predicate_expression` setting. Winter Zhang [https://github.com/yandex/ClickHouse/pull/3231]
- Fixed a parsing error in CSV format with floating-point numbers if a non-default CSV separator is used, such as `;` #3155 [https://github.com/yandex/ClickHouse/pull/3155]
- Fixed the `arrayCumSumNonNegative` function (it does not accumulate negative values if the accumulator is less than zero). Aleksey Studnev [https://github.com/yandex/ClickHouse/pull/3163]
- Fixed how `Merge` tables work on top of `Distributed` tables when using `PREWHERE`. #3165 [https://github.com/yandex/ClickHouse/pull/3165]
- Bug fixes in the `ALTER UPDATE` query.
- Fixed bugs in the `odbc` table function that appeared in version 18.12. #3197 [https://github.com/yandex/ClickHouse/pull/3197]
- Fixed the operation of aggregate functions with `StateArray` combinators. #3188 [https://github.com/yandex/ClickHouse/pull/3188]
- Fixed a crash when dividing a `Decimal` value by zero. 69dd6609 [https://github.com/yandex/ClickHouse/commit/69dd6609193beb4e7acd3e6ad216eca0ccfb8179]
- Fixed output of types for operations using `Decimal` and integer arguments. #3224 [https://github.com/yandex/ClickHouse/pull/3224]
- Fixed the segfault during `GROUP BY` on `Decimal128`. 3359ba06 [https://github.com/yandex/ClickHouse/commit/3359ba06c39fcd05bfdb87d6c64154819621e13a]
- The `log_query_threads` setting (logging information about each thread of query execution) now takes effect only if the `log_queries` option (logging information about queries) is set to 1. Since the `log_query_threads` option is enabled by default, information about threads was previously logged even if query logging was disabled. #3241 [https://github.com/yandex/ClickHouse/pull/3241]
- Fixed an error in the distributed operation of the quantiles aggregate function (the error message `Not found column quantile...`). 292a8855 [https://github.com/yandex/ClickHouse/commit/292a885533b8e3b41ce8993867069d14cbd5a664]
- Fixed the compatibility problem when working on a cluster of version 18.12.17 servers and older servers at the same time. For distributed queries with GROUP BY keys of both fixed and non-fixed length, if there was a large amount of data to aggregate, the returned data was not always fully aggregated (two different rows contained the same aggregation keys). #3254 [https://github.com/yandex/ClickHouse/pull/3254]
- Fixed handling of substitutions in `clickhouse-performance-test`, if the query contains only part of the substitutions declared in the test. #3263 [https://github.com/yandex/ClickHouse/pull/3263]
- Fixed an error when using `FINAL` with `PREWHERE`. #3298 [https://github.com/yandex/ClickHouse/pull/3298]
- Fixed an error when using `PREWHERE` over columns that were added during `ALTER`. #3298 [https://github.com/yandex/ClickHouse/pull/3298]
- Added a check for the absence of `arrayJoin` for `DEFAULT` and `MATERIALIZED` expressions. Previously, `arrayJoin` led to an error when inserting data. #3337 [https://github.com/yandex/ClickHouse/pull/3337]
- Added a check for the absence of `arrayJoin` in a `PREWHERE` clause. Previously, this led to messages like `Size ... doesn't match` or `Unknown compression method` when executing queries. #3357 [https://github.com/yandex/ClickHouse/pull/3357]
- Fixed segfault that could occur in rare cases after optimization that replaced AND chains from equality evaluations with the corresponding IN expression. liuyimin-bytedance [https://github.com/yandex/ClickHouse/pull/3339]
- Minor corrections to `clickhouse-benchmark`: previously, client information was not sent to the server; now the number of queries executed is calculated more accurately when shutting down and for limiting the number of iterations. #3351 [https://github.com/yandex/ClickHouse/pull/3351] #3352 [https://github.com/yandex/ClickHouse/pull/3352]

Backward incompatible changes:

- Removed the `allow_experimental_decimal_type` option. The `Decimal` data type is available for default use. #3329 [https://github.com/yandex/ClickHouse/pull/3329]

ClickHouse release 18.12.17, 2018-09-16

New features:

- `invalidate_query` (the ability to specify a query to check whether an external dictionary needs to be updated) is implemented for the `clickhouse` source. #3126 [https://github.com/yandex/ClickHouse/pull/3126]
- Added the ability to use `UInt*`, `Int*`, and `DateTime` data types (along with the `Date` type) as a `range_hashed` external dictionary key that defines the boundaries of ranges. Now `NULL` can be used to designate an open range. Vasily Nemkov [https://github.com/yandex/ClickHouse/pull/3123]
- The `Decimal` type now supports `var*` and `stddev*` aggregate functions. #3129 [https://github.com/yandex/ClickHouse/pull/3129]
- The `Decimal` type now supports mathematical functions (`exp`, `sin` and so on.) #3129 [https://github.com/yandex/ClickHouse/pull/3129]
- The `system.part_log` table now has the `partition_id` column. #3089 [https://github.com/yandex/ClickHouse/pull/3089]

Bug fixes:

- `Merge` now works correctly on `Distributed` tables. Winter Zhang [https://github.com/yandex/ClickHouse/pull/3159]
- Fixed incompatibility (unnecessary dependency on the `glibc` version) that made it impossible to run ClickHouse on `Ubuntu Precise` and older versions. The incompatibility arose in version 18.12.13. #3130 [https://github.com/yandex/ClickHouse/pull/3130]
- Fixed errors in the `enable_optimize_predicate_expression` setting. Winter Zhang [https://github.com/yandex/ClickHouse/pull/3107]
- Fixed a minor issue with backwards compatibility that appeared when working with a cluster of replicas on versions earlier than 18.12.13 and simultaneously creating a new replica of a table on a server with a newer version (shown in the message `Can not clone replica, because the ... updated to new ClickHouse version`, which is logical, but shouldn't happen). #3122 [https://github.com/yandex/ClickHouse/pull/3122]

Backward incompatible changes:

- The `enable_optimize_predicate_expression` option is enabled by default (which is rather optimistic). If query analysis errors occur that are related to searching for the column names, set `enable_optimize_predicate_expression` to 0. Winter Zhang [https://github.com/yandex/ClickHouse/pull/3107]

ClickHouse release 18.12.14, 2018-09-13

New features:

- Added support for `ALTER UPDATE` queries. #3035 [https://github.com/yandex/ClickHouse/pull/3035]
- Added the `allow_ddl` option, which restricts the user's access to DDL queries. #3104 [https://github.com/yandex/ClickHouse/pull/3104]
- Added the `min_merge_bytes_to_use_direct_io` option for `MergeTree` engines, which allows you to set a threshold for the total size of the merge (when above the threshold, data part files will be handled using O_DIRECT). #3117 [https://github.com/yandex/ClickHouse/pull/3117]
- The `system.merges` system table now contains the `partition_id` column. #3099 [https://github.com/yandex/ClickHouse/pull/3099]

Improvements

- If a data part remains unchanged during mutation, it isn't downloaded by replicas.#3103 [https://github.com/yandex/ClickHouse/pull/3103]
- Autocomplete is available for names of settings when working with `clickhouse-client`. #3106 [https://github.com/yandex/ClickHouse/pull/3106]

Bug fixes:

- Added a check for the sizes of arrays that are elements of `Nested` type fields when inserting. #3118 [https://github.com/yandex/ClickHouse/pull/3118]
- Fixed an error updating external dictionaries with the `ODBC` source and `hashed` storage. This error occurred in version 18.12.13.
- Fixed a crash when creating a temporary table from a query with an `IN` condition. Winter Zhang [https://github.com/yandex/ClickHouse/pull/3098]
- Fixed an error in aggregate functions for arrays that can have `NULL` elements. Winter Zhang [https://github.com/yandex/ClickHouse/pull/3097]

ClickHouse release 18.12.13, 2018-09-10

New features:

- Added the `DECIMAL(digits, scale)` data type (`Decimal32(scale)`, `Decimal64(scale)`, `Decimal128(scale)`). To enable it, use the setting `allow_experimental_decimal_type`. #2846 [https://github.com/yandex/ClickHouse/pull/2846] #2970 [https://github.com/yandex/ClickHouse/pull/2970] #3008 [https://github.com/yandex/ClickHouse/pull/3008] #3047 [https://github.com/yandex/ClickHouse/pull/3047]
- New `WITH ROLLUP` modifier for `GROUP BY` (alternative syntax: `GROUP BY ROLLUP(...)`). #2948 [https://github.com/yandex/ClickHouse/pull/2948]
- In requests with JOIN, the star character expands to a list of columns in all tables, in compliance with the SQL standard. You can restore the old behavior by setting `asterisk_left_columns_only` to 1 on the user configuration level. Winter Zhang [https://github.com/yandex/ClickHouse/pull/2787]
- Added support for JOIN with table functions. Winter Zhang [https://github.com/yandex/ClickHouse/pull/2907]
- Autocomplete by pressing Tab in clickhouse-client. Sergey Shcherbin [https://github.com/yandex/ClickHouse/pull/2447]
- Ctrl+C in clickhouse-client clears a query that was entered. #2877 [https://github.com/yandex/ClickHouse/pull/2877]
- Added the `join_default_strictness` setting (values: `"`, `'any'`, `'all'`). This allows you to not specify `ANY` or `ALL` for `JOIN`. #2982 [https://github.com/yandex/ClickHouse/pull/2982]
- Each line of the server log related to query processing shows the query ID.#2482 [https://github.com/yandex/ClickHouse/pull/2482]
- Now you can get query execution logs in clickhouse-client (use the `send_logs_level` setting). With distributed query processing, logs are cascaded from all the servers. #2482 [https://github.com/yandex/ClickHouse/pull/2482]
- The `system.query_log` and `system.processes` (`SHOW PROCESSLIST`) tables now have information about all changed settings when you run a query (the nested structure of the `Settings` data). Added the `log_query_settings` setting. #2482 [https://github.com/yandex/ClickHouse/pull/2482]
- The `system.query_log` and `system.processes` tables now show information about the number of threads that are participating in query execution (see the `thread_numbers` column). #2482 [https://github.com/yandex/ClickHouse/pull/2482]
- Added `ProfileEvents` counters that measure the time spent on reading and writing over the network and reading and writing to disk, the number of network errors, and the time spent waiting when network bandwidth is limited.

#2482 [https://github.com/yandex/ClickHouse/pull/2482]

- Added `ProfileEvents` counters that contain the system metrics from rusage (you can use them to get information about CPU usage in userspace and the kernel, page faults, and context switches), as well as taskstats metrics (use these to obtain information about I/O wait time, CPU wait time, and the amount of data read and recorded, both with and without page cache). #2482 [https://github.com/yandex/ClickHouse/pull/2482]
- The `ProfileEvents` counters are applied globally and for each query, as well as for each query execution thread, which allows you to profile resource consumption by query in detail. #2482 [https://github.com/yandex/ClickHouse/pull/2482]
- Added the `system.query_thread_log` table, which contains information about each query execution thread. Added the `log_query_threads` setting. #2482 [https://github.com/yandex/ClickHouse/pull/2482]
- The `system.metrics` and `system.events` tables now have built-in documentation. #3016 [https://github.com/yandex/ClickHouse/pull/3016]
- Added the `arrayEnumerateDense` function. Amos Bird [https://github.com/yandex/ClickHouse/pull/2975]
- Added the `arrayCumSumNonNegative` and `arrayDifference` functions. Aleksey Studnev [https://github.com/yandex/ClickHouse/pull/2942]
- Added the `retention` aggregate function. Sundy Li [https://github.com/yandex/ClickHouse/pull/2887]
- Now you can add (merge) states of aggregate functions by using the plus operator, and multiply the states of aggregate functions by a nonnegative constant. #3062 [https://github.com/yandex/ClickHouse/pull/3062] #3034 [https://github.com/yandex/ClickHouse/pull/3034]
- Tables in the MergeTree family now have the virtual column `_partition_id`. #3089 [https://github.com/yandex/ClickHouse/pull/3089]

Experimental features:

- Added the `LowCardinality(T)` data type. This data type automatically creates a local dictionary of values and allows data processing without unpacking the dictionary. #2830 [https://github.com/yandex/ClickHouse/pull/2830]
- Added a cache of JIT-compiled functions and a counter for the number of uses before compiling. To JIT compile expressions, enable the `compile_expressions` setting. #2990 [https://github.com/yandex/ClickHouse/pull/2990] #3077 [https://github.com/yandex/ClickHouse/pull/3077]

Improvements:

- Fixed the problem with unlimited accumulation of the replication log when there are abandoned replicas. Added an effective recovery mode for replicas with a long lag.
- Improved performance of `GROUP BY` with multiple aggregation fields when one of them is string and the others are fixed length.
- Improved performance when using `PREWHERE` and with implicit transfer of expressions in `PREWHERE`.
- Improved parsing performance for text formats (`CSV`, `TSV`). Amos Bird [https://github.com/yandex/ClickHouse/pull/2977] #2980 [https://github.com/yandex/ClickHouse/pull/2980]
- Improved performance of reading strings and arrays in binary formats. Amos Bird [https://github.com/yandex/ClickHouse/pull/2955]
- Increased performance and reduced memory consumption for queries to `system.tables` and `system.columns` when there is a very large number of tables on a single server. #2953 [https://github.com/yandex/ClickHouse/pull/2953]
- Fixed a performance problem in the case of a large stream of queries that result in an error (the `_dl_addr` function is visible in `perf top`, but the server isn't using much CPU). #2938 [https://github.com/yandex/ClickHouse/pull/2938]
- Conditions are cast into the View (when `enable_optimize_predicate_expression` is enabled). Winter Zhang [https://github.com/yandex/ClickHouse/pull/2907]
- Improvements to the functionality for the `UUID` data type. #3074 [https://github.com/yandex/ClickHouse/pull/3074] #2985 [https://github.com/yandex/ClickHouse/pull/2985]

- The `UUID` data type is supported in The-Alchemist dictionaries. #2822 [https://github.com/yandex/ClickHouse/pull/2822]
- The `visitParamExtractRaw` function works correctly with nested structures. Winter Zhang [https://github.com/yandex/ClickHouse/pull/2974]
- When the `input_format_skip_unknown_fields` setting is enabled, object fields in `JSONEachRow` format are skipped correctly. BlahGeek [https://github.com/yandex/ClickHouse/pull/2958]
- For a `CASE` expression with conditions, you can now omit `ELSE`, which is equivalent to `ELSE NULL`. #2920 [https://github.com/yandex/ClickHouse/pull/2920]
- The operation timeout can now be configured when working with ZooKeeper. urykhy [https://github.com/yandex/ClickHouse/pull/2971]
- You can specify an offset for `LIMIT n, m` as `LIMIT n OFFSET m`. #2840 [https://github.com/yandex/ClickHouse/pull/2840]
- You can use the `SELECT TOP n` syntax as an alternative for `LIMIT`. #2840 [https://github.com/yandex/ClickHouse/pull/2840]
- Increased the size of the queue to write to system tables, so the `SystemLog parameter queue is full` error doesn't happen as often.
- The `windowFunnel` aggregate function now supports events that meet multiple conditions. Amos Bird [https://github.com/yandex/ClickHouse/pull/2801]
- Duplicate columns can be used in a `USING` clause for `JOIN`. #3006 [https://github.com/yandex/ClickHouse/pull/3006]
- `Pretty` formats now have a limit on column alignment by width. Use the `output_format_pretty_max_column_pad_width` setting. If a value is wider, it will still be displayed in its entirety, but the other cells in the table will not be too wide. #3003 [https://github.com/yandex/ClickHouse/pull/3003]
- The `odbc` table function now allows you to specify the database/schema name. Amos Bird [https://github.com/yandex/ClickHouse/pull/2885]
- Added the ability to use a username specified in the `clickhouse-client` config file. Vladimir Kozbin [https://github.com/yandex/ClickHouse/pull/2909]
- The `ZooKeeperExceptions` counter has been split into three counters: `ZooKeeperUserExceptions`, `ZooKeeperHardwareExceptions`, and `ZooKeeperOtherExceptions`.
- `ALTER DELETE` queries work for materialized views.
- Added randomization when running the cleanup thread periodically for `ReplicatedMergeTree` tables in order to avoid periodic load spikes when there are a very large number of `ReplicatedMergeTree` tables.
- Support for `ATTACH TABLE ... ON CLUSTER` queries. #3025 [https://github.com/yandex/ClickHouse/pull/3025]

Bug fixes:

- Fixed an issue with `Dictionary` tables (throws the `Size of offsets doesn't match size of column` or `Unknown compression method` exception). This bug appeared in version 18.10.3. #2913 [https://github.com/yandex/ClickHouse/issues/2913]
- Fixed a bug when merging `CollapsingMergeTree` tables if one of the data parts is empty (these parts are formed during merge or `ALTER DELETE` if all data was deleted), and the `vertical` algorithm was used for the merge. #3049 [https://github.com/yandex/ClickHouse/pull/3049]
- Fixed a race condition during `DROP` or `TRUNCATE` for `Memory` tables with a simultaneous `SELECT`, which could lead to server crashes. This bug appeared in version 1.1.54388. #3038 [https://github.com/yandex/ClickHouse/pull/3038]
- Fixed the possibility of data loss when inserting in `Replicated` tables if the `Session is expired` error is returned (data loss can be detected by the `ReplicatedDataLoss` metric). This error occurred in version 1.1.54378. #2939 [https://github.com/yandex/ClickHouse/pull/2939] #2949 [https://github.com/yandex/ClickHouse/pull/2949] #2964 [https://github.com/yandex/ClickHouse/pull/2964]

- Fixed a segfault during `JOIN ... ON`. #3000 [https://github.com/yandex/ClickHouse/pull/3000]
- Fixed the error searching column names when the `WHERE` expression consists entirely of a qualified column name, such as `WHERE table.column`. #2994 [https://github.com/yandex/ClickHouse/pull/2994]
- Fixed the "Not found column" error that occurred when executing distributed queries if a single column consisting of an IN expression with a subquery is requested from a remote server. #3087 [https://github.com/yandex/ClickHouse/pull/3087]
- Fixed the `Block structure mismatch in UNION stream: different number of columns` error that occurred for distributed queries if one of the shards is local and the other is not, and optimization of the move to `PREWHERE` is triggered. #2226 [https://github.com/yandex/ClickHouse/pull/2226] #3037 [https://github.com/yandex/ClickHouse/pull/3037] #3055 [https://github.com/yandex/ClickHouse/pull/3055] #3065 [https://github.com/yandex/ClickHouse/pull/3065] #3073 [https://github.com/yandex/ClickHouse/pull/3073] #3090 [https://github.com/yandex/ClickHouse/pull/3090] #3093 [https://github.com/yandex/ClickHouse/pull/3093]
- Fixed the `pointInPolygon` function for certain cases of non-convex polygons. #2910 [https://github.com/yandex/ClickHouse/pull/2910]
- Fixed the incorrect result when comparing `nan` with integers. #3024 [https://github.com/yandex/ClickHouse/pull/3024]
- Fixed an error in the `zlib-ng` library that could lead to segfault in rare cases. #2854 [https://github.com/yandex/ClickHouse/pull/2854]
- Fixed a memory leak when inserting into a table with `AggregateFunction` columns, if the state of the aggregate function is not simple (allocates memory separately), and if a single insertion request results in multiple small blocks. #3084 [https://github.com/yandex/ClickHouse/pull/3084]
- Fixed a race condition when creating and deleting the same `Buffer` or `MergeTree` table simultaneously.
- Fixed the possibility of a segfault when comparing tuples made up of certain non-trivial types, such as tuples.#2989 [https://github.com/yandex/ClickHouse/pull/2989]
- Fixed the possibility of a segfault when running certain `ON CLUSTER` queries. Winter Zhang [https://github.com/yandex/ClickHouse/pull/2960]
- Fixed an error in the `arrayDistinct` function for `Nullable` array elements. #2845 [https://github.com/yandex/ClickHouse/pull/2845] #2937 [https://github.com/yandex/ClickHouse/pull/2937]
- The `enable_optimize_predicate_expression` option now correctly supports cases with `SELECT *`. Winter Zhang [https://github.com/yandex/ClickHouse/pull/2929]
- Fixed the segfault when re-initializing the ZooKeeper session.#2917 [https://github.com/yandex/ClickHouse/pull/2917]
- Fixed potential blocking when working with ZooKeeper.
- Fixed incorrect code for adding nested data structures in a `SummingMergeTree`.
- When allocating memory for states of aggregate functions, alignment is correctly taken into account, which makes it possible to use operations that require alignment when implementing states of aggregate functions. chenxing-xc [https://github.com/yandex/ClickHouse/pull/2808]

Security fix:

- Safe use of ODBC data sources. Interaction with ODBC drivers uses a separate `clickhouse-odbc-bridge` process. Errors in third-party ODBC drivers no longer cause problems with server stability or vulnerabilities. #2828 [https://github.com/yandex/ClickHouse/pull/2828] #2879 [https://github.com/yandex/ClickHouse/pull/2879] #2886 [https://github.com/yandex/ClickHouse/pull/2886] #2893 [https://github.com/yandex/ClickHouse/pull/2893] #2921 [https://github.com/yandex/ClickHouse/pull/2921]
- Fixed incorrect validation of the file path in the `catBoostPool` table function. #2894 [https://github.com/yandex/ClickHouse/pull/2894]
- The contents of system tables (`tables`, `databases`, `parts`, `columns`, `parts_columns`, `merges`, `mutations`,

`replicas`, and `replication_queue`) are filtered according to the user's configured access to databases (`allow_databases`). Winter Zhang [https://github.com/yandex/ClickHouse/pull/2856]

Backward incompatible changes:

- In requests with JOIN, the star character expands to a list of columns in all tables, in compliance with the SQL standard. You can restore the old behavior by setting `asterisk_left_columns_only` to 1 on the user configuration level.

Build changes:

- Most integration tests can now be run by commit.
- Code style checks can also be run by commit.
- The `memcpy` implementation is chosen correctly when building on CentOS7/Fedora. Etienne Champetier [https://github.com/yandex/ClickHouse/pull/2912]
- When using clang to build, some warnings from `-Weverything` have been added, in addition to the regular `-Wall-Wextra -Werror`. #2957 [https://github.com/yandex/ClickHouse/pull/2957]
- Debugging the build uses the `jemalloc` debug option.
- The interface of the library for interacting with ZooKeeper is declared abstract. #2950 [https://github.com/yandex/ClickHouse/pull/2950]

ClickHouse release 18.10.3, 2018-08-13

New features:

- HTTPS can be used for replication. #2760 [https://github.com/yandex/ClickHouse/pull/2760]
- Added the functions `murmurHash2_64`, `murmurHash3_32`, `murmurHash3_64`, and `murmurHash3_128` in addition to the existing `murmurHash2_32`. #2791 [https://github.com/yandex/ClickHouse/pull/2791]
- Support for Nullable types in the ClickHouse ODBC driver (`ODBCDriver2` output format). #2834 [https://github.com/yandex/ClickHouse/pull/2834]
- Support for `UUID` in the key columns.

Improvements:

- Clusters can be removed without restarting the server when they are deleted from the config files. #2777 [https://github.com/yandex/ClickHouse/pull/2777]
- External dictionaries can be removed without restarting the server when they are removed from config files. #2779 [https://github.com/yandex/ClickHouse/pull/2779]
- Added `SETTINGS` support for the `Kafka` table engine. Alexander Marshalov [https://github.com/yandex/ClickHouse/pull/2781]
- Improvements for the `UUID` data type (not yet complete). #2618 [https://github.com/yandex/ClickHouse/pull/2618]
- Support for empty parts after merges in the `SummingMergeTree`, `CollapsingMergeTree` and `VersionedCollapsingMergeTree` engines. #2815 [https://github.com/yandex/ClickHouse/pull/2815]
- Old records of completed mutations are deleted (`ALTER DELETE`). #2784 [https://github.com/yandex/ClickHouse/pull/2784]
- Added the `system.merge_tree_settings` table. Kirill Shvakov [https://github.com/yandex/ClickHouse/pull/2841]
- The `system.tables` table now has dependency columns: `dependencies_database` and `dependencies_table`. Winter Zhang [https://github.com/yandex/ClickHouse/pull/2851]
- Added the `max_partition_size_to_drop` config option. #2782 [https://github.com/yandex/ClickHouse/pull/2782]
- Added the `output_format_json_escape_forward_slashes` option. Alexander Bocharov [https://github.com/yandex/ClickHouse/pull/2812]

- Added the `max_fetch_partition_retries_count` setting. #2831 [https://github.com/yandex/ClickHouse/pull/2831]
- Added the `prefer_localhost_replica` setting for disabling the preference for a local replica and going to a local replica without inter-process interaction. #2832 [https://github.com/yandex/ClickHouse/pull/2832]
- The `quantileExact` aggregate function returns `nan` in the case of aggregation on an empty `Float32` or `Float64` set. Sundy Li [https://github.com/yandex/ClickHouse/pull/2855]

Bug fixes:

- Removed unnecessary escaping of the connection string parameters for ODBC, which made it impossible to establish a connection. This error occurred in version 18.6.0.
- Fixed the logic for processing `REPLACE PARTITION` commands in the replication queue. If there are two `REPLACE` commands for the same partition, the incorrect logic could cause one of them to remain in the replication queue and not be executed. #2814 [https://github.com/yandex/ClickHouse/pull/2814]
- Fixed a merge bug when all data parts were empty (parts that were formed from a merge or from `ALTER DELETE` if all data was deleted). This bug appeared in version 18.1.0. #2930 [https://github.com/yandex/ClickHouse/pull/2930]
- Fixed an error for concurrent `Set` or `Join` . Amos Bird [https://github.com/yandex/ClickHouse/pull/2823]
- Fixed the `Block structure mismatch in UNION stream: different number of columns` error that occurred for `UNION ALL` queries inside a sub-query if one of the `SELECT` queries contains duplicate column names. Winter Zhang [https://github.com/yandex/ClickHouse/pull/2094]
- Fixed a memory leak if an exception occurred when connecting to a MySQL server.
- Fixed incorrect clickhouse-client response code in case of a request error.
- Fixed incorrect behavior of materialized views containing DISTINCT.#2795 [https://github.com/yandex/ClickHouse/issues/2795]

Backward incompatible changes

- Removed support for CHECK TABLE queries for Distributed tables.

Build changes:

- The allocator has been replaced: `jemalloc` is now used instead of `tcmalloc` . In some scenarios, this increases speed up to 20%. However, there are queries that have slowed by up to 20%. Memory consumption has been reduced by approximately 10% in some scenarios, with improved stability. With highly competitive loads, CPU usage in userspace and in system shows just a slight increase. #2773 [https://github.com/yandex/ClickHouse/pull/2773]
- Use of libressl from a submodule. #1983 [https://github.com/yandex/ClickHouse/pull/1983] #2807 [https://github.com/yandex/ClickHouse/pull/2807]
- Use of unixodbc from a submodule. #2789 [https://github.com/yandex/ClickHouse/pull/2789]
- Use of mariadb-connector-c from a submodule. #2785 [https://github.com/yandex/ClickHouse/pull/2785]
- Added functional test files to the repository that depend on the availability of test data (for the time being, without the test data itself).

ClickHouse release 18.6.0, 2018-08-02

New features:

- Added support for ON expressions for the JOIN ON syntax: `JOIN ON Expr([table.]column ...) = Expr([table.]column, ...) [AND Expr([table.]column, ...) = Expr([table.]column, ...) ...]` The expression must be a chain of equalities joined by the AND operator. Each side of the equality can be an arbitrary expression over the columns of one of the tables. The use of fully qualified column names is supported (`table.name` , `database.table.name` , `table_alias.name` , `subquery_alias.name`) for the right table. #2742 [https://github.com/yandex/ClickHouse/pull/2742]

- HTTPS can be enabled for replication. #2760 [https://github.com/yandex/ClickHouse/pull/2760]

Improvements:

- The server passes the patch component of its version to the client. Data about the patch version component is in `system.processes` and `query_log`. #2646 [https://github.com/yandex/ClickHouse/pull/2646]

ClickHouse release 18.5.1, 2018-07-31

New features:

- Added the hash function `murmurHash2_32` #2756 [https://github.com/yandex/ClickHouse/pull/2756].

Improvements:

- Now you can use the `from_env` #2741 [https://github.com/yandex/ClickHouse/pull/2741] attribute to set values in config files from environment variables.
- Added case-insensitive versions of the `coalesce`, `ifNull`, and `nullIf functions` #2752 [https://github.com/yandex/ClickHouse/pull/2752].

Bug fixes:

- Fixed a possible bug when starting a replica #2759 [https://github.com/yandex/ClickHouse/pull/2759].

ClickHouse release 18.4.0, 2018-07-28

New features:

- Added system tables: `formats`, `data_type_families`, `aggregate_function_combinators`, `table_functions`, `table_engines`, `collations` #2721 [https://github.com/yandex/ClickHouse/pull/2721].
- Added the ability to use a table function instead of a table as an argument of a `remote` or `cluster table function` #2708 [https://github.com/yandex/ClickHouse/pull/2708].
- Support for `HTTP Basic` authentication in the replication protocol #2727 [https://github.com/yandex/ClickHouse/pull/2727].
- The `has` function now allows searching for a numeric value in an array of `Enum` values Maxim Khrisanfov [https://github.com/yandex/ClickHouse/pull/2699].
- Support for adding arbitrary message separators when reading from `Kafka` Amos Bird [https://github.com/yandex/ClickHouse/pull/2701].

Improvements:

- The `ALTER TABLE t DELETE WHERE` query does not rewrite data parts that were not affected by the WHERE condition #2694 [https://github.com/yandex/ClickHouse/pull/2694].
- The `use_minimalistic_checksums_in_zookeeper` option for `ReplicatedMergeTree` tables is enabled by default. This setting was added in version 1.1.54378, 2018-04-16. Versions that are older than 1.1.54378 can no longer be installed.
- Support for running `KILL` and `OPTIMIZE` queries that specify `ON CLUSTER` Winter Zhang [https://github.com/yandex/ClickHouse/pull/2689].

Bug fixes:

- Fixed the error `Column ... is not under an aggregate function and not in GROUP BY` for aggregation with an IN expression. This bug appeared in version 18.1.0. (bbdd780b [https://github.com/yandex/ClickHouse/commit/bbdd780be0be06a0f336775941cdd536878dd2c2])
- Fixed a bug in the `windowFunnel aggregate function` Winter Zhang [https://github.com/yandex/ClickHouse/pull/2735].

- Fixed a bug in the `anyHeavy` aggregate function (a2101df2 [https://github.com/yandex/ClickHouse/commit/a2101df25a6a0fba99aa71f8793d762af2b801ee]).
- Fixed server crash when using the `countArray()` aggregate function.

Backward incompatible changes:

- Parameters for `Kafka` engine was changed from `Kafka(kafka_broker_list, kafka_topic_list, kafka_group_name, kafka_format[, kafka_schema, kafka_num_consumers])` to `Kafka(kafka_broker_list, kafka_topic_list, kafka_group_name, kafka_format[, kafka_row_delimiter, kafka_schema, kafka_num_consumers])`. If your tables use `kafka_schema` or `kafka_num_consumers` parameters, you have to manually edit the metadata files `path/metadata/database/table.sql` and add `kafka_row_delimiter` parameter with `''` value.

ClickHouse release 18.1.0, 2018-07-23

New features:

- Support for the `ALTER TABLE t DELETE WHERE` query for non-replicated MergeTree tables (#2634 [https://github.com/yandex/ClickHouse/pull/2634]).
- Support for arbitrary types for the `uniq*` family of aggregate functions (#2010 [https://github.com/yandex/ClickHouse/issues/2010]).
- Support for arbitrary types in comparison operators (#2026 [https://github.com/yandex/ClickHouse/issues/2026]).
- The `users.xml` file allows setting a subnet mask in the format `10.0.0.1/255.255.255.0`. This is necessary for using masks for IPv6 networks with zeros in the middle (#2637 [https://github.com/yandex/ClickHouse/pull/2637]).
- Added the `arrayDistinct` function (#2670 [https://github.com/yandex/ClickHouse/pull/2670]).
- The SummingMergeTree engine can now work with AggregateFunction type columns (Constantin S. Pan [https://github.com/yandex/ClickHouse/pull/2566]).

Improvements:

- Changed the numbering scheme for release versions. Now the first part contains the year of release (A.D., Moscow timezone, minus 2000), the second part contains the number for major changes (increases for most releases), and the third part is the patch version. Releases are still backwards compatible, unless otherwise stated in the changelog.
- Faster conversions of floating-point numbers to a string (Amos Bird [https://github.com/yandex/ClickHouse/pull/2664]).
- If some rows were skipped during an insert due to parsing errors (this is possible with the `input_allow_errors_num` and `input_allow_errors_ratio` settings enabled), the number of skipped rows is now written to the server log (Leonardo Cecchi [https://github.com/yandex/ClickHouse/pull/2669]).

Bug fixes:

- Fixed the TRUNCATE command for temporary tables (Amos Bird [https://github.com/yandex/ClickHouse/pull/2624]).
- Fixed a rare deadlock in the ZooKeeper client library that occurred when there was a network error while reading the response (c315200 [https://github.com/yandex/ClickHouse/commit/c315200e64b87e44bdf740707fc857d1fdf7e947]).
- Fixed an error during a CAST to Nullable types (#1322 [https://github.com/yandex/ClickHouse/issues/1322]).
- Fixed the incorrect result of the `maxIntersection()` function when the boundaries of intervals coincided (Michael Furmur [https://github.com/yandex/ClickHouse/pull/2657]).
- Fixed incorrect transformation of the OR expression chain in a function argument (chenxing-xc [https://github.com/yandex/ClickHouse/pull/2663]).
- Fixed performance degradation for queries containing `IN (subquery)` expressions inside another subquery (#2571

[https://github.com/yandex/ClickHouse/issues/2571]).

- Fixed incompatibility between servers with different versions in distributed queries that use a `CAST` function that isn't in uppercase letters (fe8c4d6 [https://github.com/yandex/ClickHouse/commit/fe8c4d64e434cacd4ceef34faa9005129f2190a5]).
- Added missing quoting of identifiers for queries to an external DBMS (#2635 [https://github.com/yandex/ClickHouse/issues/2635]).

Backward incompatible changes:

- Converting a string containing the number zero to DateTime does not work. Example: `SELECT toDateTime('0')`. This is also the reason that `DateTime DEFAULT '0'` does not work in tables, as well as `<null_value>0</null_value>` in dictionaries. Solution: replace `0` with `0000-00-00 00:00:00`.

ClickHouse release 1.1.54394, 2018-07-12

New features:

- Added the `histogram` aggregate function (Mikhail Surin [https://github.com/yandex/ClickHouse/pull/2521]).
- Now `OPTIMIZE TABLE ... FINAL` can be used without specifying partitions for `ReplicatedMergeTree` (Amos Bird [https://github.com/yandex/ClickHouse/pull/2600]).

Bug fixes:

- Fixed a problem with a very small timeout for sockets (one second) for reading and writing when sending and downloading replicated data, which made it impossible to download larger parts if there is a load on the network or disk (it resulted in cyclical attempts to download parts). This error occurred in version 1.1.54388.
- Fixed issues when using chroot in ZooKeeper if you inserted duplicate data blocks in the table.
- The `has` function now works correctly for an array with Nullable elements (#2115 [https://github.com/yandex/ClickHouse/issues/2115]).
- The `system.tables` table now works correctly when used in distributed queries. The `metadata_modification_time` and `engine_full` columns are now non-virtual. Fixed an error that occurred if only these columns were requested from the table.
- Fixed how an empty `TinyLog` table works after inserting an empty data block (#2563 [https://github.com/yandex/ClickHouse/issues/2563]).
- The `system.zookeeper` table works if the value of the node in ZooKeeper is NULL.

ClickHouse release 1.1.54390, 2018-07-06

New features:

- Queries can be sent in `multipart/form-data` format (in the `query` field), which is useful if external data is also sent for query processing (Olga Hvostikova [https://github.com/yandex/ClickHouse/pull/2490]).
- Added the ability to enable or disable processing single or double quotes when reading data in CSV format. You can configure this in the `format_csv_allow_single_quotes` and `format_csv_allow_double_quotes` settings (Amos Bird [https://github.com/yandex/ClickHouse/pull/2574]).
- Now `OPTIMIZE TABLE ... FINAL` can be used without specifying the partition for non-replicated variants of `MergeTree` (Amos Bird [https://github.com/yandex/ClickHouse/pull/2599]).

Improvements:

- Improved performance, reduced memory consumption, and correct memory consumption tracking with use of the IN operator when a table index could be used (#2584 [https://github.com/yandex/ClickHouse/pull/2584]).

- Removed redundant checking of checksums when adding a data part. This is important when there are a large number of replicas, because in these cases the total number of checks was equal to N^2.
- Added support for `Array(Tuple(...))` arguments for the `arrayEnumerateUniq` function (#2573 [https://github.com/yandex/ClickHouse/pull/2573]).
- Added `Nullable` support for the `runningDifference` function (#2594 [https://github.com/yandex/ClickHouse/pull/2594]).
- Improved query analysis performance when there is a very large number of expressions (#2572 [https://github.com/yandex/ClickHouse/pull/2572]).
- Faster selection of data parts for merging in `ReplicatedMergeTree` tables. Faster recovery of the ZooKeeper session (#2597 [https://github.com/yandex/ClickHouse/pull/2597]).
- The `format_version.txt` file for `MergeTree` tables is re-created if it is missing, which makes sense if ClickHouse is launched after copying the directory structure without files (Ciprian Hacman [https://github.com/yandex/ClickHouse/pull/2593]).

Bug fixes:

- Fixed a bug when working with ZooKeeper that could make it impossible to recover the session and readonly states of tables before restarting the server.
- Fixed a bug when working with ZooKeeper that could result in old nodes not being deleted if the session is interrupted.
- Fixed an error in the `quantileTDigest` function for Float arguments (this bug was introduced in version 1.1.54388) (Mikhail Surin [https://github.com/yandex/ClickHouse/pull/2553]).
- Fixed a bug in the index for MergeTree tables if the primary key column is located inside the function for converting types between signed and unsigned integers of the same size (#2603 [https://github.com/yandex/ClickHouse/pull/2603]).
- Fixed segfault if `macros` are used but they aren't in the config file (#2570 [https://github.com/yandex/ClickHouse/pull/2570]).
- Fixed switching to the default database when reconnecting the client (#2583 [https://github.com/yandex/ClickHouse/pull/2583]).
- Fixed a bug that occurred when the `use_index_for_in_with_subqueries` setting was disabled.

Security fix:

- Sending files is no longer possible when connected to MySQL (`LOAD DATA LOCAL INFILE`).

ClickHouse release 1.1.54388, 2018-06-28

New features:

- Support for the `ALTER TABLE t DELETE WHERE` query for replicated tables. Added the `system.mutations` table to track progress of this type of queries.
- Support for the `ALTER TABLE t [REPLACE|ATTACH] PARTITION` query for *MergeTree tables.
- Support for the `TRUNCATE TABLE` query (Winter Zhang [https://github.com/yandex/ClickHouse/pull/2260])
- Several new `SYSTEM` queries for replicated tables (`RESTART REPLICAS` , `SYNC REPLICA` , `[STOP|START] [MERGES|FETCHES|SENDS REPLICATED|REPLICATION QUEUES]`).
- Added the ability to write to a table with the MySQL engine and the corresponding table function (sundy-li [https://github.com/yandex/ClickHouse/pull/2294]).
- Added the `url()` table function and the `URL` table engine (Alexander Sapin [https://github.com/yandex/ClickHouse/pull/2501]).
- Added the `windowFunnel` aggregate function (sundy-li [https://github.com/yandex/ClickHouse/pull/2352]).

- New `startsWith` and `endsWith` functions for strings (Vadim Plakhtinsky [https://github.com/yandex/ClickHouse/pull/2429]).
- The `numbers()` table function now allows you to specify the offset (Winter Zhang [https://github.com/yandex/ClickHouse/pull/2535]).
- The password to `clickhouse-client` can be entered interactively.
- Server logs can now be sent to syslog (Alexander Krasheninnikov [https://github.com/yandex/ClickHouse/pull/2459]).
- Support for logging in dictionaries with a shared library source (Alexander Sapin [https://github.com/yandex/ClickHouse/pull/2472]).
- Support for custom CSV delimiters (Ivan Zhukov [https://github.com/yandex/ClickHouse/pull/2263])
- Added the `date_time_input_format` setting. If you switch this setting to `'best_effort'`, DateTime values will be read in a wide range of formats.
- Added the `clickhouse-obfuscator` utility for data obfuscation. Usage example: publishing data used in performance tests.

Experimental features:

- Added the ability to calculate `and` arguments only where they are needed (Anastasia Tsarkova [https://github.com/yandex/ClickHouse/pull/2272])
- JIT compilation to native code is now available for some expressions (pyos [https://github.com/yandex/ClickHouse/pull/2277]).

Bug fixes:

- Duplicates no longer appear for a query with `DISTINCT` and `ORDER BY`.
- Queries with `ARRAY JOIN` and `arrayFilter` no longer return an incorrect result.
- Fixed an error when reading an array column from a Nested structure (#2066 [https://github.com/yandex/ClickHouse/issues/2066]).
- Fixed an error when analyzing queries with a HAVING clause like `HAVING tuple IN (...)`.
- Fixed an error when analyzing queries with recursive aliases.
- Fixed an error when reading from ReplacingMergeTree with a condition in PREWHERE that filters all rows (#2525 [https://github.com/yandex/ClickHouse/issues/2525]).
- User profile settings were not applied when using sessions in the HTTP interface.
- Fixed how settings are applied from the command line parameters in clickhouse-local.
- The ZooKeeper client library now uses the session timeout received from the server.
- Fixed a bug in the ZooKeeper client library when the client waited for the server response longer than the timeout.
- Fixed pruning of parts for queries with conditions on partition key columns (#2342 [https://github.com/yandex/ClickHouse/issues/2342]).
- Merges are now possible after `CLEAR COLUMN IN PARTITION` (#2315 [https://github.com/yandex/ClickHouse/issues/2315]).
- Type mapping in the ODBC table function has been fixed (sundy-li [https://github.com/yandex/ClickHouse/pull/2268]).
- Type comparisons have been fixed for `DateTime` with and without the time zone (Alexander Bocharov [https://github.com/yandex/ClickHouse/pull/2400]).
- Fixed syntactic parsing and formatting of the `CAST` operator.
- Fixed insertion into a materialized view for the Distributed table engine (Babacar Diassé [https://github.com/yandex/ClickHouse/pull/2411]).
- Fixed a race condition when writing data from the `Kafka` engine to materialized views (Yangkuan Liu

[https://github.com/yandex/ClickHouse/pull/2448]).

- Fixed SSRF in the remote() table function.
- Fixed exit behavior of `clickhouse-client` in multiline mode (#2510 [https://github.com/yandex/ClickHouse/issues/2510]).

Improvements:

- Background tasks in replicated tables are now performed in a thread pool instead of in separate threads (Silviu Caragea [https://github.com/yandex/ClickHouse/pull/1722]).
- Improved LZ4 compression performance.
- Faster analysis for queries with a large number of JOINs and sub-queries.
- The DNS cache is now updated automatically when there are too many network errors.
- Table inserts no longer occur if the insert into one of the materialized views is not possible because it has too many parts.
- Corrected the discrepancy in the event counters `Query`, `SelectQuery`, and `InsertQuery`.
- Expressions like `tuple IN (SELECT tuple)` are allowed if the tuple types match.
- A server with replicated tables can start even if you haven't configured ZooKeeper.
- When calculating the number of available CPU cores, limits on cgroups are now taken into account (Atri Sharma [https://github.com/yandex/ClickHouse/pull/2325]).
- Added chown for config directories in the systemd config file (Mikhail Shiryaev [https://github.com/yandex/ClickHouse/pull/2421]).

Build changes:

- The gcc8 compiler can be used for builds.
- Added the ability to build llvm from submodule.
- The version of the librdkafka library has been updated to v0.11.4.
- Added the ability to use the system libcpuid library. The library version has been updated to 0.4.0.
- Fixed the build using the vectorclass library (Babacar Diassé [https://github.com/yandex/ClickHouse/pull/2274]).
- Cmake now generates files for ninja by default (like when using `-G Ninja`).
- Added the ability to use the libtinfo library instead of libtermcap (Georgy Kondratiev [https://github.com/yandex/ClickHouse/pull/2519]).
- Fixed a header file conflict in Fedora Rawhide (#2520 [https://github.com/yandex/ClickHouse/issues/2520]).

Backward incompatible changes:

- Removed escaping in `Vertical` and `Pretty*` formats and deleted the `VerticalRaw` format.
- If servers with version 1.1.54388 (or newer) and servers with an older version are used simultaneously in a distributed query and the query has the `cast(x, 'Type')` expression without the `AS` keyword and doesn't have the word `cast` in uppercase, an exception will be thrown with a message like `Not found column cast(0, 'UInt8') in block`. Solution: Update the server on the entire cluster.

ClickHouse release 1.1.54385, 2018-06-01

Bug fixes:

- Fixed an error that in some cases caused ZooKeeper operations to block.

ClickHouse release 1.1.54383, 2018-05-22

Bug fixes:

- Fixed a slowdown of replication queue if a table has many replicas.

ClickHouse release 1.1.54381, 2018-05-14

Bug fixes:

- Fixed a nodes leak in ZooKeeper when ClickHouse loses connection to ZooKeeper server.

ClickHouse release 1.1.54380, 2018-04-21

New features:

- Added the table function `file(path, format, structure)`. An example reading bytes from `/dev/urandom`: ``ln -s /dev/urandom /var/lib/clickhouse/user_files/random``clickhouse-client -q "SELECT * FROM file('random', 'RowBinary', 'd UInt8') LIMIT 10"``.

Improvements:

- Subqueries can be wrapped in `()` brackets to enhance query readability. For example: `(SELECT 1) UNION ALL (SELECT 1)`.
- Simple `SELECT` queries from the `system.processes` table are not included in the `max_concurrent_queries` limit.

Bug fixes:

- Fixed incorrect behavior of the `IN` operator when select from `MATERIALIZED VIEW`.
- Fixed incorrect filtering by partition index in expressions like `partition_key_column IN (...)`.
- Fixed inability to execute `OPTIMIZE` query on non-leader replica if `REANAME` was performed on the table.
- Fixed the authorization error when executing `OPTIMIZE` or `ALTER` queries on a non-leader replica.
- Fixed freezing of `KILL QUERY`.
- Fixed an error in ZooKeeper client library which led to loss of watches, freezing of distributed DDL queue, and slowdowns in the replication queue if a non-empty `chroot` prefix is used in the ZooKeeper configuration.

Backward incompatible changes:

- Removed support for expressions like `(a, b) IN (SELECT (a, b))` (you can use the equivalent expression `(a, b) IN (SELECT a, b)`). In previous releases, these expressions led to undetermined `WHERE` filtering or caused errors.

ClickHouse release 1.1.54378, 2018-04-16

New features:

- Logging level can be changed without restarting the server.
- Added the `SHOW CREATE DATABASE` query.
- The `query_id` can be passed to `clickhouse-client` (elBroom).
- New setting: `max_network_bandwidth_for_all_users`.
- Added support for `ALTER TABLE ... PARTITION ...` for `MATERIALIZED VIEW`.
- Added information about the size of data parts in uncompressed form in the system table.
- Server-to-server encryption support for distributed tables (`<secure>1</secure>` in the replica config in `<remote_servers>`).
- Configuration of the table level for the `ReplicatedMergeTree` family in order to minimize the amount of data stored in

Zookeeper: : `use_minimalistic_checksums_in_zookeeper = 1`

- Configuration of the `clickhouse-client` prompt. By default, server names are now output to the prompt. The server's display name can be changed. It's also sent in the `X-ClickHouse-Display-Name` HTTP header (Kirill Shvakov).
- Multiple comma-separated `topics` can be specified for the `Kafka` engine (Tobias Adamson)
- When a query is stopped by `KILL QUERY` or `replace_running_query`, the client receives the `Query was cancelled` exception instead of an incomplete result.

Improvements:

- `ALTER TABLE ... DROP/DETACH PARTITION` queries are run at the front of the replication queue.
- `SELECT ... FINAL` and `OPTIMIZE ... FINAL` can be used even when the table has a single data part.
- A `query_log` table is recreated on the fly if it was deleted manually (Kirill Shvakov).
- The `lengthUTF8` function runs faster (zhang2014).
- Improved performance of synchronous inserts in `Distributed` tables (`insert_distributed_sync = 1`) when there is a very large number of shards.
- The server accepts the `send_timeout` and `receive_timeout` settings from the client and applies them when connecting to the client (they are applied in reverse order: the server socket's `send_timeout` is set to the `receive_timeout` value received from the client, and vice versa).
- More robust crash recovery for asynchronous insertion into `Distributed` tables.
- The return type of the `countEqual` function changed from `UInt32` to `UInt64` (谢磊).

Bug fixes:

- Fixed an error with `IN` when the left side of the expression is `Nullable` .
- Correct results are now returned when using tuples with `IN` when some of the tuple components are in the table index.
- The `max_execution_time` limit now works correctly with distributed queries.
- Fixed errors when calculating the size of composite columns in the `system.columns` table.
- Fixed an error when creating a temporary table `CREATE TEMPORARY TABLE IF NOT EXISTS.`
- Fixed errors in `StorageKafka` (##2075)
- Fixed server crashes from invalid arguments of certain aggregate functions.
- Fixed the error that prevented the `DETACH DATABASE` query from stopping background tasks for `ReplicatedMergeTree` tables.
- `Too many parts` state is less likely to happen when inserting into aggregated materialized views (##2084).
- Corrected recursive handling of substitutions in the config if a substitution must be followed by another substitution on the same level.
- Corrected the syntax in the metadata file when creating a `VIEW` that uses a query with `UNION ALL` .
- `SummingMergeTree` now works correctly for summation of nested data structures with a composite key.
- Fixed the possibility of a race condition when choosing the leader for `ReplicatedMergeTree` tables.

Build changes:

- The build supports `ninja` instead of `make` and uses `ninja` by default for building releases.
- Renamed packages: `clickhouse-server-base` in `clickhouse-common-static` ; `clickhouse-server-common` in `clickhouse-server` ; `clickhouse-common-dbg` in `clickhouse-common-static-dbg` . To install, use `clickhouse-server clickhouse-client` . Packages with the old names will still load in the repositories for backward compatibility.

Backward incompatible changes:

- Removed the special interpretation of an IN expression if an array is specified on the left side. Previously, the expression `arr IN (set)` was interpreted as "at least one `arr` element belongs to the `set`". To get the same behavior in the new version, write `arrayExists(x -> x IN (set), arr)`.
- Disabled the incorrect use of the socket option `SO_REUSEPORT`, which was incorrectly enabled by default in the Poco library. Note that on Linux there is no longer any reason to simultaneously specify the addresses `::` and `0.0.0.0` for listen – use just `::`, which allows listening to the connection both over IPv4 and IPv6 (with the default kernel config settings). You can also revert to the behavior from previous versions by specifying `<listen_reuse_port>1</listen_reuse_port>` in the config.

ClickHouse release 1.1.54370, 2018-03-16

New features:

- Added the `system.macros` table and auto updating of macros when the config file is changed.
- Added the `SYSTEM RELOAD CONFIG` query.
- Added the `maxIntersections(left_col, right_col)` aggregate function, which returns the maximum number of simultaneously intersecting intervals `[left; right]`. The `maxIntersectionsPosition(left, right)` function returns the beginning of the "maximum" interval. (Michael Furmur [https://github.com/yandex/ClickHouse/pull/2012]).

Improvements:

- When inserting data in a `Replicated` table, fewer requests are made to `ZooKeeper` (and most of the user-level errors have disappeared from the `ZooKeeper` log).
- Added the ability to create aliases for data sets. Example: `WITH (1, 2, 3) AS set SELECT number IN set FROM system.numbers LIMIT 10`.

Bug fixes:

- Fixed the `Illegal PREWHERE` error when reading from Merge tables for `Distributed` tables.
- Added fixes that allow you to start clickhouse-server in IPv4-only Docker containers.
- Fixed a race condition when reading from system `system.parts_columns tables.`
- Removed double buffering during a synchronous insert to a `Distributed` table, which could have caused the connection to timeout.
- Fixed a bug that caused excessively long waits for an unavailable replica before beginning a `SELECT` query.
- Fixed incorrect dates in the `system.parts` table.
- Fixed a bug that made it impossible to insert data in a `Replicated` table if `chroot` was non-empty in the configuration of the `ZooKeeper` cluster.
- Fixed the vertical merging algorithm for an empty `ORDER BY` table.
- Restored the ability to use dictionaries in queries to remote tables, even if these dictionaries are not present on the requestor server. This functionality was lost in release 1.1.54362.
- Restored the behavior for queries like `SELECT * FROM remote('server2', default.table) WHERE col IN (SELECT col2 FROM default.table)` when the right side of the `IN` should use a remote `default.table` instead of a local one. This behavior was broken in version 1.1.54358.
- Removed extraneous error-level logging of `Not found column ... in block`.

Clickhouse Release 1.1.54362, 2018-03-11

New features:

- Aggregation without `GROUP BY` for an empty set (such as `SELECT count(*) FROM table WHERE 0`) now returns a result with one row with null values for aggregate functions, in compliance with the SQL standard. To restore the old behavior (return an empty result), set `empty_result_for_aggregation_by_empty_set` to 1.
- Added type conversion for `UNION ALL`. Different alias names are allowed in `SELECT` positions in `UNION ALL`, in compliance with the SQL standard.
- Arbitrary expressions are supported in `LIMIT BY` clauses. Previously, it was only possible to use columns resulting from `SELECT`.
- An index of `MergeTree` tables is used when `IN` is applied to a tuple of expressions from the columns of the primary key. Example: `WHERE (UserID, EventDate) IN ((123, '2000-01-01'), ...)` (Anastasiya Tsarkova).
- Added the `clickhouse-copier` tool for copying between clusters and resharding data (beta).
- Added consistent hashing functions: `yandexConsistentHash`, `jumpConsistentHash`, `sumburConsistentHash`. They can be used as a sharding key in order to reduce the amount of network traffic during subsequent reshardings.
- Added functions: `arrayAny`, `arrayAll`, `hasAny`, `hasAll`, `arrayIntersect`, `arrayResize`.
- Added the `arrayCumSum` function (Javi Santana).
- Added the `parseDateTimeBestEffort`, `parseDateTimeBestEffortOrZero`, and `parseDateTimeBestEffortOrNull` functions to read the DateTime from a string containing text in a wide variety of possible formats.
- Data can be partially reloaded from external dictionaries during updating (load just the records in which the value of the specified field greater than in the previous download) (Arsen Hakobyan).
- Added the `cluster` table function. Example: `cluster(cluster_name, db, table)`. The `remote` table function can accept the cluster name as the first argument, if it is specified as an identifier.
- The `remote` and `cluster` table functions can be used in `INSERT` requests.
- Added the `create_table_query` and `engine_full` virtual columns to the `system.tables` table . The `metadata_modification_time` column is virtual.
- Added the `data_path` and `metadata_path` columns to `system.tables` and `system.databases` tables, and added the `path` column to the `system.parts` and `system.parts_columns` tables.
- Added additional information about merges in the `system.part_log` table.
- An arbitrary partitioning key can be used for the `system.query_log` table (Kirill Shvakov).
- The `SHOW TABLES` query now also shows temporary tables. Added temporary tables and the `is_temporary` column to `system.tables` (zhang2014).
- Added `DROP TEMPORARY TABLE` and `EXISTS TEMPORARY TABLE` queries (zhang2014).
- Support for `SHOW CREATE TABLE` for temporary tables (zhang2014).
- Added the `system_profile` configuration parameter for the settings used by internal processes.
- Support for loading `object_id` as an attribute in `MongoDB` dictionaries (Pavel Litvinenko).
- Reading `null` as the default value when loading data for an external dictionary with the `MongoDB` source (Pavel Litvinenko).
- Reading `DateTime` values in the `Values` format from a Unix timestamp without single quotes.
- Failover is supported in `remote` table functions for cases when some of the replicas are missing the requested table.
- Configuration settings can be overridden in the command line when you run `clickhouse-server`. Example: `clickhouse-server -- --logger.level=information`.
- Implemented the `empty` function from a `FixedString` argument: the function returns 1 if the string consists entirely of null bytes (zhang2014).
- Added the `listen_try` configuration parameter for listening to at least one of the listen addresses without quitting, if some of the addresses can't be listened to (useful for systems with disabled support for IPv4 or IPv6).
- Added the `VersionedCollapsingMergeTree` table engine.

- Support for rows and arbitrary numeric types for the `library` dictionary source.
- `MergeTree` tables can be used without a primary key (you need to specify `ORDER BY tuple()`).
- A `Nullable` type can be `CAST` to a non- `Nullable` type if the argument is not `NULL` .
- `RENAME TABLE` can be performed for `VIEW` .
- Added the `throwIf` function.
- Added the `odbc_default_field_size` option, which allows you to extend the maximum size of the value loaded from an ODBC source (by default, it is 1024).
- The `system.processes` table and `SHOW PROCESSLIST` now have the `is_cancelled` and `peak_memory_usage` columns.

Improvements:

- Limits and quotas on the result are no longer applied to intermediate data for `INSERT SELECT` queries or for `SELECT` subqueries.
- Fewer false triggers of `force_restore_data` when checking the status of `Replicated` tables when the server starts.
- Added the `allow_distributed_ddl` option.
- Nondeterministic functions are not allowed in expressions for `MergeTree` table keys.
- Files with substitutions from `config.d` directories are loaded in alphabetical order.
- Improved performance of the `arrayElement` function in the case of a constant multidimensional array with an empty array as one of the elements. Example: `[[1], []][x]` .
- The server starts faster now when using configuration files with very large substitutions (for instance, very large lists of IP networks).
- When running a query, table valued functions run once. Previously, `remote` and `mysql` table valued functions performed the same query twice to retrieve the table structure from a remote server.
- The `MkDocs` documentation generator is used.
- When you try to delete a table column that `DEFAULT` / `MATERIALIZED` expressions of other columns depend on, an exception is thrown (zhang2014).
- Added the ability to parse an empty line in text formats as the number 0 for `Float` data types. This feature was previously available but was lost in release 1.1.54342.
- `Enum` values can be used in `min` , `max` , `sum` and some other functions. In these cases, it uses the corresponding numeric values. This feature was previously available but was lost in the release 1.1.54337.
- Added `max_expanded_ast_elements` to restrict the size of the AST after recursively expanding aliases.

Bug fixes:

- Fixed cases when unnecessary columns were removed from subqueries in error, or not removed from subqueries containing `UNION ALL` .
- Fixed a bug in merges for `ReplacingMergeTree` tables.
- Fixed synchronous insertions in `Distributed` tables (`insert_distributed_sync = 1`).
- Fixed segfault for certain uses of `FULL` and `RIGHT JOIN` with duplicate columns in subqueries.
- Fixed segfault for certain uses of `replace_running_query` and `KILL QUERY` .
- Fixed the order of the `source` and `last_exception` columns in the `system.dictionaries` table.
- Fixed a bug when the `DROP DATABASE` query did not delete the file with metadata.
- Fixed the `DROP DATABASE` query for `Dictionary` databases.
- Fixed the low precision of `uniqHLL12` and `uniqCombined` functions for cardinalities greater than 100 million items (Alex Bocharov).

- Fixed the calculation of implicit default values when necessary to simultaneously calculate default explicit expressions in `INSERT` queries (zhang2014).
- Fixed a rare case when a query to a `MergeTree` table couldn't finish (chenxing-xc).
- Fixed a crash that occurred when running a `CHECK` query for `Distributed` tables if all shards are local (chenxing.xc).
- Fixed a slight performance regression with functions that use regular expressions.
- Fixed a performance regression when creating multidimensional arrays from complex expressions.
- Fixed a bug that could cause an extra `FORMAT` section to appear in an `.sql` file with metadata.
- Fixed a bug that caused the `max_table_size_to_drop` limit to apply when trying to delete a `MATERIALIZED VIEW` looking at an explicitly specified table.
- Fixed incompatibility with old clients (old clients were sometimes sent data with the `DateTime('timezone')` type, which they do not understand).
- Fixed a bug when reading `Nested` column elements of structures that were added using `ALTER` but that are empty for the old partitions, when the conditions for these columns moved to `PREWHERE`.
- Fixed a bug when filtering tables by virtual `_table` columns in queries to `Merge` tables.
- Fixed a bug when using `ALIAS` columns in `Distributed` tables.
- Fixed a bug that made dynamic compilation impossible for queries with aggregate functions from the `quantile` family.
- Fixed a race condition in the query execution pipeline that occurred in very rare cases when using `Merge` tables with a large number of tables, and when using `GLOBAL` subqueries.
- Fixed a crash when passing arrays of different sizes to an `arrayReduce` function when using aggregate functions from multiple arguments.
- Prohibited the use of queries with `UNION ALL` in a `MATERIALIZED VIEW`.
- Fixed an error during initialization of the `part_log` system table when the server starts (by default, `part_log` is disabled).

Backward incompatible changes:

- Removed the `distributed_ddl_allow_replicated_alter` option. This behavior is enabled by default.
- Removed the `strict_insert_defaults` setting. If you were using this functionality, write to `clickhouse-feedback@yandex-team.com`.
- Removed the `UnsortedMergeTree` engine.

Clickhouse Release 1.1.54343, 2018-02-05

- Added macros support for defining cluster names in distributed DDL queries and constructors of Distributed tables: `CREATE TABLE distr ON CLUSTER '{cluster}' (...) ENGINE = Distributed('{cluster}', 'db', 'table')`.
- Now queries like `SELECT ... FROM table WHERE expr IN (subquery)` are processed using the `table` index.
- Improved processing of duplicates when inserting to Replicated tables, so they no longer slow down execution of the replication queue.

Clickhouse Release 1.1.54342, 2018-01-22

This release contains bug fixes for the previous release 1.1.54337:

- Fixed a regression in 1.1.54337: if the default user has readonly access, then the server refuses to start up with the message `Cannot create database in readonly mode`.
- Fixed a regression in 1.1.54337: on systems with systemd, logs are always written to syslog regardless of the configuration; the watchdog script still uses init.d.

- Fixed a regression in 1.1.54337: wrong default configuration in the Docker image.
- Fixed nondeterministic behavior of GraphiteMergeTree (you can see it in log messages `Data after merge is not byte-identical to the data on another replicas`).
- Fixed a bug that may lead to inconsistent merges after OPTIMIZE query to Replicated tables (you may see it in log messages `Part ... intersects the previous part`).
- Buffer tables now work correctly when MATERIALIZED columns are present in the destination table (by zhang2014).
- Fixed a bug in implementation of NULL.

Clickhouse Release 1.1.54337, 2018-01-18

New features:

- Added support for storage of multi-dimensional arrays and tuples (`Tuple` data type) in tables.
- Support for table functions for `DESCRIBE` and `INSERT` queries. Added support for subqueries in `DESCRIBE`. Examples: `DESC TABLE remote('host', default.hits)`; `DESC TABLE (SELECT 1)`; `INSERT INTO TABLE FUNCTION remote('host', default.hits)`. Support for `INSERT INTO TABLE` in addition to `INSERT INTO`.
- Improved support for time zones. The `DateTime` data type can be annotated with the timezone that is used for parsing and formatting in text formats. Example: `DateTime('Europe/Moscow')`. When timezones are specified in functions for `DateTime` arguments, the return type will track the timezone, and the value will be displayed as expected.
- Added the functions `toTimeZone`, `timeDiff`, `toQuarter`, `toRelativeQuarterNum`. The `toRelativeHour`/`Minute`/`Second` functions can take a value of type `Date` as an argument. The `now` function name is case-sensitive.
- Added the `toStartOfFifteenMinutes` function (Kirill Shvakov).
- Added the `clickhouse format` tool for formatting queries.
- Added the `format_schema_path` configuration parameter (Marek Vavruşa). It is used for specifying a schema in `Cap'n Proto` format. Schema files can be located only in the specified directory.
- Added support for config substitutions (`incl` and `conf.d`) for configuration of external dictionaries and models (Pavel Yakunin).
- Added a column with documentation for the `system.settings` table (Kirill Shvakov).
- Added the `system.parts_columns` table with information about column sizes in each data part of `MergeTree` tables.
- Added the `system.models` table with information about loaded `CatBoost` machine learning models.
- Added the `mysql` and `odbc` table function and corresponding `MySQL` and `ODBC` table engines for accessing remote databases. This functionality is in the beta stage.
- Added the possibility to pass an argument of type `AggregateFunction` for the `groupArray` aggregate function (so you can create an array of states of some aggregate function).
- Removed restrictions on various combinations of aggregate function combinators. For example, you can use `avgForEachIf` as well as `avgIfForEach` aggregate functions, which have different behaviors.
- The `-ForEach` aggregate function combinator is extended for the case of aggregate functions of multiple arguments.
- Added support for aggregate functions of `Nullable` arguments even for cases when the function returns a non-`Nullable` result (added with the contribution of Silviu Caragea). Example: `groupArray`, `groupUniqArray`, `topK`.
- Added the `max_client_network_bandwidth` for `clickhouse-client` (Kirill Shvakov).
- Users with the `readonly = 2` setting are allowed to work with TEMPORARY tables (CREATE, DROP, INSERT...) (Kirill Shvakov).
- Added support for using multiple consumers with the `Kafka` engine. Extended configuration options for `Kafka` (Marek Vavruša).

- Added the `intExp3` and `intExp4` functions.
- Added the `sumKahan` aggregate function.
- Added the to * Number *OrNull functions, where * Number*is a numeric type.
- Added support for `WITH` clauses for an `INSERT SELECT` query (author: zhang2014).
- Added settings: `http_connection_timeout`, `http_send_timeout`, `http_receive_timeout`. In particular, these settings are used for downloading data parts for replication. Changing these settings allows for faster failover if the network is overloaded.
- Added support for `ALTER` for tables of type `Null` (Anastasiya Tsarkova).
- The `reinterpretAsString` function is extended for all data types that are stored contiguously in memory.
- Added the `--silent` option for the `clickhouse-local` tool. It suppresses printing query execution info in stderr.
- Added support for reading values of type `Date` from text in a format where the month and/or day of the month is specified using a single digit instead of two digits (Amos Bird).

Performance optimizations:

- Improved performance of aggregate functions `min`, `max`, `any`, `anyLast`, `anyHeavy`, `argMin`, `argMax` from string arguments.
- Improved performance of the functions `isInfinite`, `isFinite`, `isNaN`, `roundToExp2`.
- Improved performance of parsing and formatting `Date` and `DateTime` type values in text format.
- Improved performance and precision of parsing floating point numbers.
- Lowered memory usage for `JOIN` in the case when the left and right parts have columns with identical names that are not contained in `USING`.
- Improved performance of aggregate functions `varSamp`, `varPop`, `stddevSamp`, `stddevPop`, `covarSamp`, `covarPop`, `corr` by reducing computational stability. The old functions are available under the names `varSampStable`, `varPopStable`, `stddevSampStable`, `stddevPopStable`, `covarSampStable`, `covarPopStable`, `corrStable`.

Bug fixes:

- Fixed data deduplication after running a `DROP` or `DETACH PARTITION` query. In the previous version, dropping a partition and inserting the same data again was not working because inserted blocks were considered duplicates.
- Fixed a bug that could lead to incorrect interpretation of the `WHERE` clause for `CREATE MATERIALIZED VIEW` queries with `POPULATE`.
- Fixed a bug in using the `root_path` parameter in the `zookeeper_servers` configuration.
- Fixed unexpected results of passing the `Date` argument to `toStartOfDay`.
- Fixed the `addMonths` and `subtractMonths` functions and the arithmetic for `INTERVAL n MONTH` in cases when the result has the previous year.
- Added missing support for the `UUID` data type for `DISTINCT`, `JOIN`, and `uniq` aggregate functions and external dictionaries (Evgeniy Ivanov). Support for `UUID` is still incomplete.
- Fixed `SummingMergeTree` behavior in cases when the rows summed to zero.
- Various fixes for the `Kafka` engine (Marek Vavruša).
- Fixed incorrect behavior of the `Join` table engine (Amos Bird).
- Fixed incorrect allocator behavior under FreeBSD and OS X.
- The `extractAll` function now supports empty matches.
- Fixed an error that blocked usage of `libressl` instead of `openssl`.
- Fixed the `CREATE TABLE AS SELECT` query from temporary tables.
- Fixed non-atomicity of updating the replication queue. This could lead to replicas being out of sync until the server

restarts.

- Fixed possible overflow in `gcd` , `lcm` and `modulo` (`%` operator) (Maks Skorokhod).
- `-preprocessed` files are now created after changing `umask` (`umask` can be changed in the config).
- Fixed a bug in the background check of parts (`MergeTreePartChecker`) when using a custom partition key.
- Fixed parsing of tuples (values of the `Tuple` data type) in text formats.
- Improved error messages about incompatible types passed to `multiIf` , `array` and some other functions.
- Redesigned support for `Nullable` types. Fixed bugs that may lead to a server crash. Fixed almost all other bugs related to `NULL` support: incorrect type conversions in INSERT SELECT, insufficient support for Nullable in HAVING and PREWHERE, `join_use_nulls` mode, Nullable types as arguments of `OR` operator, etc.
- Fixed various bugs related to internal semantics of data types. Examples: unnecessary summing of `Enum` type fields in `SummingMergeTree` ; alignment of `Enum` types in `Pretty` formats, etc.
- Stricter checks for allowed combinations of composite columns.
- Fixed the overflow when specifying a very large parameter for the `FixedString` data type.
- Fixed a bug in the `topK` aggregate function in a generic case.
- Added the missing check for equality of array sizes in arguments of n-ary variants of aggregate functions with an `-Array` combinator.
- Fixed a bug in `--pager` for `clickhouse-client` (author: ks1322).
- Fixed the precision of the `exp10` function.
- Fixed the behavior of the `visitParamExtract` function for better compliance with documentation.
- Fixed the crash when incorrect data types are specified.
- Fixed the behavior of `DISTINCT` in the case when all columns are constants.
- Fixed query formatting in the case of using the `tupleElement` function with a complex constant expression as the tuple element index.
- Fixed a bug in `Dictionary` tables for `range_hashed` dictionaries.
- Fixed a bug that leads to excessive rows in the result of `FULL` and `RIGHT JOIN` (Amos Bird).
- Fixed a server crash when creating and removing temporary files in `config.d` directories during config reload.
- Fixed the `SYSTEM DROP DNS CACHE` query: the cache was flushed but addresses of cluster nodes were not updated.
- Fixed the behavior of `MATERIALIZED VIEW` after executing `DETACH TABLE` for the table under the view (Marek Vavruša).

Build improvements:

- The `pbuilder` tool is used for builds. The build process is almost completely independent of the build host environment.
- A single build is used for different OS versions. Packages and binaries have been made compatible with a wide range of Linux systems.
- Added the `clickhouse-test` package. It can be used to run functional tests.
- The source tarball can now be published to the repository. It can be used to reproduce the build without using GitHub.
- Added limited integration with Travis CI. Due to limits on build time in Travis, only the debug build is tested and a limited subset of tests are run.
- Added support for `Cap'n'Proto` in the default build.
- Changed the format of documentation sources from `Restricted Text` to `Markdown` .
- Added support for `systemd` (Vladimir Smirnov). It is disabled by default due to incompatibility with some OS images and can be enabled manually.

- For dynamic code generation, `clang` and `lld` are embedded into the `clickhouse` binary. They can also be invoked as `clickhouse clang` and `clickhouse lld` .
- Removed usage of GNU extensions from the code. Enabled the `-Wextra` option. When building with `clang` the default is `libc++` instead of `libstdc++` .
- Extracted `clickhouse_parsers` and `clickhouse_common_io` libraries to speed up builds of various tools.

Backward incompatible changes:

- The format for marks in `Log` type tables that contain `Nullable` columns was changed in a backward incompatible way. If you have these tables, you should convert them to the `TinyLog` type before starting up the new server version. To do this, replace `ENGINE = Log` with `ENGINE = TinyLog` in the corresponding `.sql` file in the `metadata` directory. If your table doesn't have `Nullable` columns or if the type of your table is not `Log` , then you don't need to do anything.
- Removed the `experimental_allow_extended_storage_definition_syntax` setting. Now this feature is enabled by default.
- The `runningIncome` function was renamed to `runningDifferenceStartingWithFirstvalue` to avoid confusion.
- Removed the `FROM ARRAY JOIN arr` syntax when ARRAY JOIN is specified directly after FROM with no table (Amos Bird).
- Removed the `BlockTabSeparated` format that was used solely for demonstration purposes.
- Changed the state format for aggregate functions `varSamp` , `varPop` , `stddevSamp` , `stddevPop` , `covarSamp` , `covarPop` , `corr` . If you have stored states of these aggregate functions in tables (using the `AggregateFunction` data type or materialized views with corresponding states), please write to clickhouse-feedback@yandex-team.com.
- In previous server versions there was an undocumented feature: if an aggregate function depends on parameters, you can still specify it without parameters in the AggregateFunction data type. Example: `AggregateFunction(quantiles, UInt64)` instead of `AggregateFunction(quantiles(0.5, 0.9), UInt64)` . This feature was lost. Although it was undocumented, we plan to support it again in future releases.
- Enum data types cannot be used in min/max aggregate functions. This ability will be returned in the next release.

Please note when upgrading:

- When doing a rolling update on a cluster, at the point when some of the replicas are running the old version of ClickHouse and some are running the new version, replication is temporarily stopped and the message `unknown parameter 'shard'` appears in the log. Replication will continue after all replicas of the cluster are updated.
- If different versions of ClickHouse are running on the cluster servers, it is possible that distributed queries using the following functions will have incorrect results: `varSamp` , `varPop` , `stddevSamp` , `stddevPop` , `covarSamp` , `covarPop` , `corr` . You should update all cluster nodes.

ClickHouse release 1.1.54327, 2017-12-21

This release contains bug fixes for the previous release 1.1.54318:

- Fixed bug with possible race condition in replication that could lead to data loss. This issue affects versions 1.1.54310 and 1.1.54318. If you use one of these versions with Replicated tables, the update is strongly recommended. This issue shows in logs in Warning messages like `Part ... from own log doesn't exist.` The issue is relevant even if you don't see these messages in logs.

ClickHouse release 1.1.54318, 2017-11-30

This release contains bug fixes for the previous release 1.1.54310:

- Fixed incorrect row deletions during merges in the SummingMergeTree engine

- Fixed a memory leak in unreplicated MergeTree engines
- Fixed performance degradation with frequent inserts in MergeTree engines
- Fixed an issue that was causing the replication queue to stop running
- Fixed rotation and archiving of server logs

ClickHouse release 1.1.54310, 2017-11-01

New features:

- Custom partitioning key for the MergeTree family of table engines.
- Kafka [https://clickhouse.yandex/docs/en/single/index.html#document-table_engines/kafka] table engine.
- Added support for loading CatBoost [https://catboost.yandex/] models and applying them to data stored in ClickHouse.
- Added support for time zones with non-integer offsets from UTC.
- Added support for arithmetic operations with time intervals.
- The range of values for the Date and DateTime types is extended to the year 2105.
- Added the `CREATE MATERIALIZED VIEW x TO y` query (specifies an existing table for storing the data of a materialized view).
- Added the `ATTACH TABLE` query without arguments.
- The processing logic for Nested columns with names ending in -Map in a SummingMergeTree table was extracted to the sumMap aggregate function. You can now specify such columns explicitly.
- Max size of the IP trie dictionary is increased to 128M entries.
- Added the getSizeOfEnumType function.
- Added the sumWithOverflow aggregate function.
- Added support for the Cap'n Proto input format.
- You can now customize compression level when using the zstd algorithm.

Backward incompatible changes:

- Creation of temporary tables with an engine other than Memory is not allowed.
- Explicit creation of tables with the View or MaterializedView engine is not allowed.
- During table creation, a new check verifies that the sampling key expression is included in the primary key.

Bug fixes:

- Fixed hangups when synchronously inserting into a Distributed table.
- Fixed nonatomic adding and removing of parts in Replicated tables.
- Data inserted into a materialized view is not subjected to unnecessary deduplication.
- Executing a query to a Distributed table for which the local replica is lagging and remote replicas are unavailable does not result in an error anymore.
- Users don't need access permissions to the `default` database to create temporary tables anymore.
- Fixed crashing when specifying the Array type without arguments.
- Fixed hangups when the disk volume containing server logs is full.
- Fixed an overflow in the toRelativeWeekNum function for the first week of the Unix epoch.

Build improvements:

- Several third-party libraries (notably Poco) were updated and converted to git submodules.

ClickHouse release 1.1.54304, 2017-10-19

New features:

- TLS support in the native protocol (to enable, set `tcp_ssl_port` in `config.xml`).

Bug fixes:

- `ALTER` for replicated tables now tries to start running as soon as possible.
- Fixed crashing when reading data with the setting `preferred_block_size_bytes=0.`
- Fixed crashes of `clickhouse-client` when pressing `Page Down`
- Correct interpretation of certain complex queries with `GLOBAL IN` and `UNION ALL`
- `FREEZE PARTITION` always works atomically now.
- Empty POST requests now return a response with code 411.
- Fixed interpretation errors for expressions like `CAST(1 AS Nullable(UInt8)).`
- Fixed an error when reading `Array(Nullable(String))` columns from `MergeTree` tables.
- Fixed crashing when parsing queries like `SELECT dummy AS dummy, dummy AS b`
- Users are updated correctly with invalid `users.xml`
- Correct handling when an executable dictionary returns a non-zero response code.

ClickHouse release 1.1.54292, 2017-09-20

New features:

- Added the `pointInPolygon` function for working with coordinates on a coordinate plane.
- Added the `sumMap` aggregate function for calculating the sum of arrays, similar to `SummingMergeTree` .
- Added the `trunc` function. Improved performance of the rounding functions (`round` , `floor` , `ceil` , `roundToExp2`) and corrected the logic of how they work. Changed the logic of the `roundToExp2` function for fractions and negative numbers.
- The ClickHouse executable file is now less dependent on the libc version. The same ClickHouse executable file can run on a wide variety of Linux systems. There is still a dependency when using compiled queries (with the setting `compile = 1` , which is not used by default).
- Reduced the time needed for dynamic compilation of queries.

Bug fixes:

- Fixed an error that sometimes produced `part ... intersects previous part` messages and weakened replica consistency.
- Fixed an error that caused the server to lock up if ZooKeeper was unavailable during shutdown.
- Removed excessive logging when restoring replicas.
- Fixed an error in the UNION ALL implementation.
- Fixed an error in the concat function that occurred if the first column in a block has the Array type.
- Progress is now displayed correctly in the system.merges table.

ClickHouse release 1.1.54289, 2017-09-13

New features:

- `SYSTEM` queries for server administration: `SYSTEM RELOAD DICTIONARY`, `SYSTEM RELOAD DICTIONARIES`, `SYSTEM DROP DNS CACHE`, `SYSTEM SHUTDOWN`, `SYSTEM KILL`.
- Added functions for working with arrays: `concat`, `arraySlice`, `arrayPushBack`, `arrayPushFront`, `arrayPopBack`, `arrayPopFront`.
- Added `root` and `identity` parameters for the ZooKeeper configuration. This allows you to isolate individual users on the same ZooKeeper cluster.
- Added aggregate functions `groupBitAnd`, `groupBitOr`, and `groupBitXor` (for compatibility, they are also available under the names `BIT_AND`, `BIT_OR`, and `BIT_XOR`).
- External dictionaries can be loaded from MySQL by specifying a socket in the filesystem.
- External dictionaries can be loaded from MySQL over SSL (`ssl_cert`, `ssl_key`, `ssl_ca` parameters).
- Added the `max_network_bandwidth_for_user` setting to restrict the overall bandwidth use for queries per user.
- Support for `DROP TABLE` for temporary tables.
- Support for reading `DateTime` values in Unix timestamp format from the `CSV` and `JSONEachRow` formats.
- Lagging replicas in distributed queries are now excluded by default (the default threshold is 5 minutes).
- FIFO locking is used during ALTER: an ALTER query isn't blocked indefinitely for continuously running queries.
- Option to set `umask` in the config file.
- Improved performance for queries with `DISTINCT`.

Bug fixes:

- Improved the process for deleting old nodes in ZooKeeper. Previously, old nodes sometimes didn't get deleted if there were very frequent inserts, which caused the server to be slow to shut down, among other things.
- Fixed randomization when choosing hosts for the connection to ZooKeeper.
- Fixed the exclusion of lagging replicas in distributed queries if the replica is localhost.
- Fixed an error where a data part in a `ReplicatedMergeTree` table could be broken after running `ALTER MODIFY` on an element in a `Nested` structure.
- Fixed an error that could cause SELECT queries to "hang".
- Improvements to distributed DDL queries.
- Fixed the query `CREATE TABLE ... AS <materialized view>`.
- Resolved the deadlock in the `ALTER ... CLEAR COLUMN IN PARTITION` query for `Buffer` tables.
- Fixed the invalid default value for `Enum` s (0 instead of the minimum) when using the `JSONEachRow` and `TSKV` formats.
- Resolved the appearance of zombie processes when using a dictionary with an `executable` source.
- Fixed segfault for the HEAD query.

Improved workflow for developing and assembling ClickHouse:

- You can use `pbuilder` to build ClickHouse.
- You can use `libc++` instead of `libstdc++` for builds on Linux.
- Added instructions for using static code analysis tools: `Coverage`, `clang-tidy`, `cppcheck`.

Please note when upgrading:

- There is now a higher default value for the MergeTree setting `max_bytes_to_merge_at_max_space_in_pool` (the maximum total size of data parts to merge, in bytes): it has increased from 100 GiB to 150 GiB. This might result in large merges running after the server upgrade, which could cause an increased load on the disk subsystem. If the free space available on the server is less than twice the total amount of the merges that are running, this will cause all other merges to stop running, including merges of small data parts. As a result, INSERT requests will fail with the message

"Merges are processing significantly slower than inserts." Use the `SELECT * FROM system.merges` request to monitor the situation. You can also check the `DiskSpaceReservedForMerge` metric in the `system.metrics` table, or in Graphite. You don't need to do anything to fix this, since the issue will resolve itself once the large merges finish. If you find this unacceptable, you can restore the previous value for the `max_bytes_to_merge_at_max_space_in_pool` setting. To do this, go to the section in config.xml, set `<merge_tree>``<max_bytes_to_merge_at_max_space_in_pool>107374182400</max_bytes_to_merge_at_max_space_in_pool>` and restart the server.

ClickHouse release 1.1.54284, 2017-08-29

- This is a bugfix release for the previous 1.1.54282 release. It fixes leaks in the parts directory in ZooKeeper.

ClickHouse release 1.1.54282, 2017-08-23

This release contains bug fixes for the previous release 1.1.54276:

- Fixed `DB::Exception: Assertion violation: !_path.empty()` when inserting into a Distributed table.
- Fixed parsing when inserting in RowBinary format if input data starts with';'.
- Errors during runtime compilation of certain aggregate functions (e.g. `groupArray()`).

Clickhouse Release 1.1.54276, 2017-08-16

New features:

- Added an optional WITH section for a SELECT query. Example query: `WITH 1+1 AS a SELECT a, a*a`
- INSERT can be performed synchronously in a Distributed table: OK is returned only after all the data is saved on all the shards. This is activated by the setting insert_distributed_sync=1.
- Added the UUID data type for working with 16-byte identifiers.
- Added aliases of CHAR, FLOAT and other types for compatibility with the Tableau.
- Added the functions toYYYYMM, toYYYYMMDD, and toYYYYMMDDhhmmss for converting time into numbers.
- You can use IP addresses (together with the hostname) to identify servers for clustered DDL queries.
- Added support for non-constant arguments and negative offsets in the function `substring(str, pos, len).`
- Added the max_size parameter for the `groupArray(max_size)(column)` aggregate function, and optimized its performance.

Main changes:

- Security improvements: all server files are created with 0640 permissions (can be changed viaconfig parameter).
- Improved error messages for queries with invalid syntax.
- Significantly reduced memory consumption and improved performance when merging large sections of MergeTree data.
- Significantly increased the performance of data merges for the ReplacingMergeTree engine.
- Improved performance for asynchronous inserts from a Distributed table by combining multiple source inserts. To enable this functionality, use the setting distributed_directory_monitor_batch_inserts=1.

Backward incompatible changes:

- Changed the binary format of aggregate states of `groupArray(array_column)` functions for arrays.

Complete list of changes:

- Added the `output_format_json_quote_denormals` setting, which enables outputting nan and inf values in JSON

format.

- Optimized stream allocation when reading from a Distributed table.
- Settings can be configured in readonly mode if the value doesn't change.
- Added the ability to retrieve non-integer granules of the MergeTree engine in order to meet restrictions on the block size specified in the preferred_block_size_bytes setting. The purpose is to reduce the consumption of RAM and increase cache locality when processing queries from tables with large columns.
- Efficient use of indexes that contain expressions like `toStartOfHour(x)` for conditions like `toStartOfHour(x) op constexpr.`
- Added new settings for MergeTree engines (the merge_tree section in config.xml):
- replicated_deduplication_window_seconds sets the number of seconds allowed for deduplicating inserts in Replicated tables.
- cleanup_delay_period sets how often to start cleanup to remove outdated data.
- replicated_can_become_leader can prevent a replica from becoming the leader (and assigning merges).
- Accelerated cleanup to remove outdated data from ZooKeeper.
- Multiple improvements and fixes for clustered DDL queries. Of particular interest is the new setting distributed_ddl_task_timeout, which limits the time to wait for a response from the servers in the cluster.
- Improved display of stack traces in the server logs.
- Added the "none" value for the compression method.
- You can use multiple dictionaries_config sections in config.xml.
- It is possible to connect to MySQL through a socket in the file system.
- The system.parts table has a new column with information about the size of marks, in bytes.

Bug fixes:

- Distributed tables using a Merge table now work correctly for a SELECT query with a condition on the `_table` field.
- Fixed a rare race condition in ReplicatedMergeTree when checking data parts.
- Fixed possible freezing on "leader election" when starting a server.
- The max_replica_delay_for_distributed_queries setting was ignored when using a local replica of the data source. This has been fixed.
- Fixed incorrect behavior of `ALTER TABLE CLEAR COLUMN IN PARTITION` when attempting to clean a non-existing column.
- Fixed an exception in the multiIf function when using empty arrays or strings.
- Fixed excessive memory allocations when deserializing Native format.
- Fixed incorrect auto-update of Trie dictionaries.
- Fixed an exception when running queries with a GROUP BY clause from a Merge table when using SAMPLE.
- Fixed a crash of GROUP BY when using distributed_aggregation_memory_efficient=1.
- Now you can specify the database.table in the right side of IN and JOIN.
- Too many threads were used for parallel aggregation. This has been fixed.
- Fixed how the "if" function works with FixedString arguments.
- SELECT worked incorrectly from a Distributed table for shards with a weight of 0. This has been fixed.
- Running `CREATE VIEW IF EXISTS no longer causes crashes.`
- Fixed incorrect behavior when input_format_skip_unknown_fields=1 is set and there are negative numbers.
- Fixed an infinite loop in the `dictGetHierarchy()` function if there is some invalid data in the dictionary.

- Fixed `Syntax error: unexpected (...)` errors when running distributed queries with subqueries in an IN or JOIN clause and Merge tables.
- Fixed an incorrect interpretation of a SELECT query from Dictionary tables.
- Fixed the "Cannot mremap" error when using arrays in IN and JOIN clauses with more than 2 billion elements.
- Fixed the failover for dictionaries with MySQL as the source.

Improved workflow for developing and assembling ClickHouse:

- Builds can be assembled in Arcadia.
- You can use gcc 7 to compile ClickHouse.
- Parallel builds using ccache+distcc are faster now.

ClickHouse release 1.1.54245, 2017-07-04

New features:

- Distributed DDL (for example, `CREATE TABLE ON CLUSTER`)
- The replicated request `ALTER TABLE CLEAR COLUMN IN PARTITION.`
- The engine for Dictionary tables (access to dictionary data in the form of a table).
- Dictionary database engine (this type of database automatically has Dictionary tables available for all the connected external dictionaries).
- You can check for updates to the dictionary by sending a request to the source.
- Qualified column names
- Quoting identifiers using double quotation marks.
- Sessions in the HTTP interface.
- The OPTIMIZE query for a Replicated table can can run not only on the leader.

Backward incompatible changes:

- Removed SET GLOBAL.

Minor changes:

- Now after an alert is triggered, the log prints the full stack trace.
- Relaxed the verification of the number of damaged/extra data parts at startup (there were too many false positives).

Bug fixes:

- Fixed a bad connection "sticking" when inserting into a Distributed table.
- GLOBAL IN now works for a query from a Merge table that looks at a Distributed table.
- The incorrect number of cores was detected on a Google Compute Engine virtual machine. This has been fixed.
- Changes in how an executable source of cached external dictionaries works.
- Fixed the comparison of strings containing null characters.
- Fixed the comparison of Float32 primary key fields with constants.
- Previously, an incorrect estimate of the size of a field could lead to overly large allocations.
- Fixed a crash when querying a Nullable column added to a table using ALTER.
- Fixed a crash when sorting by a Nullable column, if the number of rows is less than LIMIT.
- Fixed an ORDER BY subquery consisting of only constant values.
- Previously, a Replicated table could remain in the invalid state after a failed DROP TABLE.

- Aliases for scalar subqueries with empty results are no longer lost.
- Now a query that used compilation does not fail with an error if the .so file gets damaged.

Fixed in ClickHouse Release 1.1.54388, 2018-06-28

CVE-2018-14668

"remote" table function allowed arbitrary symbols in "user", "password" and "default_database" fields which led to Cross Protocol Request Forgery Attacks.

Credits: Andrey Krasichkov of Yandex Information Security Team

Fixed in ClickHouse Release 1.1.54390, 2018-07-06

CVE-2018-14669

ClickHouse MySQL client had "LOAD DATA LOCAL INFILE" functionality enabled that allowed a malicious MySQL database read arbitrary files from the connected ClickHouse server.

Credits: Andrey Krasichkov and Evgeny Sidorov of Yandex Information Security Team

Fixed in ClickHouse Release 1.1.54131, 2017-01-10

CVE-2018-14670

Incorrect configuration in deb package could lead to unauthorized use of the database.

Credits: the UK's National Cyber Security Centre (NCSC)

©2016–2018 Yandex LLC