

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、Paracel (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

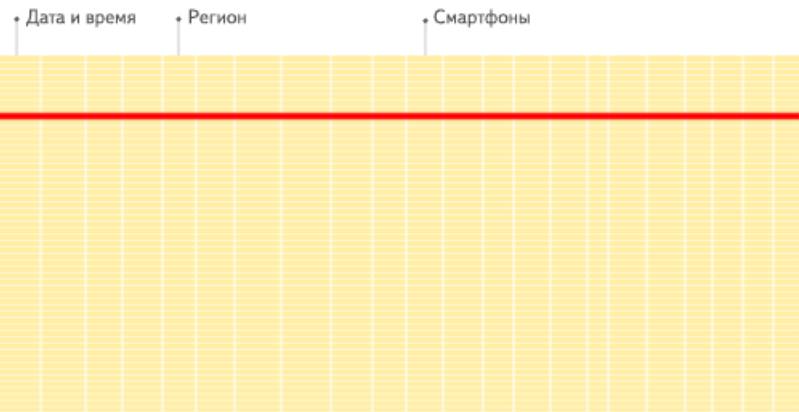
- 较少的查询(通常每台服务器每秒数百个查询或更少)
- 对于简单查询，允许延迟大约50毫秒
- 列中的数据相对较小： 数字和短字符串(例如，每个URL 60个字节)
- 处理单个查询时需要高吞吐量（每个服务器每秒高达数十亿行）
- 事务不是必须的
- 对数据一致性要求低
- 每一个查询除了一个大表外都很小
- 查询结果明显小于源数据，换句话说，数据被过滤或聚合后能够被盛放在单台服务器的内存中

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。

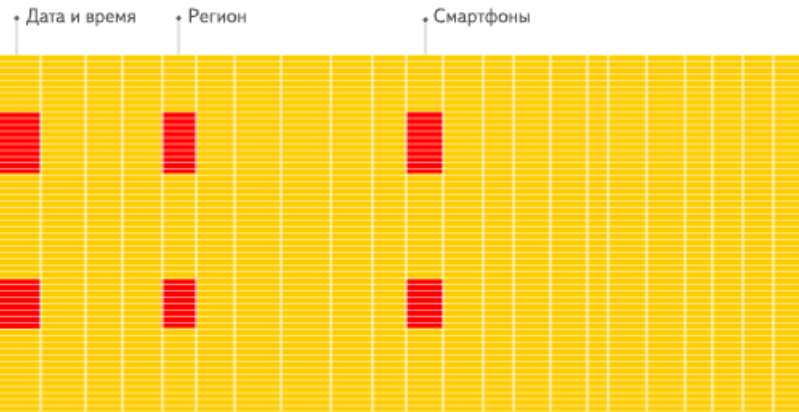
列式数据库更适合OLAP场景的原因

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，所以压缩是很容易的。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。

ClickHouse的独特功能

真正的列式数据库管理系统

在一个真正的列式数据库管理系统中，除了数据本身外不应该存在其他额外的数据。这意味着为了避免在值旁边存储它们的长度“number”，你必须支持固定长度数值类型。例如，10亿个UInt8类型的数据在未压缩的情况下大约消耗1GB左右的空间，如果不是这样的话，这将对CPU的使用产生强烈影响。即使是在未压缩的情况下，紧凑的存储数据也是非常重要的，因为解压缩的速度主要取决于未压缩数据的大小。

这是非常值得注意的，因为在一些其他系统中也可以将不同的列分别进行存储，但由于对其他场景进行的优化，使其无法有效的处理分析查询。例如：HBase，BigTable，Cassandra，HyperTable。在这些系统中，你可以得到每秒数十万的吞吐能力，但是无法得到每秒几亿行的吞吐能力。

需要说明的是，ClickHouse不单单是一个数据库，它是一个数据库管理系统。因为它允许在运行时创建表和数据库、加载数据和运行查询，而无需重新配置或重启服务。

数据压缩

在一些列式数据库管理系统中(例如：InfiniDB CE and MonetDB) 不是用数据压缩。但是, 数据压缩在实现优异的存储系统中确实起着关键的作用。

数据的磁盘存储

许多的列式数据库(如 SAP HANA, Google PowerDrill)只能在内存中工作，这种方式会造成比实际更多的设备预算。ClickHouse被设计用于工作在传统磁盘上的系统，它提供每GB更低的存储成本，但如果有可以使用SSD和内存，它也会合理的利用这些资源。

多核心并行处理

大型查询可以以很自然的方式在ClickHouse中进行并行化处理，以此来使用当前服务器上可用的所有资源。

多服务器分布式处理

上面提到的列式数据库管理系统中，几乎没有一个支持分布式的查询处理。在ClickHouse中，数据可以保存在不同的shard上，每一个shard都由一组用于容错的replica组成，查询可以并行的在所有shard上进行处理。这些对用户来说是透明的

支持SQL

ClickHouse支持基于SQL的查询语言，该语言大部分情况下是与SQL标准兼容的。支持的查询包括 GROUP BY，ORDER BY，IN，JOIN以及非相关子查询。不支持窗口函数和相关子查询。

向量引擎

为了高效的使用CPU，数据不仅仅按列存储，同时还按向量(列的一部分)进行处理。

实时的数据更新

ClickHouse支持在表中定义主键。为了使查询能够快速在主键中进行范围查找，数据总是以增量的方式有序的存储在MergeTree中。因此，数据可以持续不断高效的写入到表中，并且写入的过程中不会存在任何加锁的行为。

索引

按照主键对数据进行排序，这将帮助ClickHouse以几十毫秒的低延迟对数据进行特定值查找或范围查找。

适合在线查询

在线查询意味着在没有对数据做任何预处理的情况下以极低的延迟处理查询并将结果加载到用户的页面中。

支持近似计算

ClickHouse提供各种各样在允许牺牲数据精度的情况下对查询进行加速的方法：

1. 用于近似计算的各类聚合函数，如：distinct values, medians, quantiles
2. 基于数据的部分样本进行近似查询。这时，仅会从磁盘检索少部分比例的数据。
3. 不使用全部的聚合条件，通过随机选择有限个数据聚合条件进行聚合。这在数据聚合条件满足某些分布条件下，在提供相当准确的聚合结果的同时降低了计算资源的使用。

支持数据复制和数据完整性

ClickHouse使用异步的多主复制技术。当数据被写入任何一个可用副本后，系统会在后台将数据分发给其他副本，以保证系统在不同副本上保持相同的数据。在大多数情况下ClickHouse能在故障后自动恢复，在一些复杂的情况下需要少量的手动恢复。

更多信息，参见 [数据复制](#)。

ClickHouse可以考虑缺点的功能

1. 没有完整的事物支持。
2. 缺少高频率，低延迟的修改或删除已存在数据的能力。仅能用于批量删除或修改数据，但这符合 [GDPR](#)。
3. 稀疏索引使得ClickHouse不适合通过其键检索单行的点查询。

Performance

根据Yandex的内部测试结果，ClickHouse表现出了比同类可比较产品更优的性能。你可以在 [这里](#) 查看具体的测试结果。

许多其他的测试也证实这一点。你可以使用互联网搜索到它们，或者你也可以从 [我们收集的部分相关连接](#) 中查看。

单个大查询的吞吐量

吞吐量可以使用每秒处理的行数或每秒处理的字节数来衡量。如果数据被放置在page cache中，则一个不太复杂的查询在单个服务器上大约能够以2-10GB/s（未压缩）的速度进行处理（对于简单的查询，速度可以达到30GB/s）。如果数据没有在page cache中的话，那么速度将取决于你的磁盘系统和数据的压缩率。例如，如果一个磁盘允许以400MB/s的速度读取数据，并且数据压缩率是3，则数据的处理速度为1.2GB/s。这意味着，如果你是在提取一个10字节的列，那么它的处理速度大约是1-2亿行每秒。

对于分布式处理，处理速度几乎是线性扩展的，但这受限于聚合或排序的结果不是那么大的情况下。

处理短查询的延迟时间

如果一个查询使用主键并且没有太多行(几十万)进行处理，并且没有查询太多的列，那么在数据被page cache缓存的情况下，它的延迟应该小于50毫秒(在最佳的情况下应该小于10毫秒)。否则，延迟取决于数据的查找次数。如果你当前使用的是HDD，在数据没有加载的情况下，查询所需要的延迟可以通过以下公式计算得知：查找时间（10 ms）* 查询的列的数量 * 查询的数据块的数量。

处理大量短查询的吞吐量

在相同的情况下，ClickHouse可以在单个服务器上每秒处理数百个查询（在最佳的情况下最多可以处理数千个）。但是由于这不适用于分析型场景。因此我们建议每秒最多查询100次。

数据的写入性能

我们建议每次写入不少于1000行的批量写入，或每秒不超过一个写入请求。当使用tab-separated格式将一份数据写入到MergeTree表中时，写入速度大约为50到200MB/s。如果您写入的数据每行为1Kb，那么写入的速度为50,000到200,000行每秒。如果您的行更小，那么写入速度将更高。为了提高写入性能，您可以使用多个INSERT进行并行写入，这将带来线性的性能提升。

Yandex.Metrica的使用案例

ClickHouse最初是为 **Yandex.Metrica 世界第二大Web分析平台** 而开发的。多年来一直作为该系统的核心组件被该系统持续使用着。目前为止，该系统在ClickHouse中有超过13万亿条记录，并且每天超过200多亿个事件被处理。它允许直接从原始数据中动态查询并生成报告。本文简要介绍了ClickHouse在其早期发展阶段的目标。

Yandex.Metrica基于用户定义的字段，对实时访问、连接会话，生成实时的统计报表。这种需求往往需要复杂聚合方式，比如对访问用户进行去重。构建报表的数据，是实时接收存储的新数据。

截至2014年4月，Yandex.Metrica每天跟踪大约120亿个事件（用户的点击和浏览）。为了可以创建自定义的报表，我们必须存储全部这些事件。同时，这些查询可能需要在几百毫秒内扫描数百万行的数据，或在几秒内扫描数亿行的数据。

Yandex.Metrica以及其他Yandex服务的使用案例

在Yandex.Metrica中，ClickHouse被用于多个场景中。

它的主要任务是使用原始数据在线的提供各种数据报告。它使用374台服务器的集群，存储了20.3万亿行的数据。在去除重复与副本数据的情况下，压缩后的数据达到了2PB。未压缩前（TSV格式）它大概有17PB。

ClickHouse还被使用在：

- 存储来自Yandex.Metrica会话重放数据。
- 处理中间数据
- 与Analytics一起构建全球报表。
- 为调试Yandex.Metrica引擎运行查询
- 分析来自API和用户界面的日志数据

ClickHouse在其他Yandex服务中至少有12个安装：search verticals, Market, Direct, business analytics, mobile development, AdFox, personal services等。

聚合与非聚合数据

有一种流行的观点认为，想要有效的计算统计数据，必须要聚合数据，因为聚合将降低数据量。

但是数据聚合是一个有诸多限制的解决方案，例如：

- 你必须提前知道用户定义的报表的字段列表
- 用户无法自定义报表
- 当聚合条件过多时，可能不会减少数据，聚合是无用的。
- 存在大量报表时，有太多的聚合变化（组合爆炸）
- 当聚合条件有非常大的基数时（如：url），数据量没有太大减少（少于两倍）
- 聚合的数据量可能会增长而不是收缩
- 用户不会查看我们为他生成的所有报告，大部分计算将是无用的
- 各种聚合可能违背了数据的逻辑完整性

如果我们直接使用非聚合数据而不进行任何聚合时，我们的计算量可能是减少的。

然而，相对于聚合中很大一部分工作被离线完成，在线计算需要尽快的完成计算，因为用户在等待结果。

Yandex.Metrica 有一个专门用于聚合数据的系统，称为Metrage，它可以用作大部分报表。

从2009年开始，Yandex.Metrica还为非聚合数据使用专门的OLAP数据库，称为OLAPServer，它以前用于报表构建系统。OLAPServer可以很好的工作在非聚合数据上，但是它有诸多限制，导致无法根据需要将其用于所有报表中。如，缺少对数据类型的支持（只支持数据），无法实时增量的更新数据（只能通过每天重写数据完成）。OLAPServer不是一个数据库管理系统，它只是一个数据库。

为了消除OLAPServer的这些局限性，解决所有报表使用非聚合数据的问题，我们开发了ClickHouse数据库管理系统。

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-get install dirmngr # optional
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --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/>。

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/>

CentOS或RHEL安装包 : <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 2018`
do
for m in `seq 1 12`
do
wget https://transtats.bts.gov/PREZIP/On_Time_Reporting_Carrier_On_Time_Performance_1987_present_${s}_${m}.zip
done
done
```

(引用 <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

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

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>和<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.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.borocode pickup_borocode,
pick_up.ct2010 pickup_ct2010,
pick_up.borocode pickup_borocode,
pick_up.cdligibil pickup_cdligibil,
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.borocode dropoff_borocode,
drop_off.ct2010 dropoff_ct2010,
drop_off.borocode dropoff_borocode,
drop_off.cdligibil dropoff_cdligibil,
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：

```

CREATE TABLE trips

```

```

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',
'RE', '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_boroc2010, '0000000'), 7) AS pickup_boroc2010,
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,
```

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

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

assumeNotNull(dropoff_nyct2010_gid) AS dropoff_nyct2010_gid,

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

assumeNotNull(dropoff_borocode) AS dropoff_borocode,

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

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

toFixedString(ifNull(dropoff_boroct2010, '0000000'), 7) AS dropoff_boroct2010,

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

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

CAST(assumeNotNull(dropoff_ntaname) AS Enum16('' = 0, 'Airport' = 1, 'Allerton-Pelham Gardens' = 2, 'Annadale-Huguenot-Prince's Bay-Eltingville' = 3, 'Arden Heights' = 4, 'Astoria' = 5, 'Auburndale' = 6, 'Baisley Park' = 7, 'Bath Beach' = 8, 'Battery Park City-Lower Manhattan' = 9, 'Bay Ridge' = 10, 'Bayside-Bayside Hills' = 11, 'Bedford' = 12,

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

```

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

```

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/>

需要您在<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
```

```

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 --

```

```
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/>

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 $link | sed -r 's/pagecounts-([0-9]{4})([0-9]{2})[0-9]
{2}-[0-9]+\.gz/1-12/'})/$link; done
ls -l /opt/wikistat/ | grep gz | while read i; do echo $i; gzip -cd /opt/wikistat/$i | ./wikistat-loader --time="$(echo -n $i |
sed -r 's/pagecounts-([0-9]{4})([0-9]{2})([0-9]{2})-([0-9]{2})([0-9]{2})([0-9]{2})\.gz/1-12-13 \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/>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,
```



```
    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：

```
INSERT INTO critео 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 critео_log;  
  
DROP TABLE critео_log;
```

Star Schema Benchmark

Compiling dbgen:

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

Generating data:

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

Creating tables in ClickHouse:

```
CREATE TABLE customer  
(  
    C_CUSTKEY    UInt32,  
    C_NAME      String,  
    C_ADDRESS   String,  
    C_CITY      LowCardinality(String),  
    C_NATION     LowCardinality(String),  
    C_REGION     LowCardinality(String),  
    C_PHONE     String,  
    C_MKTSEGMENT LowCardinality(String)  
)  
ENGINE = MergeTree ORDER BY (C_CUSTKEY);  
  
CREATE TABLE lineorder  
(  
    LO_ORDERKEY    UInt32,  
    LO_PARTKEY      UInt32,  
    LO_SUPPKEY      UInt32,  
    LO_DATE         Date,  
    LO_QUANTITY     UInt32,  
    LO_EXTENDEDPRICE UInt32,  
    LO_DISCOUNT    Float32,  
    LO_AMOUNT       Float32,  
    LO_TAX           Float32,  
    LO_PART         String,  
    LO_SUPP         String,  
    LO_NATION        LowCardinality(String),  
    LO_REGION       LowCardinality(String),  
    LO_MKTSEGMENT    LowCardinality(String),  
    LO_COMMENT       String
```

```

LO_ORDERKEY      UInt32,
LO_LINENUMBER    UInt8,
LO_CUSTKEY       UInt32,
LO_PARTKEY       UInt32,
LO_SUPPKEY       UInt32,
LO_ORDERDATE     Date,
LO_ORDERPRIORITY LowCardinality(String),
LO_SHIPPRIORITY  UInt8,
LO_QUANTITY      UInt8,
LO_EXTENDEDPRICE UInt32,
LO_ORDTOTALPRICE UInt32,
LO_DISCOUNT     UInt8,
LO_REVENUE       UInt32,
LO_SUPPLYCOST    UInt32,
LO_TAX           UInt8,
LO_COMMITDATE    Date,
LO_SHIPMODE      LowCardinality(String)
)
ENGINE = MergeTree PARTITION BY toYear(LO_ORDERDATE) ORDER BY (LO_ORDERDATE, LO_ORDERKEY);

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

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

```

Inserting data:

```

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

```

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

```

SET max_memory_usage = 20000000000, allow_experimental_multiple_joins_emulation = 1;

CREATE TABLE lineorder_flat
ENGINE = MergeTree
PARTITION BY toYear(LO_ORDERDATE)

```

```

ORDER BY (LO_ORDERDATE, LO_ORDERKEY) AS
SELECT *
FROM lineorder
ANY INNER JOIN customer ON LO_CUSTKEY = C_CUSTKEY
ANY INNER JOIN supplier ON LO_SUPPKEY = S_SUPPKEY
ANY INNER JOIN part ON LO_PARTKEY = P_PARTKEY;

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

```

Running the queries:

Q1.1

```

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

```

Q1.2

```

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

```

Q1.3

```

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

```

Q2.1

```

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

```

Q2.2

```

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

```

Q2.3

```

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

```

Q3.1

```

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

```

Q3.2

```

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

```

Q3.3

```

SELECT C_CITY, S_CITY, toYear(LO_ORDERDATE) AS year, sum(LO_REVENUE) AS revenue FROM lineorder_flat WHERE
(C_CITY = 'UNITED KI1' OR C_CITY = 'UNITED KI5') AND (S_CITY = 'UNITED KI1' OR S_CITY = 'UNITED KI5') AND year >=
1992 AND year <= 1997 GROUP BY C_CITY, S_CITY, year ORDER BY year asc, revenue desc;

```

Q3.4

```

SELECT C_CITY, S_CITY, toYear(LO_ORDERDATE) AS year, sum(LO_REVENUE) AS revenue FROM lineorder_flat WHERE
(C_CITY = 'UNITED KI1' OR C_CITY = 'UNITED KI5') AND (S_CITY = 'UNITED KI1' OR S_CITY = 'UNITED KI5') AND
toYYYYMM(LO_ORDERDATE) = '199712' GROUP BY C_CITY, S_CITY, year ORDER BY year asc, revenue desc;

```

Q4.1

```

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

```

Q4.2

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

Q4.3

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

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**，记录在案，易于使用。
- **本地人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 * JDBC驱动程序 * ODBC驱动程序**

还有许多第三方库可供使用ClickHouse：*** 客户端库 * 集成 * 可视界面**

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"**

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` 程序都需要一些时间。

需要消耗几十毫秒时间。

在交互模式下，每条查询过后，您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，或通过是在命令行中使用 `--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`, `дщпщге`, `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](#)".

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**，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 ClickHouse 源代码进行逆向工程 [从这里开始](#)）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
```



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

2. 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"](#) 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"](#) 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"](#) 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"](#) 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"](#) 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	✓	✓
TabSeparatedRaw	X	✓
TabSeparatedWithNames	✓	✓
TabSeparatedWithNamesAndTypes	✓	✓
CSV	✓	✓
CSVWithNames	✓	✓
Values	✓	✓
Vertical	X	✓
VerticalRaw	X	✓
JSON	X	✓
JSONCompact	X	✓
JSONEachRow	✓	✓
TSKV	✓	✓
Pretty	X	✓
PrettyCompact	X	✓
PrettyCompactMonoBlock	X	✓
PrettyNoEscapes	X	✓
PrettySpace	X	✓
RowBinary	✓	✓
Native	✓	✓
Null	X	✓
XML	X	✓
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` 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` 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)。

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
```

&ast默认情况下间隔符是 `,`，在 `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
```

```
{
  "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，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 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

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`），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`），后跟有序的数组元素。

对于 `NULL` 的支持，一个为 1 或 0 的字节会加在每个 `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` 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，空格是允许的（除了数组值之外的空格，这并不允许）。

似。在格式化过程中，个别八进制转空格，但在十进制过程中，空格是假几行并跳过的（原↓数组值八进制的空格，这是个几行的）。**NULL** 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 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` 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](#)。
- 三方提供的 JDBC 驱动 [ClickHouse-Native-JDBC](#)。

ODBC 驱动

- ClickHouse官方有 ODBC 的驱动。见 [这里](#)。

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](#)
 - [clickhouse-driver](#)
 - [clickhouse-client](#)
- PHP
 - [SeasClick](#)
 - [phpClickHouse](#)
 - [clickhouse-php-client](#)
 - [clickhouse-client](#)
 - [PhpClickHouseClient](#)
- Go
 - [clickhouse](#)
 - [go-clickhouse](#)
 - [mailrugo-clickhouse](#)
 - [golang-clickhouse](#)
- NodeJs
 - [clickhouse \(NodeJs\)](#)
 - [node-clickhouse](#)

- [node-clickhouse](#)
- Perl
 - [perl-DBD-ClickHouse](#)
 - [HTTP-ClickHouse](#)
 - [AnyEvent-ClickHouse](#)
- Ruby
 - [clickhouse \(Ruby\)](#)
- R
 - [clickhouse-r](#)
 - [RClickhouse](#)
- Java
 - [clickhouse-client-java](#)
- Scala
 - [clickhouse-scala-client](#)
- Kotlin
 - [AORM](#)
- C#
 - [ClickHouse.Ado](#)
 - [ClickHouse.Net](#)
- C++
 - [clickhouse-cpp](#)
- Elixir
 - [clickhouseex](#)
- Nim
 - [nim-clickhouse](#)

第三方集成库

声明

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。

基建产品

- 关系数据库管理系统
 - [MySQL](#)
 - [ProxySQL](#)
 - [clickhouse-mysql-data-reader](#)
 - [horgh-replicator](#)
 - [PostgreSQL](#)
 - [clickhousedb_fdw](#)
 - [infi.clickhouse_fdw](#) (使用 [infi.clickhouse_orm](#))
 - [pg2ch](#)
 - [MSSQL](#)
 - [ClickHouseMightrator](#)
- 消息队列
 - [Kafka](#)
 - [clickhouse_sinker](#) (使用 [Go client](#))
- 对象存储
 - [S3](#)
 - [clickhouse-backup](#)
- 容器编排
 - [Kubernetes](#)
 - [clickhouse-operator](#)
- 配置管理
 - [puppet](#)

- [innogames/clickhouse](#)
 - [mfedotov/clickhouse](#)
- 监控
 - [Graphite](#)
 - [graphouse](#)
 - [carbon-clickhouse](#)
 - [Grafana](#)
 - [clickhouse-grafana](#)
 - [Prometheus](#)
 - [clickhouse_exporter](#)
 - [PromHouse](#)
 - [Nagios](#)
 - [check_clickhouse](#)
 - [Zabbix](#)
 - [clickhouse-zabbix-template](#)
 - [Sematext](#)
 - [clickhouse积分](#)
- 记录
 - [rsyslog](#)
 - [omclickhouse](#)
 - [fluentd](#)
 - [loghouse](#) (对于 Kubernetes)
 - [logagent](#)
 - [logagent output-plugin-clickhouse](#)
- 地理
 - [MaxMind](#)
 - [clickhouse-maxmind-geoip](#)

编程语言生态系统

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

第三方开发的可视化界面

开源

Tabix

ClickHouse Web 界面 **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.

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

LightHouse

LightHouse 是ClickHouse的轻量级Web界面。

特征：

- 包含过滤和元数据的表列表。
- 带有过滤和排序的表格预览。
- 只读查询执行。

DBeaver

DBeaver 具有ClickHouse支持的通用桌面数据库客户端。

特征：

- 使用语法高亮显示查询开发。
- 表格预览。
- 自动完成。

clickhouse-cli

clickhouse-cli 是ClickHouse的替代命令行客户端，用Python 3编写。

特征：

- 自动完成。

- 查询和数据输出的语法高亮显示。
- 寻呼机支持数据输出。
- 自定义PostgreSQL类命令。

商业

DataGrip

DataGrip 是JetBrains的数据库IDE，专门支持ClickHouse。它还嵌入到其他基于IntelliJ的工具中：PyCharm，IntelliJ IDEA，GoLand，PhpStorm等。

特征：

- 非常快速的代码完成。
- ClickHouse语法高亮显示。
- 支持ClickHouse特有的功能，例如嵌套列，表引擎。
- 数据编辑器。
- 重构。
- 搜索和导航。

来自第三方开发人员的代理服务

chproxy 是ClickHouse数据库的http代理和负载均衡器。

特征

每用户路由和响应缓存。

灵活的限制。

**自动SSL证书续订。*

在Go中实现。

KittenHouse

KittenHouse 设计为ClickHouse和应用程序服务器之间的本地代理，以防在应用程序端缓冲INSERT数据是不可能或不方便的。

特征：

内存和磁盘数据缓冲。

每表路由。

**负载均衡和健康检查。*

在Go中实现。

ClickHouse-Bulk

ClickHouse-Bulk 是一个简单的ClickHouse插入收集器。

特征：

分组请求并按阈值或间隔发送。

多个远程服务器。

**基本身份验证。*

在Go中实现。

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。

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)──┐  
| 0.099999999999999998 |
```

- 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](#) 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。

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)`

- Decimal104(S1) Decimal32(S2) -> Decimal104(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
```



```
SELECT toDecimal2(1, 0) < 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。

UUID

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

The example of UUID type value is represented below:

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

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

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

How to generate

To generate the UUID value, ClickHouse provides the `generateUUIDv4` function.

Usage example

Example 1

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

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

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

```
:) SELECT * FROM t_uuid
```

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

Example 2

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

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

```
:) SELECT * FROM t_uuid
```

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

Restrictions

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

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

Date

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`。

`Enum` 包含在 `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_nullable 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** 或存在 **Nullable** 类型元素,那么数组的元素类型将会变成 **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**。

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 函数可以用来对临时列进行分组。更多信息，请参阅 **IN 操作符** and **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**，那这个元组对应元素是 **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**) 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** 和 **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

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` 的类型为 `Nullable(Nothing)`。详情请见 [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.
```

Domains

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

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

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

Extra Features of Domains

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

Limitations

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

IPv4

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

Basic Usage

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

```
DESCRIBE TABLE hits;
```

```
┌─name─┐┌─type─┐┌─default type─┐┌─default expression─┐┌─comment─┐┌─codec expression─┐
```

name	type	default_type	default_expression	comment	code_expression
url	String				
from	IPv4				

OR you can use IPv4 domain as a key:

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

IPv4 domain supports custom input format as IPv4-strings:

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

```
SELECT * FROM hits;
```

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

Values are stored in compact binary form:

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

toTypeName(from)	hex(from)
IPv4	B7F7E83A

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

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

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

toTypeName(IPv4NumToString(from))	s
String	183.247.232.58

Or cast to a `UInt32` value:

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

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

IPv6

IPv6 is a domain based on `FixedString(16)` type and serves as typed replacement for storing IPv6 values. It

provides compact storage with human-friendly input-output format, and column type information on inspection.

Basic Usage

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

DESCRIBE TABLE hits;
```

name	type	default_type	default_expression	comment	codec_expression
url	String				
from	IPv6				

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

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

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

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

SELECT * FROM hits;
```

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

Values are stored in compact binary form:

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

toTypeName(from)	hex(from)
IPv6	200144C8012926320033000002520002

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

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

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

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

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

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

```
┌toTypeName(CAST(from, 'FixedString(16)'))┐└i┐
├ FixedString(16)                        │ ◆◆◆ │
└────────────────────────────────────────┴──┘
```

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

Clickhouse 中最强大的表引擎当属 MergeTree（合并树）引擎及该系列（*MergeTree）中的其他引擎。

MergeTree 引擎系列的基本理念如下。当你有巨量数据要插入到表中，你要高效地一批批写入数据片段，并希望这些数据片段在后台按照一定规则合并。相比在插入时不断修改（重写）数据进存储，这种策略会高效很多。

主要特点：

- 存储的数据按主键排序。

这让你可以创建一个用于快速检索数据的小稀疏索引。

- 允许使用分区，如果指定了 **主键** 的话。

在相同数据集和相同结果集的情况下 ClickHouse 中某些带分区的操作会比普通操作更快。查询中指定了分区键时 ClickHouse 会自动截取分区数据。这也有效增加了查询性能。

- 支持数据副本。

ReplicatedMergeTree 系列的表便是用于此。更多信息，请参阅 **数据副本** 一节。

- 支持数据采样。

需要的话，你可以给表设置一个采样方法。

注意

Merge 引擎并不属于 *MergeTree 系列。

建表

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
    INDEX index_name1 expr1 TYPE type1(...) GRANULARITY value1,
    INDEX index_name2 expr2 TYPE type2(...) GRANULARITY value2
```

```
INDEX index_name2 expr2 TYPE type2(...) GRANULARITY value2
) ENGINE = MergeTree()
[PARTITION BY expr]
[ORDER BY expr]
[PRIMARY KEY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

请求参数的描述，参考 [请求描述](#)。

子句

- **ENGINE** - 引擎名和参数。 **ENGINE = MergeTree()**. **MergeTree** 引擎没有参数。
- **PARTITION BY** — [分区键](#)。

要按月分区，可以使用表达式 `toYYYYMM(date_column)`，这里的 `date_column` 是一个 **Date** 类型的列。这里该分区名格式会是 `"YYYYMM"` 这样。

- **ORDER BY** — 表的排序键。

可以是一组列的元组或任意的表达式。例如: `ORDER BY (CounterID, EventDate)`。

- **PRIMARY KEY** - 主键，如果要设成 [跟排序键不相同](#)。

默认情况下主键跟排序键（由 **ORDER BY** 子句指定）相同。
因此，大部分情况下不需要再专门指定一个 **PRIMARY KEY** 子句。

- **SAMPLE BY** — 用于抽样的表达式。

如果要用抽样表达式，主键中必须包含这个表达式。例如：

```
SAMPLE BY intHash32(UserID) ORDER BY (CounterID, EventDate, intHash32(UserID))。
```

- **SETTINGS** — 影响 **MergeTree** 性能的额外参数：
 - **index_granularity** — 索引粒度。即索引中相邻『标记』间的数据行数。默认值，**8192**。该列表中所有可用的参数可以从[这里查看 MergeTreeSettings.h](#)。
 - **use_minimalistic_part_header_in_zookeeper** — 数据片段头在 ZooKeeper 中的存储方式。如果设置了 **use_minimalistic_part_header_in_zookeeper=1**，ZooKeeper 会存储更少的数据。更多信息参考『服务配置参数』这章中的 [设置描述](#)。
 - **min_merge_bytes_to_use_direct_io** — 使用直接 I/O 来操作磁盘的合并操作时要求的最小数据量。合并数据片段时，ClickHouse 会计算要被合并的所有数据的总存储空间。如果大小超过了 **min_merge_bytes_to_use_direct_io** 设置的字节数，则 ClickHouse 将使用直接 I/O 接口（**O_DIRECT** 选项）对磁盘读写。如果设置 **min_merge_bytes_to_use_direct_io = 0**，则会禁用直接 I/O。默认值：**10 * 1024 * 1024 * 1024** 字节。

示例配置

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

示例中，我们设为按月分区。

同时我们设置了一个按用户 ID 哈希的抽样表达式。这让你可以有该表中每个 **CounterID** 和 **EventDate** 下面的数据的伪随机分布。如果你在查询时指定了 **SAMPLE** 子句。ClickHouse 会返回对于用户子集的一个均匀的伪随机数据采样。

index_granularity 可省略，默认值为 **8192**。

▼ 已弃用的建表方法

注意

不要在新版项目中使用该方法，可能的话，请将旧项目切换到上述方法。

```
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() 参数

- `date-column` — 类型为 `Date` 的列名。ClickHouse 会自动依据这个列按月创建分区。分区名格式为 `"YYYYMM"`。
- `sampling_expression` — 采样表达式。
- `(primary, key)` — 主键。类型 — `Tuple()`
- `index_granularity` — 索引粒度。即索引中相邻『标记』间的数据行数。设为 `8192` 可以适用大部分场景。

示例

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

对于主要的配置方法，这里 `MergeTree` 引擎跟前面的例子一样，可以以同样的方式配置。

数据存储

表由按主键排序的数据 片段 组成。

当数据被插入到表中时，会分成数据片段并按主键的字典序排序。例如，主键是 `(CounterID, Date)` 时，片段中数据按 `CounterID` 排序，具有相同 `CounterID` 的部分按 `Date` 排序。

不同分区的数据会被分成不同的片段，ClickHouse 在后台合并数据片段以便更高效存储。不会合并来自不同分区的数据片段。这个合并机制并不保证相同主键的所有行都会合并到同一个数据片段中。

ClickHouse 会为每个数据片段创建一个索引文件，索引文件包含每个索引行（『标记』）的主键值。索引行号定义为 `n * index_granularity`。最大的 `n` 等于总行数除以 `index_granularity` 的值的整数部分。对于每列，跟主键相同的索引行处也会写入『标记』。这些『标记』让你可以直接找到数据所在的列。

你可以只用一单一表并不断地一块块往里面加入数据 - `MergeTree` 引擎的就是为了这样的场景。

主键和索引在查询中的表现

我们以 `(CounterID, Date)` 以主键。排序好的索引的图示会是下面这样：

```
全部数据： [-----]
CounterID: [aaaaaaaaaaaaaaaaabbbbcdeeeeeeeeeeeefgggggggghhhhhhhhhiiiiiiiiklllllll]
Date:      [111111122222233331233211111222223332111111212222223111112223311122333]
标记:      | | | | | | | | | | |
           a,1 a,2 a,3 b,3 e,2 e,3 g,1 h,2 i,1 i,3 l,3
标记号:    0  1  2  3  4  5  6  7  8  9  10
```

如果指定查询如下：

- `CounterID in ('a', 'h')`，服务器会读取标记号在 `[0, 3)` 和 `[6, 8)` 区间中的数据。
- `CounterID IN ('a', 'h') AND Date = 3`，服务器会读取标记号在 `[1, 3)` 和 `[7, 8)` 区间中的数据。
- `Date = 3`，服务器会读取标记号在 `[1, 10]` 区间中的数据。

上面例子可以看出使用索引通常会比全表描述要高效。

稀疏索引会引起额外的数据读取。当读取主键单个区间范围的数据时，每个数据块中最多会多读 `index_granularity * 2` 行额外

的数据。大部分情况下，当 `index_granularity = 8192` 时，ClickHouse的性能并不会降级。

稀疏索引让你能操作有巨量行的表。因为这些索引是常驻内存（RAM）的。

ClickHouse 不要求主键惟一。所以，你可以插入多条具有相同主键的行。

主键的选择

主键中列的数量并没有明确的限制。依据数据结构，你应该让主键包含多些或少些列。这样可以：

- 改善索引的性能。

如果当前主键是 `(a, b)`，然后加入另一个 `c` 列，满足下面条件时，则可以改善性能：

- 有带有 `c` 列条件的查询。
- 很长的数据范围（`index_granularity` 的数倍）里 `(a, b)` 都是相同的值，并且这种情况很普遍。换言之，就是加入另一列后，可以让你的查询略过很长的数据范围。

- 改善数据压缩。

ClickHouse 以主键排序片段数据，所以，数据的一致性越高，压缩越好。

- **CollapsingMergeTree** 和 **SummingMergeTree** 引擎里，数据合并时，会有额外的处理逻辑。

在这种情况下，指定一个跟主键不同的 **排序键** 也是有意义的。

长的主键会对插入性能和内存消耗有负面影响，但主键中额外的列并不影响 `SELECT` 查询的性能。

选择跟排序键不一样主键

指定一个跟排序键（用于排序数据片段中行的表达式）

不一样的主键（用于计算写到索引文件的每个标记值的表达式）是可以的。

这种情况下，主键表达式元组必须是排序键表达式元组的一个前缀。

当使用 **SummingMergeTree** 和

AggregatingMergeTree 引擎时，这个特性非常有用。

通常，使用这类引擎时，表里列分两种：**维度** 和 **度量**。

典型的查询是在 `GROUP BY` 并过滤维度的情况下统计度量列的值。

像 **SummingMergeTree** 和 **AggregatingMergeTree**，用相同的排序键值统计行时，

通常会加上所有的维度。结果就是，这键的表达式会是一长串的列组成，

并且这组列还会因为新加维度必须频繁更新。

这种情况下，主键中仅预留少量列保证高效范围扫描，

剩下的维度列放到排序键元组里。这样是合理的。

排序键的修改 是轻量级的操作，因为一个新列同时被加入到表里和排序键后时，已存在的数据片段并不需要修改。由于旧的排序键是新排序键的前缀，并且刚刚添加的列中没有数据，因此在表修改时的数据对于新旧的排序键来说都是有序的。

索引和分区在查询中的应用

对于 `SELECT` 查询，ClickHouse 分析是否可以使用索引。如果 `WHERE/PREWHERE` 子句具有下面这些表达式（作为谓词链接一子项或整个）则可以使用索引：基于主键或分区键的列或表达式的部分的等式或比较运算表达式；基于主键或分区键的列或表达式的固定前缀的 `IN` 或 `LIKE` 表达式；基于主键或分区键的列的某些函数；基于主键或分区键的表达式逻辑表达式。

因此，在索引键的一个或多个区间上快速地跑查询都是可能的。下面例子中，指定标签；指定标签和日期范围；指定标签和日期；指定多个标签和日期范围等运行查询，都会非常快。

当引擎配置如下时：

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

这种情况下，这些查询：

```
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 会依据主键索引剪掉不符合的数据，依据按月分区的分区键剪掉那些不包含符合数据的分区。

上文的查询显示，即使索引用于复杂表达式。因为读表操作是组织好的，所以，使用索引不会比完整扫描慢。

下面这个例子中，不会使用索引。

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

要检查 ClickHouse 执行一个查询时能否使用索引，可设置 `force_index_by_date` 和 `force_primary_key`。

按月分区的分区键是只能读取包含适当范围日期的数据块。这种情况下，数据块会包含很多天（最多整月）的数据。在块中，数据按主键排序，主键第一列可能不包含日期。因此，仅使用日期而没有带主键前缀条件的查询将会导致读取超过这个日期范围。

跳数索引（分段汇总索引，实验性的）

需要设置 `allow_experimental_data_skipping_indices` 为 1 才能使用此索引。（执行 `SET allow_experimental_data_skipping_indices = 1`）。

此索引在 `CREATE` 语句的列部分里定义。

```
INDEX index_name expr TYPE type(...) GRANULARITY granularity_value
```

`*MergeTree` 系列的表都能指定跳数索引。

这些索引是由数据块按粒度分割后的每部分在指定表达式上汇总信息 `granularity_value` 组成（粒度大小用表引擎里 `index_granularity` 的指定）。

这些汇总信息有助于用 `where` 语句跳过大片不满足的数据，从而减少 `SELECT` 查询从磁盘读取的数据量，

示例

```
CREATE TABLE table_name
(
    u64 UInt64,
    i32 Int32,
    s String,
    ...
    INDEX a (u64 * i32, s) TYPE minmax GRANULARITY 3,
    INDEX b (u64 * length(s)) TYPE set(1000) GRANULARITY 4
) ENGINE = MergeTree()
...
```

上例中的索引能让 ClickHouse 执行下面这些查询时减少读取数据量。

```
SELECT count() FROM table WHERE s < 'z'
SELECT count() FROM table WHERE u64 * i32 == 10 AND u64 * length(s) >= 1234
```

索引的可用类型

- `minmax`
存储指定表达式的极值（如果表达式是 `tuple`，则存储 `tuple` 中每个元素的极值），这些信息用于跳过数据块，类似主键。
- `set(max_rows)`
存储指定表达式的唯一值（不超过 `max_rows` 个，`max_rows=0` 则表示『无限制』）。这些信息可用于检查 `WHERE` 表达式是否满足某个数据块。
- `ngrambf_v1(n, size_of_bloom_filter_in_bytes, number_of_hash_functions, random_seed)`
存储包含数据块中所有 `n` 元短语的 **布隆过滤器**。只可用在字符串上。
可用于优化 `equals`，`like` 和 `in` 表达式的性能。
`n` -- 短语长度。
`size_of_bloom_filter_in_bytes` -- 布隆过滤器大小，单位字节。（因为压缩得好，可以指定比较大的值，如256或512）。
`number_of_hash_functions` -- 布隆过滤器中使用的 `hash` 函数的个数。
`random_seed` -- `hash` 函数的随机种子。
- `tokenbf_v1(size_of_bloom_filter_in_bytes, number_of_hash_functions, random_seed)`
跟 `ngrambf_v1` 类似，不同于 `ngrams` 存储字符串指定长度的所有片段。它只存储被非字母数据字符分割的片段。

```
INDEX sample_index (u64 * length(s)) TYPE minmax GRANULARITY 4
INDEX sample_index2 (u64 * length(str), i32 + f64 * 100, date, str) TYPE set(100) GRANULARITY 4
INDEX sample_index3 (lower(str), str) TYPE ngrambf_v1(3, 256, 2, 0) GRANULARITY 4
```

并发数据访问

应对表的并发访问，我们使用多版本机制。换言之，当同时读和更新表时，数据从当前查询到的一组片段中读取。没有冗长的锁。插入不会阻碍读取。

对表的读操作是自动并行的。

数据副本

只有 `MergeTree` 系列里的表可支持副本：

- `ReplicatedMergeTree`
- `ReplicatedSummingMergeTree`
- `ReplicatedReplacingMergeTree`
- `ReplicatedAggregatingMergeTree`
- `ReplicatedCollapsingMergeTree`
- `ReplicatedVersionedCollapsingMergeTree`
- `ReplicatedGraphiteMergeTree`

副本是表级别的，不是整个服务器级的。所以，服务器里可以同时有复制表和非复制表。

副本不依赖分片。每个分片有它自己的独立副本。

对于 `INSERT` 和 `ALTER` 语句操作数据的会在压缩的情况下被复制（更多信息，看 **ALTER**）。

而 `CREATE`，`DROP`，`ATTACH`，`DETACH` 和 `RENAME` 语句只会在单个服务器上执行，不会被复制。

- `The CREATE TABLE` 在运行此语句的服务器上创建一个新的可复制表。如果此表已存在其他服务器上，则给该表添加新副本。
- `The DROP TABLE` 删除运行此查询的服务器上的副本。
- `The RENAME` 重命名一个副本。换句话说，可复制表不同的副本可以有不同的名称。

要使用副本，需在配置文件中设置 `ZooKeeper` 集群的地址。例如：

```
<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>
```

需要 ZooKeeper 3.4.5 或更高版本。

你可以配置任何现有的 ZooKeeper 集群，系统会使用里面的目录来存取元数据（该目录在创建可复制表时指定）。

如果配置文件中没有设置 ZooKeeper，则无法创建复制表，并且任何现有的复制表都将变为只读。

SELECT 查询并不需要借助 ZooKeeper，副本并不影响 **SELECT** 的性能，查询复制表与非复制表速度是一样的。查询分布式表时，ClickHouse 的处理方式可通过设置 **max_replica_delay_for_distributed_queries** 和 **fallback_to_stale_replicas_for_distributed_queries** 修改。

对于每个 **INSERT** 语句，会通过几个事务将十来个记录添加到 ZooKeeper。（确切地说，这是针对每个插入的数据块；每个 **INSERT** 语句的每 **max_insert_block_size = 1048576** 行和最后剩余的都各算作一个块。）相比非复制表，写 zk 会导致 **INSERT** 的延迟略长一些。但只要你按照建议每秒不超过一个 **INSERT** 地批量插入数据，不会有任何问题。一个 ZooKeeper 集群能给整个 ClickHouse 集群支撑协调每秒几百个 **INSERT**。数据插入的吞吐量（每秒的行数）可以跟不用复制的数据一样高。

对于非常大的集群，你可以把不同的 ZooKeeper 集群用于不同的分片。然而，即使 Yandex.Metrica 集群（大约 300 台服务器）也证明还不需要这么做。

复制是多主异步。**INSERT** 语句（以及 **ALTER**）可以发给任意可用的服务器。数据会先插入到执行该语句的服务器上，然后被复制到其他服务器。由于它是异步的，在其他副本上最近插入的数据会有一些延迟。如果部分副本不可用，则数据在其可用时再写入。副本可用的情况下，则延迟时长是通过网络传输压缩数据块所需的时间。

默认情况下，**INSERT** 语句仅等待一个副本写入成功后返回。如果数据只成功写入一个副本后该副本所在的服务器不再存在，则存储的数据会丢失。要启用数据写入多个副本才确认返回，使用 **insert_quorum** 选项。

单个数据块写入是原子的。**INSERT** 的数据按每块最多 **max_insert_block_size = 1048576** 行进行分块，换句话说，如果 **INSERT** 插入的行少于 1048576，则该 **INSERT** 是原子的。

数据块会去重。对于被多次写的相同数据块（大小相同且具有相同顺序的相同行的数据块），该块仅会写入一次。这样设计的原因是万一在网络故障时客户端应用程序不知道数据是否成功写入 DB，此时可以简单地重复 **INSERT**。把相同的数据发送给多个副本 **INSERT** 并不会有问题。因为这些 **INSERT** 是完全相同的（会被去重）。去重参数参看服务器设置 **merge_tree**。（注意：Replicated*MergeTree 才会去重，不需要 zookeeper 的不带 MergeTree 不会去重）

在复制期间，只有要插入的源数据通过网络传输。进一步的数据转换（合并）会在所有副本上以相同的方式进行处理执行。这样可以最大限度地减少网络使用，这意味着即使副本在不同的数据中心，数据同步也能工作良好。（能在不同数据中心中的同步数据是副本机制的主要目标。）

你可以给数据做任意多的副本。Yandex.Metrica 在生产中使用双副本。某一些情况下，给每台服务器都使用 RAID-5 或 RAID-6 和 RAID-10。是一种相对可靠和方便的解决方案。

系统会监视副本数据同步情况，并能在发生故障后恢复。故障转移是自动的（对于小的数据差异）或半自动的（当数据差异很大时，这可能意味是有配置错误）。

创建复制表

在表引擎名称上加上 `Replicated` 前缀。例如：`ReplicatedMergeTree`。

Replicated*MergeTree 参数

- `zoo_path` — ZooKeeper 中该表的路径。
- `replica_name` — ZooKeeper 中的该表的副本名称。

示例：

```
CREATE TABLE table_name
(
    EventDate DateTime,
    CounterID UInt32,
    UserID UInt32
) ENGINE = ReplicatedMergeTree('/clickhouse/tables/{layer}-{shard}/table_name', '{replica}')
PARTITION BY toYYYYMM(EventDate)
ORDER BY (CounterID, EventDate, intHash32(UserID))
SAMPLE BY intHash32(UserID)
```

已弃用的建表语法示例：

```
CREATE TABLE table_name
(
    EventDate DateTime,
    CounterID UInt32,
    UserID UInt32
) ENGINE = ReplicatedMergeTree('/clickhouse/tables/{layer}-{shard}/table_name', '{replica}', EventDate,
intHash32(UserID), (CounterID, EventDate, intHash32(UserID), EventTime), 8192)
```

如上例所示，这些参数可以包含宏替换的占位符，即大括号的部分。它们会被替换为配置文件里 'macros' 那部分配置的值。
示例：

```
<macros>
  <layer>05</layer>
  <shard>02</shard>
  <replica>example05-02-1.yandex.ru</replica>
</macros>
```

“ZooKeeper 中该表的路径”对每个可复制表都要是唯一的。不同分片上的表要有不同的路径。

这种情况下，路径包含下面这些部分：

`/clickhouse/tables/` 是公共前缀，我们推荐使用这个。

`{layer}-{shard}` 是分片标识部分。在此示例中，由于 Yandex.Metrica 集群使用了两级分片，所以它是由两部分组成的。但对于大多数情况来说，你只需保留 `{shard}` 占位符即可，它会替换展开为分片标识。

`table_name` 是该表在 ZooKeeper 中的名称。使其与 ClickHouse 中的表名相同比较好。这里它被明确定义，跟 ClickHouse 表名不一样，它并不会被 `RENAME` 语句修改。

HINT: you could add a database name in front of `table_name` as well. E.g. `db_name.table_name`

副本名称用于标识同一个表分片的不同副本。你可以使用服务器名称，如上例所示。同个分片中不同副本的副本名称要唯一。

你也可以显式指定这些参数，而不是使用宏替换。对于测试和配置小型集群这可能会很方便。但是，这种情况下，则不能使用分布式 DDL 语句（`ON CLUSTER`）。

使用大型集群时，我们建议使用宏替换，因为它可以降低出错的可能性。

在每个副本服务器上运行 `CREATE TABLE` 查询。将创建新的复制表，或给现有表添加新副本。

如果其他副本上已包含了某些数据，在表上添加新副本，则在运行语句后，数据会从其他副本复制到新副本。换句话说，新副本会与其他副本同步。

要删除副本，使用 `DROP TABLE`。但它只删除那个 - 位于运行该语句的服务器上的副本。

故障恢复

如果服务器启动时 ZooKeeper 不可用，则复制表会切换为只读模式。系统会定期尝试去连接 ZooKeeper。

如果在 `INSERT` 期间 ZooKeeper 不可用，或者在与 ZooKeeper 交互时发生错误，则抛出异常。

连接到 ZooKeeper 后，系统会检查本地文件系统中的数据是否与预期的数据集（ZooKeeper 存储此信息）一致。如果存在轻微的不一致，系统会通过副本同步数据来解决。

如果系统检测到损坏的数据片段（文件大小错误）或无法识别的片段（写入文件系统但未记录在 ZooKeeper 中的部分），则会把它们移动到 'detached' 子目录（不会删除）。而副本中其他任何缺少的但正常数据片段都会被复制同步。

注意，ClickHouse 不会执行任何破坏性操作，例如自动删除大量数据。

当服务器启动（或与 ZooKeeper 建立新会话）时，它只检查所有文件的数量和大小。如果文件大小一致但中间某处已有字节被修改过，不会立即被检测到，只有在尝试读取 `SELECT` 查询的数据时才会检测到。该查询会引发校验和不匹配或压缩块大小不一致的异常。这种情况下，数据片段会添加到验证队列中，并在必要时从其他副本中复制。

如果本地数据集与预期数据的差异太大，则会触发安全机制。服务器在日志中记录此内容并拒绝启动。这种情况很可能是配置错误，例如，一个分片上的副本意外配置为别的分片上的副本。然而，此机制的阈值设置得相当低，在正常故障恢复期间可能会出现这种情况。在这种情况下，数据恢复则是半自动模式，通过用户主动操作触发。

要触发启动恢复，可在 ZooKeeper 中创建节点 `/path_to_table/replica_name/flags/force_restore_data`，节点值可以是任何内容，或运行命令来恢复所有的可复制表：

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

然后重启服务器。启动时，服务器会删除这些标志并开始恢复。

在数据完全丢失后的恢复

如果其中一个服务器的所有数据和元数据都消失了，请按照以下步骤进行恢复：

1. 在服务器上安装 ClickHouse。在包含分片标识符和副本的配置文件中正确定义宏配置，如果有用到的话，
2. 如果服务器上有非复制表则必须手动复制，可以从副本服务器上（在 `/var/lib/clickhouse/data/db_name/table_name/` 目录中）复制它们的数据。
3. 从副本服务器上中复制位于 `/var/lib/clickhouse/metadata/` 中的表定义信息。如果在表定义信息中显式指定了分片或副本标识符，请更正它以使其对应于该副本。（另外，启动服务器，然后会在 `/var/lib/clickhouse/metadata/` 中的 .sql 文件中生成所有的 `ATTACH TABLE` 语句。）
4. 要开始恢复，ZooKeeper 中创建节点 `/path_to_table/replica_name/flags/force_restore_data`，节点内容不限，或运行命令来恢复所有复制的表：`sudo -u clickhouse touch /var/lib/clickhouse/flags/force_restore_data`

然后启动服务器（如果它已运行则重启）。数据会从副本中下载。

另一种恢复方式是从 ZooKeeper（`/path_to_table/replica_name`）中删除有数据丢失的副本的所有元信息，然后再按照“[创建可复制表](#)”中的描述重新创建副本。

恢复期间的网络带宽没有限制。特别注意这一点，尤其是要一次恢复很多副本。

MergeTree 转换为 ReplicatedMergeTree

我们使用 `MergeTree` 来表示 `MergeTree` 系列中的所有表引擎，`ReplicatedMergeTree` 同理。

如果你有一个手动同步的 `MergeTree` 表，您可以将其转换为可复制表。如果你已经在 `MergeTree` 表中收集了大量数据，并

且现在要启用复制，则可以执行这些操作。

如果各个副本上的数据不一致，则首先对其进行同步，或者除保留的一个副本外，删除其他所有副本上的数据。

重命名现有的 MergeTree 表，然后使用旧名称创建 ReplicatedMergeTree 表。

将数据从旧表移动到新表（/var/lib/clickhouse/data/db_name/table_name/）目录内的 'detached' 目录中。

然后在其中一个副本上运行 ALTER TABLE ATTACH PARTITION，将这些数据片段添加到工作集中。

ReplicatedMergeTree 转换为 MergeTree

使用其他名称创建 MergeTree 表。将具有 ReplicatedMergeTree 表数据的目录中的所有数据移动到新表的数据目录中。然后删除 ReplicatedMergeTree 表并重新启动服务器。

如果你想在不起动服务器的情况下清除 ReplicatedMergeTree 表：

If you want to get rid of a ReplicatedMergeTree table without launching the server:

- 删除元数据目录中的相应 .sql 文件（/var/lib/clickhouse/metadata/）。
- 删除 ZooKeeper 中的相应路径（/path_to_table/replica_name）。

之后，你可以启动服务器，创建一个 MergeTree 表，将数据移动到其目录，然后重新启动服务器。

当 ZooKeeper 集群中的元数据丢失或损坏时恢复方法

如果 ZooKeeper 中的数据丢失或损坏，如上所述，你可以通过将数据转移到非复制表来保存数据。

自定义分区键

MergeTree 系列的表（包括 可复制表）可以使用分区。基于 MergeTree 表的 物化视图 也支持分区。

一个分区是指按指定规则逻辑组合一起的表的记录集。可以按任意标准进行分区，如按月，按日或按事件类型。为了减少需要操作的数据，每个分区都是分开存储的。访问数据时，ClickHouse 尽量使用这些分区的最小子集。

分区是在 建表的 PARTITION BY expr 子句中指定。分区键可以是关于列的任何表达式。例如，指定按月分区，表达式为 toYYYYMM(date_column)：

```
CREATE TABLE visits
(
    VisitDate Date,
    Hour UInt8,
    ClientID UUID
)
ENGINE = MergeTree()
PARTITION BY toYYYYMM(VisitDate)
ORDER BY Hour;
```

分区键也可以是表达式元组（类似 主键）。例如：

```
ENGINE = ReplicatedCollapsingMergeTree('/clickhouse/tables/name', 'replica1', Sign)
PARTITION BY (toMonday(StartDate), EventType)
ORDER BY (CounterID, StartDate, intHash32(UserID));
```

上例中，我们设置按一周内的事件类型分区。

新数据插入到表中时，这些数据会存储为按主键排序的新片段（块）。插入后 10-15 分钟，同一分区的各个片段会合并为整个片段。

注意

那些有相同分区表达式值的数据片段才会合并。这意味着 你不应该用太精细的分区方案（超过一千个分区）。否则，会因为文件系统上的文件数量和需要找开的文件描述符过多，导致 SELECT 查询效率不佳。

可以通过 `system.parts` 表查看表片段和分区信息。例如，假设我们有一个 `visits` 表，按月分区。对 `system.parts` 表执行 `SELECT`：

```
SELECT
  partition,
  name,
  active
FROM system.parts
WHERE table = 'visits'
```

partition	name	active
201901	201901_1_3_1	0
201901	201901_1_9_2	1
201901	201901_8_8_0	0
201901	201901_9_9_0	0
201902	201902_4_6_1	1
201902	201902_10_10_0	1
201902	201902_11_11_0	1

`partition` 列存储分区的名称。此示例中有两个分区：`201901` 和 `201902`。在 `ALTER ... PARTITION` 语句中你可以使用该列值来指定分区名称。

`name` 列为分区中数据片段的名称。在 `ALTER ATTACH PART` 语句中你可以使用此列值中来指定片段名称。

这里我们拆解下第一部分的名称：`201901_1_3_1`：

- `201901` 是分区名称。
- `1` 是数据块的最小编号。
- `3` 是数据块的最大编号。
- `1` 是块级别（即在由块组成的合并树中，该块在树中的深度）。

注意

旧类型表的片段名称为：`20190117_20190123_2_2_0`（最小日期 - 最大日期 - 最小块编号 - 最大块编号 - 块级别）。

`active` 列为片段状态。`1` 激活状态；`0` 非激活状态。非激活片段是那些在合并到较大片段之后剩余的源数据片段。损坏的数据片段也表示为非活动状态。

正如在示例中所看到的，同一分区中有几个独立的片段（例如，`201901_1_3_1`和`201901_1_9_2`）。这意味着这些片段尚未合并。ClickHouse 大约在插入后15分钟定期报告合并操作，合并插入的数据片段。此外，你也可以使用 `OPTIMIZE` 语句直接执行合并。例：

```
OPTIMIZE TABLE visits PARTITION 201902;
```

partition	name	active
201901	201901_1_3_1	0
201901	201901_1_9_2	1
201901	201901_8_8_0	0
201901	201901_9_9_0	0
201902	201902_4_6_1	0
201902	201902_4_11_2	1
201902	201902_10_10_0	0
201902	201902_11_11_0	0

非激活片段会在合并后的10分钟左右删除。

查看片段和分区信息的另一种方法是进入表的目录：`/var/lib/clickhouse/data/<database>/<table>/`。例如：

```
dev:/var/lib/clickhouse/data/default/visits$ ls -l
total 40
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  1 16:48 201901_1_3_1
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  5 16:17 201901_1_9_2
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  5 15:52 201901_8_8_0
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  5 15:52 201901_9_9_0
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  5 16:17 201902_10_10_0
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  5 16:17 201902_11_11_0
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  5 16:19 201902_4_11_2
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  5 12:09 201902_4_6_1
drwxr-xr-x 2 clickhouse clickhouse 4096 Feb  1 16:48 detached
```

文件夹 `'201901_1_1_0'`、`'201901_1_7_1'` 等是片段的目录。每个片段都与一个对应的分区相关，并且只包含这个月的数据（本例中的表按月分区）。

`detached` 目录存放着使用 **DETACH** 语句从表中分离的片段。损坏的片段也会移到该目录，而不是删除。服务器不使用 `detached` 目录中的片段。可以随时添加，删除或修改此目录中的数据 - 在运行 **ATTACH** 语句前，服务器不会感知到。

注意，在操作服务器时，你不能手动更改文件系统上的片段集或其数据，因为服务器不会感知到这些修改。对于非复制表，可以在服务器停止时执行这些操作，但不建议这样做。对于复制表，在任何情况下都不要更改片段文件。

ClickHouse 支持对分区执行这些操作：删除分区，从一个表复制到另一个表，或创建备份。了解分区的所有操作，请参阅 [分区和片段的操作](#) 一节。

ReplacingMergeTree

该引擎和 **MergeTree** 的不同之处在于它会删除具有相同主键的重复项。

数据的去重只会在合并的过程中出现。合并会在未知的时间在后台进行，因此你无法预先作出计划。有一些数据可能仍未被处理。尽管你可以调用 `OPTIMIZE` 语句发起计划外的合并，但请不要指望使用它，因为 `OPTIMIZE` 语句会引发对大量数据的读和写。

因此，`ReplacingMergeTree` 适用于在后台清除重复的数据以节省空间，但是它不保证没有重复的数据出现。

建表

```
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, ...]
```

请求参数的描述，参考[请求参数](#)。

ReplacingMergeTree Parameters

- `ver` — 版本列。类型为 `UInt*`、`Date` 或 `DateTime`。可选参数。

合并的时候，`ReplacingMergeTree` 从所有具有相同主键的行中选择一行留下：

- 如果 `ver` 列未指定，选择最后一条。

- 如果 `ver` 列已指定，选择 `ver` 值最大的版本。

子句

创建 `ReplacingMergeTree` 表时，需要与创建 `MergeTree` 表时相同的子句。

▼ 已弃用的建表方法

注意

不要在新项目中使用该方法，可能的话，请将旧项目切换到上述方法。

```
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])
```

除了 `ver` 的所有参数都与 `MergeTree` 中的含义相同。

- `ver` - 版本列。可选参数，有关说明，请参阅上文。

SummingMergeTree

该引擎继承自 `MergeTree`。区别在于，当合并 `SummingMergeTree` 表的数据片段时，ClickHouse 会把所有具有相同主键的行合并为一行，该行包含了被合并的行中具有数值数据类型的列的汇总值。如果主键的组合方式使得单个键值对应于大量的行，则可以显著的减少存储空间并加快数据查询的速度。

我们推荐将该引擎和 `MergeTree` 一起使用。例如，在准备做报告的时候，将完整的数据存储在 `MergeTree` 表中，并且使用 `SummingMergeTree` 来存储聚合数据。这种方法可以使你避免因为使用不正确的主键组合方式而丢失有价值的数

建表

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = SummingMergeTree([columns])
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

请求参数的描述，参考 [请求描述](#)。

SummingMergeTree 的参数

- `columns` - 包含了将要被汇总的列的列名的元组。可选参数。
所选的列必须是数值类型，并且不可位于主键中。

如果没有指定 `columns`，ClickHouse 会把所有不在主键中的数值类型的列都进行汇总。

子句

创建 `SummingMergeTree` 表时，需要与创建 `MergeTree` 表时相同的子句。

▼ 已弃用的建表方法

注意

不要在新项目中使用该方法，可能的话，请将旧项目切换到上述方法。

```
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])
```

除 `columns` 外的所有参数都与 `MergeTree` 中的含义相同。

- `columns` — 包含将要被汇总的列的列名的元组。可选参数。有关说明，请参阅上文。

用法示例

考虑如下的表：

```
CREATE TABLE summtt
(
    key UInt32,
    value UInt32
)
ENGINE = SummingMergeTree()
ORDER BY key
```

向其中插入数据：

```
:) INSERT INTO summtt Values(1,1),(1,2),(2,1)
```

ClickHouse 可能不会完整的汇总所有行（[见下文](#)），因此在查询中使用了聚合函数 `sum` 和 `GROUP BY` 子句。

```
SELECT key, sum(value) FROM summtt GROUP BY key
```

key	sum(value)
2	1
1	3

数据处理

当数据被插入到表中时，他们将被原样保存。ClickHouse 定期合并插入的数据片段，并在这个时候对所有具有相同主键的行中的列进行汇总，将这些行替换为包含汇总数据的一行记录。

ClickHouse 会按片段合并数据，以至于不同的数据片段中会包含具有相同主键的行，即单个汇总片段将会是不完整的。因此，聚合函数 `sum()` 和 `GROUP BY` 子句应该在（`SELECT`）查询语句中被使用，如上文中的例子所述。

汇总的通用规则

列中数值类型的值会被汇总。这些列的集合在参数 `columns` 中被定义。

如果用于汇总的所有列中的值均为0，则该行会被删除。

如果列不在主键中且无法被汇总，则会在现有的值中任选一个。

主键所在的列中的值不会被汇总。

AggregateFunction 列中的汇总

对于 **AggregateFunction** 类型的列，ClickHouse 根据对应函数表现为 **AggregatingMergeTree** 引擎的聚合。

嵌套结构

表中可以具有以特殊方式处理的嵌套数据结构。

如果嵌套表的名称以 **Map** 结尾，并且包含至少两个符合以下条件的列：

- 第一列是数值类型 (***Int***, **Date**, **DateTime**)，我们称之为 **key**,
- 其他的列是可计算的 (***Int***, **Float32/64**)，我们称之为 (**values...**),

然后这个嵌套表会被解释为一个 **key => (values...)** 的映射，当合并它们的行时，两个数据集中的元素会被根据 **key** 合并为相应的 (**values...**) 的汇总值。

示例：

```
[(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)]
```

请求数据时，使用 **sumMap(key, value)** 函数来对 **Map** 进行聚合。

对于嵌套数据结构，你无需在列的元组中指定列以进行汇总。

AggregatingMergeTree

该引擎继承自 **MergeTree**，并改变了数据片段的合并逻辑。ClickHouse 会将相同主键的所有行（在一个数据片段内）替换为单个存储一系列聚合函数状态的行。

可以使用 **AggregatingMergeTree** 表来做增量数据统计聚合，包括物化视图的数据聚合。

引擎需使用 **AggregateFunction** 类型来处理所有列。

如果要按一组规则来合并减少行数，则使用 **AggregatingMergeTree** 是合适的。

建表

```
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, ...]
```

语句参数的说明，请参阅 [语句描述](#)。

子句

创建 **AggregatingMergeTree** 表时，需用跟创建 **MergeTree** 表一样的子句。

▼ 已弃用的建表方法

注意

不要在新项目中使用该方法，可能的话，请将旧项目切换到上述方法。

```
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)
```

上面的所有参数跟 `MergeTree` 中的一样。

SELECT 和 INSERT

插入数据，需使用带有聚合 `-State-` 函数的 **INSERT SELECT** 语句。

从 `AggregatingMergeTree` 表中查询数据时，需使用 `GROUP BY` 子句并且要使用与插入时相同的聚合函数，但后缀要改为 `-Merge`。

在 `SELECT` 查询的结果中，对于 ClickHouse 的所有输出格式 `AggregateFunction` 类型的值都实现了特定的二进制表示法。如果直接用 `SELECT` 导出这些数据，例如如用 `TabSeparated` 格式，那么这些导出数据也能直接用 `INSERT` 语句加载导入。

聚物化视图的示例

创建一个跟踪 `test.visits` 表的 `AggregatingMergeTree` 物化视图：

```
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;
```

向 `test.visits` 表中插入数据。

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

数据会同时插入到表和视图中，并且视图 `test.basic` 会将里面的数据聚合。

要获取聚合数据，我们需要在 `test.basic` 视图上执行类似 `SELECT ... GROUP BY ...` 这样的查询：

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

CollapsingMergeTree

该引擎继承于 `MergeTree`，并在数据块合并算法中添加了折叠行的逻辑。

`CollapsingMergeTree` 会异步的删除（折叠）这些除了特定列 `Sign` 有 `1` 和 `-1` 的值以外，其余所有字段的值都相等的成对的行。没有成对的行会被保留。更多的细节请看本文的 [折叠](#) 部分。

因此，该引擎可以显著的降低存储量并提高 `SELECT` 查询效率。

建表

```
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, ...]
```

请求参数的描述，参考[请求参数](#)。

CollapsingMergeTree 参数

- `sign` — 类型列的名称：`1` 是“状态”行，`-1` 是“取消”行。

列数据类型 — `Int8`。

子句

创建 `CollapsingMergeTree` 表时，需要与创建 `MergeTree` 表时相同的[子句](#)。

▼ 已弃用的建表方法

注意

不要在新项目中使用该方法，可能的话，请将旧项目切换到上述方法。

```
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)
```

除了 `sign` 的所有参数都与 `MergeTree` 中的含义相同。

- `sign` — 类型列的名称：`1` 是“状态”行，`-1` 是“取消”行。

列数据类型 — `Int8`。

折叠

数据

考虑你需要为某个对象保存不断变化的数据的情景。似乎为一个对象保存一行记录并在其发生任何变化时更新记录是合乎逻辑的，但是更新操作对 DBMS 来说是昂贵且缓慢的，因为它需要重写存储中的数据。如果你需要快速的写入数据，则更新操作是不可接受的，但是你可以按下面的描述顺序地更新一个对象的变化。

在写入行的时候使用特定的列 `Sign`。如果 `Sign = 1` 则表示这一行是对象的状态，我们称之为“状态”行。如果 `Sign = -1` 则表示是对具有相同属性的状态行的取消，我们称之为“取消”行。

例如，我们想要计算用户在某个站点访问的页面页面数以及他们在那里停留的时间。在某个时候，我们将用户的活动状态写入下面这样的行。

UserID	PageViews	Duration	Sign
4324182021466249494	5	146	1

一段时间后，我们写入下面的两行来记录用户活动的变化。

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

第一行取消了这个对象（用户）的状态。它需要复制被取消的状态行的所有除了 `Sign` 的属性。

第二行包含了当前的状态。

因为我们只需要用户活动的最后状态，这些行

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

可以在折叠对象的失效（老的）状态的时候被删除。`CollapsingMergeTree` 会在合并数据片段的时候做这件事。

为什么我们每次改变需要 2 行可以阅读[算法段](#)。

这种方法的特殊属性

1. 写入的程序应该记住对象的状态从而可以取消它。“取消”字符串应该是“状态”字符串的复制，除了相反的 `Sign`。它增加了存储的初始数据的大小，但使得写入数据更快速。
2. 由于写入的负载，列中长的增长阵列会降低引擎的效率。数据越简单，效率越高。
3. `SELECT` 的结果很大程度取决于对象变更历史的一致性。在准备插入数据时要准确。在不一致的数据中会得到不可预料的结果，例如，像会话深度这种非负指标的负值。

算法

当 ClickHouse 合并数据片段时，每组具有相同主键的连续行被减少到不超过两行，一行 `Sign = 1`（“状态”行），另一行 `Sign = -1`（“取消”行），换句话说，数据项被折叠了。

对每个结果的数据部分 ClickHouse 保存：

1. 第一个“取消”和最后一个“状态”行，如果“状态”和“取消”行的数量匹配
2. 最后一个“状态”行，如果“状态”行比“取消”行多一个。
3. 第一个“取消”行，如果“取消”行比“状态”行多一个。
4. 没有行，在其他所有情况下。

合并会继续，但是 ClickHouse 会把此情况视为逻辑错误并将其记录在服务日志中。这个错误会在相同的数据被插入超过一次时出现。

因此，折叠不应该改变统计数据的结果。

变化逐渐地被折叠，因此最终几乎每个对象都只剩下了最后的状态。

`Sign` 是必须的因为合并算法不保证所有有相同主键的行都会在同一个结果数据片段中，甚至是在同一台物理服务器上。ClickHouse 用多线程来处理 `SELECT` 请求，所以它不能预测结果中行的顺序。如果要从 `CollapsingMergeTree` 表中获取完全“折叠”后的数据，则需要聚合。

要完成折叠，请使用 `GROUP BY` 子句和用于处理符号的聚合函数编写请求。例如，要计算数量，使用 `sum(Sign)` 而不是 `count()`。要计算某物的总和，使用 `sum(Sign * x)` 而不是 `sum(x)`，并添加 `HAVING sum(Sign) > 0` 子句。

聚合体 `count`, `sum` 和 `avg` 可以用这种方式计算。如果一个对象至少有一个未被折叠的状态，则可以计算 `uniq` 聚合。`min` 和 `max` 聚合无法计算，因为 `CollapsingMergeTree` 不会保存折叠状态的值的历史记录。

如果你需要在不进行聚合的情况下获取数据（例如，要检查是否存在最新值与特定条件匹配的行），你可以在 `FROM` 从句中使用 `FINAL` 修饰符。这种方法显然是更低效的。

示例

示例数据:

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

建表:

```
CREATE TABLE UAct
(
    UserID UInt64,
    PageViews UInt8,
    Duration UInt8,
    Sign Int8
)
ENGINE = CollapsingMergeTree(Sign)
ORDER BY UserID
```

插入数据:

```
INSERT INTO UAct VALUES (4324182021466249494, 5, 146, 1)
```

```
INSERT INTO UAct VALUES (4324182021466249494, 5, 146, -1),(4324182021466249494, 6, 185, 1)
```

我们使用两次 `INSERT` 请求来创建两个不同的数据片段。如果我们使用一个请求插入数据，ClickHouse 只会创建一个数据片段且不会执行任何合并操作。

获取数据：

```
SELECT * FROM UAct
```

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

UserID	PageViews	Duration	Sign
4324182021466249494	5	146	1

我们看到了什么，哪里有折叠？

通过两个 `INSERT` 请求，我们创建了两个数据片段。`SELECT` 请求在两个线程中被执行，我们得到了随机顺序的行。没有发生折叠是因为还没有合并数据片段。ClickHouse 在一个我们无法预料的未知时刻合并数据片段。

因此我们需要聚合：

```
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

如果我们不需要聚合并想要强制进行折叠，我们可以在 `FROM` 从句中使用 `FINAL` 修饰语。

```
SELECT * FROM UAct FINAL
```

UserID	PageViews	Duration	Sign
4324182021466249494	6	185	1

这种查询数据的方法是非常低效的。不要在大表中使用它。

VersionedCollapsingMergeTree

This engine:

- Allows quick writing of object states that are continually changing.
- Deletes old object states in the background. This significantly reduces the volume of storage.

See the section [Collapsing](#) for details.

The engine inherits from [MergeTree](#) and adds the logic for collapsing rows to the algorithm for merging data parts. `VersionedCollapsingMergeTree` serves the same purpose as [CollapsingMergeTree](#) but uses a different collapsing algorithm that allows inserting the data in any order with multiple threads. In particular, the `Version` column helps to collapse the rows properly even if they are inserted in the wrong order. In contrast, `CollapsingMergeTree` allows only strictly consecutive insertion.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
  name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
  name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
  ...
) ENGINE = VersionedCollapsingMergeTree(sign, version)
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

For a description of query parameters, see the [query description](#).

Engine Parameters

Engine Parameters

VersionedCollapsingMergeTree(sign, version)

- **sign** — Name of the column with the type of row: **1** is a "state" row, **-1** is a "cancel" row.

The column data type should be **Int8**.

- **version** — Name of the column with the version of the object state.

The column data type should be **UInt***.

Query Clauses

When creating a **VersionedCollapsingMergeTree** table, the same **clauses** are required as when creating a **MergeTree** table.

▼ Deprecated Method for Creating a Table

Attention

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

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

All of the parameters except **sign** and **version** have the same meaning as in **MergeTree**.

- **sign** — Name of the column with the type of row: **1** is a "state" row, **-1** is a "cancel" row.

Column Data Type — **Int8**.

- **version** — Name of the column with the version of the object state.

The column data type should be **UInt***.

Collapsing

Data

Consider a situation where you need to save continually changing data for some object. It is reasonable to have one row for an object and update the row whenever there are changes. However, the update operation is expensive and slow for a DBMS because it requires rewriting the data in the storage. Update is not acceptable if you need to write data quickly, but you can write the changes to an object sequentially as follows.

Use the **Sign** column when writing the row. If **Sign = 1** it means that the row is a state of an object (let's call it the "state" row). If **Sign = -1** it indicates the cancellation of the state of an object with the same attributes (let's call it the "cancel" row). Also use the **Version** column, which should identify each state of an object with a separate number.

For example, we want to calculate how many pages users visited on some site and how long they were there. At some point in time we write the following row with the state of user activity:

```
┌───────────┴───┐UserID├──PageViews├──Duration├──Sign├──Version└──┐
```

4324182021466249494	5	146	1	1
---------------------	---	-----	---	---

At some point later we register the change of user activity and write it with the following two rows.

UserID	PageViews	Duration	Sign	Version
4324182021466249494	5	146	-1	1
4324182021466249494	6	185	1	2

The first row cancels the previous state of the object (user). It should copy all of the fields of the canceled state except `Sign`.

The second row contains the current state.

Because we need only the last state of user activity, the rows

UserID	PageViews	Duration	Sign	Version
4324182021466249494	5	146	1	1
4324182021466249494	5	146	-1	1

can be deleted, collapsing the invalid (old) state of the object. `VersionedCollapsingMergeTree` does this while merging the data parts.

To find out why we need two rows for each change, see [Algorithm](#).

Notes on Usage

1. The program that writes the data should remember the state of an object in order to cancel it. The "cancel" string should be a copy of the "state" string with the opposite `Sign`. This increases the initial size of storage but allows to write the data quickly.
2. Long growing arrays in columns reduce the efficiency of the engine due to the load for writing. The more straightforward the data, the better the efficiency.
3. `SELECT` results depend strongly on the consistency of the history of object changes. Be accurate when preparing data for inserting. You can get unpredictable results with inconsistent data, such as negative values for non-negative metrics like session depth.

Algorithm

When ClickHouse merges data parts, it deletes each pair of rows that have the same primary key and version and different `Sign`. The order of rows does not matter.

When ClickHouse inserts data, it orders rows by the primary key. If the `Version` column is not in the primary key, ClickHouse adds it to the primary key implicitly as the last field and uses it for ordering.

Selecting Data

ClickHouse doesn't guarantee that all of the rows with the same primary key will be in the same resulting data part or even on the same physical server. This is true both for writing the data and for subsequent merging of the data parts. In addition, ClickHouse processes `SELECT` queries with multiple threads, and it cannot predict the order of rows in the result. This means that aggregation is required if there is a need to get completely "collapsed" data from a `VersionedCollapsingMergeTree` table.

To finalize collapsing, write a query with a `GROUP BY` clause and aggregate functions that account for the sign. For example, to calculate quantity, use `sum(Sign)` instead of `count()`. To calculate the sum of something, use `sum(Sign * x)` instead of `sum(x)`, and add `HAVING sum(Sign) > 0`.

The aggregates `count`, `sum` and `avg` can be calculated this way. The aggregate `uniq` can be calculated if an object has at least one non-collapsed state. The aggregates `min` and `max` can't be calculated because `VersionedCollapsingMergeTree` does not save the history of values of collapsed states.

If you need to extract the data with "collapsing" but without aggregation (for example, to check whether rows are present whose newest values match certain conditions), you can use the `FINAL` modifier for the `FROM` clause. This approach is inefficient and should not be used with large tables.

Example of Use

Example data:

UserID	PageViews	Duration	Sign	Version
4324182021466249494	5	146	1	1
4324182021466249494	5	146	-1	1
4324182021466249494	6	185	1	2

Creating the table:

```
CREATE TABLE UAct
(
    UserID UInt64,
    PageViews UInt8,
    Duration UInt8,
    Sign Int8,
    Version UInt8
)
ENGINE = VersionedCollapsingMergeTree(Sign, Version)
ORDER BY UserID
```

Inserting the data:

```
INSERT INTO UAct VALUES (4324182021466249494, 5, 146, 1, 1)
```

```
INSERT INTO UAct VALUES (4324182021466249494, 5, 146, -1, 1),(4324182021466249494, 6, 185, 1, 2)
```

We use two `INSERT` queries to create two different data parts. If we insert the data with a single query, ClickHouse creates one data part and will never perform any merge.

Getting the data:

```
SELECT * FROM UAct
```

UserID	PageViews	Duration	Sign	Version
4324182021466249494	5	146	1	1
4324182021466249494	5	146	-1	1
4324182021466249494	6	185	1	2

What do we see here and where are the collapsed parts?

We created two data parts using two `INSERT` queries. The `SELECT` query was performed in two threads, and

the result is a random order of rows.

Collapsing did not occur because the data parts have not been merged yet. ClickHouse merges data parts at an unknown point in time which we cannot predict.

This is why we need aggregation:

```
SELECT
  UserID,
  sum(PageViews * Sign) AS PageViews,
  sum(Duration * Sign) AS Duration,
  Version
FROM UAct
GROUP BY UserID, Version
HAVING sum(Sign) > 0
```

UserID	PageViews	Duration	Version
4324182021466249494	6	185	2

If we don't need aggregation and want to force collapsing, we can use the `FINAL` modifier for the `FROM` clause.

```
SELECT * FROM UAct FINAL
```

UserID	PageViews	Duration	Sign	Version
4324182021466249494	6	185	1	2

This is a very inefficient way to select data. Don't use it for large tables.

GraphiteMergeTree

This engine is designed for thinning and aggregating/averaging (rollup) **Graphite** data. It may be helpful to developers who want to use ClickHouse as a data store for Graphite.

You can use any ClickHouse table engine to store the Graphite data if you don't need rollup, but if you need a rollup use `GraphiteMergeTree`. The engine reduces the volume of storage and increases the efficiency of queries from Graphite.

The engine inherits properties from **MergeTree**.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
  Path String,
  Time DateTime,
  Value <Numeric_type>,
  Version <Numeric_type>
  ...
) ENGINE = GraphiteMergeTree(config_section)
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

For a description of request parameters, see [request description](#).

A table for the Graphite data should have the following columns:

- Column with the metric name (Graphite sensor). Data type: `String`.
- Column with the time of measuring the metric. Data type: `DateTime`.
- Column with the value of the metric. Data type: any numeric.
- Column with the version of the metric. Data type: any numeric.

ClickHouse saves the rows with the highest version or the last written if versions are the same. Other rows are deleted during the merge of data parts.

The names of these columns should be set in the rollup configuration.

GraphiteMergeTree parameters

- `config_section` — Name of the section in the configuration file, where are the rules of rollup set.

Query clauses

When creating a `GraphiteMergeTree` table, the same [clauses](#) are required, as when creating a `MergeTree` table.

▼ Deprecated Method for Creating a Table

Attention

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

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    EventDate Date,
    Path String,
    Time DateTime,
    Value <Numeric_type>,
    Version <Numeric_type>
    ...
) ENGINE [=] GraphiteMergeTree(date-column [, sampling_expression], (primary, key), index_granularity, config_section)
```

All of the parameters excepting `config_section` have the same meaning as in `MergeTree`.

- `config_section` — Name of the section in the configuration file, where are the rules of rollup set.

Rollup configuration

The settings for rollup are defined by the [graphite_rollup](#) parameter in the server configuration. The name of the parameter could be any. You can create several configurations and use them for different tables.

Rollup configuration structure:

```
required-columns
pattern
  regexp
  function
pattern
  regexp
  age + precision
...
pattern
  regexp
```

```

    regexp
    function
    age + precision
    ...
pattern
    ...
default
    function
    age + precision
    ...

```

Important: The order of patterns should be next:

1. Patterns *without* function *or* retention.
2. Patterns *with* both function *and* retention.
3. Pattern default.

When processing a row, ClickHouse checks the rules in the `pattern` sections. Each of `pattern` (including `default`) sections could contain `function` parameter for aggregation, `retention` parameters or both. If the metric name matches the `regexp`, the rules from the `pattern` section (or sections) are applied; otherwise, the rules from the `default` section are used.

Fields for `pattern` and `default` sections:

- `regexp` – A pattern for the metric name.
- `age` – The minimum age of the data in seconds.
- `precision` – How precisely to define the age of the data in seconds. Should be a divisor for 86400 (seconds in a day).
- `function` – The name of the aggregating function to apply to data whose age falls within the range [`age`, `age + precision`].

The `required-columns`:

- `path_column_name` — Column with the metric name (Graphite sensor).
- `time_column_name` — Column with the time of measuring the metric.
- `value_column_name` — Column with the value of the metric at the time set in `time_column_name`.
- `version_column_name` — Column with the version of the metric.

Example of settings:

```

<graphite_rollup>
  <path_column_name>Path</path_column_name>
  <time_column_name>Time</time_column_name>
  <value_column_name>Value</value_column_name>
  <version_column_name>Version</version_column_name>
  <pattern>
    <regexp>click_cost</regexp>
    <function>any</function>
    <retention>
      <age>0</age>
      <precision>5</precision>
    </retention>
    <retention>
      <age>86400</age>
      <precision>60</precision>
    </retention>
  </pattern>
  <default>
    <function>max</function>
    <retention>

```



```
<age>0</age>
<precision>60</precision>
</retention>
<retention>
  <age>3600</age>
  <precision>300</precision>
</retention>
<retention>
  <age>86400</age>
  <precision>3600</precision>
</retention>
</default>
</graphite_rollup>
```

日志引擎系列

这些引擎是为了需要写入许多小数据量（少于一百万行）的表的场景而开发的。

这系列的引擎有：

- **StripeLog**
- **Log**
- **TinyLog**

共同属性

引擎：

- 数据存储于磁盘上。
- 写入时将数据追加在文件末尾。
- 不支持**突变**操作。
- 不支持索引。

这意味着 **SELECT** 在范围查询时效率不高。

- 非原子地写入数据。

如果某些事情破坏了写操作，例如服务器的异常关闭，你将会得到一张包含了损坏数据的表。

差异

Log 和 **StripeLog** 引擎支持：

- 并发访问数据的锁。

INSERT 请求执行过程中表会被锁定，并且其他的读写数据的请求都会等待直到锁定被解除。如果没有写数据的请求，任意数量的读请求都可以并发执行。

- 并行读取数据。

在读取数据时，ClickHouse 使用多线程。每个线程处理不同的数据块。

Log 引擎为表中的每一列使用不同的文件。**StripeLog** 将所有数据存储在一个文件中。因此 **StripeLog** 引擎在操作系统中使用更少的描述符，但是 **Log** 引擎提供更高的读性能。

TinyLog 引擎是该系列中最简单的引擎并且提供了最少的功能和最低的性能。**TinyLog** 引擎不支持并行读取和并发数据访问，并将每一列存储在不同的文件中。它比其余两种支持并行读取的引擎的读取速度更慢，并且使用了和 **Log** 引擎同样多的描述符。你可以在简单的低负载的情景下使用它。

StripeLog

该引擎属于日志引擎系列。请在[日志引擎系列](#)文章中查看引擎的共同属性和差异。

在你需要写入许多小数据量（小于一百万行）的表的场景下使用这个引擎。

建表

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    column1_name [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    column2_name [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = StripeLog
```

查看 [建表](#) 请求的详细说明。

写数据

`StripeLog` 引擎将所有列存储在一个文件中。对每一次 `Insert` 请求，ClickHouse 将数据块追加在表文件的末尾，逐列写入。

ClickHouse 为每张表写入以下文件：

- `data.bin` — 数据文件。
- `index.mrk` — 带标记的文件。标记包含了已插入的每个数据块中每列的偏移量。

`StripeLog` 引擎不支持 `ALTER UPDATE` 和 `ALTER DELETE` 操作。

读数据

带标记的文件使得 ClickHouse 可以并行的读取数据。这意味着 `SELECT` 请求返回行的顺序是不可预测的。使用 `ORDER BY` 子句对行进行排序。

使用示例

建表：

```
CREATE TABLE stripe_log_table
(
    timestamp DateTime,
    message_type String,
    message String
)
ENGINE = StripeLog
```

插入数据：

```
INSERT INTO stripe_log_table VALUES (now(),'REGULAR','The first regular message')
INSERT INTO stripe_log_table VALUES (now(),'REGULAR','The second regular message'),(now(),'WARNING','The first warning message')
```

我们使用两次 `INSERT` 请求从而在 `data.bin` 文件中创建两个数据块。

ClickHouse 在查询数据时使用多线程。每个线程读取单独的数据块并在完成后独立的返回结果行。这样的结果是，大多数情况下，输出中块的顺序和输入时相应块的顺序是不同的。例如：

```
SELECT * FROM stripe_log_table
```

```
┌──timestamp──┴──message_type──┴──message──┐
| 2019-01-18 14:27:32 | REGULAR    | The second regular message |
```

timestamp	message_type	message
2019-01-18 14:34:53	WARNING	The first warning message
2019-01-18 14:23:43	REGULAR	The first regular message

对结果排序（默认增序）：

```
SELECT * FROM stripe_log_table ORDER BY timestamp
```

timestamp	message_type	message
2019-01-18 14:23:43	REGULAR	The first regular message
2019-01-18 14:27:32	REGULAR	The second regular message
2019-01-18 14:34:53	WARNING	The first warning message

Log

日志与 TinyLog 的不同之处在于，"标记" 的小文件与列文件存在一起。这些标记写在每个数据块上，并且包含偏移量，这些偏移量指示从哪里开始读取文件以便跳过指定的行数。这使得可以在多个线程中读取表数据。对于并发数据访问，可以同时执行读取操作，而写入操作则阻塞读取和其它写入。**Log** 引擎不支持索引。同样，如果写入表失败，则该表将被破坏，并且从该表读取将返回错误。**Log** 引擎适用于临时数据，**write-once** 表以及测试或演示目的。

TinyLog

最简单的表引擎，用于将数据存储磁盘上。每列都存储在单独的压缩文件中。写入时，数据将附加到文件末尾。

并发数据访问不受任何限制：

- 如果同时从表中读取并在不同的查询中写入，则读取操作将抛出异常
- 如果同时写入多个查询中的表，则数据将被破坏。

这种表引擎的典型用法是 **write-once**：首先只写入一次数据，然后根据需要多次读取。查询在单个流中执行。换句话说，此引擎适用于相对较小的表（建议最多1,000,000行）。如果您有许多小表，则使用此表引擎是适合的，因为它比**Log**引擎更简单（需要打开的文件更少）。当您拥有大量小表时，可能会导致性能低下，但在可能已经在其它 **DBMS** 时使用过，则您可能会发现切换使用 **TinyLog** 类型的表更容易。**不支持索引**。

在 Yandex.Metrica 中，TinyLog 表用于小批量处理的中间数据。

Kafka

此引擎与 **Apache Kafka** 结合使用。

Kafka 特性：

- 发布或者订阅数据流。
- 容错存储机制。
- 处理流数据。

老版格式：

```
Kafka(kafka_broker_list, kafka_topic_list, kafka_group_name, kafka_format
[, kafka_row_delimiter, kafka_schema, kafka_num_consumers])
```

新版格式：

```
Kafka SETTINGS
```

```
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
```

必要参数：

- `kafka_broker_list` - 以逗号分隔的 `brokers` 列表 (`localhost:9092`)。
- `kafka_topic_list` - `topic` 列表 (`my_topic`)。
- `kafka_group_name` - Kafka 消费组名称 (`group1`)。如果不希望消息在集群中重复，请在每个分片中使用相同的组名。
- `kafka_format` - 消息体格式。使用与 SQL 部分的 `FORMAT` 函数相同表示方法，例如 `JSONEachRow`。了解详细信息，请参考 `Formats` 部分。

可选参数：

- `kafka_row_delimiter` - 每个消息体（记录）之间的分隔符。
- `kafka_schema` - 如果解析格式需要一个 `schema` 时，此参数必填。例如，`Cap'n Proto` 需要 `schema` 文件路径以及根对象 `schema.capnp:Message` 的名字。
- `kafka_num_consumers` - 单个表的消费者数量。默认值是：`1`，如果一个消费者的吞吐量不足，则指定更多的消费者。消费者的总数不应该超过 `topic` 中分区的数量，因为每个分区只能分配一个消费者。

示例：

```
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;
```

消费的消息会被自动追踪，因此每个消息在不同的消费组里只会记录一次。如果希望获得两次数据，则使用另一个组名创建副本。

消费组可以灵活配置并且在集群之间同步。例如，如果群集中有10个主题和5个表副本，则每个副本将获得2个主题。如果副本数量发生变化，主题将自动在副本中重新分配。了解更多信息请访问 <http://kafka.apache.org/intro>。

`SELECT` 查询对于读取消息并不是很有用（调试除外），因为每条消息只能被读取一次。使用物化视图创建实时线程更实用。

您可以这样做：

1. 使用引擎创建一个 **Kafka** 消费者并作为一条数据流。
2. 创建一个结构表。
3. 创建物化视图，改视图会在后台转换引擎中的数据并将其放入之前创建的表中。

当 **MATERIALIZED VIEW** 添加至引擎，它将会在后台收集数据。可以持续不断地从 **Kafka** 收集数据并通过 **SELECT** 将数据转换为所需要的格式。

示例：

```
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;
```

为了提高性能，接受的消息被分组为 **max_insert_block_size** 大小的块。如果未在 **stream_flush_interval_ms** 毫秒内形成块，则不关心块的完整性，都会将数据刷新到表中。

停止接收主题数据或更改转换逻辑，请 **detach** 物化视图：

```
DETACH TABLE consumer;  
ATTACH MATERIALIZED VIEW consumer;
```

如果使用 **ALTER** 更改目标表，为了避免目标表与视图中的数据之间存在差异，推荐停止物化视图。

配置

与 **GraphiteMergeTree** 类似，**Kafka** 引擎支持使用 ClickHouse 配置文件进行扩展配置。可以使用两个配置键：全局 (**kafka**) 和主题级别 (**kafka_***)。首先应用全局配置，然后应用主题级配置（如果存在）。

```
<!-- 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>
```

有关详细配置选项列表，请参阅 [librdkafka configuration reference](#)。在 ClickHouse 配置中使用下划线 (**_**)，并不是使用点 (**.**)。例如，**check.crcs=true** 将是 **<check_crcs>true</check_crcs>**。

MySQL

MySQL 引擎可以对存储在远程 MySQL 服务器上的数据执行 `SELECT` 查询。

调用格式：

```
MySQL('host:port', 'database', 'table', 'user', 'password', replace_query, 'on_duplicate_clause');
```

调用参数

- `host:port` — MySQL 服务器地址。
- `database` — 数据库的名称。
- `table` — 表名称。
- `user` — 数据库用户。
- `password` — 用户密码。
- `replace_query` — 将 `INSERT INTO` 查询是否替换为 `REPLACE INTO` 的标志。如果 `replace_query=1`，则替换查询
- `'on_duplicate_clause'` — 将 `ON DUPLICATE KEY UPDATE 'on_duplicate_clause'` 表达式添加到 `INSERT` 查询语句中。例如：`impression = VALUES(impression) + impression`。如果需要指定 `'on_duplicate_clause'`，则需要设置 `replace_query=0`。如果同时设置 `replace_query = 1` 和 `'on_duplicate_clause'`，则会抛出异常。

此时，简单的 `WHERE` 子句（例如 `=, !=, >, >=, <, <=`）是在 MySQL 服务器上执行。

其余条件以及 `LIMIT` 采样约束语句仅在对 MySQL 的查询完成后才在 ClickHouse 中执行。

MySQL 引擎不支持 `Nullable` 数据类型，因此，当从 MySQL 表中读取数据时，`NULL` 将转换为指定列类型的默认值（通常为 0 或空字符串）。

Distributed

分布式引擎本身不存储数据，但可以在多个服务器上进行分布式查询。

读是自动并行的。读取时，远程服务器表的索引（如果有的话）会被使用。

分布式引擎参数：服务器配置文件中的集群名，远程数据库名，远程表名，数据分片键（可选）。

示例：

```
Distributed(logs, default, hits[, sharding_key])
```

将会从位于“logs”集群中 `default.hits` 表所有服务器上读取数据。

远程服务器不仅用于读取数据，还会对尽可能数据做部分处理。

例如，对于使用 `GROUP BY` 的查询，数据首先在远程服务器聚合，之后返回聚合函数的中间状态给查询请求的服务器。再在请求的服务器上进一步汇总数据。

数据库名参数除了用数据库名之外，也可用返回字符串的常量表达式。例如：`currentDatabase()`。

logs – 服务器配置文件中的集群名称。

集群示例配置如下：

```
<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>
```

```

    </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>

```

这里定义了一个名为‘logs’的集群，它由两个分片组成，每个分片包含两个副本。分片是指包含数据不同部分的服务器（要读取所有数据，必须访问所有分片）。副本是存储复制数据的服务器（要读取所有数据，访问任一副本上的数据即可）。

集群名称不能包含点号。

每个服务器需要指定 **host**，**port**，和可选的 **user**，**password**，**secure**，**compression** 的参数：

- **host** - 远程服务器地址。可以域名、IPv4或IPv6。如果指定域名，则服务在启动时发起一个 DNS 请求，并且请求结果会在服务器运行期间一直被记录。如果 DNS 请求失败，则服务不会启动。如果你修改了 DNS 记录，则需要重启服务。
- **port** - 消息传递的 TCP 端口（「tcp_port」配置通常设为 9000）。不要跟 http_port 混淆。
- **user** - 用于连接远程服务器的用户名。默认值：default。该用户必须有权限访问该远程服务器。访问权限配置在 users.xml 文件中。更多信息，请查看“访问权限”部分。
- **password** - 用于连接远程服务器的密码。默认值：空字符串。
- **secure** - 是否使用ssl进行连接，设为true时，通常也应该设置 **port** = 9440。服务器也要监听 9440 并有正确的证书。
- **compression** - 是否使用数据压缩。默认值：true。

配置了副本，读取操作会从每个分片里选择一个可用的副本。可配置负载均衡算法（挑选副本的方式） - 请参阅“load_balancing”设置。

如果跟服务器的连接不可用，则在尝试超时的重连。如果重连失败，则选择下一个副本，依此类推。如果跟所有副本的连接尝试都失败，则尝试用相同的方式再重复几次。

该机制有利于系统可用性，但不保证完全容错：如有远程服务器能够接受连接，但无法正常工作或状况不佳。

你可以配置一个（这种情况下，查询操作更应该称为远程查询，而不是分布式查询）或任意多个分片。在每个分片中，可以配置一个或任意多个副本。不同分片可配置不同数量的副本。

可以在配置中配置任意数量的集群。

要查看集群，可使用“system.clusters”表。

通过分布式引擎可以像使用本地服务器一样使用集群。但是，集群不是自动扩展的：你必须编写集群配置到服务器配置文件中（最好，给所有集群的服务器写上完整配置）。

不支持用分布式表查询别的分布式表（除非该表只有一个分片）。或者说，要用分布表查查询“最终”的数据表。

分布式引擎需要将集群信息写入配置文件。配置文件中的集群信息会即时更新，无需重启服务器。如果你每次是要向不确定的

一组分片和副本发送查询，则不适合创建分布式表 - 而应该使用“远程”表函数。请参阅“表函数”部分。

向集群写数据的方法有两种：

一，自己指定要将哪些数据写入哪些服务器，并直接在每个分片上执行写入。换句话说，在分布式表上“查询”，在数据表上 INSERT。

这是最灵活的解决方案 - 你可以使用任何分片方案，对于复杂业务特性的需求，这可能是非常重要的。

这也是最佳解决方案，因为数据可以完全独立地写入不同的分片。

二，在分布式表上执行 INSERT。在这种情况下，分布式表会跨服务器分发插入数据。

为了写入分布式表，必须要配置分片键（最后一个参数）。当然，如果只有一个分片，则写操作在没有分片键的情况下也能工作，因为这种情况下分片键没有意义。

每个分片都可以在配置文件中定义权重。默认情况下，权重等于1。数据依据分片权重按比例分发到分片上。例如，如果有两个分片，第一个分片的权重是9，而第二个分片的权重是10，则发送 9 / 19 的行到第一个分片，10 / 19 的行到第二个分片。

分片可在配置文件中定义 'internal_replication' 参数。

此参数设置为“true”时，写操作只选一个正常的副本写入数据。如果分布式表的子表是复制表(*ReplicaMergeTree)，请使用此方案。换句话说，这其实是把数据的复制工作交给实际需要写入数据的表本身而不是分布式表。

若此参数设置为“false”（默认值），写操作会将数据写入所有副本。实质上，这意味着要分布式表本身来复制数据。这种方式不如使用复制表的好，因为不会检查副本的一致性，并且随着时间的推移，副本数据可能会有些不一样。

选择将一行数据发送到哪个分片的方法是，首先计算分片表达式，然后将这个计算结果除以所有分片的权重总和得到余数。该行会发送到那个包含该余数的从 'prev_weight' 到 'prev_weights + weight' 的半闭半开区间对应的分片上，其中 'prev_weights' 是该分片前面的所有分片的权重和，'weight' 是该分片的权重。例如，如果有两个分片，第一个分片权重为9，而第二个分片权重为10，则余数在 [0,9) 中的行发给第一个分片，余数在 [9,19) 中的行发给第二个分片。

分片表达式可以是由常量和表列组成的任何返回整数表达式。例如，您可以使用表达式 'rand()' 来随机分配数据，或者使用 'UserID' 来按用户 ID 的余数分布（相同用户的数据将分配到单个分片上，这可降低带有用户信息的 IN 和 JOIN 的语句运行的复杂度）。如果该列数据分布不够均匀，可以将其包装在散列函数中：intHash64(UserID)。

这种简单的用余数来选择分片的方案是有局限的，并不总适用。它适用于中型和大型数据（数十台服务器）的场景，但不适用于巨量数据（数百台或更多服务器）的场景。后一种情况下，应根据业务特性需求考虑的分片方案，而不是直接用分布式表的多分片。

SELECT 查询会被发送到所有分片，并且无论数据在分片中如何分布（即使数据完全随机分布）都可正常工作。添加新分片时，不必将旧数据传输到该分片。你可以给新分片分配大权重然后写新数据 - 数据可能会稍分布不均，但查询会正确高效地运行。

下面的情况，你需要关注分片方案：

- 使用需要特定键连接数据（IN 或 JOIN）的查询。如果数据是用该键进行分片，则应使用本地 IN 或 JOIN 而不是 GLOBAL IN 或 GLOBAL JOIN，这样效率更高。
- 使用大量服务器（上百或更多），但有大量小查询（个别客户的查询 - 网站，广告商或合作伙伴）。为了使小查询不影响整个集群，让单个客户的数据处于单个分片上是有意义的。或者，正如我们在 Yandex.Metrica 中所做的那样，你可以配置两级分片：将整个集群划分为“层”，一个层可以包含多个分片。单个客户的数据位于单个层上，根据需要 will 将分片添加到层中，层中的数据随机分布。然后给每层创建分布式表，再创建一个全局的分布式表用于全局的查询。

数据是异步写入的。对于分布式表的 INSERT，数据块只写本地文件系统。之后会尽快地在后台发送到远程服务器。你可以通过查看表目录中的文件列表（等待发送的数据）来检查数据是否成功发送：/var/lib/clickhouse/data/database/table/。

如果在 INSERT 到分布式表时服务器节点丢失或重启（如，设备故障），则插入的数据可能会丢失。如果在表目录中检测到损坏的数据分片，则会将其转移到“broken”子目录，并不再使用。

启用 max_parallel_replicas 选项后，会在分表的所有副本上并行查询处理。更多信息，请参阅“设置，max_parallel_replicas”部分。

External Data for Query Processing

ClickHouse 允许向服务器发送处理查询所需的数据以及 SELECT 查询。这些数据放在一个临时表中（请参阅 "临时表" 一节），可以在查询中使用（例如，在 IN 操作符中）。

例如，如果您有一个包含重要用户标识符的文本文件，则可以将其与使用此列表过滤的查询一起上传到服务器。

如果需要使用大量外部数据运行多个查询，请不要使用该特性。最好提前把数据上传到数据库。

可以使用命令行客户端（在非交互模式下）或使用 HTTP 接口上传外部数据。

在命令行客户端中，您可以指定格式的参数部分

```
--external --file=... [--name=...] [--format=...] [--types=...|--structure=...]
```

对于传输的表的数量，可能有多个这样的部分。

--external - 标记子句的开始。

--file - 带有表存储的文件的路径，或者，它指的是STDIN。

只能从 stdin 中检索单个表。

以下的参数是可选的：**--name** - 表的名称，如果省略，则采用 `_data`。

--format - 文件中的数据格式。如果省略，则使用 TabSeparated。

以下的参数必选一个：**--types** - 逗号分隔列类型的列表。例如：`UInt64,String`。列将被命名为 `_1`，`_2`，...

--structure - 表结构的格式 `UserID UInt64`，`URL String`。定义列的名字以及类型。

在 "file" 中指定的文件将由 "format" 中指定的格式解析，使用在 "types" 或 "structure" 中指定的数据类型。该表将被上传到服务器，并在作为名称为 "name" 临时表。

示例：

```
echo -ne "1\n2\n3\n" | clickhouse-client --query="SELECT count() FROM test.visits WHERE TrafficSourceID IN _data" --external --file=- --types=Int8
849897
cat /etc/passwd | sed 's:/\t/g' | clickhouse-client --query="SELECT shell, count() AS c FROM passwd GROUP BY shell ORDER BY c DESC" --external --file=- --name=passwd --structure='login String, unused String, uid UInt16, gid UInt16, comment String, home String, shell String'
/bin/sh 20
/bin/false 5
/bin/bash 4
/usr/sbin/nologin 1
/bin/sync 1
```

当使用HTTP接口时，外部数据以 multipart/form-data 格式传递。每个表作为一个单独的文件传输。表名取自文件名。

"query_string" 传递参数 "name_format"、"name_types"和"name_structure"，其中 "name" 是这些参数对应的表的名称。参数的含义与使用命令行客户端时的含义相同。

示例：

```
cat /etc/passwd | sed 's:/\t/g' > passwd.tsv

curl -F 'passwd=@passwd.tsv;' 'http://localhost:8123/?query=SELECT+shell,+count()+AS+c+FROM+passwd+GROUP+BY+shell+ORDER+BY+c+DESC&passwd_structure=login+String,+unused+String,+uid+UInt16,+gid+UInt16,+comment+String,+home+String,+shell+String'
/bin/sh 20
/bin/false 5
/bin/bash 4
/usr/sbin/nologin 1
```

对于分布式查询，将临时表发送到所有远程服务器。

Dictionary

Dictionary 引擎将字典数据展示为一个ClickHouse的表。

例如，考虑使用一个具有以下配置的 **products** 字典：

```
<ictionaries>
<dictionary>
  <name>products</name>
  <source>
    <odbc>
      <table>products</table>
      <connection_string>DSN=some-db-server</connection_string>
    </odbc>
  </source>
  <lifetime>
    <min>300</min>
    <max>360</max>
  </lifetime>
  <layout>
    <flat/>
  </layout>
  <structure>
    <id>
      <name>product_id</name>
    </id>
    <attribute>
      <name>title</name>
      <type>String</type>
      <null_value></null_value>
    </attribute>
  </structure>
</dictionary>
</ictionaries>
```

查询字典中的数据：

```
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

products	id	uint64	title	String	23003370	173032	ODBC: products

你可以使用 **dictGet*** 函数来获取这种格式的字典数据。

当你需要获取原始数据，或者是想要使用 **JOIN** 操作的时候，这种视图并没有什么帮助。对于这些情况，你可以使用 **Dictionary** 引擎，它可以将字典数据展示在表中。

语法：

```
CREATE TABLE %table_name% (%fields%) engine = Dictionary(%dictionary_name%)\`
```

示例：

```
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.
```

看一看表中的内容。

```
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

Merge 引擎 (不要跟 **MergeTree** 引擎混淆) 本身不存储数据，但可用于同时从任意多个其他的表中读取数据。读是自动并行的，不支持写入。读取时，那些被真正读取到数据的表的索引（如果有的话）会被使用。

Merge 引擎的参数：一个数据库名和一个用于匹配表名的正则表达式。

示例：

```
Merge(hits, '^WatchLog')
```

数据会从 **hits** 数据库中表名匹配正则 **'^WatchLog'** 的表中读取。

除了数据库名，你也可以用一个返回字符串的常量表达式。例如， `currentDatabase()` 。

正则表达式 — `re2` (支持 PCRE 一个子集的功能)，大小写敏感。

了解关于正则表达式中转义字符的说明可参看 "match" 一节。

当选择需要读的表时，`Merge` 表本身会被排除，即使它匹配上了该正则。这样设计为了避免循环。

当然，是能够创建两个相互无限递归读取对方数据的 `Merge` 表的，但这并没有什么意义。

`Merge` 引擎的一个典型应用是可以像使用一张表一样使用大量的 `TinyLog` 表。

示例 2：

我们假定你有一个旧表 (`WatchLog_old`)，你想改变数据分区了，但又不想把旧数据转移到新表 (`WatchLog_new`) 里，并且你需要同时能看到这两个表的数据。

```
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

虚拟列

虚拟列是一种由表引擎提供而不是在表定义中的列。换种说法就是，这些列并没有在 `CREATE TABLE` 中指定，但可以在 `SELECT` 中使用。

下面列出虚拟列跟普通列的不同点：

- 虚拟列不在表结构定义里指定。
- 不能用 `INSERT` 向虚拟列写数据。
- 使用不指定列名的 `INSERT` 语句时，虚拟列要会被忽略掉。
- 使用星号通配符 (`SELECT *`) 时虚拟列不会包含在里面。
- 虚拟列不会出现在 `SHOW CREATE TABLE` 和 `DESC TABLE` 的查询结果里。

`Merge` 类型的表包括一个 `String` 类型的 `_table` 虚拟列。（如果该表本来已有了一个 `_table` 的列，那这个虚拟列会命名为 `_table1`；如果 `_table1` 也本就存在了，那这个虚拟列会被命名为 `_table2`，依此类推）该列包含被读数据的表名。

如果 `WHERE/PREWHERE` 子句包含了带 `_table` 的条件，并且没有依赖其他的列（如作为表达式谓词链接的一个子项或作为整个的表达式），这些条件的作用会像索引一样。这些条件会在那些可能被读数据的表的表名上执行，并且读操作只会在那些满足了该条件的表上去执行。

File(InputFormat)

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

Usage examples:

Usage examples:

- Data export from ClickHouse to file.
- Convert data from one format to another.
- Updating data in ClickHouse via editing a file on a disk.

Usage in ClickHouse Server

File(Format)

Format should be supported for either `INSERT` and `SELECT`. For the full list of supported formats see [Formats](#).

ClickHouse does not allow to specify filesystem path for `File`. It will use folder defined by `path` setting in server configuration.

When creating table using `File(Format)` it creates empty subdirectory in that folder. When data is written to that table, it's put into `data.Format` file in that subdirectory.

You may manually create this subfolder and file in server filesystem and then **ATTACH** it to table information with matching name, so you can query data from that file.

Warning

Be careful with this functionality, because ClickHouse does not keep track of external changes to such files. The result of simultaneous writes via ClickHouse and outside of ClickHouse is undefined.

Example:

1. Set up the `file_engine_table` table:

```
CREATE TABLE file_engine_table (name String, value UInt32) ENGINE=File(TabSeparated)
```

By default ClickHouse will create folder `/var/lib/clickhouse/data/default/file_engine_table`.

2. Manually create `/var/lib/clickhouse/data/default/file_engine_table/data.TabSeparated` containing:

```
$ cat data.TabSeparated
one 1
two 2
```

3. Query the data:

```
SELECT * FROM file_engine_table
```

name	value
one	1
two	2

Usage in Clickhouse-local

In **clickhouse-local** File engine accepts file path in addition to `Format`. Default input/output streams can be specified using numeric or human-readable names like `0` or `stdin`, `1` or `stdout`.

Example:

```
$ echo -e "1 2\n3 4" | clickhouse-local -- "CREATE TABLE table (a Int64, b Int64) ENGINE = File(CSV, stdin); SELECT a, b
```

```
$ echo "CREATE TABLE table (a Int64, b Int64) ENGINE = File(CSV, 'data'); 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

当写入 Null 类型的表时，将忽略数据。从 Null 类型的表中读取时，返回空。

但是，可以在 Null 类型的表上创建物化视图。写入表的数据将转发到视图中。

Set

始终存在于 RAM 中的数据块。它适用于 IN 运算符的右侧（请参见 "IN 运算符" 部分）。

可以使用 INSERT 向表中插入数据。新元素将添加到数据集中，而重复项将被忽略。但是不能对此类型表执行 SELECT 语句。检索数据的唯一方法是在 IN 运算符的右半部分使用它。

数据始终存在于 RAM 中。对于 INSERT，插入数据块也会写入磁盘上的表目录。启动服务器时，此数据将加载到 RAM。也就是说，重新启动后，数据仍然存在。

对于强制服务器重启，磁盘上的数据块可能会丢失或损坏。在数据块损坏的情况下，可能需要手动删除包含损坏数据的文件。

Join

加载好的 JOIN 表数据会常驻内存中。

```
Join(ANY|ALL, LEFT|INNER, k1[, k2, ...])
```

引擎参数：ANY|ALL - 连接修饰；LEFT|INNER - 连接类型。更多信息可参考 [JOIN 子句](#)。

这些参数设置不用带引号，但必须与要 JOIN 表匹配。k1, k2, 是 USING 子句中要用于连接的关键列。

此引擎表不能用于 GLOBAL JOIN。

类似于 Set 引擎，可以使用 INSERT 向表中添加数据。设置为 ANY 时，重复键的数据会被忽略（仅一条用于连接）。设置为 ALL 时，重复键的数据都会用于连接。不能直接对 JOIN 表进行 SELECT。检索其数据的唯一方法是将其作为 JOIN 语句右边的表。

跟 Set 引擎类似，Join 引擎把数据存储于磁盘中。

URL(URL, Format)

Manages data on a remote HTTP/HTTPS server. This engine is similar to the [File](#) engine.

Using the engine in the ClickHouse server

The format must be one that ClickHouse can use in

SELECT queries and, if necessary, in INSERTs. For the full list of supported formats, see [Formats](#).

The URL must conform to the structure of a Uniform Resource Locator. The specified URL must point to a server

server

that uses HTTP or HTTPS. This does not require any additional headers for getting a response from the server.

INSERT and SELECT queries are transformed to POST and GET requests, respectively. For processing POST requests, the remote server must support **Chunked transfer encoding**.

Example:

1. Create a url_engine_table table on the server :

```
CREATE TABLE url_engine_table (word String, value UInt64)
ENGINE=URL('http://127.0.0.1:12345/', CSV)
```

2. Create a basic HTTP server using the standard Python 3 tools and start it:

```
from http.server import BaseHTTPRequestHandler, HTTPServer

class CSVHTTPServer(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/csv')
        self.end_headers()

        self.wfile.write(bytes('Hello,1\nWorld,2\n', "utf-8"))

if __name__ == "__main__":
    server_address = ('127.0.0.1', 12345)
    HTTPServer(server_address, CSVHTTPServer).serve_forever()
```

```
python3 server.py
```

3. Request data:

```
SELECT * FROM url_engine_table
```

word	value
Hello	1
World	2

Details of Implementation

- Reads and writes can be parallel
- Not supported:
 - ALTER and SELECT...SAMPLE operations.
 - Indexes.
 - Replication.

View

用于构建视图（有关更多信息，请参阅 **CREATE VIEW 查询**）。它不存储数据，仅存储指定的 **SELECT** 查询。从表中读取时，它会运行此查询（并从查询中删除所有不必要的列）。

物化视图

物化视图的使用（更多信息请参阅 [CREATE TABLE](#)）。它需要使用一个不同的引擎来存储数据，这个引擎要在创建物化视图时指定。当从表中读取时，它就会使用该引擎。

Memory

Memory 引擎以未压缩的形式将数据存储在 RAM 中。数据完全以读取时获得的形式存储。换句话说，从这张表中读取是很轻松的。并发数据访问是同步的。锁范围小：读写操作不会相互阻塞。不支持索引。阅读是并行化的。在简单查询上达到最大生产率（超过10 GB /秒），因为没有磁盘读取，不需要解压缩或反序列化数据。（值得注意的是，在许多情况下，与 MergeTree 引擎的性能几乎一样高）。重新启动服务器时，表中的数据消失，表将变为空。通常，使用此表引擎是不合理的。但是，它可用于测试，以及在相对较少的行（最多约100,000,000）上需要最高性能的查询。

Memory 引擎是由系统用于临时表进行外部数据的查询（请参阅 "外部数据用于请求处理" 部分），以及用于实现 GLOBAL IN（请参见 "IN 运算符" 部分）。

Buffer

缓冲数据写入 RAM 中，周期性地将数据刷新到另一个表。在读取操作时，同时从缓冲区和另一个表读取数据。

```
Buffer(database, table, num_layers, min_time, max_time, min_rows, max_rows, min_bytes, max_bytes)
```

引擎的参数：database，table - 要刷新数据的表。可以使用返回字符串的常量表达式而不是数据库名称。num_layers - 并行层数。在物理上，该表将表示为 num_layers 个独立缓冲区。建议值为16。

min_time，max_time，min_rows，max_rows，min_bytes，max_bytes - 从缓冲区刷新数据的条件。

如果满足所有 "min" 条件或至少一个 "max" 条件，则从缓冲区刷新数据并将其写入目标表。min_time，max_time — 从第一次写入缓冲区时起以秒为单位的时间条件。min_rows，max_rows - 缓冲区中行数的条件。min_bytes，max_bytes - 缓冲区中字节数的条件。

写入时，数据从 num_layers 个缓冲区中随机插入。或者，如果插入数据的大小足够大（大于 max_rows 或 max_bytes），则会绕过缓冲区将其写入目标表。

每个 "num_layers" 缓冲区刷新数据的条件是分别计算。例如，如果 num_layers = 16 且 max_bytes = 100000000，则最大RAM消耗将为1.6 GB。

示例：

```
CREATE TABLE merge.hits_buffer AS merge.hits ENGINE = Buffer(merge, hits, 16, 10, 100, 10000, 1000000, 10000000, 100000000)
```

创建一个 "merge.hits_buffer" 表，其结构与 "merge.hits" 相同，并使用 Buffer 引擎。写入此表时，数据缓冲在 RAM 中，然后写入 "merge.hits" 表。创建了16个缓冲区。如果已经过了100秒，或者已写入100万行，或者已写入100 MB数据，则刷新每个缓冲区的数据；或者如果同时已经过了10秒并且已经写入了10,000行和10 MB的数据。例如，如果只写了一行，那么在100秒之后，都会被刷新。但是如果写了很多行，数据将会更快地刷新。

当服务器停止时，使用 DROP TABLE 或 DETACH TABLE，缓冲区数据也会刷新到目标表。

可以为数据库和表名在单个引号中设置空字符串。这表示没有目的地表。在这种情况下，当达到数据刷新条件时，缓冲器被简单地清除。这可能对于保持数据窗口在内存中是有用的。

从 Buffer 表读取时，将从缓冲区和目标表（如果有）处理数据。

请注意，Buffer 表不支持索引。换句话说，缓冲区中的数据被完全扫描，对于大缓冲区来说可能很慢。（对于目标表中的数据，将使用它支持的索引。）

如果 Buffer 表中的列集与目标表中的列集不匹配，则会插入两个表中存在的列的子集。

如果类型与 **Buffer** 表和目标表中的某列不匹配，则会在服务器日志中输入错误消息并清除缓冲区。

如果在刷新缓冲区时目标表不存在，则会发生同样的情况。

如果需要为目标表和 **Buffer** 表运行 **ALTER**，我们建议先删除 **Buffer** 表，为目标表运行 **ALTER**，然后再次创建 **Buffer** 表。

如果服务器异常重启，缓冲区中的数据将丢失。

PREWHERE，**FINAL** 和 **SAMPLE** 对缓冲表不起作用。这些条件将传递到目标表，但不用于处理缓冲区中的数据。因此，我们建议只使用 **Buffer** 表进行写入，同时从目标表进行读取。

将数据添加到缓冲区时，其中一个缓冲区被锁定。如果同时从表执行读操作，则会导致延迟。

插入到 **Buffer** 表中的数据可能以不同的顺序和不同的块写入目标表中。因此，**Buffer** 表很难用于正确写入 **CollapsingMergeTree**。为避免出现问题，您可以将 "num_layers" 设置为 1。

如果目标表是复制表，则在写入 **Buffer** 表时会丢失复制表的某些预期特征。数据部分的行次序和大小的随机变化导致数据不能去重，这意味着无法对复制表进行可靠的 "exactly once" 写入。

由于这些缺点，我们只建议在极少数情况下使用 **Buffer** 表。

当在单位时间内从大量服务器接收到太多 **INSERTs** 并且在插入之前无法缓冲数据时使用 **Buffer** 表，这意味着这些 **INSERTs** 不能足够快地执行。

请注意，一次插入一行数据是没有意义的，即使对于 **Buffer** 表也是如此。这将只产生每秒几千行的速度，而插入更大的数据块每秒可以产生超过一百万行（参见 "性能" 部分）。

JDBC

Allows ClickHouse to connect to external databases via **JDBC**.

To implement the JDBC connection, ClickHouse uses the separate program **clickhouse-jdbc-bridge** that should run as a daemon.

This engine supports the **Nullable** data type.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name
ENGINE = JDBC(dbms_uri, external_database, external_table)
```

Engine Parameters

- dbms_uri** — URI of an external DBMS.

Format: **jdbc:<driver_name>://<host_name>:<port>/?user=<username>&password=<password>**.

Example for MySQL: **jdbc:mysql://localhost:3306/?user=root&password=root**.

- external_database** — Database in an external DBMS.
- external_table** — Name of the table in **external_database**.

Usage Example

Creating a table in MySQL server by connecting directly with its console client:

```
mysql> CREATE TABLE `test`.`test` (
->   `int_id` INT NOT NULL AUTO_INCREMENT,
->   `int_nullable` INT NULL DEFAULT NULL,
->   `float` FLOAT NOT NULL,
->   `float_nullable` FLOAT NULL DEFAULT NULL,
```

```
-> float_nullable FLOAT NULL DEFAULT NULL,
-> PRIMARY KEY (`int_id`));
Query OK, 0 rows affected (0,09 sec)

mysql> insert into test (`int_id`, `float`) VALUES (1,2);
Query OK, 1 row affected (0,00 sec)
```

```
mysql> select * from test;
+-----+-----+-----+-----+
| int_id | int_nullable | float | float_nullable |
+-----+-----+-----+-----+
| 1 | NULL | 2 | NULL |
+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

Creating a table in ClickHouse server and selecting data from it:

```
CREATE TABLE jdbc_table ENGINE JDBC('jdbc:mysql://localhost:3306/?user=root&password=root', 'test', 'test')
```

Ok.

```
DESCRIBE TABLE jdbc_table
```

name	type	default_type	default_expression
int_id	Int32		
int_nullable	Nullable(Int32)		
float	Float32		
float_nullable	Nullable(Float32)		

10 rows in set. Elapsed: 0.031 sec.

```
SELECT *
FROM jdbc_table
```

int_id	int_nullable	float	float_nullable
1	NULL	2	NULL

1 rows in set. Elapsed: 0.055 sec.

See Also

- [JDBC table function](#).

ODBC

Allows ClickHouse to connect to external databases via [ODBC](#).

To implement ODBC connection safely, ClickHouse uses the separate program `clickhouse-odbc-bridge`. If the ODBC driver is loaded directly from the `clickhouse-server` program, the problems in the driver can crash the ClickHouse server. ClickHouse starts the `clickhouse-odbc-bridge` program automatically when it is required. The ODBC bridge program is installed by the same package as the `clickhouse-server`.

This engine supports the [Nullable](#) data type.

Creating a Table

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
```

```
name1 [type1],
name2 [type2],
...
)
ENGINE = ODBC(connection_settings, external_database, external_table)
```

See the detailed description of the **CREATE TABLE** query.

The table structure can be not the same as the source table structure:

- Names of columns should be the same as in the source table, but you can use just some of these columns in any order.
- Types of columns may differ from the types in the source table. ClickHouse tries to **cast** values into the ClickHouse data types.

Engine Parameters

- `connection_settings` — Name of the section with connection settings in the `odbc.ini` file.
- `external_database` — Name of a database in an external DBMS.
- `external_table` — Name of a table in the `external_database`.

Usage Example

Getting data from the local MySQL installation via ODBC

This example is for linux Ubuntu 18.04 and MySQL server 5.7.

Ensure that there are unixODBC and MySQL Connector are installed.

By default (if installed from packages) ClickHouse starts on behalf of the user `clickhouse`. Thus, you need to create and configure this user at MySQL server.

```
sudo mysql
mysql> CREATE USER 'clickhouse'@'localhost' IDENTIFIED BY 'clickhouse';
mysql> GRANT ALL PRIVILEGES ON *.* TO 'clickhouse'@'clickhouse' WITH GRANT OPTION;
```

Then configure the connection in `/etc/odbc.ini`.

```
$ cat /etc/odbc.ini
[mysqlconn]
DRIVER = /usr/local/lib/libmyodbc5w.so
SERVER = 127.0.0.1
PORT = 3306
DATABASE = test
USERNAME = clickhouse
PASSWORD = clickhouse
```

You can check the connection by the `isql` utility from the unixODBC installation.

```
isql -v mysqlconn  
+-----+  
| Connected! |  
|           |  
... 
```

Table in MySQL:

```
mysql> CREATE TABLE `test`.`test` (
->  `int_id` INT NOT NULL AUTO_INCREMENT,
->  `int_nullable` INT NULL DEFAULT NULL,
->  `float` FLOAT NOT NULL,
->  `float_nullable` FLOAT NULL DEFAULT NULL,
->  PRIMARY KEY (`int_id`));
Query OK, 0 rows affected (0,09 sec)
```

```
mysql> insert into test (`int_id`, `float`) VALUES (1,2);
Query OK, 1 row affected (0,00 sec)
```

```
mysql> select * from test;
+-----+-----+-----+-----+
| int_id | int_nullable | float | float_nullable |
+-----+-----+-----+-----+
| 1 | NULL | 2 | NULL |
+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

Table in ClickHouse, getting data from the MySQL table:

```
CREATE TABLE odbc_t
(
  `int_id` Int32,
  `float_nullable` Nullable(Float32)
)
ENGINE = ODBC('DSN=mysqlconn', 'test', 'test')
```

```
SELECT * FROM odbc_t
```

```
┌ int_id ─┴─ float_nullable ─┐
└── 1 ─┴─ NULL ─┘
```

See Also

- [ODBC external dictionaries](#)
- [ODBC table function](#).

SQL Reference

- [SELECT](#)
- [INSERT INTO](#)
- [CREATE](#)
- [ALTER](#)
- [Other types of queries](#)

SELECT 查询语法

SELECT 语句用于执行数据的检索。

```
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]
```

所有的子句都是可选的，除了SELECT之后的表达式列表(expr_list)。

下面将按照查询运行的顺序逐一对各个子句进行说明。

如果查询中不包含DISTINCT，GROUP BY，ORDER BY子句以及IN和JOIN子查询，那么它将仅使用O(1)数量的内存来完全流式的处理查询

否则，这个查询将消耗大量的内存，除非你指定了这些系统配置：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。它们规定了可以使用外部排序（将临时表存储到磁盘中）以及外部聚合，目前系统不存在关于Join的配置，更多关于它们的信息，可以参见“配置”部分。

FROM 子句

如果查询中不包含FROM子句，那么将读取system.one。

system.one中仅包含一行数据（此表实现了与其他数据库管理系统中的DUAL相同的功能）。

FROM子句规定了将从哪个表、或子查询、或表函数中读取数据；同时ARRAY JOIN子句和JOIN子句也可以出现在这里（见后文）。

可以使用包含在括号里的子查询来替代表。

在这种情况下，子查询的处理将会构建在外部的查询内部。

不同于SQL标准，子查询后无需指定别名。为了兼容，你可以在子查询后添加‘AS 别名’，但是指定的名字不能被使用在任何地方。

也可以使用表函数来代替表，有关信息，参见“表函数”。

执行查询时，在查询中列出的所有列都将从对应的表中提取数据；如果你使用的是子查询的方式，则任何在外部查询中没有使用的列，子查询将从查询中忽略它们；

如果你的查询没有列出任何的列（例如，SELECT count() FROM t），则将额外的从表中提取一些列（最好的情况下是最小的列），以便计算行数。

最后的FINAL修饰符仅能够被使用在SELECT from CollapsingMergeTree场景中。当你为FROM指定了FINAL修饰符时，你的查询结果将会在查询过程中被聚合。需要注意的是，在这种情况下，查询将在单个流中读取所有相关的主键列，同时对需要的数据进行合并。这意味着，当使用FINAL修饰符时，查询将会处理的更慢。在大多数情况下，你应该避免使用FINAL修饰符。更多信息，请参阅“CollapsingMergeTree引擎”部分。

SAMPLE 子句

通过SAMPLE子句用户可以进行近似查询处理，近似查询处理仅能工作在MergeTree*类型的表中，并且在创建表时需要您指定采样表达式（参见“MergeTree 引擎”部分）。

SAMPLE子句可以使用SAMPLE k来表示，其中k可以是0到1的小数值，或者是一个足够大的正整数。

当k为0到1的小数时，查询将使用'k'作为百分比选取数据。例如，SAMPLE 0.1查询只会检索数据总量的10%。

当k为一个足够大的正整数时，查询将使用'k'作为最大样本数。例如，SAMPLE 10000000查询只会检索最多10,000,000行数据。

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

```

在这个例子中，查询将检索数据总量的0.1（10%）的数据。值得注意的是，查询不会自动校正聚合函数最终的结果，所以为了得到更加精确的结果，需要将 `count()` 的结果手动乘以10。

当使用像 `SAMPLE 10000000` 这样的方式进行近似查询时，由于没有了任何关于将会处理了哪些数据或聚合函数应该被乘以几的信息，所以这种方式不适合在这种场景下使用。

使用相同的采样率得到的结果总是一致的：如果我们能够看到所有可能存在于表中的数据，那么相同的采样率总是能够得到相同的结果（在建表时使用相同的采样表达式），换句话说，系统在不同的时间，不同的服务器，不同表上总以相同的方式对数据进行采样。

例如，我们可以使用采样的方式获取到与不进行采样相同的用户ID的列表。这将表明，你可以在 `IN` 子查询中使用采样，或者使用采样的结果与其他查询进行关联。

ARRAY JOIN 子句

ARRAY JOIN子句可以帮助查询进行与数组和 `nested` 数据类型的连接。它有点类似 `arrayJoin` 函数，但它的功能更广泛。

ARRAY JOIN 本质上等同于 `INNERT JOIN` 数组。例如：

```

:) 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.

你还可以为ARRAY JOIN子句指定一个别名，这时你可以通过这个别名来访问数组中的数据，但是数据本身仍然可以通过原来的名称进行访问。例如：

```
:) 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.

当多个具有相同大小的数组使用逗号分割出现在ARRAY JOIN子句中时，ARRAY JOIN会将它们同时执行（直接合并，而不是它们的笛卡尔积）。例如：

```
:) 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.

另外ARRAY JOIN也可以工作在nested数据结构上。例如：

```
) 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.

当你在ARRAY JOIN指定nested数据类型的名称时，其作用与与包含所有数组元素的ARRAY JOIN相同，例如：

```
:) 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.

这种方式也是可以运行的：

```
:) 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.

为了方便使用原来的nested类型的数组，你可以为nested类型定义一个别名。例如：

```
:) 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.
```

使用arrayEnumerate函数的示例：

```
:) 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.
```

在一个查询中只能出现一个ARRAY JOIN子句。

如果在WHERE/PREWHERE子句中使用了ARRAY JOIN子句的结果，它将优先于WHERE/PREWHERE子句执行，否则它将在WHERE/PRWHERE子句之后执行，以便减少计算。

JOIN 子句

JOIN子句用于连接数据，作用与SQL JOIN的定义相同。

注意

与 ARRAY JOIN 没有关系。

```
SELECT <expr_list>
FROM <left_subquery>
[GLOBAL] [ANY|ALL] INNER|LEFT|RIGHT|FULL|CROSS [OUTER] JOIN <right_subquery>
(ON <expr_list>)|(USING <column_list>) ...
```

可以使用具体的表名来代替<left_subquery>与<right_subquery>。但这与使用SELECT * FROM table子查询的方式相同。除非你的表是[Join](../operations/table_engines/join.md)

支持的JOIN类型

- INNER JOIN
- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN
- CROSS JOIN

你可以跳过默认的OUTER关键字。

ANY 与 ALL

在使用ALL修饰符对JOIN进行修饰时，如果右表中存在多个与左表关联的数据，那么系统则将右表中所有可以与左表关联的数据全部返回在结果中。这与SQL标准的JOIN行为相同。

在使用ANY修饰符对JOIN进行修饰时，如果右表中存在多个与左表关联的数据，那么系统仅返回第一个与左表匹配的结果。如果左表与右表一一对应，不存在多余的行时，ANY与ALL的结果相同。

你可以在会话中通过设置 `join_default_strictness` 来指定默认的JOIN修饰符。

GLOBAL distribution

当使用普通的JOIN时，查询将被发送给远程的服务器。并在这些远程服务器上生成右表并与它们关联。换句话说，右表来自于各个服务器本身。

当使用 `GLOBAL ... JOIN`，首先会在请求服务器上计算右表并以临时表的方式将其发送到所有服务器。这时每台服务器将直接使用它进行计算。

使用 `GLOBAL` 时需要小心。更多信息，参阅 [Distributed subqueries](#) 部分。

使用建议

从子查询中删除所有JOIN不需要的列。

当执行JOIN查询时，因为与其他阶段相比没有进行执行顺序的优化：JOIN优先于WHERE与聚合执行。因此，为了显示的指定执行顺序，我们推荐你使用子查询的方式执行JOIN。

示例：

```
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

子查询不允许您设置别名或在其他地方引用它们。

`USING` 中指定的列必须在两个子查询中具有相同的名称，而其他列必须具有不同的名称。您可以通过使用别名的方式来更改子查询中的列名（示例中就分别使用了'hits'与'visits'别名）。

USING 子句用于指定要进行链接的一个或多个列，系统会将这些列在两张表中相等的值连接起来。如果列是一个列表，不需要使用括号包裹。同时**JOIN**不支持其他更复杂的Join方式。

右表（子查询的结果）将会保存在内存中。如果没有足够的内存，则无法运行**JOIN**。

只能在查询中指定一个**JOIN**。若要运行多个**JOIN**，你可以将它们放入子查询中。

每次运行相同的**JOIN** 查询，总是会再次计算 - 没有缓存结果。为了避免这种情况，可以使用'Join'引擎，它是一个预处理的Join数据结构，总是保存在内存中。更多信息，参见“Join引擎”部分。

在一些场景下，使用**IN**代替**JOIN**将会得到更高的效率。在各种类型的**JOIN**中，最高效的是**ANY LEFT JOIN**，然后是**ANY INNER JOIN**，效率最差的是**ALL LEFT JOIN**以及**ALL INNER JOIN**。

如果你需要使用**JOIN**来关联一些纬度表（包含纬度属性的一些相对较小的表，例如广告活动的名称），那么**JOIN**可能不是好的选择，因为语法负责，并且每次查询都将重新访问这些表。对于这种情况，您应该使用“外部字典”的功能来替换**JOIN**。更多信息，参见 [外部字典](#) 部分。

Null的处理

JOIN的行为受 **join_use_nulls** 的影响。当**join_use_nulls=1**时，**JOIN**的工作与SQL标准相同。

如果**JOIN**的key是 **Nullable** 类型的字段，则其中至少一个存在 **NULL** 值的key不会被关联。

WHERE 子句

如果存在**WHERE**子句, 则在该子句中必须包含一个**UInt8**类型的表达式。这个表达是通常是一个带有比较和逻辑的表达式。这个表达式将会在所有数据转换前用来过滤数据。

如果在支持索引的数据库表引擎中，这个表达式将被评估是否使用索引。

PREWHERE 子句

这个子句与**WHERE**子句的意思相同。主要的不同之处在于表数据的读取。

当使用**PREWHERE**时，首先只读取**PREWHERE**表达式中需要的列。然后在根据**PREWHERE**执行的结果读取其他需要的列。

如果在过滤条件中有少量不适合索引过滤的列，但是它们又可以提供很强的过滤能力。这时使用**PREWHERE**是有意义的，因为它将帮助减少数据的读取。

例如，在一个需要提取大量列的查询中为少部分列编写**PREWHERE**是很有作用的。

PREWHERE 仅支持***MergeTree**系列引擎。

在一个查询中可以同时指定**PREWHERE**和**WHERE**，在这种情况下，**PREWHERE**优先于**WHERE**执行。

值得注意的是，**PREWHERE**不适合用于已经存在于索引中的列，因为当列已经存在于索引中的情况下，只有满足索引的数据块才会被读取。

如果将'**optimize_move_to_prewhere**'设置为1，并且在查询中不包含**PREWHERE**，则系统将自动的把适合**PREWHERE**表达式的部分从**WHERE**中抽离到**PREWHERE**中。

GROUP BY 子句

这是列式数据库管理系统中最重要的一部分。

如果存在**GROUP BY**子句，则在该子句中必须包含一个表达式列表。其中每个表达式将会被称之为“key”。

SELECT，**HAVING**，**ORDER BY**子句中的表达式列表必须来自于这些“key”或聚合函数。简而言之，被选择的列中不能包含非聚合函数或key之外的其他列。

如果查询表达式列表中仅包含聚合函数，则可以省略**GROUP BY**子句，这时会假定将所有数据聚合成一组空“key”。

Example:

```
SELECT
```

```
SELECT
    count(),
    median(FetchTiming > 60 ? 60 : FetchTiming),
    count() - sum(Refresh)
FROM hits
```

与SQL标准不同的是，如果表中不存在任何数据（可能表本身中就不存在任何数据，或者由于被WHERE条件过滤掉了），将返回一个空结果，而不是一个包含聚合函数初始值的结果。

与MySQL不同的是（实际上这是符合SQL标准的），你不能够获得一个不在key中的非聚合函数列（除了常量表达式）。但是你可以使用‘any’（返回遇到的第一个值）、max、min等聚合函数使它工作。

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
```

GROUP BY子句会为遇到的每一个不同的key计算一组聚合函数的值。

在GROUP BY子句中不支持使用Array类型的列。

常量不能作为聚合函数的参数传入聚合函数中。例如：sum(1)。这种情况下你可以省略常量。例如：count()。

NULL 处理

对于GROUP BY子句，ClickHouse将 NULL 解释为一个值，并且支持 NULL=NULL。

下面这个例子将说明这将意味着什么。

假设你有这样一张表：

x	y
1	2
2	NULL
3	2
3	3
3	NULL

运行SELECT sum(x), y FROM t_null_big GROUP BY y你将得到如下结果：

sum(x)	y
4	2
3	3
5	NULL

你可以看到GROUP BY为y=NULL的聚合了x。

如果你在向GROUP BY中放入几个key，结果将列出所有的组合可能。就像 NULL 是一个特定的值一样。

WITH TOTALS 修饰符

如果你指定了WITH TOTALS修饰符，你将会在结果中得到一个被额外计算出的行。在这一行中将包含所有key的默认值（零或者空值），以及所有聚合函数对所有被选择数据行的聚合结果。

该行仅在JSON*, TabSeparated*, Pretty*输出格式中与其他行分开输出。

在JSON*输出格式中，这行将出现在Json的‘totals’字段中。在TabSeparated*输出格式中，这行将位于其他结果之后，同时与其他结果使用空白行分隔。在Pretty*输出格式中，这行将作为单独的表在所有结果之后输出。

当 WITH TOTALS 与 HAVING 子句同时存在时，它的行为受‘totals_mode’配置的影响。

默认情况下，totals_mode = 'before_having'，这时 WITH TOTALS 将会在 HAVING 前计算最多不超过 max_rows_to_group_by 行的数据。

在 group_by_overflow_mode = 'any' 并指定了 max_rows_to_group_by 的情况下，WITH TOTALS 的行为受 totals_mode 的影响。

after_having_exclusive - 在 HAVING 后进行计算，计算不超过 max_rows_to_group_by 行的数据。

after_having_inclusive - 在 HAVING 后进行计算，计算不少于 max_rows_to_group_by 行的数据。

after_having_auto - 在 HAVING 后进行计算，采用统计通过 HAVING 的行数，在超过不超过‘max_rows_to_group_by’指定值（默认为50%）的情况下，包含所有行的结果。否则排除这些结果。

totals_auto_threshold - 默认 0.5，是 after_having_auto 的参数。

如果 group_by_overflow_mode != 'any' 并没有指定 max_rows_to_group_by 情况下，所有的模式都与 after_having 相同。

你可以在子查询，包含子查询的 JOIN 子句中使用 WITH TOTALS（在这种情况下，它们各自的总值会被组合在一起）。

GROUP BY 使用外部存储设备

你可以在 GROUP BY 中允许将临时数据转存到磁盘上，以限制对内存的使用。

max_bytes_before_external_group_by 这个配置确定了在 GROUP BY 中启动将临时数据转存到磁盘上的内存阈值。如果你将它设置为 0（这是默认值），这项功能将被禁用。

当使用 max_bytes_before_external_group_by 时，我们建议将 max_memory_usage 设置为它的两倍。这是因为一个聚合需要两个阶段来完成：（1）读取数据并形成中间数据 （2）合并中间数据。临时数据的转存只会发生在第一个阶段。如果没有发生临时文件的转存，那么阶段二将最多消耗与1阶段相同的内存大小。

例如：如果将 max_memory_usage 设置为 10000000000 并且你想要开启外部聚合，那么你需要

将 max_bytes_before_external_group_by 设置为 10000000000 的同时将 max_memory_usage 设置为 20000000000。当外部聚合被触发时（如果刚好只形成了一份临时数据），它的内存使用量将会稍高与 max_bytes_before_external_group_by。

在分布式查询处理中，外部聚合将会在远程的服务器中执行。为了使请求服务器只使用较少的内存，可以设置 distributed_aggregation_memory_efficient 为 1。

当合并被刷到磁盘的临时数据以及合并远程的服务器返回的结果时，如果在启动 distributed_aggregation_memory_efficient 的情况下，将会消耗 $1/256 * \text{线程数的总内存大小}$ 。

当启动外部聚合时，如果数据的大小小于 max_bytes_before_external_group_by 设置的值（数据没有被刷到磁盘中），那么数据的聚合速度将会和没有启动外部聚合时一样快。如果有临时数据被刷到了磁盘中，那么这个查询的运行时间将会被延长几倍（大约是3倍）。

如果你在 GROUP BY 后面存在 ORDER BY 子句，并且 ORDER BY 后面存在一个极小限制的 LIMIT，那么 ORDER BY 子句将不会使用太多内存。

否则请不要忘记启动外部排序(max_bytes_before_external_sort)。

LIMIT N BY 子句

LIMIT N BY 子句和 LIMIT 没有关系，LIMIT N BY COLUMNS 子句可以用来在每一个 COLUMNS 分组中求得最大的 N 行数据。我们可以将它们同时用在一个查询中。LIMIT N BY 子句中可以包含任意多个分组字段表达式列表。

示例：

```
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

```

查询将会为每个 `domain, device_type` 的组合选出前5个访问最多的数据，但是结果最多将不超过100行（`LIMIT n BY + LIMIT`）。

HAVING 子句

HAVING子句可以用来过滤GROUP BY之后的数据，类似于WHERE子句。

WHERE于HAVING不同之处在于WHERE在聚合前(GROUP BY)执行，HAVING在聚合后执行。

如果不存在聚合，则不能使用HAVING。

ORDER BY 子句

如果存在ORDER BY 子句，则该子句中必须存在一个表达式列表，表达式列表中每一个表达式都可以分配一个DESC或ASC（排序的方向）。如果没有指明排序的方向，将假定以ASC的方式进行排序。其中ASC表示按照升序排序，DESC按照降序排序。示例：`ORDER BY Visits DESC, SearchPhrase`

对于字符串的排序来讲，你可以为其指定一个排序规则，在指定排序规则时，排序总是不会区分大小写。并且如果与ASC或DESC同时出现时，排序规则必须在它们的后面指定。例如：`ORDER BY SearchPhrase COLLATE 'tr'` - 使用土耳其字母表对它进行升序排序，同时排序时不会区分大小写，并按照UTF-8字符集进行编码。

我们推荐只在少量的数据集中使用COLLATE，因为COLLATE的效率远低于正常的字节排序。

针对排序表达式中相同值的行将以任意的顺序进行输出，这是不确定的（每次都可能不同）。

如果省略ORDER BY子句，则结果的顺序也是不固定的。

NaN 和 **NULL** 的排序规则：

- 当使用 **NULLS FIRST** 修饰符时，将会先输出 **NULL**，然后是 **NaN**，最后才是其他值。
- 当使用 **NULLS LAST** 修饰符时，将会先输出其他值，然后是 **NaN**，最后才是 **NULL**。
- 默认情况下与使用 **NULLS LAST** 修饰符相同。

示例:

假设存在如下一张表

x	y
1	NULL
2	2
1	nan
2	2
3	4
5	6
6	nan
7	NULL
6	7
8	9

运行查询 `SELECT * FROM t_null_nan ORDER BY y NULLS FIRST` 将获得如下结果:

x	y
---	---

1	NULL
7	NULL
1	nan
6	nan
2	2
2	2
3	4
5	6
6	7
8	9

当使用浮点类型的数值进行排序时，不管排序的顺序如何，NaNs总是出现在所有值的后面。换句话说，当你使用升序排列一个浮点数值列时，NaNs好像比所有值都要大。反之，当你使用降序排列一个浮点数值列时，NaNs好像比所有值都小。

如果你在ORDER BY子句后面存在LIMIT并给定了较小的数值，则将会使用较少的内存。否则，内存的使用量将与需要排序的数据成正比。对于分布式查询，如果省略了GROUP BY，则在远程服务器上执行部分排序，最后在请求服务器上合并排序结果。这意味这对于分布式查询而言，要排序的数据量可以大于单台服务器的内存。

如果没有足够的内存，可以使用外部排序（在磁盘中创建一些临时文件）。可以使用 `max_bytes_before_external_sort` 来设置外部排序，如果你讲它设置为0（默认），则表示禁用外部排序功能。如果启用该功能。当要排序的数据量达到所指定的字节数时，当前排序的结果会被转存到一个临时文件中。当全部数据读取完毕后，所有的临时文件将会合并成最终输出结果。这些临时文件将会写到config文件配置的/var/lib/clickhouse/tmp/目录中（默认值，你可以通过修改'tmp_path'配置调整该目录的位置）。

查询运行使用的内存要高于‘max_bytes_before_external_sort’，为此，这个配置必须要远远小于‘max_memory_usage’配置的值。例如，如果你的服务器有128GB的内存去运行一个查询，那么推荐你将‘max_memory_usage’设置为100GB，‘max_bytes_before_external_sort’设置为80GB。

外部排序效率要远低于在内存中排序。

SELECT 子句

在完成上述列出的所有子句后，将对SELECT子句中的表达式进行分析。

具体来讲，如果在存在聚合函数的情况下，将对聚合函数之前的表达式进行分析。

聚合函数与聚合函数之前的表达式都将在聚合期间完成计算（GROUP BY）。

就像他们本身就已经存在结果上一样。

DISTINCT 子句

如果存在DISTINCT子句，则会对结果中的完全相同的行进行去重。

在GROUP BY不包含聚合函数，并对全部SELECT部分都包含在GROUP BY中时的作用一样。但该子句还是与GROUP BY子句存在以下几点不同：

- 可以与GROUP BY配合使用。
- 当不存在ORDER BY子句并存在LIMIT子句时，查询将在同时满足DISTINCT与LIMIT的情况下立即停止查询。
- 在处理数据的同时输出结果，并不是等待整个查询全部完成。

在SELECT表达式中存在Array类型的列时，不能使用DISTINCT。

DISTINCT可以与 **NULL**一起工作，就好像NULL仅是一个特殊的值一样，并且NULL=NULL。换言之，在DISTINCT的结果中，与NULL不同的组合仅能出现一次。

LIMIT 子句

LIMIT m 用于在查询结果中选择前m行数据。

LIMIT n, m 用于在查询结果中选择从n行开始的m行数据。

‘n’与‘m’必须是正整数。

如果没有指定ORDER BY子句，则结果可能是任意的顺序，并且是不确定的。

UNION ALL 子句

UNION ALL子句可以组合任意数量的查询，例如：

```
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
```

仅支持UNION ALL，不支持其他UNION规则(UNION DISTINCT)。如果你需要UNION DISTINCT，你可以使用UNION ALL中包含SELECT DISTINCT的子查询的方式。

UNION ALL中的查询可以同时运行，它们的结果将被混合到一起。

这些查询的结果必须相同（列的数量和类型）。列名可以是不同的。在这种情况下，最终结果的列名将从第一个查询中获取。UNION会为查询之间进行类型转换。例如，如果组合的两个查询中包含相同的字段，并且是类型兼容的 `Nullable` 和 `non-Nullable`，则结果将会将该字段转换为 `Nullable` 类型的字段。

作为UNION ALL查询的部分不能包含在括号内。ORDER BY与LIMIT子句应该被应用在每个查询中，而不是最终的查询中。如果你需要做最终结果转换，你可以将UNION ALL作为一个子查询包含在FROM子句中。

INTO OUTFILE 子句

`INTO OUTFILE filename` 子句用于将查询结果重定向输出到指定文件中（filename是一个字符串类型的值）。

与MySQL不同，执行的结果文件将在客户端建立，如果文件已存在，查询将会失败。

此命令可以工作在命令行客户端与clickhouse-local中（通过HTTP借口发送将会失败）。

默认的输出格式是TabSeparated（与命令行客户端的批处理模式相同）。

FORMAT 子句

'FORMAT format' 子句用于指定返回数据的格式。

你可以使用它方便的转换或创建数据的转储。

更多信息，参见“输入输出格式”部分。

如果不存在FORMAT子句，则使用默认的格式，这将取决与DB的配置以及所使用的客户端。对于批量模式的HTTP客户端和命令行客户端而言，默认的格式是TabSeparated。对于交互模式下的命令行客户端，默认的格式是PrettyCompact（它有更加美观的格式）。

当使用命令行客户端时，数据以内部高效的格式在服务器和客户端之间进行传递。客户端将单独的解析FORMAT子句，以帮助数据格式的转换（这将减轻网络和服务器的负载）。

IN 运算符

对于 `IN`、`NOT IN`、`GLOBAL IN`、`GLOBAL NOT IN` 操作符被分别实现，因为它们的功能非常丰富。

运算符的左侧是单列或列的元组。

示例：

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

如果左侧是单个列并且是一个索引，并且右侧是一组常量时，系统将使用索引来处理查询。

不要在列表中列出太多的值（百万）。如果数据集很大，将它们放入临时表中（可以参考“”），然后使用子查询。
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.

右侧可以是一个由常量表达式组成的元组列表（像上面的例子一样），或者是一个数据库中的表的名称，或是一个包含在括号中的子查询。

如果右侧是一个表的名字（例如，`UserID IN users`），这相当于 `UserID IN (SELECT * FROM users)`。在查询与外部数据表组合使用时可以使用该方法。例如，查询与包含user IDS的‘users’临时表一起被发送的同时需要对结果进行过滤时。

如果操作符的右侧是一个Set引擎的表时（数据总是在内存中准备好），则不会每次都为查询创建新的数据集。

子查询可以指定一个列以上的元组来进行过滤。

示例：

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

IN操作符的左右两侧应具有相同的类型。

IN操作符的子查询中可以出现任意子句，包含聚合函数与lambda函数。

示例：

```
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

为3月17日之后的每一天计算与3月17日访问该网站的用户浏览网页的百分比。

IN子句中的子查询仅在单个服务器上运行一次。不能够是相关子查询。

NULL 处理

在处理中，IN操作符总是假定 `NULL` 值的操作结果总是等于 `0`，而不管 `NULL` 位于左侧还是右侧。`NULL` 值不应该包含在任何数据集中，它们彼此不能够对应，并且不能够比较。

下面的示例中有一个 `t_null` 表：

x	y
1	NULL
2	3

运行查询 `SELECT x FROM t_null WHERE y IN (NULL,3)` 将得到如下结果：

```
┌─x─┐
└─2─┘
```

你可以看到在查询结果中不存在 `y = NULL` 的结果。这是因为 ClickHouse 无法确定 `NULL` 是否包含在 `(NULL,3)` 数据集中，对于这次比较操作返回了 `0`，并且在 `SELECT` 的最终输出中排除了这行。

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

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

分布式子查询

对于带有子查询的（类似与 `JOINS`）`IN` 中，有两种选择：普通的 `IN / JOIN` 与 `GLOBAL IN / GLOBAL JOIN`。它们对于分布式查询的处理运行方式是不同的。

注意

请记住，下面描述的算法可能因为根据 `settings` 配置的不同而不同。

当使用普通的 `IN` 时，查询总是被发送到远程的服务器，并且在每个服务器中运行“`IN`”或“`JOIN`”子句中的子查询。

当使用 `GLOBAL IN / GLOBAL JOIN` 时，首先会为 `GLOBAL IN / GLOBAL JOIN` 运行所有子查询，并将结果收集到临时表中，并将临时表发送到每个远程服务器，并使用该临时表运行查询。

对于非分布式查询，请使用普通的 `IN / JOIN`。

在分布式查询中使用 `IN / JOIN` 子句中使用子查询需要小心。

让我们来看一些例子。假设集群中的每个服务器都存在一个正常表 `local_table`。与一个分布式表 `distributed_table`。

对于所有查询 `distributed_table` 的查询，查询会被发送到所有的远程服务器并使用 `local_table` 表运行查询。

例如，查询

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

将发送如下查询到所有远程服务器

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

这时将并行的执行它们，直到达到可以组合数据的中间结果状态。然后中间结果将返回到请求服务器并在请求服务器上合并，最终将结果发送给客户端。

现在让我运行一个带有 `IN` 的查询：

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

- 计算两个站点的用户交集。

此查询将被发送给所有的远程服务器

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

换句话说，IN子句中的数据集将被在每台服务器上被独立的收集，仅与每台服务器上的本地存储上的数据计算交集。

如果您已经将数据分散到了集群的每台服务器上，并且单个UserID的数据完全分布在单个服务器上，那么这将是正确且最佳的查询方式。在这种情况下，所有需要的数据都可以在每台服务器的本地进行获取。否则，结果将是不准确的。我们将这种查询称为“local IN”。

为了修正这种在数据随机分布的集群中的工作，你可以在子查询中使用**distributed_table**。查询将更改为这样：

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

此查询将被发送给所有的远程服务器

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

子查询将在每个远程服务器上执行。因为子查询使用分布式表，所有每个远程服务器上的子查询将查询再次发送给所有的远程服务器

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

例如，如果你拥有100台服务器的集群，执行整个查询将需要10,000次请求，这通常被认为是不可接受的。

在这种情况下，你应该使用GLOBAL IN来替代IN。让我们看一下它是如何工作的。

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

在请求服务器上运行子查询

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

将结果放入内存中的临时表中。然后将请求发送到每一台远程服务器

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

临时表_data1也会随着查询一起发送到每一台远程服务器（临时表的名称由具体实现定义）。

这比使用普通的IN更加理想，但是，请注意以下几点：

1. 创建临时表时，数据不是唯一的，为了减少通过网络传输的数据量，请在子查询中使用DISTINCT（你不需要在普通的IN中这么做）
2. 临时表将发送到所有远程服务器。其中传输不考虑网络的拓扑结构。例如，如果你有10个远程服务器存在与请求服务器非常远的数据中心中，则数据将通过通道发送数据到远程数据中心10次。使用GLOBAL IN时应避免大数据集。
3. 当向远程服务器发送数据时，网络带宽的限制是不可配置的，这可能会网络的负载造成压力。

4. 尝试将数据跨服务器分布，这样你将不需要使用GLOBAL IN。
5. 如果你需要经常使用GLOBAL IN，请规划你的ClickHouse集群位置，以便副本之间不存在跨数据中心，并且它们之间具有快速的网络交换能力，以便查询可以完全在一个数据中心内完成。

另外，在GLOBAL IN子句中使用本地表也是有用的，比如，本地表仅在请求服务器上可用，并且您希望在远程服务器上使用来自本地表的数据。

Extreme Values

除了结果外，你还可以获得结果列的最大值与最小值，可以将**extremes**配置设置成1来做到这一点。最大值最小值的计算是针对数字类型，日期类型进行计算的，对于其他列，将会输出默认值。

额外计算的两行结果 - 最大值与最小值，这两行额外的结果仅在JSON*, TabSeparated*, and Pretty* 格式与其他行分开的输出方式输出，不支持其他输出格式。

在JSON*格式中，Extreme值在单独的'extremes'字段中。在TabSeparated*格式中，在其他结果与'totals'之后输出，并使用空行与其分隔。在Pretty* 格式中，将在其他结果与'totals'后以单独的表格输出。

如果在计算Extreme值的同时包含LIMIT。extremes的计算结果将包含offset跳过的行。在流式的请求中，它可能还包含多余LIMIT的少量行的值。

注意事项

不同于MySQL，GROUP BY与ORDER BY子句不支持使用列的位置信息作为参数，但这实际上是符合SQL标准的。

例如，GROUP BY 1, 2将被解释为按照常量进行分组（即，所有的行将会被聚合成一行）。

可以在查询的任何部分使用AS。

可以在查询的任何部分添加星号，而不仅仅是表达式。在分析查询时，星号被替换为所有的列（不含MATERIALIZED与ALIAS的列）。

只有少数情况下使用星号是合理的：

- 创建表转储时。
- 对于仅包含几个列的表，如系统表。
- 获取表中的列信息。在这种情况下应该使用LIMIT 1。但是，更好的办法是使用DESC TABLE。
- 当使用PREWHERE在少数的几个列上做强过滤时。
- 在子查询中（因为外部查询不需要的列被排除在子查询之外）。

在所有的其他情况下，我们不建议使用星号，因为它是列式数据库的缺点而不是优点。

INSERT

INSERT查询主要用于向系统中添加数据。

查询的基本格式：

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

您可以在查询中指定插入的列的列表，如：[(c1, c2, c3)]。对于存在于表结构中但不存在于插入列表中的列，它们将会按照如下方式填充数据：

- 如果存在DEFAULT表达式，根据DEFAULT表达式计算被填充的值。
- 如果没有定义DEFAULT表达式，则填充零或空字符串。

如果strict_insert_defaults=1，你必须在查询中列出所有没有定义DEFAULT表达式的列。

数据可以以ClickHouse支持的任何输入输出格式传递给INSERT。格式的名称必须显示的指定在查询中：

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

例如，下面的查询所使用的输入格式就与上面INSERT ... VALUES的中使用的输入格式相同：

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

ClickHouse会清除数据前所有的空白字符与一行摘要信息（如果需要的话）。所以在进行查询时，我们建议您将数据放入到输入输出格式名称后的新的一行中去（如果数据是以空白字符开始的，这将非常重要）。

示例：

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

在使用命令行客户端或HTTP客户端时，你可以将具体的查询语句与数据分开发送。更多具体信息，请参考“[客户端](#)”部分。

使用SELECT的结果写入

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

写入与SELECT的列的对应关系是使用位置来进行对应的，尽管它们在SELECT表达式与INSERT中的名称可能是不同的。如果需要，会对它们执行对应的类型转换。

除了VALUES格式之外，其他格式中的数据都不允许出现诸如now()，1 + 2等表达式。VALUES格式允许您有限度的使用这些表达式，但是不建议您这么做，因为执行这些表达式总是低效的。

系统不支持的其他用于修改数据的查询：UPDATE, DELETE, REPLACE, MERGE, UPSERT, INSERT UPDATE。但是，您可以使用 ALTER TABLE ... DROP PARTITION查询来删除一些旧的数据。

性能的注意事项

在进行INSERT时将会对写入的数据进行一些处理，按照主键排序，按照月份对数据进行分区等。所以如果在您的写入数据中包含多个月份的混合数据时，将会显著的降低INSERT的性能。为了避免这种情况：

- 数据总是以尽量大的batch进行写入，如每次写入100,000行。
- 数据在写入ClickHouse前预先的对数据进行分组。

在以下的情况下，性能不会下降：

- 数据总是被实时的写入。
- 写入的数据已经按照时间排序。

CREATE DATABASE

该查询用于根据指定名称创建数据库。

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

数据库其实只是用于存放表的一个目录。

如果查询中存在 IF NOT EXISTS，则当数据库已经存在时，该查询不会返回任何错误。

CREATE TABLE

对于CREATE TABLE，存在以下几种方式。

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
```



```
name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
...
) ENGINE = engine
```

在指定的‘db’数据库中创建一个名为‘name’的表，如果查询中没有包含‘db’，则默认使用当前选择的数据库作为‘db’。后面的是包含在括号中的表结构以及表引擎的声明。

其中表结构声明是一个包含一组列描述声明的组合。如果表引擎是支持索引的，那么可以在表引擎的参数中对其进行说明。

在最简单的情况下，列描述是指名称 类型 这样的子句。例如：`RegionID UInt32`。

但是也可以为列另外定义默认值表达式（见后文）。

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

创建一个与 `db2.name2` 具有相同结构的表，同时你可以对其指定不同的表引擎声明。如果没有表引擎声明，则创建的表将与 `db2.name2` 使用相同的表引擎。

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

使用指定的引擎创建一个与 `SELECT` 子句的结果具有相同结构的表，并使用 `SELECT` 子句的结果填充它。

以上所有情况，如果指定了 `IF NOT EXISTS`，那么在该表已经存在的情况下，查询不会返回任何错误。在这种情况下，查询几乎不会做任何事情。

在 `ENGINE` 子句后还可能存在一些其他的子句，更详细的信息可以参考 [表引擎](#) 中关于建表的描述。

默认值

在列描述中你可以通过以下方式之一为列指定默认表达式：`DEFAULT expr`，`MATERIALIZED expr`，`ALIAS expr`。

示例：`URLDomain String DEFAULT domain(URL)`。

如果在列描述中未定义任何默认表达式，那么系统将会根据类型设置对应的默认值，如：数值类型为零、字符串类型为空字符串、数组类型为空数组、日期类型为‘0000-00-00’以及时间类型为‘0000-00-00 00:00:00’。不支持使用 `NULL` 作为普通类型的默认值。

如果定义了默认表达式，则可以不定义列的类型。如果没有明确的定义类的类型，则使用默认表达式的类型。例如：`EventDate DEFAULT toDate(EventTime)` - 最终‘EventDate’将使用‘Date’作为类型。

如果同时指定了默认表达式与列的类型，则将使用类型转换函数将默认表达式转换为指定的类型。例如：`Hits UInt32 DEFAULT 0` 与 `Hits UInt32 DEFAULT toDate(UInt32(0))` 意思相同。

默认表达式可以包含常量或表的任意其他列。当创建或更改表结构时，系统将会运行检查，确保不会包含循环依赖。对于 `INSERT`，它仅检查表达式是否是可解析的 - 它们可以从中计算出所有需要的列的默认值。

DEFAULT expr

普通的默认值，如果 `INSERT` 中不包含指定的列，那么将通过表达式计算它的默认值并填充它。

MATERIALIZED expr

物化表达式，被该表达式指定的列不能包含在 `INSERT` 的列表中，因为它总是被计算出来的。

对于 `INSERT` 而言，不需要考虑这些列。

另外，在 `SELECT` 查询中如果包含星号，此列不会被用来替换星号，这是因为考虑到数据转储，在使用 `SELECT *` 查询出的结果总能够被 `'INSERT'` 回表。

ALIAS expr

别名。这样的列不会存储在表中。

它的值不能够通过 `INSERT` 写入，同时使用 `SELECT` 查询星号时，这些列也不会被用来替换星号。

但是它们可以显示的用于SELECT中，在这种情况下，在查询分析中别名将被替换。

当使用ALTER查询对添加新的列时，不同于为所有旧数据添加这个列，对于需要在旧数据中查询新列，只会在查询时动态计算这个新列的值。但是如果新列的默认表示中依赖其他列的值进行计算，那么同样会加载这些依赖的列的数据。

如果你向表中添加一个新列，并在之后的一段时间后修改它的默认表达式，则旧数据中的值将会被改变。请注意，在运行后台合并时，缺少的列的值将被计算后写入到合并后的数据部分中。

不能够为nested类型的列设置默认值。

临时表

ClickHouse支持临时表，其具有以下特征：

- 当会话结束时，临时表将随会话一起消失，这包含链接中断。
- 临时表仅能够使用Memory表引擎。
- 无法为临时表指定数据库。它是在数据库之外创建的。
- 如果临时表与另一个表名称相同，那么当在查询时没有显示的指定db的情况下，将优先使用临时表。
- 对于分布式处理，查询中使用的临时表将被传递到远程服务器。

可以使用下面的语法创建一个临时表：

```
CREATE TEMPORARY TABLE [IF NOT EXISTS] table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
)
```

大多数情况下，临时表不是手动创建的，只有在分布式查询处理中使用 (GLOBAL) IN 时为外部数据创建。更多信息，可以参考相关章节。

分布式DDL查询（ON CLUSTER 子句）

对于 CREATE， DROP， ALTER，以及RENAME 查询，系统支持其运行在整个集群上。

例如，以下查询将在 cluster 集群的所有节点上创建名为 all_hits 的 Distributed 表：

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

为了能够正确的运行这种查询，每台主机必须具有相同的cluster声明（为了简化配置的同步，你可以使用zookeeper的方式进行配置）。同时这些主机还必须链接到zookeeper服务器。

这个查询将最终在集群的每台主机上运行，即使一些主机当前处于不可用状态。同时它还保证了所有的查询在单台主机中的执行顺序。

replicated系列表还没有支持ALTER 查询。

CREATE VIEW

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

创建一个视图。它存在两种可选择的类型：普通视图与物化视图。

普通视图不存储任何数据，只是执行从另一个表中的读取。换句话说，普通视图只是保存了视图的查询，当从视图中查询时，此查询被作为子查询用于替换FROM子句。

举个例子，假设你已经创建了一个视图：

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

还有一个查询：

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

这个查询完全等价于：

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

物化视图存储的数据是由相应的SELECT查询转换得来的。

在创建物化视图时，你还必须指定表的引擎 - 将会使用这个表引擎存储数据。

目前物化视图的工作原理：当将数据写入到物化视图中SELECT子句所指定的表时，插入的数据会通过SELECT子句查询进行转换并将最终结果插入到视图中。

如果创建物化视图时指定了POPULATE子句，则在创建时将该表的数据插入到物化视图中。就像使用CREATE TABLE ... AS SELECT ...一样。否则，物化视图只会包含在物化视图创建后的新写入的数据。我们不推荐使用POPULATE，因为在视图创建期间写入的数据将不会写入其中。

当一个SELECT子句包含DISTINCT, GROUP BY, ORDER BY, LIMIT时，请注意，这些仅会在插入数据时在每个单独的数据块上执行。例如，如果你在其中包含了GROUP BY，则只会在查询期间进行聚合，但聚合范围仅限于单个批的写入数据。数据不会进一步被聚合。但是当你使用一些其他数据聚合引擎时这是例外的，如：SummingMergeTree。

目前对物化视图执行ALTER是不支持的，因此这可能是不方便的。如果物化视图是使用的TO [db].name的方式进行构建的，你可以使用DETACH语句现将视图剥离，然后使用ALTER运行在目标表上，然后使用ATTACH将之前剥离的表重新加载进来。

视图看起来和普通的表相同。例如，你可以通过SHOW TABLES查看到它们。

没有单独的删除视图的语法。如果要删除视图，请使用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|CLEAR|COMMENT|MODIFY COLUMN ...
```

In the query, specify a list of one or more comma-separated actions. Each action is an operation on a column.

The following actions are supported:

- **ADD COLUMN** — Adds a new column to the table.
- **DROP COLUMN** — Deletes the column.
- **CLEAR COLUMN** — Resets column values.
- **COMMENT COLUMN** — Adds a text comment to the column.
- **MODIFY COLUMN** — Changes column's type and/or default expression.

Detailed description of these actions is shown below.

ADD COLUMN

```
ADD COLUMN [IF NOT EXISTS] name [type] [default_expr] [AFTER name_after]
```

Adds a new column to the table with the specified `name`, `type`, and `default_expr` (see the section [Default expressions](#)).

If the `IF NOT EXISTS` clause is included, the query won't return an error if the column already exists. If you specify `AFTER name_after` (the name of another column), the column is added after the specified one in the list of table columns. Otherwise, the column is added to the end of the table. Note that there is no way to add a column to the beginning of a table. For a chain of actions, `name_after` can be the name of a column that is added in one of the previous actions.

Adding a column just changes the table structure, without performing any actions with data. The data doesn't appear on the disk after `ALTER`. If the data is missing for a column when reading from the table, it is filled in with default values (by performing the default expression if there is one, or using zeros or empty strings). The column appears on the disk after merging data parts (see [MergeTree](#)).

This approach allows us to complete the `ALTER` query instantly, without increasing the volume of old data.

Example:

```
ALTER TABLE visits ADD COLUMN browser String AFTER user_id
```

DROP COLUMN

```
DROP COLUMN [IF EXISTS] name
```

Deletes the column with the name `name`. If the `IF EXISTS` clause is specified, the query won't return an error if the column doesn't exist.

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

Example:

```
ALTER TABLE visits DROP COLUMN browser
```

CLEAR COLUMN

```
CLEAR COLUMN [IF EXISTS] name IN PARTITION partition_name
```

Resets all data in a column for a specified partition. Read more about setting the partition name in the section [How to specify the partition expression](#).

If the `IF EXISTS` clause is specified, the query won't return an error if the column doesn't exist.

Example:

```
ALTER TABLE visits CLEAR COLUMN browser IN PARTITION tuple()
```

COMMENT COLUMN

```
COMMENT COLUMN [IF EXISTS] name 'comment'
```

Adds a comment to the column. If the `IF EXISTS` clause is specified, the query won't return an error if the column doesn't exist.

column doesn't exist.

Each column can have one comment. If a comment already exists for the column, a new comment overwrites the previous comment.

Comments are stored in the `comment_expression` column returned by the **DESCRIBE TABLE** query.

Example:

```
ALTER TABLE visits COMMENT COLUMN browser 'The table shows the browser used for accessing the site.'
```

MODIFY COLUMN

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

This query changes the `name` column's type to `type` and/or the default expression to `default_expr`. If the **IF EXISTS** clause is specified, the query won't return an error if the column doesn't exist.

When changing the type, values are converted as if the **toType** functions were applied to them. If only the default expression is changed, the query doesn't do anything complex, and is completed almost instantly.

Example:

```
ALTER TABLE visits MODIFY COLUMN browser Array(String)
```

Changing the column type is the only complex action – it changes the contents of files with data. For large tables, this may take a long time.

There are several processing stages:

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

Only the first stage takes time. If there is a failure at this stage, the data is not changed.

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

The **ALTER** query for changing columns is replicated. The instructions are saved in ZooKeeper, then each replica applies them. All **ALTER** queries are run in the same order. The query waits for the appropriate actions to be completed on the other replicas. However, a query to change columns in a replicated table can be interrupted, and all actions will be performed asynchronously.

ALTER Query Limitations

The **ALTER** query lets you create and delete separate elements (columns) in nested data structures, but not whole nested data structures. To add a nested data structure, you can add columns with a name like `name.nested_name` and the type `Array(T)`. A nested data structure is equivalent to multiple array columns with a name that has the same prefix before the dot.

There is no support for deleting columns in the primary key or the sampling key (columns that are used in the **ENGINE** expression). Changing the type for columns that are included in the primary key is only possible if this change does not cause the data to be modified (for example, it is allowed to add values to an Enum or to change a type from `DateTime` to `UInt32`).

If the **ALTER** query is not sufficient to make the table changes you need, you can create a new table, copy the

data to it using the **INSERT SELECT** query, then switch the tables using the **RENAME** query and delete the old table. You can use the **clickhouse-copier** as an alternative to the **INSERT SELECT** query.

The **ALTER** query blocks all reads and writes for the table. In other words, if a long **SELECT** is running at the time of the **ALTER** query, the **ALTER** query will wait for it to complete. At the same time, all new queries to the same table will wait while this **ALTER** is running.

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

Manipulations With Key Expressions

The following command is supported:

```
MODIFY ORDER BY new_expression
```

It only works for tables in the **MergeTree** family (including **replicated** tables). The command changes the **sorting key** of the table to **new_expression** (an expression or a tuple of expressions). Primary key remains the same.

The command is lightweight in a sense that it only changes metadata. To keep the property that data part rows are ordered by the sorting key expression you cannot add expressions containing existing columns to the sorting key (only columns added by the **ADD COLUMN** command in the same **ALTER** query).

Manipulations With Data Skipping Indices

It only works for tables in the ***MergeTree** family (including **replicated** tables). The following operations are available:

- **ALTER TABLE [db].name ADD INDEX name expression TYPE type GRANULARITY value AFTER name [AFTER name2]-** Adds index description to tables metadata.
- **ALTER TABLE [db].name DROP INDEX name** - Removes index description from tables metadata and deletes index files from disk.

These commands are lightweight in a sense that they only change metadata or remove files. Also, they are replicated (syncing indices metadata through ZooKeeper).

Manipulations With Partitions and Parts

The following operations with **partitions** are available:

- **DETACH PARTITION** - Moves a partition to the **detached** directory and forget it.
- **DROP PARTITION** - Deletes a partition.
- **ATTACH PART|PARTITION** - Adds a part or partition from the **detached** directory to the table.
- **REPLACE PARTITION** - Copies the data partition from one table to another.
- **CLEAR COLUMN IN PARTITION** - Resets the value of a specified column in a partition.
- **FREEZE PARTITION** - Creates a backup of a partition.
- **FETCH PARTITION** - Downloads a partition from another server.

DETACH PARTITION

```
ALTER TABLE table_name DETACH PARTITION partition_expr
```

Moves all data for the specified partition to the **detached** directory. The server forgets about the detached data partition as if it does not exist. The server will not know about this data until you make the **ATTACH**

query.

Example:

```
ALTER TABLE visits DETACH PARTITION 201901
```

Read about setting the partition expression in a section [How to specify the partition expression](#).

After the query is executed, you can do whatever you want with the data in the `detached` directory — delete it from the file system, or just leave it.

This query is replicated – it moves the data to the `detached` directory on all replicas. Note that you can execute this query only on a leader replica. To find out if a replica is a leader, perform the `SELECT` query to the `system.replicas` table. Alternatively, it is easier to make a `DETACH` query on all replicas - all the replicas throw an exception, except the leader replica.

DROP PARTITION

```
ALTER TABLE table_name DROP PARTITION partition_expr
```

Deletes the specified partition from the table. This query tags the partition as inactive and deletes data completely, approximately in 10 minutes.

Read about setting the partition expression in a section [How to specify the partition expression](#).

The query is replicated – it deletes data on all replicas.

ATTACH PARTITION|PART

```
ALTER TABLE table_name ATTACH PARTITION|PART partition_expr
```

Adds data to the table from the `detached` directory. It is possible to add data for an entire partition or for a separate part. Examples:

```
ALTER TABLE visits ATTACH PARTITION 201901;  
ALTER TABLE visits ATTACH PART 201901_2_2_0;
```

Read more about setting the partition expression in a section [How to specify the partition expression](#).

This query is replicated. Each replica checks whether there is data in the `detached` directory. If the data is in this directory, the query checks the integrity, verifies that it matches the data on the server that initiated the query. If everything is correct, the query adds data to the replica. If not, it downloads data from the query requestor replica, or from another replica where the data has already been added.

So you can put data to the `detached` directory on one replica, and use the `ALTER ... ATTACH` query to add it to the table on all replicas.

REPLACE PARTITION

```
ALTER TABLE table2 REPLACE PARTITION partition_expr FROM table1
```

This query copies the data partition from the `table1` to `table2`. Note that data won't be deleted from `table1`.

For the query to run successfully, the following conditions must be met:

- Both tables must have the same structure.

- Both tables must have the same partition key.

CLEAR COLUMN IN PARTITION

```
ALTER TABLE table_name CLEAR COLUMN column_name IN PARTITION partition_expr
```

Resets all values in the specified column in a partition. If the `DEFAULT` clause was determined when creating a table, this query sets the column value to a specified default value.

Example:

```
ALTER TABLE visits CLEAR COLUMN hour in PARTITION 201902
```

FREEZE PARTITION

```
ALTER TABLE table_name FREEZE [PARTITION partition_expr]
```

This query creates a local backup of a specified partition. If the `PARTITION` clause is omitted, the query creates the backup of all partitions at once.

Note that for old-styled tables you can specify the prefix of the partition name (for example, '2019') - then the query creates the backup for all the corresponding partitions. Read about setting the partition expression in a section [How to specify the partition expression](#).

Note

The entire backup process is performed without stopping the server.

At the time of execution, for a data snapshot, the query creates hardlinks to a table data. Hardlinks are placed in the directory `/var/lib/clickhouse/shadow/N/...`, where:

- `/var/lib/clickhouse/` is the working ClickHouse directory specified in the config.
- `N` is the incremental number of the backup.

The same structure of directories is created inside the backup as inside `/var/lib/clickhouse/`. The query performs 'chmod' for all files, forbidding writing into them.

After creating the backup, you can copy the data from `/var/lib/clickhouse/shadow/` to the remote server and then delete it from the local server. Note that the `ALTER t FREEZE PARTITION` query is not replicated. It creates a local backup only on the local server.

The query creates backup almost instantly (but first it waits for the current queries to the corresponding table to finish running).

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

To restore data from a backup, do the following:

1. Create the table if it does not exist. To view the query, use the .sql file (replace `ATTACH` in it with `CREATE`).
2. Copy the data from the `data/database/table/` directory inside the backup to the `/var/lib/clickhouse/data/database/table/detached/` directory.
3. Run `ALTER TABLE t ATTACH PARTITION` queries to add the data to a table.

Restoring from a backup doesn't require stopping the server.

For more information about backups and restoring data, see the [Data Backup](#) section.

FETCH PARTITION

```
ALTER TABLE table_name FETCH PARTITION partition_expr FROM 'path-in-zookeeper'
```

Downloads a partition from another server. This query only works for the replicated tables.

The query does the following:

1. Downloads the partition from the specified shard. In 'path-in-zookeeper' you must specify a path to the shard in ZooKeeper.
2. Then the query puts the downloaded data to the `detached` directory of the `table_name` table. Use the **ATTACH PARTITION|PART** query to add the data to the table.

For example:

```
ALTER TABLE users FETCH PARTITION 201902 FROM '/clickhouse/tables/01-01/visits';  
ALTER TABLE users ATTACH PARTITION 201902;
```

Note that:

- The `ALTER ... FETCH PARTITION` query isn't replicated. It places the partition to the `detached` directory only on the local server.
- The `ALTER TABLE ... ATTACH` query is replicated. It adds the data to all replicas. The data is added to one of the replicas from the `detached` directory, and to the others - from neighboring replicas.

Before downloading, the system checks if the partition exists and the table structure matches. The most appropriate replica is selected automatically from the healthy replicas.

Although the query is called `ALTER TABLE`, it does not change the table structure and does not immediately change the data available in the table.

How To Set Partition Expression

You can specify the partition expression in `ALTER ... PARTITION` queries in different ways:

- As a value from the `partition` column of the `system.parts` table. For example, `ALTER TABLE visits DETACH PARTITION 201901`.
- As the expression from the table column. Constants and constant expressions are supported. For example, `ALTER TABLE visits DETACH PARTITION toYYYYMM(toDate('2019-01-25'))`.
- Using the partition ID. Partition ID is a string identifier of the partition (human-readable, if possible) that is used as the names of partitions in the file system and in ZooKeeper. The partition ID must be specified in the `PARTITION ID` clause, in a single quotes. For example, `ALTER TABLE visits DETACH PARTITION ID '201901'`.
- In the **ALTER ATTACH PART** query, to specify the name of a part, use a value from the `name` column of the `system.parts` table. For example, `ALTER TABLE visits ATTACH PART 201901_1_1_0`.

Usage of quotes when specifying the partition depends on the type of partition expression. For example, for the `String` type, you have to specify its name in quotes ('). For the `Date` and `Int*` types no quotes are needed.

For old-style tables, you can specify the partition either as a number `201901` or a string `'201901'`. The syntax for the new-style tables is stricter with types (similar to the parser for the `VALUES` input format).

All the rules above are also true for the **OPTIMIZE** query. If you need to specify the only partition when optimizing a non-partitioned table, set the expression `PARTITION tuple()`. For example:

```
OPTIMIZE TABLE table_not_partitioned PARTITION tuple() FINAL;
```


The examples of `ALTER ... PARTITION` queries are demonstrated in the tests `00502_custom_partitioning_local` and `00502_custom_partitioning_replicated_zookeeper`.

Synchronicity of ALTER Queries

For non-replicable tables, all `ALTER` queries are performed synchronously. For replicable tables, the query just adds instructions for the appropriate actions to `ZooKeeper`, and the actions themselves are performed as soon as possible. However, the query can wait for these actions to be completed on all the replicas.

For `ALTER ... ATTACH|DETACH|DROP` queries, you can use the `replication_alter_partitions_sync` setting to set up waiting.

Possible values: `0` - do not wait; `1` - only wait for own execution (default); `2` - wait for all.

Mutations

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

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

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

Currently available commands:

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

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

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

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

One query can contain several commands separated by commas.

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

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

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

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

Miscellaneous Queries

ATTACH

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

- Instead of the word `CREATE` it uses the word `ATTACH`.
- The query does not create data on the disk, but assumes that data is already in the appropriate places, and just adds information about the table to the server.

After executing an `ATTACH` query, the server will know about the existence of the table.

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

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

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

CHECK TABLE

Checks if the data in the table is corrupted.

```
CHECK TABLE [db.]name
```

The `CHECK TABLE` query compares actual file sizes with the expected values which are stored on the server. If the file sizes do not match the stored values, it means the data is corrupted. This can be caused, for example, by a system crash during query execution.

The query response contains the `result` column with a single row. The row has a value of **Boolean** type:

- 0 - The data in the table is corrupted.
- 1 - The data maintains integrity.

The `CHECK TABLE` query is only supported for the following table engines:

- **Log**
- **TinyLog**
- **StripeLog**

These engines do not provide automatic data recovery on failure. Use the `CHECK TABLE` query to track data loss in a timely manner.

To avoid data loss use the **MergeTree** family tables.

If the data is corrupted

If the table is corrupted, you can copy the non-corrupted data to another table. To do this:

1. Create a new table with the same structure as damaged table. To do this execute the query `CREATE TABLE <new_table_name> AS <damaged_table_name>`.
2. Set the **max_threads** value to 1 to process the next query in a single thread. To do this run the query `SET max_threads = 1`

```
SET max_threads = 1.
```

3. Execute the query `INSERT INTO <new_table_name> SELECT * FROM <damaged_table_name>`. This request copies the non-corrupted data from the damaged table to another table. Only the data before the corrupted part will be copied.
4. Restart the `clickhouse-client` to reset the `max_threads` value.

DESCRIBE TABLE

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

Returns the following `String` type columns:

- `name` — Column name.
- `type` — Column type.
- `default_type` — Clause that is used in **default expression** (`DEFAULT`, `MATERIALIZED` or `ALIAS`). Column contains an empty string, if the default expression isn't specified.
- `default_expression` — Value specified in the `DEFAULT` clause.
- `comment_expression` — Comment text.

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

DETACH

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

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

This does not delete the table's data or metadata. On the next server launch, the server will read the metadata and find out about the table again.

Similarly, a "detached" table can be re-attached using the `ATTACH` query (with the exception of system tables, which do not have metadata stored for them).

There is no `DETACH DATABASE` query.

DROP

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

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

Deletes all tables inside the 'db' database, then deletes the 'db' database itself.

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

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

Deletes the table.

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

EXISTS

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

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

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

KILL QUERY

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

Attempts to forcibly terminate the currently running queries.

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

Examples:

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

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

Read-only users can only stop their own queries.

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

The synchronous version (`SYNC`) waits for all queries to stop and displays information about each process as it stops.

The response contains the `kill_status` column, which can take the following values:

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

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

KILL MUTATION

```
KILL MUTATION [ON CLUSTER cluster]
WHERE <where expression to SELECT FROM system.mutations query>
[TEST]
[FORMAT format]
```

Tries to cancel and remove **mutations** that are currently executing. Mutations to cancel are selected from the `system.mutations` table using the filter specified by the `WHERE` clause of the `KILL` query.

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

Examples:

```
-- Cancel and remove all mutations of the single table:
KILL MUTATION WHERE database = 'default' AND table = 'table'

-- Cancel the specific mutation:
KILL MUTATION WHERE database = 'default' AND table = 'table' AND mutation_id = 'mutation_3.txt'
```

The query is useful when a mutation is stuck and cannot finish (e.g. if some function in the mutation query throws an exception when applied to the data contained in the table).

Changes already made by the mutation are not rolled back.

OPTIMIZE

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

Asks the table engine to do something for optimization.

Supported only by `*MergeTree` engines, in which this query initializes a non-scheduled merge of data parts.

If you specify a `PARTITION`, only the specified partition will be optimized.

If you specify `FINAL`, optimization will be performed even when all the data is already in one part.

Warning

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

RENAME

Renames one or more tables.

```
RENAME TABLE [db11.]name11 TO [db12.]name12, [db21.]name21 TO [db22.]name22, ... [ON CLUSTER cluster]
```

All tables are renamed under global locking. Renaming tables is a light operation. If you indicated another database after TO, the table will be moved to this database. However, the directories with databases must reside in the same file system (otherwise, an error is returned).

SET

```
SET param = value
```

Allows you to set `param` to `value`. You can also make all the settings from the specified settings profile in a single query. To do this, specify 'profile' as the setting name. For more information, see the section "Settings".

The setting is made for the session, or for the server (globally) if `GLOBAL` is specified.

When making a global setting, the setting is not applied to sessions already running, including the current session. It will only be used for new sessions.

When the server is restarted, global settings made using `SET` are lost.

To make settings that persist after a server restart, you can only use the server's config file.

SHOW CREATE TABLE

```
SHOW CREATE [TEMPORARY] TABLE [db.]table [INTO OUTFILE filename] [FORMAT format]
```

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

SHOW DATABASES

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

Prints a list of all databases.

This query is identical to `SELECT name FROM system.databases [INTO OUTFILE filename] [FORMAT format]`.

See also the section "Formats".

SHOW PROCESSLIST

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

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

Prints a table containing the columns:

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

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

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

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

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

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

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

This query is nearly identical to: `SELECT * FROM system.processes`. The difference is that the `SHOW PROCESSLIST` query does not show itself in a list, when the `SELECT .. FROM system.processes` query does.

Tip (execute in the console):

```
watch -n1 "clickhouse-client --query='SHOW PROCESSLIST'"
```

SHOW TABLES

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

Displays a list of tables

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

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

See also the section "LIKE operator".

TRUNCATE

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

Removes all data from a table. When the clause `IF EXISTS` is omitted, the query returns an error if the table does not exist.

The `TRUNCATE` query is not supported for **View**, **File**, **URL** and **Null** table engines.

USE

USE db

Lets you set the current database for the session.

The current database is used for searching for tables if the database is not explicitly defined in the query with a dot before the table name.

This query can't be made when using the HTTP protocol, since there is no concept of a session.

Functions

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

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

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

Strong typing

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

Common subexpression elimination

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

Types of results

All functions return a single 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.

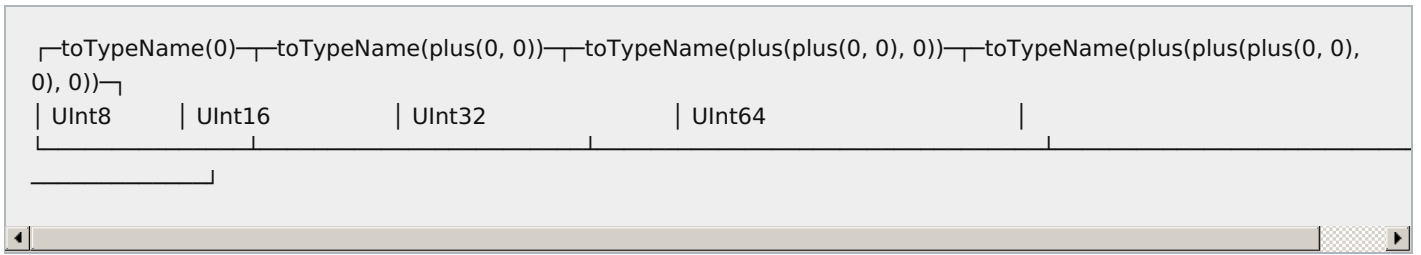
If a function in a query is performed on the requestor server, but you need to perform it on remote servers, you can wrap it in an 'any' aggregate function or add it to a key in `GROUP BY`.

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)
```



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, $a \neq b$ and $a <> b$

less, $a < b$ operator

greater, $a > b$ operator

lessOrEquals, $a \leq b$ operator

greaterOrEquals, $a \geq b$ operator

Logical functions

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

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

and, AND operator

or, OR operator

not, NOT operator

xor

Type conversion functions

toUInt8, toUInt16, toUInt32, toUInt64

toInt8, toInt16, toInt32, toInt64

toFloat32, toFloat64

toDate, toDateTime

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

toUInt8OrNull, toUInt16OrNull, toUInt32OrNull,
toUInt64OrNull, toInt8OrNull, toInt16OrNull, toInt32OrNull,
toInt64OrNull, toFloat32OrNull, toFloat64OrNull, toDateOrNull,
toDateTimeOrNull

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

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

toString

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

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

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

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

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

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

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

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

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

C++.

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

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

now_local	now_yekat
2016-06-15 00:11:21	2016-06-15 02:11:21

Also see the `toUnixTimestamp` function.

toFixedString(s, N)

Converts a `String` type argument to a `FixedString(N)` type (a string with fixed length `N`). `N` must be a constant.

If the string has fewer bytes than `N`, it is passed with null bytes to the right. If the string has more bytes than `N`, an exception is thrown.

toStringCutToZero(s)

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

Example:

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

s	s_cut
foo\0\0\0\0\0	foo

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

s	s_cut
foo\0bar\0	foo

`reinterpretAsUInt8`, `reinterpretAsUInt16`, `reinterpretAsUInt32`,
`reinterpretAsUInt64`

`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

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.

reinterpretAsFixedString

This function accepts a number or date or date with time, and returns a FixedString containing bytes representing the corresponding value in host order (little endian). Null bytes are dropped from the end. For example, a UInt32 type value of 255 is a FixedString that is one byte long.

CAST(x, t)

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

Example:

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

timestamp	datetime	date	string	fixed_string
2016-06-15 23:00:00	2016-06-15 23:00:00	2016-06-15	2016-06-15 23:00:00	2016-06-15 23:00:00\0\0\0

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

Type conversion to **Nullable** and back is supported. Example:

```
SELECT toTypeName(x) FROM t_null
```

toTypeName(x)
Int8
Int8

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

toTypeName(CAST(x, 'Nullable(UInt16)'))
Nullable(UInt16)
Nullable(UInt16)

toIntervalYear, toIntervalQuarter, toIntervalMonth,
toIntervalWeek, toIntervalDay, toIntervalHour.

toIntervalWeek, toIntervalDay, toIntervalHour, toIntervalMinute, toIntervalSecond

Converts a Number type argument to a Interval type (duration).

The interval type is actually very useful, you can use this type of data to perform arithmetic operations directly with Date or DateTime. At the same time, ClickHouse provides a more convenient syntax for declaring Interval type data. For example:

```
WITH
  toDate('2019-01-01') AS date,
  INTERVAL 1 WEEK AS interval_week,
  toIntervalWeek(1) AS interval_to_week
SELECT
  date + interval_week,
  date + interval_to_week
```

plus(date, interval_week)	plus(date, interval_to_week)
2019-01-08	2019-01-08

parseDateTimeBestEffort

Parse a number type argument to a Date or DateTime type.

different from toDate and toDateTime, parseDateTimeBestEffort can progress more complex date format.

For more information, see the link: [Complex Date Format](#)

parseDateTimeBestEffortOrNull

Same as for [parseDateTimeBestEffort](#) except that it returns null when it encounters a date format that cannot be processed.

parseDateTimeBestEffortOrZero

Same as for [parseDateTimeBestEffort](#) except that it returns zero date or zero date time when it encounters a date format that cannot be processed.

Functions for working with dates and times

Support for time zones

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

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

time	date_local	date_yekat	time_samoa
2016-06-15 23:00:00	2016-06-15	2016-06-16	2016-06-15 09:00:00

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

toTimeZone

toTimezone

Convert time or date and time to the specified time zone.

toYear

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

toQuarter

Converts a date or date with time to a UInt8 number containing the quarter number.

toMonth

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

toDayOfYear

Converts a date or date with time to a UInt8 number containing the number of the day of the year (1-366).

toDayOfMonth

Converts a date or date with time to a UInt8 number containing the number of the day of the month (1-31).

toDayOfWeek

Converts a date or date with time to a UInt8 number containing the number of the day of the week (Monday is 1, and Sunday is 7).

toHour

Converts a date with time to a UInt8 number containing the number of the hour in 24-hour time (0-23). This function assumes that if clocks are moved ahead, it is by one hour and occurs at 2 a.m., and if clocks are moved back, it is by one hour and occurs at 3 a.m. (which is not always true – even in Moscow the clocks were twice changed at a different time).

toMinute

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

toSecond

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

toUnixTimestamp

Converts a date with time to a unix timestamp.

toStartOfYear

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

toStartOfISOYear

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

toStartOfQuarter

Rounds down a date or date with time to the first day of the quarter.
The first day of the quarter is either 1 January, 1 April, 1 July, or 1 October.
Returns the date.

Returns the date.

toStartOfMonth

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

Attention

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

toMonday

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

toStartOfDay

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

toStartOfHour

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

toStartOfMinute

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

toStartOfFiveMinute

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

toStartOfTenMinutes

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

toStartOfFifteenMinutes

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

toStartOfInterval(time_or_data, INTERVAL x unit [, time_zone])

This is a generalization of other functions named `toStartOf*`. For example,
`toStartOfInterval(t, INTERVAL 1 year)` returns the same as `toStartOfYear(t)`,
`toStartOfInterval(t, INTERVAL 1 month)` returns the same as `toStartOfMonth(t)`,
`toStartOfInterval(t, INTERVAL 1 day)` returns the same as `toStartOfDay(t)`,
`toStartOfInterval(t, INTERVAL 15 minute)` returns the same as `toStartOfFifteenMinutes(t)` etc.

toTime

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

toRelativeYearNum

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

toRelativeQuarterNum

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

toRelativeMonthNum

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

toRelativeWeekNum

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

toRelativeDayNum

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

toRelativeHourNum

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

toRelativeMinuteNum

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

toRelativeSecondNum

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

toISOYear

Converts a date or date with time to a UInt16 number containing the ISO Year number.

toISOWeek

Converts a date or date with time to a UInt8 number containing the ISO Week number.

now

Accepts zero arguments and returns the current time at one of the moments of request execution. This function returns a constant, even if the request took a long time to complete.

today

Accepts zero arguments and returns the current date at one of the moments of request execution. The same as 'toDate(now())'.

yesterday

Accepts zero arguments and returns yesterday's date at one of the moments of request execution. The same as 'today() - 1'.

timeSlot

Rounds the time to the half hour.

This function is specific to Yandex.Metrica, since half an hour is the minimum amount of time for breaking a session into two sessions if a tracking tag shows a single user's consecutive pageviews that differ in time by strictly more than this amount. This means that tuples (the tag ID, user ID, and time slot) can be used to search for pageviews that are included in the corresponding session.

toYYYYMM

Converts a date or date with time to a UInt32 number containing the year and month number (YYYY * 100 + MM).

toYYYYMMDD

Converts a date or date with time to a UInt32 number containing the year and month number (YYYY * 10000

Converts a date or date with time to a UInt32 number containing the year and month number (YYYY * 10000 + MM * 100 + DD).

toYYYYMMDDhhmmss

Converts a date or date with time to a UInt64 number containing the year and month number (YYYY * 10000000000 + MM * 100000000 + DD * 1000000 + hh * 10000 + mm * 100 + ss).

addYears, addMonths, addWeeks, addDays, addHours, addMinutes, addSeconds, addQuarters

Function adds a Date/DateTime interval to a Date/DateTime and then return the Date/DateTime. For example:

```
WITH
  toDate('2018-01-01') AS date,
  toDateTime('2018-01-01 00:00:00') AS date_time
SELECT
  addYears(date, 1) AS add_years_with_date,
  addYears(date_time, 1) AS add_years_with_date_time
```

add_years_with_date	add_years_with_date_time
2019-01-01	2019-01-01 00:00:00

subtractYears, subtractMonths, subtractWeeks, subtractDays, subtractHours, subtractMinutes, subtractSeconds, subtractQuarters

Function subtract a Date/DateTime interval to a Date/DateTime and then return the Date/DateTime. For example:

```
WITH
  toDate('2019-01-01') AS date,
  toDateTime('2019-01-01 00:00:00') AS date_time
SELECT
  subtractYears(date, 1) AS subtract_years_with_date,
  subtractYears(date_time, 1) AS subtract_years_with_date_time
```

subtract_years_with_date	subtract_years_with_date_time
2018-01-01	2018-01-01 00:00:00

dateDiff('unit', t1, t2, [timezone])

Return the difference between two times expressed in 'unit' e.g. 'hours'. 't1' and 't2' can be Date or DateTime, If 'timezone' is specified, it applied to both arguments. If not, timezones from datatypes 't1' and 't2' are used. If that timezones are not the same, the result is unspecified.

Supported unit values:

unit

second

unit
hour
day
week
month
quarter
year

timeSlots(StartTime, Duration,[, Size])

For a time interval starting at 'StartTime' and continuing for 'Duration' seconds, it returns an array of moments in time, consisting of points from this interval rounded down to the 'Size' in seconds. 'Size' is an optional parameter: a constant UInt32, set to 1800 by default.

For example, `timeSlots(toDateTime('2012-01-01 12:20:00'), 600) = [toDateTime('2012-01-01 12:00:00'), toDateTime('2012-01-01 12:30:00')]`.

This is necessary for searching for pageviews in the corresponding session.

formatDateTime(Time, Format[, Timezone])

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

Supported modifiers for Format:

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

Modifier	Description	Example
%C	year divided by 100 and truncated to integer (00-99)	20
%d	day of the month, zero-padded (01-31)	02
%D	Short MM/DD/YY date, equivalent to %m/%d/%y	01/02/2018
%e	day of the month, space-padded (1-31)	2
%F	short YYYY-MM-DD date, equivalent to %Y-%m-%d	2018-01-02
%H	hour in 24h format (00-23)	22
%I	hour in 12h format (01-12)	10
%j	day of the year (001-366)	002
%m	month as a decimal number (01-12)	01
%M	minute (00-59)	33
%n	new-line character ('\n')	

Modifier %p	Description AM or PM designation	Example PM
%R	24-hour HH:MM time, equivalent to %H:%M	22:33
%S	second (00-59)	44
%t	horizontal-tab character ('\t')	
%T	ISO 8601 time format (HH:MM:SS), equivalent to %H:%M:%S	22:33:44
%u	ISO 8601 weekday as number with Monday as 1 (1-7)	2
%V	ISO 8601 week number (01-53)	01
%w	weekday as a decimal number with Sunday as 0 (0-6)	2
%y	Year, last two digits (00-99)	18
%Y	Year	2018
%%	a % sign	%

Functions for working with strings

empty

Returns 1 for an empty string or 0 for a non-empty string.

The result type is UInt8.

A string is considered non-empty if it contains at least one byte, even if this is a space or a null byte.

The function also works for arrays.

notEmpty

Returns 0 for an empty string or 1 for a non-empty string.

The result type is UInt8.

The function also works for arrays.

length

Returns the length of a string in bytes (not in characters, and not in code points).

The result type is UInt64.

The function also works for arrays.

lengthUTF8

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

The result type is UInt64.

char_length, CHAR_LENGTH

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

The result type is UInt64

The result type is UInt64.

character_length, CHARACTER_LENGTH

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

The result type is UInt64.

lower, lcase

Converts ASCII Latin symbols in a string to lowercase.

upper, ucase

Converts ASCII Latin symbols in a string to uppercase.

lowerUTF8

Converts a string to lowercase, assuming the string contains a set of bytes that make up a UTF-8 encoded text.

It doesn't detect the language. So for Turkish the result might not be exactly correct.

If the length of the UTF-8 byte sequence is different for upper and lower case of a code point, the result may be incorrect for this code point.

If the string contains a set of bytes that is not UTF-8, then the behavior is undefined.

upperUTF8

Converts a string to uppercase, assuming the string contains a set of bytes that make up a UTF-8 encoded text.

It doesn't detect the language. So for Turkish the result might not be exactly correct.


If the length of the UTF-8 byte sequence is different for upper and lower case of a code point, the result may be incorrect for this code point.

If the string contains a set of bytes that is not UTF-8, then the behavior is undefined.

isValidUTF8

Returns 1, if the set of bytes is valid UTF-8 encoded, otherwise 0.

toValidUTF8

Replaces invalid UTF-8 characters by the  (U+FFFD) character. All running in a row invalid characters are collapsed into the one replacement character.

```
toValidUTF8( input_string )
```






Parameters:

- input_string — Any set of bytes represented as the **String** data type object.

Returned value: Valid UTF-8 string.

Example

```
SELECT toValidUTF8('\x61\xf0\x80\x80\x80b')
```

```
┌toValidUTF8('ab')┐  
└ab┘
```


reverse

Reverses the string (as a sequence of bytes).

reverseUTF8

Reverses a sequence of Unicode code points, assuming that the string contains a set of bytes representing a UTF-8 text. Otherwise, it does something else (it doesn't throw an exception).

format(pattern, s0, s1, ...)

Formatting constant pattern with the string listed in the arguments. `pattern` is a simplified Python format pattern. Format string contains "replacement fields" surrounded by curly braces `{}`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: `{{` and `}}`. Field names can be numbers (starting from zero) or empty (then they are treated as consequence numbers).

```
SELECT format('{1} {0} {1}', 'World', 'Hello')

└─format('{1} {0} {1}', 'World', 'Hello')─┐
| Hello World Hello                       |
└────────────────────────────────────────┘

SELECT format('{} {}'. 'Hello', 'World')

└─format('{} {}'. 'Hello', 'World')─┐
| Hello World                        |
└──────────────────────────────────┘
```

concat(s1, s2, ...)

Concatenates the strings listed in the arguments, without a separator.

concatAssumeInjective(s1, s2, ...)

Same as `concat`, the difference is that you need to ensure that `concat(s1, s2, s3) -> s4` is injective, it will be used for optimization of GROUP BY

substring(s, offset, length), mid(s, offset, length), substr(s, offset, length)

Returns a substring starting with the byte from the 'offset' index that is 'length' bytes long. Character indexing starts from one (as in standard SQL). The 'offset' and 'length' arguments must be constants.

substringUTF8(s, offset, length)

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

appendTrailingCharIfAbsent(s, c)

If the 's' string is non-empty and does not contain the 'c' character at the end, it appends the 'c' character to the end.

convertCharset(s, from, to)

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

base64Encode(s)

Encodes 's' string into base64

base64Decode(s)

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

tryBase64Decode(s)

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

endsWith(s, suffix)

Returns whether to end with the specified suffix. Returns 1 if the string ends with the specified suffix, otherwise it returns 0.

startsWith(s, prefix)

Returns whether to start with the specified prefix. Returns 1 if the string starts with the specified prefix, otherwise it returns 0.

trimLeft(s)

Returns a string that removes the whitespace characters on left side.

trimRight(s)

Returns a string that removes the whitespace characters on right side.

trimBoth(s)

Returns a string that removes the whitespace characters on either side.

Functions for Searching Strings

The search is case-sensitive by default in all these functions. There are separate variants for case insensitive search.

position(haystack, needle), locate(haystack, needle)

Search for the substring `needle` in the string `haystack`.

Returns the position (in bytes) of the found substring, starting from 1, or returns 0 if the substring was not found.

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

positionUTF8(haystack, needle)

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

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

multiSearchAllPositions(haystack, [needle₁, needle₂, ..., needle_n])

The same as `position`, but returns `Array` of the `positions` for all `needlei`.

For a case-insensitive search or/and in UTF-8 format use functions `multiSearchAllPositionsCaseInsensitive`, `multiSearchAllPositionsUTF8`, `multiSearchAllPositionsCaseInsensitiveUTF8`.

multiSearchFirstPosition(haystack, [needle₁, needle₂, ..., needle_n])

The same as `position` but returns the leftmost offset of the string `haystack` that is matched to some of the needles.

For a case-insensitive search or/and in UTF-8 format use functions `multiSearchFirstPositionCaseInsensitive`, `multiSearchFirstPositionUTF8`, `multiSearchFirstPositionCaseInsensitiveUTF8`.

multiSearchFirstIndex(haystack, [needle₁, needle₂, ..., needle_n])

Returns the index `i` (starting from 1) of the leftmost found needle_i in the string `haystack` and 0 otherwise.

For a case-insensitive search or/and in UTF-8 format use functions `multiSearchFirstIndexCaseInsensitive`, `multiSearchFirstIndexUTF8`, `multiSearchFirstIndexCaseInsensitiveUTF8`.

multiSearchAny(haystack, [needle₁, needle₂, ..., needle_n])

Returns 1, if at least one string needle_i matches the string `haystack` and 0 otherwise.

For a case-insensitive search or/and in UTF-8 format use functions `multiSearchAnyCaseInsensitive`, `multiSearchAnyUTF8`, `multiSearchAnyCaseInsensitiveUTF8`.

Note: in all `multiSearch*` functions the number of needles should be less than 2⁸ because of implementation specification.

match(haystack, pattern)

Checks whether the string matches the `pattern` regular expression. A `re2` regular expression. The **syntax** of the `re2` regular expressions is more limited than the syntax of the Perl regular expressions.

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

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

The regular expression works with the string as if it is a set of bytes. The regular expression can't contain null bytes.

For patterns to search for substrings in a string, it is better to use `LIKE` or `'position'`, since they work much faster.

multiMatchAny(haystack, [pattern₁, pattern₂, ..., pattern_n])

The same as `match`, but returns 0 if none of the regular expressions are matched and 1 if any of the patterns matches. It uses **hyperscan** library. For patterns to search substrings in a string, it is better to use `multiSearchAny` since it works much faster.

Note: the length of any of the `haystack` string must be less than 2³² bytes otherwise the exception is thrown. This restriction takes place because of hyperscan API.

multiMatchAnyIndex(haystack, [pattern₁, pattern₂, ..., pattern_n])

The same as `multiMatchAny`, but returns any index that matches the haystack.

multiFuzzyMatchAny(haystack, distance, [pattern₁, pattern₂, ..., pattern_n])

..., pattern_n])

The same as `multiMatchAny`, but returns 1 if any pattern matches the haystack within a constant **edit distance**. This function is also in an experimental mode and can be extremely slow. For more information see [hyperscan documentation](#).

`multiFuzzyMatchAnyIndex(haystack, distance, [pattern1, pattern2, ..., patternn])`

The same as `multiFuzzyMatchAny`, but returns any index that matches the haystack within a constant edit distance.

Note: `multiFuzzyMatch*` functions do not support UTF-8 regular expressions, and such expressions are treated as bytes because of hyperscan restriction.

Note: to turn off all functions that use hyperscan, use setting `SET allow_hyperscan = 0;`.

`extract(haystack, pattern)`

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

`extractAll(haystack, pattern)`

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

`like(haystack, pattern)`, haystack LIKE pattern operator

Checks whether a string matches a simple regular expression.

The regular expression can contain the metasympols `%` and `_`.

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

`_` indicates any one byte.

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

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

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

The same thing as 'like', but negative.

`ngramDistance(haystack, needle)`

Calculates the 4-gram distance between `haystack` and `needle`: counts the symmetric difference between two multisets of 4-grams and normalizes it by the sum of their cardinalities. Returns float number from 0 to 1 -- the closer to zero, the more strings are similar to each other. If the constant `needle` or `haystack` is more than 32Kb, throws an exception. If some of the non-constant `haystack` or `needle` strings are more than 32Kb, the distance is always one.

For case-insensitive search or/and in UTF-8 format use functions `ngramDistanceCaseInsensitive`, `ngramDistanceUTF8`, `ngramDistanceCaseInsensitiveUTF8`.

## ngramSearch(haystack, needle)

Same as `ngramDistance` but calculates the non-symmetric difference between `needle` and `haystack` -- the number of n-grams from `needle` minus the common number of n-grams normalized by the number of `needle` n-grams. Can be useful for fuzzy string search.

For case-insensitive search or/and in UTF-8 format use functions `ngramSearchCaseInsensitive`, `ngramSearchUTF8`, `ngramSearchCaseInsensitiveUTF8`.

**Note: For UTF-8 case we use 3-gram distance. All these are not perfectly fair n-gram distances. We use 2-byte hashes to hash n-grams and then calculate the (non-)symmetric difference between these hash tables -- collisions may occur. With UTF-8 case-insensitive format we do not use fair `tolower` function -- we zero the 5-th bit (starting from zero) of each codepoint byte and first bit of zeroth byte if bytes more than one -- this works for Latin and mostly for all Cyrillic letters.**

## Functions for searching and replacing in strings

### replaceOne(haystack, pattern, replacement)

Replaces the first occurrence, if it exists, of the 'pattern' substring in 'haystack' with the 'replacement' substring.

Hereafter, 'pattern' and 'replacement' must be constants.

### replaceAll(haystack, pattern, replacement), replace(haystack, pattern, replacement)

Replaces all occurrences of the 'pattern' substring in 'haystack' with the 'replacement' substring.

### replaceRegexpOne(haystack, pattern, replacement)

Replacement using the 'pattern' regular expression. A re2 regular expression.

Replaces only the first occurrence, if it exists.

A pattern can be specified as 'replacement'. This pattern can include substitutions `\0-\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\\0') AS res
```

```
└─res
```

```
| Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello,
```

```
World!Hello, World! |
```

## replaceRegexpAll(haystack, pattern, replacement)

This does the same thing, but replaces all the occurrences. Example:

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

```
┌res┐
│ HHeellloo,, 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! |
```

## regexQuoteMeta(s)

The function adds a backslash before some predefined characters in the string.

Predefined characters: '0', '\', '|', '(', ')', '^', '\$', ':', '[', ']', '?', '\*', '+', '{', '}', '-'

This implementation slightly differs from `re2::RE2::QuoteMeta`. It escapes zero byte as `\0` instead of `\x00` and it escapes only required characters.

For more information, see the link: [RE2](#)

## Conditional functions

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

Returns `then` if `cond != 0`, or else if `cond = 0`.

cond must be of type `UInt8`, and then `and` and `else` must have the lowest common type.

then and else can be NULL

## multif

Allows you to write the **CASE** operator more compactly in the query.

```
multilf(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 multilf(isNull(y) x, y < 3, y, NULL) FROM t_null`. Result:

multilf(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  $\pi$ .

### `exp(x)`

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

### `log(x)`, `ln(x)`

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

### `exp2(x)`

Accepts a numeric argument and returns a Float64 number close to 2 to the power of  $x$ .

### `log2(x)`

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

### `exp10(x)`

Accepts a numeric argument and returns a Float64 number close to 10 to the power of  $x$ .

Accepts a numeric argument and returns a Float64 number close to 10 to the power of x.

## log10(x)

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

## sqrt(x)

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

## cbrt(x)

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

## erf(x)

If 'x' is non-negative, then  $\text{erf}(x / \sigma\sqrt{2})$  is the probability that a random variable having a normal distribution with standard deviation ' $\sigma$ ' takes the value that is separated from the expected value by more than 'x'.

Example (three sigma rule):

```
SELECT erf(3 / sqrt(2))
```

```
┌erf(divide(3, sqrt(2)))┐
└0.9973002039367398 ┘
```

## erfc(x)

Accepts a numeric argument and returns a Float64 number close to  $1 - \text{erf}(x)$ , but without loss of precision for large 'x' values.

## lgamma(x)

The logarithm of the gamma function.

## tgamma(x)

Gamma function.

## sin(x)

The sine.

## cos(x)

The cosine.

## tan(x)

The tangent.

## asin(x)

The arc sine.

## acos(x)

The arc cosine.

## atan(x)



The arc tangent.

## `pow(x, y), power(x, y)`

Takes two numeric arguments `x` and `y`. Returns a `Float64` number close to `x` to the power of `y`.

## `intExp2`

Accepts a numeric argument and returns a `UInt64` number close to 2 to the power of `x`.

## `intExp10`

Accepts a numeric argument and returns a `UInt64` number close to 10 to the power of `x`.

# Rounding functions

## `floor(x[, N])`

Returns the largest round number that is less than or equal to `x`. A round number is a multiple of  $1/10^N$ , or the nearest number of the appropriate data type if  $1 / 10^N$  isn't exact.

'`N`' is an integer constant, optional parameter. By default it is zero, which means to round to an integer.

'`N`' may be negative.

Examples: `floor(123.45, 1) = 123.4`, `floor(123.45, -1) = 120`.

`x` is any numeric type. The result is a number of the same type.

For integer arguments, it makes sense to round with a negative '`N`' value (for non-negative '`N`', the function doesn't do anything).

If rounding causes overflow (for example, `floor(-128, -1)`), an implementation-specific result is returned.

## `ceil(x[, N]), ceiling(x[, N])`

Returns the smallest round number that is greater than or equal to '`x`'. In every other way, it is the same as the '`floor`' function (see above).

## `round(x[, N])`

Rounds a value to a specified number of decimal places.

The function returns the nearest number of the specified order. In case when given number has equal distance to surrounding numbers the function returns the number having the nearest even digit (banker's rounding).

```
round(expression [, decimal_places])
```

### Parameters:

- `expression` — A number to be rounded. Can be any **expression** returning the numeric **data type**.
- `decimal-places` — An integer value.
  - If `decimal-places > 0` then the function rounds the value to the right of the decimal point.
  - If `decimal-places < 0` then the function rounds the value to the left of the decimal point.
  - If `decimal-places = 0` then the function rounds the value to integer. In this case the argument can be omitted.

### Returned value:

The rounded number of the same type as the input number.

## Examples

- . . .

## Example of use

```
SELECT number / 2 AS x, round(x) FROM system.numbers LIMIT 3
```

x	round(divide(number, 2))
0	0
0.5	0
1	1

## Examples of rounding

Rounding to the nearest number.

```
round(3.2, 0) = 3
round(4.1267, 2) = 4.13
round(22,-1) = 20
round(467,-2) = 500
round(-467,-2) = -500
```

Banker's rounding.

```
round(3.5) = 4
round(4.5) = 4
round(3.55, 1) = 3.6
round(3.65, 1) = 3.6
```

## roundToExp2(num)

Accepts a number. If the number is less than one, it returns 0. Otherwise, it rounds the number down to the nearest (whole non-negative) degree of two.

## roundDuration(num)

Accepts a number. If the number is less than one, it returns 0. Otherwise, it rounds the number down to numbers from the set: 1, 10, 30, 60, 120, 180, 240, 300, 600, 1200, 1800, 3600, 7200, 18000, 36000. This function is specific to Yandex.Metrica and used for implementing the report on session length

## roundAge(num)

Accepts a number. If the number is less than 18, it returns 0. Otherwise, it rounds the number down to a number from the set: 18, 25, 35, 45, 55. This function is specific to Yandex.Metrica and used for implementing the report on user age.

## roundDown(num, arr)

Accept a number, round it down to an element in the specified array. If the value is less than the lowest bound, the lowest bound is returned.

# Functions for working with arrays

## empty

Returns 1 for an empty array, or 0 for a non-empty array.

The result type is UInt8.

The function also works for strings.

## notEmpty

Returns 0 for an empty array, or 1 for a non-empty array.

The result type is UInt8.

The function also works for strings.

## length

Returns the number of items in the array.

The result type is UInt64.

The function also works for strings.

emptyArrayUInt8, emptyArrayUInt16, emptyArrayUInt32,  
emptyArrayUInt64

emptyArrayInt8, emptyArrayInt16, emptyArrayInt32,  
emptyArrayInt64

emptyArrayFloat32, emptyArrayFloat64

emptyArrayDate, emptyArrayDateTime

emptyArrayString

Accepts zero arguments and returns an empty array of the appropriate type.

## emptyArrayToSingle

Accepts an empty array and returns a one-element array that is equal to the default value.

## range(N)

Returns an array of numbers from 0 to N-1.

Just in case, an exception is thrown if arrays with a total length of more than 100,000,000 elements are created in a data block.

## array(x1, ...), operator [x1, ...]

Creates an array from the function arguments.

The arguments must be constants and have types that have the smallest common type. At least one argument must be passed, because otherwise it isn't clear which type of array to create. That is, you can't use this function to create an empty array (to do that, use the 'emptyArray\*' function described above).

Returns an 'Array(T)' type result, where 'T' is the smallest common type out of the passed arguments.

## arrayConcat

Combines arrays passed as arguments.

```
arrayConcat(arrays)
```

### Parameters

- arrays – Arbitrary number of arguments of **Array** type.

#### Example

```
SELECT arrayConcat([1, 2], [3, 4], [5, 6]) AS res
```

```
res
[1,2,3,4,5,6] |
```

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

Get the element with the index `n` from the array `arr`. `n` must be any integer type.

Indexes in an array begin from one.

Negative indexes are supported. In this case, it selects the corresponding element numbered from the end.

For example, `arr[-1]` is the last item in the array.

If the index falls outside of the bounds of an array, it returns some default value (0 for numbers, an empty string for strings, etc.).

## has(arr, elem)

Checks whether the 'arr' array has the 'elem' element.

Returns 0 if the the element is not in the array, or 1 if it is.

`NULL` is processed as a value.

```
SELECT has([1, 2, NULL], NULL)
```

```
┌has([1, 2, NULL], NULL)┐
└────────── 1 ─────────┘
```

## hasAll

Checks whether one array is a subset of another.

```
hasAll(set, subset)
```

### Parameters

- `set` – Array of any type with a set of elements.
- `subset` – Array of any type with elements that should be tested to be a subset of `set`.

### Return values

- 1, if `set` contains all of the elements from `subset`.
- 0, otherwise.

### Peculiar properties

- An empty array is a subset of any array.
- `Null` processed as a value.
- Order of values in both of arrays doesn't matter.

### Examples

```
SELECT hasAll([], []) returns 1.
```

```
SELECT hasAll([1, Null], [Null]) returns 1.
```

```
SELECT hasAll([1.0, 2, 3, 4], [1, 3]) returns 1.
```

```
SELECT hasAll(['a', 'b'], ['a']) returns 1.
```

```
SELECT hasAll([1, 2, 3], [4, 5]) returns 0.
```

```
SELECT hasAll([1], [a]) returns 0.
```

```
SELECT hasAll([[1, 2], [3, 4]], [[1, 2], [3, 5]]) returns 0.
```

## hasAny

Checks whether two arrays have intersection by some elements.

```
hasAny(array1, array2)
```

### Parameters

- `array1` – Array of any type with a set of elements.
- `array2` – Array of any type with a set of elements.

### Return values

- 1, if `array1` and `array2` have one similar element at least.
- 0, otherwise.

### Peculiar properties

- `Null` processed as a value.
- Order of values in both of arrays doesn't matter.

### Examples

```
SELECT hasAny([1], []) returns 0.
```

```
SELECT hasAny([Null], [Null, 1]) returns 1.
```

```
SELECT hasAny([-128, 1., 512], [1]) returns 1.
```

```
SELECT hasAny([[1, 2], [3, 4]], ['a', 'c']) returns 0.
```

```
SELECT hasAll([[1, 2], [3, 4]], [[1, 2], [1, 2]]) returns 1.
```

## indexOf(arr, x)

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

Example:

```
:) SELECT indexOf([1,3,NULL,NULL],NULL)
```

```
SELECT indexOf([1, 3, NULL, NULL], NULL)
```

```
└─indexOf([1, 3, NULL, NULL], NULL)─┐
| |
| 3 |
└──────────────────────────────────┘
```

Elements set to `NULL` are handled as normal values.

## countEqual(arr, x)

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

`NULL` elements are handled as separate values.

Example:

```
SELECT countEqual([1, 2, NULL, NULL], NULL)
```

```
┌countEqual([1, 2, NULL, NULL], NULL)┐
└──────────────────────────────────┘
2 |
```

## arrayEnumerate(arr)

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

This function is normally used with ARRAY JOIN. It allows counting something just once for each array after applying ARRAY JOIN. Example:

```
SELECT
 count() AS Reaches,
 countIf(num = 1) AS Hits
FROM test.hits
ARRAY JOIN
 GoalsReached,
 arrayEnumerate(GoalsReached) AS num
WHERE CounterID = 160656
LIMIT 10
```

```
┌Reaches┐┌Hits┐
└──95606┘└31406┘
└────────┘└──┘
```

In this example, Reaches is the number of conversions (the strings received after applying ARRAY JOIN), and Hits is the number of pageviews (strings before ARRAY JOIN). In this particular case, you can get the same result in an easier way:

```
SELECT
 sum(length(GoalsReached)) AS Reaches,
 count() AS Hits
FROM test.hits
WHERE (CounterID = 160656) AND notEmpty(GoalsReached)
```

```
┌Reaches┐┌Hits┐
└──95606┘└31406┘
└────────┘└──┘
```

This function can also be used in higher-order functions. For example, you can use it to get array indexes for elements that match a condition.

## arrayEnumerateUniq(arr, ...)

Returns an array the same size as the source array, indicating for each element what its position is among elements with the same value.

For example: `arrayEnumerateUniq([10, 20, 10, 30]) = [1, 1, 2, 1]`.

This function is useful when using ARRAY JOIN and aggregation of array elements.  
Example:

```
SELECT
 Goals.ID AS GoalID,
```

```

sum(Sign) AS Reaches,
sumIf(Sign, num = 1) AS Visits
FROM test.visits
ARRAY JOIN
 Goals,
 arrayEnumerateUniq(Goals.ID) AS num
WHERE CounterID = 160656
GROUP BY GoalID
ORDER BY Reaches DESC
LIMIT 10

```

GoalID	Reaches	Visits
53225	3214	1097
2825062	3188	1097
56600	2803	488
1989037	2401	365
2830064	2396	910
1113562	2372	373
3270895	2262	812
1084657	2262	345
56599	2260	799
3271094	2256	812

In this example, each goal ID has a calculation of the number of conversions (each element in the Goals nested data structure is a goal that was reached, which we refer to as a conversion) and the number of sessions. Without ARRAY JOIN, we would have counted the number of sessions as sum(Sign). But in this particular case, the rows were multiplied by the nested Goals structure, so in order to count each session one time after this, we apply a condition to the value of the arrayEnumerateUniq(Goals.ID) function.

The arrayEnumerateUniq function can take multiple arrays of the same size as arguments. In this case, uniqueness is considered for tuples of elements in the same positions in all the arrays.

```
SELECT arrayEnumerateUniq([1, 1, 1, 2, 2, 2], [1, 1, 2, 1, 1, 2]) AS res
```

res
[1,2,1,1,2,1]

This is necessary when using ARRAY JOIN with a nested data structure and further aggregation across multiple elements in this structure.

## arrayPopBack

Removes the last item from the array.

```
arrayPopBack(array)
```

### Parameters

- `array` – Array.

### Example

```
SELECT arrayPopBack([1, 2, 3]) AS res
```

```
┌res┐
└─┴─┘
[1,2] |
```

## arrayPopFront

Removes the first item from the array.

```
arrayPopFront(array)
```

### Parameters

- `array` – Array.

### Example

```
SELECT arrayPopFront([1, 2, 3]) AS res
```

```
┌res┐
└─┴─┘
[2,3] |
```

## arrayPushBack

Adds one item to the end of the array.

```
arrayPushBack(array, single_value)
```

### Parameters

- `array` – Array.
- `single_value` – A single value. Only numbers can be added to an array with numbers, and only strings can be added to an array of strings. When adding numbers, ClickHouse automatically sets the `single_value` type for the data type of the array. For more information about the types of data in ClickHouse, see "[Data types](#)". Can be `NULL`. The function adds a `NULL` element to an array, and the type of array elements converts to `Nullable`.

### Example

```
SELECT arrayPushBack(['a'], 'b') AS res
```

```
┌res┐
└─┴─┘
['a','b'] |
```

## arrayPushFront

Adds one element to the beginning of the array.

```
arrayPushFront(array, single_value)
```



## Parameters

- `array` – Array.
- `single_value` – A single value. Only numbers can be added to an array with numbers, and only strings can be added to an array of strings. When adding numbers, ClickHouse automatically sets the `single_value` type for the data type of the array. For more information about the types of data in ClickHouse, see "[Data types](#)". Can be `NULL`. The function adds a `NULL` element to an array, and the type of array elements converts to `Nullable`.

## Example

```
SELECT arrayPushBack(['b'], 'a') AS res
```

```
┌res┐
│ ['a','b'] │
└───┘
```

# arrayResize

Changes the length of the array.

```
arrayResize(array, size[, extender])
```

## Parameters:

- `array` — Array.
- `size` — Required length of the array.
  - If `size` is less than the original size of the array, the array is truncated from the right.
- If `size` is larger than the initial size of the array, the array is extended to the right with `extender` values or default values for the data type of the array items.
- `extender` — Value for extending an array. Can be `NULL`.

## Returned value:

An array of length `size`.

## Examples of calls

```
SELECT arrayResize([1], 3)
```

```
┌arrayResize([1], 3)┐
│ [1,0,0] │
└──────────────────┘
```

```
SELECT arrayResize([1], 3, NULL)
```

```
┌arrayResize([1], 3, NULL)┐
│ [1,NULL,NULL] │
└────────────────────────┘
```

# arraySlice

Returns a slice of the array.

```
arraySlice(array, offset[, length])
```

## Parameters

- `array` – Array of data.
- `offset` – Indent from the edge of the array. A positive value indicates an offset on the left, and a negative value is an indent on the right. Numbering of the array items begins with 1.
- `length` – The length of the required slice. If you specify a negative value, the function returns an open slice `[offset, array_length - length)`. If you omit the value, the function returns the slice `[offset, the_end_of_array]`.

## Example

```
SELECT arraySlice([1, 2, NULL, 4, 5], 2, 3) AS res
```

```
┌res┐
└─┴─┘
| [2,NULL,4] |
```

Array elements set to `NULL` are handled as normal values.

## arraySort([func,] arr, ...)

Sorts the elements of the `arr` array in ascending order. If the `func` function is specified, sorting order is determined by the result of the `func` function applied to the elements of the array. If `func` accepts multiple arguments, the `arraySort` function is passed several arrays that the arguments of `func` will correspond to. Detailed examples are shown at the end of `arraySort` description.

Example of integer values sorting:

```
SELECT arraySort([1, 3, 3, 0]);
```

```
┌arraySort([1, 3, 3, 0])┐
└─┴─┘
| [0,1,3,3] |
```

Example of string values sorting:

```
SELECT arraySort(['hello', 'world', '!']);
```

```
┌arraySort(['hello', 'world', '!'])┐
└─┴─┘
| ['!','hello','world'] |
```

Consider the following sorting order for the `NULL`, `NaN` and `Inf` values:

```
SELECT arraySort([1, nan, 2, NULL, 3, nan, -4, NULL, inf, -inf]);
```

```
┌arraySort([1, nan, 2, NULL, 3, nan, -4, NULL, inf, -inf])┐
└─┴─┘
| [-inf,-4,1,2,3,inf,nan,nan,NULL,NULL] |
```

- `-Inf` values are first in the array.
- `NULL` values are last in the array.
- `NaN` values are right before `NULL`.
- `Inf` values are right before `NaN`.

Note that `arraySort` is a **higher-order function**. You can pass a lambda function to it as the first argument. In this case, sorting order is determined by the result of the lambda function applied to the elements of the array.

Let's consider the following example:

```
SELECT arraySort((x) -> -x, [1, 2, 3]) as res;
```

```
┌res┐
│ [3,2,1] │
```

For each element of the source array, the lambda function returns the sorting key, that is, `[1 -> -1, 2 -> -2, 3 -> -3]`. Since the `arraySort` function sorts the keys in ascending order, the result is `[3, 2, 1]`. Thus, the `(x) -> -x` lambda function sets the **descending order** in a sorting.

The lambda function can accept multiple arguments. In this case, you need to pass the `arraySort` function several arrays of identical length that the arguments of lambda function will correspond to. The resulting array will consist of elements from the first input array; elements from the next input array(s) specify the sorting keys. For example:

```
SELECT arraySort((x, y) -> y, ['hello', 'world'], [2, 1]) as res;
```

```
┌res┐
│ ['world', 'hello'] │
```

Here, the elements that are passed in the second array `([2, 1])` define a sorting key for the corresponding element from the source array `(['hello', 'world'])`, that is, `['hello' -> 2, 'world' -> 1]`. Since the lambda function doesn't use `x`, actual values of the source array don't affect the order in the result. So, 'hello' will be the second element in the result, and 'world' will be the first.

Other examples are shown below.

```
SELECT arraySort((x, y) -> y, [0, 1, 2], ['c', 'b', 'a']) as res;
```

```
┌res┐
│ [2,1,0] │
```

```
SELECT arraySort((x, y) -> -y, [0, 1, 2], [1, 2, 3]) as res;
```

```
┌res┐
│ [2,1,0] │
```

```
| [2,1,0] |
```

## Note

To improve sorting efficiency, the **Schwartzian transform** is used.

## arrayReverseSort([func,] arr, ...)

Sorts the elements of the `arr` array in descending order. If the `func` function is specified, `arr` is sorted according to the result of the `func` function applied to the elements of the array, and then the sorted array is reversed. If `func` accepts multiple arguments, the `arrayReverseSort` function is passed several arrays that the arguments of `func` will correspond to. Detailed examples are shown at the end of `arrayReverseSort` description.

Example of integer values sorting:

```
SELECT arrayReverseSort([1, 3, 3, 0]);
```

```
┌arrayReverseSort([1, 3, 3, 0])┐
| [3,3,1,0] |
```

Example of string values sorting:

```
SELECT arrayReverseSort(['hello', 'world', '!']);
```

```
┌arrayReverseSort(['hello', 'world', '!'])┐
| ['world','hello','!'] |
```

Consider the following sorting order for the `NULL`, `NaN` and `Inf` values:

```
SELECT arrayReverseSort([1, nan, 2, NULL, 3, nan, -4, NULL, inf, -inf]) as res;
```

```
┌res┐
| [inf,3,2,1,-4,-inf,nan,nan,NULL,NULL] |
```

- `Inf` values are first in the array.
- `NULL` values are last in the array.
- `NaN` values are right before `NULL`.
- `-Inf` values are right before `NaN`.

Note that the `arrayReverseSort` is a **higher-order function**. You can pass a lambda function to it as the first argument. Example is shown below.

```
SELECT arrayReverseSort((x) -> -x, [1, 2, 3]) as res;
```

```
┌res┐
| [1,2,3] |
```

The array is sorted in the following way:

1. At first, the source array ([1, 2, 3]) is sorted according to the result of the lambda function applied to the elements of the array. The result is an array [3, 2, 1].
2. Array that is obtained on the previous step, is reversed. So, the final result is [1, 2, 3].

The lambda function can accept multiple arguments. In this case, you need to pass the `arrayReverseSort` function several arrays of identical length that the arguments of lambda function will correspond to. The resulting array will consist of elements from the first input array; elements from the next input array(s) specify the sorting keys. For example:

```
SELECT arrayReverseSort((x, y) -> y, ['hello', 'world'], [2, 1]) as res;
```

```
┌res┐
└─┘
┌ ['hello','world'] ─┘
```

In this example, the array is sorted in the following way:

1. At first, the source array (['hello', 'world']) is sorted according to the result of the lambda function applied to the elements of the arrays. The elements that are passed in the second array ([2, 1]), define the sorting keys for corresponding elements from the source array. The result is an array ['world', 'hello'].
2. Array that was sorted on the previous step, is reversed. So, the final result is ['hello', 'world'].

Other examples are shown below.

```
SELECT arrayReverseSort((x, y) -> y, [4, 3, 5], ['a', 'b', 'c']) AS res;
```

```
┌res┐
└─┘
┌ [5,3,4] ─┘
```

```
SELECT arrayReverseSort((x, y) -> -y, [4, 3, 5], [1, 2, 3]) AS res;
```

```
┌res┐
└─┘
┌ [4,3,5] ─┘
```

## arrayUniq(arr, ...)

If one argument is passed, it counts the number of different elements in the array.

If multiple arguments are passed, it counts the number of different tuples of elements at corresponding positions in multiple arrays.

If you want to get a list of unique items in an array, you can use `arrayReduce('groupUniqArray', arr)`.

## arrayJoin(arr)

A special function. See the section ["ArrayJoin function"](#).

## arrayDifference(arr)

Takes an array, returns an array with the difference between all pairs of neighboring elements. For example:

```
SELECT arrayDifference([1, 2, 3, 4])
```

```
┌arrayDifference([1, 2, 3, 4])┐
└ [0,1,1,1] ───────────┘
```

## arrayDistinct(arr)

Takes an array, returns an array containing the different elements in all the arrays. For example:

```
SELECT arrayDistinct([1, 2, 2, 3, 1])
```

```
┌arrayDistinct([1, 2, 2, 3, 1])┐
└ [1,2,3] ───────────┘
```

## arrayEnumerateDense(arr)

Returns an array of the same size as the source array, indicating where each element first appears in the source array. For example: `arrayEnumerateDense([10,20,10,30]) = [1,2,1,3]`.

## arrayIntersect(arr)

Takes an array, returns the intersection of all array elements. For example:

```
SELECT
 arrayIntersect([1, 2], [1, 3], [2, 3]) AS no_intersect,
 arrayIntersect([1, 2], [1, 3], [1, 4]) AS intersect
```

```
┌no_intersect┐┌intersect┐
└ [] ───┘└ [1] ───┘
```

## arrayReduce(agg\_func, arr1, ...)

Applies an aggregate function to array and returns its result. If aggregate function has multiple arguments, then this function can be applied to multiple arrays of the same size.

`arrayReduce('agg_func', arr1, ...)` - apply the aggregate function `agg_func` to arrays `arr1...`. If multiple arrays passed, then elements on corresponding positions are passed as multiple arguments to the aggregate function. For example: `SELECT arrayReduce('max', [1,2,3]) = 3`

## arrayReverse(arr)

Returns an array of the same size as the source array, containing the result of inverting all elements of the source array.

# Functions for splitting and merging strings and arrays

## splitByChar(separator, s)

Splits a string `s` into an array of strings by splitting it at each occurrence of `separator`.

Splits a string into substrings separated by 'separator'. 'separator' must be a string constant consisting of exactly one character.

Returns an array of selected substrings. Empty substrings may be selected if the separator occurs at the beginning or end of the string, or if there are multiple consecutive separators.

## splitByString(separator, s)

The same as above, but it uses a string of multiple characters as the separator. The string must be non-empty.

## arrayStringConcat(arr[, separator])

Concatenates the strings listed in the array with the separator. 'separator' is an optional parameter: a constant string, set to an empty string by default.

Returns the string.

## alphaTokens(s)

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

### Example:

```
SELECT alphaTokens('abca1abc')
```

```
└─alphaTokens('abca1abc')─┐
| ['abca','abc'] |
└────────────────────────┘
```

## Bit functions

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

The result type is an integer with bits equal to the maximum bits of its arguments. If at least one of the arguments is signed, the result is a signed number. If an argument is a floating-point number, it is cast to Int64.

bitAnd(a, b)

bitOr(a, b)

bitXor(a, b)

bitNot(a)

bitShiftLeft(a, b)

bitShiftRight(a, b)

bitRotateLeft(a, b)

bitRotateRight(a, b)

bitTest(a, b)

bitTestAll(a, b)

bitTestAny(a, b)

## Bitmap functions

Bitmap functions work for two bitmaps Object value calculation, it is to return new bitmap or cardinality while using formula calculation, such as and, or, xor, and not, etc.

There are 2 kinds of construction methods for Bitmap Object. One is to be constructed by aggregation function groupBitmap with -State, the other is to be constructed by Array Object. It is also to convert Bitmap Object to Array Object.

RoaringBitmap is wrapped into a data structure while actual storage of Bitmap objects. When the cardinality is less than or equal to 32, it uses Set objet. When the cardinality is greater than 32, it uses RoaringBitmap object. That is why storage of low cardinality set is faster.

For more information on RoaringBitmap, see: [CRoaring](#).

## bitmapBuild

Build a bitmap from unsigned integer array.

```
bitmapBuild(array)
```

### Parameters

- `array` – unsigned integer array.

### Example

```
SELECT bitmapBuild([1, 2, 3, 4, 5]) AS res
```

## bitmapToArray

Convert bitmap to integer array.

```
bitmapToArray(bitmap)
```

### Parameters

- `bitmap` – bitmap object.

### Example

```
SELECT bitmapToArray(bitmapBuild([1, 2, 3, 4, 5])) AS res
```

```
┌res┐
└─┴─┘
└─┴─┘ [1,2,3,4,5] ─┴─┘
```

## bitmapHasAny

Analogous to `hasAny(array, array)` returns 1 if bitmaps have any common elements, 0 otherwise. For empty bitmaps returns 0.

```
bitmapHasAny(bitmap,bitmap)
```

### Parameters



- `bitmap` – bitmap object.

### Example

```
SELECT bitmapHasAny(bitmapBuild([1,2,3]),bitmapBuild([3,4,5])) AS res
```

res
1

## bitmapHasAll

Analogous to `hasAll(array, array)` returns 1 if the first bitmap contains all the elements of the second one, 0 otherwise.

If the second argument is an empty bitmap then returns 1.

```
bitmapHasAll(bitmap,bitmap)
```

### Parameters

- `bitmap` – bitmap object.

### Example

```
SELECT bitmapHasAll(bitmapBuild([1,2,3]),bitmapBuild([3,4,5])) AS res
```

res
0

## bitmapAnd

Two bitmap and calculation, the result is a new bitmap.

```
bitmapAnd(bitmap,bitmap)
```

### Parameters

- `bitmap` – bitmap object.

### Example

```
SELECT bitmapToArray(bitmapAnd(bitmapBuild([1,2,3]),bitmapBuild([3,4,5]))) AS res
```

res
[3]

## bitmapOr

Two bitmap or calculation, the result is a new bitmap.

---

```
bitmapOr(bitmap,bitmap)
```

### Parameters

- `bitmap` – bitmap object.

### Example

```
SELECT bitmapToArray(bitmapOr(bitmapBuild([1,2,3]),bitmapBuild([3,4,5]))) AS res
```

```
res
| [1,2,3,4,5] |
```

## bitmapXor

Two bitmap xor calculation, the result is a new bitmap.

```
bitmapXor(bitmap,bitmap)
```

### Parameters

- `bitmap` – bitmap object.

### Example

```
SELECT bitmapToArray(bitmapXor(bitmapBuild([1,2,3]),bitmapBuild([3,4,5]))) AS res
```

```
res
| [1,2,4,5] |
```

## bitmapAndnot

Two bitmap andnot calculation, the result is a new bitmap.

```
bitmapAndnot(bitmap,bitmap)
```

### Parameters

- `bitmap` – bitmap object.

### Example

```
SELECT bitmapToArray(bitmapAndnot(bitmapBuild([1,2,3]),bitmapBuild([3,4,5]))) AS res
```

```
res
| [1,2] |
```

## bitmapCardinality

Retrun bitmap cardinality of type UInt64.

```
bitmapCardinality(bitmap)
```

### Parameters

- `bitmap` – bitmap object.

### Example

```
SELECT bitmapCardinality(bitmapBuild([1, 2, 3, 4, 5])) AS res
```

res
5

## bitmapAndCardinality

Two bitmap and calculation, return cardinality of type UInt64.

```
bitmapAndCardinality(bitmap,bitmap)
```

### Parameters

- `bitmap` – bitmap object.

### Example

```
SELECT bitmapAndCardinality(bitmapBuild([1,2,3]),bitmapBuild([3,4,5])) AS res;
```

res
1

## bitmapOrCardinality

Two bitmap or calculation, return cardinality of type UInt64.

```
bitmapOrCardinality(bitmap,bitmap)
```

### Parameters

- `bitmap` – bitmap object.

### Example

```
SELECT bitmapOrCardinality(bitmapBuild([1,2,3]),bitmapBuild([3,4,5])) AS res;
```

res
5

## bitmapXorCardinality

Two bitmap xor calculation, return cardinality of type UInt64.

```
bitmapXorCardinality(bitmap,bitmap)
```

### Parameters

- `bitmap` – bitmap object.

### Example

```
SELECT bitmapXorCardinality(bitmapBuild([1,2,3]),bitmapBuild([3,4,5])) AS res;
```

res
4

## bitmapAndnotCardinality

Two bitmap andnot calculation, return cardinality of type UInt64.

```
bitmapAndnotCardinality(bitmap,bitmap)
```

### Parameters

- `bitmap` – bitmap object.

### Example

```
SELECT bitmapAndnotCardinality(bitmapBuild([1,2,3]),bitmapBuild([3,4,5])) AS res;
```

res
2

## Hash functions

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

### halfMD5

**Interprets** all the input parameters as strings and calculates the MD5 hash value for each of them. Then combines hashes. Then from the resulting string, takes the first 8 bytes of the hash and interprets them as `UInt64` in big-endian byte order.

```
halfMD5(par1, ...)
```

The function works relatively slow (5 million short strings per second per processor core). Consider using the **sipHash64** function instead.

### Parameters

## Parameters

The function takes a variable number of input parameters. Parameters can be any of the **supported data types**.

## Returned Value

Hash value having the **UInt64** data type.

## Example

```
SELECT halfMD5(array('e','x','a'), 'mple', 10, toDateTime('2019-06-15 23:00:00')) AS halfMD5hash,
toTypeName(halfMD5hash) AS type
```

halfMD5hash	type
186182704141653334	UInt64

# MD5

Calculates the MD5 from a string and returns the resulting set of bytes as FixedString(16).

If you don't need MD5 in particular, but you need a decent cryptographic 128-bit hash, use the 'sipHash128' function instead.

If you want to get the same result as output by the md5sum utility, use lower(hex(MD5(s))).

# sipHash64

Produces 64-bit **SipHash** hash value.

```
sipHash64(par1,...)
```

This function **interprets** all the input parameters as strings and calculates the hash value for each of them. Then combines hashes.

This is a cryptographic hash function. It works at least three times faster than the **MD5** function.

## Parameters

The function takes a variable number of input parameters. Parameters can be any of the **supported data types**.

## Returned Value

Hash value having the **UInt64** data type.

## Example

```
SELECT sipHash64(array('e','x','a'), 'mple', 10, toDateTime('2019-06-15 23:00:00')) AS SipHash, toTypeName(SipHash)
AS type
```

SipHash	type
13726873534472839665	UInt64

# sipHash128

Calculates SipHash from a string

Calculates sipHash from a string.

Accepts a String-type argument. Returns FixedString(16).

Differs from sipHash64 in that the final xor-folding state is only done up to 128 bytes.

## cityHash64

Produces 64-bit hash value.

```
cityHash64(par1,...)
```

This is the fast non-cryptographic hash function. It uses **CityHash** algorithm for string parameters and implementation-specific fast non-cryptographic hash function for the parameters with other data types. To get the final result, the function uses the CityHash combinator.

### Parameters

The function takes a variable number of input parameters. Parameters can be any of the **supported data types**.

### Returned Value

Hash value having the **UInt64** data type.

### Examples

Call example:

```
SELECT cityHash64(array('e','x','a'), 'mple', 10, toDateTime('2019-06-15 23:00:00')) AS CityHash, toTypeName(CityHash) AS type
```

CityHash	type
12072650598913549138	UInt64

The following example shows how to compute the checksum of the entire table with accuracy up to the row order:

```
SELECT sum(cityHash64(*)) FROM table
```

## intHash32

Calculates a 32-bit hash code from any type of integer.

This is a relatively fast non-cryptographic hash function of average quality for numbers.

## intHash64

Calculates a 64-bit hash code from any type of integer.

It works faster than intHash32. Average quality.

## SHA1

## SHA224

## SHA256

Calculates SHA-1, SHA-224, or SHA-256 from a string and returns the resulting set of bytes as FixedString(20), FixedString(28), or FixedString(32).

The function works fairly slowly (SHA-1 processes about 5 million short strings per second on x86\_64).

The function works fairly slowly (SHA-1 processes about 5 million short strings per second per processor core, while SHA-224 and SHA-256 process about 2.2 million).

We recommend using this function only in cases when you need a specific hash function and you can't select it.

Even in these cases, we recommend applying the function offline and pre-calculating values when inserting them into the table, instead of applying it in SELECTS.

## URLHash(url[, N])

A fast, decent-quality non-cryptographic hash function for a string obtained from a URL using some type of normalization.

`URLHash(s)` – Calculates a hash from a string without one of the trailing symbols `/`, `?` or `#` at the end, if present.

`URLHash(s, N)` – Calculates a hash from a string up to the N level in the URL hierarchy, without one of the trailing symbols `/`, `?` or `#` at the end, if present.

Levels are the same as in `URLHierarchy`. This function is specific to Yandex.Metrica.

## farmHash64

Produces a 64-bit **FarmHash** hash value.

```
farmHash64(par1, ...)
```

The function uses the `Hash64` method from all **available methods**.

### Parameters

The function takes a variable number of input parameters. Parameters can be any of the **supported data types**.

### Returned Value

Hash value having the **UInt64** data type.

### Example

```
SELECT farmHash64(array('e','x','a'), 'mple', 10, toDateTime('2019-06-15 23:00:00')) AS FarmHash,
toTypeName(FarmHash) AS type
```

FarmHash type	
17790458267262532859	UInt64

## javaHash

Calculates JavaHash from a string.

Accepts a String-type argument. Returns Int32.

For more information, see the link: **JavaHash**

## hiveHash

Calculates HiveHash from a string.

Accepts a String-type argument. Returns Int32.

Same as for **JavaHash**, except that the return value never has a negative number.

## metroHash64

Produces a 64-bit **MetroHash** hash value.

Produces a 64-bit **MetroHash** hash value.

```
metroHash64(par1, ...)
```

### Parameters

The function takes a variable number of input parameters. Parameters can be any of the **supported data types**.

### Returned Value

Hash value having the **UInt64** data type.

### Example

```
SELECT metroHash64(array('e','x','a'), 'mple', 10, toDateTime('2019-06-15 23:00:00')) AS MetroHash,
toTypeName(MetroHash) AS type
```

MetroHash	type
14235658766382344533	UInt64

## jumpConsistentHash

Calculates jumpConsistentHash from a UInt64.

Accepts a UInt64-type argument. Returns Int32.

For more information, see the link: [JumpConsistentHash](#)

## murmurHash2\_32, murmurHash2\_64

Produces a **MurmurHash2** hash value.

```
murmurHash2_32(par1, ...)
murmurHash2_64(par1, ...)
```

### Parameters

Both functions take a variable number of input parameters. Parameters can be any of the **supported data types**.

### Returned Value

- The `murmurHash2_32` function returns hash value having the **UInt32** data type.
- The `murmurHash2_64` function returns hash value having the **UInt64** data type.

### Example

```
SELECT murmurHash2_64(array('e','x','a'), 'mple', 10, toDateTime('2019-06-15 23:00:00')) AS MurmurHash2,
toTypeName(MurmurHash2) AS type
```

MurmurHash2	type
11832096901709403633	UInt64

## murmurHash3 32. murmurHash3 64



Produces a **MurmurHash3** hash value.

```
murmurHash3_32(par1, ...)
murmurHash3_64(par1, ...)
```

### Parameters

Both functions take a variable number of input parameters. Parameters can be any of the **supported data types**.

### Returned Value

- The `murmurHash3_32` function returns hash value having the **UInt32** data type.
- The `murmurHash3_64` function returns hash value having the **UInt64** data type.

### Example

```
SELECT murmurHash3_32(array('e','x','a'), 'mple', 10, toDateTime('2019-06-15 23:00:00')) AS MurmurHash3,
toTypeName(MurmurHash3) AS type
```

```
┌MurmurHash3┐┌type┐
├ 2152717 │ UInt32 │
└──────────┴────────┘
```

## murmurHash3\_128

Produces a 128-bit **MurmurHash3** hash value.

```
murmurHash3_128(expr)
```

### Parameters

- `expr` — **Expressions** returning **String**-typed value.

### Returned Value

Hash value having **FixedString(16)** data type.

### Example

```
SELECT murmurHash3_128('example_string') AS MurmurHash3, toTypeName(MurmurHash3) AS type
```

```
┌MurmurHash3┐┌type┐
├ 614"S5KT~~q │ FixedString(16) │
└──────────┴────────┘
```

## xxHash32, xxHash64

Calculates xxHash from a string.

Accepts a String-type argument. Returns UInt64 Or UInt32.

For more information, see the link: **xxHash**

## Functions for generating pseudo-random numbers

Non-cryptographic generators of pseudo-random numbers are used.

All the functions accept zero arguments or one argument.

If an argument is passed, it can be any type, and its value is not used for anything.

The only purpose of this argument is to prevent common subexpression elimination, so that two different instances of the same function return different columns with different random numbers.

## rand

Returns a pseudo-random UInt32 number, evenly distributed among all UInt32-type numbers.

Uses a linear congruential generator.

## rand64

Returns a pseudo-random UInt64 number, evenly distributed among all UInt64-type numbers.

Uses a linear congruential generator.

## randConstant

Returns a pseudo-random UInt32 number, The value is one for different blocks.

# Encoding functions

## hex

Accepts arguments of types: `String`, `unsigned integer`, `Date`, or `DateTime`. Returns a string containing the argument's hexadecimal representation. Uses uppercase letters A-F. Does not use `0x` prefixes or `h` suffixes. For strings, all bytes are simply encoded as two hexadecimal numbers. Numbers are converted to big endian ("human readable") format. For numbers, older zeros are trimmed, but only by entire bytes. For example, `hex(1) = '01'`. `Date` is encoded as the number of days since the beginning of the Unix epoch. `DateTime` is encoded as the number of seconds since the beginning of the Unix epoch.

## unhex(str)

Accepts a string containing any number of hexadecimal digits, and returns a string containing the corresponding bytes. Supports both uppercase and lowercase letters A-F. The number of hexadecimal digits does not have to be even. If it is odd, the last digit is interpreted as the younger half of the 00-0F byte. If the argument string contains anything other than hexadecimal digits, some implementation-defined result is returned (an exception isn't thrown).

If you want to convert the result to a number, you can use the `'reverse'` and `'reinterpretAsType'` functions.

## UUIDStringToNum(str)

Accepts a string containing 36 characters in the format `123e4567-e89b-12d3-a456-426655440000`, and returns it as a set of bytes in a `FixedString(16)`.

## UUIDNumToString(str)

Accepts a `FixedString(16)` value. Returns a string containing 36 characters in text format.

## bitmaskToList(num)

Accepts an integer. Returns a string containing the list of powers of two that total the source number when summed. They are comma-separated without spaces in text format, in ascending order.

## bitmaskToArray(num)

Accepts an integer. Returns an array of UInt64 numbers containing the list of powers of two that total the source number when summed. Numbers in the array are in ascending order.

## Functions for working with UUID

# FUNCTIONS FOR WORKING WITH UUID

The functions for working with UUID are listed below.

## generateUUIDv4

Generates the **UUID** of **version 4**.

```
generateUUIDv4()
```

### Returned value

The UUID type value.

### Usage example

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

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

:) INSERT INTO t_uuid SELECT generateUUIDv4()

:) SELECT * FROM t_uuid
```

x
f4bf890f-f9dc-4332-ad5c-0c18e73f28e9

## toUUID (x)

Converts String type value to UUID type.

```
toUUID(String)
```

### Returned value

The UUID type value.

### Usage example

```
:) SELECT toUUID('61f0c404-5cb3-11e7-907b-a6006ad3dba0') AS uuid
```

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

## UUIDStringToNum

Accepts a string containing 36 characters in the format `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx`, and returns it as a set of bytes in a **FixedString(16)**.

```
UUIDStringToNum(String)
```

### Returned value

FixedString(16)

## Usage examples

```
:) SELECT
 '612f3c40-5d3b-217e-707b-6a546a3d7b29' AS uuid,
 UUIDStringToNum(uuid) AS bytes
```

uuid	bytes
612f3c40-5d3b-217e-707b-6a546a3d7b29	a/<@];!~p{jTj={}

## UUIDNumToString

Accepts a [FixedString\(16\)](#) value, and returns a string containing 36 characters in text format.

```
UUIDNumToString(FixedString(16))
```

### Returned value

String.

### Usage example

```
SELECT
 'a/<@];!~p{jTj={}' AS bytes,
 UUIDNumToString(toFixedString(bytes, 16)) AS uuid
```

bytes	uuid
a/<@];!~p{jTj={}	612f3c40-5d3b-217e-707b-6a546a3d7b29

## See also

- [dictGetUUID](#)
- [dictGetUUIDOrDefault](#)

## Functions for working with URLs

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

### Functions that extract part of a URL

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

#### protocol

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

#### domain

Gets the domain.

#### domainWithoutWWW

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

#### topLevelDomain

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

#### firstSignificantSubdomain

Returns the "first significant subdomain". This is a non-standard concept specific to Yandex.Metrica. The first significant subdomain is a second-level domain if it is 'com', 'net', 'org', or 'co'. Otherwise, it is a third-level domain. For example, `firstSignificantSubdomain('https://news.yandex.ru/') = 'yandex '`, `firstSignificantSubdomain('https://news.yandex.com.tr/') = 'yandex '`. The list of "insignificant" second-level domains and other implementation details may change in the future.

## cutToFirstSignificantSubdomain

Returns the part of the domain that includes top-level subdomains up to the "first significant subdomain" (see the explanation above).

For example, `cutToFirstSignificantSubdomain('https://news.yandex.com.tr/') = 'yandex.com.tr'`.

## path

Returns the path. Example: `/top/news.html` The path does not include the query string.

## pathFull

The same as above, but including query string and fragment. Example: `/top/news.html?page=2#comments`

## queryString

Returns the query string. Example: `page=1&lr=213`. query-string does not include the initial question mark, as well as `#` and everything after `#`.

## fragment

Returns the fragment identifier. fragment does not include the initial hash symbol.

## queryStringAndFragment

Returns the query string and fragment identifier. Example: `page=1#29390`.

## extractURLParameter(URL, name)

Returns the value of the 'name' parameter in the URL, if present. Otherwise, an empty string. If there are many parameters with this name, it returns the first occurrence. This function works under the assumption that the parameter name is encoded in the URL exactly the same way as in the passed argument.

## extractURLParameters(URL)

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

## extractURLParameterNames(URL)

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

## URLHierarchy(URL)

Returns an array containing the URL, truncated at the end by the symbols `/,?` in the path and query-string. Consecutive separator characters are counted as one. The cut is made in the position after all the consecutive separator characters. Example:

## URLPathHierarchy(URL)

The same as above, but without the protocol and host in the result. The `/` element (root) is not included. Example: the function is used to implement tree reports the URL in Yandex. Metric.

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

```
['/browse/CONV-6788']
```

## decodeURIComponent(URL)

Returns the decoded URL.

Example:

```
SELECT decodeURIComponent('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('2A0206B80000000000000000000011'), 16)) AS addr
```

addr
2a02:6b8::11

```
SELECT
 IPv6NumToString(ClientIP6 AS k),
 count() AS c
FROM hits_all
WHERE EventDate = today() AND substring(ClientIP6, 1, 12) != unhex('00000000000000000000FFFF')
GROUP BY k
ORDER BY c DESC
LIMIT 10
```

IPv6NumToString(ClientIP6)	c
2a02:2168:aaa:bbbb::2	24695
2a02:2698:abcd:abcd:abcd:8888:5555	22408
2a02:6b8:0:fff::ff	16389
2a01:4f8:111:6666::2	16016
2a02:2168:888:222::1	15896
2a01:7e00::ffff:ffff:ffff:222	14774
2a02:8109:eee:ee:eeee:eeee:eeee:eeee	14443
2a02:810b:8888:888:8888:8888:8888:8888	14345
2a02:6b8:0:444:4444:4444:4444:4444	14279
2a01:7e00::ffff:ffff:ffff:ffff	13880

```

SELECT
 IPv6NumToString(ClientIP6 AS k),
 count() AS c
FROM hits_all
WHERE EventDate = today()
GROUP BY k
ORDER BY c DESC
LIMIT 10

```

IPv6NumToString(ClientIP6)	c
::ffff:94.26.111.111	747440
::ffff:37.143.222.4	529483
::ffff:5.166.111.99	317707
::ffff:46.38.11.77	263086
::ffff:79.105.111.111	186611
::ffff:93.92.111.88	176773
::ffff:84.53.111.33	158709
::ffff:217.118.11.22	154004
::ffff:217.118.11.33	148449
::ffff:217.118.11.44	148243

## IPv6StringToNum(s)

The reverse function of IPv6NumToString. If the IPv6 address has an invalid format, it returns a string of null bytes.

HEX can be uppercase or lowercase.

## IPv4ToIPv6(x)

Takes a UInt32 number. Interprets it as an IPv4 address in big endian. Returns a FixedString(16) value containing the IPv6 address in binary format. Examples:

```

SELECT IPv6NumToString(IPv4ToIPv6(IPv4StringToNum('192.168.0.1'))) AS addr

```

addr
::ffff:192.168.0.1

## cutIPv6(x, bitsToCutForIPv6, bitsToCutForIPv4)

Accepts a FixedString(16) value containing the IPv6 address in binary format. Returns a string containing the address of the specified number of bits removed in text format. For example:

```

WITH
 IPv6StringToNum('2001:0DB8:AC10:FE01:FEED:BABE:CAFE:F00D') AS ipv6,
 IPv4ToIPv6(IPv4StringToNum('192.168.0.1')) AS ipv4
SELECT
 cutIPv6(ipv6, 2, 0),
 cutIPv6(ipv4, 0, 2)

```

cutIPv6(ipv6, 2, 0)	cutIPv6(ipv4, 0, 2)
2001:db8:ac10:fe01:feed:babe:cafe:0	::ffff:192.168.0.0



## IPv4CIDRtoIPv4Range(ipv4, cidr),

Accepts an IPv4 and an UInt8 value containing the CIDR. Return a tuple with two IPv4 containing the lower range and the higher range of the subnet.

```
SELECT IPv4CIDRtoIPv4Range(toIPv4('192.168.5.2'), 16)
```

```
┌IPv4CIDRtoIPv4Range(toIPv4('192.168.5.2'), 16)┐
├ ('192.168.0.0','192.168.255.255') │
└──┘
```

## IPv6CIDRtoIPv6Range(ipv6, cidr),

Accepts an IPv6 and an UInt8 value containing the CIDR. Return a tuple with two IPv6 containing the lower range and the higher range of the subnet.

```
SELECT IPv6CIDRtoIPv6Range(toIPv6('2001:0db8:0000:85a3:0000:0000:ac1f:8001'), 32);
```

```
┌IPv6CIDRtoIPv6Range(toIPv6('2001:0db8:0000:85a3:0000:0000:ac1f:8001'), 32)┐
├ ('2001:db8::','2001:db8:ffff:ffff:ffff:ffff:ffff:ffff') │
└──┘
```

## toIPv4(string)

An alias to `IPv4StringToNum()` that takes a string form of IPv4 address and returns value of **IPv4** type, which is binary equal to value returned by `IPv4StringToNum()`.

```
WITH
 '171.225.130.45' as IPv4_string
SELECT
 toTypeName(IPv4StringToNum(IPv4_string)),
 toTypeName(toIPv4(IPv4_string))
```

```
┌toTypeName(IPv4StringToNum(IPv4_string))┐┌toTypeName(toIPv4(IPv4_string))┐
├ UInt32 │ IPv4 │
└──┘└────────────────────────────────┘
```

```
WITH
 '171.225.130.45' as IPv4_string
SELECT
 hex(IPv4StringToNum(IPv4_string)),
 hex(toIPv4(IPv4_string))
```

```
┌hex(IPv4StringToNum(IPv4_string))┐┌hex(toIPv4(IPv4_string))┐
├ ABE1822D │ ABE1822D │
└──────────────────────────────────┘└──────────────────────────┘
```

## toIPv6(string)

An alias to `IPv6StringToNum()` that takes a string form of IPv6 address and returns value of **IPv6** type, which is binary equal to value returned by `IPv6StringToNum()`.

```
WITH
 '2001:438:ffff::407d:1bc1' as IPv6_string
SELECT
 toTypeName(IPv6StringToNum(IPv6_string)),
 toTypeName(toIPv6(IPv6_string))
```

```
┌toTypeName(IPv6StringToNum(IPv6_string))┐toTypeName(toIPv6(IPv6_string))┐
├ FixedString(16) │ IPv6 │
└──────────────────────────────────┴──────────────────────────────────┘
```

```
WITH
 '2001:438:ffff::407d:1bc1' as IPv6_string
SELECT
 hex(IPv6StringToNum(IPv6_string)),
 hex(toIPv6(IPv6_string))
```

```
┌hex(IPv6StringToNum(IPv6_string))┐hex(toIPv6(IPv6_string))┐
├ 20010438FFFF000000000000407D1BC1 │ 20010438FFFF000000000000407D1BC1 │
└──────────────────────────────────┴──────────────────────────────────┘
```

## 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)

`visitParamExtractRaw(params, name)`

Parses a true/false value. The result is UInt8.

## visitParamExtractRaw(params, name)

Returns the value of a field, including separators.

Examples:

```
visitParamExtractRaw('{ "abc": "\n\u0000" }', 'abc') = '"\n\u0000"'
visitParamExtractRaw('{ "abc": { "def": [1,2,3] } }', 'abc') = '{ "def": [1,2,3] }'
```

## visitParamExtractString(params, name)

Parses the string in double quotes. The value is unescaped. If unescaping failed, it returns an empty string.

Examples:

```
visitParamExtractString('{ "abc": "\n\u0000" }', 'abc') = '\n\0'
visitParamExtractString('{ "abc": "\u263a" }', 'abc') = '☺'
visitParamExtractString('{ "abc": "\u263" }', 'abc') = ''
visitParamExtractString('{ "abc": "hello" }', 'abc') = ''
```

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

The following functions are based on [simdjson](#) designed for more complex JSON parsing requirements. The assumption 2 mentioned above still applies.

## JSONHas(json[, indices\_or\_keys]...)

If the value exists in the JSON document, `1` will be returned.

If the value does not exist, `0` will be returned.

Examples:

```
select JSONHas('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b') = 1
select JSONHas('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b', 4) = 0
```

`indices_or_keys` is a list of zero or more arguments each of them can be either string or integer.

- String = access object member by key.
- Positive integer = access the n-th member/key from the beginning.
- Negative integer = access the n-th member/key from the end.

You may use integers to access both JSON arrays and JSON objects.

So, for example:

```
select JSONExtractKey('{ "a": "hello", "b": [-100, 200.0, 300] }', 1) = 'a'
select JSONExtractKey('{ "a": "hello", "b": [-100, 200.0, 300] }', 2) = 'b'
select JSONExtractKey('{ "a": "hello", "b": [-100, 200.0, 300] }', -1) = 'b'
select JSONExtractKey('{ "a": "hello", "b": [-100, 200.0, 300] }', -2) = 'a'
select JSONExtractString('{ "a": "hello", "b": [-100, 200.0, 300] }', 1) = 'hello'
```

## JSONLength(json[, indices\_or\_keys]...)

Return the length of a JSON array or a JSON object.

If the value does not exist or has a wrong type, `0` will be returned.

Examples:

```
select JSONLength('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b') = 3
select JSONLength('{ "a": "hello", "b": [-100, 200.0, 300] }') = 2
```

## JSONType(json[, indices\_or\_keys]...)

Return the type of a JSON value.

If the value does not exist, `Null` will be returned.

Examples:

```
select JSONType('{ "a": "hello", "b": [-100, 200.0, 300] }') = 'Object'
select JSONType('{ "a": "hello", "b": [-100, 200.0, 300] }', 'a') = 'String'
select JSONType('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b') = 'Array'
```

## JSONExtractUInt(json[, indices\_or\_keys]...)

## JSONExtractInt(json[, indices\_or\_keys]...)

## JSONExtractFloat(json[, indices\_or\_keys]...)

## JSONExtractBool(json[, indices\_or\_keys]...)

Parses a JSON and extract a value. These functions are similar to `visitParam` functions.

If the value does not exist or has a wrong type, `0` will be returned.

Examples:

```
select JSONExtractInt('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b', 1) = -100
select JSONExtractFloat('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b', 2) = 200.0
select JSONExtractUInt('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b', -1) = 300
```

## JSONExtractString(json[, indices\_or\_keys]...)

Parses a JSON and extract a string. This function is similar to `visitParamExtractString` functions.

If the value does not exist or has a wrong type, an empty string will be returned.

The value is unescaped. If unescaping failed, it returns an empty string.

Examples:

```
select JSONExtractString('{ "a": "hello", "b": [-100, 200.0, 300] }', 'a') = 'hello'
select JSONExtractString('{ "abc": "\n\u0000" }', 'abc') = '\n\u0000'
select JSONExtractString('{ "abc": "\u263a" }', 'abc') = '☺'
select JSONExtractString('{ "abc": "\u263" }', 'abc') = ''
select JSONExtractString('{ "abc": "hello" }', 'abc') = ''
```

## JSONExtract(json[, indices\_or\_keys...], return\_type)

Parses a JSON and extract a value of the given ClickHouse data type.

This is a generalization of the previous `JSONExtract<type>` functions.

This means

`JSONExtract(..., 'String')` returns exactly the same as `JSONExtractString()`,

`JSONExtract(..., 'Float64')` returns exactly the same as `JSONExtractFloat()`.

Examples:

```
SELECT JSONExtract('{ "a": "hello", "b": [-100, 200.0, 300] }', 'Tuple(String, Array(Float64))') = ('hello', [-100, 200, 300])
SELECT JSONExtract('{ "a": "hello", "b": [-100, 200.0, 300] }', 'Tuple(b Array(Float64), a String)') = ([-100, 200, 300], 'hello')
SELECT JSONExtract('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b', 'Array(Nullable(Int8))') = [-100, NULL, NULL]
SELECT JSONExtract('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b', 4, 'Nullable(Int64)') = NULL
SELECT JSONExtract('{ "passed": true }', 'passed', 'UInt8') = 1
SELECT JSONExtract('{ "day": "Thursday" }', 'day', 'Enum8(\'Sunday\' = 0, \'Monday\' = 1, \'Tuesday\' = 2, \'Wednesday\' = 3, \'Thursday\' = 4, \'Friday\' = 5, \'Saturday\' = 6)') = 'Thursday'
SELECT JSONExtract('{ "day": 5 }', 'day', 'Enum8(\'Sunday\' = 0, \'Monday\' = 1, \'Tuesday\' = 2, \'Wednesday\' = 3, \'Thursday\' = 4, \'Friday\' = 5, \'Saturday\' = 6)') = 'Friday'
```

## JSONExtractKeysAndValues(json[, indices\_or\_keys...], value\_type)

Parse key-value pairs from a JSON where the values are of the given ClickHouse data type.

Example:

```
SELECT JSONExtractKeysAndValues('{ "x": { "a": 5, "b": 7, "c": 11 } }', 'x', 'Int8') = [(('a',5),('b',7),('c',11))];
```

## JSONExtractRaw(json[, indices\_or\_keys]...)

Returns a part of JSON.

If the part does not exist or has a wrong type, an empty string will be returned.

Example:

```
select JSONExtractRaw('{ "a": "hello", "b": [-100, 200.0, 300] }', 'b') = '[-100, 200.0, 300]'
```

## Higher-order functions

### -> operator, lambda(params, expr) function

Allows describing a lambda function for passing to a higher-order function. The left side of the arrow has a formal parameter, which is any ID, or multiple formal parameters – any IDs in a tuple. The right side of the arrow has an expression that can use these formal parameters, as well as any table columns.

Examples: `x -> 2 * x`, `str -> str != Referer`.

Higher-order functions can only accept lambda functions as their functional argument.

A lambda function that accepts multiple arguments can be passed to a higher-order function. In this case, the higher-order function is passed several arrays of identical length that these arguments will correspond to.

For all functions other than 'arrayMap' and 'arrayFilter', the first argument (the lambda function) can be omitted. In this case, identical mapping is assumed.

`arrayMap(func arr1 ...)`

## arrayMap(func, arr1, ...)

Returns an array obtained from the original application of the 'func' function to each element in the 'arr' array.

## arrayFilter(func, arr1, ...)

Returns an array containing only the elements in 'arr1' for which 'func' returns something other than 0.

Examples:

```
SELECT arrayFilter(x -> x LIKE '%World%', ['Hello', 'abc World']) AS res
```

```
┌res┐
└─┴─┘
| ['abc World'] |
└──────────┘
```

```
SELECT
 arrayFilter(
 (i, x) -> x LIKE '%World%',
 arrayEnumerate(arr),
 ['Hello', 'abc World'] AS arr)
 AS res
```

```
┌res┐
└─┴─┘
| [2] |
└──────────┘
```

## arrayCount([func,] arr1, ...)

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

## arrayExists([func,] arr1, ...)

Returns 1 if there is at least one element in 'arr' for which 'func' returns something other than 0. Otherwise, it returns 0.

## arrayAll([func,] arr1, ...)

Returns 1 if 'func' returns something other than 0 for all the elements in 'arr'. Otherwise, it returns 0.

## arraySum([func,] arr1, ...)

Returns the sum of the 'func' values. If the function is omitted, it just returns the sum of the array elements.

## arrayFirst(func, arr1, ...)

Returns the first element in the 'arr1' array for which 'func' returns something other than 0.

## arrayFirstIndex(func, arr1, ...)

Returns the index of the first element in the 'arr1' array for which 'func' returns something other than 0.

## arrayCumSum([func,] arr1, ...)

Returns an array of partial sums of elements in the source array (a running sum). If the func function is specified, then the values of the array elements are converted by this function before summing.

Example:

```
SELECT arrayCumSum([1, 1, 1, 1]) AS res
```

```
┌res┐
└─┘
[1, 2, 3, 4] |
```

## arrayCumSumNonNegative(arr)

Same as arrayCumSum, returns an array of partial sums of elements in the source array (a running sum). Different arrayCumSum, when then returned value contains a value less than zero, the value is replace with zero and the subsequent calculation is performed with zero parameters. For example:

```
SELECT arrayCumSumNonNegative([1, 1, -4, 1]) AS res
```

```
┌res┐
└─┘
[1,2,0,1] |
```

## arraySort([func,] arr1, ...)

Returns an array as result of sorting the elements of arr1 in ascending order. If the func function is specified, sorting order is determined by the result of the function func applied to the elements of array (arrays)

The **Schwartzian transform** is used to improve sorting efficiency.

Example:

```
SELECT arraySort((x, y) -> y, ['hello', 'world'], [2, 1]);
```

```
┌res┐
└─┘
['world', 'hello'] |
```

Note that NULLs and NaNs go last (NaNs go before NULLs). For example:

```
SELECT arraySort([1, nan, 2, NULL, 3, nan, 4, NULL])
```

```
┌arraySort([1, nan, 2, NULL, 3, nan, 4, NULL])┐
└─┘
[1,2,3,4,nan,nan,NULL,NULL] |
```

## arrayReverseSort([func,] arr1, ...)

Returns an array as result of sorting the elements of arr1 in descending order. If the func function is specified, sorting order is determined by the result of the function func applied to the elements of array (arrays)

Note that NULLs and NaNs go last (NaNs go before NULLs). For example:

```
SELECT arrayReverseSort([1, nan, 2, NULL, 3, nan, 4, NULL])
```

```
arrayReverseSort([1, nan, 2, NULL, 3, nan, 4, NULL])
[4,3,2,1,nan,nan,NULL,NULL]
```

## Functions for working with external dictionaries

For information on connecting and configuring external dictionaries, see [External dictionaries](#).

dictGetUInt8, dictGetUInt16, dictGetUInt32, dictGetUInt64

dictGetInt8, dictGetInt16, dictGetInt32, dictGetInt64

dictGetFloat32, dictGetFloat64

dictGetDate, dictGetDateTime

dictGetUUID

dictGetString

dictGetT('dict\_name', 'attr\_name', id)

- Get the value of the attr\_name attribute from the dict\_name dictionary using the 'id' key. dict\_name and attr\_name are constant strings. id must be UInt64.  
If there is no id key in the dictionary, it returns the default value specified in the dictionary description.

### dictGetTOrDefault

dictGetTOrDefault('dict\_name', 'attr\_name', id, default)

The same as the dictGetT functions, but the default value is taken from the function's last argument.

### dictIsIn

dictIsIn ('dict\_name', child\_id, ancestor\_id)

- For the 'dict\_name' hierarchical dictionary, finds out whether the 'child\_id' key is located inside 'ancestor\_id' (or matches 'ancestor\_id'). Returns UInt8.

### dictGetHierarchy

dictGetHierarchy('dict\_name', id)

- For the 'dict\_name' hierarchical dictionary, returns an array of dictionary keys starting from 'id' and continuing along the chain of parent elements. Returns Array(UInt64).

### dictHas

dictHas('dict\_name', id)

- Check whether the dictionary has the key. Returns a UInt8 value equal to 0 if there is no key and 1 if there is a key.

## Functions for working with Yandex.Metrica dictionaries

In order for the functions below to work, the server config must specify the paths and addresses for getting all the Yandex.Metrica dictionaries. The dictionaries are loaded at the first call of any of these functions. If the reference lists can't be loaded, an exception is thrown.

For information about creating reference lists, see the section "Dictionaries".



# Multiple geobases

ClickHouse supports working with multiple alternative geobases (regional hierarchies) simultaneously, in order to support various perspectives on which countries certain regions belong to.

The 'clickhouse-server' config specifies the file with the regional

hierarchy:: `<path_to_regions_hierarchy_file>/opt/geo/regions_hierarchy.txt</path_to_regions_hierarchy_file>`

Besides this file, it also searches for files nearby that have the `_` symbol and any suffix appended to the name (before the file extension).

For example, it will also find the file `/opt/geo/regions_hierarchy_ua.txt`, if present.

`ua` is called the dictionary key. For a dictionary without a suffix, the key is an empty string.

All the dictionaries are re-loaded in runtime (once every certain number of seconds, as defined in the `builtin_dictionaries_reload_interval` config parameter, or once an hour by default). However, the list of available dictionaries is defined one time, when the server starts.

All functions for working with regions have an optional argument at the end – the dictionary key. It is referred to as the geobase.

Example:

```
regionToCountry(RegionID) - Uses the default dictionary: /opt/geo/regions_hierarchy.txt
regionToCountry(RegionID, '') - Uses the default dictionary: /opt/geo/regions_hierarchy.txt
regionToCountry(RegionID, 'ua') - Uses the dictionary for the 'ua' key: /opt/geo/regions_hierarchy_ua.txt
```

## regionToCity(id[, geobase])

Accepts a UInt32 number – the region ID from the Yandex geobase. If this region is a city or part of a city, it returns the region ID for the appropriate city. Otherwise, returns 0.

## regionToArea(id[, geobase])

Converts a region to an area (type 5 in the geobase). In every other way, this function is the same as 'regionToCity'.

```
SELECT DISTINCT regionToName(regionToArea(toUInt32(number), 'ua'))
FROM system.numbers
LIMIT 15
```

```
┌regionToName(regionToArea(toUInt32(number), \'ua\'))┐
├──┤
│ Moscow and Moscow region │
│ St. Petersburg and Leningrad region │
│ Belgorod region │
│ Ivanovsk region │
│ Kaluga region │
│ Kostroma region │
│ Kursk region │
│ Lipetsk region │
│ Orlov region │
│ Ryazan region │
│ Smolensk region │
│ Tambov region │
│ Tver region │
│ Tula region │
└──┘
```

## regionToDistrict(id[, geobase])

Converts a region to a federal district (type 4 in the geobase). In every other way, this function is the same as 'regionToCity'.

```
SELECT DISTINCT regionToName(regionToDistrict(toUInt32(number), 'ua'))
FROM system.numbers
LIMIT 15
```

```
┌regionToName(regionToDistrict(toUInt32(number), \'ua\'))┐
├──┤
│ Central federal district │
│ Northwest federal district │
│ South federal district │
│ North Caucasus federal district │
│ Privolga federal district │
│ Ural federal district │
│ Siberian federal district │
│ Far East federal district │
│ Scotland │
│ Faroe Islands │
│ Flemish region │
│ Brussels capital region │
│ Wallonia │
│ Federation of Bosnia and Herzegovina │
└──┘
```

## regionToCountry(id[, geobase])

Converts a region to a country. In every other way, this function is the same as 'regionToCity'.

Example: `regionToCountry(toUInt32(213)) = 225` converts Moscow (213) to Russia (225).

## regionToContinent(id[, geobase])

Converts a region to a continent. In every other way, this function is the same as 'regionToCity'.

Example: `regionToContinent(toUInt32(213)) = 10001` converts Moscow (213) to Eurasia (10001).

## regionToPopulation(id[, geobase])

Gets the population for a region.

The population can be recorded in files with the geobase. See the section "External dictionaries".

If the population is not recorded for the region, it returns 0.

In the Yandex geobase, the population might be recorded for child regions, but not for parent regions.

## regionIn(lhs, rhs[, geobase])

Checks whether a 'lhs' region belongs to a 'rhs' region. Returns a UInt8 number equal to 1 if it belongs, or 0 if it doesn't belong.

The relationship is reflexive – any region also belongs to itself.

## regionHierarchy(id[, geobase])

Accepts a UInt32 number – the region ID from the Yandex geobase. Returns an array of region IDs consisting of the passed region and all parents along the chain.

Example: `regionHierarchy(toUInt32(213)) = [213,1,3,225,10001,10000]`.

## regionToName(id[, lang])

Accepts a UInt32 number – the region ID from the Yandex geobase. A string with the name of the language can be passed as a second argument. Supported languages are: ru, en, ua, uk, by, kz, tr. If the second

argument is omitted, the language 'ru' is used. If the language is not supported, an exception is thrown. Returns a string – the name of the region in the corresponding language. If the region with the specified ID doesn't exist, an empty string is returned.

`ua` and `uk` both mean Ukrainian.

## Functions for implementing the IN operator `in`, `notIn`, `globalIn`, `globalNotIn`

See the section [IN operators](#).

### `tuple(x, y, ...)`, `operator (x, y, ...)`

A function that allows grouping multiple columns.

For columns with the types `T1`, `T2`, ..., it returns a `Tuple(T1, T2, ...)` type tuple containing these columns. There is no cost to execute the function.

Tuples are normally used as intermediate values for an argument of IN operators, or for creating a list of formal parameters of lambda functions. Tuples can't be written to a table.

### `tupleElement(tuple, n)`, `operator x.N`

A function that allows getting a column from a tuple.

'N' is the column index, starting from 1. N must be a constant. 'N' must be a constant. 'N' must be a strict positive integer no greater than the size of the tuple.

There is no cost to execute the function.

## arrayJoin function

This is a very unusual function.

Normal functions don't change a set of rows, but just change the values in each row (map).

Aggregate functions compress a set of rows (fold or reduce).

The 'arrayJoin' function takes each row and generates a set of rows (unfold).

This function takes an array as an argument, and propagates the source row to multiple rows for the number of elements in the array.

All the values in columns are simply copied, except the values in the column where this function is applied; it is replaced with the corresponding array value.

A query can use multiple `arrayJoin` functions. In this case, the transformation is performed multiple times.

Note the ARRAY JOIN syntax in the SELECT query, which provides broader possibilities.

Example:

```
SELECT arrayJoin([1, 2, 3] AS src) AS dst, 'Hello', src
```

	dst	'Hello'	src
1	Hello	[1,2,3]	
2	Hello	[1,2,3]	
3	Hello	[1,2,3]	

## Functions for working with geographical coordinates `greatCircleDistance`

Calculate the distance between two points on the Earth's surface using [the great-circle formula](#).

```
greatCircleDistance(lon1Deg, lat1Deg, lon2Deg, lat2Deg)
```

### Input parameters

- `lon1Deg` — Longitude of the first point in degrees. Range: `[-180°, 180°]`.
- `lat1Deg` — Latitude of the first point in degrees. Range: `[-90°, 90°]`.
- `lon2Deg` — Longitude of the second point in degrees. Range: `[-180°, 180°]`.
- `lat2Deg` — Latitude of the second point in degrees. Range: `[-90°, 90°]`.

Positive values correspond to North latitude and East longitude, and negative values correspond to South latitude and West longitude.

### Returned value

The distance between two points on the Earth's surface, in meters.

Generates an exception when the input parameter values fall outside of the range.

### Example

```
SELECT greatCircleDistance(55.755831, 37.617673, -55.755831, -37.617673)
```

```
└─greatCircleDistance(55.755831, 37.617673, -55.755831, -37.617673)─┐
└─14132374.194975413 ─┘
```

## pointInEllipses

Checks whether the point belongs to at least one of the ellipses.

```
pointInEllipses(x, y, x0, y0, a0, b0,...,xn, yn, an, bn)
```

### Input parameters

- `x, y` — Coordinates of a point on the plane.
- `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

## pointInPolygon

Checks whether the point belongs to the polygon on the plane.

```
pointInPolygon((x, y), [(a, b), (c, d) ...], ...)
```

### Input values

- `(x, y)` — Coordinates of a point on the plane. Data type — **Tuple** — A tuple of two numbers.
- `[(a, b), (c, d) ...]` — Polygon vertices. Data type — **Array**. Each vertex is represented by a pair of coordinates `(a, b)`. Vertices should be specified in a clockwise or counterclockwise order. The minimum number of vertices is 3. The polygon must be constant.
- The function also supports polygons with holes (cut out sections). In this case, add polygons that define the cut out sections using additional arguments of the function. The function does not support non-simply-connected polygons.

### Returned values

`1` if the point is inside the polygon, `0` if it is not.

If the point is on the polygon boundary, the function may return either 0 or 1.

### Example

```
SELECT pointInPolygon((3., 3.), [(6, 0), (8, 4), (5, 8), (0, 2)]) AS res
```

```
┌res┐
│ 1 │
```

## geohashEncode

Encodes latitude and longitude as a geohash-string, please see (<http://geohash.org/>, <https://en.wikipedia.org/wiki/Geohash>).

```
geohashEncode(longitude, latitude, [precision])
```

### Input values

- longitude - longitude part of the coordinate you want to encode. Floating in range `[-180°, 180°]`
- latitude - latitude part of the coordinate you want to encode. Floating in range `[-90°, 90°]`
- precision - Optional, length of the resulting encoded string, defaults to `12`. Integer in range `[1, 12]`. Any value less than `1` or greater than `12` is silently converted to `12`.

### Returned values

- alphanumeric **String** of encoded coordinate (modified version of the base32-encoding alphabet is used).

### Example

```
SELECT geohashEncode(-5.60302734375, 42.593994140625, 0) AS res
```

```
┌res┐
│ ezs42d000000 │
```

# geohashDecode

Decodes any geohash-encoded string into longitude and latitude.

## Input values

- encoded string - geohash-encoded string.

## Returned values

- (longitude, latitude) - 2-tuple of `Float64` values of longitude and latitude.

## Example

```
SELECT geohashDecode('ezs42') AS res
```

```
┌ res ───────────────────────────────────┐
│ (-5.60302734375,42.60498046875) │
```

# Functions for working with Nullable aggregates

## isNull

Checks whether the argument is `NULL`.

```
isNull(x)
```

## Parameters

- `x` — A value with a non-compound data type.

## Returned value

- 1 if `x` is `NULL`.
- 0 if `x` is not `NULL`.

## Example

Input table

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

Query

```
:) SELECT x FROM t_null WHERE isNull(y)
```

```
SELECT x
FROM t_null
WHERE isNull(y)
```

```
┌ x ───┐
│ 1 │
```

1 rows in set. Elapsed: 0.010 sec.

## isNotNull

Checks whether the argument is **NULL**.

```
isNotNull(x)
```

### Parameters:

- **x** — A value with a non-compound data type.

### Returned value

- 0 if **x** is **NULL**.
- 1 if **x** is not **NULL**.

### Example

Input table

x	y
1	NULL
2	3

Query

```
:) SELECT x FROM t_null WHERE isNotNull(y)
```

```
SELECT x
FROM t_null
WHERE isNotNull(y)
```

x
2

1 rows in set. Elapsed: 0.010 sec.

## coalesce

Checks from left to right whether **NULL** arguments were passed and returns the first non-**NULL** argument.

```
coalesce(x,...)
```

### Parameters:

- Any number of parameters of a non-compound type. All parameters must be compatible by data type.

### Returned values

- The first non-**NULL** argument.
- **NULL**, if all arguments are **NULL**.

### Example

Consider a list of contacts that may specify multiple ways to contact a customer.

name	mail	phone	icq
client 1	NULL	123-45-67	123
client 2	NULL	NULL	NULL

The `mail` and `phone` fields are of type `String`, but the `icq` field is `UInt32`, so it needs to be converted to `String`.

Get the first available contact method for the customer from the contact list:

```
:) SELECT coalesce(mail, phone, CAST(icq,'Nullable(String)')) FROM aBook
```

```
SELECT coalesce(mail, phone, CAST(icq, 'Nullable(String)'))
FROM aBook
```

name	coalesce(mail, phone, CAST(icq, 'Nullable(String)'))
client 1	123-45-67
client 2	NULL

2 rows in set. Elapsed: 0.006 sec.

## ifNull

Returns an alternative value if the main argument is `NULL`.

```
ifNull(x,alt)
```

### Parameters:

- `x` — The value to check for `NULL`.
- `alt` — The value that the function returns if `x` is `NULL`.

### Returned values

- The value `x`, if `x` is not `NULL`.
- The value `alt`, if `x` is `NULL`.

### Example

```
SELECT ifNull('a', 'b')
```

ifNull('a', 'b')
a

```
SELECT ifNull(NULL, 'b')
```

ifNull(NULL, 'b')
b

## nullIf



Returns `NULL` if the arguments are equal.

```
nullIf(x, y)
```

#### Parameters:

`x`, `y` — Values for comparison. They must be compatible types, or ClickHouse will generate an exception.

#### Returned values

- `NULL`, if the arguments are equal.
- The `x` value, if the arguments are not equal.

#### Example

```
SELECT nullIf(1, 1)
```

nullIf(1, 1)	
NULL	

```
SELECT nullIf(1, 2)
```

nullIf(1, 2)	
1	

## assumeNotNull

Results in a value of type `Nullable` for a non-`Nullable`, if the value is not `NULL`.

```
assumeNotNull(x)
```

#### Parameters:

- `x` — The original value.

#### Returned values

- The original value from the non-`Nullable` type, if it is not `NULL`.
- The default value for the non-`Nullable` type if the original value was `NULL`.

#### Example

Consider the `t_null` table.

```
SHOW CREATE TABLE t_null
```

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

x		y	
1		NULL	
2		3	

Apply the `resumenotnull` function to the `y` column.

```
SELECT assumeNotNull(y) FROM t_null
```

assumeNotNull(y)
0
3

```
SELECT toTypeName(assumeNotNull(y)) FROM t_null
```

toTypeName(assumeNotNull(y))
Int8
Int8

## toNullable

Converts the argument type to `Nullable`.

```
toNullable(x)
```

### Parameters:

- `x` — The value of any non-compound type.

### Returned value

- The input value with a non-`Nullable` type.

### Example

```
SELECT toTypeName(10)
```

toTypeName(10)
UInt8

```
SELECT toTypeName(toNullable(10))
```

toTypeName(toNullable(10))
Nullable(UInt8)

## Other functions

### hostName()

Returns a string with the name of the host that this function was performed on. For distributed processing, this is the name of the remote server host, if the function is performed on a remote server.

### basename

Extracts trailing part of a string after the last slash or backslash. This function is often used to extract the filename from the path.

```
basename(expr)
```

## Parameters

- `expr` — Expression, resulting in the `String`-type value. All the backslashes must be escaped in the resulting value.

## Returned Value

A String-type value that contains:

- Trailing part of a string after the last slash or backslash in it.

If the input string contains a path, ending with slash or backslash, for example, `/` or `c:\`, the function returns an empty string.

- Original string if there are no slashes or backslashes in it.

### Example

```
SELECT 'some/long/path/to/file' AS a, basename(a)
```

[illegible]

```
SELECT 'some\\long\\path\\to\\file' AS a, basename(a)
```

The diagram shows the output of the `basename()` function. It consists of two parts:

- A variable `a` containing the string `some\long\path\to\file`.
- The expression `basename('some\\long\\path\\to\\file')`, which evaluates to `file`.

Arrows indicate the flow from the input string to the function call and then to the resulting filename.

```
SELECT 'some-file-name' AS a, basename(a)
```

```

graph LR
 a[a] --- some_file_name[some-file-name]
 some_file_name --> basename_method[basename]
 basename_method --> result[some-file-name]
 style some_file_name fill:#fff,stroke:#333,stroke-width:1px
 style result fill:#fff,stroke:#333,stroke-width:1px

```

## 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)
```

The diagram illustrates the evaluation of the SQL query `SELECT visibleWidth(NULL)`. It shows a single row in a table with the value `4` in the `visibleWidth(NULL)` column. A bracket below the row indicates the result of the function call.

```
└─visibleWidth(NULL)─┘
| |
| 4 |
|
```

## toTypeName(x)

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

If `NULL` is passed to the function as input, then it returns the `Nullable(Nothing)` type, which corresponds to an internal `NULL` representation in ClickHouse.

## blockSize()

Gets the size of the block.

In ClickHouse, queries are always run on blocks (sets of column parts). This function allows getting the size of the block that you called it for.

## materialize(x)

Turns a constant into a full column containing just one value.

In ClickHouse, full columns and constants are represented differently in memory. Functions work differently for constant arguments and normal arguments (different code is executed), although the result is almost always the same. This function is for debugging this behavior.

## ignore(...)

Accepts any arguments, including `NULL`. Always returns 0.

However, the argument is still evaluated. This can be used for benchmarks.

## sleep(seconds)

Sleeps 'seconds' seconds on each data block. You can specify an integer or a floating-point number.

## sleepEachRow(seconds)

Sleeps 'seconds' seconds on each row. You can specify an integer or a floating-point number.

## currentDatabase()

Returns the name of the current database.

You can use this function in table engine parameters in a `CREATE TABLE` query where you need to specify the database.

## isFinite(x)

Accepts `Float32` and `Float64` and returns `UInt8` equal to 1 if the argument is not infinite and not a NaN, otherwise 0.

## isInfinite(x)

Accepts `Float32` and `Float64` and returns `UInt8` equal to 1 if the argument is infinite, otherwise 0. Note that 0 is returned for a NaN.

## isNaN(x)

Accepts `Float32` and `Float64` and returns `UInt8` equal to 1 if the argument is a NaN, otherwise 0.

## hasColumnInTable(['hostname'[, 'username'[, 'password']], 'database', 'table', 'column')

Accepts constant strings: database name, table name, and column name. Returns a `UInt8` constant expression equal to 1 if there is a column, otherwise 0. If the hostname parameter is set, the test will run on a remote server.

The function throws an exception if the table does not exist.

For elements in a nested data structure, the function checks for the existence of a column. For the nested

data structure itself, the function returns 0.

bar

Allows building a unicode-art diagram.

`bar(x, min, max, width)` draws a band with a width proportional to  $(x - \text{min})$  and equal to `width` characters when  $x = \text{max}$ .

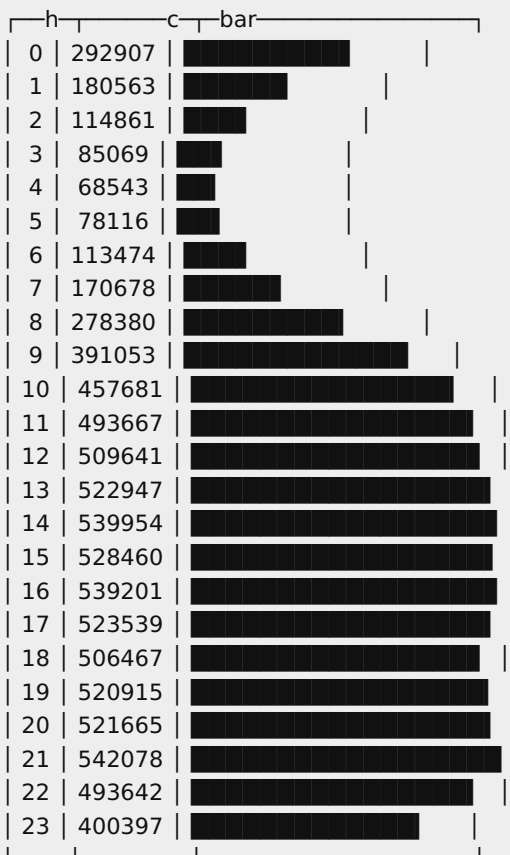
### Parameters:

- `x` — Size to display.
- `min, max` — Integer constants. The value must fit in `Int64`.
- `width` — Constant, positive integer, can be fractional.

The band is drawn with accuracy to one eighth of a symbol.

Example:

```
SELECT
 toHour(EventTime) AS h,
 count() AS c,
 bar(c, 0, 600000, 20) AS bar
FROM test.hits
GROUP BY h
ORDER BY h ASC
```



## transform

Transforms a value according to the explicitly defined mapping of some elements to other ones. There are two variations of this function:

1. `transform(x, array from, array to, default)`

`x` - What to transform.

`array_from` - Constant array of values for converting.

`array_to` - Constant array of values to convert the values in 'from' to.

`default` - Which value to use if 'x' is not equal to any of the values in 'from'.

`array_from` and `array_to` - Arrays of the same size.

Types:

`transform(T, Array(T), Array(U), U) -> U`

`T` and `U` can be numeric, string, or Date or DateTime types.

Where the same letter is indicated (T or U), for numeric types these might not be matching types, but types that have a common type.

For example, the first argument can have the Int64 type, while the second has the Array(UInt16) type.

If the 'x' value is equal to one of the elements in the 'array\_from' array, it returns the existing element (that is numbered the same) from the 'array\_to' array. Otherwise, it returns 'default'. If there are multiple matching elements in 'array\_from', it returns one of the matches.

Example:

```
SELECT
 transform(SearchEngineID, [2, 3], ['Yandex', 'Google'], 'Other') AS title,
 count() AS c
FROM test.hits
WHERE SearchEngineID != 0
GROUP BY title
ORDER BY c DESC
```

title	c
Yandex	498635
Google	229872
Other	104472

## 2. `transform(x, array_from, array_to)`

Differs from the first variation in that the 'default' argument is omitted.

If the 'x' value is equal to one of the elements in the 'array\_from' array, it returns the matching element (that is numbered the same) from the 'array\_to' array. Otherwise, it returns 'x'.

Types:

`transform(T, Array(T), Array(T)) -> T`

Example:

```
SELECT
 transform(domain(Referer), ['yandex.ru', 'google.ru', 'vk.com'], ['www.yandex', 'example.com']) AS s,
 count() AS c
FROM test.hits
GROUP BY domain(Referer)
ORDER BY count() DESC
LIMIT 10
```

S		C	
	2906259		
www.yandex	867767		
████████.ru	313599		
mail.yandex.ru	107147		
████████.ru	100355		
████████.ru	65040		
news.yandex.ru	64515		
████████.net	59141		
example.com	57316		

## formatReadableSize(x)

Accepts the size (number of bytes). Returns a rounded size with a suffix (KiB, MiB, etc.) as a string.

Example:

```
SELECT
 arrayJoin([1, 1024, 1024*1024, 192851925]) AS filesize_bytes,
 formatReadableSize(filesize_bytes) AS filesize
```

filesize_bytes	filesize
1	1.00 B
1024	1.00 KiB
1048576	1.00 MiB
192851925	183.92 MiB

## least(a, b)

Returns the smallest value from a and b.

## greatest(a, b)

Returns the largest value of a and b.

## uptime()

Returns the server's uptime in seconds.

## version()

Returns the version of the server as a string.

## timezone()

Returns the timezone of the server.

## blockNumber

Returns the sequence number of the data block where the row is located.

## rowNumberInBlock

Returns the ordinal number of the row in the data block. Different data blocks are always recalculated.

## rowNumberInAllBlocks()

## ROWNUMBERHATABLOCKS()

Returns the ordinal number of the row in the data block. This function only considers the affected data blocks.

## runningDifference(x)

Calculates the difference between successive row values in the data block.

Returns 0 for the first row and the difference from the previous row for each subsequent row.

The result of the function depends on the affected data blocks and the order of data in the block.

If you make a subquery with ORDER BY and call the function from outside the subquery, you can get the expected result.

Example:

```
SELECT
 EventID,
 EventTime,
 runningDifference(EventTime) AS delta
FROM
(
 SELECT
 EventID,
 EventTime
 FROM events
 WHERE EventDate = '2016-11-24'
 ORDER BY EventTime ASC
 LIMIT 5
)
```

EventID	EventTime	delta
1106	2016-11-24 00:00:04	0
1107	2016-11-24 00:00:05	1
1108	2016-11-24 00:00:05	0
1109	2016-11-24 00:00:09	4
1110	2016-11-24 00:00:10	1

## runningDifferenceStartingWithFirstValue

Same as for **runningDifference**, the difference is the value of the first row, returned the value of the first row, and each subsequent row returns the difference from the previous row.

## MACNumToString(num)

Accepts a UInt64 number. Interprets it as a MAC address in big endian. Returns a string containing the corresponding MAC address in the format AA:BB:CC:DD:EE:FF (colon-separated numbers in hexadecimal form).

## MACStringToNum(s)

The inverse function of MACNumToString. If the MAC address has an invalid format, it returns 0.

## MACStringToOUI(s)

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

## getsizeofEnumType



Returns the number of fields in **Enum**.

```
getsizeofEnumType(value)
```

#### Parameters:

- **value** — Value of type **Enum**.

#### Returned values

- The number of fields with **Enum** input values.
- An exception is thrown if the type is not **Enum**.

#### Example

```
SELECT sizeofEnumType(CAST('a' AS Enum8('a' = 1, 'b' = 2))) AS x
```

x
2

## toColumnName

Returns the name of the class that represents the data type of the column in RAM.

```
toColumnName(value)
```

#### Parameters:

- **value** — Any type of value.

#### Returned values

- A string with the name of the class that is used for representing the **value** data type in RAM.

#### Example of the difference between **toTypeName** ' and ' **toColumnName**

```
:) select toTypeName(cast('2018-01-01 01:02:03' AS DateTime))
```

```
SELECT toTypeName(CAST('2018-01-01 01:02:03', 'DateTime'))
```

toTypeName(CAST('2018-01-01 01:02:03', 'DateTime'))
DateTime

1 rows in set. Elapsed: 0.008 sec.

```
:) select toColumnName(cast('2018-01-01 01:02:03' AS DateTime))
```

```
SELECT toColumnName(CAST('2018-01-01 01:02:03', 'DateTime'))
```

toColumnName(CAST('2018-01-01 01:02:03', 'DateTime'))
Const(UInt32)

The example shows that the **DateTime** data type is stored in memory as **Const(UInt32)**.

Outputs a detailed description of data structures in RAM

dumpColumnStructure(value)

- **value** — Any type of value.

- A string describing the structure that is used for representing the **value** data type in RAM.

```
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))┐
```

Outputs the default value for the data type.

Does not include default values for custom columns set by the user.

defaultValueOfArgumentType(expression)

- expression** — Arbitrary type of value or an expression that results in a value of an arbitrary type.

- 0 for numbers.
- Empty string for strings.
- NULL for Nullable.

```
:) 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 defaultValueOfType(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

- 1.

### Example

Here is a table with the test data for `ontime`.

```
SELECT count() FROM ontime
```

count()
4276457

The table has indexes for the fields `(FlightDate, (Year, FlightDate))`.

Create a selection by date like this:

```
:) SELECT FlightDate AS k, count() FROM ontime GROUP BY k ORDER BY k
```

```
SELECT
 FlightDate AS k,
 count()
FROM ontime
GROUP BY k
ORDER BY k ASC
```

k	count()
2017-01-01	13970
2017-01-02	15882
.....	
2017-09-28	16411
2017-09-29	16384
2017-09-30	12520

273 rows in set. Elapsed: 0.072 sec. Processed 4.28 million rows, 8.55 MB (59.00 million rows/s., 118.01 MB/s.)

In this selection, the index is not used and ClickHouse processed the entire table (Processed 4.28 million rows). To apply the index, select a specific date and run the following query:

```
:) SELECT FlightDate AS k, count() FROM ontime WHERE k = '2017-09-15' GROUP BY k ORDER BY k
```

```
SELECT
 FlightDate AS k,
 count()
FROM ontime
WHERE k = '2017-09-15'
GROUP BY k
```

```
GROUP BY k
ORDER BY k ASC
```

k	count()
2017-09-15	16428

1 rows in set. Elapsed: 0.014 sec. Processed 32.74 thousand rows, 65.49 KB (2.31 million rows/s., 4.63 MB/s.)

The last line of output shows that by using the index, ClickHouse processed a significantly smaller number of rows (Processed 32.74 thousand rows).

Now pass the expression `k = '2017-09-15'` to the `indexHint` function:

```
:) SELECT FlightDate AS k, count() FROM ontime WHERE indexHint(k = '2017-09-15') GROUP BY k ORDER BY k
```

```
SELECT
 FlightDate AS k,
 count()
FROM ontime
WHERE indexHint(k = '2017-09-15')
GROUP BY k
ORDER BY k ASC
```

k	count()
2017-09-14	7071
2017-09-15	16428
2017-09-16	1077
2017-09-30	8167

4 rows in set. Elapsed: 0.004 sec. Processed 32.74 thousand rows, 65.49 KB (8.97 million rows/s., 17.94 MB/s.)

The response to the request shows that ClickHouse applied the index in the same way as the previous time (Processed 32.74 thousand rows). However, the resulting set of rows shows that the expression `k = '2017-09-15'` was not used when generating the result.

Because the index is sparse in ClickHouse, "extra" data ends up in the response when reading a range (in this case, the adjacent dates). Use the `indexHint` function to see it.

## replicate

Creates an array with a single value.

Used for internal implementation of `arrayJoin`.

```
replicate(x, arr)
```

### Parameters:

- `arr` — Original array. ClickHouse creates a new array of the same length as the original and fills it with the value `x`.
- `x` — The value that the resulting array will be filled with.

### Output value

- An array filled with the value `x`.

### Example

```
SELECT replicate(1, ['a', 'b', 'c'])
```

```
┌replicate(1, ['a', 'b', 'c'])┐
└ [1,1,1] ┘
```

## filesystemAvailable

Returns the remaining space information of the disk, in bytes. This information is evaluated using the configured by path.

## filesystemCapacity

Returns the capacity information of the disk, in bytes. This information is evaluated using the configured by path.

## finalizeAggregation

Takes state of aggregate function. Returns result of aggregation (finalized state).

## runningAccumulate

Takes the states of the aggregate function and returns a column with values, are the result of the accumulation of these states for a set of block lines, from the first to the current line.

For example, takes state of aggregate function (example `runningAccumulate(uniqState(UserID))`), and for each row of block, return result of aggregate function on merge of states of all previous rows and current row.

So, result of function depends on partition of data to blocks and on order of data in block.

## joinGet('join\_storage\_table\_name', 'get\_column', join\_key)

Get data from a table of type Join using the specified join key.

## modelEvaluate(model\_name, ...)

Evaluate external model.

Accepts a model name and model arguments. Returns Float64.

## throwIf(x)

Throw an exception if the argument is non zero.

## Aggregate functions

Aggregate functions work in the **normal** way as expected by database experts.

ClickHouse also supports:

- **Parametric aggregate functions**, which accept other parameters in addition to columns.
- **Combinators**, which change the behavior of aggregate functions.

## NULL processing

During aggregation, all **NULLs** are skipped.

### Examples:

Consider this table:

```
┌x┐└y┐
```

1	2
2	NULL
3	2
3	3
3	NULL

Let's say you need to total the values in the `y` column:

```
:) SELECT sum(y) FROM t_null_big
```

```
SELECT sum(y)
FROM t_null_big
```

sum(y)
7

1 rows in set. Elapsed: 0.002 sec.

The `sum` function interprets `NULL` as `0`. In particular, this means that if the function receives input of a selection where all the values are `NULL`, then the result will be `0`, not `NULL`.

Now you can use the `groupArray` function to create an array from the `y` column:

```
:) SELECT groupArray(y) FROM t_null_big
```

```
SELECT groupArray(y)
FROM t_null_big
```

groupArray(y)
[2,2,3]

1 rows in set. Elapsed: 0.002 sec.

`groupArray` does not include `NULL` in the resulting array.

## Function reference

### count()

Counts the number of rows. Accepts zero arguments and returns `UInt64`.

The syntax `COUNT(DISTINCT x)` is not supported. The separate `uniq` aggregate function exists for this purpose.

A `SELECT count() FROM table` query is not optimized, because the number of entries in the table is not stored separately. It will select some small column from the table and count the number of values in it.

### any(x)

Selects the first encountered value.

The query can be executed in any order and even in a different order each time, so the result of this function is indeterminate.

To get a determinate result, you can use the 'min' or 'max' function instead of 'any'.

In some cases, you can rely on the order of execution. This applies to cases when `SELECT` comes from a subquery that uses `ORDER BY`.

When a `SELECT` query has the `GROUP BY` clause or at least one aggregate function, ClickHouse (in contrast to

When a `SELECT` query has the `GROUP BY` clause or at least one aggregate function, ClickHouse (in contrast to MySQL) requires that all expressions in the `SELECT`, `HAVING`, and `ORDER BY` clauses be calculated from keys or from aggregate functions. In other words, each column selected from the table must be used either in keys or inside aggregate functions. To get behavior like in MySQL, you can put the other columns in the `any` aggregate function.

## anyHeavy(x)

Selects a frequently occurring value using the **heavy hitters** algorithm. If there is a value that occurs more than in half the cases in each of the query's execution threads, this value is returned. Normally, the result is nondeterministic.

```
anyHeavy(column)
```

### Arguments

- `column` – The column name.

### Example

Take the **OnTime** data set and select any frequently occurring value in the `AirlineID` column.

```
SELECT anyHeavy(AirlineID) AS res
FROM ontime
```

```
┌ res ─┐
│ 19690 │
```

## anyLast(x)

Selects the last value encountered.

The result is just as indeterminate as for the `any` function.

## groupBitAnd

Applies bitwise `AND` for series of numbers.

```
groupBitAnd(expr)
```

### Parameters

`expr` – An expression that results in `UInt*` type.

### Return value

Value of the `UInt*` type.

### Example

Test data:

```
binary decimal
00101100 = 44
00011100 = 28
00001101 = 13
01010101 = 85
```

Query:

```
SELECT groupBitAnd(num) FROM t
```

Where `num` is the column with the test data.

Result:

binary	decimal
00000100	= 4

## groupBitOr

Applies bitwise `OR` for series of numbers.

```
groupBitOr(expr)
```

### Parameters

`expr` – An expression that results in `UInt*` type.

### Return value

Value of the `UInt*` type.

### Example

Test data:

binary	decimal
00101100	= 44
00011100	= 28
00001101	= 13
01010101	= 85

Query:

```
SELECT groupBitOr(num) FROM t
```

Where `num` is the column with the test data.

Result:

binary	decimal
01111101	= 125

## groupBitXor

Applies bitwise `XOR` for series of numbers.

```
groupBitXor(expr)
```

### Example



## Parameters

`expr` – An expression that results in `UInt*` type.

## Return value

Value of the `UInt*` type.

## Example

Test data:

```
binary decimal
00101100 = 44
00011100 = 28
00001101 = 13
01010101 = 85
```

Query:

```
SELECT groupBitXor(num) FROM t
```

Where `num` is the column with the test data.

Result:

```
binary decimal
01101000 = 104
```

# groupBitmap

Bitmap or Aggregate calculations from a unsigned integer column, return cardinality of type `UInt64`, if add suffix `-State`, then return **bitmap object**.

```
groupBitmap(expr)
```

## Parameters

`expr` – An expression that results in `UInt*` type.

## Return value

Value of the `UInt64` type.

## Example

Test data:

```
userid
1
1
2
3
```

Query:

```
SELECT groupBitmap(userid) as num FROM t
```

```
SELECT groupBitmap(user_id) as num FROM t
```

Result:

```
num
3
```

## min(x)

Calculates the minimum.

## max(x)

Calculates the maximum.

## argMin(arg, val)

Calculates the 'arg' value for a minimal 'val' value. If there are several different values of 'arg' for minimal values of 'val', the first of these values encountered is output.

**Example:**

user	salary
director	5000
manager	3000
worker	1000

```
SELECT argMin(user, salary) FROM salary
```

argMin(user, salary)
worker

## argMax(arg, val)

Calculates the 'arg' value for a maximum 'val' value. If there are several different values of 'arg' for maximum values of 'val', the first of these values encountered is output.

## sum(x)

Calculates the sum.

Only works for numbers.

## sumWithOverflow(x)

Computes the sum of the numbers, using the same data type for the result as for the input parameters. If the sum exceeds the maximum value for this data type, the function returns an error.

Only works for numbers.

## sumMap(key, value)

Totals the 'value' array according to the keys specified in the 'key' array.

The number of elements in 'key' and 'value' must be the same for each row that is totaled.

Returns a tuple of two arrays: keys in sorted order, and values summed for the corresponding keys.

Example:

```
CREATE TABLE sum_map(
 date Date,
 timeslot DateTime,
 statusMap Nested(
 status UInt16,
 requests UInt64
)
) ENGINE = Log;
INSERT INTO sum_map VALUES
 ('2000-01-01', '2000-01-01 00:00:00', [1, 2, 3], [10, 10, 10]),
 ('2000-01-01', '2000-01-01 00:00:00', [3, 4, 5], [10, 10, 10]),
 ('2000-01-01', '2000-01-01 00:01:00', [4, 5, 6], [10, 10, 10]),
 ('2000-01-01', '2000-01-01 00:01:00', [6, 7, 8], [10, 10, 10]);
SELECT
 timeslot,
 sumMap(statusMap.status, statusMap.requests)
FROM sum_map
GROUP BY timeslot
```

timeslot	sumMap(statusMap.status, statusMap.requests)
2000-01-01 00:00:00	([1,2,3,4,5],[10,10,20,10,10])
2000-01-01 00:01:00	([4,5,6,7,8],[10,10,20,10,10])

## timeSeriesGroupSum(uid, timestamp, value)

timeSeriesGroupSum can aggregate different time series that sample timestamp not alignment. It will use linear interpolation between two sample timestamp and then sum time-series together.

**uid** is the time series unique id, UInt64.

**timestamp** is Int64 type in order to support millisecond or microsecond.

**value** is the metric.

Before use this function, timestamp should be in ascend order

Example:

uid	timestamp	value
1	2	0.2
1	7	0.7
1	12	1.2
1	17	1.7
1	25	2.5
2	3	0.6
2	8	1.6
2	12	2.4
2	18	3.6
2	24	4.8

```
CREATE TABLE time_series(
 uid UInt64,
 timestamp Int64,
 value Float64
) ENGINE = Memory;
INSERT INTO time_series VALUES
 (1,2,0.2),(1,7,0.7),(1,12,1.2),(1,17,1.7),(1,25,2.5),
 (2,3,0.6),(2,8,1.6),(2,12,2.4),(2,18,3.6),(2,24,4.8);
```

```
SELECT timeSeriesGroupSum(uid, timestamp, value)
FROM (
 SELECT * FROM time_series order by timestamp ASC
);
```

And the result will be:

```
[(2,0.2),(3,0.9),(7,2.1),(8,2.4),(12,3.6),(17,5.1),(18,5.4),(24,7.2),(25,2.5)]
```

## timeSeriesGroupRateSum(uid, ts, val)

Similarly timeSeriesGroupRateSum, timeSeriesGroupRateSum will Calculate the rate of time-series and then sum rates together.

Also, timestamp should be in ascend order before use this function.

Use this function, the result above case will be:

```
[(2,0),(3,0.1),(7,0.3),(8,0.3),(12,0.3),(17,0.3),(18,0.3),(24,0.3),(25,0.1)]
```

## avg(x)

Calculates the average.

Only works for numbers.

The result is always Float64.

## uniq(x)

Calculates the approximate number of different values of the argument. Works for numbers, strings, dates, date-with-time, and for multiple arguments and tuple arguments.

Uses an adaptive sampling algorithm: for the calculation state, it uses a sample of element hash values with a size up to 65536.

This algorithm is also very accurate for data sets with low cardinality (up to 65536) and very efficient on CPU (when computing not too many of these functions, using `uniq` is almost as fast as using other aggregate functions).

The result is determinate (it doesn't depend on the order of query processing).

This function provides excellent accuracy even for data sets with extremely high cardinality (over 10 billion elements). It is recommended for default use.

## uniqCombined(HLL\_precision)(x)

Calculates the approximate number of different values of the argument. Works for numbers, strings, dates, date-with-time, and for multiple arguments and tuple arguments.

A combination of three algorithms is used: array, hash table and **HyperLogLog** with an error correction table. For small number of distinct elements, the array is used. When the set size becomes larger the hash table is used, while it is smaller than HyperLogLog data structure. For larger number of elements, the HyperLogLog is used, and it will occupy fixed amount of memory.

The parameter "HLL\_precision" is the base-2 logarithm of the number of cells in HyperLogLog. You can omit the parameter (omit first parens). The default value is 17, that is effectively 96 KiB of space ( $2^{17}$  cells of 6 bits each). The memory consumption is several times smaller than for the `uniq` function, and the accuracy is several times higher. Performance is slightly lower than for the `uniq` function, but sometimes it can be even higher than it, such as with distributed queries that transmit a large number of aggregation states over the network

network.

The result is deterministic (it doesn't depend on the order of query processing).

The `uniqCombined` function is a good default choice for calculating the number of different values, but keep in mind that the estimation error for large sets (200 million elements and more) will become larger than theoretical value due to poor choice of hash function.

## uniqHLL12(x)

Uses the **HyperLogLog** algorithm to approximate the number of different values of the argument.

212 5-bit cells are used. The size of the state is slightly more than 2.5 KB. The result is not very accurate (up to ~10% error) for small data sets (<10K elements). However, the result is fairly accurate for high-cardinality data sets (10K-100M), with a maximum error of ~1.6%. Starting from 100M, the estimation error increases, and the function will return very inaccurate results for data sets with extremely high cardinality (1B+ elements).

The result is determinate (it doesn't depend on the order of query processing).

We don't recommend using this function. In most cases, use the `uniq` or `uniqCombined` function.

## uniqExact(x)

Calculates the number of different values of the argument, exactly.

There is no reason to fear approximations. It's better to use the `uniq` function.

Use the `uniqExact` function if you definitely need an exact result.

The `uniqExact` function uses more memory than the `uniq` function, because the size of the state has unbounded growth as the number of different values increases.

## groupArray(x), groupArray(max\_size)(x)

Creates an array of argument values.

Values can be added to the array in any (indeterminate) order.

The second version (with the `max_size` parameter) limits the size of the resulting array to `max_size` elements. For example, `groupArray (1) (x)` is equivalent to `[any (x)]`.

In some cases, you can still rely on the order of execution. This applies to cases when `SELECT` comes from a subquery that uses `ORDER BY`.

## groupArrayInsertAt(x)

Inserts a value into the array in the specified position.

Accepts the value and position as input. If several values are inserted into the same position, any of them might end up in the resulting array (the first one will be used in the case of single-threaded execution). If no value is inserted into a position, the position is assigned the default value.

Optional parameters:

- The default value for substituting in empty positions.
- The length of the resulting array. This allows you to receive arrays of the same size for all the aggregate keys. When using this parameter, the default value must be specified.

## groupUniqArray(x), groupUniqArray(max\_size)(x)

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

The second version (with the `max_size` parameter) limits the size of the resulting array to `max_size` elements. For example, `groupUniqArray (1) (x)` is equivalent to `[any (x)]`.

## quantile(level)(x)

Approximates the `level` quantile. `level` is a constant, a floating-point number from 0 to 1.

We recommend using a `level` value in the range of `[0.01, 0.99]`

Don't use a `level` value equal to 0 or 1 – use the `min` and `max` functions for these cases.

In this function, as well as in all functions for calculating quantiles, the `level` parameter can be omitted. In this case, it is assumed to be equal to 0.5 (in other words, the function will calculate the median).

Works for numbers, dates, and dates with times.

Returns: for numbers – `Float64`; for dates – a date; for dates with times – a date with time.

Uses **reservoir sampling** with a reservoir size up to 8192.

If necessary, the result is output with linear approximation from the two neighboring values.

This algorithm provides very low accuracy. See also: `quantileTiming`, `quantileTDigest`, `quantileExact`.

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

When using multiple `quantile` (and similar) functions with different levels in a query, the internal states are not combined (that is, the query works less efficiently than it could). In this case, use the `quantiles` (and similar) functions.

## quantileDeterministic(level)(x, determinator)

Works the same way as the `quantile` function, but the result is deterministic and does not depend on the order of query execution.

To achieve this, the function takes a second argument – the "determinator". This is a number whose hash is used instead of a random number generator in the reservoir sampling algorithm. For the function to work correctly, the same determinator value should not occur too often. For the determinator, you can use an event ID, user ID, and so on.

Don't use this function for calculating timings. There is a more suitable function for this purpose: `quantileTiming`.

## quantileTiming(level)(x)

Computes the quantile of 'level' with a fixed precision.

Works for numbers. Intended for calculating quantiles of page loading time in milliseconds.

If the value is greater than 30,000 (a page loading time of more than 30 seconds), the result is equated to 30,000.

If the total value is not more than about 5670, then the calculation is accurate.

Otherwise:

- if the time is less than 1024 ms, then the calculation is accurate.
- otherwise the calculation is rounded to a multiple of 16 ms.

When passing negative values to the function, the behavior is undefined.

The returned value has the `Float32` type. If no values were passed to the function (when using `quantileTimingIf`), 'nan' is returned. The purpose of this is to differentiate these instances from zeros. See the note on sorting NaNs in "ORDER BY clause".

The result is determinate (it doesn't depend on the order of query processing).

For its purpose (calculating quantiles of page loading times), using this function is more effective and the result is more accurate than for the `quantile` function.

## quantileTimingWeighted(level)(x, weight)

Differs from the `quantileTiming` function in that it has a second argument, "weights". Weight is a non-negative integer.

The result is calculated as if the `x` value were passed `weight` number of times to the `quantileTiming` function.

## quantileExact(level)(x)

Computes the quantile of 'level' exactly. To do this, all the passed values are combined into an array, which is then partially sorted. Therefore, the function consumes  $O(n)$  memory, where 'n' is the number of values that were passed. However, for a small number of values, the function is very effective.

## quantileExactWeighted(level)(x, weight)

Computes the quantile of 'level' exactly. In addition, each value is counted with its weight, as if it is present 'weight' times. The arguments of the function can be considered as histograms, where the value 'x' corresponds to a histogram "column" of the height 'weight', and the function itself can be considered as a summation of histograms.

A hash table is used as the algorithm. Because of this, if the passed values are frequently repeated, the function consumes less RAM than `quantileExact`. You can use this function instead of `quantileExact` and specify the weight as 1.

## quantileTDigest(level)(x)

Approximates the quantile level using the **t-digest** algorithm. The maximum error is 1%. Memory consumption by State is proportional to the logarithm of the number of passed values.

The performance of the function is lower than for `quantile` or `quantileTiming`. In terms of the ratio of State size to precision, this function is much better than `quantile`.

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

## median(x)

All the quantile functions have corresponding median functions: `median`, `medianDeterministic`, `medianTiming`, `medianTimingWeighted`, `medianExact`, `medianExactWeighted`, `medianTDigest`. They are synonyms and their behavior is identical.

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

All the quantile functions also have corresponding quantiles functions: `quantiles`, `quantilesDeterministic`, `quantilesTiming`, `quantilesTimingWeighted`, `quantilesExact`, `quantilesExactWeighted`, `quantilesTDigest`. These functions calculate all the quantiles of the listed levels in one pass, and return an array of the resulting values.

## varSamp(x)

Calculates the amount  $\sum((x - \bar{x})^2) / (n - 1)$ , where `n` is the sample size and  $\bar{x}$  is the average value of `x`.

It represents an unbiased estimate of the variance of a random variable, if the values passed to the function are a sample of this random amount.

Returns `Float64`. When `n <= 1`, returns `+∞`.

## varPop(x)

Calculates the amount  $\sum((x - \bar{x})^2) / n$ , where `n` is the sample size and  $\bar{x}$  is the average value of `x`.

In other words, dispersion for a set of values. Returns `Float64`.

## stddevSamp(x)

The result is equal to the square root of `varSamp(x)`.

## stddevPop(x)

The result is equal to the square root of `varPop(x)`.

## topK(N)(column)

Returns an array of the most frequent values in the specified column. The resulting array is sorted in descending order of frequency of values (not by the values themselves).

Implements the [Filtered Space-Saving](#) algorithm for analyzing TopK, based on the reduce-and-combine algorithm from [Parallel Space Saving](#).

```
topK(N)(column)
```

This function doesn't provide a guaranteed result. In certain situations, errors might occur and it might return frequent values that aren't the most frequent values.

We recommend using the `N < 10` value; performance is reduced with large `N` values. Maximum value of `N` = 65536.

### Arguments

- 'N' is the number of values.
- 'x' – The column.

### Example

Take the [OnTime](#) data set and select the three most frequently occurring values in the `AirlineID` column.

```
SELECT topK(3)(AirlineID) AS res
FROM ontime
```

```
res
[19393,19790,19805]
```

## covarSamp(x, y)

Calculates the value of  $\Sigma((x - \bar{x})(y - \bar{y})) / (n - 1)$ .

Returns Float64. When `n <= 1`, returns  $+\infty$ .

## covarPop(x, y)

Calculates the value of  $\Sigma((x - \bar{x})(y - \bar{y})) / n$ .

## corr(x, y)

Calculates the Pearson correlation coefficient:  $\Sigma((x - \bar{x})(y - \bar{y})) / \sqrt{\Sigma((x - \bar{x})^2) * \Sigma((y - \bar{y})^2)}$ .

## simpleLinearRegression

Performs simple (unidimensional) linear regression.

```
simpleLinearRegression(x, y)
```



Parameters:

- `x` — Column with values of dependent variable.
- `y` — Column with explanatory variable.

Returned values:

Parameters `(a, b)` of the resulting line  $x = a*y + b$ .

## Examples

```
SELECT arrayReduce('simpleLinearRegression', [0, 1, 2, 3], [0, 1, 2, 3])
```

```
└─arrayReduce('simpleLinearRegression', [0, 1, 2, 3], [0, 1, 2, 3])┐
├ (1,0) ───┤
└──┘
```

```
SELECT arrayReduce('simpleLinearRegression', [0, 1, 2, 3], [3, 4, 5, 6])
```

```
└─arrayReduce('simpleLinearRegression', [0, 1, 2, 3], [3, 4, 5, 6])┐
├ (1,3) ───┤
└──┘
```

# Aggregate function combinators

The name of an aggregate function can have a suffix appended to it. This changes the way the aggregate function works.

## -If

The suffix `-If` can be appended to the name of any aggregate function. In this case, the aggregate function accepts an extra argument – a condition (UInt8 type). The aggregate function processes only the rows that trigger the condition. If the condition was not triggered even once, it returns a default value (usually zeros or empty strings).

Examples: `sumIf(column, cond)`, `countIf(cond)`, `avgIf(x, cond)`, `quantilesTimingIf(level1, level2)(x, cond)`, `argMinIf(arg, val, cond)` and so on.

With conditional aggregate functions, you can calculate aggregates for several conditions at once, without using subqueries and `JOINS`. For example, in Yandex.Metrica, conditional aggregate functions are used to implement the segment comparison functionality.

## -Array

The `-Array` suffix can be appended to any aggregate function. In this case, the aggregate function takes arguments of the 'Array(T)' type (arrays) instead of 'T' type arguments. If the aggregate function accepts multiple arguments, this must be arrays of equal lengths. When processing arrays, the aggregate function works like the original aggregate function across all array elements.

Example 1: `sumArray(arr)` - Totals all the elements of all 'arr' arrays. In this example, it could have been written more simply: `sum(arraySum(arr))`.

Example 2: `uniqArray(arr)` - Count the number of unique elements in all 'arr' arrays. This could be done an easier way: `uniq(arrayJoin(arr))`, but it's not always possible to add 'arrayJoin' to a query.

-If and -Array can be combined. However, 'Array' must come first, then 'If'. Examples: `uniqArrayIf(arr, cond)`, `quantilesTimingArrayIf(level1, level2)(arr, cond)`. Due to this order, the 'cond' argument can't be an array.

## -State

If you apply this combinator, the aggregate function doesn't return the resulting value (such as the number of unique values for the `uniq` function), but an intermediate state of the aggregation (for `uniq`, this is the hash table for calculating the number of unique values). This is an `AggregateFunction(...)` that can be used for further processing or stored in a table to finish aggregating later. See the sections "AggregatingMergeTree" and "Functions for working with intermediate aggregation states".

## -Merge

If you apply this combinator, the aggregate function takes the intermediate aggregation state as an argument, combines the states to finish aggregation, and returns the resulting value.

## -MergeState.

Merges the intermediate aggregation states in the same way as the -Merge combinator. However, it doesn't return the resulting value, but an intermediate aggregation state, similar to the -State combinator.

## -ForEach

Converts an aggregate function for tables into an aggregate function for arrays that aggregates the corresponding array items and returns an array of results. For example, `sumForEach` for the arrays `[1, 2]`, `[3, 4, 5]` and `[6, 7]` returns the result `[10, 13, 5]` after adding together the corresponding array items.

## Parametric aggregate functions

Some aggregate functions can accept not only argument columns (used for compression), but a set of parameters – constants for initialization. The syntax is two pairs of brackets instead of one. The first is for parameters, and the second is for arguments.

## `sequenceMatch(pattern)(time, cond1, cond2, ...)`

Pattern matching for event chains.

`pattern` is a string containing a pattern to match. The pattern is similar to a regular expression.

`time` is the time of the event, type support: `Date`, `DateTime`, and other unsigned integer types.

`cond1`, `cond2` ... is from one to 32 arguments of type `UInt8` that indicate whether a certain condition was met for the event.

The function collects a sequence of events in RAM. Then it checks whether this sequence matches the pattern.

It returns `UInt8`: 0 if the pattern isn't matched, or 1 if it matches.

Example: `sequenceMatch ('(?1).*(?2)')(EventTime, URL LIKE '%company%', URL LIKE '%cart%')`

- whether there was a chain of events in which a pageview with 'company' in the address occurred earlier than a pageview with 'cart' in the address.

This is a singular example. You could write it using other aggregate functions:

```
minIf(EventTime, URL LIKE '%company%') < maxIf(EventTime, URL LIKE '%cart%').
```

However, there is no such solution for more complex situations.

Pattern syntax:

(?1) refers to the condition (any number can be used in place of 1).

.\* is any number of any events.

(?t>=1800) is a time condition.

Any quantity of any type of events is allowed over the specified time.

Instead of >=, the following operators can be used: <, >, <=.

Any number may be specified in place of 1800.

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

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

Works the same way as the sequenceMatch function, but instead of returning whether there is an event chain, it returns UInt64 with the number of event chains found.

Chains are searched for without overlapping. In other words, the next chain can start only after the end of the previous one.

## windowFunnel(window)(timestamp, cond1, cond2, cond3, ...)

Searches for event chains in a sliding time window and calculates the maximum number of events that occurred from the chain.

```
windowFunnel(window)(timestamp, cond1, cond2, cond3, ...)
```

### Parameters:

- `window` — Length of the sliding window in seconds.
- `timestamp` — Name of the column containing the timestamp. Data type support: `Date`, `DateTime`, and other unsigned integer types (Note that though timestamp support `UInt64` type, there is a limitation it's value can't overflow maximum of `Int64`, which is  $2^{63} - 1$ ).
- `cond1`, `cond2`... — Conditions or data describing the chain of events. Data type: `UInt8`. Values can be 0 or 1.

### Algorithm

- The function searches for data that triggers the first condition in the chain and sets the event counter to 1. This is the moment when the sliding window starts.
- If events from the chain occur sequentially within the window, the counter is incremented. If the sequence of events is disrupted, the counter isn't incremented.
- If the data has multiple event chains at varying points of completion, the function will only output the size of the longest chain.

### Returned value

- Integer. The maximum number of consecutive triggered conditions from the chain within the sliding time window. All the chains in the selection are analyzed.

### Example

Determine if one hour is enough for the user to select a phone and purchase it in the online store.

Set the following chain of events:

- The user logged in to their account on the store (`eventID=1001`).
- The user searches for a phone (`eventID = 1003`, `product = 'phone'`)

2. The user searches for a phone (eventID = 1003, product = 'phone').
3. The user placed an order (eventID = 1009).

To find out how far the user `user_id` could get through the chain in an hour in January of 2017, make the query:

```
SELECT
 level,
 count() AS c
FROM
 (
 SELECT
 user_id,
 windowFunnel(3600)(timestamp, eventID = 1001, eventID = 1003 AND product = 'phone', eventID = 1009) AS
 level
 FROM trend_event
 WHERE (event_date >= '2017-01-01') AND (event_date <= '2017-01-31')
 GROUP BY user_id
)
GROUP BY level
ORDER BY level
```

Simply, the level value could only be 0, 1, 2, 3, it means the maximum event action stage that one user could reach.

## retention(cond1, cond2, ...)

Retention refers to the ability of a company or product to retain its customers over some specified periods.

`cond1`, `cond2` ... is from one to 32 arguments of type UInt8 that indicate whether a certain condition was met for the event

Example:

Consider you are doing a website analytics, intend to calculate the retention of customers

This could be easily calculate by `retention`

```
SELECT
 sum(r[1]) AS r1,
 sum(r[2]) AS r2,
 sum(r[3]) AS r3
FROM
 (
 SELECT
 uid,
 retention(date = '2018-08-10', date = '2018-08-11', date = '2018-08-12') AS r
 FROM events
 WHERE date IN ('2018-08-10', '2018-08-11', '2018-08-12')
 GROUP BY uid
)
```

Simply, `r1` means the number of unique visitors who met the `cond1` condition, `r2` means the number of unique visitors who met `cond1` and `cond2` conditions, `r3` means the number of unique visitors who met `cond1` and `cond3` conditions.

## uniqUpTo(N)(x)

Calculates the number of different argument values if it is less than or equal to N. If the number of different argument values is greater than N, it returns N + 1.

Recommended for use with small Ns, up to 10. The maximum value of N is 100.

For the state of an aggregate function, it uses the amount of memory equal to  $1 + N \times \text{the size of one value of bytes}$ .

For strings, it stores a non-cryptographic hash of 8 bytes. That is, the calculation is approximated for strings.

The function also works for several arguments.

It works as fast as possible, except for cases when a large N value is used and the number of unique values is slightly less than N.

Usage example:

Problem: Generate a report that shows only keywords that produced at least 5 unique users.

Solution: Write in the GROUP BY query SearchPhrase HAVING uniqUpTo(4)(UserID) >= 5

## sumMapFiltered(keys\_to\_keep)(keys, values)

Same behavior as `sumMap` except that an array of keys is passed as a parameter. This can be especially useful when working with a high cardinality of keys.

## Table functions

Table functions can be specified in the FROM clause instead of the database and table names.

Table functions can only be used if 'readonly' is not set.

Table functions aren't related to other functions.

## file

Creates a table from a file.

```
file(path, format, structure)
```

### Input parameters

- `path` — The relative path to the file from `user_files_path`.
- `format` — The `format` of the file.
- `structure` — Structure of the table. Format `'column1_name column1_type, column2_name column2_type, ...'`.

### Returned value

A table with the specified structure for reading or writing data in the specified file.

### Example

Setting `user_files_path` and the contents of the file `test.csv`:

```
$ grep user_files_path /etc/clickhouse-server/config.xml
<user_files_path>/var/lib/clickhouse/user_files/</user_files_path>

$ cat /var/lib/clickhouse/user_files/test.csv
1,2,3
3,2,1
78,43,45
```

Table from `test.csv` and selection of the first two rows from it:

```
SELECT *
FROM file('test.csv', 'CSV', 'column1 UInt32, column2 UInt32, column3 UInt32')
LIMIT 2
```

column1	column2	column3
1	2	3
3	2	1

```
-- getting the first 10 lines of a table that contains 3 columns of UInt32 type from a CSV file
SELECT * FROM file('test.csv', 'CSV', 'column1 UInt32, column2 UInt32, column3 UInt32') LIMIT 10
```

## merge

`merge(db_name, 'tables_regexp')` – Creates a temporary Merge table. For more information, see the section "Table engines, Merge".

The table structure is taken from the first table encountered that matches the regular expression.

## numbers

`numbers(N)` – Returns a table with the single 'number' column (UInt64) that contains integers from 0 to N-1.  
`numbers(N, M)` – Returns a table with the single 'number' column (UInt64) that contains integers from N to (N + M - 1).

Similar to the `system.numbers` table, it can be used for testing and generating successive values, `numbers(N, M)` more efficient than `system.numbers`.

The following queries are equivalent:

```
SELECT * FROM numbers(10);
SELECT * FROM numbers(0, 10);
SELECT * FROM system.numbers LIMIT 10;
```

Examples:

```
-- Generate a sequence of dates from 2010-01-01 to 2010-12-31
select toDate('2010-01-01') + number as d FROM numbers(365);
```

## remote, remoteSecure

Allows you to access remote servers without creating a `Distributed` table.

Signatures:

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

`addresses_expr` – An expression that generates addresses of remote servers. This may be just one server address. The server address is `host:port`, or just `host`. The host can be specified as the server name, or as the IPv4 or IPv6 address. An IPv6 address is specified in square brackets. The port is the TCP port on the remote server. If the port is omitted, it uses `tcp_port` from the server's config file (by default, 9000).

Important

important

The port is required for an IPv6 address.

Examples:

```
example01-01-1
example01-01-1:9000
localhost
127.0.0.1
[::]:9000
[2a02:6b8:0:1111::11]:9000
```

Multiple addresses can be comma-separated. In this case, ClickHouse will use distributed processing, so it will send the query to all specified addresses (like to shards with different data).

Example:

```
example01-01-1,example01-02-1
```

Part of the expression can be specified in curly brackets. The previous example can be written as follows:

```
example01-0{1,2}-1
```

Curly brackets can contain a range of numbers separated by two dots (non-negative integers). In this case, the range is expanded to a set of values that generate shard addresses. If the first number starts with zero, the values are formed with the same zero alignment. The previous example can be written as follows:

```
example01-{01..02}-1
```

If you have multiple pairs of curly brackets, it generates the direct product of the corresponding sets.

Addresses and parts of addresses in curly brackets can be separated by the pipe symbol (|). In this case, the corresponding sets of addresses are interpreted as replicas, and the query will be sent to the first healthy replica. However, the replicas are iterated in the order currently set in the [load\\_balancing](#) setting.

Example:

```
example01-{01..02}-{1|2}
```

This example specifies two shards that each have two replicas.

The number of addresses generated is limited by a constant. Right now this is 1000 addresses.

Using the `remote` table function is less optimal than creating a `Distributed` table, because in this case, the server connection is re-established for every request. In addition, if host names are set, the names are resolved, and errors are not counted when working with various replicas. When processing a large number of queries, always create the `Distributed` table ahead of time, and don't use the `remote` table function.

The `remote` table function can be useful in the following cases:

- Accessing a specific server for data comparison, debugging, and testing.
- Queries between various ClickHouse clusters for research purposes.
- Infrequent distributed requests that are made manually.
- Distributed requests where the set of servers is re-defined each time.

If the user is not specified, `default` is used.

If the password is not specified, an empty password is used.

`remoteSecure` - same as `remote` but with secured connection. Default port — `tcp_port_secure` from config or 9440.

## url

`url(URL, format, structure)` - returns a table created from the `URL` with given `format` and `structure`.

URL - HTTP or HTTPS server address, which can accept `GET` and/or `POST` requests.

format - `format` of the data.

structure - table structure in `'UserID UInt64, Name String'` format. Determines column names and types.

### Example

```
-- getting the first 3 lines of a table that contains columns of String and UInt32 type from HTTP-server which answers in CSV format.
SELECT * FROM url('http://127.0.0.1:12345/', CSV, 'column1 String, column2 UInt32') LIMIT 3
```

## mysql

Allows to perform `SELECT` queries on data that is stored on a remote MySQL server.

```
mysql('host:port', 'database', 'table', 'user', 'password'[, replace_query, 'on_duplicate_clause']);
```

### Parameters

- `host:port` — MySQL server address.
- `database` — Remote database name.
- `table` — Remote table name.
- `user` — MySQL user.
- `password` — User password.
- `replace_query` — If `replace_query=1`, the `REPLACE` query will be performed instead of `INSERT`.
- `on_duplicate_clause` — The `ON DUPLICATE KEY on_duplicate_clause` expression that is added to the `INSERT` query.

Example: `INSERT INTO t (c1,c2) VALUES ('a', 2) ON DUPLICATE KEY UPDATE c2 = c2 + 1`, where `on_duplicate_clause` is `UPDATE c2 = c2 + 1`. See MySQL documentation to find which `on_duplicate_clause` you can use with `ON DUPLICATE KEY` clause.

To specify `on_duplicate_clause` you need to pass `0` to the `replace_query` parameter. If you simultaneously pass `replace_query = 1` and `on_duplicate_clause`, ClickHouse generates an exception.

At this time, simple `WHERE` clauses such as `=`, `!=`, `>`, `>=`, `<`, `<=` are executed on the MySQL server.

The rest of the conditions and the `LIMIT` sampling constraint are executed in ClickHouse only after the query to MySQL finishes.

### Returned Value

A table object with the same columns as the original MySQL table.

## Usage Example

Table in MySQL:



## Table in MySQL.

```
mysql> CREATE TABLE `test`.`test` (
-> `int_id` INT NOT NULL AUTO_INCREMENT,
-> `int_nullable` INT NULL DEFAULT NULL,
-> `float` FLOAT NOT NULL,
-> `float_nullable` FLOAT NULL DEFAULT NULL,
-> PRIMARY KEY (`int_id`));
Query OK, 0 rows affected (0,09 sec)

mysql> insert into test (`int_id`, `float`) VALUES (1,2);
Query OK, 1 row affected (0,00 sec)
```

```
mysql> select * from test;
+-----+-----+-----+-----+
| int_id | int_nullable | float | float_nullable |
+-----+-----+-----+-----+
| 1 | NULL | 2 | NULL |
+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

Selection of the data from ClickHouse:

```
SELECT * FROM mysql('localhost:3306', 'test', 'test', 'bayonet', '123')
```

```
┌int_id┐┌int_nullable┐┌float┐┌float_nullable┐
└ 1 ┘└ NULL ┘└ 2 ┘└ NULL ┘
```

## See Also

- [The 'MySQL' table engine](#)
- [Using MySQL as a source of external dictionary](#)

## jdbc

`jdbc(jdbc_connection_uri, schema, table)` - returns table that is connected via JDBC driver.

This table function requires separate `clickhouse-jdbc-bridge` program to be running.  
It supports Nullable types (based on DDL of remote table that is queried).

### Examples

```
SELECT * FROM jdbc('jdbc:mysql://localhost:3306/?user=root&password=root', 'schema', 'table')
```

```
SELECT * FROM jdbc('mysql://localhost:3306/?user=root&password=root', 'schema', 'table')
```

```
SELECT * FROM jdbc('datasource://mysql-local', 'schema', 'table')
```

## odbc

Returns table that is connected via [ODBC](#).

```
odbc(connection settings, external database, external table)
```

- To implement ODBC connection safely, ClickHouse uses the separate program `clickhouse-odbc-bridge`. If the ODBC driver is loaded directly from the `clickhouse-server` program, the problems in the driver can crash the ClickHouse server. ClickHouse starts the `clickhouse-odbc-bridge` program automatically when it is required. The ODBC bridge program is installed by the same package as the `clickhouse-server`.

## Usage example

This example is for linux Ubuntu 18.04 and MySQL server 5.7.

By default (if installed from packages) ClickHouse starts on behalf of the user `clickhouse`. Thus you need to create and configure this user at MySQL server.

```
sudo mysql
mysql> CREATE USER 'clickhouse'@'localhost' IDENTIFIED BY 'clickhouse';
mysql> GRANT ALL PRIVILEGES ON *.* TO 'clickhouse'@'clickhouse' WITH GRANT OPTION;
```

```
$ cat /etc/odbc.ini
[mysqlconn]
DRIVER = /usr/local/lib/libmyodbc5w.so
SERVER = 127.0.0.1
PORT = 3306
DATABASE = test
USERNAME = clickhouse
PASSWORD = clickhouse
```

```
isql -v mysqlconn
+-----+
| Connected! |
| |
...
```

```
mysql> CREATE TABLE `test`.`test` (
-> `int_id` INT NOT NULL AUTO_INCREMENT,
-> `int_nullable` INT NULL DEFAULT NULL,
-> `float` FLOAT NOT NULL,
-> `float_nullable` FLOAT NULL DEFAULT NULL,
```

```

-> float_nullable FLOAT NULL DEFAULT NULL,
-> PRIMARY KEY (`int_id`));
Query OK, 0 rows affected (0,09 sec)

mysql> insert into test (`int_id`, `float`) VALUES (1,2);
Query OK, 1 row affected (0,00 sec)

mysql> select * from test;
+-----+-----+-----+-----+
| int_id | int_nullable | float | float_nullable |
+-----+-----+-----+-----+
| 1 | NULL | 2 | NULL |
+-----+-----+-----+-----+
1 row in set (0,00 sec)

```

Getting data from the MySQL table:

```
SELECT * FROM odbc('DSN=mysqlconn', 'test', 'test')
```

```

+-----+-----+-----+-----+
| int_id | int_nullable | float | float_nullable |
+-----+-----+-----+-----+
| 1 | 0 | 2 | 0 |
+-----+-----+-----+-----+

```

## See Also

- [ODBC external dictionaries](#)
- [ODBC table engine](#).

## Dictionaries

A dictionary is a mapping (`key -> attributes`) that is convenient for various types of reference lists.

ClickHouse supports special functions for working with dictionaries that can be used in queries. It is easier and more efficient to use dictionaries with functions than a `JOIN` with reference tables.

**NULL** values can't be stored in a dictionary.

ClickHouse supports:

- **Built-in dictionaries** with a specific **set of functions**.
- **Plug-in (external) dictionaries** with a **set of functions**.

## External Dictionaries

You can add your own dictionaries from various data sources. The data source for a dictionary can be a local text or executable file, an HTTP(s) resource, or another DBMS. For more information, see "[Sources for external dictionaries](#)".

ClickHouse:

- Fully or partially stores dictionaries in RAM.
- Periodically updates dictionaries and dynamically loads missing values. In other words, dictionaries can be loaded dynamically.

The configuration of external dictionaries is located in one or more files. The path to the configuration is specified in the `dictionaries_config` parameter.

Dictionaries can be loaded at server startup or at first use, depending on the `dictionaries_lazy_load` setting.

The dictionary config file has the following format:

```
<yandex>
 <comment>An optional element with any content. Ignored by the ClickHouse server.</comment>

 <!--Optional element. File name with substitutions-->
 <include_from>/etc/metrika.xml</include_from>

 <dictionary>
 <!-- Dictionary configuration -->
 </dictionary>

 ...

 <dictionary>
 <!-- Dictionary configuration -->
 </dictionary>
</yandex>
```

You can **configure** any number of dictionaries in the same file. The file format is preserved even if there is only one dictionary (i.e. `<yandex><dictionary> <!--configuration -> </dictionary></yandex>`).

See also "**Functions for working with external dictionaries**".

#### Attention

You can convert values for a small dictionary by describing it in a `SELECT` query (see the **transform** function). This functionality is not related to external dictionaries.

## Configuring an External Dictionary

The dictionary configuration has the following structure:

```
<dictionary>
 <name>dict_name</name>

 <source>
 <!-- Source configuration -->
 </source>

 <layout>
 <!-- Memory layout configuration -->
 </layout>

 <structure>
 <!-- Complex key configuration -->
 </structure>

 <lifetime>
 <!-- Lifetime of dictionary in memory -->
 </lifetime>
</dictionary>
```

- **name** – The identifier that can be used to access the dictionary. Use the characters `[a-zA-Z0-9_\-]`.
- **source** — Source of the dictionary.
- **layout** — Dictionary layout in memory.
- **structure** — Structure of the dictionary . A key and attributes that can be retrieved by this key.

- **lifetime** — Frequency of dictionary updates.

## Storing Dictionaries in Memory

There are a variety of ways to store dictionaries in memory.

We recommend **flat**, **hashed** and **complex\_key\_hashed**, which provide optimal processing speed.

Caching is not recommended because of potentially poor performance and difficulties in selecting optimal parameters. Read more in the section "**cache**".

There are several ways to improve dictionary performance:

- Call the function for working with the dictionary after **GROUP BY**.
- Mark attributes to extract as injective. An attribute is called injective if different attribute values correspond to different keys. So when **GROUP BY** uses a function that fetches an attribute value by the key, this function is automatically taken out of **GROUP BY**.

ClickHouse generates an exception for errors with dictionaries. Examples of errors:

- The dictionary being accessed could not be loaded.
- Error querying a **cached** dictionary.

You can view the list of external dictionaries and their statuses in the **system.dictionaries** table.

The configuration looks like this:

```
<yandex>
 <dictionary>
 ...
 <layout>
 <layout_type>
 <!-- layout settings -->
 </layout_type>
 </layout>
 ...
 </dictionary>
</yandex>
```

## Ways to Store Dictionaries in Memory

- **flat**
- **hashed**
- **cache**
- **range\_hashed**
- **complex\_key\_hashed**
- **complex\_key\_cache**
- **ip\_trie**

### flat

The dictionary is completely stored in memory in the form of flat arrays. How much memory does the dictionary use? The amount is proportional to the size of the largest key (in space used).

The dictionary key has the **UInt64** type and the value is limited to 500,000. If a larger key is discovered when creating the dictionary, ClickHouse throws an exception and does not create the dictionary.

All types of sources are supported. When updating, data (from a file or from a table) is read in its entirety.

This method provides the best performance among all available methods of storing the dictionary.

Configuration example:

```
<layout>
 <flat />
</layout>
```

## hashed

The dictionary is completely stored in memory in the form of a hash table. The dictionary can contain any number of elements with any identifiers. In practice, the number of keys can reach tens of millions of items.

All types of sources are supported. When updating, data (from a file or from a table) is read in its entirety.

Configuration example:

```
<layout>
 <hashed />
</layout>
```

## complex\_key\_hashed

This type of storage is for use with composite **keys**. Similar to `hashed`.

Configuration example:

```
<layout>
 <complex_key_hashed />
</layout>
```

## range\_hashed

The dictionary is stored in memory in the form of a hash table with an ordered array of ranges and their corresponding values.

This storage method works the same way as `hashed` and allows using date/time ranges in addition to the key, if they appear in the dictionary.

Example: The table contains discounts for each advertiser in the format:

```
+-----+-----+-----+-----+
| advertiser id | discount start date | discount end date | amount |
+=====+=====+=====+=====+
| 123 | 2015-01-01 | 2015-01-15 | 0.15 |
+-----+-----+-----+-----+
| 123 | 2015-01-16 | 2015-01-31 | 0.25 |
+-----+-----+-----+-----+
| 456 | 2015-01-01 | 2015-01-15 | 0.05 |
+-----+-----+-----+-----+
```

To use a sample for date ranges, define the `range_min` and `range_max` elements in the **structure**.

Example:

```
<structure>
 <id>
 <name>Id</name>
 </id>
```

```

<range_min>
 <name>first</name>
</range_min>
<range_max>
 <name>last</name>
</range_max>
...

```

To work with these dictionaries, you need to pass an additional date argument to the `dictGetT` function:

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

This function returns the value for the specified `ids` and the date range that includes the passed date.

Details of the algorithm:

- If the `id` is not found or a range is not found for the `id`, it returns the default value for the dictionary.
- If there are overlapping ranges, you can use any.
- If the range delimiter is `NULL` or an invalid date (such as 1900-01-01 or 2039-01-01), the range is left open. The range can be open on both sides.

Configuration example:

```

<yandex>
 <dictionary>

 ...

 <layout>
 <range_hashed />
 </layout>

 <structure>
 <id>
 <name>Abcdef</name>
 </id>
 <range_min>
 <name>StartDate</name>
 </range_min>
 <range_max>
 <name>EndDate</name>
 </range_max>
 <attribute>
 <name>XXXType</name>
 <type>String</type>
 <null_value />
 </attribute>
 </structure>

 </dictionary>
</yandex>

```

## cache

The dictionary is stored in a cache that has a fixed number of cells. These cells contain frequently used elements.

When searching for a dictionary, the cache is searched first. For each block of data, all keys that are not found in the cache or are outdated are requested from the source using `SELECT attrs... FROM db.table WHERE id`

IN (k1, k2, ...). The received data is then written to the cache.

For cache dictionaries, the expiration **lifetime** of data in the cache can be set. If more time than **lifetime** has passed since loading the data in a cell, the cell's value is not used, and it is re-requested the next time it needs to be used.

This is the least effective of all the ways to store dictionaries. The speed of the cache depends strongly on correct settings and the usage scenario. A cache type dictionary performs well only when the hit rates are high enough (recommended 99% and higher). You can view the average hit rate in the **system.dictionaries** table.

To improve cache performance, use a subquery with **LIMIT**, and call the function with the dictionary externally.

Supported **sources**: MySQL, ClickHouse, executable, HTTP.

Example of settings:

```
<layout>
 <cache>
 <!-- The size of the cache, in number of cells. Rounded up to a power of two. -->
 <size_in_cells>1000000000</size_in_cells>
 </cache>
</layout>
```

Set a large enough cache size. You need to experiment to select the number of cells:

1. Set some value.
2. Run queries until the cache is completely full.
3. Assess memory consumption using the **system.dictionaries** table.
4. Increase or decrease the number of cells until the required memory consumption is reached.

Warning

Do not use ClickHouse as a source, because it is slow to process queries with random reads.

## complex\_key\_cache

This type of storage is for use with composite **keys**. Similar to **cache**.

## ip\_trie

This type of storage is for mapping network prefixes (IP addresses) to metadata such as ASN.

Example: The table contains network prefixes and their corresponding AS number and country code:

```
+-----+-----+-----+
| prefix | asn | cca2 |
+=====+=====+=====+
| 202.79.32.0/20 | 17501 | NP |
+-----+-----+-----+
| 2620:0:870::/48 | 3856 | US |
+-----+-----+-----+
| 2a02:6b8:1::/48 | 13238 | RU |
+-----+-----+-----+
| 2001:db8::/32 | 65536 | ZZ |
+-----+-----+-----+
```

When using this type of layout, the structure must have a composite key.

Example:



```

<structure>
 <key>
 <attribute>
 <name>prefix</name>
 <type>String</type>
 </attribute>
 </key>
 <attribute>
 <name>asn</name>
 <type>UInt32</type>
 <null_value />
 </attribute>
 <attribute>
 <name>cca2</name>
 <type>String</type>
 <null_value>??</null_value>
 </attribute>
 ...

```

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**:

- For a text file, it checks the time of modification. If the time differs from the previously recorded time, the dictionary is updated.
- For MyISAM tables, the time of modification is checked using a `SHOW TABLE STATUS` query.
- Dictionaries from other sources are updated every time by default.

For MySQL (InnoDB), ODBC and ClickHouse sources, you can set up a query that will update the dictionaries only if they really changed, rather than each time. To do this, follow these steps:

- The dictionary table must have a field that always changes when the source data is updated.
- The settings of the source must specify a query that retrieves the changing field. The ClickHouse server interprets the query result as a row, and if this row has changed relative to its previous state, the dictionary is updated. Specify the query in the `<invalidate_query>` field in the settings for the **source**.

Example of settings:

```
<dictionary>
...
<odbc>
...
 <invalidate_query>SELECT update_time FROM dictionary_source where id = 1</invalidate_query>
</odbc>
...
</dictionary>
```

## Sources of External Dictionaries

An external dictionary can be connected from many different sources.

The configuration looks like this:

```
<yandex>
<dictionary>
...
<source>
 <source_type>
 <!-- Source configuration -->
 </source_type>
</source>
...
</dictionary>
...
</yandex>
```

The source is configured in the `source` section.

Types of sources (`source_type`):

- Local file
- Executable file
- HTTP(s)
- DBMS
  - MySQL
  - ClickHouse
  - MongoDB
  - ODBC

## Local File

Example of settings:

```
<source>
 <file>
 <path>/opt/dictionaries/os.tsv</path>
 <format>TabSeparated</format>
 </file>
</source>
```

Setting fields:

- `path` - The absolute path to the file.
- `format` - The file format. All the formats described in "[Formats](#)" are supported.

## Executable File

Working with executable files depends on [how the dictionary is stored in memory](#). If the dictionary is stored using `cache` and `complex_key_cache`, ClickHouse requests the necessary keys by sending a request to the executable file's STDIN. Otherwise, ClickHouse starts executable file and treats its output as dictionary data.

Example of settings:

```
<source>
 <executable>
 <command>cat /opt/dictionaries/os.tsv</command>
 <format>TabSeparated</format>
 </executable>
</source>
```

Setting fields:

- `command` - The absolute path to the executable file, or the file name (if the program directory is written to `PATH`).
- `format` - The file format. All the formats described in "[Formats](#)" are supported.

## HTTP(s)

Working with an HTTP(s) server depends on [how the dictionary is stored in memory](#). If the dictionary is stored using `cache` and `complex_key_cache`, ClickHouse requests the necessary keys by sending a request via the `POST` method.

Example of settings:

```
<source>
 <http>
 <url>http://[::1]/os.tsv</url>
 <format>TabSeparated</format>
 </http>
</source>
```

In order for ClickHouse to access an HTTPS resource, you must [configure openssl](#) in the server configuration.

Setting fields:

- `url` – The source URL.
- `format` – The file format. All the formats described in "[Formats](#)" are supported.

## ODBC

You can use this method to connect any database that has an ODBC driver.

Example of settings:

```
<odbc>
 <db>DatabaseName</db>
 <table>ShemaName.TableName</table>
 <connection_string>DSN=some_parameters</connection_string>
 <invalidate_query>SQL_QUERY</invalidate_query>
</odbc>
```

Setting fields:

- `db` – Name of the database. Omit it if the database name is set in the `<connection_string>` parameters.
- `table` – Name of the table and schema if exists.
- `connection_string` – Connection string.
- `invalidate_query` – Query for checking the dictionary status. Optional parameter. Read more in the section [Updating dictionaries](#).

ClickHouse receives quoting symbols from ODBC-driver and quote all settings in queries to driver, so it's necessary to set table name accordingly to table name case in database.

If you have a problems with encodings when using Oracle, see the corresponding [FAQ](#) article.

## Known vulnerability of the ODBC dictionary functionality

Attention

When connecting to the database through the ODBC driver connection parameter `Servename` can be substituted. In this case values of `USERNAME` and `PASSWORD` from `odbc.ini` are sent to the remote server and can be compromised.

### Example of insecure use

Let's configure unixODBC for PostgreSQL. Content of `/etc/odbc.ini`:

```
[gregtest]
Driver = /usr/lib/psqlodbca.so
Servename = localhost
PORT = 5432
DATABASE = test_db
```

```
##OPTION = 3
USERNAME = test
PASSWORD = test
```

If you then make a query such as

```
SELECT * FROM odbc('DSN=greptest;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.

- `port` – The port on the MySQL server. You can specify it for all replicas, or for each one individually (inside `<replica>` ).
- `user` – Name of the MySQL user. You can specify it for all replicas, or for each one individually (inside `<replica>` ).
- `password` – Password of the MySQL user. You can specify it for all replicas, or for each one individually (inside `<replica>` ).
- `replica` – Section of replica configurations. There can be multiple sections.
  - `replica/host` – The MySQL host.
  - \* `replica/priority` – The replica priority. When attempting to connect, ClickHouse traverses the replicas in order of priority. The lower the number, the higher the priority.
- `db` – Name of the database.
- `table` – Name of the table.
- `where` – The selection criteria. Optional parameter.
- `invalidate_query` – Query for checking the dictionary status. Optional parameter. Read more in the section [Updating dictionaries](#).

MySQL can be connected on a local host via sockets. To do this, set `host` and `socket`.

Example of settings:

```
<source>
<mysql>
 <host>localhost</host>
 <socket>/path/to/socket/file.sock</socket>
 <user>clickhouse</user>
 <password>qwerty</password>
 <db>db_name</db>
 <table>table_name</table>
 <where>id=10</where>
 <invalidate_query>SQL_QUERY</invalidate_query>
</mysql>
</source>
```

## ClickHouse

Example of settings:

```
<source>
<clickhouse>
 <host>example01-01-1</host>
 <port>9000</port>
 <user>default</user>
 <password></password>
 <db>default</db>
 <table>ids</table>
 <where>id=10</where>
</clickhouse>
</source>
```

Setting fields:

- `host` – The ClickHouse host. If it is a local host, the query is processed without any network activity. To improve fault tolerance, you can create a [Distributed](#) table and enter it in subsequent configurations.



improve read tolerance, you can create a [distributed](#) table and enter it in subsequent configurations.

- `port` – The port on the ClickHouse server.
- `user` – Name of the ClickHouse user.
- `password` – Password of the ClickHouse user.
- `db` – Name of the database.
- `table` – Name of the table.
- `where` – The selection criteria. May be omitted.
- `invalidate_query` – Query for checking the dictionary status. Optional parameter. Read more in the section [Updating dictionaries](#).

## MongoDB

Example of settings:

```
<source>
 <mongodb>
 <host>localhost</host>
 <port>27017</port>
 <user></user>
 <password></password>
 <db>test</db>
 <collection>dictionary_source</collection>
 </mongodb>
</source>
```

Setting fields:

- `host` – The MongoDB host.
- `port` – The port on the MongoDB server.
- `user` – Name of the MongoDB user.
- `password` – Password of the MongoDB user.
- `db` – Name of the database.
- `collection` – Name of the collection.

## Dictionary Key and Fields

The `<structure>` clause describes the dictionary key and fields available for queries.

Overall structure:

```
<dictionary>
 <structure>
 <id>
 <name>Id</name>
 </id>

 <attribute>
 <!-- Attribute parameters -->
 </attribute>

 ...

 </structure>
</dictionary>
```

Columns are described in the structure:

- `<id>` - [key column](#).
- `<attribute>` - [data column](#). There can be a large number of columns.

# Key

ClickHouse supports the following types of keys:

- Numeric key. UInt64. Defined in the tag `<id>` .
- Composite key. Set of values of different types. Defined in the tag `<key>` .

A structure can contain either `<id>` or `<key>` .

Warning

The key doesn't need to be defined separately in attributes.

## Numeric Key

Format: `UInt64` .

Configuration example:

```
<id>
 <name>Id</name>
</id>
```

Configuration fields:

- name - The name of the column with keys.

## Composite Key

The key can be a `tuple` from any types of fields. The **layout** in this case must be `complex_key_hashed` or `complex_key_cache` .

Tip

A composite key can consist of a single element. This makes it possible to use a string as the key, for instance.

The key structure is set in the element `<key>` . Key fields are specified in the same format as the dictionary **attributes**. Example:

```
<structure>
 <key>
 <attribute>
 <name>field1</name>
 <type>String</type>
 </attribute>
 <attribute>
 <name>field2</name>
 <type>UInt32</type>
 </attribute>
 ...
 </key>
 ...
```

For a query to the `dictGet*` function, a tuple is passed as the key. Example: `dictGetString('dict_name', 'attr_name', tuple('string for field1', num_for_field2))` .

## Attributes

Configuration example:

```

<structure>
...
 <attribute>
 <name>Name</name>
 <type>Type</type>
 <null_value></null_value>
 <expression>rand64()</expression>
 <hierarchical>true</hierarchical>
 <injective>true</injective>
 <is_object_id>true</is_object_id>
 </attribute>
</structure>

```

Configuration fields:

- `name` – The column name.
- `type` – The column type. Sets the method for interpreting data in the source. For example, for MySQL, the field might be `TEXT`, `VARCHAR`, or `BLOB` in the source table, but it can be uploaded as `String`.
- `null_value` – The default value for a non-existing element. In the example, it is an empty string.
- `expression` – The attribute can be an expression. The tag is not required.
- `hierarchical` – Hierarchical support. Mirrored to the parent identifier. By default, `false`.
- `injective` – Whether the `id -> attribute` image is injective. If `true`, then you can optimize the `GROUP BY` clause. By default, `false`.
- `is_object_id` – Whether the query is executed for a MongoDB document by `ObjectID`.

## Internal dictionaries

ClickHouse contains a built-in feature for working with a geobase.

This allows you to:

- Use a region's ID to get its name in the desired language.
- Use a region's ID to get the ID of a city, area, federal district, country, or continent.
- Check whether a region is part of another region.
- Get a chain of parent regions.

All the functions support "translocality," the ability to simultaneously use different perspectives on region ownership. For more information, see the section "Functions for working with Yandex.Metrica dictionaries".

The internal dictionaries are disabled in the default package.

To enable them, uncomment the parameters `path_to_regions_hierarchy_file` and `path_to_regions_names_files` in the server configuration file.

The geobase is loaded from text files.

Place the `regions_hierarchy*.txt` files into the `path_to_regions_hierarchy_file` directory. This configuration parameter must contain the path to the `regions_hierarchy.txt` file (the default regional hierarchy), and the other files (`regions_hierarchy_ua.txt`) must be located in the same directory.

Put the `regions_names_*.txt` files in the `path_to_regions_names_files` directory.

You can also create these files yourself. The file format is as follows:

`regions_hierarchy*.txt`: TabSeparated (no header), columns:

- region ID (`UInt32`)
- parent region ID (`UInt32`)
- region type (`UInt8`): 1 - continent, 3 - country, 4 - federal district, 5 - region, 6 - city; other types don't

have values

- population (UInt32) — optional column

regions\_names\_\*.txt: TabSeparated (no header), columns:

- region ID (UInt32)
- region name (String) — Can't contain tabs or line feeds, even escaped ones.

A flat array is used for storing in RAM. For this reason, IDs shouldn't be more than a million.

Dictionaries can be updated without restarting the server. However, the set of available dictionaries is not updated.

For updates, the file modification times are checked. If a file has changed, the dictionary is updated.

The interval to check for changes is configured in the builtin\_dictionaries\_reload\_interval parameter.

Dictionary updates (other than loading at first use) do not block queries. During updates, queries use the old versions of dictionaries. If an error occurs during an update, the error is written to the server log, and queries continue using the old version of dictionaries.

We recommend periodically updating the dictionaries with the geobase. During an update, generate new files and write them to a separate location. When everything is ready, rename them to the files used by the server.

There are also functions for working with OS identifiers and Yandex.Metrica search engines, but they shouldn't be used.

## 操作符

所有的操作符（运算符）都会在查询时依据他们的优先级及其结合顺序在被解析时转换为对应的函数。下面按优先级从高到低列出各组运算符及其对应的函数：

### 下标运算符

$a[N]$  - 数组中的第N个元素；对应函数 `arrayElement(a, N)`

$a.N$  - 元组中第N个元素；对应函数 `tupleElement(a, N)`

### 负号

$-a$  - 对应函数 `negate(a)`

### 乘号、除号和取余

$a * b$  - 对应函数 `multiply(a, b)`

$a / b$  - 对应函数 `divide(a, b)`

$a \% b$  - 对应函数 `modulo(a, b)`

### 加号和减号

$a + b$  - 对应函数 `plus(a, b)`

$a - b$  - 对应函数 `minus(a, b)`

### 关系运算符

$a = b$  - 对应函数 `equals(a, b)`

$a == b$  - 对应函数 `equals(a, b)`

$a != b$  - 对应函数 `notEquals(a, b)`

$a < b$  - 对应函数 `notEquals(a, b)`

`a <> b` - 对应函数 `notEquals(a, b)`

`a <= b` - 对应函数 `lessOrEquals(a, b)`

`a >= b` - 对应函数 `greaterOrEquals(a, b)`

`a < b` - 对应函数 `less(a, b)`

`a > b` - 对应函数 `greater(a, b)`

`a LIKE s` - 对应函数 `like(a, b)`

`a NOT LIKE s` - 对应函数 `notLike(a, b)`

`a BETWEEN b AND c` - 等价于 `a >= b AND a <= c`

## 集合关系运算符

详见此节 [IN 相关操作符](#)。

`a IN ...` - 对应函数 `in(a, b)`

`a NOT IN ...` - 对应函数 `notIn(a, b)`

`a GLOBAL IN ...` - 对应函数 `globalIn(a, b)`

`a GLOBAL NOT IN ...` - 对应函数 `globalNotIn(a, b)`

## 逻辑非

`NOT a` - 对应函数 `not(a)`

## 逻辑与

`a AND b` - 对应函数 `and(a, b)`

## 逻辑或

`a OR b` - 对应函数 `or(a, b)`

## 条件运算符

`a ? b : c` - 对应函数 `if(a, b, c)`

注意:

条件运算符会先计算表达式**b**和表达式**c**的值，再根据表达式**a**的真假，返回相应的值。如果表达式**b**和表达式**c**是 `arrayJoin()` 函数，则不管表达式**a**是真是假，每行都会被复制展开。

## CASE 条件表达式

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

如果指定了 `x`，该表达式会转换为 `transform(x, [a, ...], [b, ...], c)` 函数。否则转换为 `multIf(a, b, ..., c)`

如果该表达式中没有 `ELSE c` 子句，则默认值就是 `NULL`

但 `transform` 函数不支持 `NULL`

法拉运营经

过做心升何

`s1 || s2` - 对应函数 `concat(s1, s2)`

## 创建 Lambda 函数

`x -> expr` - 对应函数 `lambda(x, expr)`

接下来的这些操作符因为其本身是括号没有优先级：

## 创建数组

`[x1, ...]` - 对应函数 `array(x1, ...)`

## 创建元组

`(x1, x2, ...)` - 对应函数 `tuple(x2, x2, ...)`

## 结合方式

所有的同级操作符从左到右结合。例如，`1 + 2 + 3` 会转换成 `plus(plus(1, 2), 3)`。所以，有时他们会跟我们预期的不太一样。例如，`SELECT 4 > 2 > 3` 的结果是0。

为了高效，`and` 和 `or` 函数支持任意多参数，一连串的 `AND` 和 `OR` 运算符会转换成其对应的单个函数。

## 判断是否为 NULL

ClickHouse 支持 `IS NULL` 和 `IS NOT NULL`。

### IS NULL

- 对于 **Nullable** 类型的值，`IS NULL` 会返回：
  - `1` 值为 `NULL`
  - `0` 否则
- 对于其他类型的值，`IS NULL` 总会返回 `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

- 对于 **Nullable** 类型的值，`IS NOT NULL` 会返回：
  - `0` 值为 `NULL`
  - `1` 否则
- 对于其他类型的值，`IS NOT NULL` 总会返回 `1`

```
:) SELECT * FROM t_null WHERE y IS NOT NULL
```

```
SELECT *
FROM t_null
WHERE isNotNull(y)
```

```
┌x┐┌y┐
```

2	3
---	---

1 rows in set. Elapsed: 0.002 sec.

## Syntax

There are two types of parsers in the system: the full SQL parser (a recursive descent parser), and the data format parser (a fast stream parser).

In all cases except the `INSERT` query, only the full SQL parser is used.

The `INSERT` query uses both parsers:

```
INSERT INTO t VALUES (1, 'Hello, world'), (2, 'abc'), (3, 'def')
```

The `INSERT INTO t VALUES` fragment is parsed by the full parser, and the data `(1, 'Hello, world'), (2, 'abc'), (3, 'def')` is parsed by the fast stream parser. You can also turn on the full parser for the data by using the `input_format_values_interpret_expressions` setting. When `input_format_values_interpret_expressions = 1`, ClickHouse first tries to parse values with the fast stream parser. If it fails, ClickHouse tries to use the full parser for the data, treating it like an SQL `expression`.

Data can have any format. When a query is received, the server calculates no more than `max_query_size` bytes of the request in RAM (by default, 1 MB), and the rest is stream parsed.

This means the system doesn't have problems with large `INSERT` queries, like MySQL does.

When using the `Values` format in an `INSERT` query, it may seem that data is parsed the same as expressions in a `SELECT` query, but this is not true. The `Values` format is much more limited.

Next we will cover the full parser. For more information about format parsers, see the [Formats](#) section.

## Spaces

There may be any number of space symbols between syntactical constructions (including the beginning and end of a query). Space symbols include the space, tab, line feed, CR, and form feed.

## Comments

SQL-style and C-style comments are supported.

SQL-style comments: from `--` to the end of the line. The space after `--` can be omitted.

Comments in C-style: from `/*` to `*/`. These comments can be multiline. Spaces are not required here, either.

## Keywords

Keywords (such as `SELECT`) are not case-sensitive. Everything else (column names, functions, and so on), in contrast to standard SQL, is case-sensitive.

Keywords are not reserved (they are just parsed as keywords in the corresponding context). If you use `identifiers` the same as the keywords, enclose them into quotes. For example, the query `SELECT "FROM" FROM table_name` is valid if the table `table_name` has column with the name `"FROM"`.

## Identifiers

Identifiers are:

- Cluster, database, table, partition and column names.
- Functions.
- Data types.
- `Expression aliases`.

Identifiers can be quoted or non-quoted. It is recommended to use non-quoted identifiers.

Non-quoted identifiers must match the regex `^[a-zA-Z][0-9a-zA-Z]*$` and can not be equal to **keywords**.

Examples: `x`, `_1`, `X_y__Z123_`.

If you want to use identifiers the same as keywords or you want to use other symbols in identifiers, quote it using double quotes or backticks, for example, `"id"`, ``id``.

## Literals

There are: numeric, string, compound and `NULL` literals.

### Numeric

A numeric literal tries to be parsed:

- First as a 64-bit signed number, using the `strtoull` function.
- If unsuccessful, as a 64-bit unsigned number, using the `strtoll` function.
- If unsuccessful, as a floating-point number using the `strtod` function.
- Otherwise, an error is returned.

The corresponding value will have the smallest type that the value fits in.

For example, 1 is parsed as `UInt8`, but 256 is parsed as `UInt16`. For more information, see **Data types**.

Examples: `1`, `18446744073709551615`, `0xDEADBEEF`, `01`, `0.1`, `1e100`, `-1e-100`, `inf`, `nan`.

### String

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

The minimum set of characters that you need to escape in string literals: `'` and `\`. Single quote can be escaped with the single quote, literals `'It\'s'` and `'It"s'` are equal.

### Compound

Constructions are supported for arrays: `[1, 2, 3]` and tuples: `(1, 'Hello, world!', 2)`.

Actually, these are not literals, but expressions with the array creation operator and the tuple creation operator, respectively.

An array must consist of at least one item, and a tuple must have at least two items.

Tuples have a special purpose for use in the `IN` clause of a `SELECT` query. Tuples can be obtained as the result of a query, but they can't be saved to a database (with the exception of **Memory** tables).

### NULL

Indicates that the value is missing.

In order to store `NULL` in a table field, it must be of the **Nullable** type.

Depending on the data format (input or output), `NULL` may have a different representation. For more information, see the documentation for **data formats**.

There are many nuances to processing `NULL`. For example, if at least one of the arguments of a comparison operation is `NULL`, the result of this operation will also be `NULL`. The same is true for multiplication, addition, and other operations. For more information, read the documentation for each operation.

In queries, you can check `NULL` using the **IS NULL** and **IS NOT NULL** operators and the related functions `isNull` and `isNotNull`.

## Functions



Functions are written like an identifier with a list of arguments (possibly empty) in brackets. In contrast to standard SQL, the brackets are required, even for an empty arguments list. Example: `now()`. There are regular and aggregate functions (see the section "Aggregate functions"). Some aggregate functions can contain two lists of arguments in brackets. Example: `quantile (0.9) (x)`. These aggregate functions are called "parametric" functions, and the arguments in the first list are called "parameters". The syntax of aggregate functions without parameters is the same as for regular functions.

## Operators

Operators are converted to their corresponding functions during query parsing, taking their priority and associativity into account.

For example, the expression `1 + 2 * 3 + 4` is transformed to `plus(plus(1, multiply(2, 3)), 4)`.

## Data Types and Database Table Engines

Data types and table engines in the `CREATE` query are written the same way as identifiers or functions. In other words, they may or may not contain an arguments list in brackets. For more information, see the sections "Data types," "Table engines," and "CREATE".

## Expression Aliases

An alias is a user-defined name for an expression in a query.

```
expr AS alias
```

- `AS` — The keyword for defining aliases. You can define the alias for a table name or a column name in a `SELECT` clause without using the `AS` keyword.

For example, `SELECT table_name_alias.column_name FROM table_name table_name_alias`.

In the `CAST` function, the `AS` keyword has another meaning. See the description of the function.

- `expr` — Any expression supported by ClickHouse.

For example, `SELECT column_name * 2 AS double FROM some_table`.

- `alias` — Name for `expr`. Aliases should comply with the `identifiers` syntax.

For example, `SELECT "table t".column_name FROM table_name AS "table t"`.

## Notes on Usage

Aliases are global for a query or subquery and you can define an alias in any part of a query for any expression. For example, `SELECT (1 AS n) + 2, n`.

Aliases are not visible in subqueries and between subqueries. For example, while executing the query `SELECT (SELECT sum(b.a) + num FROM b) - a.a AS num FROM a` ClickHouse generates the exception `Unknown identifier: num`.

If an alias is defined for the result columns in the `SELECT` clause of a subquery, these columns are visible in the outer query. For example, `SELECT n + m FROM (SELECT 1 AS n, 2 AS m)`.

Be careful with aliases that are the same as column or table names. Let's consider the following example:

```
CREATE TABLE t
(
 a Int,
 b Int
)
ENGINE = TinyLog()
```

```
SELECT
 argMax(a, b),
 sum(b) AS b
FROM t
```

Received exception from server (version 18.14.17):

Code: 184. DB::Exception: Received from localhost:9000, 127.0.0.1. DB::Exception: Aggregate function sum(b) is found inside another aggregate function in query.

In this example, we declared table `t` with column `b`. Then, when selecting data, we defined the `sum(b) AS b` alias. As aliases are global, ClickHouse substituted the literal `b` in the expression `argMax(a, b)` with the expression `sum(b)`. This substitution caused the exception.

## Asterisk

In a `SELECT` query, an asterisk can replace the expression. For more information, see the section "SELECT".

## Expressions

An expression is a function, identifier, literal, application of an operator, expression in brackets, subquery, or asterisk. It can also contain an alias.

A list of expressions is one or more expressions separated by commas.

Functions and operators, in turn, can have expressions as arguments.

## Operations

## Requirements

### CPU

For installation from prebuilt deb packages, use a CPU with x86\_64 architecture and support for SSE 4.2 instructions. To run ClickHouse with processors that do not support SSE 4.2 or have AArch64 or PowerPC64LE architecture, you should build ClickHouse from sources.

ClickHouse implements parallel data processing and uses all the hardware resources available. When choosing a processor, take into account that ClickHouse works more efficiently at configurations with a large number of cores but a lower clock rate than at configurations with fewer cores and a higher clock rate. For example, 16 cores with 2600 MHz is preferable to 8 cores with 3600 MHz.

Use of **Turbo Boost** and **hyper-threading** technologies is recommended. It significantly improves performance with a typical load.

### RAM

We recommend to use a minimum of 4GB of RAM in order to perform non-trivial queries. The ClickHouse server can run with a much smaller amount of RAM, but it requires memory for processing queries.

The required volume of RAM depends on:

- The complexity of queries.
- The amount of data that is processed in queries.

To calculate the required volume of RAM, you should estimate the size of temporary data for **GROUP BY**, **DISTINCT**, **JOIN** and other operations you use.

ClickHouse can use external memory for temporary data. See **GROUP BY in External Memory** for details.

## Swap File

Disable the swap file for production environments.

## Storage Subsystem

You need to have 2GB of free disk space to install ClickHouse.

The volume of storage required for your data should be calculated separately. Assessment should include:

- Estimation of the data volume.

You can take a sample of the data and get the average size of a row from it. Then multiply the value by the number of rows you plan to store.

- The data compression coefficient.

To estimate the data compression coefficient, load a sample of your data into ClickHouse and compare the actual size of the data with the size of the table stored. For example, clickstream data is usually compressed by 6-10 times.

To calculate the final volume of data to be stored, apply the compression coefficient to the estimated data volume. If you plan to store data in several replicas, then multiply the estimated volume by the number of replicas.

## Network

If possible, use networks of 10G or higher class.

The network bandwidth is critical for processing distributed queries with a large amount of intermediate data. In addition, network speed affects replication processes.

## Software

ClickHouse is developed for the Linux family of operating systems. The recommended Linux distribution is Ubuntu. The `tzdata` package should be installed in the system.

ClickHouse can also work in other operating system families. See details in the [Getting started](#) section of the documentation.

## Monitoring

You can monitor:

- Utilization of hardware resources.
- ClickHouse server metrics.

## Resource Utilization

ClickHouse does not monitor the state of hardware resources by itself.

It is highly recommended to set up monitoring for:

- Load and temperature on processors.  
You can use `dmesg`, `turbostat` or other instruments.
- Utilization of storage system, RAM and network.

## ClickHouse Server Metrics

ClickHouse server has embedded instruments for self-state monitoring.

To track server events use server logs. See the [logger](#) section of the configuration file.

ClickHouse collects:

- Different metrics of how the server uses computational resources.
- Common statistics on query processing.

You can find metrics in the `system.metrics`, `system.events`, and `system.asynchronous_metrics` tables.

You can configure ClickHouse to export metrics to `Graphite`. See the `Graphite` section in the ClickHouse server configuration file. Before configuring export of metrics, you should set up Graphite by following their official guide <https://graphite.readthedocs.io/en/latest/install.html>.

Additionally, you can monitor server availability through the HTTP API. Send the `HTTP GET` request to `/`. If the server is available, it responds with `200 OK`.

To monitor servers in a cluster configuration, you should set the `max_replica_delay_for_distributed_queries` parameter and use the HTTP resource `/replicas-delay`. A request to `/replicas-delay` returns `200 OK` if the replica is available and is not delayed behind the other replicas. If a replica is delayed, it returns information about the gap.

## Troubleshooting

- [Installation](#)
- [Connecting to the server](#)
- [Query processing](#)
- [Efficiency of query processing](#)

## Installation

### You Cannot Get Deb Packages from ClickHouse Repository With apt-get

- Check firewall settings.
- If you cannot access the repository for any reason, download packages as described in the [Getting started](#) article and install them manually using the `sudo dpkg -i <packages>` command. You will also need the `tzdata` package.

## Connecting to the Server

Possible issues:

- The server is not running.
- Unexpected or wrong configuration parameters.

## Server Is Not Running

### Check if server is running

Command:

```
sudo service clickhouse-server status
```

If the server is not running, start it with the command:

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

### Check logs

The main log of `clickhouse-server` is in `/var/log/clickhouse-server/clickhouse-server.log` by default.

If the server started successfully, you should see the strings:

- `<Information> Application: starting up.` — Server started.
- `<Information> Application: Ready for connections.` — Server is running and ready for connections.

If `clickhouse-server` start failed with a configuration error, you should see the `<Error>` string with an error description. For example:

```
2019.01.11 15:23:25.549505 [45] {} <Error> ExternalDictionaries: Failed reloading 'event2id' external dictionary: Poco::Exception. Code: 1000, e.code() = 111, e.displayText() = Connection refused, e.what() = Connection refused
```

If you don't see an error at the end of the file, look through the entire file starting from the string:

```
<Information> Application: starting up.
```

If you try to start a second instance of `clickhouse-server` on the server, you see the following log:

```
2019.01.11 15:25:11.151730 [1] {} <Information> : Starting ClickHouse 19.1.0 with revision 54413
2019.01.11 15:25:11.154578 [1] {} <Information> Application: starting up
2019.01.11 15:25:11.156361 [1] {} <Information> StatusFile: Status file ./status already exists - unclean restart.
Contents:
PID: 8510
Started at: 2019-01-11 15:24:23
Revision: 54413

2019.01.11 15:25:11.156673 [1] {} <Error> Application: DB::Exception: Cannot lock file ./status. Another server
instance in same directory is already running.
2019.01.11 15:25:11.156682 [1] {} <Information> Application: shutting down
2019.01.11 15:25:11.156686 [1] {} <Debug> Application: Uninitializing subsystem: Logging Subsystem
2019.01.11 15:25:11.156716 [2] {} <Information> BaseDaemon: Stop SignalListener thread
```

## See system.d logs

If you don't find any useful information in `clickhouse-server` logs or there aren't any logs, you can view `system.d` logs using the command:

```
sudo journalctl -u clickhouse-server
```

## Start clickhouse-server in interactive mode

```
sudo -u clickhouse /usr/bin/clickhouse-server --config-file /etc/clickhouse-server/config.xml
```

This command starts the server as an interactive app with standard parameters of the autostart script. In this mode `clickhouse-server` prints all the event messages in the console.

## Configuration Parameters

Check:

- Docker settings.

If you run ClickHouse in Docker in an IPv6 network, make sure that `network=host` is set.

- Endpoint settings.

Check `listen_host` and `tcp_port` settings.

ClickHouse server accepts localhost connections only by default

ClickHouse server accepts localhost connections only by default.

- HTTP protocol settings.

Check protocol settings for the HTTP API.

- Secure connection settings.

Check:

- The `tcp_port_secure` setting.
- Settings for `SSL certificates`.

Use proper parameters while connecting. For example, use the `port_secure` parameter with `clickhouse_client`.

- User settings.

You might be using the wrong user name or password.

## Query Processing

If ClickHouse is not able to process the query, it sends an error description to the client. In the `clickhouse-client` you get a description of the error in the console. If you are using the HTTP interface, ClickHouse sends the error description in the response body. For example:

```
$ curl 'http://localhost:8123/' --data-binary "SELECT a"
Code: 47, e.displayText() = DB::Exception: Unknown identifier: a. Note that there are no tables (FROM clause) in your query, context: required_names: 'a' source_tables: table_aliases: private_aliases: column_aliases: public_columns: 'a' masked_columns: array_join_columns: source_columns: , e.what() = DB::Exception
```

If you start `clickhouse-client` with the `stack-trace` parameter, ClickHouse returns the server stack trace with the description of an error.

You might see a message about a broken connection. In this case, you can repeat the query. If the connection breaks every time you perform the query, check the server logs for errors.

## Efficiency of Query Processing

If you see that ClickHouse is working too slowly, you need to profile the load on the server resources and network for your queries.

You can use the `clickhouse-benchmark` utility to profile queries. It shows the number of queries processed per second, the number of rows processed per second, and percentiles of query processing times.

## Usage Recommendations

### CPU

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.

```
echo 'performance' | sudo tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
```

## CPU Limitations

Processors can overheat. Use `dmesg` to see if the CPU's clock rate was limited due to overheating.

The restriction can also be set externally at the datacenter level. You can use `turbostat` to monitor it under a load.

## RAM

For small amounts of data (up to ~200 GB compressed), it is best to use as much memory as the volume of data.

For large amounts of data and when processing interactive (online) queries, you should use a reasonable amount of RAM (128 GB or more) so the hot data subset will fit in the cache of pages.

Even for data volumes of ~50 TB per server, using 128 GB of RAM significantly improves query performance compared to 64 GB.

## 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:  $2 * \text{num\_devices} * \text{chunk\_size\_in\_bytes} / 4096$ .

A block size of 1025 KB is sufficient for all RAID configurations.  
Never set the block size too small or too large.

You can use RAID-0 on SSD.  
Regardless of RAID use, always use replication for data security.

Enable NCQ with a long queue. For HDD, choose the CFQ scheduler, and for SSD, choose noop. Don't reduce the 'readahead' setting.  
For HDD, enable the write cache.

## File System

Ext4 is the most reliable option. Set the mount options `noatime, nobarrier`.  
XFS is also suitable, but it hasn't been as thoroughly tested with ClickHouse.  
Most other file systems should also work fine. File systems with delayed allocation work better.

## Linux Kernel

Don't use an outdated Linux kernel.

## Network

If you are using IPv6, increase the size of the route cache.  
The Linux kernel prior to 3.2 had a multitude of problems with IPv6 implementation.

Use at least a 10 GB network, if possible. 1 Gb will also work, but it will be much worse for patching replicas with tens of terabytes of data, or for processing distributed queries with a large amount of intermediate data.

## ZooKeeper

You are probably already using ZooKeeper for other purposes. You can use the same installation of ZooKeeper, if it isn't already overloaded.

It's best to use a fresh version of ZooKeeper – 3.4.9 or later. The version in stable Linux distributions may be outdated.

You should never use manually written scripts to transfer data between different ZooKeeper clusters, because the result will be incorrect for sequential nodes. Never use the "zkcopy" utility for the same reason: <https://github.com/ksprojects/zkcopy/issues/15>

If you want to divide an existing ZooKeeper cluster into two, the correct way is to increase the number of its replicas and then reconfigure it as two independent clusters.

Do not run ZooKeeper on the same servers as ClickHouse. Because ZooKeeper is very sensitive for latency and ClickHouse may utilize all available system resources.

With the default settings, ZooKeeper is a time bomb:

The ZooKeeper server won't delete files from old snapshots and logs when using the default configuration (see `autopurge`), and this is the responsibility of the operator.

This bomb must be defused.

The ZooKeeper (3.5.1) configuration below is used in the Yandex.Metrica production environment as of May



20, 2017:

zoo.cfg:

```
http://hadoop.apache.org/zookeeper/docs/current/zookeeperAdmin.html

The number of milliseconds of each tick
tickTime=2000
The number of ticks that the initial
synchronization phase can take
initLimit=30000
The number of ticks that can pass between
sending a request and getting an acknowledgement
syncLimit=10

maxClientCnxns=2000

maxSessionTimeout=60000000
the directory where the snapshot is stored.
dataDir=/opt/zookeeper/{ { cluster['name'] } }/data
Place the dataLogDir to a separate physical disc for better performance
dataLogDir=/opt/zookeeper/{ { cluster['name'] } }/logs

autopurge.snapRetainCount=10
autopurge.purgeInterval=1

To avoid seeks ZooKeeper allocates space in the transaction log file in
blocks of preAllocSize kilobytes. The default block size is 64M. One reason
for changing the size of the blocks is to reduce the block size if snapshots
are taken more often. (Also, see snapCount).
preAllocSize=131072

Clients can submit requests faster than ZooKeeper can process them,
especially if there are a lot of clients. To prevent ZooKeeper from running
out of memory due to queued requests, ZooKeeper will throttle clients so that
there is no more than globalOutstandingLimit outstanding requests in the
system. The default limit is 1,000. ZooKeeper logs transactions to a
transaction log. After snapCount transactions are written to a log file a
snapshot is started and a new transaction log file is started. The default
snapCount is 10,000.
snapCount=3000000

If this option is defined, requests will be will logged to a trace file named
traceFile.year.month.day.
##traceFile=

Leader accepts client connections. Default value is "yes". The leader machine
coordinates updates. For higher update throughput at the slight expense of
read throughput the leader can be configured to not accept clients and focus
on coordination.
leaderServes=yes

standaloneEnabled=false
dynamicConfigFile=/etc/zookeeper-{ { cluster['name'] } }/conf/zoo.cfg.dynamic
```

Java version:

```
Java(TM) SE Runtime Environment (build 1.8.0_25-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
```

## JVM parameters:

```
NAME=zookeeper-{{ cluster['name'] }}
ZOO_CFG_DIR=/etc/$NAME/conf

TODO this is really ugly
How to find out, which jars are needed?
seems, that log4j requires the log4j.properties file to be in the classpath
CLASSPATH="$ZOO_CFG_DIR:/usr/build/classes:/usr/build/lib/*.jar:/usr/share/zookeeper/zookeeper-3.5.1-
metrika.jar:/usr/share/zookeeper/slf4j-log4j12-1.7.5.jar:/usr/share/zookeeper/slf4j-api-
1.7.5.jar:/usr/share/zookeeper/servlet-api-2.5-20081211.jar:/usr/share/zookeeper/netty-
3.7.0.Final.jar:/usr/share/zookeeper/log4j-1.2.16.jar:/usr/share/zookeeper/jline-2.11.jar:/usr/share/zookeeper/jetty-util-
6.1.26.jar:/usr/share/zookeeper/jetty-6.1.26.jar:/usr/share/zookeeper/javacc.jar:/usr/share/zookeeper/jackson-mapper-
asl-1.9.11.jar:/usr/share/zookeeper/jackson-core-asl-1.9.11.jar:/usr/share/zookeeper/commons-cli-
1.2.jar:/usr/src/java/lib/*.jar:/usr/etc/zookeeper"

ZOO_CFG="$ZOO_CFG_DIR/zoo.cfg"
ZOO_LOG_DIR=/var/log/$NAME
USER=zookeeper
GROUP=zookeeper
PIDDIR=/var/run/$NAME
PIDFILE=$PIDDIR/$NAME.pid
SCRIPTNAME=/etc/init.d/$NAME
JAVA=/usr/bin/java
ZOO_MAIN="org.apache.zookeeper.server.quorum.QuorumPeerMain"
ZOO_LOG4J_PROP="INFO,ROLLINGFILE"
JMXLOCALONLY=false
JAVA_OPTS="-Xms{{ cluster.get('xms','128M') }} \
-Xmx{{ cluster.get('xmx','1G') }} \
-Xloggc:/var/log/$NAME/zookeeper-gc.log \
-XX:+UseGCLogFileRotation \
-XX:NumberOfGCLogFiles=16 \
-XX:GCLogFileSize=16M \
-verbose:gc \
-XX:+PrintGCTimeStamps \
-XX:+PrintGCDateStamps \
-XX:+PrintGCDetails \
-XX:+PrintTenuringDistribution \
-XX:+PrintGCApplicationStoppedTime \
-XX:+PrintGCApplicationConcurrentTime \
-XX:+PrintSafepointStatistics \
-XX:+UseParNewGC \
-XX:+UseConcMarkSweepGC \
-XX:+CMSParallelRemarkEnabled"
```

## Salt init:

```
description "zookeeper-{{ cluster['name'] }} centralized coordination service"

start on runlevel [2345]
stop on runlevel [!2345]

respawn

limit nofile 8192 8192

pre-start script
[-r "/etc/zookeeper-{{ cluster['name'] }}/conf/environment"] || exit 0
. /etc/zookeeper-{{ cluster['name'] }}/conf/environment
```

```

[-d $ZOO_LOG_DIR] || mkdir -p $ZOO_LOG_DIR
chown $USER:$GROUP $ZOO_LOG_DIR
end script

script
. /etc/zookeeper-{{ cluster['name'] }}/conf/environment
[-r /etc/default/zookeeper] && . /etc/default/zookeeper
if [-z "$JMXDISABLE"]; then
 JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote -
Dcom.sun.management.jmxremote.local.only=$JMXLOCALONLY"
fi
exec start-stop-daemon --start -c $USER --exec $JAVA --name zookeeper-{{ cluster['name'] }} \
-- -cp $CLASSPATH $JAVA_OPTS -Dzookeeper.log.dir=${ZOO_LOG_DIR} \
-Dzookeeper.root.logger=${ZOO_LOG4J_PROP} $ZOOMAIN $ZOOCFG
end script

```

## ClickHouse Update

If ClickHouse was installed from deb packages, execute the following commands on the server:

```

sudo apt-get update
sudo apt-get install clickhouse-client clickhouse-server
sudo service clickhouse-server restart

```

If you installed ClickHouse using something other than the recommended deb packages, use the appropriate update method.

ClickHouse does not support a distributed update. The operation should be performed consecutively on each separate server. Do not update all the servers on a cluster simultaneously, or the cluster will be unavailable for some time.

## Access Rights

Users and access rights are set up in the user config. This is usually `users.xml`.

Users are recorded in the `users` section. Here is a fragment of the `users.xml` file:

```

<!-- 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_sha
 256_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\_regex> Regular expression for host names. For example, ^example\d\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](#) 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/metrica.xml`) containing the `networks` substitution. Here is a fragment of it:

<yandex>

...

<networks>

<ip>::/64</ip>

<ip>203.0.113.0/24</ip>

<ip>2001:DB8::/32</ip>

...

```
</networks>
</yandex>
```

You could define this list of networks directly in `users.xml`, or in a file in the `users.d` directory (for more information, see the section "[Configuration files](#)").

The config includes comments explaining how to open access from everywhere.

For use in production, only specify `ip` elements (IP addresses and their masks), since using `host` and `host_regex` might cause extra latency.

Next the user settings profile is specified (see the section "[Settings profiles](#)"). You can specify the default profile, `default`. The profile can have any name. You can specify the same profile for different users. The most important thing you can write in the settings profile is `readonly=1`, which ensures read-only access. Then specify the quota to be used (see the section "[Quotas](#)"). You can specify the default quota: `default`. It is set in the config by default to only count resource usage, without restricting it. The quota can have any name. You can specify the same quota for different users – in this case, resource usage is calculated for each user individually.

In the optional `<allow_databases>` section, you can also specify a list of databases that the user can access. By default, all databases are available to the user. You can specify the `default` database. In this case, the user will receive access to the database by default.

Access to the `system` database is always allowed (since this database is used for processing queries).

The user can get a list of all databases and tables in them by using `SHOW` queries or system tables, even if access to individual databases isn't allowed.

Database access is not related to the `readonly` setting. You can't grant full access to one database and `readonly` access to another one.

## Data Backup

While [replication](#) provides protection from hardware failures, it does not protect against human errors: accidental deletion of data, deletion of the wrong table or a table on the wrong cluster, and software bugs that result in incorrect data processing or data corruption. In many cases mistakes like these will affect all replicas. ClickHouse has built-in safeguards to prevent some types of mistakes — for example, by default [you can't just drop tables with a MergeTree-like engine containing more than 50 Gb of data](#). However, these safeguards don't cover all possible cases and can be circumvented.

In order to effectively mitigate possible human errors, you should carefully prepare a strategy for backing up and restoring your data **in advance**.

Each company has different resources available and business requirements, so there's no universal solution for ClickHouse backups and restores that will fit every situation. What works for one gigabyte of data likely won't work for tens of petabytes. There are a variety of possible approaches with their own pros and cons, which will be discussed below. It is a good idea to use several approaches instead of just one in order to compensate for their various shortcomings.

### Note

Keep in mind that if you backed something up and never tried to restore it, chances are that restore will not work properly when you actually need it (or at least it will take longer than business can tolerate). So whatever backup approach you choose, make sure to automate the restore process as well, and practice it on a spare ClickHouse cluster regularly.

## Duplicating Source Data Somewhere Else

Often data that is ingested into ClickHouse is delivered through some sort of persistent queue, such as

**Apache Kafka**. In this case it is possible to configure an additional set of subscribers that will read the same data stream while it is being written to ClickHouse and store it in cold storage somewhere. Most companies already have some default recommended cold storage, which could be an object store or a distributed filesystem like **HDFS**.

## Filesystem Snapshots

Some local filesystems provide snapshot functionality (for example, **ZFS**), but they might not be the best choice for serving live queries. A possible solution is to create additional replicas with this kind of filesystem and exclude them from the **Distributed** tables that are used for **SELECT** queries. Snapshots on such replicas will be out of reach of any queries that modify data. As a bonus, these replicas might have special hardware configurations with more disks attached per server, which would be cost-effective.

## clickhouse-copier

**clickhouse-copier** is a versatile tool that was initially created to re-shard petabyte-sized tables. It can also be used for backup and restore purposes because it reliably copies data between ClickHouse tables and clusters.

For smaller volumes of data, a simple `INSERT INTO ... SELECT ...` to remote tables might work as well.

## Manipulations with Parts

ClickHouse allows using the `ALTER TABLE ... FREEZE PARTITION ...` query to create a local copy of table partitions. This is implemented using hardlinks to the `/var/lib/clickhouse/shadow/` folder, so it usually does not consume extra disk space for old data. The created copies of files are not handled by ClickHouse server, so you can just leave them there: you will have a simple backup that doesn't require any additional external system, but it will still be prone to hardware issues. For this reason, it's better to remotely copy them to another location and then remove the local copies. Distributed filesystems and object stores are still a good options for this, but normal attached file servers with a large enough capacity might work as well (in this case the transfer will occur via the network filesystem or maybe **rsync**).

For more information about queries related to partition manipulations, see the **ALTER documentation**.

A third-party tool is available to automate this approach: **clickhouse-backup**.

## Configuration Files

The main server config file is `config.xml`. It resides in the `/etc/clickhouse-server/` directory.

Individual settings can be overridden in the `*.xml` and `*.conf` files in the `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` 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/settings.md`)).

Substitutions can also be performed from ZooKeeper. To do this, specify the attribute `from_zk =`

`"/path/to/node"`. The element value is replaced with the contents of the node at `/path/to/node` in ZooKeeper. You can also put an entire XML subtree on the ZooKeeper node and it will be fully inserted into the source element.

The `config.xml` file can specify a separate config with user settings, profiles, and quotas. The relative path to this config is set in the `'users_config'` element. By default, it is `users.xml`. If `users_config` is omitted, the user settings, profiles, and quotas are specified directly in `config.xml`.

In addition, `users_config` may have overrides in files from the `users_config.d` directory (for example, `users.d`) and substitutions. For example, you can have separate config file for each user like this:

```
$ cat /etc/clickhouse-server/users.d/alice.xml
<yandex>
 <users>
 <alice>
 <profile>analytics</profile>
 <networks>
 <ip>::/0</ip>
 </networks>
 <password_sha256_hex>...</password_sha256_hex>
 <quota>analytics</quota>
 </alice>
 </users>
</yandex>
```

For each config file, the server also generates `file-preprocessed.xml` files when starting. These files contain all the completed substitutions and overrides, and they are intended for informational use. If ZooKeeper substitutions were used in the config files but ZooKeeper is not available on the server start, the server loads the configuration from the preprocessed file.

The server tracks changes in config files, as well as files and ZooKeeper nodes that were used when performing substitutions and overrides, and reloads the settings for users and clusters on the fly. This means that you can modify the cluster, users, and their settings without restarting the server.

## Quotas

Quotas allow you to limit resource usage over a period of time, or simply track the use of resources. Quotas are set up in the user config. This is usually `'users.xml'`.

The system also has a feature for limiting the complexity of a single query. See the section "Restrictions on query complexity").

In contrast to query complexity restrictions, quotas:

- Place restrictions on a set of queries that can be run over a period of time, instead of limiting a single query.
- Account for resources spent on all remote servers for distributed query processing.

Let's look at the section of the `'users.xml'` file that defines quotas.

```
<!-- Quotas -->
<quotas>
 <!-- Quota name. -->
 <default>
 <!-- Restrictions for a time period. You can set many intervals with different restrictions. -->
 <interval>
 <!-- Length of the interval. -->
 <duration>3600</duration>
```

```

<!-- Unlimited. Just collect data for the specified time interval. -->
<queries>0</queries>
<errors>0</errors>
<result_rows>0</result_rows>
<read_rows>0</read_rows>
<execution_time>0</execution_time>
</interval>
</default>

```

By default, the quota just tracks resource consumption for each hour, without limiting usage. The resource consumption calculated for each interval is output to the server log after each request.

```

<statbox>
 <!-- Restrictions for a time period. You can set many intervals with different restrictions. -->
 <interval>
 <!-- Length of the interval. -->
 <duration>3600</duration>

 <queries>1000</queries>
 <errors>100</errors>
 <result_rows>1000000000</result_rows>
 <read_rows>100000000000</read_rows>
 <execution_time>900</execution_time>
 </interval>

 <interval>
 <duration>86400</duration>

 <queries>10000</queries>
 <errors>1000</errors>
 <result_rows>5000000000</result_rows>
 <read_rows>500000000000</read_rows>
 <execution_time>7200</execution_time>
 </interval>
</statbox>

```

For the 'statbox' quota, restrictions are set for every hour and for every 24 hours (86,400 seconds). The time interval is counted starting from an implementation-defined fixed moment in time. In other words, the 24-hour interval doesn't necessarily begin at midnight.

When the interval ends, all collected values are cleared. For the next hour, the quota calculation starts over.

Here are the amounts that can be restricted:

**queries** – The total number of requests.

**errors** – The number of queries that threw an exception.

**result\_rows** – The total number of rows given as the result.

**read\_rows** – The total number of source rows read from tables for running the query, on all remote servers.

**execution\_time** – The total query execution time, in seconds (wall time).

If the limit is exceeded for at least one time interval, an exception is thrown with a text about which restriction was exceeded, for which interval, and when the new interval begins (when queries can be sent again).

Quotas can use the "quota key" feature in order to report on resources for multiple keys independently. Here is an example of this:



```

<!-- For the global reports designer. -->
<web_global>
 <!-- keyed - The quota_key "key" is passed in the query parameter,
 and the quota is tracked separately for each key value.
 For example, you can pass a Yandex.Metrica username as the key,
 so the quota will be counted separately for each username.
 Using keys makes sense only if quota_key is transmitted by the program, not by a user.

 You can also write <keyed_by_ip /> so the IP address is used as the quota key.
 (But keep in mind that users can change the IPv6 address fairly easily.)
 -->
 <keyed />

```

The quota is assigned to users in the 'users' section of the config. See the section "Access rights".

For distributed query processing, the accumulated amounts are stored on the requestor server. So if the user goes to another server, the quota there will "start over".

When the server is restarted, quotas are reset.

## System tables

System tables are used for implementing part of the system's functionality, and for providing access to information about how the system is working.

You can't delete a system table (but you can perform DETACH).

System tables don't have files with data on the disk or files with metadata. The server creates all the system tables when it starts.

System tables are read-only.

They are located in the 'system' database.

## system.asynchronous\_metrics

Contain metrics used for profiling and monitoring.

They usually reflect the number of events currently in the system, or the total resources consumed by the system.

Example: The number of SELECT queries currently running; the amount of memory in use. `system.asynchronous_metrics` and `system.metrics` differ in their sets of metrics and how they are calculated.

## system.clusters

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

Columns:

```

cluster String — The cluster name.
shard_num UInt32 — The shard number in the cluster, starting from 1.
shard_weight UInt32 — The relative weight of the shard when writing data.
replica_num UInt32 — The replica number in the shard, starting from 1.
host_name String — The host name, as specified in the config.
String host_address — The host IP address obtained from DNS.
port UInt16 — The port to use for connecting to the server.
user String — The name of the user for connecting to the server.

```

## system.columns

Contains information about the columns in all the tables.

You can use this table to get information similar to the **DESCRIBE TABLE** query, but for multiple tables at once.

columns:

The `system.columns` table contains the following columns (the type of the corresponding column is shown in brackets):

- `database` (String) — Database name.
- `table` (String) — Table name.
- `name` (String) — Column name.
- `type` (String) — Column type.
- `default_kind` (String) — Expression type (`DEFAULT`, `MATERIALIZED`, `ALIAS`) for the default value, or an empty string if it is not defined.
- `default_expression` (String) — Expression for the default value, or an empty string if it is not defined.
- `data_compressed_bytes` (UInt64) — The size of compressed data, in bytes.
- `data_uncompressed_bytes` (UInt64) — The size of decompressed data, in bytes.
- `marks_bytes` (UInt64) — The size of marks, in bytes.
- `comment` (String) — The comment about column, or an empty string if it is not defined.
- `is_in_partition_key` (UInt8) — Flag that indicates whether the column is in partition expression.
- `is_in_sorting_key` (UInt8) — Flag that indicates whether the column is in sorting key expression.
- `is_in_primary_key` (UInt8) — Flag that indicates whether the column is in primary key expression.
- `is_in_sampling_key` (UInt8) — Flag that indicates whether the column is in sampling key expression.

## system.databases

This table contains a single String column called 'name' – the name of a database.

Each database that the server knows about has a corresponding entry in the table.

This system table is used for implementing the `SHOW DATABASES` query.

## system.detached\_parts

Contains information about detached parts of

`MergeTree` tables. The `reason` column specifies

why the part was detached. For user-detached parts, the reason is empty. Such

parts can be attached with `ALTER TABLE ATTACH PARTITION|PART`

command. For the description of other columns, see `system.parts`.

## system.dictionaries

Contains information about external dictionaries.

Columns:

- `name` String — Dictionary name.
- `type` String — Dictionary type: Flat, Hashed, Cache.
- `origin` String — Path to the configuration file that describes the dictionary.
- `attribute.names` Array(String) — Array of attribute names provided by the dictionary.
- `attribute.types` Array(String) — Corresponding array of attribute types that are provided by the dictionary.
- `has_hierarchy` UInt8 — Whether the dictionary is hierarchical.
- `bytes_allocated` UInt64 — The amount of RAM the dictionary uses.
- `hit_rate` Float64 — For cache dictionaries, the percentage of uses for which the value was in the cache.
- `element_count` UInt64 — The number of items stored in the dictionary.
- `load_factor` Float64 — The percentage full of the dictionary (for a hashed dictionary, the percentage filled in the hash table).
- `creation_time` DateTime — The time when the dictionary was created or last successfully reloaded.
- `last_exception` String — Text of the error that occurs when creating or reloading the dictionary if the dictionary couldn't be created.
- `source` String — Text describing the data source for the dictionary.

Note that the amount of memory used by the dictionary is not proportional to the number of items stored in it. So for flat and cached dictionaries, all the memory cells are pre-assigned, regardless of how full the

in server has and loaded elements; on the memory some are pre assigned, regardless of how full the dictionary actually is.

## system.events

Contains information about the number of events that have occurred in the system. This is used for profiling and monitoring purposes.

Example: The number of processed SELECT queries.

Columns: 'event String' - the event name, and 'value UInt64' - the quantity.

## system.functions

Contains information about normal and aggregate functions.

Columns:

- `name(String)` - The name of the function.
- `is_aggregate(UInt8)` - Whether the function is aggregate.

## system.graphite\_retentions

Contains information about parameters `graphite_rollup` which use in tables with `*GraphiteMergeTree` engines.

Columns:

- `config_name(String)` - `graphite_rollup` parameter name.
- `regexp(String)` - A pattern for the metric name.
- `function(String)` - The name of the aggregating function.
- `age(UInt64)` - The minimum age of the data in seconds.
- `precision(UInt64)` - How precisely to define the age of the data in seconds.
- `priority(UInt16)` - Pattern priority.
- `is_default(UInt8)` - Is pattern default or not.
- `Tables.database(Array(String))` - Array of databases names of tables, which use `config_name` parameter.
- `Tables.table(Array(String))` - Array of tables names, which use `config_name` parameter.

## system.merges

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

Columns:

- `database String` - The name of the database the table is in.
- `table String` - Table name.
- `elapsed Float64` - The time elapsed (in seconds) since the merge started.
- `progress Float64` - The percentage of completed work from 0 to 1.
- `num_parts UInt64` - The number of pieces to be merged.
- `result_part_name String` - The name of the part that will be formed as the result of merging.
- `is_mutation UInt8` - 1 if this process is a part mutation.
- `total_size_bytes_compressed UInt64` - The total size of the compressed data in the merged chunks.
- `total_size_marks UInt64` - The total number of marks in the merged parts.
- `bytes_read_uncompressed UInt64` - Number of bytes read, uncompressed.
- `rows_read UInt64` - Number of rows read.
- `bytes_written_uncompressed UInt64` - Number of bytes written, uncompressed.
- `rows_written UInt64` - Number of lines rows written.

## system.metrics

## system.numbers

This table contains a single UInt64 column named 'number' that contains almost all the natural numbers starting from zero.

You can use this table for tests, or if you need to do a brute force search.

Reads from this table are not parallelized.

## system.numbers\_mt

The same as 'system.numbers' but reads are parallelized. The numbers can be returned in any order.

Used for tests.

## system.one

This table contains a single row with a single 'dummy' UInt8 column containing the value 0.

This table is used if a SELECT query doesn't specify the FROM clause.

This is similar to the DUAL table found in other DBMSs.

## system.parts

Contains information about parts of MergeTree tables.

Each row describes one part of the data.

Columns:

- partition (String) – The partition name. To learn what a partition is, see the description of the ALTER query.

Formats:

- YYYYMM for automatic partitioning by month.
- any\_string when partitioning manually.

- name (String) – Name of the data part.
- active (UInt8) – Indicates whether the part is active. If a part is active, it is used in a table; otherwise, it will be deleted. Inactive data parts remain after merging.
- marks (UInt64) – The number of marks. To get the approximate number of rows in a data part, multiply marks by the index granularity (usually 8192).
- marks\_size (UInt64) – The size of the file with marks.
- rows (UInt64) – The number of rows.
- bytes (UInt64) – The number of bytes when compressed.
- modification\_time (DateTime) – The modification time of the directory with the data part. This usually corresponds to the time of data part creation.
- remove\_time (DateTime) – The time when the data part became inactive.
- refcount (UInt32) – The number of places where the data part is used. A value greater than 2 indicates that the data part is used in queries or merges.
- min\_date (Date) – The minimum value of the date key in the data part.
- max\_date (Date) – The maximum value of the date key in the data part.
- min\_block\_number (UInt64) – The minimum number of data parts that make up the current part after merging.
- max\_block\_number (UInt64) – The maximum number of data parts that make up the current part after merging.

merging.

- level (UInt32) – Depth of the merge tree. If a merge was not performed, level=0.
- primary\_key\_bytes\_in\_memory (UInt64) – The amount of memory (in bytes) used by primary key values.
- primary\_key\_bytes\_in\_memory\_allocated (UInt64) – The amount of memory (in bytes) reserved for primary key values.
- database (String) – Name of the database.
- table (String) – Name of the table.
- engine (String) – Name of the table engine without parameters.

## system.part\_log

The `system.part_log` table is created only if the `part_log` server setting is specified.

This table contains information about the events that occurred with the `data parts` in the `MergeTree` family tables. For instance, adding or merging data.

The `system.part_log` table contains the following columns:

- event\_type (Enum) — Type of the event that occurred with the data part. Can have one of the following values: NEW\_PART — inserting, MERGE\_PARTS — merging, DOWNLOAD\_PART — downloading, REMOVE\_PART — removing or detaching using DETACH PARTITION, MUTATE\_PART — updating.
- event\_date (Date) — Event date.
- event\_time (DateTime) — Event time.
- duration\_ms (UInt64) — Duration.
- database (String) — Name of the database the data part is in.
- table (String) — Name of the table the data part is in.
- part\_name (String) — Name of the data part.
- partition\_id (String) — ID of the partition that the data part was inserted to. The column takes the 'all' value if the partitioning is by tuple().
- rows (UInt64) — The number of rows in the data part.
- size\_in\_bytes (UInt64) — Size of the data part in bytes.
- merged\_from (Array(String)) — An array of names of the parts which the current part was made up from (after the merge).
- bytes\_uncompressed (UInt64) — Size of uncompressed bytes.
- read\_rows (UInt64) — The number of rows was read during the merge.
- read\_bytes (UInt64) — The number of bytes was read during the merge.
- error (UInt16) — The code number of the occurred error.
- exception (String) — Text message of the occurred error.

The `system.part_log` table is created after the first inserting data to the `MergeTree` table.

## system.processes

This system table is used for implementing the `SHOW PROCESSLIST` query.

Columns:

user String	- Name of the user who made the request. For distributed query processing, this is the user who helped the requestor server send the query to this server, not the user who made the distributed request on the requestor server.
address String	- The IP address the request was made from. The same for distributed processing.
elapsed Float64	- The time in seconds since request execution started.

`rows_read UInt64` - The number of rows read from the table. For distributed processing, on the requestor server, this is the total for all remote servers.

`bytes_read UInt64` - The number of uncompressed bytes read from the table. For distributed processing, on the requestor server, this is the total for all remote servers.

`total_rows_approx UInt64` - The approximation of the total number of rows that should be read. For distributed processing, on the requestor server, this is the total for all remote servers. It can be updated during request processing, when new sources to process become known.

`memory_usage UInt64` - How much memory the request uses. It might not include some types of dedicated memory.

`query String` - The query text. For INSERT, it doesn't include the data to insert.

`query_id String` - Query ID, if defined.

## system.query\_log

Contains information about queries execution. For each query, you can see processing start time, duration of processing, error message and other information.

### Note

The table doesn't contain input data for `INSERT` queries.

ClickHouse creates this table only if the `query_log` server parameter is specified. This parameter sets the logging rules. For example, a logging interval or name of a table the queries will be logged in.

To enable query logging, set the parameter `log_queries` to 1. For details, see the [Settings](#) section.

The `system.query_log` table registers two kinds of queries:

1. Initial queries, that were run directly by the client.
2. Child queries that were initiated by other queries (for distributed query execution). For such a kind of queries, information about the parent queries is shown in the `initial_*` columns.

### Columns:

- `type` (UInt8) — Type of event that occurred when executing the query. Possible values:
  - 1 — Successful start of query execution.
  - 2 — Successful end of query execution.
  - 3 — Exception before the start of query execution.
  - 4 — Exception during the query execution.
- `event_date` (Date) — Event date.
- `event_time` (DateTime) — Event time.
- `query_start_time` (DateTime) — Time of the query processing start.
- `query_duration_ms` (UInt64) — Duration of the query processing.
- `read_rows` (UInt64) — Number of read rows.
- `read_bytes` (UInt64) — Number of read bytes.
- `written_rows` (UInt64) — For `INSERT` queries, number of written rows. For other queries, the column value is 0.
- `written_bytes` (UInt64) — For `INSERT` queries, number of written bytes. For other queries, the column value is 0.
- `result_rows` (UInt64) — Number of rows in a result.
- `result_bytes` (UInt64) — Number of bytes in a result.
- `memory_usage` (UInt64) — Memory consumption by the query.
- `query` (String) — Query string.
- `exception` (String) — Exception message.

`initial_query_start_time` (DateTime) — Initial query start time. `initial_query_duration_ms` (UInt64) — Initial query duration. `initial_read_rows` (UInt64) — Initial read rows. `initial_read_bytes` (UInt64) — Initial read bytes. `initial_written_rows` (UInt64) — Initial written rows. `initial_written_bytes` (UInt64) — Initial written bytes. `initial_result_rows` (UInt64) — Initial result rows. `initial_result_bytes` (UInt64) — Initial result bytes. `initial_memory_usage` (UInt64) — Initial memory usage. `initial_query` (String) — Initial query string. `initial_exception` (String) — Initial exception message.

- `stack_trace` (String) — Stack trace (a list of methods called before the error occurred). An empty string, if the query is completed successfully.
- `is_initial_query` (UInt8) — Kind of query. Possible values:
  - 1 — Query was initiated by the client.
  - 0 — Query was initiated by another query for distributed query execution.
- `user` (String) — Name of the user initiated the current query.
- `query_id` (String) — ID of the query.
- `address` (FixedString(16)) — IP address the query was initiated from.
- `port` (UInt16) — A server port that was used to receive the query.
- `initial_user` (String) — Name of the user who run the parent query (for distributed query execution).
- `initial_query_id` (String) — ID of the parent query.
- `initial_address` (FixedString(16)) — IP address that the parent query was launched from.
- `initial_port` (UInt16) — A server port that was used to receive the parent query from the client.
- `interface` (UInt8) — Interface that the query was initiated from. Possible values:
  - 1 — TCP.
  - 2 — HTTP.
- `os_user` (String) — User's OS.
- `client_hostname` (String) — Server name that the `clickhouse-client` is connected to.
- `client_name` (String) — The `clickhouse-client` name.
- `client_revision` (UInt32) — Revision of the `clickhouse-client`.
- `client_version_major` (UInt32) — Major version of the `clickhouse-client`.
- `client_version_minor` (UInt32) — Minor version of the `clickhouse-client`.
- `client_version_patch` (UInt32) — Patch component of the `clickhouse-client` version.
- `http_method` (UInt8) — HTTP method initiated the query. Possible values:
  - 0 — The query was launched from the TCP interface.
  - 1 — GET method is used.
  - 2 — POST method is used.
- `http_user_agent` (String) — The UserAgent header passed in the HTTP request.
- `quota_key` (String) — The quota key specified in `quotas` setting.
- `revision` (UInt32) — ClickHouse revision.
- `thread_numbers` (Array(UInt32)) — Number of threads that are participating in query execution.
- `ProfileEvents.Names` (Array(String)) — Counters that measure the following metrics:
  - Time spent on reading and writing over the network.
  - Time spent on reading and writing to a disk.
  - Number of network errors.
  - Time spent on waiting when the network bandwidth is limited.
- `ProfileEvents.Values` (Array(UInt64)) — Values of metrics that are listed in the `ProfileEvents.Names` column.
- `Settings.Names` (Array(String)) — Names of settings that were changed when the client run a query. To enable logging of settings changing, set the `log_query_settings` parameter to 1.
- `Settings.Values` (Array(String)) — Values of settings that are listed in the `Settings.Names` column.

Each query creates one or two rows in the `query_log` table, depending on the status of the query:

1. If the query execution is successful, two events with types 1 and 2 are created (see the `type` column).
2. If the error occurred during the query processing, two events with types 1 and 4 are created.
3. If the error occurred before the query launching, a single event with type 3 is created.

By default, logs are added into the table at intervals of 7,5 seconds. You can set this interval in the `query_log` server setting (see the `flush_interval_milliseconds` parameter). To flush the logs forcibly from the memory buffer into the table, use the `SYSTEM FLUSH LOGS` query.

When the table is deleted manually, it will be automatically created on the fly. Note that all the previous logs will be deleted.

#### Note

The storage period for logs is unlimited; the logs aren't automatically deleted from the table. You need to

organize the removing of non-actual logs yourself.

You can specify an arbitrary partitioning key for the `system.query_log` table in the `query_log` server setting (see the `partition_by` parameter).

## system.replicas

Contains information and status for replicated tables residing on the local server.

This table can be used for monitoring. The table contains a row for every Replicated\* table.

Example:

```
SELECT *
FROM system.replicas
WHERE table = 'visits'
FORMAT Vertical
```

Row 1:

```
database: merge
table: visits
engine: ReplicatedCollapsingMergeTree
is_leader: 1
is_readonly: 0
is_session_expired: 0
future_parts: 1
parts_to_check: 0
zookeeper_path: /clickhouse/tables/01-06/visits
replica_name: example01-06-1.yandex.ru
replica_path: /clickhouse/tables/01-06/visits/replicas/example01-06-1.yandex.ru
columns_version: 9
queue_size: 1
inserts_in_queue: 0
merges_in_queue: 1
log_max_index: 596273
log_pointer: 596274
total_replicas: 2
active_replicas: 2
```

Columns:

```
database: Database name
table: Table name
engine: Table engine name
```

`is_leader`: Whether the replica is the leader.

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 is_session_expired
OR future_parts > 20
OR parts_to_check > 10
OR queue_size > 20
OR inserts_in_queue > 10
OR log_max_index - log_pointer > 10
OR total_replicas < 2
OR active_replicas < total_replicas
```

If this query doesn't return anything, it means that everything is fine.

## system.settings

Contains information about settings that are currently in use.

I.e. used for executing the query you are using to read from the `system.settings` table.

Columns:

```
name String — Setting name.
value String — Setting value.
changed UInt8 — Whether the setting was explicitly defined in the config or explicitly changed.
```

Example:

```
SELECT *
FROM system.settings
WHERE changed
```

name	value	changed
max_threads	8	1
use_uncompressed_cache	0	1
load_balancing	random	1
max_memory_usage	10000000000	1

## system.tables

Contains metadata of each table that the server knows about. Detached tables are not shown in `system.tables`.

This table contains the following columns (the type of the corresponding column is shown in brackets):

- `database` (String) — The name of database the table is in.
- `name` (String) — Table name.
- `engine` (String) — Table engine name (without parameters).
- `is_temporary` (UInt8) - Flag that indicates whether the table is temporary.
- `data_path` (String) - Path to the table data in the file system.
- `metadata_path` (String) - Path to the table metadata in the file system.
- `metadata_modification_time` (DateTime) - Time of latest modification of the table metadata.
- `dependencies_database` (Array(String)) - Database dependencies.
- `dependencies_table` (Array(String)) - Table dependencies (**MaterializedView** tables based on the current table).
- `create_table_query` (String) - The query that was used to create the table.
- `engine_full` (String) - Parameters of the table engine.
- `partition_key` (String) - The partition key expression specified in the table.
- `sorting_key` (String) - The sorting key expression specified in the table.

- `primary_key` (String) - The primary key expression specified in the table.
- `sampling_key` (String) - The sampling key expression specified in the table.

The `system.tables` is used in `SHOW TABLES` query implementation.

## system.zookeeper

The table does not exist if ZooKeeper is not configured. Allows reading data from the ZooKeeper cluster defined in the config.

The query must have a 'path' equality condition in the WHERE clause. This is the path in ZooKeeper for the children that you want to get data for.

The query `SELECT * FROM system.zookeeper WHERE path = '/clickhouse'` outputs data for all children on the `/clickhouse` node.

To output data for all root nodes, write `path = '/'`.

If the path specified in 'path' doesn't exist, an exception will be thrown.

Columns:

- `name` String — The name of the node.
- `path` String — The path to the node.
- `value` String — Node value.
- `dataLength` Int32 — Size of the value.
- `numChildren` Int32 — Number of descendants.
- `czxid` Int64 — ID of the transaction that created the node.
- `mzxid` Int64 — ID of the transaction that last changed the node.
- `pzxid` Int64 — ID of the transaction that last deleted or added descendants.
- `ctime` DateTime — Time of node creation.
- `mtime` DateTime — Time of the last modification of the node.
- `version` Int32 — Node version: the number of times the node was changed.
- `cversion` Int32 — Number of added or removed descendants.
- `aversion` Int32 — Number of changes to the ACL.
- `ephemeralOwner` Int64 — For ephemeral nodes, the ID of the session that owns this node.

Example:

```
SELECT *
FROM system.zookeeper
WHERE path = '/clickhouse/tables/01-08/visits/replicas'
FORMAT Vertical
```

Row 1:

```
name: example01-08-1.yandex.ru
value:
czxid: 932998691229
mzxid: 932998691229
ctime: 2015-03-27 16:49:51
mtime: 2015-03-27 16:49:51
version: 0
cversion: 47
aversion: 0
ephemeralOwner: 0
dataLength: 0
numChildren: 7
pzxid: 987021031383
path: /clickhouse/tables/01-08/visits/replicas
```

```
Row 2:
name: example01-08-2.yandex.ru
value:
czxid: 933002738135
mzxid: 933002738135
ctime: 2015-03-27 16:57:01
mtime: 2015-03-27 16:57:01
version: 0
cversion: 37
aversion: 0
ephemeralOwner: 0
dataLength: 0
numChildren: 7
pzxid: 987021252247
path: /clickhouse/tables/01-08/visits/replicas
```

## system.mutations

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

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

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

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

**create\_time** - When this mutation command was submitted for execution.

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

**parts\_to\_do** - The number of data parts that need to be mutated for the mutation to finish.

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

If there were problems with mutating some parts the following columns contain additional information:

**latest\_failed\_part** - The name of the most recent part that could not be mutated.

**latest\_fail\_time** - The time of the most recent part mutation failure.

**latest\_fail\_reason** - The exception message that caused the most recent part mutation failure.

## Server configuration parameters

This section contains descriptions of server settings that cannot be changed at the session or query level.

These settings are stored in the `config.xml` file on the ClickHouse server.

Other settings are described in the "**Settings**" section.

Before studying the settings, read the **Configuration files** section and note the use of substitutions (the `incl`

and optional attributes).

## Server settings

### builtin\_dictionaries\_reload\_interval

The interval in seconds before reloading built-in dictionaries.

ClickHouse reloads built-in dictionaries every x seconds. This makes it possible to edit dictionaries "on the fly" without restarting the server.

Default value: 3600.

#### Example

```
<builtin_dictionaries_reload_interval>3600</builtin_dictionaries_reload_interval>
```

## compression

Data compression settings.

#### Warning

Don't use it if you have just started using ClickHouse.

The configuration looks like this:

```
<compression>
 <case>
 <parameters/>
 </case>
 ...
</compression>
```

You can configure multiple sections `<case>`.

Block field `<case>`:

- `min_part_size` - The minimum size of a table part.
- `min_part_size_ratio` - The ratio of the minimum size of a table part to the full size of the table.
- `method` - Compression method. Acceptable values : `lz4` or `zstd` (experimental).

ClickHouse checks `min_part_size` and `min_part_size_ratio` and processes the `case` blocks that match these conditions. If none of the `<case>` matches, ClickHouse applies the `lz4` compression algorithm.

#### Example

```
<compression incl="clickhouse_compression">
 <case>
 <min_part_size>10000000000</min_part_size>
 <min_part_size_ratio>0.01</min_part_size_ratio>
 <method>zstd</method>
 </case>
</compression>
```

## default\_database

The default database.

To get a list of databases, use the **SHOW DATABASES** query.

### Example

```
<default_database>default</default_database>
```

## default\_profile

Default settings profile.

Settings profiles are located in the file specified in the parameter `user_config`.

### Example

```
<default_profile>default</default_profile>
```

## dictionaries\_config

The path to the config file for external dictionaries.

Path:

- Specify the absolute path or the path relative to the server config file.
- The path can contain wildcards \* and ?.

See also "**External dictionaries**".

### Example

```
<dictionaries_config>*_dictionary.xml</dictionaries_config>
```

## dictionaries\_lazy\_load

Lazy loading of dictionaries.

If `true`, then each dictionary is created on first use. If dictionary creation failed, the function that was using the dictionary throws an exception.

If `false`, all dictionaries are created when the server starts, and if there is an error, the server shuts down.

The default is `true`.

### Example

```
<dictionaries_lazy_load>true</dictionaries_lazy_load>
```

## format\_schema\_path

The path to the directory with the schemes for the input data, such as schemas for the **CapnProto** format.

### Example

```
<!-- Directory containing schema files for various input formats. -->
<format_schema_path>format_schemas/</format_schema_path>
```

## graphite

Sending data to [Graphite](#).

Settings:

- host – The Graphite server.
- port – The port on the Graphite server.
- interval – The interval for sending, in seconds.
- timeout – The timeout for sending data, in seconds.
- root\_path – Prefix for keys.
- metrics – Sending data from a `:ref:system_tables-system.metrics` table.
- events – Sending data from a `:ref:system_tables-system.events` table.
- asynchronous\_metrics – Sending data from a `:ref:system_tables-system.asynchronous_metrics` table.

You can configure multiple `<graphite>` clauses. For instance, you can use this for sending different data at different intervals.

### Example

```
<graphite>
 <host>localhost</host>
 <port>42000</port>
 <timeout>0.1</timeout>
 <interval>60</interval>
 <root_path>one_min</root_path>
 <metrics>true</metrics>
 <events>true</events>
 <asynchronous_metrics>true</asynchronous_metrics>
</graphite>
```

## graphite\_rollup

Settings for thinning data for Graphite.

For more details, see [GraphiteMergeTree](#).

### Example

```
<graphite_rollup_example>
 <default>
 <function>max</function>
 <retention>
 <age>0</age>
 <precision>60</precision>
 </retention>
 <retention>
 <age>3600</age>
 <precision>300</precision>
 </retention>
 <retention>
 <age>86400</age>
 <precision>3600</precision>
 </retention>
 </default>
</graphite_rollup_example>
```

## http\_port/https\_port

The port for connecting to the server over HTTP(s).

If `https_port` is specified, **openSSL** must be configured.

If `http_port` is specified, the openSSL configuration is ignored even if it is set.

### Example

```
<https>0000</https>
```

## http\_server\_default\_response

The page that is shown by default when you access the ClickHouse HTTP(s) server.

### Example

Opens `https://tabix.io/` when accessing `http://localhost: http_port`.

```
<http_server_default_response>
<![CDATA[<html ng-app="SMI2"><head><base href="http://ui.tabix.io/"></head><body><div ui-view=""
class="content-ui"></div><script src="http://loader.tabix.io/master.js"></script></body></html>]]>
</http_server_default_response>
```

## include\_from

The path to the file with substitutions.

For more information, see the section "**Configuration files**".

### Example

```
<include_from>/etc/metrica.xml</include_from>
```

## interserver\_http\_port

Port for exchanging data between ClickHouse servers.

### Example

```
<interserver_http_port>9009</interserver_http_port>
```

## interserver\_http\_host

The host name that can be used by other servers to access this server.

If omitted, it is defined in the same way as the `hostname-f` command.

Useful for breaking away from a specific network interface.

### Example

```
<interserver_http_host>example.yandex.ru</interserver_http_host>
```

## keep\_alive\_timeout

The number of seconds that ClickHouse waits for incoming requests before closing the connection. Defaults to 3 seconds.

### Example



```
<keep_alive_timeout>3</keep_alive_timeout>
```

## listen\_host

Restriction on hosts that requests can come from. If you want the server to answer all of them, specify `::`.

Examples:

```
<listen_host>::1</listen_host>
<listen_host>127.0.0.1</listen_host>
```

## logger

Logging settings.

Keys:

- `level` – Logging level. Acceptable values: `trace`, `debug`, `information`, `warning`, `error`.
- `log` – The log file. Contains all the entries according to `level`.
- `errorlog` – Error log file.
- `size` – Size of the file. Applies to `log` and `errorlog`. Once the file reaches `size`, ClickHouse archives and renames it, and creates a new log file in its place.
- `count` – The number of archived log files that ClickHouse stores.

### Example

```
<logger>
 <level>trace</level>
 <log>/var/log/clickhouse-server/clickhouse-server.log</log>
 <errorlog>/var/log/clickhouse-server/clickhouse-server.err.log</errorlog>
 <size>1000M</size>
 <count>10</count>
</logger>
```

Writing to the syslog is also supported. Config example:

```
<logger>
 <use_syslog>1</use_syslog>
 <syslog>
 <address>syslog.remote:10514</address>
 <hostname>myhost.local</hostname>
 <facility>LOG_LOCAL6</facility>
 <format>syslog</format>
 </syslog>
</logger>
```

Keys:

- `user_syslog` — Required setting if you want to write to the syslog.
- `address` — The host[:port] of syslogd. If omitted, the local daemon is used.
- `hostname` — Optional. The name of the host that logs are sent from.
- `facility` — **The syslog facility keyword** in uppercase letters with the "LOG\_" prefix: (`LOG_USER`, `LOG_DAEMON`, `LOG_LOCAL3`, and so on).  
Default value: `LOG_USER` if `address` is specified, `LOG_DAEMON` otherwise.
- `format` – Message format. Possible values: `bsd` and `syslog`.

## macros

Parameter substitutions for replicated tables.

Can be omitted if replicated tables are not used.

For more information, see the section "[Creating replicated tables](#)".

### Example

```
<macros incl="macros" optional="true" />
```

## mark\_cache\_size

Approximate size (in bytes) of the cache of "marks" used by [MergeTree](#).

The cache is shared for the server and memory is allocated as needed. The cache size must be at least 5368709120.

### Example

```
<mark_cache_size>5368709120</mark_cache_size>
```

## max\_concurrent\_queries

The maximum number of simultaneously processed requests.

### Example

```
<max_concurrent_queries>100</max_concurrent_queries>
```

## max\_connections

The maximum number of inbound connections.

### Example

```
<max_connections>4096</max_connections>
```

## max\_open\_files

The maximum number of open files.

By default: `maximum`.

We recommend using this option in Mac OS X, since the `getrlimit()` function returns an incorrect value.

### Example

```
<max_open_files>262144</max_open_files>
```

## max\_table\_size\_to\_drop

Restriction on deleting tables.

If the size of a [MergeTree](#) table exceeds `max_table_size_to_drop` (in bytes), you can't delete it using a DROP

query.

If you still need to delete the table without restarting the ClickHouse server, create the `<clickhouse-path>/flags/force_drop_table` file and run the DROP query.

Default value: 50 GB.

The value 0 means that you can delete all tables without any restrictions.

### Example

```
<max_table_size_to_drop>0</max_table_size_to_drop>
```

## merge\_tree

Fine tuning for tables in the [MergeTree](#).

For more information, see the MergeTreeSettings.h header file.

### Example

```
<merge_tree>
 <max_suspicious_broken_parts>5</max_suspicious_broken_parts>
</merge_tree>
```

## openssl

SSL client/server configuration.

Support for SSL is provided by the `libpoco` library. The interface is described in the file [SSLManager.h](#)

Keys for server/client settings:

- `privateKeyFile` – The path to the file with the secret key of the PEM certificate. The file may contain a key and certificate at the same time.
- `certificateFile` – The path to the client/server certificate file in PEM format. You can omit it if `privateKeyFile` contains the certificate.
- `caConfig` – The path to the file or directory that contains trusted root certificates.
- `verificationMode` – The method for checking the node's certificates. Details are in the description of the [Context](#) class. Possible values: `none`, `relaxed`, `strict`, `once`.
- `verificationDepth` – The maximum length of the verification chain. Verification will fail if the certificate chain length exceeds the set value.
- `loadDefaultCAFile` – Indicates that built-in CA certificates for OpenSSL will be used. Acceptable values: `true`, `false`.
- `cipherList` – Supported OpenSSL encryptions. For example: `ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH`.
- `cacheSessions` – Enables or disables caching sessions. Must be used in combination with `sessionIdContext`. Acceptable values: `true`, `false`.
- `sessionIdContext` – A unique set of random characters that the server appends to each generated identifier. The length of the string must not exceed `SSL_MAX_SSL_SESSION_ID_LENGTH`. This parameter is always recommended, since it helps avoid problems both if the server caches the session and if the client requested caching. Default value: `${application.name}`.
- `sessionCacheSize` – The maximum number of sessions that the server caches. Default value: `1024*20`. 0 – Unlimited sessions.
- `sessionTimeout` – Time for caching the session on the server.
- `extendedVerification` – Automatically extended verification of certificates after the session ends. Acceptable values: `true`, `false`.
- `requireTLSv1` – Require a TLSv1 connection. Acceptable values: `true`, `false`.

- `requireTLSv1_1` - Require a TLSv1.1 connection. Acceptable values: `true`, `false`.
- `requireTLSv1` - Require a TLSv1.2 connection. Acceptable values: `true`, `false`.
- `fips` - Activates OpenSSL FIPS mode. Supported if the library's OpenSSL version supports FIPS.
- `privateKeyPassphraseHandler` - Class (`PrivateKeyPassphraseHandler` subclass) that requests the passphrase for accessing the private key. For example: `<privateKeyPassphraseHandler>`, `<name>KeyFileHandler</name>`, `<options><password>test</password></options>`, `</privateKeyPassphraseHandler>`.
- `invalidCertificateHandler` - Class (subclass of `CertificateHandler`) for verifying invalid certificates. For example: `<invalidCertificateHandler>` `<name>ConsoleCertificateHandler</name>` `</invalidCertificateHandler>`.
- `disableProtocols` - Protocols that are not allowed to use.
- `preferServerCiphers` - Preferred server ciphers on the client.

### Example of settings:

```
<openssl>
 <server>
 <!-- openssl req -subj "/CN=localhost" -new -newkey rsa:2048 -days 365 -nodes -x509 -keyout /etc/clickhouse-
server/server.key -out /etc/clickhouse-server/server.crt -->
 <certificateFile>/etc/clickhouse-server/server.crt</certificateFile>
 <privateKeyFile>/etc/clickhouse-server/server.key</privateKeyFile>
 <!-- openssl dhparam -out /etc/clickhouse-server/dhparam.pem 4096 -->
 <dhParamsFile>/etc/clickhouse-server/dhparam.pem</dhParamsFile>
 <verificationMode>none</verificationMode>
 <loadDefaultCAFile>true</loadDefaultCAFile>
 <cacheSessions>true</cacheSessions>
 <disableProtocols>ssl2,ssl3</disableProtocols>
 <preferServerCiphers>true</preferServerCiphers>
 </server>
 <client>
 <loadDefaultCAFile>true</loadDefaultCAFile>
 <cacheSessions>true</cacheSessions>
 <disableProtocols>ssl2,ssl3</disableProtocols>
 <preferServerCiphers>true</preferServerCiphers>
 <!-- Use for self-signed: <verificationMode>none</verificationMode> -->
 <invalidCertificateHandler>
 <!-- Use for self-signed: <name>AcceptCertificateHandler</name> -->
 <name>RejectCertificateHandler</name>
 </invalidCertificateHandler>
 </client>
</openssl>
```

## part\_log

Logging events that are associated with **MergeTree**. For instance, adding or merging data. You can use the log to simulate merge algorithms and compare their characteristics. You can visualize the merge process.

Queries are logged in the **system.part\_log** table, not in a separate file. You can configure the name of this table in the `table` parameter (see below).

Use the following parameters to configure logging:

- `database` - Name of the database.
- `table` - Name of the system table.
- `partition_by` - Sets a **custom partitioning key**.
- `flush_interval_milliseconds` - Interval for flushing data from the buffer in memory to the table.

### Example

```
<part_log>
```

```
<database>system</database>
<table>part_log</table>
<partition_by>toMonday(event_date)</partition_by>
<flush_interval_milliseconds>7500</flush_interval_milliseconds>
</part_log>
```

## path

The path to the directory containing data.

Note

The trailing slash is mandatory.

### Example

```
<path>/var/lib/clickhouse/</path>
```

## query\_log

Setting for logging queries received with the `log_queries=1` setting.

Queries are logged in the `system.query_log` table, not in a separate file. You can change the name of the table in the `table` parameter (see below).

Use the following parameters to configure logging:

- `database` – Name of the database.
- `table` – Name of the system table the queries will be logged in.
- `partition_by` – Sets a **custom partitioning key** for a system table.
- `flush_interval_milliseconds` – Interval for flushing data from the buffer in memory to the table.

If the table doesn't exist, ClickHouse will create it. If the structure of the query log changed when the ClickHouse server was updated, the table with the old structure is renamed, and a new table is created automatically.

### Example

```
<query_log>
 <database>system</database>
 <table>query_log</table>
 <partition_by>toMonday(event_date)</partition_by>
 <flush_interval_milliseconds>7500</flush_interval_milliseconds>
</query_log>
```

## remote\_servers

Configuration of clusters used by the Distributed table engine.

For more information, see the section "**Table engines/Distributed**".

### Example

```
<remote_servers incl="clickhouse_remote_servers" />
```

For the value of the `incl` attribute, see the section "**Configuration files**".

## timezone

## timezone

The server's time zone.

Specified as an IANA identifier for the UTC time zone or geographic location (for example, Africa/Abidjan).

The time zone is necessary for conversions between String and DateTime formats when DateTime fields are output to text format (printed on the screen or in a file), and when getting DateTime from a string. In addition, the time zone is used in functions that work with the time and date if they didn't receive the time zone in the input parameters.

### Example

```
<timezone>Europe/Moscow</timezone>
```

## tcp\_port

Port for communicating with clients over the TCP protocol.

### Example

```
<tcp_port>9000</tcp_port>
```

## tcp\_port\_secure

TCP port for secure communication with clients. Use it with [OpenSSL](#) settings.

### Possible values

Positive integer.

### Default value

```
<tcp_port_secure>9440</tcp_port_secure>
```

## tmp\_path

Path to temporary data for processing large queries.

Note

The trailing slash is mandatory.

### Example

```
<tmp_path>/var/lib/clickhouse/tmp/</tmp_path>
```

## uncompressed\_cache\_size

Cache size (in bytes) for uncompressed data used by table engines from the [MergeTree](#).

There is one shared cache for the server. Memory is allocated on demand. The cache is used if the option [use\\_uncompressed\\_cache](#) is enabled.

The uncompressed cache is advantageous for very short queries in individual cases.

### Example

```
uncompressed_cache_size 8589934592 </uncompressed_cache_size>
```

```
<uncompressed_cache_size>8589934592</uncompressed_cache_size>
```

## user\_files\_path

The directory with user files. Used in the table function `file()`.

### Example

```
<user_files_path>/var/lib/clickhouse/user_files/</user_files_path>
```

## users\_config

Path to the file that contains:

- User configurations.
- Access rights.
- Settings profiles.
- Quota settings.

### Example

```
<users_config>users.xml</users_config>
```

## zookeeper

Contains settings that allow ClickHouse to interact with **ZooKeeper** cluster.

ClickHouse uses ZooKeeper for storing metadata of replicas when using replicated tables. If replicated tables are not used, this parameter section can be omitted.

This parameter section contains the following parameters:

- `node` — ZooKeeper endpoint. You can set a few endpoints.

For example:

```
xml <node index="1"> <host>example_host</host> <port>2181</port> </node>
```

The `index` attribute is not used in ClickHouse. The only reason for this attribute is to allow some other programs to use the same configuraton.

- `session_timeout_ms` — Maximum timeout for client session in milliseconds (default: 30000).
- `operation_timeout_ms` — Maximum timeout for operation in milliseconds (default: 10000).
- `root` — ZNode, that is used as root for znodes used by ClickHouse server. Optional.
- `identity` — User and password, required by ZooKeeper to give access to requested znodes. Optional.

### Example configuration

```
<zookeeper>
 <node>
 <host>example1</host>
 <port>2181</port>
 </node>
 <node>
 <host>example2</host>
 <port>2181</port>
 </node>
 <session_timeout_ms>30000</session_timeout_ms>
```

```
<operation_timeout_ms>10000</operation_timeout_ms>
<!-- Optional. Chroot suffix. Should exist. -->
<root>/path/to/zookeeper/node</root>
<!-- Optional. Zookeeper digest ACL string. -->
<identity>user:password</identity>
</zookeeper>
```

## See Also

- [Replication](#)
- [ZooKeeper Programmer's Guide](#)

# use\_minimalistic\_part\_header\_in\_zookeeper

Storage method for data part headers in ZooKeeper.

This setting only applies to the `MergeTree` family. It can be specified:

- Globally in the `merge_tree` section of the `config.xml` file.

ClickHouse uses the setting for all the tables on the server. You can change the setting at any time. Existing tables change their behavior when the setting changes.

- For each individual table.

When creating a table, specify the corresponding [engine setting](#). The behavior of an existing table with this setting does not change, even if the global setting changes.

## Possible values

- 0 — Functionality is turned off.
- 1 — Functionality is turned on.

If `use_minimalistic_part_header_in_zookeeper = 1`, then [replicated](#) tables store the headers of the data parts compactly using a single `znode`. If the table contains many columns, this storage method significantly reduces the volume of the data stored in Zookeeper.

## Attention

After applying `use_minimalistic_part_header_in_zookeeper = 1`, you can't downgrade the ClickHouse server to a version that doesn't support this setting. Be careful when upgrading ClickHouse on servers in a cluster. Don't upgrade all the servers at once. It is safer to test new versions of ClickHouse in a test environment, or on just a few servers of a cluster.

Data part headers already stored with this setting can't be restored to their previous (non-compact) representation.

**Default value:** 0.

# Settings

There are multiple ways to make all the settings described below.

Settings are configured in layers, so each subsequent layer redefines the previous settings.

Ways to configure settings, in order of priority:

- Settings in the `users.xml` server configuration file.

Set in the element `<profiles>`.

- Session settings.



Send `SET setting=value` from the ClickHouse console client in interactive mode.

Similarly, you can use ClickHouse sessions in the HTTP protocol. To do this, you need to specify the `session_id` HTTP parameter.

- Query settings.
  - When starting the ClickHouse console client in non-interactive mode, set the startup parameter `--setting=value`.
  - When using the HTTP API, pass CGI parameters (`URL?setting_1=value&setting_2=value...`).

Settings that can only be made in the server config file are not covered in this section.

## Permissions for queries

Queries in ClickHouse can be divided into several types:

1. Read data queries: `SELECT`, `SHOW`, `DESCRIBE`, `EXISTS`.
2. Write data queries: `INSERT`, `OPTIMIZE`.
3. Change settings queries: `SET`, `USE`.
4. **DDL** queries: `CREATE`, `ALTER`, `RENAME`, `ATTACH`, `DETACH`, `DROP`, `TRUNCATE`.
5. `KILL QUERY`.

The following settings regulate user permissions by the type of query:

- **readonly** — Restricts permissions for all types of queries except DDL queries.
- **allow\_ddl** — Restricts permissions for DDL queries.

`KILL QUERY` can be performed with any settings.

### readonly

Restricts permissions for read data, write data and change settings queries.

See how the queries are divided into types **above**.

#### Possible values

- 0 — All queries are allowed.
- 1 — Only read data queries are allowed.
- 2 — Read data and change settings queries are allowed.

After setting `readonly = 1`, the user can't change `readonly` and `allow_ddl` settings in the current session.

When using the `GET` method in the **HTTP interface**, `readonly = 1` is set automatically. To modify data, use the `POST` method.

#### Default value

0

### allow\_ddl

Allows/denies **DDL** queries.

See how the queries are divided into types **above**.

#### Possible values

- 0 — DDL queries are not allowed.
- 1 — DDL queries are allowed.

You cannot execute `SET allow_ddl = 1` if `allow_ddl = 0` for the current session.

## Default value

1

# Restrictions on query complexity

Restrictions on query complexity are part of the settings.

They are used in order to provide safer execution from the user interface.

Almost all the restrictions only apply to SELECTs. For distributed query processing, restrictions are applied on each server separately.

Restrictions on the "maximum amount of something" can take the value 0, which means "unrestricted".

Most restrictions also have an 'overflow\_mode' setting, meaning what to do when the limit is exceeded.

It can take one of two values: `throw` or `break`. Restrictions on aggregation (`group_by_overflow_mode`) also have the value `any`.

`throw` – Throw an exception (default).

`break` – Stop executing the query and return the partial result, as if the source data ran out.

`any` (only for `group_by_overflow_mode`) – Continuing aggregation for the keys that got into the set, but don't add new keys to the set.

## readonly

With a value of 0, you can execute any queries.

With a value of 1, you can only execute read requests (such as SELECT and SHOW). Requests for writing and changing settings (INSERT, SET) are prohibited.

With a value of 2, you can process read queries (SELECT, SHOW) and change settings (SET).

After enabling readonly mode, you can't disable it in the current session.

When using the GET method in the HTTP interface, 'readonly = 1' is set automatically. In other words, for queries that modify data, you can only use the POST method. You can send the query itself either in the POST body, or in the URL parameter.

## max\_memory\_usage

The maximum amount of RAM to use for running a query on a single server.

In the default configuration file, the maximum is 10 GB.

The setting doesn't consider the volume of available memory or the total volume of memory on the machine. The restriction applies to a single query within a single server.

You can use `SHOW PROCESSLIST` to see the current memory consumption for each query.

In addition, the peak memory consumption is tracked for each query and written to the log.

Memory usage is not monitored for the states of certain aggregate functions.

Memory usage is not fully tracked for states of the aggregate functions `min`, `max`, `any`, `anyLast`, `argMin`, `argMax` from `String` and `Array` arguments.

Memory consumption is also restricted by the parameters `max_memory_usage_for_user` and `max_memory_usage_for_all_queries`.

## max\_memory\_usage\_for\_user

The maximum amount of RAM to use for running a user's queries on a single server.

Default values are defined in `Settings.h`. By default, the amount is not restricted (`max_memory_usage_for_user = 0`).

See also the description of [max\\_memory\\_usage](#).

## max\_memory\_usage\_for\_all\_queries

The maximum amount of RAM to use for running all queries on a single server.

Default values are defined in [Settings.h](#). By default, the amount is not restricted (`max_memory_usage_for_all_queries = 0`).

See also the description of [max\\_memory\\_usage](#).

## max\_rows\_to\_read

The following restrictions can be checked on each block (instead of on each row). That is, the restrictions can be broken a little.

When running a query in multiple threads, the following restrictions apply to each thread separately.

Maximum number of rows that can be read from a table when running a query.

## max\_bytes\_to\_read

Maximum number of bytes (uncompressed data) that can be read from a table when running a query.

## read\_overflow\_mode

What to do when the volume of data read exceeds one of the limits: 'throw' or 'break'. By default, throw.

## max\_rows\_to\_group\_by

Maximum number of unique keys received from aggregation. This setting lets you limit memory consumption when aggregating.

## group\_by\_overflow\_mode

What to do when the number of unique keys for aggregation exceeds the limit: 'throw', 'break', or 'any'. By default, throw.

Using the 'any' value lets you run an approximation of GROUP BY. The quality of this approximation depends on the statistical nature of the data.

## max\_rows\_to\_sort

Maximum number of rows before sorting. This allows you to limit memory consumption when sorting.

## max\_bytes\_to\_sort

Maximum number of bytes before sorting.

## sort\_overflow\_mode

What to do if the number of rows received before sorting exceeds one of the limits: 'throw' or 'break'. By default, throw.

## max\_result\_rows

Limit on the number of rows in the result. Also checked for subqueries, and on remote servers when running parts of a distributed query.

## max\_result\_bytes

Limit on the number of bytes in the result. The same as the previous setting.

## result\_overflow\_mode

## 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, 50,000.

## max\_rows\_in\_set

## 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**.

ClickHouse applies this setting when the query contains the product of distributed tables, i.e. when the query for a distributed table contains a non-GLOBAL subquery for the distributed table.

Restrictions:

- Only applied for IN and JOIN subqueries.
- Only if the FROM section uses a distributed table containing more than one shard.
- If the subquery concerns a distributed table containing more than one shard,
- Not used for a table-valued **remote** function.

Possible values:

- **deny** — Default value. Prohibits using these types of subqueries (returns the "Double-distributed in/JOIN subqueries is denied" exception).
- **local** — Replaces the database and table in the subquery with local ones for the destination server (shard), leaving the normal **IN/JOIN**.
- **global** — Replaces the **IN/JOIN** query with **GLOBAL IN/GLOBAL JOIN**.
- **allow** — Allows the use of these types of subqueries.

# enable\_optimize\_predicate\_expression

Turns on predicate pushdown in `SELECT` queries.

Predicate pushdown may significantly reduce network traffic for distributed queries.

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 0.

## Usage

Consider the following queries:

1. `SELECT count() FROM test_table WHERE date = '2018-10-10'`
2. `SELECT count() FROM (SELECT * FROM test_table) WHERE date = '2018-10-10'`

If `enable_optimize_predicate_expression = 1`, then the execution time of these queries is equal, because ClickHouse applies `WHERE` to the subquery when processing it.

If `enable_optimize_predicate_expression = 0`, then the execution time of the second query is much longer, because the `WHERE` clause applies to all the data after the subquery finishes.

# fallback\_to\_stale\_replicas\_for\_distributed\_queries

Forces a query to an out-of-date replica if updated data is not available. See "[Replication](#)".

ClickHouse selects the most relevant from the outdated replicas of the table.

Used when performing `SELECT` from a distributed table that points to replicated tables.

By default, 1 (enabled).

# force\_index\_by\_date

Disables query execution if the index can't be used by date.

Works with tables in the MergeTree family.

If `force_index_by_date=1`, ClickHouse checks whether the query has a date key condition that can be used for restricting data ranges. If there is no suitable condition, it throws an exception. However, it does not check whether the condition actually reduces the amount of data to read. For example, the condition `Date != '2000-01-01'` is acceptable even when it matches all the data in the table (i.e., running the query requires a full scan). For more information about ranges of data in MergeTree tables, see "[MergeTree](#)".

# force\_primary\_key

Disables query execution if indexing by the primary key is not possible.

Works with tables in the MergeTree family.

If `force_primary_key=1`, ClickHouse checks to see if the query has a primary key condition that can be used for restricting data ranges. If there is no suitable condition, it throws an exception. However, it does not check whether the condition actually reduces the amount of data to read. For more information about data ranges in MergeTree tables, see "[MergeTree](#)".

# fsync\_metadata

Enables or disables `fsync` when writing `.sql` files. Enabled by default.

It makes sense to disable it if the server has millions of tiny table chunks that are constantly being created and destroyed.

## enable\_http\_compression

Enables or disables data compression in the response to an HTTP request.

For more information, read the [HTTP interface description](#).

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 0.

## http\_zlib\_compression\_level

Sets the level of data compression in the response to an HTTP request if `enable_http_compression = 1`.

Possible values: Numbers from 1 to 9.

Default value: 3.

## http\_native\_compression\_disable\_checksumming\_on\_decompress

Enables or disables checksum verification when decompressing the HTTP POST data from the client. Used only for ClickHouse native compression format (not used with `gzip` or `deflate`).

For more information, read the [HTTP interface description](#).

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 0.

## input\_format\_allow\_errors\_num

Sets the maximum number of acceptable errors when reading from text formats (CSV, TSV, etc.).

The default value is 0.

Always pair it with `input_format_allow_errors_ratio`. To skip errors, both settings must be greater than 0.

If an error occurred while reading rows but the error counter is still less than `input_format_allow_errors_num`, ClickHouse ignores the row and moves on to the next one.

If `input_format_allow_errors_num` is exceeded, ClickHouse throws an exception.

## input\_format\_allow\_errors\_ratio

Sets the maximum percentage of errors allowed when reading from text formats (CSV, TSV, etc.).

The percentage of errors is set as a floating-point number between 0 and 1.

The default value is 0.

Always pair it with `input_format_allow_errors_num`. To skip errors, both settings must be greater than 0.

If an error occurred while reading rows but the error counter is still less than `input_format_allow_errors_ratio`, ClickHouse ignores the row and moves on to the next one.

If `input format allow errors ratio` is exceeded, ClickHouse throws an exception.

## input\_format\_values\_interpret\_expressions

Enables or disables the full SQL parser if the fast stream parser can't parse the data. This setting is used only for the **Values** format at the data insertion. For more information about syntax parsing, see the **Syntax** section.

Possible values:

- 0 — Disabled.

In this case, you must provide formatted data. See the **Formats** section.

- 1 — Enabled.

In this case, you can use an SQL expression as a value, but data insertion is much slower this way. If you insert only formatted data, then ClickHouse behaves as if the setting value is 0.

Default value: 1.

### Example of Use

Insert the **DateTime** type value with the different settings.

```
SET input_format_values_interpret_expressions = 0;
INSERT INTO datetime_t VALUES (now())
```

Exception on client:

Code: 27. DB::Exception: Cannot parse input: expected ) before: now(): (at row 1)

```
SET input_format_values_interpret_expressions = 1;
INSERT INTO datetime_t VALUES (now())
```

Ok.

The last query is equivalent to the following:

```
SET input_format_values_interpret_expressions = 0;
INSERT INTO datetime_t SELECT now()
```

Ok.

## input\_format\_defaults\_for\_omitted\_fields

Turns on/off the extended data exchange between a ClickHouse client and a ClickHouse server. This setting applies for **INSERT** queries.

When executing the **INSERT** query, the ClickHouse client prepares data and sends it to the server for writing. The client gets the table structure from the server when preparing the data. In some cases, the client needs more information than the server sends by default. Turn on the extended data exchange with `input_format_defaults_for_omitted_fields = 1`.

When the extended data exchange is enabled, the server sends the additional metadata along with the table structure. The composition of the metadata depends on the operation.

Operations where you may need the extended data exchange enabled:

- Inserting data in **JSONEachRow** format.



For all other operations, ClickHouse doesn't apply the setting.

#### Note

The extended data exchange functionality consumes additional computing resources on the server and can reduce performance.

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 0.

## input\_format\_skip\_unknown\_fields

Enables or disables skipping of insertion of extra data.

When writing data, ClickHouse throws an exception if input data contain columns that do not exist in the target table. If skipping is enabled, ClickHouse doesn't insert extra data and doesn't throw an exception.

Supported formats: [JSONEachRow](#), [CSVWithNames](#), [TabSeparatedWithNames](#), [TSKV](#).

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 0.

## input\_format\_with\_names\_use\_header

Enables or disables checking the column order when inserting data.

We recommend disabling check, if you are sure that the column order of the input data is the same as in the target table. It increases ClickHouse performance.

Supported formats: [CSVWithNames](#), [TabSeparatedWithNames](#).

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 1.

## join\_default\_strictness

Sets default strictness for [JOIN clauses](#).

### Possible values

- `ALL` — If the right table has several matching rows, the data is multiplied by the number of these rows. This is the normal `JOIN` behavior from standard SQL.
- `ANY` — If the right table has several matching rows, only the first one found is joined. If the right table has only one matching row, the results of `ANY` and `ALL` are the same.
- Empty string — If `ALL` or `ANY` is not specified in the query, ClickHouse throws an exception.

**Default value:** `ALL`

## join\_use\_nulls

Sets the type of [JOIN](#) behavior. When merging tables, empty cells may appear. ClickHouse fills them

sets the type of **join** behavior. When merging tables, empty cells may appear. ClickHouse fills them differently based on this setting.

#### Possible values

- 0 — The empty cells are filled with the default value of the corresponding field type.
- 1 — **JOIN** behaves the same way as in standard SQL. The type of the corresponding field is converted to **Nullable**, and empty cells are filled with **NULL**.

**Default value:** 0.

## max\_block\_size

In ClickHouse, data is processed by blocks (sets of column parts). The internal processing cycles for a single block are efficient enough, but there are noticeable expenditures on each block. The `max_block_size` setting is a recommendation for what size of block (in number of rows) to load from tables. The block size shouldn't be too small, so that the expenditures on each block are still noticeable, but not too large, so that the query with **LIMIT** that is completed after the first block is processed quickly. The goal is to avoid consuming too much memory when extracting a large number of columns in multiple threads, and to preserve at least some cache locality.

Default value: 65,536.

Blocks the size of `max_block_size` are not always loaded from the table. If it is obvious that less data needs to be retrieved, a smaller block is processed.

## preferred\_block\_size\_bytes

Used for the same purpose as `max_block_size`, but it sets the recommended block size in bytes by adapting it to the number of rows in the block.

However, the block size cannot be more than `max_block_size` rows.

By default: 1,000,000. It only works when reading from MergeTree engines.

## merge\_tree\_uniform\_read\_distribution

ClickHouse uses multiple threads when reading from **MergeTree\*** tables. This setting turns on/off the uniform distribution of reading tasks over the working threads. The algorithm of the uniform distribution aims to make execution time for all the threads approximately equal in a **SELECT** query.

#### Possible values

- 0 — Do not use uniform read distribution.
- 1 — Use uniform read distribution.

**Default value:** 1.

## merge\_tree\_min\_rows\_for\_concurrent\_read

If the number of rows to be read from a file of a **MergeTree\*** table exceeds `merge_tree_min_rows_for_concurrent_read` then ClickHouse tries to perform a concurrent reading from this file on several threads.

#### Possible values

Any positive integer.

**Default value:** 163840.

## merge\_tree\_min\_rows\_for\_seek

If the distance between two data blocks to be read in one file is less than `merge_tree_min_rows_for_seek` rows, then ClickHouse does not seek through the file, but reads the data sequentially.

### Possible values

Any positive integer.

**Default value:** 0.

## merge\_tree\_coarse\_index\_granularity

When searching data, ClickHouse checks the data marks in the index file. If ClickHouse finds that required keys are in some range, it divides this range into `merge_tree_coarse_index_granularity` subranges and searches the required keys there recursively.

### Possible values

Any positive even integer.

**Default value:** 8.

## merge\_tree\_max\_rows\_to\_use\_cache

If ClickHouse should read more than `merge_tree_max_rows_to_use_cache` rows in one query, it does not use the cache of uncompressed blocks. The `uncompressed_cache_size` server setting defines the size of the cache of uncompressed blocks.

### Possible values

Any positive integer.

**Default value:** 1048576.

## min\_bytes\_to\_use\_direct\_io

The minimum data volume required for using direct I/O access to the storage disk.

ClickHouse uses this setting when reading data from tables. If the total storage volume of all the data to be read exceeds `min_bytes_to_use_direct_io` bytes, then ClickHouse reads the data from the storage disk with the `O_DIRECT` option.

### Possible values

- 0 — Direct I/O is disabled.
- Positive integer.

**Default value:** 0.

## log\_queries

Setting up query logging.

Queries sent to ClickHouse with this setup are logged according to the rules in the `query_log` server configuration parameter.

### Example:

```
log_queries=1
```

## max\_insert\_block\_size

The size of blocks to form for insertion into a table.

This setting only applies in cases when the server forms the blocks.

For example, for an INSERT via the HTTP interface, the server parses the data format and forms blocks of the

For example, for an INSERT via the HTTP interface, the server parses the data format and forms blocks of the specified size.

But when using clickhouse-client, the client parses the data itself, and the 'max\_insert\_block\_size' setting on the server doesn't affect the size of the inserted blocks.

The setting also doesn't have a purpose when using INSERT SELECT, since data is inserted using the same blocks that are formed after SELECT.

Default value: 1,048,576.

The default is slightly more than `max_block_size`. The reason for this is because certain table engines (`*MergeTree`) form a data part on the disk for each inserted block, which is a fairly large entity. Similarly, `*MergeTree` tables sort data during insertion, and a large enough block size allows sorting more data in RAM.

## max\_replica\_delay\_for\_distributed\_queries

Disables lagging replicas for distributed queries. See "[Replication](#)".

Sets the time in seconds. If a replica lags more than the set value, this replica is not used.

Default value: 300.

Used when performing `SELECT` from a distributed table that points to replicated tables.

## max\_threads

The maximum number of query processing threads, excluding threads for retrieving data from remote servers (see the 'max\_distributed\_connections' parameter).

This parameter applies to threads that perform the same stages of the query processing pipeline in parallel. For example, when reading from a table, if it is possible to evaluate expressions with functions, filter with WHERE and pre-aggregate for GROUP BY in parallel using at least 'max\_threads' number of threads, then 'max\_threads' are used.

Default value: 2.

If less than one SELECT query is normally run on a server at a time, set this parameter to a value slightly less than the actual number of processor cores.

For queries that are completed quickly because of a LIMIT, you can set a lower 'max\_threads'. For example, if the necessary number of entries are located in every block and max\_threads = 8, then 8 blocks are retrieved, although it would have been enough to read just one.

The smaller the `max_threads` value, the less memory is consumed.

## max\_compress\_block\_size

The maximum size of blocks of uncompressed data before compressing for writing to a table. By default, 1,048,576 (1 MiB). If the size is reduced, the compression rate is significantly reduced, the compression and decompression speed increases slightly due to cache locality, and memory consumption is reduced. There usually isn't any reason to change this setting.

Don't confuse blocks for compression (a chunk of memory consisting of bytes) with blocks for query processing (a set of rows from a table).

## min\_compress\_block\_size

For `MergeTree` tables. In order to reduce latency when processing queries, a block is compressed when writing the next mark if its size is at least 'min\_compress\_block\_size'. By default, 65,536.

The actual size of the block, if the uncompressed data is less than 'max\_compress\_block\_size', is no less than this value and no less than the volume of data for one mark.

Let's look at an example. Assume that 'index\_granularity' was set to 8192 during table creation.

We are writing a UInt32-type column (4 bytes per value). When writing 8192 rows, the total will be 32 KB of data. Since min\_compress\_block\_size = 65,536, a compressed block will be formed for every two marks.

We are writing a URL column with the String type (average size of 60 bytes per value). When writing 8192 rows, the average will be slightly less than 500 KB of data. Since this is more than 65,536, a compressed block will be formed for each mark. In this case, when reading data from the disk in the range of a single mark, extra data won't be decompressed.

There usually isn't any reason to change this setting.

## max\_query\_size

The maximum part of a query that can be taken to RAM for parsing with the SQL parser.

The INSERT query also contains data for INSERT that is processed by a separate stream parser (that consumes O(1) RAM), which is not included in this restriction.

Default value: 256 KiB.

## interactive\_delay

The interval in microseconds for checking whether request execution has been canceled and sending the progress.

Default value: 100,000 (checks for canceling and sends the progress ten times per second).

## connect\_timeout, receive\_timeout, send\_timeout

Timeouts in seconds on the socket used for communicating with the client.

Default value: 10, 300, 300.

## poll\_interval

Lock in a wait loop for the specified number of seconds.

Default value: 10.

## max\_distributed\_connections

The maximum number of simultaneous connections with remote servers for distributed processing of a single query to a single Distributed table. We recommend setting a value no less than the number of servers in the cluster.

Default value: 1024.

The following parameters are only used when creating Distributed tables (and when launching a server), so there is no reason to change them at runtime.

## distributed\_connections\_pool\_size

The maximum number of simultaneous connections with remote servers for distributed processing of all queries to a single Distributed table. We recommend setting a value no less than the number of servers in the cluster.

Default value: 1024.

## connect\_timeout\_with\_failover\_ms

The timeout in milliseconds for connecting to a remote server for a Distributed table engine, if the 'shard' and 'replica' sections are used in the cluster definition.

If unsuccessful, several attempts are made to connect to various replicas.

Default value: 50.

## connections\_with\_failover\_max\_tries

The maximum number of connection attempts with each replica for the Distributed table engine.

Default value: 3.

## extremes

Whether to count extreme values (the minimums and maximums in columns of a query result). Accepts 0 or 1. By default, 0 (disabled).

For more information, see the section "Extreme values".

## use\_uncompressed\_cache

Whether to use a cache of uncompressed blocks. Accepts 0 or 1. By default, 0 (disabled).

Using the uncompressed cache (only for tables in the MergeTree family) can significantly reduce latency and increase throughput when working with a large number of short queries. Enable this setting for users who send frequent short requests. Also pay attention to the [uncompressed\\_cache\\_size](#) configuration parameter (only set in the config file) – the size of uncompressed cache blocks. By default, it is 8 GiB. The uncompressed cache is filled in as needed and the least-used data is automatically deleted.

For queries that read at least a somewhat large volume of data (one million rows or more), the uncompressed cache is disabled automatically in order to save space for truly small queries. This means that you can keep the 'use\_uncompressed\_cache' setting always set to 1.

## replace\_running\_query

When using the HTTP interface, the 'query\_id' parameter can be passed. This is any string that serves as the query identifier.

If a query from the same user with the same 'query\_id' already exists at this time, the behavior depends on the 'replace\_running\_query' parameter.

**0** (default) – Throw an exception (don't allow the query to run if a query with the same 'query\_id' is already running).

**1** – Cancel the old query and start running the new one.

Yandex.Metrica uses this parameter set to 1 for implementing suggestions for segmentation conditions. After entering the next character, if the old query hasn't finished yet, it should be canceled.

## schema

This parameter is useful when you are using formats that require a schema definition, such as [Cap'n Proto](#). The value depends on the format.

## stream\_flush\_interval\_ms

Works for tables with streaming in the case of a timeout, or when a thread generates [max\\_insert\\_block\\_size](#) rows.

The default value is 7500.

The smaller the value, the more often data is flushed into the table. Setting the value too low leads to poor performance.

## load\_balancing

Specifies the algorithm of replicas selection that is used for distributed query processing

specifies the algorithm of replicas selection that is used for distributed query processing.

ClickHouse supports the following algorithms of choosing replicas:

- **Random** (by default)
- **Nearest hostname**
- **In order**
- **First or random**

## Random (by default)

```
load_balancing = random
```

The number of errors is counted for each replica. The query is sent to the replica with the fewest errors, and if there are several of these, to any one of them.

Disadvantages: Server proximity is not accounted for; if the replicas have different data, you will also get different data.

## Nearest Hostname

```
load_balancing = nearest_hostname
```

The number of errors is counted for each replica. Every 5 minutes, the number of errors is integrally divided by 2. Thus, the number of errors is calculated for a recent time with exponential smoothing. If there is one replica with a minimal number of errors (i.e. errors occurred recently on the other replicas), the query is sent to it. If there are multiple replicas with the same minimal number of errors, the query is sent to the replica with a host name that is most similar to the server's host name in the config file (for the number of different characters in identical positions, up to the minimum length of both host names).

For instance, example01-01-1 and example01-01-2.yandex.ru are different in one position, while example01-01-1 and example01-02-2 differ in two places.

This method might seem primitive, but it doesn't require external data about network topology, and it doesn't compare IP addresses, which would be complicated for our IPv6 addresses.

Thus, if there are equivalent replicas, the closest one by name is preferred.

We can also assume that when sending a query to the same server, in the absence of failures, a distributed query will also go to the same servers. So even if different data is placed on the replicas, the query will return mostly the same results.

## In Order

```
load_balancing = in_order
```

Replicas with the same number of errors are accessed in the same order as they are specified in configuration.

This method is appropriate when you know exactly which replica is preferable.

## First or Random

```
load_balancing = first_or_random
```

This algorithm chooses the first replica in order or a random replica if the first one is unavailable. It is effective in cross-replication topology setups, but it useless in other configurations.

The `first or random` algorithm solves the problem of the `in order` algorithm. The problem is: if one replica goes down, the next one handles twice the usual load while remaining ones handle usual traffic. When using the

`first or random` algorithm, the load on replicas is leveled.

## prefer\_localhost\_replica

Enables/disables preferable using the localhost replica when processing distributed queries.

Possible values:

- 1 — ClickHouse always sends a query to the localhost replica if it exists.
- 0 — ClickHouse uses the balancing strategy specified by the `load_balancing` setting.

Default value: 1.

## totals\_mode

How to calculate TOTALS when HAVING is present, as well as when `max_rows_to_group_by` and `group_by_overflow_mode = 'any'` are present.

See the section "WITH TOTALS modifier".

## totals\_auto\_threshold

The threshold for `totals_mode = 'auto'`.

See the section "WITH TOTALS modifier".

## max\_parallel\_replicas

The maximum number of replicas for each shard when executing a query.

For consistency (to get different parts of the same data split), this option only works when the sampling key is set.

Replica lag is not controlled.

## compile

Enable compilation of queries. By default, 0 (disabled).

Compilation is only used for part of the query-processing pipeline: for the first stage of aggregation (GROUP BY).

If this portion of the pipeline was compiled, the query may run faster due to deployment of short cycles and inlining aggregate function calls. The maximum performance improvement (up to four times faster in rare cases) is seen for queries with multiple simple aggregate functions. Typically, the performance gain is insignificant. In very rare cases, it may slow down query execution.

## min\_count\_to\_compile

How many times to potentially use a compiled chunk of code before running compilation. By default, 3.

For testing, the value can be set to 0: compilation runs synchronously and the query waits for the end of the compilation process before continuing execution. For all other cases, use values starting with 1. Compilation normally takes about 5-10 seconds.

If the value is 1 or more, compilation occurs asynchronously in a separate thread. The result will be used as soon as it is ready, including queries that are currently running.

Compiled code is required for each different combination of aggregate functions used in the query and the type of keys in the GROUP BY clause.

The results of compilation are saved in the build directory in the form of .so files. There is no restriction on the number of compilation results, since they don't use very much space. Old results will be used after server restarts, except in the case of a server upgrade – in this case, the old results are deleted.

## output\_format\_json\_quote\_64bit\_integers

If the value is true, integers appear in quotes when using JSON\* Int64 and UInt64 formats (for compatibility with most JavaScript implementations); otherwise, integers are output without the quotes.



# format\_csv\_delimiter

The character interpreted as a delimiter in the CSV data. By default, the delimiter is `,`.

# insert\_quorum

Enables quorum writes.

- If `insert_quorum < 2`, the quorum writes are disabled.
- If `insert_quorum >= 2`, the quorum writes are enabled.

Default value: 0.

## Quorum writes

`INSERT` succeeds only when ClickHouse manages to correctly write data to the `insert_quorum` of replicas during the `insert_quorum_timeout`. If for any reason the number of replicas with successful writes does not reach the `insert_quorum`, the write is considered failed and ClickHouse will delete the inserted block from all the replicas where data has already been written.

All the replicas in the quorum are consistent, i.e., they contain data from all previous `INSERT` queries. The `INSERT` sequence is linearized.

When reading the data written from the `insert_quorum`, you can use the `select_sequential_consistency` option.

## ClickHouse generates an exception

- If the number of available replicas at the time of the query is less than the `insert_quorum`.
- At an attempt to write data when the previous block has not yet been inserted in the `insert_quorum` of replicas. This situation may occur if the user tries to perform an `INSERT` before the previous one with the `insert_quorum` is completed.

## See also the following parameters:

- `insert_quorum_timeout`
- `select_sequential_consistency`

# insert\_quorum\_timeout

Quorum write timeout in seconds. If the timeout has passed and no write has taken place yet, ClickHouse will generate an exception and the client must repeat the query to write the same block to the same or any other replica.

Default value: 60 seconds.

## See also the following parameters:

- `insert_quorum`
- `select_sequential_consistency`

# select\_sequential\_consistency

Enables or disables sequential consistency for `SELECT` queries:

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 0.

## Usage

When sequential consistency is enabled, ClickHouse allows the client to execute the `SELECT` query only for those replicas that contain data from all previous `INSERT` queries executed with `insert_quorum`. If the client refers to a partial replica, ClickHouse will generate an exception. The `SELECT` query will not include data that has not yet been written to the quorum of replicas.

## See Also

- `insert_quorum`
- `insert_quorum_timeout`

# allow\_experimental\_cross\_to\_join\_conversion

Enables or disables:

1. Rewriting of queries with multiple `JOIN clauses` from the syntax with commas to the `JOIN ON/USING` syntax. If the setting value is 0, ClickHouse doesn't process queries with the syntax with commas, and throws an exception.
2. Converting of `CROSS JOIN` into `INNER JOIN` if conditions of join allow it.

Possible values:

- 0 — Disabled.
- 1 — Enabled.

Default value: 1.

## Settings profiles

A settings profile is a collection of settings grouped under the same name. Each ClickHouse user has a profile.

To apply all the settings in a profile, set the `profile` setting.

Example:

Install the `web` profile.

```
SET profile = 'web'
```

Settings profiles are declared in the user config file. This is usually `users.xml`.

Example:

```
<!-- Settings profiles -->
<profiles>
 <!-- Default settings -->
 <default>
 <!-- The maximum number of threads when running a single query. -->
 <max_threads>8</max_threads>
 </default>

 <!-- Settings for queries from the user interface -->
 <web>
 <max_rows_to_read>1000000000</max_rows_to_read>
 <max_bytes_to_read>100000000000</max_bytes_to_read>

 <max_rows_to_group_by>1000000</max_rows_to_group_by>
 <group_by_overflow_mode>any</group_by_overflow_mode>

 <max_rows_to_sort>1000000</max_rows_to_sort>
```

```

<max_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` — Allows running SQL queries on data without stopping the ClickHouse server, similar to how `awk` does this.
- `clickhouse-copier` — Copies (and reshard) data from one cluster to another cluster.

## clickhouse-copier

Copies data from the tables in one cluster to tables in another (or the same) cluster.

You can run multiple `clickhouse-copier` instances on different servers to perform the same job. ZooKeeper is used for syncing the processes.

After starting, `clickhouse-copier`:

- Connects to ZooKeeper and receives:
  - Copying jobs.
  - The state of the copying jobs.
- It performs the jobs.

Each running process chooses the "closest" shard of the source cluster and copies the data into the destination cluster, resharding the data if necessary.

`clickhouse-copier` tracks the changes in ZooKeeper and applies them on the fly.

To reduce network traffic, we recommend running `clickhouse-copier` on the same server where the source data is located.

# Running clickhouse-copier

The utility should be run manually:

```
clickhouse-copier copier --daemon --config zookeeper.xml --task-path /task/path --base-dir /path/to/dir
```

Parameters:

- `daemon` — Starts `clickhouse-copier` in daemon mode.
- `config` — The path to the `zookeeper.xml` file with the parameters for the connection to ZooKeeper.
- `task-path` — The path to the ZooKeeper node. This node is used for syncing `clickhouse-copier` processes and storing tasks. Tasks are stored in `$task-path/description`.
- `base-dir` — The path to logs and auxiliary files. When it starts, `clickhouse-copier` creates `clickhouse-copier_YYYYMMHHSS_<PID>` subdirectories in `$base-dir`. If this parameter is omitted, the directories are created in the directory where `clickhouse-copier` was launched.

## Format of zookeeper.xml

```
<yandex>
 <logger>
 <level>trace</level>
 <size>100M</size>
 <count>3</count>
 </logger>

 <zookeeper>
 <node index="1">
 <host>127.0.0.1</host>
 <port>2181</port>
 </node>
 </zookeeper>
</yandex>
```

## Configuration of copying tasks

```
<yandex>
 <!-- Configuration of clusters as in an ordinary server config -->
 <remote_servers>
 <source_cluster>
 <shard>
 <internal_replication>false</internal_replication>
 <replica>
 <host>127.0.0.1</host>
 <port>9000</port>
 </replica>
 </shard>
 ...
 </source_cluster>

 <destination_cluster>
 ...
 </destination_cluster>
 </remote_servers>

 <!-- How many simultaneously active workers are possible. If you run more workers superfluous workers will sleep. -->
 <max_workers>2</max_workers>

 <!-- Setting used to fetch (pull) data from source cluster tables -->
```

<!-- Setting used to fetch (pull) data from source cluster tables -->

```
<settings_pull>
 <readonly>1</readonly>
</settings_pull>
```

<!-- Setting used to insert (push) data to destination cluster tables -->

```
<settings_push>
 <readonly>0</readonly>
</settings_push>
```

<!-- Common setting for fetch (pull) and insert (push) operations. Also, copier process context uses it.

They are overlaid by <settings\_pull/> and <settings\_push/> respectively. -->

```
<settings>
 <connect_timeout>3</connect_timeout>
 <!-- Sync insert is set forcibly, leave it here just in case. -->
 <insert_distributed_sync>1</insert_distributed_sync>
</settings>
```

<!-- Copying tasks description.

You could specify several table task in the same task description (in the same ZooKeeper node), they will be performed sequentially.

-->

<tables>

<!-- A table task, copies one table. -->

<table\_hits>

<!-- Source cluster name (from <remote\_servers/> section) and tables in it that should be copied -->

<cluster\_pull>source\_cluster</cluster\_pull>

<database\_pull>test</database\_pull>

<table\_pull>hits</table\_pull>

<!-- Destination cluster name and tables in which the data should be inserted -->

<cluster\_push>destination\_cluster</cluster\_push>

<database\_push>test</database\_push>

<table\_push>hits2</table\_push>

<!-- Engine of destination tables.

If destination tables have not be created, workers create them using columns definition from source tables and engine definition from here.

NOTE: If the first worker starts insert data and detects that destination partition is not empty then the partition will

be dropped and refilled, take it into account if you already have some data in destination tables. You could directly

specify partitions that should be copied in <enabled\_partitions/>, they should be in quoted format like partition column of system.parts table.

-->

<engine>

ENGINE=ReplicatedMergeTree('/clickhouse/tables/{cluster}/{shard}/hits2', '{replica}')

PARTITION BY toMonday(date)

ORDER BY (CounterID, EventDate)

</engine>

<!-- Sharding key used to insert data to destination cluster -->

<sharding\_key>jumpConsistentHash(intHash64(UserID), 2)</sharding\_key>

<!-- Optional expression that filter data while pull them from source servers -->

<where\_condition>CounterID != 0</where\_condition>

<!-- This section specifies partitions that should be copied, other partition will be ignored.

Partition names should have the same format as

partition column of system.parts table (i.e. a quoted text).  
Since partition key of source and destination cluster could be different,  
these partition names specify destination partitions.

NOTE: In spite of this section is optional (if it is not specified, all partitions will be copied),  
it is strictly recommended to specify them explicitly.

If you already have some ready partitions on destination cluster they  
will be removed at the start of the copying since they will be interpreted  
as unfinished data from the previous copying!!!

-->

<enabled\_partitions>

<partition>'2018-02-26'</partition>

<partition>'2018-03-05'</partition>

...

</enabled\_partitions>

</table\_hits>

<!-- Next table to copy. It is not copied until previous table is copying. -->

</table\_visits>

...

</table\_visits>

...

</tables>

</yandex>

clickhouse-copier tracks the changes in `/task/path/description` and applies them on the fly. For instance, if you change the value of `max_workers`, the number of processes running tasks will also change.

## clickhouse-local

The `clickhouse-local` program enables you to perform fast processing on local files, without having to deploy and configure the ClickHouse server.

Accepts data that represent tables and queries them using [ClickHouse SQL dialect](#).

`clickhouse-local` uses the same core as ClickHouse server, so it supports most of the features and the same set of formats and table engines.

By default `clickhouse-local` does not have access to data on the same host, but it supports loading server configuration using `--config-file` argument.

### Warning

It is not recommended to load production server configuration into `clickhouse-local` because data can be damaged in case of human error.

## Usage

Basic usage:

```
clickhouse-local --structure "table_structure" --input-format "format_of_incoming_data" -q "query"
```

Arguments:

- `-S`, `--structure` — table structure for input data.
- `-if`, `--input-format` — input format, `TSV` by default.
- `-f`, `--file` — path to data, `stdin` by default.
- `-q` `--query` — queries to execute with `;` as delimiter.
- `-N`, `--table` — table name where to put output data, `table` by default.
- `-o`, `--output-format` — output format, `TSV` by default.

- `-ot`, `--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。其中最常见开源解决方案是 **Apache Hadoop**。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。

# What to do if I have a problem with encodings when using Oracle through ODBC?

If you use Oracle through ODBC driver as a source of external dictionaries, you need to set up correctly value for the `NLS_LANG` variable in the `/etc/default/clickhouse`. For more details see the [Oracle NLS\\_LANG FAQ](#).

## Example

```
NLS_LANG=CHINESE_CHINA.ZHS16GBK
```

## ClickHouse 开发

# Overview of ClickHouse Architecture

ClickHouse is a true column-oriented DBMS. Data is stored by columns, and during the execution of arrays (vectors or chunks of columns). Whenever possible, operations are dispatched on arrays, rather than on individual values. This is called "vectorized query execution," and it helps lower the cost of actual data processing.

This idea is nothing new. It dates back to the `APL` programming language and its descendants: `A +`, `J`, `K`, and `Q`. Array programming is used in scientific data processing. Neither is this idea something new in relational databases: for example, it is used in the `Vectorwise` system.

There are two different approaches for speeding up the query processing: vectorized query execution and runtime code generation. In the latter, the code is generated for every kind of query on the fly, removing all indirection and dynamic dispatch. Neither of these approaches is strictly better than the other. Runtime code generation can be better when it fuses many operations together, thus fully utilizing CPU execution units and the pipeline. Vectorized query execution can be less practical, because it involves temporary vectors that must be written to the cache and read back. If the temporary data does not fit in the L2 cache, this becomes an issue. But vectorized query execution more easily utilizes the SIMD capabilities of the CPU. A [research paper](#) written by our friends shows that it is better to combine both approaches. ClickHouse uses vectorized query execution and has limited initial support for runtime code generation.

## Columns

To represent columns in memory (actually, chunks of columns), the `IColumn` interface is used. This interface provides helper methods for implementation of various relational operators. Almost all operations are immutable: they do not modify the original column, but create a new modified one. For example, the `IColumn::filter` method accepts a filter byte mask. It is used for the `WHERE` and `HAVING` relational operators. Additional examples: the `IColumn::permute` method to support `ORDER BY`, the `IColumn::cut` method to support `LIMIT`, and so on.

Various `IColumn` implementations (`ColumnUInt8`, `ColumnString` and so on) are responsible for the memory layout of columns. Memory layout is usually a contiguous array. For the integer type of columns it is just one contiguous array, like `std::vector`. For `String` and `Array` columns, it is two vectors: one for all array elements, placed contiguously, and a second one for offsets to the beginning of each array. There is also `ColumnConst` that stores just one value in memory, but looks like a column.

## Field

Nevertheless, it is possible to work with individual values as well. To represent an individual value, the `Field` is used. `Field` is just a discriminated union of `UInt64`, `Int64`, `Float64`, `String` and `Array`. `IColumn` has the `operator[]` method to get the `n`-th value as a `Field`, and the `insert` method to append a `Field` to the end of a column. These methods are not very efficient, because they require dealing with temporary `Field` objects representing an individual value. There are more efficient methods, such as `insertFrom`, `insertRangeFrom`, and



representing an individual value. There are more efficient methods, such as `insertion`, `insertrange`, and so on.

`Field` doesn't have enough information about a specific data type for a table. For example, `UInt8`, `UInt16`, `UInt32`, and `UInt64` are all represented as `UInt64` in a `Field`.

## Leaky Abstractions

`IColumn` has methods for common relational transformations of data, but they don't meet all needs. For example, `ColumnUInt64` doesn't have a method to calculate the sum of two columns, and `ColumnString` doesn't have a method to run a substring search. These countless routines are implemented outside of `IColumn`.

Various functions on columns can be implemented in a generic, non-efficient way using `IColumn` methods to extract `Field` values, or in a specialized way using knowledge of inner memory layout of data in a specific `IColumn` implementation. To do this, functions are cast to a specific `IColumn` type and deal with internal representation directly. For example, `ColumnUInt64` has the `getData` method that returns a reference to an internal array, then a separate routine reads or fills that array directly. In fact, we have "leaky abstractions" to allow efficient specializations of various routines.

## Data Types

`IDataType` is responsible for serialization and deserialization: for reading and writing chunks of columns or individual values in binary or text form.

`IDataType` directly corresponds to data types in tables. For example, there are `DataTypeUInt32`, `DataTypeDateTime`, `DataTypeString` and so on.

`IDataType` and `IColumn` are only loosely related to each other. Different data types can be represented in memory by the same `IColumn` implementations. For example, `DataTypeUInt32` and `DataTypeDateTime` are both represented by `ColumnUInt32` or `ColumnConstUInt32`. In addition, the same data type can be represented by different `IColumn` implementations. For example, `DataTypeUInt8` can be represented by `ColumnUInt8` or `ColumnConstUInt8`.

`IDataType` only stores metadata. For instance, `DataTypeUInt8` doesn't store anything at all (except `vptr`) and `DataTypeFixedString` stores just `N` (the size of fixed-size strings).

`IDataType` has helper methods for various data formats. Examples are methods to serialize a value with possible quoting, to serialize a value for JSON, and to serialize a value as part of XML format. There is no direct correspondence to data formats. For example, the different data formats `Pretty` and `TabSeparated` can use the same `serializeTextEscaped` helper method from the `IDataType` interface.

## Block

A `Block` is a container that represents a subset (chunk) of a table in memory. It is just a set of triples: (`IColumn`, `IDataType`, column name). During query execution, data is processed by `Blocks`. If we have a `Block`, we have data (in the `IColumn` object), we have information about its type (in `IDataType`) that tells us how to deal with that column, and we have the column name (either the original column name from the table, or some artificial name assigned for getting temporary results of calculations).

When we calculate some function over columns in a block, we add another column with its result to the block, and we don't touch columns for arguments of the function because operations are immutable. Later, unneeded columns can be removed from the block, but not modified. This is convenient for elimination of common subexpressions.

Blocks are created for every processed chunk of data. Note that for the same type of calculation, the column names and types remain the same for different blocks, and only column data changes. It is better to split block data from the block header, because small block sizes will have a high overhead of temporary strings for copying shared\_ptrs and column names.

## Block Streams

Block streams are for processing data. We use streams of blocks to read data from somewhere, perform data transformations, or write data to somewhere. `IBlockInputStream` has the `read` method to fetch the next block while available. `IBlockOutputStream` has the `write` method to push the block somewhere.

Streams are responsible for:

1. Reading or writing to a table. The table just returns a stream for reading or writing blocks.
2. Implementing data formats. For example, if you want to output data to a terminal in `Pretty` format, you create a block output stream where you push blocks, and it formats them.
3. Performing data transformations. Let's say you have `IBlockInputStream` and want to create a filtered stream. You create `FilterBlockInputStream` and initialize it with your stream. Then when you pull a block from `FilterBlockInputStream`, it pulls a block from your stream, filters it, and returns the filtered block to you. Query execution pipelines are represented this way.

There are more sophisticated transformations. For example, when you pull from `AggregatingBlockInputStream`, it reads all data from its source, aggregates it, and then returns a stream of aggregated data for you. Another example: `UnionBlockInputStream` accepts many input sources in the constructor and also a number of threads. It launches multiple threads and reads from multiple sources in parallel.

Block streams use the "pull" approach to control flow: when you pull a block from the first stream, it consequently pulls the required blocks from nested streams, and the entire execution pipeline will work. Neither "pull" nor "push" is the best solution, because control flow is implicit, and that limits implementation of various features like simultaneous execution of multiple queries (merging many pipelines together). This limitation could be overcome with coroutines or just running extra threads that wait for each other. We may have more possibilities if we make control flow explicit: if we locate the logic for passing data from one calculation unit to another outside of those calculation units. Read this [article](#) for more thoughts.

We should note that the query execution pipeline creates temporary data at each step. We try to keep block size small enough so that temporary data fits in the CPU cache. With that assumption, writing and reading temporary data is almost free in comparison with other calculations. We could consider an alternative, which is to fuse many operations in the pipeline together, to make the pipeline as short as possible and remove much of the temporary data. This could be an advantage, but it also has drawbacks. For example, a split pipeline makes it easy to implement caching intermediate data, stealing intermediate data from similar queries running at the same time, and merging pipelines for similar queries.

## Formats

Data formats are implemented with block streams. There are "presentational" formats only suitable for output of data to the client, such as `Pretty` format, which provides only `IBlockOutputStream`. And there are input/output formats, such as `TabSeparated` or `JSONEachRow`.

There are also row streams: `IRowInputStream` and `IRowOutputStream`. They allow you to pull/push data by individual rows, not by blocks. And they are only needed to simplify implementation of row-oriented formats. The wrappers `BlockInputStreamFromRowInputStream` and `BlockOutputStreamFromRowOutputStream` allow you to convert row-oriented streams to regular block-oriented streams.

## I/O

For byte-oriented input/output, there are `ReadBuffer` and `WriteBuffer` abstract classes. They are used instead of C++ `istream`s. Don't worry: every mature C++ project is using something other than `istream`s for good reasons.

`ReadBuffer` and `WriteBuffer` are just a contiguous buffer and a cursor pointing to the position in that buffer. Implementations may own or not own the memory for the buffer. There is a virtual method to fill the buffer with the following data (for `ReadBuffer`) or to flush the buffer somewhere (for `WriteBuffer`). The virtual methods are rarely called.

Implementations of `ReadBuffer/WriteBuffer` are used for working with files and file descriptors and network

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 `IDataTypes::serializeTextJSON` method with a reference to `IColumn` and the row number as arguments. Consequently, `IDataTypes::serializeTextJSON` will call a method from `WriteHelpers.h`: for example, `writeText` for numeric types and `writeJSONString` for `DataTypeString`.

## Tables

Tables are represented by the `IStorage` interface. Different implementations of that interface are different table engines. Examples are `StorageMergeTree`, `StorageMemory`, and so on. Instances of these classes are just tables.

The most important `IStorage` methods are `read` and `write`. There are also `alter`, `rename`, `drop`, and so on. The `read` method accepts the following arguments: the set of columns to read from a table, the `AST` query to consider, and the desired number of streams to return. It returns one or multiple `IBlockInputStream` objects and information about the stage of data processing that was completed inside a table engine during query execution.

In most cases, the `read` method is only responsible for reading the specified columns from a table, not for any further data processing. All further data processing is done by the query interpreter and is outside the responsibility of `IStorage`.

But there are notable exceptions:

- The `AST` query is passed to the `read` method and the table engine can use it to derive index usage and to read less data from a table.
- Sometimes the table engine can process data itself to a specific stage. For example, `StorageDistributed` can send a query to remote servers, ask them to process data to a stage where data from different remote servers can be merged, and return that preprocessed data.

The query interpreter then finishes processing the data.

The table's `read` method can return multiple `IBlockInputStream` objects to allow parallel data processing. These multiple block input streams can read from a table in parallel. Then you can wrap these streams with various transformations (such as expression evaluation or filtering) that can be calculated independently and create a `UnionBlockInputStream` on top of them, to read from multiple streams in parallel.

There are also `TableFunctions`. These are functions that return a temporary `IStorage` object to use in the `FROM` clause of a query.

To get a quick idea of how to implement your own table engine, look at something simple, like `StorageMemory` or `StorageTinyLog`.

As the result of the `read` method, `IStorage` returns `QueryProcessingStage` – information about what parts of the query were already calculated inside storage. Currently we have only very coarse granularity for that information. There is no way for the storage to say "I have already processed this part of the expression in `WHERE`, for this range of data". We need to work on that.

# Parsers

A query is parsed by a hand-written recursive descent parser. For example, `ParserSelectQuery` just recursively calls the underlying parsers for various parts of the query. Parsers create an `AST`. The `AST` is represented by nodes, which are instances of `IAST`.

Parser generators are not used for historical reasons.

# Interpreters

Interpreters are responsible for creating the query execution pipeline from an `AST`. There are simple interpreters, such as `InterpreterExistsQuery` and `InterpreterDropQuery`, or the more sophisticated `InterpreterSelectQuery`. The query execution pipeline is a combination of block input or output streams. For example, the result of interpreting the `SELECT` query is the `IBlockInputStream` to read the result set from; the result of the `INSERT` query is the `IBlockOutputStream` to write data for insertion to; and the result of interpreting the `INSERT SELECT` query is the `IBlockInputStream` that returns an empty result set on the first read, but that copies data from `SELECT` to `INSERT` at the same time.

`InterpreterSelectQuery` uses `ExpressionAnalyzer` and `ExpressionActions` machinery for query analysis and transformations. This is where most rule-based query optimizations are done. `ExpressionAnalyzer` is quite messy and should be rewritten: various query transformations and optimizations should be extracted to separate classes to allow modular transformations or query.

# Functions

There are ordinary functions and aggregate functions. For aggregate functions, see the next section.

Ordinary functions don't change the number of rows – they work as if they are processing each row independently. In fact, functions are not called for individual rows, but for `Block`'s of data to implement vectorized query execution.

There are some miscellaneous functions, like `blockSize`, `rowNumberInBlock`, and `runningAccumulate`, that exploit block processing and violate the independence of rows.

ClickHouse has strong typing, so implicit type conversion doesn't occur. If a function doesn't support a specific combination of types, an exception will be thrown. But functions can work (be overloaded) for many different combinations of types. For example, the `plus` function (to implement the `+` operator) works for any combination of numeric types: `UInt8 + Float32`, `UInt16 + Int8`, and so on. Also, some variadic functions can accept any number of arguments, such as the `concat` function.

Implementing a function may be slightly inconvenient because a function explicitly dispatches supported data types and supported `IColumns`. For example, the `plus` function has code generated by instantiation of a C++ template for each combination of numeric types, and for constant or non-constant left and right arguments.

This is a nice place to implement runtime code generation to avoid template code bloat. Also, it will make it possible to add fused functions like fused multiply-add, or to make multiple comparisons in one loop iteration.

Due to vectorized query execution, functions are not short-circuit. For example, if you write `WHERE f(x) AND g(y)`, both sides will be calculated, even for rows, when `f(x)` is zero (except when `f(x)` is a zero constant expression). But if selectivity of the `f(x)` condition is high, and calculation of `f(x)` is much cheaper than `g(y)`, it's better to implement multi-pass calculation: first calculate `f(x)`, then filter columns by the result, and then calculate `g(y)` only for smaller, filtered chunks of data.

# Aggregate Functions

Aggregate functions are stateful functions. They accumulate passed values into some state, and allow you to

Aggregate functions are stateful functions. They accumulate passed values into some state, and allow you to get results from that state. They are managed with the `IAggregateFunction` interface. States can be rather simple (the state for `AggregateFunctionCount` is just a single `UInt64` value) or quite complex (the state of `AggregateFunctionUniqCombined` is a combination of a linear array, a hash table and a `HyperLogLog` probabilistic data structure).

To deal with multiple states while executing a high-cardinality `GROUP BY` query, states are allocated in `Arena` (a memory pool), or they could be allocated in any suitable piece of memory. States can have a non-trivial constructor and destructor: for example, complex aggregation states can allocate additional memory themselves. This requires some attention to creating and destroying states and properly passing their ownership, to keep track of who and when will destroy states.

Aggregation states can be serialized and deserialized to pass over the network during distributed query execution or to write them on disk where there is not enough RAM. They can even be stored in a table with the `DataTypeAggregateFunction` to allow incremental aggregation of data.

The serialized data format for aggregate function states is not versioned right now. This is ok if aggregate states are only stored temporarily. But we have the `AggregatingMergeTree` table engine for incremental aggregation, and people are already using it in production. This is why we should add support for backward compatibility when changing the serialized format for any aggregate function in the future.

## Server

The server implements several different interfaces:

- An HTTP interface for any foreign clients.
- A TCP interface for the native ClickHouse client and for cross-server communication during distributed query execution.
- An interface for transferring data for replication.

Internally, it is just a basic multithreaded server without coroutines, fibers, etc. Since the server is not designed to process a high rate of simple queries but is intended to process a relatively low rate of complex queries, each of them can process a vast amount of data for analytics.

The server initializes the `Context` class with the necessary environment for query execution: the list of available databases, users and access rights, settings, clusters, the process list, the query log, and so on. This environment is used by interpreters.

We maintain full backward and forward compatibility for the server TCP protocol: old clients can talk to new servers and new clients can talk to old servers. But we don't want to maintain it eternally, and we are removing support for old versions after about one year.

For all external applications, we recommend using the HTTP interface because it is simple and easy to use. The TCP protocol is more tightly linked to internal data structures: it uses an internal format for passing blocks of data and it uses custom framing for compressed data. We haven't released a C library for that protocol because it requires linking most of the ClickHouse codebase, which is not practical.

## Distributed Query Execution

Servers in a cluster setup are mostly independent. You can create a `Distributed` table on one or all servers in a cluster. The `Distributed` table does not store data itself – it only provides a "view" to all local tables on multiple nodes of a cluster. When you `SELECT` from a `Distributed` table, it rewrites that query, chooses remote nodes according to load balancing settings, and sends the query to them. The `Distributed` table requests remote servers to process a query just up to a stage where intermediate results from different servers can be merged. Then it receives the intermediate results and merges them. The distributed table tries to distribute as much work as possible to remote servers, and does not send much intermediate data over the network.

Things become more complicated when you have subqueries in IN or JOIN clauses and each of them uses a Distributed table. We have different strategies for execution of these queries.

There is no global query plan for distributed query execution. Each node has its own local query plan for its part of the job. We only have simple one-pass distributed query execution: we send queries for remote nodes and then merge the results. But this is not feasible for difficult queries with high cardinality GROUP BYs or with a large amount of temporary data for JOIN: in such cases, we need to "reshuffle" data between servers, which requires additional coordination. ClickHouse does not support that kind of query execution, and we need to work on it.

## Merge Tree

MergeTree is a family of storage engines that supports indexing by primary key. The primary key can be an arbitrary tuple of columns or expressions. Data in a MergeTree table is stored in "parts". Each part stores data in the primary key order (data is ordered lexicographically by the primary key tuple). All the table columns are stored in separate column.bin files in these parts. The files consist of compressed blocks. Each block is usually from 64 KB to 1 MB of uncompressed data, depending on the average value size. The blocks consist of column values placed contiguously one after the other. Column values are in the same order for each column (the order is defined by the primary key), so when you iterate by many columns, you get values for the corresponding rows.

The primary key itself is "sparse". It doesn't address each single row, but only some ranges of data. A separate primary.idx file has the value of the primary key for each N-th row, where N is called index\_granularity (usually, N = 8192). Also, for each column, we have column.mrk files with "marks," which are offsets to each N-th row in the data file. Each mark is a pair: the offset in the file to the beginning of the compressed block, and the offset in the decompressed block to the beginning of data. Usually compressed blocks are aligned by marks, and the offset in the decompressed block is zero. Data for primary.idx always resides in memory and data for column.mrk files is cached.

When we are going to read something from a part in MergeTree, we look at primary.idx data and locate ranges that could possibly contain requested data, then look at column.mrk data and calculate offsets for where to start reading those ranges. Because of sparseness, excess data may be read. ClickHouse is not suitable for a high load of simple point queries, because the entire range with index\_granularity rows must be read for each key, and the entire compressed block must be decompressed for each column. We made the index sparse because we must be able to maintain trillions of rows per single server without noticeable memory consumption for the index. Also, because the primary key is sparse, it is not unique: it cannot check the existence of the key in the table at INSERT time. You could have many rows with the same key in a table.

When you INSERT a bunch of data into MergeTree, that bunch is sorted by primary key order and forms a new part. To keep the number of parts relatively low, there are background threads that periodically select some parts and merge them to a single sorted part. That's why it is called MergeTree. Of course, merging leads to "write amplification". All parts are immutable: they are only created and deleted, but not modified. When SELECT is run, it holds a snapshot of the table (a set of parts). After merging, we also keep old parts for some time to make recovery after failure easier, so if we see that some merged part is probably broken, we can replace it with its source parts.

MergeTree is not an LSM tree because it doesn't contain "memtable" and "log": inserted data is written directly to the filesystem. This makes it suitable only to INSERT data in batches, not by individual row and not very frequently – about once per second is ok, but a thousand times a second is not. We did it this way for simplicity's sake, and because we are already inserting data in batches in our applications.

MergeTree tables can only have one (primary) index: there aren't any secondary indices. It would be nice to allow multiple physical representations under one logical table, for example, to store data in more than one physical order or even to allow representations with pre-aggregated data along with original data.

There are MergeTree engines that are doing additional work during background merges. Examples are



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.

## 如何构建 ClickHouse 发布包

### 安装 Git 和 Pbuilder

```
sudo apt-get update
sudo apt-get install git pbuilder debhelper lsb-release fakeroot sudo debian-archive-keyring debian-keyring
```

### 拉取 ClickHouse 源码

```
git clone --recursive --branch stable https://github.com/yandex/ClickHouse.git
```

```
git clone --recursive --branch stable https://github.com/yandex/ClickHouse.git
cd ClickHouse
```

## 运行发布脚本

```
./release
```

## 如何在开发过程中编译 ClickHouse

以下教程是在 Ubuntu Linux 中进行编译的示例。

通过适当的更改，它应该可以适用于任何其他的 Linux 发行版。

仅支持具有 SSE 4.2 的 x86\_64。对 AArch64 的支持是实验性的。

测试是否支持 SSE 4.2，执行：

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

## 安装 Git 和 CMake

```
sudo apt-get install git cmake ninja-build
```

Or cmake3 instead of cmake on older systems.

或者在早期版本的系统中用 cmake3 替代 cmake

## 安装 GCC 7

There are several ways to do this.

### 安装 PPA 包

```
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
```

### 源码安装 gcc

请查看 [ci/build-gcc-from-sources.sh](https://github.com/yandex/ClickHouse/blob/master/ci/build-gcc-from-sources.sh)

## 使用 GCC 7 来编译

```
export CC=gcc-7
export CXX=g++-7
```

## 安装所需的工具依赖库

```
sudo apt-get install libicu-dev libreadline-dev
```

## 拉取 ClickHouse 源码

```
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.

## 编译 ClickHouse

```
mkdir build
cd build
cmake ..
ninja
cd ..
```

若要创建一个执行文件，执行 `ninja clickhouse`。

这个命令会使得 `dbms/programs/clickhouse` 文件可执行，您可以使用 `client` or `server` 参数运行。

## 在 Mac OS X 中编译 ClickHouse

ClickHouse 支持在 Mac OS X 10.12 版本中编译。若您在用更早的操作系统版本，可以尝试在指令中使用 `Gentoo Prefix` 和 `clang`。

通过适当的更改，它应该可以适用于任何其他的 Linux 发行版。

## 安装 Homebrew

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

## 安装编译器，工具库

```
brew install cmake ninja gcc icu4c mariadb-connector-c openssl libtool gettext readline
```

## 拉取 ClickHouse 源码

```
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.

## 编译 ClickHouse

```
mkdir build
cd build
cmake .. -DCMAKE_CXX_COMPILER=`which g++-8` -DCMAKE_C_COMPILER=`which gcc-8`
ninja
cd ..
```

## 注意事项

若你想运行 `clickhouse-server`，请先确保增加系统的最大文件数配置。

注意

可能需要用 `sudo`

为此，请创建以下文件：

/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>
```

执行以下命令：

```
$ sudo chown root:wheel /Library/LaunchDaemons/limit.maxfiles.plist
```

然后重启。

可以通过 `ulimit -n` 命令来检查是否生效。

## 如何编写 C++ 代码

### 一般建议

1. 以下是建议，而不是要求。
2. 如果你在修改代码，遵守已有的风格是有意义的。
3. 代码的风格需保持一致。一致的风格有利于阅读代码，并且方便检索代码。
4. 许多规则没有逻辑原因；它们是由既定的做法决定的。

### 格式化

1. 大多数格式化可以用 `clang-format` 自动完成。
2. 缩进是4个空格。配置开发环境，使得 TAB 代表添加四个空格。
3. 左右花括号需在单独的行。

```
inline void readBoolText(bool & x, ReadBuffer & buf)
{
 char tmp = '0';
 readChar(tmp, buf);
 x = tmp != '0';
}
```

4. 若整个方法体仅有一行 `描述`，则可以放到单独的行上。在花括号周围放置空格（除了行尾的空格）。

```
inline size_t mask() const { return buf_size() - 1; }
inline size_t place(HashValue x) const { return x & mask(); }
```

5. 对于函数。不要在括号周围放置空格。

```
void reinsert(const Value & x)
```

```
memcpy(&buf[place_value], &x, sizeof(x));
```

6. 在 `if`，`for`，`while` 和其他表达式中，在开括号前面插入一个空格（与函数声明相反）。

```
for (size_t i = 0; i < rows; i += storage.index_granularity)
```

7. 在二元运算符（`+`，`-`，`*`，`/`，`%`，`...`）和三元运算符 `?:` 周围添加空格。

```
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. 若有换行，新行应该以运算符开头，并且增加对应的缩进。

```
if (elapsed_ns)
 message << " ("
 << rows_read_on_server * 1000000000 / elapsed_ns << " rows/s., "
 << bytes_read_on_server * 1000.0 / elapsed_ns << " MB/s.) ";
```

9. 如果需要，可以在一行内使用空格来对齐。

```
dst.ClickLogID = click.LogID;
dst.ClickEventID = click.EventID;
dst.ClickGoodEvent = click.GoodEvent;
```

10. 不要在 `.`，`->` 周围加入空格

如有必要，运算符可以包裹到下一行。在这种情况下，它前面的偏移量增加。

11. 不要使用空格来分开一元运算符（`--`，`++`，`*`，`&`，`...`）和参数。

12. 在逗号后面加一个空格，而不是在之前。同样的规则也适合 `for` 循环中的分号。

13. 不要用空格分开 `[]` 运算符。

14. 在 `template <...>` 表达式中，在 `template` 和 `<` 中加入一个空格，在 `<` 后面或在 `>` 前面都不要有空格。

```
template <typename TKey, typename TValue>
struct AggregatedStatElement
{ }
```

15. 在类和结构体中，`public`，`private` 以及 `protected` 同 `class/struct` 无需缩进，其他代码须缩进。

```
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.** 如果对整个文件使用相同的 `namespace`，并且没有其他重要的东西，则 `namespace` 中不需要偏移量。

**17.** 在 `if`, `for`, `while` 中包裹的代码块中，若代码是一个单行的 `statement`，那么大括号是可选的。可以将 `statement` 放到一行中。这个规则同样适用于嵌套的 `if`，`for`，`while`，...

但是如果内部 `statement` 包含大括号或 `else`，则外部块应该用大括号括起来。

```
/// Finish write.
for (auto & stream : streams)
 stream.second->finalize();
```

**18.** 行的某尾不应该包含空格。

**19.** 源文件应该用 UTF-8 编码。

**20.** 非ASCII字符可用于字符串文字。

```
<< ", " << (timer.elapsed() / chunks_stats.hits) << " μsec/hit.";
```

**21.** 不要在一行中写入多个表达式。

**22.** 将函数内部的代码段分组，并将它们与不超过一行的空行分开。

**23.** 将 函数，类用一个或两个空行分开。

**24.** `const` 必须写在类型名称之前。

```
//correct
const char * pos
const std::string & s
//incorrect
char const * pos
```

**25.** 声明指针或引用时，`*` 和 `&` 符号两边应该都用空格分隔。

```
//correct
const char * pos
//incorrect
const char* pos
const char *pos
```

**26.** 使用模板类型时，使用 `using` 关键字对它们进行别名（最简单的情况除外）。

换句话说，模板参数仅在 `using` 中指定，并且不在代码中重复。

`using` 可以在本地声明，例如在函数内部。

```
//correct
```

```
//correct
using FileStreams = std::map<std::string, std::shared_ptr<Stream>>;
FileStreams streams;
//incorrect
std::map<std::string, std::shared_ptr<Stream>> streams;
```

**27.** 不要在一个语句中声明不同类型的多个变量。

```
//incorrect
int x, *y;
```

**28.** 不要使用C风格类型转换。

```
//incorrect
std::cerr << (int)c << std::endl;
//correct
std::cerr << static_cast<int>(c) << std::endl;
```

**29.** 在类和结构中，组成员和函数分别在每个可见范围内。

**30.** 对于小类和结构，没有必要将方法声明与实现分开。

对于任何类或结构中的小方法也是如此。

对于模板化类和结构，不要将方法声明与实现分开（因为否则它们必须在同一个转换单元中定义）

**31.** 您可以将换行规则定在140个字符，而不是80个字符。

**32.** 如果不需要 postfix，请始终使用前缀增量/减量运算符。

```
for (Names::const_iterator it = column_names.begin(); it != column_names.end(); ++it)
```

## Comments

**1.** 请务必为所有非常重要的代码部分添加注释。

这是非常重要的。编写注释可能会帮助您意识到代码不是必需的，或者设计错误。

```
/** 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.** 注释可以尽可能详细。

**3.** 在他们描述的代码之前放置注释。在极少数情况下，注释可以在代码之后，在同一行上。

```
/** 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. 注释应该只用英文撰写。
5. 如果您正在编写库，请在主头文件中包含解释它的详细注释。
6. 请勿添加无效的注释。特别是，不要留下像这样的空注释：

```
/*
 * 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:
 */
```

这个示例来源于 <http://home.tamk.fi/~jaalto/course/coding-style/doc/unmaintainable-code/>。

7. 不要在每个文件的开头写入垃圾注释（作者，创建日期...）。
8. 单行注释用三个斜杆：`///`，多行注释以`/**`开始。这些注释会当做文档。

注意：您可以使用 Doxygen 从这些注释中生成文档。但是通常不使用 Doxygen，因为在 IDE 中导航代码更方便。

9. 多行注释的开头和结尾不得有空行（关闭多行注释的行除外）。
10. 要注释掉代码，请使用基本注释，而不是“记录”注释。
11. 在提交之前删除代码的无效注释部分。
12. 不要在注释或代码中使用亵渎语言。
13. 不要使用大写字母。不要使用过多的标点符号。

```
/// WHAT THE FAIL???
```

14. 不要使用注释来制作分隔符。

```
///*****
```

15. 不要在注释中开始讨论。

```
/// Why did you do this stuff?
```

16. 没有必要在块的末尾写一条注释来描述它的含义。

```
/// for
```

## Names

1. 在变量和类成员的名称中使用带下划线的小写字母。

```
size_t max_block_size;
```

2. 对于函数（方法）的名称，请使用以小写字母开头的驼峰标识。

```
std::string getName() const override { return "Memory"; }
```

3. 对于类（结构）的名称，使用以大写字母开头的驼峰标识。接口名称用I前缀。

```
class StorageMemory : public IStorage
```

4. `using` 的命名方式与类相同，或者以 `__t`` 命名。

5. 模板类型参数的名称：在简单的情况下，使用 `T`；`T`，`U`；`T1`，`T2`。

对于更复杂的情况，要么遵循类名规则，要么添加前缀 `T`。

```
template <typename TKey, typename TValue>
struct AggregatedStatElement
```

6. 模板常量参数的名称：遵循变量名称的规则，或者在简单的情况下使用 `N`。

```
template <bool without_www>
struct ExtractDomain
```

7. 对于抽象类型（接口），用 `I` 前缀。

```
class IBlockInputStream
```

8. 如果在本地使用变量，则可以使用短名称。

在所有其他情况下，请使用能描述含义的名称。

```
bool info_successfully_loaded = false;
```

9. `define` 和全局常量的名称使用带下划线的 `ALL_CAPS`。

```
##define MAX_SRC_TABLE_NAMES_TO_STORE 1000
```

10. 文件名应使用与其内容相同的样式。

如果文件包含单个类，则以与该类名称相同的方式命名该文件。

如果文件包含单个函数，则以与函数名称相同的方式命名文件。

11. 如果名称包含缩写，则：

- 对于变量名，缩写应使用小写字母 `mysql_connection`（不是 `mysql_connection`）。
- 对于类和函数的名称，请将大写字母保留在缩写 `MySQLConnection`（不是 `MySqlConnection`）。

**12.** 仅用于初始化类成员的构造方法参数的命名方式应与类成员相同，但最后使用下划线。

```
FileQueueProcessor(
 const std::string & path_,
 const std::string & prefix_,
 std::shared_ptr<FileHandler> handler_
 : path(path_),
 prefix(prefix_),
 handler(handler_),
 log(&Logger::get("FileQueueProcessor"))
{
}
```

如果构造函数体中未使用该参数，则可以省略下划线后缀。

**13.** 局部变量和类成员的名称没有区别（不需要前缀）。

```
timer (not m_timer)
```

**14.** 对于 `enum` 中的常量，请使用带大写字母的驼峰标识。`ALL_CAPS` 也可以接受。如果 `enum` 是非本地的，请使用 `enum class`。

```
enum class CompressionMethod
{
 QuickLZ = 0,
 LZ4 = 1,
};
```

**15.** 所有名字必须是英文。不允许音译俄语单词。

```
not Stroka
```

**16.** 缩写须是众所周知的（当您可以在维基百科或搜索引擎中轻松找到缩写的含义时）。

```
`AST`, `SQL`.

Not `NVDH` (some random letters)
```

如果缩短版本是常用的，则可以接受不完整的单词。

如果注释中旁边包含全名，您也可以使用缩写。

**17.** C++ 源码文件名称必须为 `.cpp` 拓展名。头文件必须为 `.h` 拓展名。

## 如何编写代码

**1.** 内存管理。

手动内存释放 (`delete`) 只能在库代码中使用。

在库代码中，`delete` 运算符只能在析构函数中使用。

在应用程序代码中，内存必须由拥有它的对象释放。



示例：

- 最简单的方法是将对象放在堆栈上，或使其成为另一个类的成员。
- 对于大量小对象，请使用容器。
- 对于自动释放少量在堆中的对象，可以用 `shared_ptr/unique_ptr`。

## 2. 资源管理。

使用 `RAII` 以及查看以上说明。

## 3. 错误处理。

在大多数情况下，您只需要抛出一个异常，而不需要捕获它（因为 `RAII`）。

在离线数据处理应用程序中，通常可以接受不捕获异常。

在处理用户请求的服务器中，通常足以捕获连接处理程序顶层的异常。

在线程函数中，你应该在 `join` 之后捕获并保留所有异常以在主线程中重新抛出它们。

```
/// 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();
```

不处理就不要隐藏异常。永远不要盲目地把所有异常都记录到日志中。

```
//Not correct
catch (...) {}
```

如果您需要忽略某些异常，请仅针对特定异常执行此操作并重新抛出其余异常。

```
catch (const DB::Exception & e)
{
 if (e.code() == ErrorCodes::UNKNOWN_AGGREGATE_FUNCTION)
 return nullptr;
 else
 throw;
}
```

当使用具有返回码或 `errno` 的函数时，请始终检查结果并在出现错误时抛出异常。

```
if (0 != close(fd))
 throwFromErrno("Cannot close file " + file_name, ErrorCodes::CANNOT_CLOSE_FILE);
```

不要使用断言。

## 4. 异常类型。

不需要在应用程序代码中使用复杂的异常层次结构。系统管理员应该可以理解异常文本。

## 5. 从析构函数中抛出异常。

不建议这样做，但允许这样做。

按照以下选项：

- 创建一个函数（ `done()` 或 `finalize()` ），它将提前完成所有可能导致异常的工作。如果调用了该函数，则稍后在析构函数中应该没有异常。
- 过于复杂的任务（例如通过网络发送消息）可以放在单独的方法中，类用户必须在销毁之前调用它们。
- 如果析构函数中存在异常，则最好记录它而不是隐藏它（如果 `logger` 可用）。
- 在简单的应用程序中，依赖于 `std::terminate`（对于C++ 11中默认情况下为 `noexcept` 的情况）来处理异常是可以接受的。

## 6. 匿名代码块。

您可以在单个函数内创建单独的代码块，以使某些变量成为局部变量，以便在退出块时调用析构函数。

```
Block block = data.in->read();

{
 std::lock_guard<std::mutex> lock(mutex);
 data.ready = true;
 data.block = block;
}

ready_any.set();
```

## 7. 多线程。

在离线数据处理程序中：

- 尝试在单个CPU核心上获得最佳性能。然后，您可以根据需要并行化代码。

在服务端应用中：

- 使用线程池来处理请求。此时，我们还没有任何需要用户空间上下文切换的任务。

Fork不用于并行化。

## 8. 同步线程。

通常可以使不同的线程使用不同的存储单元（甚至更好：不同的缓存线），并且不使用任何线程同步（除了 `joinAll`）。

如果需要同步，在大多数情况下，在 `lock_guard` 下使用互斥量就足够了。

在其他情况下，使用系统同步原语。不要使用忙等待。

仅在最简单的情况下才应使用原子操作。

除非是您的主要专业领域，否则不要尝试实施无锁数据结构。

## 9. 指针和引用。

大部分情况下，请用引用。

## 10. 常量。

使用 `const` 引用，指向常量的指针，`const_iterator` 和 `const` 指针。

将 `const` 视为默认值，仅在必要时使用非 `const`。

当按值传递变量时，使用 `const` 通常没有意义。

11. 无符号。

必要时使用 `unsigned`。

12. 数值类型。

使用 `UInt8`，`UInt16`，`UInt32`，`UInt64`，`Int8`，`Int16`，`Int32`，以及 `Int64`，`size_t`，`ssize_t` 还有 `ptrdiff_t`。

不要使用这些类型：`signed / unsigned long`，`long long`，`short`，`signed / unsigned char`，`char`。

13. 参数传递。

通过引用传递复杂类型（包括 `std::string`）。

如果函数中传递堆中创建的对象，则使参数类型为 `shared_ptr` 或者 `unique_ptr`。

14. 返回值

大部分情况下使用 `return`。不要使用 `[return std::move(res)][.strike]`。

如果函数在堆上分配对象并返回它，请使用 `shared_ptr` 或 `unique_ptr`。

在极少数情况下，您可能需要通过参数返回值。在这种情况下，参数应该是引用传递的。

```
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. 命名空间。

没有必要为应用程序代码使用单独的 `namespace`。

小型库也不需要这个。

对于中大型库，须将所有代码放在 `namespace` 中。

在库的 `.h` 文件中，您可以使用 `namespace detail` 来隐藏应用程序代码不需要的实现细节。

在 `.cpp` 文件中，您可以使用 `static` 或匿名命名空间来隐藏符号。

同样 `namespace` 可用于 `enum` 以防止相应的名称落入外部 `namespace`（但最好使用 `enum class`）。

16. 延迟初始化。

如果初始化需要参数，那么通常不应该编写默认构造函数。

如果稍后您需要延迟初始化，则可以添加将创建无效对象的默认构造函数。或者，对于少量对象，您可以使用 `shared_ptr` / `unique_ptr`。

```
Loader(DB::Connection * connection_, const std::string & query, size_t max_block_size);

/// For deferred initialization
Loader() {}
```

## 17. 虚函数。

如果该类不是用于多态使用，则不需要将函数设置为虚拟。这也适用于析构函数。

## 18. 编码。

在所有情况下使用 UTF-8 编码。使用 `std::string` 和 `char *`。不要使用 `std::wstring` 和 `wchar_t`。

## 19. 日志。

请参阅代码中的示例。

在提交之前，删除所有无意义和调试日志记录，以及任何其他类型的调试输出。

应该避免循环记录日志，即使在 `Trace` 级别也是如此。

日志必须在任何日志记录级别都可读。

在大多数情况下，只应在应用程序代码中使用日志记录。

日志消息必须用英文写成。

对于系统管理员来说，日志最好是可以理解的。

不要在日志中使用衰读语言。

在日志中使用 UTF-8 编码。在极少数情况下，您可以在日志中使用非 ASCII 字符。

## 20. 输入-输出。

不要使用 `iostreams` 在对应用程序性能至关重要的内部循环中（并且永远不要使用 `stringstream`）。

使用 `DB/IO` 库替代。

## 21. 日期和时间。

参考 `DateLUT` 库。

## 22. 引入头文件。

一直用 `#pragma once` 而不是其他宏。

## 23. using 语法

`using namespace` 不会被使用。您可以使用特定的 `using`。但是在类或函数中使它成为局部的。

## 24. 不要使用 `trailing return type` 为必要的功能。

```
[auto f() -> void;]{.strike}
```

## 25. 声明和初始化变量。

```
//right way
std::string s = "Hello";
std::string s{"Hello"};

//wrong way
auto s = std::string{"Hello"};
```

## 26. 对于虚函数，在基类中编写 `virtual`，但在后代类中写 `override` 而不是 `virtual`。

没有用到的 C++ 特性。

1. 不使用虚拟继承。
2. 不使用 C++03 中的异常标准。

## 平台

1. 我们为特定平台编写代码。

但在其他条件相同的情况下，首选跨平台或可移植代码。

2. 语言：C++17.
3. 编译器：gcc。此时（2017年12月），代码使用7.2版编译。（它也可以使用clang 4 编译）  
使用标准库（libstdc++ 或 libc++）。
4. 操作系统：Linux Ubuntu，不比 Precise 早。
5. 代码是为x86\_64 CPU架构编写的。  
CPU指令集是我们服务器中支持的最小集合。目前，它是SSE 4.2。
6. 使用 -Wall -Wextra -Werror 编译参数。
7. 对所有库使用静态链接，除了那些难以静态连接的库（参见 ldd 命令的输出）。
8. 使用发布的设置来开发和调试代码。

## 工具

1. KDevelop 是一个好的 IDE.
2. 调试可以使用 gdb，valgrind (memcheck)，strace，-fsanitize=...，或 tcmalloc\_minimal\_debug.
3. 对于性能分析，使用 Linux Perf，valgrind (callgrind)，或者 strace -cf。
4. 源代码用 Git 作版本控制。
5. 使用 CMake 构建。
6. 程序的发布使用 deb 安装包。
7. 提交到 master 分支的代码不能破坏编译。  
虽然只有选定的修订被认为是可行的。
8. 尽可能经常地进行提交，即使代码只是部分准备好了。  
目的明确的功能，使用分支。  
如果 master 分支中的代码尚不可构建，请在 push 之前将其从构建中排除。您需要在几天内完成或删除它。
9. 对于不重要的更改，请使用分支并在服务器上发布它们。
10. 未使用的代码将从 repo 中删除。

## 库

1. 使用C++ 14标准库（允许实验性功能），以及 boost 和 Poco 框架。
2. 如有必要，您可以使用 OS 包中提供的任何已知库。

如果有一个好的解决方案已经可用，那就使用它，即使这意味着你必须安装另一个库。

（但永远不要删除不好的库）

3. 如果软件包没有您需要的软件包或者有过时的版本或错误的编译类型，则可以安装不在软件包中的库。
4. 如果库很小并且没有自己的复杂构建系统，请将源文件放在 `contrib` 文件夹中。
5. 始终优先考虑已经使用的库。

## 一般建议

1. 尽可能精简代码。
2. 尝试用最简单的方式实现。
3. 在你知道代码是如何工作以及内部循环如何运作之前，不要编写代码。
4. 在最简单的情况下，使用 `using` 而不是类或结构。
5. 如果可能，不要编写复制构造函数，赋值运算符，析构函数（虚拟函数除外，如果类包含至少一个虚函数），移动构造函数或移动赋值运算符。换句话说，编译器生成的函数必须正常工作。您可以使用 `default`。
6. 鼓励简化代码。尽可能减小代码的大小。

## 其他建议

1. 从 `stddef.h` 明确指定 `std::` 的类型。

不推荐。换句话说，我们建议写 `size_t` 而不是 `std::size_t`，因为它更短。

也接受添加 `std::`。

2. 为标准C库中的函数明确指定 `std::`

不推荐。换句话说，写 `memcpy` 而不是 `std::memcpy`。

原因是有类似的非标准功能，例如 `memmem`。我们偶尔会使用这些功能。`namespace std` 中不存在这些函数。

如果你到处都写 `std::memcpy` 而不是 `memcpy`，那么没有 `std::` 的 `memmem` 会显得很奇怪。

不过，如果您愿意，仍然可以使用 `std::`。

3. 当标准C++库中提供相同的函数时，使用C中的函数。

如果它更高效，这是可以接受的。

例如，使用 `memcpy` 而不是 `std::copy` 来复制大块内存。

4. 函数的多行参数。

允许以下任何包装样式：

```
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)
```

```
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 测试

### 功能性测试

功能性测试是最简便使用的。绝大部分 ClickHouse 的功能可以通过功能性测试来测试，任何代码的更改都必须通过该测试。

每个功能测试会向正在运行的 ClickHouse 服务器发送一个或多个查询，并将结果与预期结果进行比较。

测试用例在 `dbms/src/tests/queries` 目录中。这里有两个子目录：`stateless` 和 `stateful` 目录。无状态的测试无需预加载测试数据集 - 通常是在测试运行期间动态创建小量的数据集。有状态测试需要来自 Yandex.Metrica 的预加载测试数据，而不向一般公众提供。我们倾向于仅使用“无状态”测试并避免添加新的“有状态”测试。

每个测试用例可以是两种类型之一：`.sql` 和 `.sh`。`.sql` 测试文件是用于管理 `clickhouse-client --multiquery --testmode` 的简单 SQL 脚本。`.sh` 测试文件是一个可以自己运行的脚本。

要运行所有测试，请使用 `dbms/tests/clickhouse-test` 工具，用 `--help` 可以获取所有的选项列表。您可以简单地运行所有测试或运行测试名称中的子字符串过滤的测试子集：`./clickhouse-test substring`。

调用功能测试最简单的方法是将 `clickhouse-client` 复制到 `/usr/bin/`，运行 `clickhouse-server`，然后从自己的目录运行 `./clickhouse-test`。

要添加新测试，请在 `dbms/src/tests/queries/0_stateless` 目录内添加新的 `.sql` 或 `.sh` 文件，手动检查，然后按以下方式生成 `.reference` 文件：`clickhouse-client -n --testmode < 00000_test.sql > 00000_test.reference` or `./00000_test.sh > ./00000_test.reference`。

测试应该只使用（创建，删除等）`test` 数据库中的表，这些表假定是事先创建的；测试也可以使用临时表。

如果要在功能测试中使用分布式查询，可以利用 `remote` 表函数和 `127.0.0.{1..2}` 地址为服务器查询自身；或者您可以在服务器配置文件中使用的预定义的测试集群，例如 `test_shard_localhost`。

有些测试在名称中标有 `zookeeper`，`shard` 或 `long`。`zookeeper` 用于使用 ZooKeeper 的测试；`shard` 用于需要服务器监听 `127.0.0.*` 的测试。`long` 适用于运行时间稍长一秒的测试。

### 已知的bug

如果我们知道一些可以通过功能测试轻松复制的错误，我们将准备好的功能测试放在 `dbms/src/tests/queries/bugs` 目录中。当修复错误时，这些测试将被移动到 `dbms/src/tests/queries/0_stateless` 目录中。

### 集成测试

集成测试允许在集群配置中测试 ClickHouse，并与其他服务器（如 MySQL，Postgres，MongoDB）进行 ClickHouse 交互。它们可用于模拟网络拆分，数据包丢弃等。这些测试在 Docker 下运行，并使用各种软件创建多个容器。

参考 `dbms/tests/integration/README.md` 文档关于如何使用集成测试。



请注意，ClickHouse 与第三方驱动程序的集成未经过测试。此外，我们目前还没有与 JDBC 和 ODBC 驱动程序进行集成测试。

## 单元测试

当您想要测试整个 ClickHouse，而不是单个独立的库或类时，单元测试非常有用。您可以使用 `ENABLE_TESTS` CMake 选项启用或禁用测试构建。单元测试（和其他测试程序）位于代码中的 `tests` 子目录中。要运行单元测试，请键入 `ninja test`。有些测试使用 `gtest`，但有些只是在测试失败时返回非零状态码。

如果代码已经被功能测试覆盖（并且功能测试通常使用起来要简单得多），则不一定要进行单元测试。

## 性能测试

性能测试允许测量和比较综合查询中 ClickHouse 的某些独立部分的性能。测试位于 `dbms/tests/performance` 目录中。每个测试都由 `.xml` 文件表示，并附有测试用例的描述。使用 `clickhouse performance-test` 工具（嵌入在 `clickhouse` 二进制文件中）运行测试。请参阅 `--help` 以进行调用。

每个测试在循环中运行一个或多个查询（可能带有参数组合），并具有一些停止条件（如“最大执行速度不会在三秒内更改”）并测量一些有关查询性能的指标（如“最大执行速度”）。某些测试可以包含预加载的测试数据集的前提条件。

如果要在某些情况下提高 ClickHouse 的性能，并且如果可以在简单查询上观察到改进，则强烈建议编写性能测试。在测试过程中使用 `perf top` 或其他 `perf` 工具总是有意义的。

性能测试不是基于每个提交运行的。不收集性能测试结果，我们手动比较它们。

## 测试工具和脚本

`tests` 目录中的一些程序不是准备测试，而是测试工具。例如，对于 `Lexer`，有一个工具 `dbms/src/Parsers/tests/lexer` 标准输出。您可以使用这些工具作为代码示例以及探索和手动测试。

您还可以将一对文件 `.sh` 和 `.reference` 与工具放在一些预定义的输入上运行它 - 然后将脚本结果与 `.reference` 文件进行比较。这些测试不是自动化的。

## 杂项测试

有一些外部字典的测试位于 `dbms/tests/external_dictionaries`，机器学习模型在 `dbms/tests/external_models` 目录。这些测试未更新，必须转移到集成测试。

对于分布式数据的插入，有单独的测试。此测试在单独的服务器上运行 ClickHouse 集群并模拟各种故障情况：网络拆分，数据包丢失（ClickHouse 节点之间，ClickHouse 和 ZooKeeper 之间，ClickHouse 服务器和客户端之间等），进行 `kill -9`，`kill -STOP` 和 `kill -CONT` 等操作，类似 `Jepsen`。然后，测试检查是否已写入所有已确认的插入，并且所有已拒绝的插入都未写入。

在 ClickHouse 开源之前，分布式测试是由单独的团队编写的，但该团队不再使用 ClickHouse，测试是在 Java 中意外编写的。由于这些原因，必须重写分布式测试并将其移至集成测试。

## 手动测试

当您开发了新的功能，做手动测试也是合理的。可以按照以下步骤来进行：

编译 ClickHouse。在命令行中运行 ClickHouse：进入 `dbms/src/programs/clickhouse-server` 目录并运行 `./clickhouse-server`。它会默认使用当前目录的配置文件 (`config.xml`，`users.xml` 以及在 `config.d` 和 `users.d` 目录的文件)。可以使用 `dbms/src/programs/clickhouse-client/clickhouse-client` 来连接数据库。

或者，您可以安装 ClickHouse 软件包：从 Yandex 存储库中获得稳定版本，或者您可以在 ClickHouse 源根目录中使用 `./release` 构建自己的软件包。然后使用 `sudo service clickhouse-server start` 启动服务器（或停止服务器）。在 `/etc/clickhouse-server/clickhouse-server.log` 中查找日志。

当您的系统上已经安装了 ClickHouse 时，您可以构建一个新的 `clickhouse` 二进制文件并替换现有的二进制文件：

```
sudo service clickhouse-server stop
```



```
sudo service clickhouse-server stop
sudo cp ./clickhouse /usr/bin/
sudo service clickhouse-server start
```

您也可以停止 `clickhouse-server` 并使用相同的配置运行您自己的服务器，日志打印到终端：

```
sudo service clickhouse-server stop
sudo -u clickhouse /usr/bin/clickhouse server --config-file /etc/clickhouse-server/config.xml
```

使用 `gdb` 的一个示例：

```
sudo -u clickhouse gdb --args /usr/bin/clickhouse server --config-file /etc/clickhouse-server/config.xml
```

如果 `clickhouse-server` 已经运行并且您不想停止它，您可以更改 `config.xml` 中的端口号（或在 `config.d` 目录中的文件中覆盖它们），配置适当的数据路径，然后运行它。

`clickhouse` 二进制文件几乎没有依赖关系，适用于各种 Linux 发行版。要快速地测试服务器上的更改，您可以简单地将新建的 `clickhouse` 二进制文件 `scp` 到其他服务器，然后按照上面的示例运行它。

## 测试环境

在将版本发布为稳定之前，我们将其部署在测试环境中 测试环境是一个处理[Yandex.Metrica]（<https://metrica.yandex.com/>）总数据的1/39部分大小的集群。我们与 Yandex.Metrica 团队公用我们的测试环境。ClickHouse 在现有数据的基础上无需停机即可升级。我们首先看到数据处理成功而不会实时滞后，复制继续工作，并且 Yandex.Metrica 团队无法看到问题。首先的检查可以通过以下方式完成：

```
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;
```

在某些情况下，我们还部署到 Yandex 的合作团队的测试环境：市场，云等。此外，我们还有一些用于开发目的的硬件服务器。

## 负载测试

部署到测试环境后，我们使用生产群集中的查询运行负载测试。这是手动完成的。

确保在生产集群中开启了 `query_log` 选项。

收集一天或更多的查询日志：

```
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
```

这是一个复杂的例子。`type = 2` 将过滤成功执行的查询。`query LIKE '%ym:%'` 用于从 Yandex.Metrica 中选择相关查询。`is_initial_query` 是仅选择由客户端发起的查询，而不是由 ClickHouse 本身（作为分布式查询处理的一部分）。

`scp` 这份日志到测试机器，并运行以下操作：

```
clickhouse benchmark --concurrency 16 < queries.tsv
```

（可能你需要指定运行的用户 `--user`）

然后离开它一晚或周末休息一下。

你要检查下 `clickhouse-server` 是否崩溃，内存占用是否合理，性能也不会随着时间的推移而降低。

由于查询和环境的高度可变性，不会记录精确的查询执行时序并且不进行比较。

## 编译测试

构建测试允许检查构建在各种替代配置和某些外部系统上是否被破坏。测试位于 `ci` 目录。它们从 Docker，Vagrant 中的源代码运行构建，有时在 Docker 中运行 `qemu-user-static`。这些测试正在开发中，测试运行不是自动化的。

动机：

通常我们会在 ClickHouse 构建的单个版本上发布并运行所有测试。但是有一些未经过彻底测试的替代构建版本。例子：

- 在 FreeBSD 中的构建；
- 在 Debian 中使用系统包中的库进行构建；
- 使用库的共享链接构建；
- 在 AArch64 平台进行构建。

例如，使用系统包构建是不好的做法，因为我们无法保证系统具有的确切版本的软件包。但 Debian 维护者确实需要这样做。出于这个原因，我们至少必须支持这种构建。另一个例子：共享链接是一个常见的麻烦来源，但是对于一些爱好者来说需要它。

虽然我们无法对所有构建版本运行所有测试，但我们想要检查至少不会破坏各种构建变体。为此，我们使用构建测试。

## 测试协议兼容性

当我们扩展 ClickHouse 网络协议时，我们手动测试旧的 `clickhouse-client` 与新的 `clickhouse-server` 和新的 `clickhouse-client` 一起使用旧的 `clickhouse-server` (只需从相应的包中运行二进制文件)

## 来自编译器的帮助

ClickHouse 主要的代码 (位于 `dbms` 目录中) 使用 `-Wall -Wextra -Werror` 构建，并带有一些其他已启用的警告。虽然没有为第三方库启用这些选项。

Clang 有更多有用的警告 - 您可以使用 `-Weverything` 查找它们并选择默认构建的东西。

对于生产构建，使用 `gcc` (它仍然生成比 `clang` 稍高效的代码)。对于开发来说，`clang` 通常更方便使用。您可以使用调试模式在自己的机器上构建 (以节省笔记本电脑的电量)，但请注意，由于更好的控制流程和过程分析，编译器使用 `-O3` 会生成更多警告。当使用 `clang` 构建时，使用 `libc++` 而不是 `libstdc++`，并且在使用调试模式构建时，使用调试版本的 `libc++`，它允许在运行时捕获更多错误。

## Sanitizers

### Address sanitizer.

我们在每个提交的基础上在 ASan 下运行功能和集成测试。

### Valgrind (Memcheck).

我们在 Valgrind 过夜进行功能测试。这需要几个小时。目前在 `re2` 库中有一个已知的误报，请参阅 [文章](#)。

### Thread sanitizer.

我们在 TSan 下进行功能测试。ClickHouse 必须通过所有测试。在 TSan 下运行不是自动化的，只是偶尔执行。

### Memory sanitizer.

目前我们不使用 MSan。

### Undefined behaviour sanitizer.

我们仍然不会在每次提交的基础上使用 UBSan。有一些地方需要解决。

### Debug allocator.

您可以使用 `DEBUG_TCMALLOC` CMake 选项启用 `tcmalloc` 的调试版本。我们在每次提交的基础上使用调试分配器运行测试。

更多请参阅 `dbms/tests/instructions/sanitizers.txt`。

## 模糊测试

我们使用简单的模糊测试来生成随机SQL查询并检查服务器是否正常，使用 `Address sanitizer` 执行模糊测试。您可以在 `00746_sql_fuzzy.pl` 找到它。测试应连续进行（过夜和更长时间）。

截至2018年12月，我们仍然不使用库代码的孤立模糊测试。

## 安全审计

Yandex Cloud 部门的人员从安全角度对 ClickHouse 功能进行了一些基本概述。

## 静态分析

我们偶尔使用静态分析。我们已经评估过 `clang-tidy`，`Coverity`，`cppcheck`，`PVS-Studio`，`tscancode`。您将在 `dbms/tests/instructions/` 目录中找到使用说明。您也可以阅读[俄文文章](#)。

如果您使用 `CLion` 作为 IDE，您可以开箱即用一些 `clang-tidy` 检查。

## 其他强化

默认情况下使用 `FORTIFY_SOURCE`。它几乎没用，但在极少数情况下仍然有意义，我们不会禁用它。

## 代码风格

代码风格在[这里](#)有说明。

要检查一些常见的样式冲突，您可以使用 `utils/check-style` 脚本。

为了强制你的代码的正确风格，你可以使用 `clang-format` 文件。`.clang-format` 位于源代码根目录，它主要与我们的实际代码风格对应。但不建议将 `clang-format` 应用于现有文件，因为它会使格式变得更糟。您可以使用 `clang-format-diff` 工具，您可以在 `clang` 源代码库中找到

或者，您可以尝试 `uncrustify` 工具来格式化您的代码。配置文件在源代码的根目录中的 `uncrustify.cfg`。它比 `clang-format` 经过更少的测试。

`CLion` 有自己的代码格式化程序，必须调整为我们的代码风格。

## Metrica B2B 测试

每个 ClickHouse 版本都经过 Yandex Metrica 和 AppMetrica 引擎的测试。测试和稳定版本的 ClickHouse 部署在虚拟机上，并使用处理输入数据固定样本的度量引擎的小副本运行。将度量引擎的两个实例的结果一起进行比较

这些测试是由单独的团队自动完成的。由于移动部件的数量很多，大部分时间的测试都是完全无关的，很难弄清楚。很可能这些测试对我们来说是负值。然而，这些测试被证明是有用的大约一个或两个倍的数百。

## 测试覆盖率

截至2018年7月，我们不会跟踪测试复盖率。

## 自动化测试

我们使用 Yandex 内部 CI 和名为"沙箱"的作业自动化系统运行测试。我们还继续使用 Jenkins（可在Yandex内部使用）。

构建作业和测试在沙箱中按每次提交的基础上运行。结果包和测试结果发布在 GitHub 上，可以通过直接链接下载，结果会被永久存储。当您在 GitHub 上发送拉取请求时，我们将其标记为"可以测试"，我们的 CI 系统将为您构建 ClickHouse 包（发布，调试，地址消除等）。

由于时间和计算能力的限制，我们不使用 Travis CI。

在 Jenkins，我们运行字典测试，指标B2B测试。我们使用 Jenkins 来准备和发布版本。Jenkins是一种传统的技术，所有的工作将被转移到沙箱中。

# Third-Party Libraries Used

Library	License
base64	BSD 2-Clause License
boost	Boost Software License 1.0
brotli	MIT
capnproto	MIT
cctz	Apache License 2.0
double-conversion	BSD 3-Clause License
FastMemcpy	MIT
googletest	BSD 3-Clause License
hyperscan	BSD 3-Clause License
libbtrie	BSD 2-Clause License
libcxxabi	BSD + MIT
libdivide	Zlib License
libgsasl	LGPL v2.1
libhdfs3	Apache License 2.0
libmetrohash	Apache License 2.0
libpcg-random	Apache License 2.0
libressl	OpenSSL License
librdkafka	BSD 2-Clause License
libwchar_width	CC0 1.0 Universal
llvm	BSD 3-Clause License
lz4	BSD 2-Clause License
mariadb-connector-c	LGPL v2.1
murmurhash	Public Domain
pdqsort	Zlib License
poco	Boost Software License - Version 1.0

Library	License
re2	BSD 3-Clause License
UnixODBC	LGPL v2.1
zlib-ng	Zlib License
zstd	BSD 3-Clause License
## Roadmap	

## Q2 2019

- DDL for dictionaries
- Integration with S3-like object stores
- Multiple storages for hot/cold data, JBOD support

## Q3 2019

- JOIN execution improvements:
  - Distributed join not limited by memory
- Resource pools for more precise distribution of cluster capacity between users
- Fine-grained authorization
- Integration with external authentication services

## ClickHouse release 19.6.2.11, 2019-05-13

### New Features

- TTL expressions for columns and tables. [#4212](#) (Anton Popov)
- Added support for brotli compression for HTTP responses (Accept-Encoding: br) [#4388](#) (Mikhail)
- Added new function `isValidUTF8` for checking whether a set of bytes is correctly utf-8 encoded. [#4934](#) (Danila Kutenin)
- Add new load balancing policy `first_or_random` which sends queries to the first specified host and if it's inaccessible send queries to random hosts of shard. Useful for cross-replication topology setups. [#5012](#) (nvartolomei)

### Experimental Features

- Add setting `index_granularity_bytes` (adaptive index granularity) for MergeTree\* tables family. [#4826](#) (alesapin)

### Improvements

- Added support for non-constant and negative size and length arguments for function `substringUTF8`. [#4989](#) (alexey-milovidov)
- Disable push-down to right table in left join, left table in right join, and both tables in full join. This fixes wrong JOIN results in some cases. [#4846](#) (Ivan)
- `clickhouse-copier`: auto upload task configuration from `--task-file` option [#4876](#) (proller)
- Added typos handler for storage factory and table functions factory. [#4891](#) (Danila Kutenin)
- Support asterisks and qualified asterisks for multiple joins without subqueries [#4898](#) (Artem Zuikov)
- Make missing column error message more user friendly. [#4915](#) (Artem Zuikov)

### Performance Improvements

- Significant speedup of ASOF JOIN [#4924](#) (Martijn Bakker)

## Backward Incompatible Changes

- HTTP header `Query-Id` was renamed to `X-ClickHouse-Query-Id` for consistency. [#4972](#) (Mikhail)

## Bug Fixes

- Fixed potential null pointer dereference in `clickhouse-copier`. [#4900](#) (proller)
- Fixed error on query with JOIN + ARRAY JOIN [#4938](#) (Artem Zuikov)
- Fixed hanging on start of the server when a dictionary depends on another dictionary via a database with `engine=Dictionary`. [#4962](#) (Vitaly Baranov)
- Partially fix `distributed_product_mode = local`. It's possible to allow columns of local tables in `where/having/order by/...` via table aliases. Throw exception if table does not have alias. There's not possible to access to the columns without table aliases yet. [#4986](#) (Artem Zuikov)
- Fix potentially wrong result for `SELECT DISTINCT` with JOIN [#5001](#) (Artem Zuikov)

## Build/Testing/Packaging Improvements

- Fixed test failures when running `clickhouse-server` on different host [#4713](#) (Vasily Nemkov)
- `clickhouse-test`: Disable color control sequences in non tty environment. [#4937](#) (alesapin)
- `clickhouse-test`: Allow use any test database (remove `test.` qualification where it possible) [#5008](#) (proller)
- Fix `ubsan` errors [#5037](#) (Vitaly Baranov)
- Yandex `LFAalloc` was added to ClickHouse to allocate `MarkCache` and `UncompressedCache` data in different ways to catch segfaults more reliable [#4995](#) (Danila Kutenin)
- Python util to help with backports and changelogs. [#4949](#) (Ivan)

## ClickHouse release 19.5.4.22, 2019-05-13

### Bug fixes

- Fixed possible crash in `bitmap*` functions [#5220](#) [#5228](#) (Andy Yang)
- Fixed very rare data race condition that could happen when executing a query with UNION ALL involving at least two SELECTs from `system.columns`, `system.tables`, `system.parts`, `system.parts_tables` or tables of Merge family and performing ALTER of columns of the related tables concurrently. [#5189](#) (alexey-milovidov)
- Fixed error `Set for IN is not created yet` in case of using single `LowCardinality` column in the left part of IN This error happened if `LowCardinality` column was the part of primary key. [#5031](#) [#5154](#) (Nikolai Kochetov)
- Modification of retention function: If a row satisfies both the first and NTH condition, only the first satisfied condition is added to the data state. Now all conditions that satisfy in a row of data are added to the data state. [#5119](#) (小路)

## ClickHouse release 19.5.3.8, 2019-04-18

### Bug fixes

- Fixed type of setting `max_partitions_per_insert_block` from boolean to `UInt64`. [#5028](#) (Mohammad Hossein Sekhavat)

## ClickHouse release 19.5.2.6, 2019-04-15

### New Features

- **Hyperscan** multiple regular expression matching was added (functions `multiMatchAny`, `multiMatchAnyIndex`, `multiFuzzyMatchAny`, `multiFuzzyMatchAnyIndex`). [#4780](#), [#4841](#) (Danila Kutenin)
- `multiSearchFirstPosition` function was added. [#4780](#) (Danila Kutenin)
- Implement the predefined expression filter per row for tables. [#4792](#) (Ivan)
- A new type of data skipping indices based on bloom filters (can be used for `equal`, `in` and `like` functions). [#4499](#) (Nikita Vasilev)
- Added `ASOF JOIN` which allows to run queries that join to the most recent value known. [#4774](#) [#4867](#)



#4863 #4875 (Martijn Bakker, Artem Zuikov)

- Rewrite multiple `COMMA JOIN` to `CROSS JOIN`. Then rewrite them to `INNER JOIN` if possible. #4661 (Artem Zuikov)

## Improvement

- `topK` and `topKWeighted` now supports custom `loadFactor` (fixes issue #4252). #4634 (Kirill Danshin)
- Allow to use `parallel_replicas_count > 1` even for tables without sampling (the setting is simply ignored for them). In previous versions it was lead to exception. #4637 (Alexey Elymanov)
- Support for `CREATE OR REPLACE VIEW`. Allow to create a view or set a new definition in a single statement. #4654 (Boris Granveaud)
- `Buffer` table engine now supports `PREWHERE`. #4671 (Yangkuan Liu)
- Add ability to start replicated table without metadata in zookeeper in `readonly` mode. #4691 (alesapin)
- Fixed flicker of progress bar in clickhouse-client. The issue was most noticeable when using `FORMAT Null` with streaming queries. #4811 (alexey-milovidov)
- Allow to disable functions with `hyperscan` library on per user basis to limit potentially excessive and uncontrolled resource usage. #4816 (alexey-milovidov)
- Add version number logging in all errors. #4824 (proller)
- Added restriction to the `multiMatch` functions which requires string size to fit into `unsigned int`. Also added the number of arguments limit to the `multiSearch` functions. #4834 (Danila Kutenin)
- Improved usage of scratch space and error handling in Hyperscan. #4866 (Danila Kutenin)
- Fill `system.graphite_detentions` from a table config of `*GraphiteMergeTree` engine tables. #4584 (Mikhail f. Shiryayev)
- Rename `trigramDistance` function to `ngramDistance` and add more functions with `CaseInsensitive` and `UTF`. #4602 (Danila Kutenin)
- Improved data skipping indices calculation. #4640 (Nikita Vasilev)
- Keep ordinary, `DEFAULT`, `MATERIALIZED` and `ALIAS` columns in a single list (fixes issue #2867). #4707 (Alex Zatelepin)

## Bug Fix

- Avoid `std::terminate` in case of memory allocation failure. Now `std::bad_alloc` exception is thrown as expected. #4665 (alexey-milovidov)
- Fixes capnproto reading from buffer. Sometimes files wasn't loaded successfully by HTTP. #4674 (Vladislav)
- Fix error `Unknown log entry type: 0` after `OPTIMIZE TABLE FINAL` query. #4683 (Amos Bird)
- Wrong arguments to `hasAny` or `hasAll` functions may lead to segfault. #4698 (alexey-milovidov)
- Deadlock may happen while executing `DROP DATABASE dictionary` query. #4701 (alexey-milovidov)
- Fix undefined behavior in `median` and `quantile` functions. #4702 (hcz)
- Fix compression level detection when `network_compression_method` in lowercase. Broken in v19.1. #4706 (proller)
- Fixed ignorance of `<timezone>UTC</timezone>` setting (fixes issue #4658). #4718 (proller)
- Fix `histogram` function behaviour with `Distributed` tables. #4741 (olegkv)
- Fixed tsan report `destroy of a locked mutex`. #4742 (alexey-milovidov)
- Fixed TSan report on shutdown due to race condition in system logs usage. Fixed potential use-after-free on shutdown when `part_log` is enabled. #4758 (alexey-milovidov)
- Fix recheck parts in `ReplicatedMergeTreeAlterThread` in case of error. #4772 (Nikolai Kochetov)
- Arithmetic operations on intermediate aggregate function states were not working for constant arguments (such as subquery results). #4776 (alexey-milovidov)
- Always backquote column names in metadata. Otherwise it's impossible to create a table with column named `index` (server won't restart due to malformed `ATTACH` query in metadata). #4782 (alexey-milovidov)
- Fix crash in `ALTER ... MODIFY ORDER BY` on `Distributed` table. #4790 (TCeason)
- Fix segfault in `JOIN ON` with enabled `enable_optimize_predicate_expression`. #4794 (Winter Zhang)
- Fix bug with adding an extraneous row after consuming a protobuf message from Kafka. #4808 (Vitaly Baranov)
- Fix crash of `JOIN` on not nullable vs nullable column. Fix `NULL` is right hand in `ANY JOIN` + `NOT NULL`

- Fix crash on JOIN on not-nullable vs nullable column. Fix NULLs in right keys in ANY JOIN + join\_use\_nulls. [#4815](#) (Artem Zuikov)
- Fix segmentation fault in clickhouse-copier. [#4835](#) (proller)
- Fixed race condition in SELECT from system.tables if the table is renamed or altered concurrently. [#4836](#) (alexey-milovidov)
- Fixed data race when fetching data part that is already obsolete. [#4839](#) (alexey-milovidov)
- Fixed rare data race that can happen during RENAME table of MergeTree family. [#4844](#) (alexey-milovidov)
- Fixed segmentation fault in function arrayIntersect. Segmentation fault could happen if function was called with mixed constant and ordinary arguments. [#4847](#) (Lixiang Qian)
- Fixed reading from Array(LowCardinality) column in rare case when column contained a long sequence of empty arrays. [#4850](#) (Nikolai Kochetov)
- Fix crash in FULL/RIGHT JOIN when we joining on nullable vs not nullable. [#4855](#) (Artem Zuikov)
- Fix No message received exception while fetching parts between replicas. [#4856](#) (alesapin)
- Fixed arrayIntersect function wrong result in case of several repeated values in single array. [#4871](#) (Nikolai Kochetov)
- Fix a race condition during concurrent ALTER COLUMN queries that could lead to a server crash (fixes issue [#3421](#)). [#4592](#) (Alex Zatelepin)
- Fix incorrect result in FULL/RIGHT JOIN with const column. [#4723](#) (Artem Zuikov)
- Fix duplicates in GLOBAL JOIN with asterisk. [#4705](#) (Artem Zuikov)
- Fix parameter deduction in ALTER MODIFY of column CODEC when column type is not specified. [#4883](#) (alesapin)
- Functions cutQueryStringAndFragment() and queryStringAndFragment() now works correctly when URL contains a fragment and no query. [#4894](#) (Vitaly Baranov)
- Fix rare bug when setting min\_bytes\_to\_use\_direct\_io is greater than zero, which occurs when thread have to seek backward in column file. [#4897](#) (alesapin)
- Fix wrong argument types for aggregate functions with LowCardinality arguments (fixes issue [#4919](#)). [#4922](#) (Nikolai Kochetov)
- Fix wrong name qualification in GLOBAL JOIN. [#4969](#) (Artem Zuikov)
- Fix function toISOWeek result for year 1970. [#4988](#) (alexey-milovidov)
- Fix DROP, TRUNCATE and OPTIMIZE queries duplication, when executed on ON CLUSTER for ReplicatedMergeTree\* tables family. [#4991](#) (alesapin)

## Backward Incompatible Change

- Rename setting insert\_sample\_with\_metadata to setting input\_format\_defaults\_for\_omitted\_fields. [#4771](#) (Artem Zuikov)
- Added setting max\_partitions\_per\_insert\_block (with value 100 by default). If inserted block contains larger number of partitions, an exception is thrown. Set it to 0 if you want to remove the limit (not recommended). [#4845](#) (alexey-milovidov)
- Multi-search functions were renamed (multiPosition to multiSearchAllPositions, multiSearch to multiSearchAny, firstMatch to multiSearchFirstIndex). [#4780](#) (Danila Kutenin)

## Performance Improvement

- Optimize Volnitsky searcher by inlining, giving about 5-10% search improvement for queries with many needles or many similar bigrams. [#4862](#) (Danila Kutenin)
- Fix performance issue when setting use\_uncompressed\_cache is greater than zero, which appeared when all read data contained in cache. [#4913](#) (alesapin)

## Build/Testing/Packaging Improvement

- Hardening debug build: more granular memory mappings and ASLR; add memory protection for mark cache and index. This allows to find more memory stomping bugs in case when ASan and MSan cannot do it. [#4632](#) (alexey-milovidov)
- Add support for cmake variables ENABLE\_PROTOBUF, ENABLE\_PARQUET and ENABLE\_BROTLI which allows to enable/disable the above features (same as we can do for librdkafka, mysql, etc). [#4669](#) (Silviu Cernescu)



Caragea)

- Add ability to print process list and stacktraces of all threads if some queries are hung after test run. [#4675](#) (alesapin)
- Add retries on Connection loss error in clickhouse-test. [#4682](#) (alesapin)
- Add freebsd build with vagrant and build with thread sanitizer to packager script. [#4712](#) [#4748](#) (alesapin)
- Now user asked for password for user 'default' during installation. [#4725](#) (proller)
- Suppress warning in rdkafka library. [#4740](#) (alexey-milovidov)
- Allow ability to build without ssl. [#4750](#) (proller)
- Add a way to launch clickhouse-server image from a custom user. [#4753](#) (Mikhail f. Shiryayev)
- Upgrade contrib boost to 1.69. [#4793](#) (proller)
- Disable usage of mremap when compiled with Thread Sanitizer. Surprisingly enough, TSan does not intercept mremap (though it does intercept mmap, munmap) that leads to false positives. Fixed TSan report in stateful tests. [#4859](#) (alexey-milovidov)
- Add test checking using format schema via HTTP interface. [#4864](#) (Vitaly Baranov)

## ClickHouse release 19.4.4.33, 2019-04-17

### Bug Fixes

- Avoid std::terminate in case of memory allocation failure. Now std::bad\_alloc exception is thrown as expected. [#4665](#) (alexey-milovidov)
- Fixes capnproto reading from buffer. Sometimes files wasn't loaded successfully by HTTP. [#4674](#) (Vladislav)
- Fix error Unknown log entry type: 0 after OPTIMIZE TABLE FINAL query. [#4683](#) (Amos Bird)
- Wrong arguments to hasAny or hasAll functions may lead to segfault. [#4698](#) (alexey-milovidov)
- Deadlock may happen while executing DROP DATABASE dictionary query. [#4701](#) (alexey-milovidov)
- Fix undefined behavior in median and quantile functions. [#4702](#) (hcz)
- Fix compression level detection when network\_compression\_method in lowercase. Broken in v19.1. [#4706](#) (proller)
- Fixed ignorance of <timezone>UTC</timezone> setting (fixes issue [#4658](#)). [#4718](#) (proller)
- Fix histogram function behaviour with Distributed tables. [#4741](#) (olegkv)
- Fixed tsan report destroy of a locked mutex. [#4742](#) (alexey-milovidov)
- Fixed TSan report on shutdown due to race condition in system logs usage. Fixed potential use-after-free on shutdown when part\_log is enabled. [#4758](#) (alexey-milovidov)
- Fix recheck parts in ReplicatedMergeTreeAlterThread in case of error. [#4772](#) (Nikolai Kochetov)
- Arithmetic operations on intermediate aggregate function states were not working for constant arguments (such as subquery results). [#4776](#) (alexey-milovidov)
- Always backquote column names in metadata. Otherwise it's impossible to create a table with column named index (server won't restart due to malformed ATTACH query in metadata). [#4782](#) (alexey-milovidov)
- Fix crash in ALTER ... MODIFY ORDER BY on Distributed table. [#4790](#) (TCeason)
- Fix segfault in JOIN ON with enabled enable\_optimize\_predicate\_expression. [#4794](#) (Winter Zhang)
- Fix bug with adding an extraneous row after consuming a protobuf message from Kafka. [#4808](#) (Vitaly Baranov)
- Fix segmentation fault in clickhouse-copier. [#4835](#) (proller)
- Fixed race condition in SELECT from system.tables if the table is renamed or altered concurrently. [#4836](#) (alexey-milovidov)
- Fixed data race when fetching data part that is already obsolete. [#4839](#) (alexey-milovidov)
- Fixed rare data race that can happen during RENAME table of MergeTree family. [#4844](#) (alexey-milovidov)
- Fixed segmentation fault in function arrayIntersect. Segmentation fault could happen if function was called with mixed constant and ordinary arguments. [#4847](#) (Lixiang Qian)
- Fixed reading from Array(LowCardinality) column in rare case when column contained a long sequence of empty arrays. [#4850](#) (Nikolai Kochetov)
- Fix No message received exception while fetching parts between replicas. [#4856](#) (alesapin)

- Fixed `arrayIntersect` function wrong result in case of several repeated values in single array. [#4871](#) (Nikolai Kochetov)
- Fix a race condition during concurrent `ALTER COLUMN` queries that could lead to a server crash (fixes issue [#3421](#)). [#4592](#) (Alex Zatelepin)
- Fix parameter deduction in `ALTER MODIFY` of column `CODEC` when column type is not specified. [#4883](#) (alesapin)
- Functions `cutQueryStringAndFragment()` and `queryStringAndFragment()` now works correctly when `URL` contains a fragment and no query. [#4894](#) (Vitaly Baranov)
- Fix rare bug when setting `min_bytes_to_use_direct_io` is greater than zero, which occurs when threads have to seek backward in column file. [#4897](#) (alesapin)
- Fix wrong argument types for aggregate functions with `LowCardinality` arguments (fixes issue [#4919](#)). [#4922](#) (Nikolai Kochetov)
- Fix function `toISOWeek` result for year 1970. [#4988](#) (alexey-milovidov)
- Fix `DROP`, `TRUNCATE` and `OPTIMIZE` queries duplication, when executed on `ON CLUSTER` for `ReplicatedMergeTree*` tables family. [#4991](#) (alesapin)

## Improvements

- Keep ordinary, `DEFAULT`, `MATERIALIZED` and `ALIAS` columns in a single list (fixes issue [#2867](#)). [#4707](#) (Alex Zatelepin)

## ClickHouse release 19.4.3.11, 2019-04-02

### Bug Fixes

- Fix crash in `FULL/RIGHT JOIN` when we joining on nullable vs not nullable. [#4855](#) (Artem Zuikov)
- Fix segmentation fault in `clickhouse-copier`. [#4835](#) (proller)

### Build/Testing/Packaging Improvement

- Add a way to launch clickhouse-server image from a custom user. [#4753](#) (Mikhail f. Shiryayev)

## ClickHouse release 19.4.2.7, 2019-03-30

### Bug Fixes

- Fixed reading from `Array(LowCardinality)` column in rare case when column contained a long sequence of empty arrays. [#4850](#) (Nikolai Kochetov)

## ClickHouse release 19.4.1.3, 2019-03-19

### Bug Fixes

- Fixed remote queries which contain both `LIMIT BY` and `LIMIT`. Previously, if `LIMIT BY` and `LIMIT` were used for remote query, `LIMIT` could happen before `LIMIT BY`, which led to too filtered result. [#4708](#) (Constantin S. Pan)

## ClickHouse release 19.4.0.49, 2019-03-09

### New Features

- Added full support for `Protobuf` format (input and output, nested data structures). [#4174](#) [#4493](#) (Vitaly Baranov)
- Added bitmap functions with Roaring Bitmaps. [#4207](#) (Andy Yang) [#4568](#) (Vitaly Baranov)
- Parquet format support. [#4448](#) (proller)
- N-gram distance was added for fuzzy string comparison. It is similar to q-gram metrics in R language. [#4466](#) (Danila Kutenin)
- Combine rules for graphite rollup from dedicated aggregation and retention patterns. [#4426](#) (Mikhail f. Shiryayev)
- Added `max_execution_speed` and `max_execution_speed_bytes` to limit resource usage. Added `min_execution_speed_bytes` setting to complement the `min_execution_speed`. [#4430](#) (Winter Zhang)

- Implemented function `flatten`. #4555 #4409 (alexey-milovidov, kzon)
- Added functions `arrayEnumerateDenseRanked` and `arrayEnumerateUniqRanked` (it's like `arrayEnumerateUniq` but allows to fine tune array depth to look inside multidimensional arrays). #4475 (proller) #4601 (alexey-milovidov)
- Multiple JOINS with some restrictions: no asterisks, no complex aliases in ON/WHERE/GROUP BY/... #4462 (Artem Zuikov)

## Bug Fixes

- This release also contains all bug fixes from 19.3 and 19.1.
- Fixed bug in data skipping indices: order of granules after INSERT was incorrect. #4407 (Nikita Vasilev)
- Fixed `set index` for `Nullable` and `LowCardinality` columns. Before it, `set index` with `Nullable` or `LowCardinality` column led to error `Data type must be deserialized with multiple streams while selecting`. #4594 (Nikolai Kochetov)
- Correctly set `update_time` on full `executable` dictionary update. #4551 (Tema Novikov)
- Fix broken progress bar in 19.3. #4627 (filimonov)
- Fixed inconsistent values of `MemoryTracker` when memory region was shrinked, in certain cases. #4619 (alexey-milovidov)
- Fixed undefined behaviour in `ThreadPool`. #4612 (alexey-milovidov)
- Fixed a very rare crash with the message `mutex lock failed: Invalid argument` that could happen when a `MergeTree` table was dropped concurrently with a `SELECT`. #4608 (Alex Zatelepin)
- ODBC driver compatibility with `LowCardinality` data type. #4381 (proller)
- FreeBSD: Fixup for `AIOcontextPool`: Found `io_event` with unknown id 0 error. #4438 (urgordeadbeef)
- `system.part_log` table was created regardless to configuration. #4483 (alexey-milovidov)
- Fix undefined behaviour in `dictIsIn` function for cache dictionaries. #4515 (alesapin)
- Fixed a deadlock when a `SELECT` query locks the same table multiple times (e.g. from different threads or when executing multiple subqueries) and there is a concurrent DDL query. #4535 (Alex Zatelepin)
- Disable `compile_expressions` by default until we get own `llvm` contrib and can test it with `clang` and `asan`. #4579 (alesapin)
- Prevent `std::terminate` when `invalidate_query` for `clickhouse` external dictionary source has returned wrong resultset (empty or more than one row or more than one column). Fixed issue when the `invalidate_query` was performed every five seconds regardless to the lifetime. #4583 (alexey-milovidov)
- Avoid deadlock when the `invalidate_query` for a dictionary with `clickhouse` source was involving `system.dictionaries` table or `Dictionaries` database (rare case). #4599 (alexey-milovidov)
- Fixes for `CROSS JOIN` with empty `WHERE`. #4598 (Artem Zuikov)
- Fixed segfault in function "replicate" when constant argument is passed. #4603 (alexey-milovidov)
- Fix lambda function with predicate optimizer. #4408 (Winter Zhang)
- Multiple JOINS multiple fixes. #4595 (Artem Zuikov)

## Improvements

- Support aliases in `JOIN ON` section for right table columns. #4412 (Artem Zuikov)
- Result of multiple JOINS need correct result names to be used in subselects. Replace flat aliases with source names in result. #4474 (Artem Zuikov)
- Improve push-down logic for joined statements. #4387 (Ivan)

## Performance Improvements

- Improved heuristics of "move to PREWHERE" optimization. #4405 (alexey-milovidov)
- Use proper lookup tables that uses `HashTable`'s API for 8-bit and 16-bit keys. #4536 (Amos Bird)
- Improved performance of string comparison. #4564 (alexey-milovidov)
- Cleanup distributed DDL queue in a separate thread so that it doesn't slow down the main loop that processes distributed DDL tasks. #4502 (Alex Zatelepin)
- When `min_bytes_to_use_direct_io` is set to 1, not every file was opened with `O_DIRECT` mode because the data size to read was sometimes underestimated by the size of one compressed block. #4526 (alexey-milovidov)

## Build/Testing/Packaging Improvement

- Added support for clang-9 [#4604](#) (alexey-milovidov)
- Fix wrong `__asm__` instructions (again) [#4621](#) (Konstantin Podshumok)
- Add ability to specify settings for `clickhouse-performance-test` from command line. [#4437](#) (alesapin)
- Add dictionaries tests to integration tests. [#4477](#) (alesapin)
- Added queries from the benchmark on the website to automated performance tests. [#4496](#) (alexey-milovidov)
- `xxhash.h` does not exist in external lz4 because it is an implementation detail and its symbols are namespaced with `XXH_NAMESPACE` macro. When lz4 is external, xxHash has to be external too, and the dependents have to link to it. [#4495](#) (Orivej Desh)
- Fixed a case when `quantileTiming` aggregate function can be called with negative or floating point argument (this fixes fuzz test with undefined behaviour sanitizer). [#4506](#) (alexey-milovidov)
- Spelling error correction. [#4531](#) (sdk2)
- Fix compilation on Mac. [#4371](#) (Vitaly Baranov)
- Build fixes for FreeBSD and various unusual build configurations. [#4444](#) (proller)

## ClickHouse release 19.3.9.1, 2019-04-02

### Bug Fixes

- Fix crash in `FULL/RIGHT JOIN` when we joining on nullable vs not nullable. [#4855](#) (Artem Zuikov)
- Fix segmentation fault in `clickhouse-copier`. [#4835](#) (proller)
- Fixed reading from `Array(LowCardinality)` column in rare case when column contained a long sequence of empty arrays. [#4850](#) (Nikolai Kochetov)

## Build/Testing/Packaging Improvement

- Add a way to launch clickhouse-server image from a custom user [#4753](#) (Mikhail f. Shiryayev)

## ClickHouse release 19.3.7, 2019-03-12

### Bug fixes

- Fixed error in [#3920](#). This error manifestate itself as random cache corruption (messages `Unknown codec family code`, `Cannot seek through file`) and segfaults. This bug first appeared in version 19.1 and is present in versions up to 19.1.10 and 19.3.6. [#4623](#) (alexey-milovidov)

## ClickHouse release 19.3.6, 2019-03-02

### Bug fixes

- When there are more than 1000 threads in a thread pool, `std::terminate` may happen on thread exit. [Azat Khuzhin](#) [#4485](#) [#4505](#) (alexey-milovidov)
- Now it's possible to create `ReplicatedMergeTree*` tables with comments on columns without defaults and tables with columns codecs without comments and defaults. Also fix comparison of codecs. [#4523](#) (alesapin)
- Fixed crash on JOIN with array or tuple. [#4552](#) (Artem Zuikov)
- Fixed crash in clickhouse-copier with the message `ThreadStatus not created`. [#4540](#) (Artem Zuikov)
- Fixed hangup on server shutdown if distributed DDLs were used. [#4472](#) (Alex Zatelepin)
- Incorrect column numbers were printed in error message about text format parsing for columns with number greater than 10. [#4484](#) (alexey-milovidov)

## Build/Testing/Packaging Improvements

- Fixed build with AVX enabled. [#4527](#) (alexey-milovidov)
- Enable extended accounting and IO accounting based on good known version instead of kernel under which it is compiled. [#4541](#) (nvartolomei)
- Allow to skip setting of `core_dump.size_limit`, warning instead of throw if limit set fail. [#4473](#) (proller)
- Removed the `inline` tags of `void readBinary(...)` in `Field.cpp`. Also merged redundant `namespace DB` blocks.

#4530 (hcz)

## ClickHouse release 19.3.5, 2019-02-21

### Bug fixes

- Fixed bug with large http insert queries processing. #4454 (alesapin)
- Fixed backward incompatibility with old versions due to wrong implementation of `send_logs_level` setting. #4445 (alexey-milovidov)
- Fixed backward incompatibility of table function `remote` introduced with column comments. #4446 (alexey-milovidov)

## ClickHouse release 19.3.4, 2019-02-16

### Improvements

- Table index size is not accounted for memory limits when doing `ATTACH TABLE` query. Avoided the possibility that a table cannot be attached after being detached. #4396 (alexey-milovidov)
- Slightly raised up the limit on max string and array size received from ZooKeeper. It allows to continue to work with increased size of `CLIENT_JVMFLAGS=-Djute.maxbuffer=...` on ZooKeeper. #4398 (alexey-milovidov)
- Allow to repair abandoned replica even if it already has huge number of nodes in its queue. #4399 (alexey-milovidov)
- Add one required argument to `SET` index (max stored rows number). #4386 (Nikita Vasilev)

### Bug Fixes

- Fixed `WITH ROLLUP` result for group by single `LowCardinality` key. #4384 (Nikolai Kochetov)
- Fixed bug in the set index (dropping a granule if it contains more than `max_rows` rows). #4386 (Nikita Vasilev)
- A lot of FreeBSD build fixes. #4397 (proller)
- Fixed aliases substitution in queries with subquery containing same alias (issue #4110). #4351 (Artem Zuikov)

### Build/Testing/Packaging Improvements

- Add ability to run `clickhouse-server` for stateless tests in docker image. #4347 (Vasily Nemkov)

## ClickHouse release 19.3.3, 2019-02-13

### New Features

- Added the `KILL MUTATION` statement that allows removing mutations that are for some reasons stuck. Added `latest_failed_part`, `latest_fail_time`, `latest_fail_reason` fields to the `system.mutations` table for easier troubleshooting. #4287 (Alex Zatelepin)
- Added aggregate function `entropy` which computes Shannon entropy. #4238 (Quid37)
- Added ability to send queries `INSERT INTO tbl VALUES (...)` to server without splitting on `query` and `data` parts. #4301 (alesapin)
- Generic implementation of `arrayWithConstant` function was added. #4322 (alexey-milovidov)
- Implemented `NOT BETWEEN` comparison operator. #4228 (Dmitry Naumov)
- Implement `sumMapFiltered` in order to be able to limit the number of keys for which values will be summed by `sumMap`. #4129 (Léo Ercolanelli)
- Added support of `Nullable` types in `mysql` table function. #4198 (Emmanuel Donin de Rosière)
- Support for arbitrary constant expressions in `LIMIT` clause. #4246 (k3box)
- Added `topKWeighted` aggregate function that takes additional argument with (unsigned integer) weight. #4245 (Andrew Golman)
- `StorageJoin` now supports `join_any_take_last_row` setting that allows overwriting existing values of the same key. #3973 (Amos Bird)
- Added function `toStartOfInterval`. #4304 (Vitaly Baranov)
- Added `RowBinaryWithNamesAndTypes` format. #4200 (Oleg V. Kozlov)



- Added `RowBinaryWithNamesAndTypes` format. #4200 (Oleg V. Kozlyuk)
- Added IPv4 and IPv6 data types. More effective implementations of IPv\* functions. #3669 (Vasily Nemkov)
- Added function `toStartOfTenMinutes()`. #4298 (Vitaly Baranov)
- Added Protobuf output format. #4005 #4158 (Vitaly Baranov)
- Added brotli support for HTTP interface for data import (INSERTs). #4235 (Mikhail)
- Added hints while user make typo in function name or type in command line client. #4239 (Danila Kutenin)
- Added `Query-Id` to Server's HTTP Response header. #4231 (Mikhail)

## Experimental features

- Added `minmax` and `set` data skipping indices for MergeTree table engines family. #4143 (Nikita Vasilev)
- Added conversion of `CROSS JOIN` to `INNER JOIN` if possible. #4221 #4266 (Artem Zuikov)

## Bug Fixes

- Fixed `Not found column` for duplicate columns in `JOIN ON` section. #4279 (Artem Zuikov)
- Make `START REPLICATED SENDS` command start replicated sends. #4229 (nvartolomei)
- Fixed aggregate functions execution with `Array(LowCardinality)` arguments. #4055 (KochetovNicolai)
- Fixed wrong behaviour when doing `INSERT ... SELECT ... FROM file(...)` query and file has `CSVWithNames` or `TSVWithNames` format and the first data row is missing. #4297 (alexey-milovidov)
- Fixed crash on dictionary reload if dictionary not available. This bug was appeared in 19.1.6. #4188 (proller)
- Fixed `ALL JOIN` with duplicates in right table. #4184 (Artem Zuikov)
- Fixed segmentation fault with `use_uncompressed_cache=1` and exception with wrong uncompressed size. This bug was appeared in 19.1.6. #4186 (alesapin)
- Fixed `compile_expressions` bug with comparison of big (more than int16) dates. #4341 (alesapin)
- Fixed infinite loop when selecting from table function `numbers(0)`. #4280 (alexey-milovidov)
- Temporarily disable predicate optimization for `ORDER BY`. #3890 (Winter Zhang)
- Fixed `Illegal instruction` error when using base64 functions on old CPUs. This error has been reproduced only when ClickHouse was compiled with gcc-8. #4275 (alexey-milovidov)
- Fixed `No message received` error when interacting with PostgreSQL ODBC Driver through TLS connection. Also fixes segfault when using MySQL ODBC Driver. #4170 (alexey-milovidov)
- Fixed incorrect result when `Date` and `DateTime` arguments are used in branches of conditional operator (function `if`). Added generic case for function `if`. #4243 (alexey-milovidov)
- ClickHouse dictionaries now load within `clickhouse` process. #4166 (alexey-milovidov)
- Fixed deadlock when `SELECT` from a table with `File` engine was retried after `No such file or directory` error. #4161 (alexey-milovidov)
- Fixed race condition when selecting from `system.tables` may give `table doesn't exist` error. #4313 (alexey-milovidov)
- `clickhouse-client` can segfault on exit while loading data for command line suggestions if it was run in interactive mode. #4317 (alexey-milovidov)
- Fixed a bug when the execution of mutations containing `IN` operators was producing incorrect results. #4099 (Alex Zatelepin)
- Fixed error: if there is a database with `Dictionary` engine, all dictionaries forced to load at server startup, and if there is a dictionary with ClickHouse source from localhost, the dictionary cannot load. #4255 (alexey-milovidov)
- Fixed error when system logs are tried to create again at server shutdown. #4254 (alexey-milovidov)
- Correctly return the right type and properly handle locks in `joinGet` function. #4153 (Amos Bird)
- Added `sumMapWithOverflow` function. #4151 (Léo Ercolanelli)
- Fixed segfault with `allow_experimental_multiple_joins_emulation`. 52de2c (Artem Zuikov)
- Fixed bug with incorrect `Date` and `DateTime` comparison. #4237 (valexey)
- Fixed fuzz test under undefined behavior sanitizer: added parameter type check for `quantile*Weighted` family of functions. #4145 (alexey-milovidov)
- Fixed rare race condition when removing of old data parts can fail with `File not found` error. #4378 (alexey-milovidov)

- Fix install package with missing `/etc/clickhouse-server/config.xml`. [#4343](#) (proller)

## Build/Testing/Packaging Improvements

- Debian package: correct `/etc/clickhouse-server/preprocessed` link according to config. [#4205](#) (proller)
- Various build fixes for FreeBSD. [#4225](#) (proller)
- Added ability to create, fill and drop tables in `perfctest`. [#4220](#) (alesapin)
- Added a script to check for duplicate includes. [#4326](#) (alexey-milovidov)
- Added ability to run queries by index in performance test. [#4264](#) (alesapin)
- Package with debug symbols is suggested to be installed. [#4274](#) (alexey-milovidov)
- Refactoring of performance-test. Better logging and signals handling. [#4171](#) (alesapin)
- Added docs to anonymized Yandex.Metrika datasets. [#4164](#) (alesapin)
- Added tool for converting an old month-partitioned part to the custom-partitioned format. [#4195](#) (Alex Zatelepin)
- Added docs about two datasets in s3. [#4144](#) (alesapin)
- Added script which creates changelog from pull requests description. [#4169](#) [#4173](#) (KochetovNicolai) (KochetovNicolai)
- Added puppet module for Clickhouse. [#4182](#) (Maxim Fedotov)
- Added docs for a group of undocumented functions. [#4168](#) (Winter Zhang)
- ARM build fixes. [#4210](#) [#4306](#) [#4291](#) (proller) (proller)
- Dictionary tests now able to run from `ctest`. [#4189](#) (proller)
- Now `/etc/ssl` is used as default directory with SSL certificates. [#4167](#) (alexey-milovidov)
- Added checking SSE and AVX instruction at start. [#4234](#) (lgr)
- Init script will wait server until start. [#4281](#) (proller)

## Backward Incompatible Changes

- Removed `allow_experimental_low_cardinality_type` setting. `LowCardinality` data types are production ready. [#4323](#) (alexey-milovidov)
- Reduce mark cache size and uncompressed cache size accordingly to available memory amount. [#4240](#) (Lopatin Konstantin)
- Added keyword `INDEX` in `CREATE TABLE` query. A column with name `index` must be quoted with backticks or double quotes: ``index``. [#4143](#) (Nikita Vasilev)
- `sumMap` now promote result type instead of overflow. The old `sumMap` behavior can be obtained by using `sumMapWithOverflow` function. [#4151](#) (Léo Ercolanelli)

## Performance Improvements

- `std::sort` replaced by `pdqsort` for queries without `LIMIT`. [#4236](#) (Evgenii Pravda)
- Now server reuse threads from global thread pool. This affects performance in some corner cases. [#4150](#) (alexey-milovidov)

## Improvements

- Implemented AIO support for FreeBSD. [#4305](#) (urgordeadbeef)
- `SELECT * FROM a JOIN b USING a, b` now return `a` and `b` columns only from the left table. [#4141](#) (Artem Zuikov)
- Allow `-C` option of client to work as `-c` option. [#4232](#) (syominsergey)
- Now option `--password` used without value requires password from stdin. [#4230](#) (BSD\_Conqueror)
- Added highlighting of unescaped metacharacters in string literals that contain `LIKE` expressions or regexps. [#4327](#) (alexey-milovidov)
- Added cancelling of HTTP read only queries if client socket goes away. [#4213](#) (nvartolomei)
- Now server reports progress to keep client connections alive. [#4215](#) (Ivan)
- Slightly better message with reason for OPTIMIZE query with `optimize_throw_if_noop` setting enabled. [#4294](#) (alexey-milovidov)
- Added support of `--version` option for clickhouse server. [#4251](#) (Lopatin Konstantin)
- Added `--help/-h` option to `clickhouse-server`. [#4233](#) (Yuriy Baranov)
- Added support for scalar subqueries with aggregate function state result. [#4348](#) (Nikolai Kochetov)

- Added support for scalar subqueries with aggregate function state result. [#4370 \(nikolai-rozhkov\)](#)
- Improved server shutdown time and ALTERs waiting time. [#4372 \(alexey-milovidov\)](#)
- Added info about the replicated\_can\_become\_leader setting to system.replicas and add logging if the replica won't try to become leader. [#4379 \(Alex Zatelepin\)](#)

## ClickHouse release 19.1.14, 2019-03-14

- Fixed error Column ... queried more than once that may happen if the setting asterisk\_left\_columns\_only is set to 1 in case of using GLOBAL JOIN with SELECT \* (rare case). The issue does not exist in 19.3 and newer. [6bac7d8d \(Artem Zuikov\)](#)

## ClickHouse release 19.1.13, 2019-03-12

This release contains exactly the same set of patches as 19.3.7.

## ClickHouse release 19.1.10, 2019-03-03

This release contains exactly the same set of patches as 19.3.6.

## ClickHouse release 19.1.9, 2019-02-21

### Bug fixes

- Fixed backward incompatibility with old versions due to wrong implementation of send\_logs\_level setting. [#4445 \(alexey-milovidov\)](#)
- Fixed backward incompatibility of table function remote introduced with column comments. [#4446 \(alexey-milovidov\)](#)

## ClickHouse release 19.1.8, 2019-02-16

### Bug Fixes

- Fix install package with missing /etc/clickhouse-server/config.xml. [#4343 \(proller\)](#)

## ClickHouse release 19.1.7, 2019-02-15

### Bug Fixes

- Correctly return the right type and properly handle locks in joinGet function. [#4153 \(Amos Bird\)](#)
- Fixed error when system logs are tried to create again at server shutdown. [#4254 \(alexey-milovidov\)](#)
- Fixed error: if there is a database with Dictionary engine, all dictionaries forced to load at server startup, and if there is a dictionary with ClickHouse source from localhost, the dictionary cannot load. [#4255 \(alexey-milovidov\)](#)
- Fixed a bug when the execution of mutations containing IN operators was producing incorrect results. [#4099 \(Alex Zatelepin\)](#)
- clickhouse-client can segfault on exit while loading data for command line suggestions if it was run in interactive mode. [#4317 \(alexey-milovidov\)](#)
- Fixed race condition when selecting from system.tables may give table doesn't exist error. [#4313 \(alexey-milovidov\)](#)
- Fixed deadlock when SELECT from a table with File engine was retried after No such file or directory error. [#4161 \(alexey-milovidov\)](#)
- Fixed an issue: local ClickHouse dictionaries are loaded via TCP, but should load within process. [#4166 \(alexey-milovidov\)](#)
- Fixed No message received error when interacting with PostgreSQL ODBC Driver through TLS connection. Also fixes segfault when using MySQL ODBC Driver. [#4170 \(alexey-milovidov\)](#)
- Temporarily disable predicate optimization for ORDER BY. [#3890 \(Winter Zhang\)](#)
- Fixed infinite loop when selecting from table function numbers(0). [#4280 \(alexey-milovidov\)](#)
- Fixed compile\_expressions bug with comparison of big (more than int16) dates. [#4341 \(alesapin\)](#)
- Fixed segmentation fault with uncompressed\_cache=1 and exception with wrong uncompressed size. [#4186 \(alesapin\)](#)



- Fixed `ALL JOIN` with duplicates in right table. [#4184](#) (Artem Zuikov)
- Fixed wrong behaviour when doing `INSERT ... SELECT ... FROM file(...)` query and file has `CSVWithNames` or `TSVWithNames` format and the first data row is missing. [#4297](#) (alexey-milovidov)
- Fixed aggregate functions execution with `Array(LowCardinality)` arguments. [#4055](#) (KochetovNicolai)
- Debian package: correct `/etc/clickhouse-server/preprocessed` link according to config. [#4205](#) (proller)
- Fixed fuzz test under undefined behavior sanitizer: added parameter type check for `quantile*Weighted` family of functions. [#4145](#) (alexey-milovidov)
- Make `START REPLICATED SENDS` command start replicated sends. [#4229](#) (nvartolomei)
- Fixed `Not found column` for duplicate columns in `JOIN ON` section. [#4279](#) (Artem Zuikov)
- Now `/etc/ssl` is used as default directory with SSL certificates. [#4167](#) (alexey-milovidov)
- Fixed crash on dictionary reload if dictionary not available. [#4188](#) (proller)
- Fixed bug with incorrect `Date` and `DateTime` comparison. [#4237](#) (valexey)
- Fixed incorrect result when `Date` and `DateTime` arguments are used in branches of conditional operator (function `if`). Added generic case for function `if`. [#4243](#) (alexey-milovidov)

## ClickHouse release 19.1.6, 2019-01-24

### New Features

- Custom per column compression codecs for tables. [#3899](#) [#4111](#) (alesapin, Winter Zhang, Anatoly)
- Added compression codec `Delta`. [#4052](#) (alesapin)
- Allow to `ALTER` compression codecs. [#4054](#) (alesapin)
- Added functions `left`, `right`, `trim`, `ltrim`, `rtrim`, `timestampadd`, `timestampsub` for SQL standard compatibility. [#3826](#) (Ivan Blinkov)
- Support for write in HDFS tables and `hdfs` table function. [#4084](#) (alesapin)
- Added functions to search for multiple constant strings from big haystack: `multiPosition`, `multiSearch`, `firstMatch` also with `-UTF8`, `-CaseInsensitive`, and `-CaseInsensitiveUTF8` variants. [#4053](#) (Danila Kutenin)
- Pruning of unused shards if `SELECT` query filters by sharding key (setting `optimize_skip_unused_shards`). [#3851](#) (Gleb Kanterov, Ivan)
- Allow Kafka engine to ignore some number of parsing errors per block. [#4094](#) (Ivan)
- Added support for `CatBoost` multiclass models evaluation. Function `modelEvaluate` returns tuple with per-class raw predictions for multiclass models. `libcatboostmodel.so` should be built with [#607](#). [#3959](#) (KochetovNicolai)
- Added functions `filesystemAvailable`, `filesystemFree`, `filesystemCapacity`. [#4097](#) (Boris Granveaud)
- Added hashing functions `xxHash64` and `xxHash32`. [#3905](#) (filimonov)
- Added `gccMurmurHash` hashing function (GCC flavoured Murmur hash) which uses the same hash seed as `gcc` [#4000](#) (sundyli)
- Added hashing functions `javaHash`, `hiveHash`. [#3811](#) (shangshujie365)
- Added table function `remoteSecure`. Function works as `remote`, but uses secure connection. [#4088](#) (proller)

### Experimental features

- Added multiple JOINS emulation (`allow_experimental_multiple_joins_emulation` setting). [#3946](#) (Artem Zuikov)

### Bug Fixes

- Make `compiled_expression_cache_size` setting limited by default to lower memory consumption. [#4041](#) (alesapin)
- Fix a bug that led to hangups in threads that perform `ALTERs` of Replicated tables and in the thread that updates configuration from ZooKeeper. [#2947](#) [#3891](#) [#3934](#) (Alex Zatelepin)
- Fixed a race condition when executing a distributed `ALTER` task. The race condition led to more than one replica trying to execute the task and all replicas except one failing with a ZooKeeper error. [#3904](#) (Alex Zatelepin)
- Fix a bug when `from_zk` config elements weren't refreshed after a request to ZooKeeper timed out. [#2947](#) [#3947](#) (Alex Zatelepin)
- Fix bug with wrong prefix for IPv4 subnet masks [#3945](#) (alesapin)

- Fix bug with wrong prefix for IPv4 subnet masks. #3945 (alesapin)
- Fixed crash (std::terminate) in rare cases when a new thread cannot be created due to exhausted resources. #3956 (alexey-milovidov)
- Fix bug when in remote table function execution when wrong restrictions were used for in getStructureOfRemoteTable. #4009 (alesapin)
- Fix a leak of netlink sockets. They were placed in a pool where they were never deleted and new sockets were created at the start of a new thread when all current sockets were in use. #4017 (Alex Zatelepin)
- Fix bug with closing /proc/self/fd directory earlier than all fds were read from /proc after forking odbc-bridge subprocess. #4120 (alesapin)
- Fixed String to UInt monotonic conversion in case of usage String in primary key. #3870 (Winter Zhang)
- Fixed error in calculation of integer conversion function monotonicity. #3921 (alexey-milovidov)
- Fixed segfault in arrayEnumerateUniq, arrayEnumerateDense functions in case of some invalid arguments. #3909 (alexey-milovidov)
- Fix UB in StorageMerge. #3910 (Amos Bird)
- Fixed segfault in functions addDays, subtractDays. #3913 (alexey-milovidov)
- Fixed error: functions round, floor, trunc, ceil may return bogus result when executed on integer argument and large negative scale. #3914 (alexey-milovidov)
- Fixed a bug induced by 'kill query sync' which leads to a core dump. #3916 (muVulDeePecker)
- Fix bug with long delay after empty replication queue. #3928 #3932 (alesapin)
- Fixed excessive memory usage in case of inserting into table with LowCardinality primary key. #3955 (KochetovNicolai)
- Fixed LowCardinality serialization for Native format in case of empty arrays. #3907 #4011 (KochetovNicolai)
- Fixed incorrect result while using distinct by single LowCardinality numeric column. #3895 #4012 (KochetovNicolai)
- Fixed specialized aggregation with LowCardinality key (in case when compile setting is enabled). #3886 (KochetovNicolai)
- Fix user and password forwarding for replicated tables queries. #3957 (alesapin) (小路)
- Fixed very rare race condition that can happen when listing tables in Dictionary database while reloading dictionaries. #3970 (alexey-milovidov)
- Fixed incorrect result when HAVING was used with ROLLUP or CUBE. #3756 #3837 (Sam Chou)
- Fixed column aliases for query with JOIN ON syntax and distributed tables. #3980 (Winter Zhang)
- Fixed error in internal implementation of quantileTDigest (found by Artem Vakhrushev). This error never happens in ClickHouse and was relevant only for those who use ClickHouse codebase as a library directly. #3935 (alexey-milovidov)

## Improvements

- Support for IF NOT EXISTS in ALTER TABLE ADD COLUMN statements along with IF EXISTS in DROP/MODIFY/CLEAR/COMMENT COLUMN. #3900 (Boris Granveaud)
- Function parseDateTimeBestEffort: support for formats DD.MM.YYYY, DD.MM.YY, DD-MM-YYYY, DD-Mon-YYYY, DD/Month/YYYY and similar. #3922 (alexey-milovidov)
- CapnProtoInputStream now support jagged structures. #4063 (Odin Hultgren Van Der Horst)
- Usability improvement: added a check that server process is started from the data directory's owner. Do not allow to start server from root if the data belongs to non-root user. #3785 (sergey-v-galtsev)
- Better logic of checking required columns during analysis of queries with JOINS. #3930 (Artem Zuikov)
- Decreased the number of connections in case of large number of Distributed tables in a single server. #3726 (Winter Zhang)
- Supported totals row for WITH TOTALS query for ODBC driver. #3836 (Maksim Koritckiy)
- Allowed to use Enums as integers inside if function. #3875 (Ivan)
- Added low\_cardinality\_allow\_in\_native\_format setting. If disabled, do not use LowCardinality type in Native format. #3879 (KochetovNicolai)
- Removed some redundant objects from compiled expressions cache to lower memory usage. #4042 (alesapin)
- Add check that SET send\_logs\_level = 'value' query accept appropriate value. #3873 (Sabyanin Maxim)

- Fixed data type check in type conversion functions. [#3896](#) (Winter Zhang)

## Performance Improvements

- Add a MergeTree setting `use_minimalistic_part_header_in_zookeeper`. If enabled, Replicated tables will store compact part metadata in a single part znode. This can dramatically reduce ZooKeeper snapshot size (especially if the tables have a lot of columns). Note that after enabling this setting you will not be able to downgrade to a version that doesn't support it. [#3960](#) (Alex Zatelepin)
- Add an DFA-based implementation for functions `sequenceMatch` and `sequenceCount` in case pattern doesn't contain time. [#4004](#) (Léo Ercolanelli)
- Performance improvement for integer numbers serialization. [#3968](#) (Amos Bird)
- Zero left padding PODArray so that -1 element is always valid and zeroed. It's used for branchless calculation of offsets. [#3920](#) (Amos Bird)
- Reverted `jemalloc` version which lead to performance degradation. [#4018](#) (alexey-milovidov)

## Backward Incompatible Changes

- Removed undocumented feature `ALTER MODIFY PRIMARY KEY` because it was superseded by the `ALTER MODIFY ORDER BY` command. [#3887](#) (Alex Zatelepin)
- Removed function `shardByHash`. [#3833](#) (alexey-milovidov)
- Forbid using scalar subqueries with result of type `AggregateFunction`. [#3865](#) (Ivan)

## Build/Testing/Packaging Improvements

- Added support for PowerPC (ppc64le) build. [#4132](#) (Danila Kutenin)
- Stateful functional tests are run on public available dataset. [#3969](#) (alexey-milovidov)
- Fixed error when the server cannot start with the `bash: /usr/bin/clickhouse-extract-from-config: Operation not permitted` message within Docker or systemd-nspawn. [#4136](#) (alexey-milovidov)
- Updated `rdkafka` library to v1.0.0-RC5. Used `cppkafka` instead of raw C interface. [#4025](#) (Ivan)
- Updated `mariadb-client` library. Fixed one of issues found by UBSan. [#3924](#) (alexey-milovidov)
- Some fixes for UBSan builds. [#3926](#) [#3021](#) [#3948](#) (alexey-milovidov)
- Added per-commit runs of tests with UBSan build.
- Added per-commit runs of PVS-Studio static analyzer.
- Fixed bugs found by PVS-Studio. [#4013](#) (alexey-milovidov)
- Fixed glibc compatibility issues. [#4100](#) (alexey-milovidov)
- Move Docker images to 18.10 and add compatibility file for glibc `>= 2.28` [#3965](#) (alesapin)
- Add env variable if user don't want to chown directories in server Docker image. [#3967](#) (alesapin)
- Enabled most of the warnings from `-Weverything` in clang. Enabled `-Wpedantic`. [#3986](#) (alexey-milovidov)
- Added a few more warnings that are available only in clang 8. [#3993](#) (alexey-milovidov)
- Link to `libLLVM` rather than to individual LLVM libs when using shared linking. [#3989](#) (Orivej Desh)
- Added sanitizer variables for test images. [#4072](#) (alesapin)
- `clickhouse-server` debian package will recommend `libcap2-bin` package to use `setcap` tool for setting capabilities. This is optional. [#4093](#) (alexey-milovidov)
- Improved compilation time, fixed includes. [#3898](#) (proller)
- Added performance tests for hash functions. [#3918](#) (filimonov)
- Fixed cyclic library dependences. [#3958](#) (proller)
- Improved compilation with low available memory. [#4030](#) (proller)
- Added test script to reproduce performance degradation in `jemalloc`. [#4036](#) (alexey-milovidov)
- Fixed misspells in comments and string literals under `dbms`. [#4122](#) (maiha)
- Fixed typos in comments. [#4089](#) (Evgenii Pravda)

## ClickHouse release 18.16.1, 2018-12-21

### Bug fixes:

- Fixed an error that led to problems with updating dictionaries with the ODBC source. [#3825](#), [#3829](#)
- JIT compilation of aggregate functions now works with LowCardinality columns. [#3838](#)

## Improvements:

- Added the `low_cardinality_allow_in_native_format` setting (enabled by default). When disabled, LowCardinality columns will be converted to ordinary columns for SELECT queries and ordinary columns will be expected for INSERT queries. [#3879](#)

## Build improvements:

- Fixes for builds on macOS and ARM.

## ClickHouse release 18.16.0, 2018-12-14

### New features:

- DEFAULT expressions are evaluated for missing fields when loading data in semi-structured input formats (JSONEachRow, TSKV). The feature is enabled with the `insert_sample_with_metadata` setting. [#3555](#)
- The ALTER TABLE query now has the `MODIFY ORDER BY` action for changing the sorting key when adding or removing a table column. This is useful for tables in the MergeTree family that perform additional tasks when merging based on this sorting key, such as `SummingMergeTree`, `AggregatingMergeTree`, and so on. [#3581](#) [#3755](#)
- For tables in the MergeTree family, now you can specify a different sorting key (`ORDER BY`) and index (`PRIMARY KEY`). The sorting key can be longer than the index. [#3581](#)
- Added the `hdfs` table function and the `HDFS` table engine for importing and exporting data to HDFS. [chenxing-xc](#)
- Added functions for working with base64: `base64Encode`, `base64Decode`, `tryBase64Decode`. [Alexander Krasheninnikov](#)
- Now you can use a parameter to configure the precision of the `uniqCombined` aggregate function (select the number of HyperLogLog cells). [#3406](#)
- Added the `system.contributors` table that contains the names of everyone who made commits in ClickHouse. [#3452](#)
- Added the ability to omit the partition for the `ALTER TABLE ... FREEZE` query in order to back up all partitions at once. [#3514](#)
- Added `dictGet` and `dictGetOrDefault` functions that don't require specifying the type of return value. The type is determined automatically from the dictionary description. [Amos Bird](#)
- Now you can specify comments for a column in the table description and change it using `ALTER`. [#3377](#)
- Reading is supported for Join type tables with simple keys. [Amos Bird](#)
- Now you can specify the options `join_use_nulls`, `max_rows_in_join`, `max_bytes_in_join`, and `join_overflow_mode` when creating a Join type table. [Amos Bird](#)
- Added the `joinGet` function that allows you to use a Join type table like a dictionary. [Amos Bird](#)
- Added the `partition_key`, `sorting_key`, `primary_key`, and `sampling_key` columns to the `system.tables` table in order to provide information about table keys. [#3609](#)
- Added the `is_in_partition_key`, `is_in_sorting_key`, `is_in_primary_key`, and `is_in_sampling_key` columns to the `system.columns` table. [#3609](#)
- Added the `min_time` and `max_time` columns to the `system.parts` table. These columns are populated when the partitioning key is an expression consisting of `DateTime` columns. [Emmanuel Donin de Rosière](#)

### Bug fixes:

- Fixes and performance improvements for the `LowCardinality` data type. `GROUP BY` using `LowCardinality(Nullable(...))`. Getting the values of `extremes`. Processing high-order functions. `LEFT ARRAY JOIN`. Distributed `GROUP BY`. Functions that return `Array`. Execution of `ORDER BY`. Writing to Distributed tables (nicelulu). Backward compatibility for `INSERT` queries from old clients that implement the `Native` protocol. Support for `LowCardinality` for `JOIN`. Improved performance when working in a single stream. [#3823](#) [#3803](#) [#3799](#) [#3769](#) [#3744](#) [#3681](#) [#3651](#) [#3649](#) [#3641](#) [#3632](#) [#3568](#) [#3523](#) [#3518](#)
- Fixed how the `select_sequential_consistency` option works. Previously, when this setting was enabled, an incomplete result was sometimes returned after beginning to write to a new partition. [#2863](#)
- Databases are correctly specified when executing DDL `ON CLUSTER` queries and `ALTER UPDATE/DELETE`. [#3772](#) [#3460](#)

- Databases are correctly specified for subqueries inside a VIEW. [#3521](#)
- Fixed a bug in PREWHERE with FINAL for VersionedCollapsingMergeTree. [7167bfd7](#)
- Now you can use KILL QUERY to cancel queries that have not started yet because they are waiting for the table to be locked. [#3517](#)
- Corrected date and time calculations if the clocks were moved back at midnight (this happens in Iran, and happened in Moscow from 1981 to 1983). Previously, this led to the time being reset a day earlier than necessary, and also caused incorrect formatting of the date and time in text format. [#3819](#)
- Fixed bugs in some cases of VIEW and subqueries that omit the database. [Winter Zhang](#)
- Fixed a race condition when simultaneously reading from a MATERIALIZED VIEW and deleting a MATERIALIZED VIEW due to not locking the internal MATERIALIZED VIEW. [#3404](#) [#3694](#)
- Fixed the error Lock handler cannot be nullptr. [#3689](#)
- Fixed query processing when the compile\_expressions option is enabled (it's enabled by default). Nondeterministic constant expressions like the now function are no longer unfolded. [#3457](#)
- Fixed a crash when specifying a non-constant scale argument in toDecimal32/64/128 functions.
- Fixed an error when trying to insert an array with NULL elements in the Values format into a column of type Array without Nullable (if input\_format\_values\_interpret\_expressions = 1). [#3487](#) [#3503](#)
- Fixed continuous error logging in DDLWorker if ZooKeeper is not available. [8f50c620](#)
- Fixed the return type for quantile\* functions from Date and DateTime types of arguments. [#3580](#)
- Fixed the WITH clause if it specifies a simple alias without expressions. [#3570](#)
- Fixed processing of queries with named sub-queries and qualified column names when enable\_optimize\_predicate\_expression is enabled. [Winter Zhang](#)
- Fixed the error Attempt to attach to nullptr thread group when working with materialized views. [Marek Vavruša](#)
- Fixed a crash when passing certain incorrect arguments to the arrayReverse function. [73e3a7b6](#)
- Fixed the buffer overflow in the extractURLParameter function. Improved performance. Added correct processing of strings containing zero bytes. [141e9799](#)
- Fixed buffer overflow in the lowerUTF8 and upperUTF8 functions. Removed the ability to execute these functions over FixedString type arguments. [#3662](#)
- Fixed a rare race condition when deleting MergeTree tables. [#3680](#)
- Fixed a race condition when reading from Buffer tables and simultaneously performing ALTER or DROP on the target tables. [#3719](#)
- Fixed a segfault if the max\_temporary\_non\_const\_columns limit was exceeded. [#3788](#)

## Improvements:

- The server does not write the processed configuration files to the /etc/clickhouse-server/ directory. Instead, it saves them in the preprocessed\_configs directory inside path. This means that the /etc/clickhouse-server/ directory doesn't have write access for the clickhouse user, which improves security. [#2443](#)
- The min\_merge\_bytes\_to\_use\_direct\_io option is set to 10 GiB by default. A merge that forms large parts of tables from the MergeTree family will be performed in O\_DIRECT mode, which prevents excessive page cache eviction. [#3504](#)
- Accelerated server start when there is a very large number of tables. [#3398](#)
- Added a connection pool and HTTP Keep-Alive for connections between replicas. [#3594](#)
- If the query syntax is invalid, the 400 Bad Request code is returned in the HTTP interface (500 was returned previously). [31bc680a](#)
- The join\_default\_strictness option is set to ALL by default for compatibility. [120e2cbe](#)
- Removed logging to stderr from the re2 library for invalid or complex regular expressions. [#3723](#)
- Added for the Kafka table engine: checks for subscriptions before beginning to read from Kafka; the kafka\_max\_block\_size setting for the table. [Marek Vavruša](#)
- The cityHash64, farmHash64, metroHash64, sipHash64, halfMD5, murmurHash2\_32, murmurHash2\_64, murmurHash3\_32, and murmurHash3\_64 functions now work for any number of arguments and for arguments in the form of tuples. [#3451](#) [#3519](#)
- The arrayReverse function now works with any types of arrays. [73e3a7b6](#)
- Added an optional parameter: the slot size for the timeSlots function. [Kirill Shvakov](#)
- For FULL and RIGHT JOIN the max\_block\_size setting is used for a stream of non-joined data from the right



- For `ROLL` and `RIGHT JOIN`, the `max_block_size` setting is used for a stream of non-joined data from the right table. [Amos Bird](#)
- Added the `--secure` command line parameter in `clickhouse-benchmark` and `clickhouse-performance-test` to enable TLS. [#3688](#) [#3690](#)
- Type conversion when the structure of a `Buffer` type table does not match the structure of the destination table. [Vitaly Baranov](#)
- Added the `tcp_keep_alive_timeout` option to enable keep-alive packets after inactivity for the specified time interval. [#3441](#)
- Removed unnecessary quoting of values for the partition key in the `system.parts` table if it consists of a single column. [#3652](#)
- The modulo function works for `Date` and `DateTime` data types. [#3385](#)
- Added synonyms for the `POWER`, `LN`, `LCASE`, `UCASE`, `REPLACE`, `LOCATE`, `SUBSTR`, and `MID` functions. [#3774](#) [#3763](#) Some function names are case-insensitive for compatibility with the SQL standard. Added syntactic sugar `SUBSTRING(expr FROM start FOR length)` for compatibility with SQL. [#3804](#)
- Added the ability to `mlock` memory pages corresponding to `clickhouse-server` executable code to prevent it from being forced out of memory. This feature is disabled by default. [#3553](#)
- Improved performance when reading from `O_DIRECT` (with the `min_bytes_to_use_direct_io` option enabled). [#3405](#)
- Improved performance of the `dictGet...OrDefault` function for a constant key argument and a non-constant default argument. [Amos Bird](#)
- The `firstSignificantSubdomain` function now processes the domains `gov`, `mil`, and `edu`. [Igor Hatarist](#)
- Improved performance. [#3628](#)
- Ability to specify custom environment variables for starting `clickhouse-server` using the `SYS-V init.d` script by defining `CLICKHOUSE_PROGRAM_ENV` in `/etc/default/clickhouse`. [Pavlo Bashynskyi](#)
- Correct return code for the `clickhouse-server` init script. [#3516](#)
- The `system.metrics` table now has the `VersionInteger` metric, and `system.build_options` has the added line `VERSION_INTEGER`, which contains the numeric form of the ClickHouse version, such as `18016000`. [#3644](#)
- Removed the ability to compare the `Date` type with a number to avoid potential errors like `date = 2018-12-17`, where quotes around the date are omitted by mistake. [#3687](#)
- Fixed the behavior of stateful functions like `rowNumberInAllBlocks`. They previously output a result that was one number larger due to starting during query analysis. [Amos Bird](#)
- If the `force_restore_data` file can't be deleted, an error message is displayed. [Amos Bird](#)

## Build improvements:

- Updated the `jemalloc` library, which fixes a potential memory leak. [Amos Bird](#)
- Profiling with `jemalloc` is enabled by default in order to debug builds. [2cc82f5c](#)
- Added the ability to run integration tests when only `Docker` is installed on the system. [#3650](#)
- Added the fuzz expression test in `SELECT` queries. [#3442](#)
- Added a stress test for commits, which performs functional tests in parallel and in random order to detect more race conditions. [#3438](#)
- Improved the method for starting `clickhouse-server` in a `Docker` image. [Elghazal Ahmed](#)
- For a `Docker` image, added support for initializing databases using files in the `/docker-entrypoint-initdb.d` directory. [Konstantin Lebedev](#)
- Fixes for builds on ARM. [#3709](#)

## Backward incompatible changes:

- Removed the ability to compare the `Date` type with a number. Instead of `toDate('2018-12-18') = 17883`, you must use explicit type conversion `= toDate(17883)` [#3687](#)

## ClickHouse release 18.14.19, 2018-12-19

### Bug fixes:

- Fixed an error that led to problems with updating dictionaries with the ODBC source. [#3825](#), [#3829](#)
- Databases are correctly specified when executing DDL `ON CLUSTER` queries. [#3460](#)

- Fixed a segfault if the `max_temporary_non_const_columns` limit was exceeded. [#3788](#)

## Build improvements:

- Fixes for builds on ARM.

## ClickHouse release 18.14.18, 2018-12-04

### Bug fixes:

- Fixed error in `dictGet...` function for dictionaries of type `range`, if one of the arguments is constant and other is not. [#3751](#)
- Fixed error that caused messages `netlink: '...': attribute type 1 has an invalid length` to be printed in Linux kernel log, that was happening only on fresh enough versions of Linux kernel. [#3749](#)
- Fixed segfault in function `empty` for argument of `FixedString` type. [Daniel, Dao Quang Minh](#)
- Fixed excessive memory allocation when using large value of `max_query_size` setting (a memory chunk of `max_query_size` bytes was preallocated at once). [#3720](#)

### Build changes:

- Fixed build with LLVM/Clang libraries of version 7 from the OS packages (these libraries are used for runtime query compilation). [#3582](#)

## ClickHouse release 18.14.17, 2018-11-30

### Bug fixes:

- Fixed cases when the ODBC bridge process did not terminate with the main server process. [#3642](#)
- Fixed synchronous insertion into the `Distributed` table with a columns list that differs from the column list of the remote table. [#3673](#)
- Fixed a rare race condition that can lead to a crash when dropping a `MergeTree` table. [#3643](#)
- Fixed a query deadlock in case when query thread creation fails with the `Resource temporarily unavailable` error. [#3643](#)
- Fixed parsing of the `ENGINE` clause when the `CREATE AS` table syntax was used and the `ENGINE` clause was specified before the `AS table` (the error resulted in ignoring the specified engine). [#3692](#)

## ClickHouse release 18.14.15, 2018-11-21

### Bug fixes:

- The size of memory chunk was overestimated while deserializing the column of type `Array(String)` that leads to "Memory limit exceeded" errors. The issue appeared in version 18.12.13. [#3589](#)

## ClickHouse release 18.14.14, 2018-11-20

### Bug fixes:

- Fixed `ON CLUSTER` queries when cluster configured as secure (flag `<secure>`). [#3599](#)

### Build changes:

- Fixed problems (Illum-7 from system, macos) [#3582](#)

## ClickHouse release 18.14.13, 2018-11-08

### Bug fixes:

- Fixed the `Block structure mismatch in MergingSorted stream` error. [#3162](#)
- Fixed `ON CLUSTER` queries in case when secure connections were turned on in the cluster config (the `<secure>` flag). [#3465](#)
- Fixed an error in queries that used `SAMPLE`, `PREWHERE` and alias columns. [#3543](#)
- Fixed a rare `unknown compression method` error when the `min_bytes_to_use_direct_io` setting was enabled. [#3544](#)

## Performance improvements:

- Fixed performance regression of queries with `GROUP BY` of columns of `UInt16` or `Date` type when executing on AMD EPYC processors. [Igor Lapko](#)
- Fixed performance regression of queries that process long strings. [#3530](#)

## Build improvements:

- Improvements for simplifying the Arcadia build. [#3475](#), [#3535](#)

## ClickHouse release 18.14.12, 2018-11-02

### Bug fixes:

- Fixed a crash on joining two unnamed subqueries. [#3505](#)
- Fixed generating incorrect queries (with an empty `WHERE` clause) when querying external databases. [hotid](#)
- Fixed using an incorrect timeout value in ODBC dictionaries. [Marek Vavruša](#)

## ClickHouse release 18.14.11, 2018-10-29

### Bug fixes:

- Fixed the error `Block structure mismatch in UNION stream: different number of columns` in `LIMIT` queries. [#2156](#)
- Fixed errors when merging data in tables containing arrays inside Nested structures. [#3397](#)
- Fixed incorrect query results if the `merge_tree_uniform_read_distribution` setting is disabled (it is enabled by default). [#3429](#)
- Fixed an error on inserts to a Distributed table in Native format. [#3411](#)

## ClickHouse release 18.14.10, 2018-10-23

- The `compile_expressions` setting (JIT compilation of expressions) is disabled by default. [#3410](#)
- The `enable_optimize_predicate_expression` setting is disabled by default.

## ClickHouse release 18.14.9, 2018-10-16

### New features:

- The `WITH CUBE` modifier for `GROUP BY` (the alternative syntax `GROUP BY CUBE(...)` is also available). [#3172](#)
- Added the `formatDateTime` function. [Alexandr Krasheninnikov](#)
- Added the `JDBC` table engine and `jdbc` table function (requires installing `clickhouse-jdbc-bridge`). [Alexandr Krasheninnikov](#)
- Added functions for working with the ISO week number: `toISOWeek`, `toISOYear`, `toStartOfISOYear`, and `toDayOfYear`. [#3146](#)
- Now you can use `Nullable` columns for `MySQL` and `ODBC` tables. [#3362](#)
- Nested data structures can be read as nested objects in `JSONEachRow` format. Added the `input_format_import_nested_json` setting. [Veloman Yunkan](#)
- Parallel processing is available for many `MATERIALIZED VIEWS` when inserting data. See the `parallel_view_processing` setting. [Marek Vavruša](#)
- Added the `SYSTEM FLUSH LOGS` query (forced log flushes to system tables such as `query_log`). [#3321](#)
- Now you can use pre-defined `database` and `table` macros when declaring `Replicated` tables. [#3251](#)
- Added the ability to read `Decimal` type values in engineering notation (indicating powers of ten). [#3153](#)

### Experimental features:

- Optimization of the `GROUP BY` clause for `LowCardinality` data types. [#3138](#)
- Optimized calculation of expressions for `LowCardinality` data types. [#3200](#)

### Improvements:



## improvements:

- Significantly reduced memory consumption for queries with `ORDER BY` and `LIMIT`. See the `max_bytes_before_remerge_sort` setting. [#3205](#)
- In the absence of `JOIN` (`LEFT`, `INNER`, ...), `INNER JOIN` is assumed. [#3147](#)
- Qualified asterisks work correctly in queries with `JOIN`. [Winter Zhang](#)
- The `ODBC` table engine correctly chooses the method for quoting identifiers in the SQL dialect of a remote database. [Alexandr Krasheninnikov](#)
- The `compile_expressions` setting (JIT compilation of expressions) is enabled by default.
- Fixed behavior for simultaneous `DROP DATABASE/TABLE IF EXISTS` and `CREATE DATABASE/TABLE IF NOT EXISTS`. Previously, a `CREATE DATABASE ... IF NOT EXISTS` query could return the error message "File ... already exists", and the `CREATE TABLE ... IF NOT EXISTS` and `DROP TABLE IF EXISTS` queries could return `Table ... is creating or attaching right now`. [#3101](#)
- `LIKE` and `IN` expressions with a constant right half are passed to the remote server when querying from MySQL or ODBC tables. [#3182](#)
- Comparisons with constant expressions in a `WHERE` clause are passed to the remote server when querying from MySQL and ODBC tables. Previously, only comparisons with constants were passed. [#3182](#)
- Correct calculation of row width in the terminal for `Pretty` formats, including strings with hieroglyphs. [Amos Bird](#).
- `ON CLUSTER` can be specified for `ALTER UPDATE` queries.
- Improved performance for reading data in `JSONEachRow` format. [#3332](#)
- Added synonyms for the `LENGTH` and `CHARACTER_LENGTH` functions for compatibility. The `CONCAT` function is no longer case-sensitive. [#3306](#)
- Added the `TIMESTAMP` synonym for the `DateTime` type. [#3390](#)
- There is always space reserved for `query_id` in the server logs, even if the log line is not related to a query. This makes it easier to parse server text logs with third-party tools.
- Memory consumption by a query is logged when it exceeds the next level of an integer number of gigabytes. [#3205](#)
- Added compatibility mode for the case when the client library that uses the Native protocol sends fewer columns by mistake than the server expects for the `INSERT` query. This scenario was possible when using the `clickhouse-cpp` library. Previously, this scenario caused the server to crash. [#3171](#)
- In a user-defined `WHERE` expression in `clickhouse-copier`, you can now use a `partition_key` alias (for additional filtering by source table partition). This is useful if the partitioning scheme changes during copying, but only changes slightly. [#3166](#)
- The workflow of the `Kafka` engine has been moved to a background thread pool in order to automatically reduce the speed of data reading at high loads. [Marek Vavruša](#).
- Support for reading `Tuple` and `Nested` values of structures like `struct` in the `Cap'n'Proto` format. [Marek Vavruša](#)
- The list of top-level domains for the `firstSignificantSubdomain` function now includes the domain `biz`. [decaseal](#)
- In the configuration of external dictionaries, `null_value` is interpreted as the value of the default data type. [#3330](#)
- Support for the `intDiv` and `intDivOrZero` functions for `Decimal`. [b48402e8](#)
- Support for the `Date`, `DateTime`, `UUID`, and `Decimal` types as a key for the `sumMap` aggregate function. [#3281](#)
- Support for the `Decimal` data type in external dictionaries. [#3324](#)
- Support for the `Decimal` data type in `SummingMergeTree` tables. [#3348](#)
- Added specializations for `UUID` in `if`. [#3366](#)
- Reduced the number of `open` and `close` system calls when reading from a `MergeTree` table. [#3283](#)
- A `TRUNCATE TABLE` query can be executed on any replica (the query is passed to the leader replica). [Kirill Shvakov](#)

## Bug fixes:

- Fixed an issue with `Dictionary` tables for `range_hashed` dictionaries. This error occurred in version 18.12.17 [#1702](#)

- Fixed an error when loading `range_hashed` dictionaries (the message `Unsupported type Nullable (...)`). This error occurred in version 18.12.17. [#3362](#)
- Fixed errors in the `pointInPolygon` function due to the accumulation of inaccurate calculations for polygons with a large number of vertices located close to each other. [#3331](#) [#3341](#)
- If after merging data parts, the checksum for the resulting part differs from the result of the same merge in another replica, the result of the merge is deleted and the data part is downloaded from the other replica (this is the correct behavior). But after downloading the data part, it couldn't be added to the working set because of an error that the part already exists (because the data part was deleted with some delay after the merge). This led to cyclical attempts to download the same data. [#3194](#)
- Fixed incorrect calculation of total memory consumption by queries (because of incorrect calculation, the `max_memory_usage_for_all_queries` setting worked incorrectly and the `MemoryTracking` metric had an incorrect value). This error occurred in version 18.12.13. [Marek Vavruša](#)
- Fixed the functionality of `CREATE TABLE ... ON CLUSTER ... AS SELECT ...`. This error occurred in version 18.12.13. [#3247](#)
- Fixed unnecessary preparation of data structures for `JOIN`s on the server that initiates the query if the `JOIN` is only performed on remote servers. [#3340](#)
- Fixed bugs in the `Kafka` engine: deadlocks after exceptions when starting to read data, and locks upon completion [Marek Vavruša](#).
- For `Kafka` tables, the optional `schema` parameter was not passed (the schema of the `Cap'n'Proto` format). [Vojtech Splichal](#)
- If the ensemble of ZooKeeper servers has servers that accept the connection but then immediately close it instead of responding to the handshake, ClickHouse chooses to connect another server. Previously, this produced the error `Cannot read all data. Bytes read: 0. Bytes expected: 4.` and the server couldn't start. [8218cf3a](#)
- If the ensemble of ZooKeeper servers contains servers for which the DNS query returns an error, these servers are ignored. [17b8e209](#)
- Fixed type conversion between `Date` and `DateTime` when inserting data in the `VALUES` format (if `input_format_values_interpret_expressions = 1`). Previously, the conversion was performed between the numerical value of the number of days in Unix Epoch time and the Unix timestamp, which led to unexpected results. [#3229](#)
- Corrected type conversion between `Decimal` and integer numbers. [#3211](#)
- Fixed errors in the `enable_optimize_predicate_expression` setting. [Winter Zhang](#)
- Fixed a parsing error in CSV format with floating-point numbers if a non-default CSV separator is used, such as `;` [#3155](#)
- Fixed the `arrayCumSumNonNegative` function (it does not accumulate negative values if the accumulator is less than zero). [Aleksey Studnev](#)
- Fixed how Merge tables work on top of Distributed tables when using `PREWHERE`. [#3165](#)
- Bug fixes in the `ALTER UPDATE` query.
- Fixed bugs in the `odbc` table function that appeared in version 18.12. [#3197](#)
- Fixed the operation of aggregate functions with `StateArray` combinators. [#3188](#)
- Fixed a crash when dividing a `Decimal` value by zero. [69dd6609](#)
- Fixed output of types for operations using `Decimal` and integer arguments. [#3224](#)
- Fixed the segfault during `GROUP BY` on `Decimal128`. [3359ba06](#)
- The `log_query_threads` setting (logging information about each thread of query execution) now takes effect only if the `log_queries` option (logging information about queries) is set to 1. Since the `log_query_threads` option is enabled by default, information about threads was previously logged even if query logging was disabled. [#3241](#)
- Fixed an error in the distributed operation of the quantiles aggregate function (the error message `Not found column quantile...`). [292a8855](#)
- Fixed the compatibility problem when working on a cluster of version 18.12.17 servers and older servers at the same time. For distributed queries with `GROUP BY` keys of both fixed and non-fixed length, if there was a large amount of data to aggregate, the returned data was not always fully aggregated (two different rows contained the same aggregation keys). [#3254](#)
- Fixed handling of substitutions in `clickhouse-performance-test`, if the query contains only part of the

substitutions declared in the test. [#3263](#)

- Fixed an error when using `FINAL` with `PREWHERE`. [#3298](#)
- Fixed an error when using `PREWHERE` over columns that were added during `ALTER`. [#3298](#)
- Added a check for the absence of `arrayJoin` for `DEFAULT` and `MATERIALIZED` expressions. Previously, `arrayJoin` led to an error when inserting data. [#3337](#)
- Added a check for the absence of `arrayJoin` in a `PREWHERE` clause. Previously, this led to messages like `Size ... doesn't match` or `Unknown compression method` when executing queries. [#3357](#)
- Fixed segfault that could occur in rare cases after optimization that replaced `AND` chains from equality evaluations with the corresponding `IN` expression. [liuyimin-bytedance](#)
- Minor corrections to `clickhouse-benchmark`: previously, client information was not sent to the server; now the number of queries executed is calculated more accurately when shutting down and for limiting the number of iterations. [#3351](#) [#3352](#)

## Backward incompatible changes:

- Removed the `allow_experimental_decimal_type` option. The `Decimal` data type is available for default use. [#3329](#)

## ClickHouse release 18.12.17, 2018-09-16

### New features:

- `invalidate_query` (the ability to specify a query to check whether an external dictionary needs to be updated) is implemented for the `clickhouse` source. [#3126](#)
- Added the ability to use `UInt*`, `Int*`, and `DateTime` data types (along with the `Date` type) as a `range_hashed` external dictionary key that defines the boundaries of ranges. Now `NULL` can be used to designate an open range. [Vasily Nemkov](#)
- The `Decimal` type now supports `var*` and `stddev*` aggregate functions. [#3129](#)
- The `Decimal` type now supports mathematical functions (`exp`, `sin` and so on.) [#3129](#)
- The `system.part_log` table now has the `partition_id` column. [#3089](#)

### Bug fixes:

- `Merge` now works correctly on `Distributed` tables. [Winter Zhang](#)
- Fixed incompatibility (unnecessary dependency on the `glibc` version) that made it impossible to run ClickHouse on `Ubuntu Precise` and older versions. The incompatibility arose in version 18.12.13. [#3130](#)
- Fixed errors in the `enable_optimize_predicate_expression` setting. [Winter Zhang](#)
- Fixed a minor issue with backwards compatibility that appeared when working with a cluster of replicas on versions earlier than 18.12.13 and simultaneously creating a new replica of a table on a server with a newer version (shown in the message `Can not clone replica, because the ... updated to new ClickHouse version, which is logical, but shouldn't happen`). [#3122](#)

## Backward incompatible changes:

- The `enable_optimize_predicate_expression` option is enabled by default (which is rather optimistic). If query analysis errors occur that are related to searching for the column names, set `enable_optimize_predicate_expression` to 0. [Winter Zhang](#)

## ClickHouse release 18.12.14, 2018-09-13

### New features:

- Added support for `ALTER UPDATE` queries. [#3035](#)
- Added the `allow_ddl` option, which restricts the user's access to DDL queries. [#3104](#)
- Added the `min_merge_bytes_to_use_direct_io` option for `MergeTree` engines, which allows you to set a threshold for the total size of the merge (when above the threshold, data part files will be handled using `O_DIRECT`). [#3117](#)
- The `system.merges` system table now contains the `partition_id` column. [#3099](#)

### Improvements

## Improvements

- If a data part remains unchanged during mutation, it isn't downloaded by replicas. [#3103](#)
- Autocomplete is available for names of settings when working with `clickhouse-client`. [#3106](#)

## Bug fixes:

- Added a check for the sizes of arrays that are elements of `Nested` type fields when inserting. [#3118](#)
- Fixed an error updating external dictionaries with the `ODBC` source and `hashed` storage. This error occurred in version 18.12.13.
- Fixed a crash when creating a temporary table from a query with an `IN` condition. [Winter Zhang](#)
- Fixed an error in aggregate functions for arrays that can have `NULL` elements. [Winter Zhang](#)

## ClickHouse release 18.12.13, 2018-09-10

### New features:

- Added the `DECIMAL(digits, scale)` data type (`Decimal32(scale)`, `Decimal64(scale)`, `Decimal128(scale)`). To enable it, use the setting `allow_experimental_decimal_type`. [#2846](#) [#2970](#) [#3008](#) [#3047](#)
- New `WITH ROLLUP` modifier for `GROUP BY` (alternative syntax: `GROUP BY ROLLUP(...)`). [#2948](#)
- In queries with `JOIN`, the star character expands to a list of columns in all tables, in compliance with the SQL standard. You can restore the old behavior by setting `asterisk_left_columns_only` to 1 on the user configuration level. [Winter Zhang](#)
- Added support for `JOIN` with table functions. [Winter Zhang](#)
- Autocomplete by pressing Tab in `clickhouse-client`. [Sergey Shcherbin](#)
- Ctrl+C in `clickhouse-client` clears a query that was entered. [#2877](#)
- Added the `join_default_strictness` setting (values: `"`, `'any'`, `'all'`). This allows you to not specify `ANY` or `ALL` for `JOIN`. [#2982](#)
- Each line of the server log related to query processing shows the query ID. [#2482](#)
- Now you can get query execution logs in `clickhouse-client` (use the `send_logs_level` setting). With distributed query processing, logs are cascaded from all the servers. [#2482](#)
- The `system.query_log` and `system.processes` (`SHOW PROCESSLIST`) tables now have information about all changed settings when you run a query (the nested structure of the `Settings` data). Added the `log_query_settings` setting. [#2482](#)
- The `system.query_log` and `system.processes` tables now show information about the number of threads that are participating in query execution (see the `thread_numbers` column). [#2482](#)
- Added `ProfileEvents` counters that measure the time spent on reading and writing over the network and reading and writing to disk, the number of network errors, and the time spent waiting when network bandwidth is limited. [#2482](#)
- Added `ProfileEvents` counters that contain the system metrics from `rusage` (you can use them to get information about CPU usage in userspace and the kernel, page faults, and context switches), as well as `taskstats` metrics (use these to obtain information about I/O wait time, CPU wait time, and the amount of data read and recorded, both with and without page cache). [#2482](#)
- The `ProfileEvents` counters are applied globally and for each query, as well as for each query execution thread, which allows you to profile resource consumption by query in detail. [#2482](#)
- Added the `system.query_thread_log` table, which contains information about each query execution thread. Added the `log_query_threads` setting. [#2482](#)
- The `system.metrics` and `system.events` tables now have built-in documentation. [#3016](#)
- Added the `arrayEnumerateDense` function. [Amos Bird](#)
- Added the `arrayCumSumNonNegative` and `arrayDifference` functions. [Aleksey Studnev](#)
- Added the `retention` aggregate function. [Sundy Li](#)
- Now you can add (merge) states of aggregate functions by using the plus operator, and multiply the states of aggregate functions by a nonnegative constant. [#3062](#) [#3034](#)
- Tables in the `MergeTree` family now have the virtual column `_partition_id`. [#3089](#)

### Experimental features:

- Added the `LowCardinality(T)` data type. This data type automatically creates a local dictionary of values and allows data processing without unpacking the dictionary. [#2920](#)

and allows data processing without unpacking the dictionary. [#2030](#)

- Added a cache of JIT-compiled functions and a counter for the number of uses before compiling. To JIT compile expressions, enable the `compile_expressions` setting. [#2990](#) [#3077](#)

## Improvements:

- Fixed the problem with unlimited accumulation of the replication log when there are abandoned replicas. Added an effective recovery mode for replicas with a long lag.
- Improved performance of `GROUP BY` with multiple aggregation fields when one of them is string and the others are fixed length.
- Improved performance when using `PREWHERE` and with implicit transfer of expressions in `PREWHERE`.
- Improved parsing performance for text formats (CSV, TSV). [Amos Bird](#) [#2980](#)
- Improved performance of reading strings and arrays in binary formats. [Amos Bird](#)
- Increased performance and reduced memory consumption for queries to `system.tables` and `system.columns` when there is a very large number of tables on a single server. [#2953](#)
- Fixed a performance problem in the case of a large stream of queries that result in an error (the `_dl_addr` function is visible in `perf top`, but the server isn't using much CPU). [#2938](#)
- Conditions are cast into the View (when `enable_optimize_predicate_expression` is enabled). [Winter Zhang](#)
- Improvements to the functionality for the `UUID` data type. [#3074](#) [#2985](#)
- The `UUID` data type is supported in The-Alchemist dictionaries. [#2822](#)
- The `visitParamExtractRaw` function works correctly with nested structures. [Winter Zhang](#)
- When the `input_format_skip_unknown_fields` setting is enabled, object fields in `JSONEachRow` format are skipped correctly. [BlahGeek](#)
- For a `CASE` expression with conditions, you can now omit `ELSE`, which is equivalent to `ELSE NULL`. [#2920](#)
- The operation timeout can now be configured when working with ZooKeeper. [urykhy](#)
- You can specify an offset for `LIMIT n, m` as `LIMIT n OFFSET m`. [#2840](#)
- You can use the `SELECT TOP n` syntax as an alternative for `LIMIT`. [#2840](#)
- Increased the size of the queue to write to system tables, so the `SystemLog parameter queue is full` error doesn't happen as often.
- The `windowFunnel` aggregate function now supports events that meet multiple conditions. [Amos Bird](#)
- Duplicate columns can be used in a `USING` clause for `JOIN`. [#3006](#)
- `Pretty` formats now have a limit on column alignment by width. Use the `output_format_pretty_max_column_pad_width` setting. If a value is wider, it will still be displayed in its entirety, but the other cells in the table will not be too wide. [#3003](#)
- The `odbc` table function now allows you to specify the database/schema name. [Amos Bird](#)
- Added the ability to use a username specified in the `clickhouse-client` config file. [Vladimir Kozbin](#)
- The `ZooKeeperExceptions` counter has been split into three counters: `ZooKeeperUserExceptions`, `ZooKeeperHardwareExceptions`, and `ZooKeeperOtherExceptions`.
- `ALTER DELETE` queries work for materialized views.
- Added randomization when running the cleanup thread periodically for `ReplicatedMergeTree` tables in order to avoid periodic load spikes when there are a very large number of `ReplicatedMergeTree` tables.
- Support for `ATTACH TABLE ... ON CLUSTER` queries. [#3025](#)

## Bug fixes:

- Fixed an issue with `Dictionary` tables (throws the `Size of offsets doesn't match size of column or Unknown compression method` exception). This bug appeared in version 18.10.3. [#2913](#)
- Fixed a bug when merging `CollapsingMergeTree` tables if one of the data parts is empty (these parts are formed during merge or `ALTER DELETE` if all data was deleted), and the `vertical` algorithm was used for the merge. [#3049](#)
- Fixed a race condition during `DROP` or `TRUNCATE` for `Memory` tables with a simultaneous `SELECT`, which could lead to server crashes. This bug appeared in version 1.1.54388. [#3038](#)
- Fixed the possibility of data loss when inserting in `Replicated` tables if the `Session is expired` error is returned (data loss can be detected by the `ReplicatedDataLoss` metric). This error occurred in version 1.1.54378. [#2939](#) [#2949](#) [#2964](#)
- Fixed a segfault during `JOIN ... ON`. [#3000](#)
- Fixed the error searching column names when the `WHERE` expression consists entirely of a qualified



Fixed the error searching column names when the `WHERE` expression consists entirely of a qualified column name, such as `WHERE table.column`. #2994

- Fixed the "Not found column" error that occurred when executing distributed queries if a single column consisting of an IN expression with a subquery is requested from a remote server. #3087
- Fixed the Block structure mismatch in UNION stream: different number of columns error that occurred for distributed queries if one of the shards is local and the other is not, and optimization of the move to `PREWHERE` is triggered. #2226 #3037 #3055 #3065 #3073 #3090 #3093
- Fixed the `pointInPolygon` function for certain cases of non-convex polygons. #2910
- Fixed the incorrect result when comparing `nan` with integers. #3024
- Fixed an error in the `zlib-ng` library that could lead to segfault in rare cases. #2854
- Fixed a memory leak when inserting into a table with `AggregateFunction` columns, if the state of the aggregate function is not simple (allocates memory separately), and if a single insertion request results in multiple small blocks. #3084
- Fixed a race condition when creating and deleting the same `Buffer` or `MergeTree` table simultaneously.
- Fixed the possibility of a segfault when comparing tuples made up of certain non-trivial types, such as tuples. #2989
- Fixed the possibility of a segfault when running certain `ON CLUSTER` queries. Winter Zhang
- Fixed an error in the `arrayDistinct` function for `Nullable` array elements. #2845 #2937
- The `enable_optimize_predicate_expression` option now correctly supports cases with `SELECT *`. Winter Zhang
- Fixed the segfault when re-initializing the ZooKeeper session. #2917
- Fixed potential blocking when working with ZooKeeper.
- Fixed incorrect code for adding nested data structures in a `SummingMergeTree`.
- When allocating memory for states of aggregate functions, alignment is correctly taken into account, which makes it possible to use operations that require alignment when implementing states of aggregate functions. chenxing-xc

## Security fix:

- Safe use of ODBC data sources. Interaction with ODBC drivers uses a separate `clickhouse-odbc-bridge` process. Errors in third-party ODBC drivers no longer cause problems with server stability or vulnerabilities. #2828 #2879 #2886 #2893 #2921
- Fixed incorrect validation of the file path in the `catBoostPool` table function. #2894
- The contents of system tables (`tables`, `databases`, `parts`, `columns`, `parts_columns`, `merges`, `mutations`, `replicas`, and `replication_queue`) are filtered according to the user's configured access to databases (`allow_databases`). Winter Zhang

## Backward incompatible changes:

- In queries with JOIN, the star character expands to a list of columns in all tables, in compliance with the SQL standard. You can restore the old behavior by setting `asterisk_left_columns_only` to 1 on the user configuration level.

## Build changes:

- Most integration tests can now be run by commit.
- Code style checks can also be run by commit.
- The `memcpy` implementation is chosen correctly when building on CentOS7/Fedora. Etienne Champetier
- When using clang to build, some warnings from `-Weverything` have been added, in addition to the regular `-Wall-Wextra -Werror`. #2957
- Debugging the build uses the `jemalloc` debug option.
- The interface of the library for interacting with ZooKeeper is declared abstract. #2950

## ClickHouse release 18.10.3, 2018-08-13

### New features:

- HTTPS can be used for replication. #2760
- Added the functions `murmurHash2_64`, `murmurHash3_32`, `murmurHash3_64`, and `murmurHash3_128` in addition to the existing `murmurHash2_32`. #2791

- Support for Nullable types in the ClickHouse ODBC driver (ODBCDriver2 output format). [#2834](#)
- Support for `UUID` in the key columns.

## Improvements:

- Clusters can be removed without restarting the server when they are deleted from the config files. [#2777](#)
- External dictionaries can be removed without restarting the server when they are removed from config files. [#2779](#)
- Added `SETTINGS` support for the `Kafka` table engine. [Alexander Marshalov](#)
- Improvements for the `UUID` data type (not yet complete). [#2618](#)
- Support for empty parts after merges in the `SummingMergeTree`, `CollapsingMergeTree` and `VersionedCollapsingMergeTree` engines. [#2815](#)
- Old records of completed mutations are deleted (`ALTER DELETE`). [#2784](#)
- Added the `system.merge_tree_settings` table. [Kirill Shvakov](#)
- The `system.tables` table now has dependency columns: `dependencies_database` and `dependencies_table`. [Winter Zhang](#)
- Added the `max_partition_size_to_drop` config option. [#2782](#)
- Added the `output_format_json_escape_forward_slashes` option. [Alexander Bocharov](#)
- Added the `max_fetch_partition_retries_count` setting. [#2831](#)
- Added the `prefer_localhost_replica` setting for disabling the preference for a local replica and going to a local replica without inter-process interaction. [#2832](#)
- The `quantileExact` aggregate function returns `nan` in the case of aggregation on an empty `Float32` or `Float64` set. [Sundy Li](#)

## Bug fixes:

- Removed unnecessary escaping of the connection string parameters for ODBC, which made it impossible to establish a connection. This error occurred in version 18.6.0.
- Fixed the logic for processing `REPLACE PARTITION` commands in the replication queue. If there are two `REPLACE` commands for the same partition, the incorrect logic could cause one of them to remain in the replication queue and not be executed. [#2814](#)
- Fixed a merge bug when all data parts were empty (parts that were formed from a merge or from `ALTER DELETE` if all data was deleted). This bug appeared in version 18.1.0. [#2930](#)
- Fixed an error for concurrent `Set` or `Join`. [Amos Bird](#)
- Fixed the `Block structure mismatch in UNION stream: different number of columns` error that occurred for `UNION ALL` queries inside a sub-query if one of the `SELECT` queries contains duplicate column names. [Winter Zhang](#)
- Fixed a memory leak if an exception occurred when connecting to a MySQL server.
- Fixed incorrect clickhouse-client response code in case of a query error.
- Fixed incorrect behavior of materialized views containing `DISTINCT`. [#2795](#)

## Backward incompatible changes

- Removed support for `CHECK TABLE` queries for Distributed tables.

## Build changes:

- The allocator has been replaced: `jemalloc` is now used instead of `tcnalloc`. In some scenarios, this increases speed up to 20%. However, there are queries that have slowed by up to 20%. Memory consumption has been reduced by approximately 10% in some scenarios, with improved stability. With highly competitive loads, CPU usage in userspace and in system shows just a slight increase. [#2773](#)
- Use of `libressl` from a submodule. [#1983](#) [#2807](#)
- Use of `unixodbc` from a submodule. [#2789](#)
- Use of `mariadb-connector-c` from a submodule. [#2785](#)
- Added functional test files to the repository that depend on the availability of test data (for the time being, without the test data itself).

## ClickHouse release 18.6.0, 2018-08-02

### New features:

- Added support for ON expressions for the JOIN ON syntax:  
`JOIN ON Expr([table.]column ...) = Expr([table.]column, ...) [AND Expr([table.]column, ...) = Expr([table.]column, ...) ...]`  
The expression must be a chain of equalities joined by the AND operator. Each side of the equality can be an arbitrary expression over the columns of one of the tables. The use of fully qualified column names is supported (`table.name`, `database.table.name`, `table_alias.name`, `subquery_alias.name`) for the right table. [#2742](#)
- HTTPS can be enabled for replication. [#2760](#)

### Improvements:

- The server passes the patch component of its version to the client. Data about the patch version component is in `system.processes` and `query_log`. [#2646](#)

## ClickHouse release 18.5.1, 2018-07-31

### New features:

- Added the hash function `murmurHash2_32`. [#2756](#).

### Improvements:

- Now you can use the `from_env` [#2741](#) attribute to set values in config files from environment variables.
- Added case-insensitive versions of the `coalesce`, `ifNull`, and `nullIf` functions [#2752](#).

### Bug fixes:

- Fixed a possible bug when starting a replica [#2759](#).

## ClickHouse release 18.4.0, 2018-07-28

### New features:

- Added system tables: `formats`, `data_type_families`, `aggregate_function_combinators`, `table_functions`, `table_engines`, `collations` [#2721](#).
- Added the ability to use a table function instead of a table as an argument of a `remote` or `cluster` table function [#2708](#).
- Support for HTTP Basic authentication in the replication protocol [#2727](#).
- The `has` function now allows searching for a numeric value in an array of `Enum` values [Maxim Khrisanfov](#).
- Support for adding arbitrary message separators when reading from `Kafka` [Amos Bird](#).

### Improvements:

- The `ALTER TABLE t DELETE WHERE` query does not rewrite data parts that were not affected by the WHERE condition [#2694](#).
- The `use_minimalistic_checksums_in_zookeeper` option for `ReplicatedMergeTree` tables is enabled by default. This setting was added in version 1.1.54378, 2018-04-16. Versions that are older than 1.1.54378 can no longer be installed.
- Support for running `KILL` and `OPTIMIZE` queries that specify `ON CLUSTER` [Winter Zhang](#).

### Bug fixes:

- Fixed the error `Column ... is not under an aggregate function and not in GROUP BY` for aggregation with an IN expression. This bug appeared in version 18.1.0. ([bbdd780b](#))
- Fixed a bug in the `windowFunnel` aggregate function [Winter Zhang](#).
- Fixed a bug in the `anyHeavy` aggregate function ([a2101df2](#))
- Fixed server crash when using the `countArray()` aggregate function.



## Backward incompatible changes:

- Parameters for `Kafka` engine was changed from `Kafka(kafka_broker_list, kafka_topic_list, kafka_group_name, kafka_format[, kafka_schema, kafka_num_consumers])` to `Kafka(kafka_broker_list, kafka_topic_list, kafka_group_name, kafka_format[, kafka_row_delimiter, kafka_schema, kafka_num_consumers])`. If your tables use `kafka_schema` or `kafka_num_consumers` parameters, you have to manually edit the metadata files `path/metadata/database/table.sql` and add `kafka_row_delimiter` parameter with `" "` value.

## ClickHouse release 18.1.0, 2018-07-23

### New features:

- Support for the `ALTER TABLE t DELETE WHERE` query for non-replicated MergeTree tables ([#2634](#)).
- Support for arbitrary types for the `uniq*` family of aggregate functions ([#2010](#)).
- Support for arbitrary types in comparison operators ([#2026](#)).
- The `users.xml` file allows setting a subnet mask in the format `10.0.0.1/255.255.255.0`. This is necessary for using masks for IPv6 networks with zeros in the middle ([#2637](#)).
- Added the `arrayDistinct` function ([#2670](#)).
- The SummingMergeTree engine can now work with AggregateFunction type columns ([Constantin S. Pan](#)).

### Improvements:

- Changed the numbering scheme for release versions. Now the first part contains the year of release (A.D., Moscow timezone, minus 2000), the second part contains the number for major changes (increases for most releases), and the third part is the patch version. Releases are still backwards compatible, unless otherwise stated in the changelog.
- Faster conversions of floating-point numbers to a string ([Amos Bird](#)).
- If some rows were skipped during an insert due to parsing errors (this is possible with the `input_allow_errors_num` and `input_allow_errors_ratio` settings enabled), the number of skipped rows is now written to the server log ([Leonardo Cecchi](#)).

### Bug fixes:

- Fixed the TRUNCATE command for temporary tables ([Amos Bird](#)).
- Fixed a rare deadlock in the ZooKeeper client library that occurred when there was a network error while reading the response ([c315200](#)).
- Fixed an error during a CAST to Nullable types ([#1322](#)).
- Fixed the incorrect result of the `maxIntersection()` function when the boundaries of intervals coincided ([Michael Furmur](#)).
- Fixed incorrect transformation of the OR expression chain in a function argument ([chenxing-xc](#)).
- Fixed performance degradation for queries containing `IN (subquery)` expressions inside another subquery ([#2571](#)).
- Fixed incompatibility between servers with different versions in distributed queries that use a `CAST` function that isn't in uppercase letters ([fe8c4d6](#)).
- Added missing quoting of identifiers for queries to an external DBMS ([#2635](#)).

## Backward incompatible changes:

- Converting a string containing the number zero to DateTime does not work. Example: `SELECT toDateTime('0')`. This is also the reason that `DateTime DEFAULT '0'` does not work in tables, as well as `<null_value>0</null_value>` in dictionaries. Solution: replace `0` with `0000-00-00 00:00:00`.

## ClickHouse release 1.1.54394, 2018-07-12

### New features:

- Added the `histogram` aggregate function ([Mikhail Surin](#)).
- Now `OPTIMIZE TABLE ... FINAL` can be used without specifying partitions for `ReplicatedMergeTree` ([Amos Bird](#)).

## Bug fixes:

- Fixed a problem with a very small timeout for sockets (one second) for reading and writing when sending and downloading replicated data, which made it impossible to download larger parts if there is a load on the network or disk (it resulted in cyclical attempts to download parts). This error occurred in version 1.1.54388.
- Fixed issues when using chroot in ZooKeeper if you inserted duplicate data blocks in the table.
- The `has` function now works correctly for an array with Nullable elements (#2115).
- The `system.tables` table now works correctly when used in distributed queries. The `metadata_modification_time` and `engine_full` columns are now non-virtual. Fixed an error that occurred if only these columns were queried from the table.
- Fixed how an empty `TinyLog` table works after inserting an empty data block (#2563).
- The `system.zookeeper` table works if the value of the node in ZooKeeper is NULL.

## ClickHouse release 1.1.54390, 2018-07-06

### New features:

- Queries can be sent in `multipart/form-data` format (in the `query` field), which is useful if external data is also sent for query processing (Olga Hvostikova).
- Added the ability to enable or disable processing single or double quotes when reading data in CSV format. You can configure this in the `format_csv_allow_single_quotes` and `format_csv_allow_double_quotes` settings (Amos Bird).
- Now `OPTIMIZE TABLE ... FINAL` can be used without specifying the partition for non-replicated variants of `MergeTree` (Amos Bird).

### Improvements:

- Improved performance, reduced memory consumption, and correct memory consumption tracking with use of the `IN` operator when a table index could be used (#2584).
- Removed redundant checking of checksums when adding a data part. This is important when there are a large number of replicas, because in these cases the total number of checks was equal to  $N^2$ .
- Added support for `Array(Tuple(...))` arguments for the `arrayEnumerateUniq` function (#2573).
- Added `Nullable` support for the `runningDifference` function (#2594).
- Improved query analysis performance when there is a very large number of expressions (#2572).
- Faster selection of data parts for merging in `ReplicatedMergeTree` tables. Faster recovery of the ZooKeeper session (#2597).
- The `format_version.txt` file for `MergeTree` tables is re-created if it is missing, which makes sense if ClickHouse is launched after copying the directory structure without files (Ciprian Hacman).

## Bug fixes:

- Fixed a bug when working with ZooKeeper that could make it impossible to recover the session and readonly states of tables before restarting the server.
- Fixed a bug when working with ZooKeeper that could result in old nodes not being deleted if the session is interrupted.
- Fixed an error in the `quantileTDigest` function for Float arguments (this bug was introduced in version 1.1.54388) (Mikhail Surin).
- Fixed a bug in the index for `MergeTree` tables if the primary key column is located inside the function for converting types between signed and unsigned integers of the same size (#2603).
- Fixed segfault if `macros` are used but they aren't in the config file (#2570).
- Fixed switching to the default database when reconnecting the client (#2583).
- Fixed a bug that occurred when the `use_index_for_in_with_subqueries` setting was disabled.

## Security fix:

- Sending files is no longer possible when connected to MySQL (`LOAD DATA LOCAL INFILE`).

# ClickHouse release 1.1.54388, 2018-06-28

## New features:

- Support for the `ALTER TABLE t DELETE WHERE` query for replicated tables. Added the `system.mutations` table to track progress of this type of queries.
- Support for the `ALTER TABLE t [REPLACE|ATTACH] PARTITION` query for `*MergeTree` tables.
- Support for the `TRUNCATE TABLE` query ([Winter Zhang](#))
- Several new `SYSTEM` queries for replicated tables (`RESTART REPLICAS`, `SYNC REPLICA`, `[STOP|START] [MERGES|FETCHES|SENDS REPLICATED|REPLICATION QUEUES]`).
- Added the ability to write to a table with the MySQL engine and the corresponding table function ([sundy-li](#)).
- Added the `url()` table function and the `URL` table engine ([Alexander Sapin](#)).
- Added the `windowFunnel` aggregate function ([sundy-li](#)).
- New `startsWith` and `endsWith` functions for strings ([Vadim Plakhtinsky](#)).
- The `numbers()` table function now allows you to specify the offset ([Winter Zhang](#)).
- The password to `clickhouse-client` can be entered interactively.
- Server logs can now be sent to syslog ([Alexander Krasheninnikov](#)).
- Support for logging in dictionaries with a shared library source ([Alexander Sapin](#)).
- Support for custom CSV delimiters ([Ivan Zhukov](#))
- Added the `date_time_input_format` setting. If you switch this setting to `'best_effort'`, `DateTime` values will be read in a wide range of formats.
- Added the `clickhouse-obfuscator` utility for data obfuscation. Usage example: publishing data used in performance tests.

## Experimental features:

- Added the ability to calculate `and` arguments only where they are needed ([Anastasia Tsarkova](#))
- JIT compilation to native code is now available for some expressions ([pyos](#)).

## Bug fixes:

- Duplicates no longer appear for a query with `DISTINCT` and `ORDER BY`.
- Queries with `ARRAY JOIN` and `arrayFilter` no longer return an incorrect result.
- Fixed an error when reading an array column from a Nested structure ([#2066](#)).
- Fixed an error when analyzing queries with a `HAVING` clause like `HAVING tuple IN (...)`.
- Fixed an error when analyzing queries with recursive aliases.
- Fixed an error when reading from `ReplacingMergeTree` with a condition in `PREWHERE` that filters all rows ([#2525](#)).
- User profile settings were not applied when using sessions in the HTTP interface.
- Fixed how settings are applied from the command line parameters in `clickhouse-local`.
- The ZooKeeper client library now uses the session timeout received from the server.
- Fixed a bug in the ZooKeeper client library when the client waited for the server response longer than the timeout.
- Fixed pruning of parts for queries with conditions on partition key columns ([#2342](#)).
- Merges are now possible after `CLEAR COLUMN IN PARTITION` ([#2315](#)).
- Type mapping in the ODBC table function has been fixed ([sundy-li](#)).
- Type comparisons have been fixed for `DateTime` with and without the time zone ([Alexander Bocharov](#)).
- Fixed syntactic parsing and formatting of the `CAST` operator.
- Fixed insertion into a materialized view for the Distributed table engine ([Babacar Diassé](#)).
- Fixed a race condition when writing data from the `Kafka` engine to materialized views ([Yangkuan Liu](#)).
- Fixed SSRF in the `remote()` table function.
- Fixed exit behavior of `clickhouse-client` in multiline mode ([#2510](#)).

## Improvements:

- Background tasks in replicated tables are now performed in a thread pool instead of in separate threads ([Silviu Caragea](#)).

- Improved LZ4 compression performance.
- Faster analysis for queries with a large number of JOINS and sub-queries.
- The DNS cache is now updated automatically when there are too many network errors.
- Table inserts no longer occur if the insert into one of the materialized views is not possible because it has too many parts.
- Corrected the discrepancy in the event counters `Query`, `SelectQuery`, and `InsertQuery`.
- Expressions like `tuple IN (SELECT tuple)` are allowed if the tuple types match.
- A server with replicated tables can start even if you haven't configured ZooKeeper.
- When calculating the number of available CPU cores, limits on cgroups are now taken into account (Atri Sharma).
- Added chown for config directories in the systemd config file (Mikhail Shiryaev).

## 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).
- Cmake now generates files for ninja by default (like when using -G Ninja).
- Added the ability to use the libtinfo library instead of libtermcap (Georgy Kondratiev).
- Fixed a header file conflict in Fedora Rawhide (#2520).

## Backward incompatible changes:

- Removed escaping in `Vertical` and `Pretty*` formats and deleted the `VerticalRaw` format.
- If servers with version 1.1.54388 (or newer) and servers with an older version are used simultaneously in a distributed query and the query has the `cast(x, 'Type')` expression without the `AS` keyword and doesn't have the word `cast` in uppercase, an exception will be thrown with a message like `Not found column cast(0, 'UInt8')` in block. Solution: Update the server on the entire cluster.

## ClickHouse release 1.1.54385, 2018-06-01

### Bug fixes:

- Fixed an error that in some cases caused ZooKeeper operations to block.

## ClickHouse release 1.1.54383, 2018-05-22

### Bug fixes:

- Fixed a slowdown of replication queue if a table has many replicas.

## ClickHouse release 1.1.54381, 2018-05-14

### Bug fixes:

- Fixed a nodes leak in ZooKeeper when ClickHouse loses connection to ZooKeeper server.

## ClickHouse release 1.1.54380, 2018-04-21

### New features:

- Added the table function `file(path, format, structure)`. An example reading bytes from `/dev/urandom`: `In -s /dev/urandom /var/lib/clickhouse/user_files/random``clickhouse-client -q "SELECT * FROM file('random', 'RowBinary', 'd UInt8') LIMIT 10"`.

### Improvements:

- Subqueries can be wrapped in `()` brackets to enhance query readability. For example: `(SELECT 1) UNION ALL (SELECT 1)`.

- Simple `SELECT` queries from the `system.processes` table are not included in the `max_concurrent_queries` limit.

## Bug fixes:

- Fixed incorrect behavior of the `IN` operator when select from `MATERIALIZED VIEW`.
- Fixed incorrect filtering by partition index in expressions like `partition_key_column IN (...)`.
- Fixed inability to execute `OPTIMIZE` query on non-leader replica if `RENAME` was performed on the table.
- Fixed the authorization error when executing `OPTIMIZE` or `ALTER` queries on a non-leader replica.
- Fixed freezing of `KILL QUERY`.
- Fixed an error in ZooKeeper client library which led to loss of watches, freezing of distributed DDL queue, and slowdowns in the replication queue if a non-empty `chroot` prefix is used in the ZooKeeper configuration.

## Backward incompatible changes:

- Removed support for expressions like `(a, b) IN (SELECT (a, b))` (you can use the equivalent expression `(a, b) IN (SELECT a, b)`). In previous releases, these expressions led to undetermined `WHERE` filtering or caused errors.

# ClickHouse release 1.1.54378, 2018-04-16

## New features:

- Logging level can be changed without restarting the server.
- Added the `SHOW CREATE DATABASE` query.
- The `query_id` can be passed to `clickhouse-client` (elBroom).
- New setting: `max_network_bandwidth_for_all_users`.
- Added support for `ALTER TABLE ... PARTITION ...` for `MATERIALIZED VIEW`.
- Added information about the size of data parts in uncompressed form in the system table.
- Server-to-server encryption support for distributed tables (`<secure>1</secure>` in the replica config in `<remote_servers>`).
- Configuration of the table level for the `ReplicatedMergeTree` family in order to minimize the amount of data stored in Zookeeper: `use_minimalistic_checksums_in_zookeeper = 1`
- Configuration of the `clickhouse-client` prompt. By default, server names are now output to the prompt. The server's display name can be changed. It's also sent in the `X-ClickHouse-Display-Name` HTTP header (Kirill Shvakov).
- Multiple comma-separated `topics` can be specified for the `Kafka` engine (Tobias Adamson)
- When a query is stopped by `KILL QUERY` or `replace_running_query`, the client receives the `Query was cancelled` exception instead of an incomplete result.

## Improvements:

- `ALTER TABLE ... DROP/DETACH PARTITION` queries are run at the front of the replication queue.
- `SELECT ... FINAL` and `OPTIMIZE ... FINAL` can be used even when the table has a single data part.
- A `query_log` table is recreated on the fly if it was deleted manually (Kirill Shvakov).
- The `lengthUTF8` function runs faster (zhang2014).
- Improved performance of synchronous inserts in `Distributed` tables (`insert_distributed_sync = 1`) when there is a very large number of shards.
- The server accepts the `send_timeout` and `receive_timeout` settings from the client and applies them when connecting to the client (they are applied in reverse order: the server socket's `send_timeout` is set to the `receive_timeout` value received from the client, and vice versa).
- More robust crash recovery for asynchronous insertion into `Distributed` tables.
- The return type of the `countEqual` function changed from `UInt32` to `UInt64` (谢磊).

## Bug fixes:

- Fixed an error with `IN` when the left side of the expression is `Nullable`.
- Correct results are now returned when using tuples with `IN` when some of the tuple components are in

the table index.

- The `max_execution_time` limit now works correctly with distributed queries.
- Fixed errors when calculating the size of composite columns in the `system.columns` table.
- Fixed an error when creating a temporary table `CREATE TEMPORARY TABLE IF NOT EXISTS`.
- Fixed errors in `StorageKafka` ([##2075](#))
- Fixed server crashes from invalid arguments of certain aggregate functions.
- Fixed the error that prevented the `DETACH DATABASE` query from stopping background tasks for `ReplicatedMergeTree` tables.
- Too many parts state is less likely to happen when inserting into aggregated materialized views ([##2084](#)).
- Corrected recursive handling of substitutions in the config if a substitution must be followed by another substitution on the same level.
- Corrected the syntax in the metadata file when creating a `VIEW` that uses a query with `UNION ALL`.
- `SummingMergeTree` now works correctly for summation of nested data structures with a composite key.
- Fixed the possibility of a race condition when choosing the leader for `ReplicatedMergeTree` tables.

## Build changes:

- The build supports `ninja` instead of `make` and uses `ninja` by default for building releases.
- Renamed packages: `clickhouse-server-base` in `clickhouse-common-static`; `clickhouse-server-common` in `clickhouse-server`; `clickhouse-common-dbg` in `clickhouse-common-static-dbg`. To install, use `clickhouse-server` `clickhouse-client`. Packages with the old names will still load in the repositories for backward compatibility.

## Backward incompatible changes:

- Removed the special interpretation of an `IN` expression if an array is specified on the left side. Previously, the expression `arr IN (set)` was interpreted as "at least one `arr` element belongs to the `set`". To get the same behavior in the new version, write `arrayExists(x -> x IN (set), arr)`.
- Disabled the incorrect use of the socket option `SO_REUSEPORT`, which was incorrectly enabled by default in the Poco library. Note that on Linux there is no longer any reason to simultaneously specify the addresses `::` and `0.0.0.0` for listen – use just `::`, which allows listening to the connection both over IPv4 and IPv6 (with the default kernel config settings). You can also revert to the behavior from previous versions by specifying `<listen_reuse_port>1</listen_reuse_port>` in the config.

## ClickHouse release 1.1.54370, 2018-03-16

### New features:

- Added the `system.macros` table and auto updating of macros when the config file is changed.
- Added the `SYSTEM RELOAD CONFIG` query.
- Added the `maxIntersections(left_col, right_col)` aggregate function, which returns the maximum number of simultaneously intersecting intervals `[left; right]`. The `maxIntersectionsPosition(left, right)` function returns the beginning of the "maximum" interval. ([Michael Furmur](#)).

### Improvements:

- When inserting data in a `Replicated` table, fewer requests are made to `ZooKeeper` (and most of the user-level errors have disappeared from the `ZooKeeper` log).
- Added the ability to create aliases for data sets. Example: `WITH (1, 2, 3) AS set SELECT number IN set FROM system.numbers LIMIT 10`.

### Bug fixes:

- Fixed the `Illegal PREWHERE` error when reading from Merge tables for `Distributed` tables.
- Added fixes that allow you to start `clickhouse-server` in IPv4-only Docker containers.
- Fixed a race condition when reading from system `system.parts_columns` tables.
- Removed double buffering during a synchronous insert to a `Distributed` table, which could have caused the connection to timeout.
- Fixed a bug that caused excessively long waits for an unavailable replica before beginning a `SELECT`



query.

- Fixed incorrect dates in the `system.parts` table.
- Fixed a bug that made it impossible to insert data in a `Replicated` table if `chroot` was non-empty in the configuration of the `ZooKeeper` cluster.
- Fixed the vertical merging algorithm for an empty `ORDER BY` table.
- Restored the ability to use dictionaries in queries to remote tables, even if these dictionaries are not present on the requestor server. This functionality was lost in release 1.1.54362.
- Restored the behavior for queries like `SELECT * FROM remote('server2', default.table) WHERE col IN (SELECT col2 FROM default.table)` when the right side of the `IN` should use a remote `default.table` instead of a local one. This behavior was broken in version 1.1.54358.
- Removed extraneous error-level logging of `Not found column ... in block`.

## Clickhouse Release 1.1.54362, 2018-03-11

### New features:

- Aggregation without `GROUP BY` for an empty set (such as `SELECT count(*) FROM table WHERE 0`) now returns a result with one row with null values for aggregate functions, in compliance with the SQL standard. To restore the old behavior (return an empty result), set `empty_result_for_aggregation_by_empty_set` to 1.
- Added type conversion for `UNION ALL`. Different alias names are allowed in `SELECT` positions in `UNION ALL`, in compliance with the SQL standard.
- Arbitrary expressions are supported in `LIMIT BY` clauses. Previously, it was only possible to use columns resulting from `SELECT`.
- An index of MergeTree tables is used when `IN` is applied to a tuple of expressions from the columns of the primary key. Example: `WHERE (UserID, EventDate) IN ((123, '2000-01-01'), ...)` (Anastasiya Tsarkova).
- Added the `clickhouse-copier` tool for copying between clusters and resharding data (beta).
- Added consistent hashing functions: `yandexConsistentHash`, `jumpConsistentHash`, `sumburConsistentHash`. They can be used as a sharding key in order to reduce the amount of network traffic during subsequent reshardings.
- Added functions: `arrayAny`, `arrayAll`, `hasAny`, `hasAll`, `arrayIntersect`, `arrayResize`.
- Added the `arrayCumSum` function (Javi Santana).
- Added the `parseDateTimeBestEffort`, `parseDateTimeBestEffortOrZero`, and `parseDateTimeBestEffortOrNull` functions to read the `DateTime` from a string containing text in a wide variety of possible formats.
- Data can be partially reloaded from external dictionaries during updating (load just the records in which the value of the specified field greater than in the previous download) (Arsen Hakobyan).
- Added the `cluster` table function. Example: `cluster(cluster_name, db, table)`. The `remote` table function can accept the cluster name as the first argument, if it is specified as an identifier.
- The `remote` and `cluster` table functions can be used in `INSERT` queries.
- Added the `create_table_query` and `engine_full` virtual columns to the `system.tables` table. The `metadata_modification_time` column is virtual.
- Added the `data_path` and `metadata_path` columns to `system.tables` and `system.databases` tables, and added the `path` column to the `system.parts` and `system.parts_columns` tables.
- Added additional information about merges in the `system.part_log` table.
- An arbitrary partitioning key can be used for the `system.query_log` table (Kirill Shvakov).
- The `SHOW TABLES` query now also shows temporary tables. Added temporary tables and the `is_temporary` column to `system.tables` (zhang2014).
- Added `DROP TEMPORARY TABLE` and `EXISTS TEMPORARY TABLE` queries (zhang2014).
- Support for `SHOW CREATE TABLE` for temporary tables (zhang2014).
- Added the `system_profile` configuration parameter for the settings used by internal processes.
- Support for loading `object_id` as an attribute in `MongoDB` dictionaries (Pavel Litvinenko).
- Reading `null` as the default value when loading data for an external dictionary with the `MongoDB` source (Pavel Litvinenko).
- Reading `DateTime` values in the `Values` format from a Unix timestamp without single quotes.
- Failover is supported in `remote` table functions for cases when some of the replicas are missing the requested table.
- Configuration settings can be overridden in the command line when you run `clickhouse-server`. Example:

`clickhouse-server -- --logger.level=information.`

- Implemented the `empty` function from a `FixedString` argument: the function returns 1 if the string consists entirely of null bytes (zhang2014).
- Added the `listen_try` configuration parameter for listening to at least one of the listen addresses without quitting, if some of the addresses can't be listened to (useful for systems with disabled support for IPv4 or IPv6).
- Added the `VersionedCollapsingMergeTree` table engine.
- Support for rows and arbitrary numeric types for the `library` dictionary source.
- `MergeTree` tables can be used without a primary key (you need to specify `ORDER BY tuple()`).
- A `Nullable` type can be `CAST` to a non-`Nullable` type if the argument is not `NULL`.
- `RENAME TABLE` can be performed for `VIEW`.
- Added the `throwIf` function.
- Added the `odbc_default_field_size` option, which allows you to extend the maximum size of the value loaded from an ODBC source (by default, it is 1024).
- The `system.processes` table and `SHOW PROCESSLIST` now have the `is_cancelled` and `peak_memory_usage` columns.

## Improvements:

- Limits and quotas on the result are no longer applied to intermediate data for `INSERT SELECT` queries or for `SELECT` subqueries.
- Fewer false triggers of `force_restore_data` when checking the status of `Replicated` tables when the server starts.
- Added the `allow_distributed_ddl` option.
- Nondeterministic functions are not allowed in expressions for `MergeTree` table keys.
- Files with substitutions from `config.d` directories are loaded in alphabetical order.
- Improved performance of the `arrayElement` function in the case of a constant multidimensional array with an empty array as one of the elements. Example: `[[1], []][x]`.
- The server starts faster now when using configuration files with very large substitutions (for instance, very large lists of IP networks).
- When running a query, table valued functions run once. Previously, `remote` and `mysql` table valued functions performed the same query twice to retrieve the table structure from a remote server.
- The `MkDocs` documentation generator is used.
- When you try to delete a table column that `DEFAULT/MATERIALIZED` expressions of other columns depend on, an exception is thrown (zhang2014).
- Added the ability to parse an empty line in text formats as the number 0 for `Float` data types. This feature was previously available but was lost in release 1.1.54342.
- `Enum` values can be used in `min`, `max`, `sum` and some other functions. In these cases, it uses the corresponding numeric values. This feature was previously available but was lost in the release 1.1.54337.
- Added `max_expanded_ast_elements` to restrict the size of the AST after recursively expanding aliases.

## Bug fixes:

- Fixed cases when unnecessary columns were removed from subqueries in error, or not removed from subqueries containing `UNION ALL`.
- Fixed a bug in merges for `ReplacingMergeTree` tables.
- Fixed synchronous insertions in `Distributed` tables (`insert_distributed_sync = 1`).
- Fixed segfault for certain uses of `FULL` and `RIGHT JOIN` with duplicate columns in subqueries.
- Fixed segfault for certain uses of `replace_running_query` and `KILL QUERY`.
- Fixed the order of the `source` and `last_exception` columns in the `system.dictionaries` table.
- Fixed a bug when the `DROP DATABASE` query did not delete the file with metadata.
- Fixed the `DROP DATABASE` query for Dictionary databases.
- Fixed the low precision of `uniqHLL12` and `uniqCombined` functions for cardinalities greater than 100 million items (Alex Bocharov).
- Fixed the calculation of implicit default values when necessary to simultaneously calculate default



explicit expressions in `INSERT` queries (zhang2014).

- Fixed a rare case when a query to a `MergeTree` table couldn't finish (chenxing-xc).
- Fixed a crash that occurred when running a `CHECK` query for `Distributed` tables if all shards are local (chenxing.xc).
- Fixed a slight performance regression with functions that use regular expressions.
- Fixed a performance regression when creating multidimensional arrays from complex expressions.
- Fixed a bug that could cause an extra `FORMAT` section to appear in an `.sql` file with metadata.
- Fixed a bug that caused the `max_table_size_to_drop` limit to apply when trying to delete a `MATERIALIZED VIEW` looking at an explicitly specified table.
- Fixed incompatibility with old clients (old clients were sometimes sent data with the `DateTime('timezone')` type, which they do not understand).
- Fixed a bug when reading `Nested` column elements of structures that were added using `ALTER` but that are empty for the old partitions, when the conditions for these columns moved to `PREWHERE`.
- Fixed a bug when filtering tables by virtual `_table` columns in queries to `Merge` tables.
- Fixed a bug when using `ALIAS` columns in `Distributed` tables.
- Fixed a bug that made dynamic compilation impossible for queries with aggregate functions from the `quantile` family.
- Fixed a race condition in the query execution pipeline that occurred in very rare cases when using `Merge` tables with a large number of tables, and when using `GLOBAL` subqueries.
- Fixed a crash when passing arrays of different sizes to an `arrayReduce` function when using aggregate functions from multiple arguments.
- Prohibited the use of queries with `UNION ALL` in a `MATERIALIZED VIEW`.
- Fixed an error during initialization of the `part_log` system table when the server starts (by default, `part_log` is disabled).

## Backward incompatible changes:

- Removed the `distributed_ddl_allow_replicated_alter` option. This behavior is enabled by default.
- Removed the `strict_insert_defaults` setting. If you were using this functionality, write to `clickhouse-feedback@yandex-team.com`.
- Removed the `UnsortedMergeTree` engine.

## Clickhouse Release 1.1.54343, 2018-02-05

- Added macros support for defining cluster names in distributed DDL queries and constructors of `Distributed` tables: `CREATE TABLE distr ON CLUSTER '{cluster}' (...) ENGINE = Distributed('{cluster}', 'db', 'table')`.
- Now queries like `SELECT ... FROM table WHERE expr IN (subquery)` are processed using the `table` index.
- Improved processing of duplicates when inserting to `Replicated` tables, so they no longer slow down execution of the replication queue.

## Clickhouse Release 1.1.54342, 2018-01-22

This release contains bug fixes for the previous release 1.1.54337:

- Fixed a regression in 1.1.54337: if the default user has readonly access, then the server refuses to start up with the message `Cannot create database in readonly mode`.
- Fixed a regression in 1.1.54337: on systems with `systemd`, logs are always written to `syslog` regardless of the configuration; the `watchdog` script still uses `init.d`.
- Fixed a regression in 1.1.54337: wrong default configuration in the Docker image.
- Fixed nondeterministic behavior of `GraphiteMergeTree` (you can see it in log messages `Data after merge is not byte-identical to the data on another replicas`).
- Fixed a bug that may lead to inconsistent merges after `OPTIMIZE` query to `Replicated` tables (you may see it in log messages `Part ... intersects the previous part`).
- Buffer tables now work correctly when `MATERIALIZED` columns are present in the destination table (by zhang2014).
- Fixed a bug in implementation of `NULL`.

# Clickhouse Release 1.1.54337, 2018-01-18

## New features:

- Added support for storage of multi-dimensional arrays and tuples (Tuple data type) in tables.
- Support for table functions for `DESCRIBE` and `INSERT` queries. Added support for subqueries in `DESCRIBE`. Examples: `DESC TABLE remote('host', default.hits);` `DESC TABLE (SELECT 1);` `INSERT INTO TABLE FUNCTION remote('host', default.hits)`. Support for `INSERT INTO TABLE` in addition to `INSERT INTO`.
- Improved support for time zones. The `DateTime` data type can be annotated with the timezone that is used for parsing and formatting in text formats. Example: `DateTime('Europe/Moscow')`. When timezones are specified in functions for `DateTime` arguments, the return type will track the timezone, and the value will be displayed as expected.
- Added the functions `toTimeZone`, `timeDiff`, `toQuarter`, `toRelativeQuarterNum`. The `toRelativeHour/Minute/Second` functions can take a value of type `Date` as an argument. The `now` function name is case-sensitive.
- Added the `toStartOfFifteenMinutes` function (Kirill Shvakov).
- Added the `clickhouse format` tool for formatting queries.
- Added the `format_schema_path` configuration parameter (Marek Vavruša). It is used for specifying a schema in `Cap'n Proto` format. Schema files can be located only in the specified directory.
- Added support for config substitutions (`incl` and `conf.d`) for configuration of external dictionaries and models (Pavel Yakunin).
- Added a column with documentation for the `system.settings` table (Kirill Shvakov).
- Added the `system.parts_columns` table with information about column sizes in each data part of `MergeTree` tables.
- Added the `system.models` table with information about loaded `CatBoost` machine learning models.
- Added the `mysql` and `odbc` table function and corresponding `MySQL` and `ODBC` table engines for accessing remote databases. This functionality is in the beta stage.
- Added the possibility to pass an argument of type `AggregateFunction` for the `groupArray` aggregate function (so you can create an array of states of some aggregate function).
- Removed restrictions on various combinations of aggregate function combinators. For example, you can use `avgForEachIf` as well as `avgIfForEach` aggregate functions, which have different behaviors.
- The `-ForEach` aggregate function combinator is extended for the case of aggregate functions of multiple arguments.
- Added support for aggregate functions of `Nullable` arguments even for cases when the function returns a non-`Nullable` result (added with the contribution of Silviu Caragea). Example: `groupArray`, `groupUniqArray`, `topK`.
- Added the `max_client_network_bandwidth` for `clickhouse-client` (Kirill Shvakov).
- Users with the `readonly = 2` setting are allowed to work with `TEMPORARY` tables (`CREATE`, `DROP`, `INSERT...`) (Kirill Shvakov).
- Added support for using multiple consumers with the `Kafka` engine. Extended configuration options for `Kafka` (Marek Vavruša).
- Added the `intExp3` and `intExp4` functions.
- Added the `sumKahan` aggregate function.
- Added the `to * Number Or Null functions, where * Number` is a numeric type.
- Added support for `WITH` clauses for an `INSERT SELECT` query (author: zhang2014).
- Added settings: `http_connection_timeout`, `http_send_timeout`, `http_receive_timeout`. In particular, these settings are used for downloading data parts for replication. Changing these settings allows for faster failover if the network is overloaded.
- Added support for `ALTER` for tables of type `Null` (Anastasiya Tsarkova).
- The `reinterpretAsString` function is extended for all data types that are stored contiguously in memory.
- Added the `--silent` option for the `clickhouse-local` tool. It suppresses printing query execution info in `stderr`.
- Added support for reading values of type `Date` from text in a format where the month and/or day of the month is specified using a single digit instead of two digits (Amos Bird).

## Performance optimizations:

- Improved performance of aggregate functions `min`, `max`, `sum`, `count`, `countDistinct`, `countIf`, `countFrom`

- 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](mailto:clickhouse-feedback@yandex-team.com).
- In previous server versions there was an undocumented feature: if an aggregate function depends on parameters, you can still specify it without parameters in the AggregateFunction data type. Example: AggregateFunction(quantiles, UInt64) instead of AggregateFunction(quantiles(0.5, 0.9), UInt64). This feature was lost. Although it was undocumented, we plan to support it again in future releases.
- Enum data types cannot be used in min/max aggregate functions. This ability will be returned in the next release.

## Please note when upgrading:

- When doing a rolling update on a cluster, at the point when some of the replicas are running the old version of ClickHouse and some are running the new version, replication is temporarily stopped and the message `unknown parameter 'shard'` appears in the log. Replication will continue after all replicas of the cluster are updated.
- If different versions of ClickHouse are running on the cluster servers, it is possible that distributed queries using the following functions will have incorrect results: `varSamp`, `varPop`, `stddevSamp`, `stddevPop`, `covarSamp`, `covarPop`, `corr`. You should update all cluster nodes.

## ClickHouse release 1.1.54327, 2017-12-21

This release contains bug fixes for the previous release 1.1.54318:

- Fixed bug with possible race condition in replication that could lead to data loss. This issue affects versions 1.1.54310 and 1.1.54318. If you use one of these versions with Replicated tables, the update is strongly recommended. This issue shows in logs in Warning messages like `Part ... from own log doesn't exist`. The issue is relevant even if you don't see these messages in logs.

## ClickHouse release 1.1.54318, 2017-11-30

This release contains bug fixes for the previous release 1.1.54310:

- Fixed incorrect row deletions during merges in the SummingMergeTree engine
- Fixed a memory leak in unreplicated MergeTree engines
- Fixed performance degradation with frequent inserts in MergeTree engines
- Fixed an issue that was causing the replication queue to stop running
- Fixed rotation and archiving of server logs

## ClickHouse release 1.1.54310, 2017-11-01

### New features:

- Custom partitioning key for the MergeTree family of table engines.
- `Kafka` table engine.
- Added support for loading `CatBoost` models and applying them to data stored in ClickHouse.
- Added support for time zones with non-integer offsets from UTC.
- Added support for arithmetic operations with time intervals.
- The range of values for the Date and DateTime types is extended to the year 2105.
- Added the `CREATE MATERIALIZED VIEW x TO y` query (specifies an existing table for storing the data of a materialized view).
- Added the `ATTACH TABLE` query without arguments.
- The processing logic for Nested columns with names ending in `-Map` in a SummingMergeTree table was extracted to the `sumMap` aggregate function. You can now specify such columns explicitly.
- Max size of the IP trie dictionary is increased to 128M entries.
- Added the `getsizeofenumtype` function.
- Added the `sumWithOverflow` aggregate function.
- Added support for the Cap'n Proto input format.
- You can now customize compression level when using the `zstd` algorithm.

### Backward incompatible changes:

- Creation of temporary tables with an engine other than Memory is not allowed.
- Explicit creation of tables with the View or MaterializedView engine is not allowed.
- During table creation, a new check verifies that the sampling key expression is included in the primary key.

### Bug fixes:

- Fixed hangups when synchronously inserting into a Distributed table.
- Fixed nonatomic adding and removing of parts in Replicated tables.



- Data inserted into a materialized view is not subjected to unnecessary deduplication.
- Executing a query to a Distributed table for which the local replica is lagging and remote replicas are unavailable does not result in an error anymore.
- Users don't need access permissions to the `default` database to create temporary tables anymore.
- Fixed crashing when specifying the Array type without arguments.
- Fixed hangups when the disk volume containing server logs is full.
- Fixed an overflow in the `toRelativeWeekNum` function for the first week of the Unix epoch.

## Build improvements:

- Several third-party libraries (notably Poco) were updated and converted to git submodules.

## ClickHouse release 1.1.54304, 2017-10-19

### New features:

- TLS support in the native protocol (to enable, set `tcp_ssl_port` in `config.xml` ).

### Bug fixes:

- `ALTER` for replicated tables now tries to start running as soon as possible.
- Fixed crashing when reading data with the setting `preferred_block_size_bytes=0`.
- Fixed crashes of `clickhouse-client` when pressing Page Down
- Correct interpretation of certain complex queries with `GLOBAL IN` and `UNION ALL`
- `FREEZE PARTITION` always works atomically now.
- Empty POST requests now return a response with code 411.
- Fixed interpretation errors for expressions like `CAST(1 AS Nullable(UInt8))`.
- Fixed an error when reading `Array(Nullable(String))` columns from `MergeTree` tables.
- Fixed crashing when parsing queries like `SELECT dummy AS dummy, dummy AS b`
- Users are updated correctly with invalid `users.xml`
- Correct handling when an executable dictionary returns a non-zero response code.

## ClickHouse release 1.1.54292, 2017-09-20

### New features:

- Added the `pointInPolygon` function for working with coordinates on a coordinate plane.
- Added the `sumMap` aggregate function for calculating the sum of arrays, similar to `SummingMergeTree`.
- Added the `trunc` function. Improved performance of the rounding functions (`round`, `floor`, `ceil`, `roundToExp2`) and corrected the logic of how they work. Changed the logic of the `roundToExp2` function for fractions and negative numbers.
- The ClickHouse executable file is now less dependent on the libc version. The same ClickHouse executable file can run on a wide variety of Linux systems. There is still a dependency when using compiled queries (with the setting `compile = 1` , which is not used by default).
- Reduced the time needed for dynamic compilation of queries.

### Bug fixes:

- Fixed an error that sometimes produced `part ... intersects previous part` messages and weakened replica consistency.
- Fixed an error that caused the server to lock up if ZooKeeper was unavailable during shutdown.
- Removed excessive logging when restoring replicas.
- Fixed an error in the `UNION ALL` implementation.
- Fixed an error in the `concat` function that occurred if the first column in a block has the Array type.
- Progress is now displayed correctly in the `system.merges` table.

## ClickHouse release 1.1.54289, 2017-09-13

### New features:

---

- `SYSTEM` queries for server administration: `SYSTEM RELOAD DICTIONARY`, `SYSTEM RELOAD DICTIONARIES`, `SYSTEM DROP DNS CACHE`, `SYSTEM SHUTDOWN`, `SYSTEM KILL`.
- Added functions for working with arrays: `concat`, `arraySlice`, `arrayPushBack`, `arrayPushFront`, `arrayPopBack`, `arrayPopFront`.
- Added `root` and `identity` parameters for the ZooKeeper configuration. This allows you to isolate individual users on the same ZooKeeper cluster.
- Added aggregate functions `groupBitAnd`, `groupBitOr`, and `groupBitXor` (for compatibility, they are also available under the names `BIT_AND`, `BIT_OR`, and `BIT_XOR`).
- External dictionaries can be loaded from MySQL by specifying a socket in the filesystem.
- External dictionaries can be loaded from MySQL over SSL (`ssl_cert`, `ssl_key`, `ssl_ca` parameters).
- Added the `max_network_bandwidth_for_user` setting to restrict the overall bandwidth use for queries per user.
- Support for `DROP TABLE` for temporary tables.
- Support for reading `DateTime` values in Unix timestamp format from the `CSV` and `JSONEachRow` formats.
- Lagging replicas in distributed queries are now excluded by default (the default threshold is 5 minutes).
- FIFO locking is used during `ALTER`: an `ALTER` query isn't blocked indefinitely for continuously running queries.
- Option to set `umask` in the config file.
- Improved performance for queries with `DISTINCT`.

## Bug fixes:

- Improved the process for deleting old nodes in ZooKeeper. Previously, old nodes sometimes didn't get deleted if there were very frequent inserts, which caused the server to be slow to shut down, among other things.
- Fixed randomization when choosing hosts for the connection to ZooKeeper.
- Fixed the exclusion of lagging replicas in distributed queries if the replica is localhost.
- Fixed an error where a data part in a `ReplicatedMergeTree` table could be broken after running `ALTER MODIFY` on an element in a `Nested` structure.
- Fixed an error that could cause `SELECT` queries to "hang".
- Improvements to distributed DDL queries.
- Fixed the query `CREATE TABLE ... AS <materialized view>`.
- Resolved the deadlock in the `ALTER ... CLEAR COLUMN IN PARTITION` query for `Buffer` tables.
- Fixed the invalid default value for `Enum`s (0 instead of the minimum) when using the `JSONEachRow` and `TSKV` formats.
- Resolved the appearance of zombie processes when using a dictionary with an `executable` source.
- Fixed segfault for the `HEAD` query.

## Improved workflow for developing and assembling ClickHouse:

- You can use `pbuilder` to build ClickHouse.
- You can use `libc++` instead of `libstdc++` for builds on Linux.
- Added instructions for using static code analysis tools: `Coverage`, `clang-tidy`, `cppcheck`.

## Please note when upgrading:

- There is now a higher default value for the `MergeTree` setting `max_bytes_to_merge_at_max_space_in_pool` (the maximum total size of data parts to merge, in bytes): it has increased from 100 GiB to 150 GiB. This might result in large merges running after the server upgrade, which could cause an increased load on the disk subsystem. If the free space available on the server is less than twice the total amount of the merges that are running, this will cause all other merges to stop running, including merges of small data parts. As a result, `INSERT` queries will fail with the message "Merges are processing significantly slower than inserts." Use the `SELECT * FROM system.merges` query to monitor the situation. You can also check the `DiskSpaceReservedForMerge` metric in the `system.metrics` table, or in Graphite. You don't need to do anything to fix this, since the issue will resolve itself once the large merges finish. If you find this unacceptable, you can restore the previous value for the `max_bytes_to_merge_at_max_space_in_pool` setting. To do this, go to the section in `config.xml`, set

```
<merge_tree>` ` <max_bytes_to_merge_at_max_space_in_pool>107374182400</max_bytes_to_merge_at_max_space_in_pool>
```

 and restart the server.

## ClickHouse release 1.1.54284, 2017-08-29

- This is a bugfix release for the previous 1.1.54282 release. It fixes leaks in the parts directory in ZooKeeper.

## ClickHouse release 1.1.54282, 2017-08-23

This release contains bug fixes for the previous release 1.1.54276:

- Fixed `DB::Exception: Assertion violation: !_path.empty()` when inserting into a Distributed table.
- Fixed parsing when inserting in RowBinary format if input data starts with `;`.
- Errors during runtime compilation of certain aggregate functions (e.g. `groupArray()`).

## Clickhouse Release 1.1.54276, 2017-08-16

### New features:

- Added an optional WITH section for a SELECT query. Example query: `WITH 1+1 AS a SELECT a, a*a`
- INSERT can be performed synchronously in a Distributed table: OK is returned only after all the data is saved on all the shards. This is activated by the setting `insert_distributed_sync=1`.
- Added the UUID data type for working with 16-byte identifiers.
- Added aliases of CHAR, FLOAT and other types for compatibility with the Tableau.
- Added the functions `toYYYYMM`, `toYYYYMMDD`, and `toYYYYMMDDhhmmss` for converting time into numbers.
- You can use IP addresses (together with the hostname) to identify servers for clustered DDL queries.
- Added support for non-constant arguments and negative offsets in the function `substring(str, pos, len)`.
- Added the `max_size` parameter for the `groupArray(max_size)(column)` aggregate function, and optimized its performance.

### Main changes:

- Security improvements: all server files are created with 0640 permissions (can be changed via config parameter).
- Improved error messages for queries with invalid syntax.
- Significantly reduced memory consumption and improved performance when merging large sections of MergeTree data.
- Significantly increased the performance of data merges for the ReplacingMergeTree engine.
- Improved performance for asynchronous inserts from a Distributed table by combining multiple source inserts. To enable this functionality, use the setting `distributed_directory_monitor_batch_inserts=1`.

### Backward incompatible changes:

- Changed the binary format of aggregate states of `groupArray(array_column)` functions for arrays.

### Complete list of changes:

- Added the `output_format_json_quote_denormals` setting, which enables outputting nan and inf values in JSON format.
- Optimized stream allocation when reading from a Distributed table.
- Settings can be configured in readonly mode if the value doesn't change.
- Added the ability to retrieve non-integer granules of the MergeTree engine in order to meet restrictions on the block size specified in the `preferred_block_size_bytes` setting. The purpose is to reduce the consumption of RAM and increase cache locality when processing queries from tables with large columns.
- Efficient use of indexes that contain expressions like `toStartOfHour(x)` for conditions like `toStartOfHour(x) op constexpr`.
- Added new settings for MergeTree engines (the `merge_tree` section in `config.xml`):



- `replicated_deduplication_window_seconds` sets the number of seconds allowed for deduplicating inserts in Replicated tables.
- `cleanup_delay_period` sets how often to start cleanup to remove outdated data.
- `replicated_can_become_leader` can prevent a replica from becoming the leader (and assigning merges).
- Accelerated cleanup to remove outdated data from ZooKeeper.
- Multiple improvements and fixes for clustered DDL queries. Of particular interest is the new setting `distributed_ddl_task_timeout`, which limits the time to wait for a response from the servers in the cluster.
- Improved display of stack traces in the server logs.
- Added the "none" value for the compression method.
- You can use multiple `dictionaries_config` sections in `config.xml`.
- It is possible to connect to MySQL through a socket in the file system.
- The `system.parts` table has a new column with information about the size of marks, in bytes.

## Bug fixes:

- Distributed tables using a Merge table now work correctly for a SELECT query with a condition on the `_table` field.
- Fixed a rare race condition in `ReplicatedMergeTree` when checking data parts.
- Fixed possible freezing on "leader election" when starting a server.
- The `max_replica_delay_for_distributed_queries` setting was ignored when using a local replica of the data source. This has been fixed.
- Fixed incorrect behavior of `ALTER TABLE CLEAR COLUMN IN PARTITION` when attempting to clean a non-existing column.
- Fixed an exception in the `multif` function when using empty arrays or strings.
- Fixed excessive memory allocations when deserializing Native format.
- Fixed incorrect auto-update of Trie dictionaries.
- Fixed an exception when running queries with a GROUP BY clause from a Merge table when using SAMPLE.
- Fixed a crash of GROUP BY when using `distributed_aggregation_memory_efficient=1`.
- Now you can specify the `database.table` in the right side of IN and JOIN.
- Too many threads were used for parallel aggregation. This has been fixed.
- Fixed how the "if" function works with `FixedString` arguments.
- SELECT worked incorrectly from a Distributed table for shards with a weight of 0. This has been fixed.
- Running `CREATE VIEW IF EXISTS` no longer causes crashes.
- Fixed incorrect behavior when `input_format_skip_unknown_fields=1` is set and there are negative numbers.
- Fixed an infinite loop in the `dictGetHierarchy()` function if there is some invalid data in the dictionary.
- Fixed Syntax error: unexpected (...) errors when running distributed queries with subqueries in an IN or JOIN clause and Merge tables.
- Fixed an incorrect interpretation of a SELECT query from Dictionary tables.
- Fixed the "Cannot 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 query `ALTER TABLE CLEAR COLUMN IN PARTITION`.

- The engine for Dictionary tables (access to dictionary data in the form of a table).
- Dictionary database engine (this type of database automatically has Dictionary tables available for all the connected external dictionaries).
- You can check for updates to the dictionary by sending a request to the source.
- Qualified column names
- Quoting identifiers using double quotation marks.
- Sessions in the HTTP interface.
- The OPTIMIZE query for a Replicated table can run not only on the leader.

## Backward incompatible changes:

- Removed SET GLOBAL.

## Minor changes:

- Now after an alert is triggered, the log prints the full stack trace.
- Relaxed the verification of the number of damaged/extra data parts at startup (there were too many false positives).

## Bug fixes:

- Fixed a bad connection "sticking" when inserting into a Distributed table.
- GLOBAL IN now works for a query from a Merge table that looks at a Distributed table.
- The incorrect number of cores was detected on a Google Compute Engine virtual machine. This has been fixed.
- Changes in how an executable source of cached external dictionaries works.
- Fixed the comparison of strings containing null characters.
- Fixed the comparison of Float32 primary key fields with constants.
- Previously, an incorrect estimate of the size of a field could lead to overly large allocations.
- Fixed a crash when querying a Nullable column added to a table using ALTER.
- Fixed a crash when sorting by a Nullable column, if the number of rows is less than LIMIT.
- Fixed an ORDER BY subquery consisting of only constant values.
- Previously, a Replicated table could remain in the invalid state after a failed DROP TABLE.
- Aliases for scalar subqueries with empty results are no longer lost.
- Now a query that used compilation does not fail with an error if the .so file gets damaged.

## Fixed in ClickHouse Release 18.12.13, 2018-09-10

### CVE-2018-14672

Functions for loading CatBoost models allowed path traversal and reading arbitrary files through error messages.

Credits: Andrey Krasichkov of Yandex Information Security Team

## Fixed in ClickHouse Release 18.10.3, 2018-08-13

### CVE-2018-14671

unixODBC allowed loading arbitrary shared objects from the file system which led to a Remote Code Execution vulnerability.

Credits: Andrey Krasichkov and Evgeny Sidorov of Yandex Information Security Team

## Fixed in ClickHouse Release 1.1.54388, 2018-06-28

### CVE-2018-14668

"remote" table function allowed arbitrary symbols in "user", "password" and "default\_database" fields which led to Cross Protocol Request Forgery Attacks.

Credits: Andrey Krasichkov of Yandex Information Security Team

## Fixed in ClickHouse Release 1.1.54390, 2018-07-06

### CVE-2018-14669

ClickHouse MySQL client had "LOAD DATA LOCAL INFILE" functionality enabled that allowed a malicious MySQL database read arbitrary files from the connected ClickHouse server.

Credits: Andrey Krasichkov and Evgeny Sidorov of Yandex Information Security Team

## Fixed in ClickHouse Release 1.1.54131, 2017-01-10

### CVE-2018-14670

Incorrect configuration in deb package could lead to unauthorized use of the database.

Credits: the UK's National Cyber Security Centre (NCSC)

