Что такое ClickHouse

ClickHouse - столбцовая система управления базами данных (СУБД) для онлайн обработки аналитических запросов (OLAP).

В обычной, "строковой" СУБД, данные хранятся в таком порядке:

Строка	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	...	...	...	...	...

То есть, значения, относящиеся к одной строке, физически хранятся рядом.

Примеры строковых СУБД: MySQL, Postgres, MS SQL Server.

В столбцовых СУБД, данные хранятся в таком порядке:

Строка:	#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	...

В примерах изображён только порядок расположения данных.
То есть, значения из разных столбцов хранятся отдельно, а данные одного столбца - вместе.

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

Разный порядок хранения данных лучше подходит для разных сценариев работы.
Сценарий работы с данными - это то, какие производятся запросы, как часто и в каком соотношении; сколько читается данных на запросы каждого вида - строк, столбцов, байт; как соотносятся чтения и обновления данных; какой рабочий размер данных и насколько локально он используется; используются ли транзакции и с какой изолированностью; какие требования к дублированию данных и логической целостности; требования к задержкам на выполнение и пропускной способности запросов каждого вида и т. п.

Чем больше нагрузка на систему, тем более важной становится специализация под сценарий работы, и тем более конкретной становится эта специализация. Не существует системы, одинаково хорошо подходящей под существенно различные сценарии работы. Если система подходит под широкое множество сценариев работы, то при достаточно большой нагрузке, система будет справляться со всеми сценариями работы плохо, или справляться хорошо только с одним из сценариев работы.

К[illegible] столбцовые СУБД для OLAP [illegible]

Ключевые особенности OLAP сценария работы

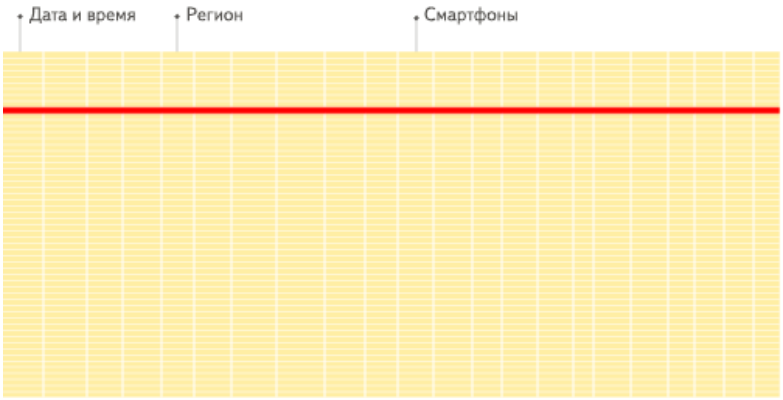
- подавляющее большинство запросов - на чтение;
- данные обновляются достаточно большими пачками (> 1000 строк), а не по одной строке, или не обновляются вообще;
- данные добавляются в БД, но не изменяются;
- при чтении, вынимается достаточно большое количество строк из БД, но только небольшое подмножество столбцов;
- таблицы являются "широкими", то есть, содержат большое количество столбцов;
- запросы идут сравнительно редко (обычно не более сотни в секунду на сервер);
- при выполнении простых запросов, допустимы задержки в районе 50 мс;
- значения в столбцах достаточно мелкие - числа и небольшие строки (пример - 60 байт на URL);
- требуется высокая пропускная способность при обработке одного запроса (до миллиардов строк в секунду на один сервер);
- транзакции отсутствуют;
- низкие требования к консистентности данных;
- в запросе одна большая таблица, все таблицы кроме одной маленькие;
- результат выполнения запроса существенно меньше исходных данных - то есть, данные фильтруются или агрегируются; результат выполнения помещается в оперативку на одном сервере.

Легко видеть, что OLAP сценарий работы существенно отличается от других распространённых сценариев работы (например, OLTP или Key-Value сценариев работы). Таким образом, не имеет никакого смысла пытаться использовать OLTP или Key-Value БД для обработки аналитических запросов, если вы хотите получить приличную производительность ("выше плинтуса"). Например, если вы попытаетесь использовать для аналитики MongoDB или Redis - вы получите анекдотически низкую производительность по сравнению с OLAP-СУБД.

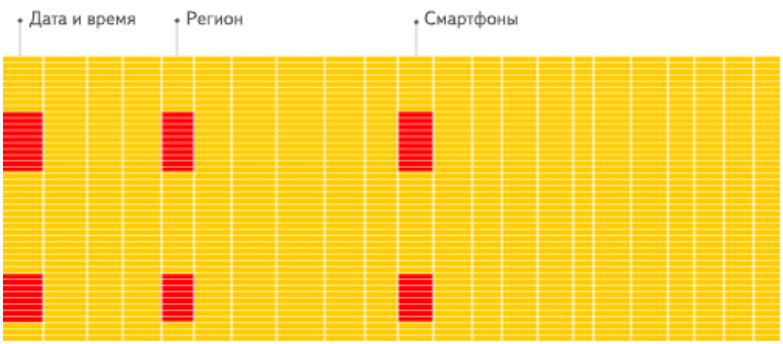
Причины, по которым столбцовые СУБД лучше подходят для OLAP сценария

Столбцовые СУБД лучше (от 100 раз по скорости обработки большинства запросов) подходят для OLAP сценария работы. Причины в деталях будут разъяснены ниже, а сам факт проще продемонстрировать визуально:

Строковые СУБД



Столбцовые СУБД



Видите разницу?

По вводу-выводу

1. Для выполнения аналитического запроса, требуется прочитать небольшое количество столбцов таблицы. В столбцовой БД для этого можно читать только нужные данные. Например, если вам требуется только 5 столбцов из 100, то следует рассчитывать на 20-кратное уменьшение ввода-вывода.
2. Так как данные читаются пачками, то их проще сжимать. Данные, лежащие по столбцам также лучше сжимаются. За счёт этого, дополнительно уменьшается объём ввода-вывода.
3. За счёт уменьшения ввода-вывода, больше данных влезает в системный кэш.

Например, для запроса "посчитать количество записей для каждой рекламной системы", требуется прочитать один столбец "идентификатор рекламной системы", который занимает 1 байт в несжатом виде. Если большинство переходов было не с рекламных систем, то можно рассчитывать хотя бы на десятикратное сжатие этого столбца. При использовании быстрого алгоритма сжатия, возможно разжатие данных со скоростью более нескольких гигабайт несжатых данных в секунду. То есть, такой запрос может выполняться со скоростью около нескольких миллиардов строк в секунду на одном сервере. На практике, такая скорость действительно достигается.

▼ Пример

```
$ 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.)

:)
```

По вычислениям

Так как для выполнения запроса надо обработать достаточно большое количество строк, становится актуальным диспетчеризовывать все операции не для отдельных строк, а для целых векторов, или реализовать движок выполнения запроса так, чтобы издержки на диспетчеризацию были примерно нулевыми. Если этого не делать, то при любой не слишком плохой дисковой подсистеме, интерпретатор запроса неизбежно упрётся в CPU.
Имеет смысл не только хранить данные по столбцам, но и обрабатывать их, по возможности, тоже по столбцам.

Есть два способа это сделать:

1. Векторный движок. Все операции пишутся не для отдельных значений, а для векторов. То есть, вызывать операции надо достаточно редко, и издержки на диспетчеризацию становятся пренебрежимо маленькими. Код операции содержит в себе хорошо оптимизированный внутренний цикл.

2. Кодогенерация. Для запроса генерируется код, в котором подставлены все косвенные вызовы.

В "обычных" БД этого не делается, так как не имеет смысла при выполнении простых запросов. Хотя есть исключения. Например, в MemSQL кодогенерация используется для уменьшения latency при выполнении SQL запросов. (Для сравнения - в аналитических СУБД, требуется оптимизация throughput, а не latency).

Стоит заметить, что для эффективности по CPU требуется, чтобы язык запросов был декларативным (SQL, MDX) или хотя бы векторным (J, K). То есть, чтобы запрос содержал циклы только в неявном виде, открывая возможности для оптимизации.

Отличительные возможности ClickHouse

По-настоящему столбцовая СУБД

В по-настоящему столбцовой СУБД рядом со значениями не хранится никаких лишних данных. Например, должны поддерживаться значения постоянной длины, чтобы не хранить рядом со значениями типа "число" их длины. Для примера, миллиард значений типа UInt8 должен действительно занимать в несжатом виде около 1GB, иначе это сильно ударит по эффективности использования CPU. Очень важно хранить данные компактно (без "мусора") в том числе в несжатом виде, так как скорость разжатия (использование CPU) зависит, в основном, от объёма несжатых данных.

Этот пункт пришлось выделить, так как существуют системы, которые могут хранить значения отдельных столбцов по отдельности, но не могут эффективно выполнять аналитические запросы в силу оптимизации под другой сценарий работы. Примеры: HBase, BigTable, Cassandra, HyperTable. В этих системах вы получите пропускную способность в районе сотен тысяч строк в секунду, но не сотен миллионов строк в секунду.

Также стоит заметить, что ClickHouse является системой управления базами данных, а не одной базой данных. То есть, ClickHouse позволяет создавать таблицы и базы данных в runtime, загружать данные и выполнять запросы без переконфигурирования и перезапуска сервера.

Сжатие данных

Некоторые столбцовые СУБД (InfiniDB CE, MonetDB) не используют сжатие данных. Однако сжатие данных действительно играет одну из ключевых ролей в демонстрации отличной производительности.

Хранение данных на диске

Многие столбцовые СУБД (SAP HANA, Google PowerDrill) могут работать только в оперативной памяти.

Такой подход стимулирует выделять больший бюджет на оборудование, чем фактически требуется для анализа в реальном времени. ClickHouse спроектирован для работы на обычных жестких дисках, что обеспечивает низкую стоимость хранения на гигабайт данных, но SSD и дополнительная оперативная память тоже полноценно используются, если доступны.

Параллельная обработка запроса на многих процессорных ядрах

Большие запросы естественным образом распараллеливаются, используя все необходимые ресурсы из доступных на сервере.

Распределённая обработка запроса на многих серверах

Почти все перечисленные ранее столбцовые СУБД не поддерживают распределённую обработку запроса.
В ClickHouse данные могут быть расположены на разных шардах. Каждый шард может представлять собой группу реплик, которые используются для отказоустойчивости. Запрос будет выполнен на всех шардах параллельно. Это делается прозрачно для пользователя.

Поддержка SQL

ClickHouse поддерживает декларативный язык запросов на основе SQL и во многих случаях совпадающий с SQL стандартом.
Поддерживаются GROUP BY, ORDER BY, подзапросы в секциях FROM, IN, JOIN, а также скалярные подзапросы.
Зависимые подзапросы и оконные функции не поддерживаются.

Векторный движок

Данные не только хранятся по столбцам, но и обрабатываются по векторам - кусочкам столбцов. За счёт этого достигается высокая эффективность по CPU.

Обновление данных в реальном времени

ClickHouse поддерживает таблицы с первичным ключом. Для того, чтобы можно было быстро выполнять запросы по диапазону первичного ключа, данные инкрементально сортируются с помощью merge дерева. За счёт этого, поддерживается постоянное добавление данных в таблицу. Блокировки при добавлении данных отсутствуют.

Наличие индекса

Физическая сортировка данных по первичному ключу позволяет получать данные для конкретных его значений или их диапазонов с низкими задержками - менее десятков миллисекунд.

Подходит для онлайн запросов

Низкие задержки позволяют не откладывать выполнение запроса и не подготавливать ответ заранее, а выполнять его именно в момент загрузки страницы пользовательского интерфейса. То есть, в режиме онлайн.

Поддержка приближённых вычислений

ClickHouse предоставляет различные способы разменять точность вычислений на производительность:

1. Система содержит агрегатные функции для приближённого вычисления количества различных значений, медианы и квантилей.
2. Поддерживается возможность выполнить запрос на основе части (выборки) данных и получить приближённый результат. При этом, с диска будет считано пропорционально меньше данных.
3. Поддерживается возможность выполнить агрегацию не для всех ключей, а для ограниченного количества первых попавшихся ключей. При выполнении некоторых условий на распределение ключей в данных, это позволяет получить достаточно точный результат с использованием

меньшего количества ресурсов.

Репликация данных и поддержка целостности

Используется асинхронная multimaster репликация. После записи на любую доступную реплику, данные распространяются на все остальные реплики в фоне. Система поддерживает полную идентичность данных на разных репликах. Восстановление после большинства сбоев осуществляется автоматически, а в сложных случаях — полуавтоматически. При необходимости, можно включить кворумную запись данных.

Подробнее смотрите раздел Репликация данных.

Особенности ClickHouse, которые могут считаться недостатками

1. Отсутствие полноценных транзакций.
2. Возможность изменять или удалять ранее записанные данные с низкими задержками и высокой частотой запросов не предоставляется. Есть массовое удаление и изменение данных для очистки более не нужного или соответствия GDPR.
3. Разреженный индекс делает ClickHouse плохо пригодным для точечных чтений одиночных строк по своим
   ключам.

Производительность

По результатам внутреннего тестирования в Яндексе, ClickHouse обладает наиболее высокой производительностью (как наиболее высокой пропускной способностью на длинных запросах, так и наиболее низкой задержкой на коротких запросах), при соответствующем сценарии работы, среди доступных для тестирования систем подобного класса. Результаты тестирования можно посмотреть на отдельной странице.

Также это подтверждают многочисленные независимые бенчмарки. Их не сложно найти в Интернете самостоятельно, либо можно воспользоваться небольшой коллекцией ссылок по теме.

Пропускная способность при обработке одного большого запроса

Пропускную способность можно измерять в строчках в секунду и в мегабайтах в секунду. При условии, что данные помещаются в page cache, не слишком сложный запрос обрабатывается на современном железе со скоростью около 2-10 GB/sec. несжатых данных на одном сервере (в простейшем случае скорость может достигать 30 GB/sec). Если данные не помещаются в page cache, то скорость работы зависит от скорости дисковой подсистемы и коэффициента сжатия данных. Например, если дисковая подсистема позволяет читать данные со скоростью 400 MB/sec., а коэффициент сжатия данных составляет 3, то скорость будет около 1.2GB/sec. Для получения скорости в строчках в секунду, следует поделить скорость в байтах в секунду на суммарный размер используемых в запросе столбцов. Например, если вынимаются столбцы на 10 байт, то скорость будет в районе 100-200 млн. строк в секунду.

При распределённой обработке запроса, скорость обработки запроса растёт почти линейно, но только при условии, что в результате агрегации или при сортировке получается не слишком большое множество строчек.

Задержки при обработке коротких запросов

Если запрос использует первичный ключ, и выбирает для обработки не слишком большое количество строчек (сотни тысяч), и использует не слишком большое количество столбцов, то вы можете рассчитывать на latency менее 50 миллисекунд (от единиц миллисекунд в лучшем случае), при условии, что данные помещаются в page cache. Иначе latency вычисляется из количества seek-ов. Если вы используйте вращающиеся диски, то на не слишком сильно нагруженной системе, latency вычисляется

по формуле: seek time (10 мс.) * количество столбцов в запросе * количество кусков с данными.

Пропускная способность при обработке многочисленных коротких запросов

При тех же условиях, ClickHouse может обработать несколько сотен (до нескольких тысяч в лучшем случае) запросов в секунду на одном сервере. Так как такой сценарий работы не является типичным для аналитических СУБД, рекомендуется рассчитывать не более чем на 100 запросов в секунду.

Производительность при вставке данных

Данные рекомендуется вставлять пачками не менее 1000 строк или не более одного запроса в секунду. При вставке в таблицу типа MergeTree из tab-separated дампа, скорость вставки будет в районе 50-200 МБ/сек. Если вставляются строчки размером около 1 КБ, то скорость будет в районе 50 000 - 200 000 строчек в секунду. Если строчки маленькие - производительность в строчках в секунду будет выше (на данных БК - `>` 500 000 строк в секунду, на данных Graphite - `>` 1 000 000 строк в секунду). Для увеличения производительности, можно производить несколько запросов INSERT параллельно - при этом производительность растёт линейно.

Постановка задачи в Яндекс.Метрике

ClickHouse изначально разрабатывался для обеспечения работы Яндекс.Метрики, второй крупнейшей в мире платформы для веб аналитики, и продолжает быть её ключевым компонентом. При более 13 триллионах записей в базе данных и более 20 миллиардах событий в сутки, ClickHouse позволяет генерировать индивидуально настроенные отчёты на лету напрямую из неагрегированных данных. Данная статья вкратце демонстрирует какие цели исторически стояли перед ClickHouse на ранних этапах его развития.

Яндекс.Метрика на лету строит индивидуальные отчёты на основе хитов и визитов, с периодом и произвольными сегментами, задаваемыми конечным пользователем. Часто требуется построение сложных агрегатов, например числа уникальных пользователей. Новые данные для построения отчета поступают в реальном времени.

На апрель 2014, в Яндекс.Метрику поступало около 12 миллиардов событий (показов страниц и кликов мыши) ежедневно. Все эти события должны быть сохранены для возможности строить произвольные отчёты. Один запрос может потребовать просканировать миллионы строк за время не более нескольких сотен миллисекунд, или сотни миллионов строк за время не более нескольких секунд.

Использование в Яндекс.Метрике и других отделах Яндекса

В Яндекс.Метрике ClickHouse используется для нескольких задач.
Основная задача - построение отчётов в режиме онлайн по неагрегированным данным. Для решения этой задачи используется кластер из 374 серверов, хранящий более 20,3 триллионов строк в базе данных. Объём сжатых данных, без учёта дублирования и репликации, составляет около 2 ПБ. Объём несжатых данных (в формате tsv) составил бы, приблизительно, 17 ПБ.

Также ClickHouse используется:

- для хранения данных Вебвизора;
- для обработки промежуточных данных;
- для построения глобальных отчётов Аналитиками;
- для выполнения запросов в целях отладки движка Метрики;
- для анализа логов работы API и пользовательского интерфейса.

ClickHouse имеет более десятка инсталляций в других отделах Яндекса: в Вертикальных сервисах, Маркете, Директе, БК, Бизнес аналитике, Мобильной разработке, AdFox, Персональных сервисах и т п.

Агрегированные и неагрегированные данные

Существует мнение, что для того, чтобы эффективно считать статистику, данные нужно агрегировать,

так как это позволяет уменьшить объем данных.

Но агрегированные данные являются очень ограниченным решением, по следующим причинам:

- вы должны заранее знать перечень отчётов, необходимых пользователю;
- то есть, пользователь не может построить произвольный отчёт;
- при агрегации по большому количеству ключей, объём данных не уменьшается и агрегация бесполезна;
- при большом количестве отчётов, получается слишком много вариантов агрегации (комбинаторный взрыв);
- при агрегации по ключам высокой кардинальности (например, URL) объём данных уменьшается не сильно (менее чем в 2 раза);
- из-за этого, объём данных при агрегации может не уменьшиться, а вырасти;
- пользователи будут смотреть не все отчёты, которые мы для них посчитаем - то есть, большая часть вычислений бесполезна;
- возможно нарушение логической целостности данных для разных агрегаций;

Как видно, если ничего не агрегировать, и работать с неагрегированными данными, то это даже может уменьшить объём вычислений.

Впрочем, при агрегации, существенная часть работы выносится в оффлайне, и её можно делать сравнительно спокойно. Для сравнения, при онлайн вычислениях, вычисления надо делать так быстро, как это возможно, так как именно в момент вычислений пользователь ждёт результата.

В Яндекс.Метрике есть специализированная система для агрегированных данных - Metrage, на основе которой работает большинство отчётов.
Также в Яндекс.Метрике с 2009 года использовалась специализированная OLAP БД для неагрегированных данных - OLAPServer, на основе которой раньше работал конструктор отчётов.
OLAPServer хорошо подходил для неагрегированных данных, но содержал много ограничений, не позволяющих использовать его для всех отчётов так, как хочется: отсутствие поддержки типов данных (только числа), невозможность инкрементального обновления данных в реальном времени (только перезаписью данных за сутки). OLAPServer не является СУБД, а является специализированной БД.

Чтобы снять ограничения OLAPServer-а и решить задачу работы с неагрегированными данными для всех отчётов, разработана СУБД ClickHouse.

Начало работы

Системные требования

ClickHouse может работать на любом Linux, FreeBSD или Mac OS X с архитектурой процессора x86_64.

Хотя предсобранные релизы обычно компилируются с использованием набора инструкций SSE 4.2, что добавляет использование поддерживающего его процессора в список системных требований. Команда для проверки наличия поддержки инструкций SSE 4.2 на текущем процессоре:

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

Установка

Из DEB пакетов

Яндекс рекомендует использовать официальные скомпилированные `deb` пакеты для Debian или Ubuntu.

Чтобы установить официальные пакеты, пропишите репозиторий Яндекса в `/etc/apt/sources.list` или в отдельный файл `/etc/apt/sources.list.d/clickhouse.list`:

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

Если вы хотите использовать наиболее свежую тестовую, замените `stable` на `testing` (не рекомендуется

Если вы хотите использовать наиболее свежую тестовую, замените `stable` на `testing` (не рекомендуется для production окружений).

Затем для самой установки пакетов выполните:

```
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
```

Также эти пакеты можно скачать и установить вручную отсюда: https://repo.yandex.ru/clickhouse/deb/stable/main/.

Из RPM пакетов

Яндекс не использует ClickHouse на поддерживающих `rpm` дистрибутивах Linux, а также `rpm` пакеты менее тщательно тестируются. Таким образом, использовать их стоит на свой страх и риск, но, тем не менее, многие другие компании успешно работают на них в production без каких-либо серьезных проблем.

Для CentOS, RHEL и Fedora возможны следующие варианты:

- Пакеты из https://repo.yandex.ru/clickhouse/rpm/stable/x86_64/ генерируются на основе официальных `deb` пакетов от Яндекса и содержат в точности тот же исполняемый файл.
- Пакеты из https://github.com/Altinity/clickhouse-rpm-install собираются независимой компанией Altinity, но широко используются без каких-либо нареканий.
- Либо можно использовать Docker (см. ниже).

Из Docker образа

Для запуска ClickHouse в Docker нужно следовать инструкции на Docker Hub. Внутри образов используются официальные `deb` пакеты.

Из исходникого кода

Для компиляции ClickHouse вручную, используйте инструкцию для Linux или Mac OS X.

Можно скомпилировать пакеты и установить их, либо использовать программы без установки пакетов. Также при ручой сборке можно отключить необходимость поддержки набора инструкций SSE 4.2 или собрать под процессоры архитектуры AArch64.

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

Для работы собранного вручную сервера необходимо создать директории для данных и метаданных, а также сделать их `chown` для желаемого пользователя. Пути к этим директориям могут быть изменены в конфигурационном файле сервера (src/dbms/programs/server/config.xml), по умолчанию используются следующие:

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

На Gentoo для установки ClickHouse из исходного кода можно использовать просто `emerge clickhouse`.

Запуск

Для запуска сервера в качестве демона, выполните:

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

Смотрите логи в директории `/var/log/clickhouse-server/`.

Если сервер не стартует, проверьте корректность конфигурации в файле `/etc/clickhouse-server/config.xml`

Также можно запустить сервер вручную из консоли:

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

При этом, лог будет выводиться в консоль, что удобно для разработки.
Если конфигурационный файл лежит в текущей директории, то указывать параметр `--config-file` не требуется, по умолчанию будет использован файл `./config.xml`.

После запуска сервера, соединиться с ним можно с помощью клиента командной строки:

```
$ clickhouse-client
```

По умолчанию он соединяется с localhost:9000, от имени пользователя `default` без пароля. Также клиент может быть использован для соединения с удалённым сервером с помощью аргумента `--host`.

Терминал должен использовать кодировку UTF-8.

Более подробная информация о клиенте располагается в разделе «Клиент командной строки».

Пример проверки работоспособности системы:

```
$ ./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.

:)
```

Поздравляем, система работает!

Для дальнейших экспериментов можно попробовать загрузить один из тестовых наборов данных или пройти пошаговое руководство для начинающих.

OnTime

Этот датасет может быть получен двумя способами:

- импорт из сырых данных;
- скачивание готовых партиций.

Импорт из сырых данных

Скачивание данных:

```
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
```

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

Создание таблицы:

```
CREATE TABLE `ontime` (
 `Year` UInt16,
 `Quarter` UInt8,
 `Month` UInt8,
 `DayofMonth` UInt8,
 `DayOfWeek` UInt8,
 `FlightDate` Date,
 `UniqueCarrier` FixedString(7),
 `AirlineID` Int32,
 `Carrier` FixedString(2),
 `TailNum` String,
 `FlightNum` String,
 `OriginAirportID` Int32,
 `OriginAirportSeqID` Int32,
 `OriginCityMarketID` Int32,
 `Origin` FixedString(5),
 `OriginCityName` String,
 `OriginState` FixedString(2),
 `OriginStateFips` String,
 `OriginStateName` String,
 `OriginWac` Int32,
 `DestAirportID` Int32,
 `DestAirportSeqID` Int32,
 `DestCityMarketID` Int32,
 `Dest` FixedString(5),
 `DestCityName` String,
 `DestState` FixedString(2),
 `DestStateFips` String,
 `DestStateName` String,
 `DestWac` Int32,
 `CRSDepTime` Int32,
 `DepTime` Int32,
 `DepDelay` Int32,
 `DepDelayMinutes` Int32,
 `DepDel15` Int32,
 `DepartureDelayGroups` String,
 `DepTimeBlk` String,
 `TaxiOut` Int32,
 `WheelsOff` Int32,
 `WheelsOn` Int32,
 `TaxiIn` Int32,
 `CRSArrTime` Int32,
 `ArrTime` Int32,
 `ArrDelay` Int32,
 `ArrDelayMinutes` Int32,
 `ArrDel15` Int32,
 `ArrivalDelayGroups` Int32,
 `ArrTimeBlk` String,
 `Cancelled` UInt8,
 `CancellationCode` FixedString(1),
 `Diverted` UInt8,
 `CRSElapsedTime` Int32,
```

```
  `ActualElapsedTime` Int32,
  `AirTime` Int32,
  `Flights` Int32,
  `Distance` Int32,
  `DistanceGroup` UInt8,
  `CarrierDelay` Int32,
  `WeatherDelay` Int32,
  `NASDelay` Int32,
  `SecurityDelay` Int32,
  `LateAircraftDelay` Int32,
  `FirstDepTime` String,
  `TotalAddGTime` String,
  `LongestAddGTime` String,
  `DivAirportLandings` String,
  `DivReachedDest` String,
  `DivActualElapsedTime` String,
  `DivArrDelay` String,
  `DivDistance` String,
  `Div1Airport` String,
  `Div1AirportID` Int32,
  `Div1AirportSeqID` Int32,
  `Div1WheelsOn` String,
  `Div1TotalGTime` String,
  `Div1LongestGTime` String,
  `Div1WheelsOff` String,
  `Div1TailNum` String,
  `Div2Airport` String,
  `Div2AirportID` Int32,
  `Div2AirportSeqID` Int32,
  `Div2WheelsOn` String,
  `Div2TotalGTime` String,
  `Div2LongestGTime` String,
  `Div2WheelsOff` String,
  `Div2TailNum` String,
  `Div3Airport` String,
  `Div3AirportID` Int32,
  `Div3AirportSeqID` Int32,
  `Div3WheelsOn` String,
  `Div3TotalGTime` String,
  `Div3LongestGTime` String,
  `Div3WheelsOff` String,
  `Div3TailNum` String,
  `Div4Airport` String,
  `Div4AirportID` Int32,
  `Div4AirportSeqID` Int32,
  `Div4WheelsOn` String,
  `Div4TotalGTime` String,
  `Div4LongestGTime` String,
  `Div4WheelsOff` String,
  `Div4TailNum` String,
  `Div5Airport` String,
  `Div5AirportID` Int32,
  `Div5AirportSeqID` Int32,
  `Div5WheelsOn` String,
  `Div5TotalGTime` String,
  `Div5LongestGTime` String,
  `Div5WheelsOff` String,
  `Div5TailNum` String
) ENGINE = MergeTree(FlightDate, (Year, FlightDate), 8192)
```

Загрузка данных:

```
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
```

Скачивание готовых партиций

```
curl -O https://clickhouse-datasets.s3.yandex.net/ontime/partitions/ontime.tar
tar xvf ontime.tar -C /var/lib/clickhouse # путь к папке с данными ClickHouse
## убедитесь, что установлены корректные права доступа на файлы
sudo service clickhouse-server restart
clickhouse-client --query "SELECT COUNT(*) FROM datasets.ontime"
```

Info

Если вы собираетесь выполнять запросы, приведенные ниже, то к имени таблицы нужно добавить имя базы, `datasets.ontime`.

Запросы:

Q0.

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

Q1. Количество полетов в день с 2000 по 2008 года

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

Q2. Количество полетов, задержанных более чем на 10 минут, с группировкой по дням неделе, за 2000-2008 года

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

Q3. Количество задержек по аэропортам за 2000-2008

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

Q4. Количество задержек по перевозчикам за 2007 год

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

Q5. Процент задержек по перевозчикам за 2007 год

```
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;
```

Более оптимальная версия того же запроса:

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

Q6. Предыдущий запрос за более широкий диапазон лет, 2000-2008

```
SELECT Carrier, c, c2, c*1000/c2 as c3
FROM
(
    SELECT
        Carrier,
        count(*) AS c
    FROM ontime
    WHERE DepDelay>10
        AND Year >= 2000 AND Year <= 2008
    GROUP BY Carrier
)
ANY INNER JOIN
(
    SELECT
        Carrier,
        count(*) AS c2
    FROM ontime
    WHERE Year >= 2000 AND Year <= 2008
    GROUP BY Carrier
) USING Carrier
ORDER BY c3 DESC;
```

Более оптимальная версия того же запроса:

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

Q7. Процент полетов, задержанных на более 10 минут, в разбивке по годам

```
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
```

Более оптимальная версия того же запроса:

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

Q8. Самые популярные направления по количеству напрямую соединенных городов для различных диапазонов лет

```
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;
```

Бонус:

```
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;
```

Данный тест производительности был создан Вадимом Ткаченко, статьи по теме:

- 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/

- 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

Данные о такси в Нью-Йорке

Этот датасет может быть получен двумя способами:

- импорт из сырых данных;
- скачивание готовых партиций.

Как импортировать сырые данные

См. https://github.com/toddwschneider/nyc-taxi-data и http://tech.marksblogg.com/billion-nyc-taxi-rides-redshift.html для описания набора данных и инструкций по загрузке.

После скачивания получится порядка 227 Гб несжатых данных в CSV файлах. Скачивание занимает порядка часа на 1 Гбит соединении (параллельное скачивание с s3.amazonaws.com утилизирует как минимум половину гигабитного канала).
Некоторые файлы могут скачаться не полностью. Проверьте размеры файлов и скачайте повторно подозрительные.

Некоторые файлы могут содержать некорректные строки. Их можно скорректировать следующим образом:

```
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
```

Далее данные должны быть предобработаны в PostgreSQL. Будут сделаны выборки точек в полигонах (для установки соответствия точек на карте с районами Нью-Йорка) и объединение всех данных в одну денормализованную плоскую таблицу с помощью JOIN. Для этого потребуется установить PostgreSQL с поддержкой PostGIS.

При запуске `initialize_database.sh`, будьте осторожны и вручную перепроверьте, что все таблицы корректно создались.

Обработка каждого месяца данных в PostgreSQL занимает около 20-30 минут, в сумме порядка 48 часов.

Проверить количество загруженных строк можно следующим образом:

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

real    7m9.164s
```

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

Данные в PostgreSQL занимают 370 Гб.

Экспорт данных из PostgreSQL:

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

```
            trips.store_and_fwd_flag,
            trips.rate_code_id,
            trips.pickup_longitude,
            trips.pickup_latitude,
            trips.dropoff_longitude,
            trips.dropoff_latitude,
            trips.passenger_count,
            trips.trip_distance,
            trips.fare_amount,
            trips.extra,
            trips.mta_tax,
            trips.tip_amount,
            trips.tolls_amount,
            trips.ehail_fee,
            trips.improvement_surcharge,
            trips.total_amount,
            trips.payment_type,
            trips.trip_type,
            trips.pickup,
            trips.dropoff,

            cab_types.type cab_type,

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

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

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

Слепок данных создается со скоростью около 50 Мб в секунду. Во время создания слепка, PostgreSQL читает с диска со скоростью около 28 Мб в секунду.

Это занимает около 5 часов. Результирующий tsv файл имеет размер в 590612904969 байт.

Создание временной таблицы в ClickHouse:

```
 CREATE TABLE trips
(
trip_id                 UInt32,
vendor_id               String,
pickup_datetime         DateTime,
dropoff_datetime        Nullable(DateTime),
store_and_fwd_flag      Nullable(FixedString(1)),
rate_code_id            Nullable(UInt8),
pickup_longitude        Nullable(Float64),
pickup_latitude         Nullable(Float64),
dropoff_longitude       Nullable(Float64),
dropoff_latitude        Nullable(Float64),
passenger_count         Nullable(UInt8),
trip_distance           Nullable(Float64),
fare_amount             Nullable(Float32),
extra                   Nullable(Float32),
mta_tax                 Nullable(Float32),
tip_amount              Nullable(Float32),
tolls_amount            Nullable(Float32),
ehail_fee               Nullable(Float32),
improvement_surcharge   Nullable(Float32),
total_amount            Nullable(Float32),
payment_type            Nullable(String),
trip_type               Nullable(UInt8),
pickup                  Nullable(String),
dropoff                 Nullable(String),
cab_type                Nullable(String),
precipitation           Nullable(UInt8),
snow_depth              Nullable(UInt8),
snowfall                Nullable(UInt8),
max_temperature         Nullable(UInt8),
min_temperature         Nullable(UInt8),
average_wind_speed      Nullable(UInt8),
pickup_nyct2010_gid     Nullable(UInt8),
pickup_ctlabel          Nullable(String),
pickup_borocode         Nullable(UInt8),
pickup_boroname         Nullable(String),
pickup_ct2010           Nullable(String),
pickup_boroct2010       Nullable(String),
pickup_cdeligibil       Nullable(FixedString(1)),
pickup_ntacode          Nullable(String),
pickup_ntaname          Nullable(String),
pickup_puma             Nullable(String),
dropoff_nyct2010_gid    Nullable(UInt8),
dropoff_ctlabel         Nullable(String),
dropoff_borocode        Nullable(UInt8),
dropoff_boroname        Nullable(String),
dropoff_ct2010          Nullable(String),
dropoff_boroct2010      Nullable(String),
dropoff_cdeligibil      Nullable(String),
dropoff_ntacode         Nullable(String),
dropoff_ntaname         Nullable(String),
dropoff_puma            Nullable(String)
) ENGINE = Log;
```

Она нужна для преобразование полей к более правильным типам данных и, если возможно, чтобы избавиться от NULL'ов.

```
```

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

real    75m56.214s
```

Данные читаются со скоростью 112-140 Мб в секунду.
Загрузка данных в таблицу типа Log в один поток заняла 76 минут.
Данные в этой таблице занимают 142 Гб.

(Импорт данных напрямую из Postgres также возможен с использованием `COPY ... TO PROGRAM`.)

К сожалению, все поля, связанные с погодой (precipitation...average_wind_speed) заполнены NULL. Из-за этого мы исключим их из финального набора данных.

Для начала мы создадим таблицу на одном сервере. Позже мы сделаем таблицу распределенной.

Создадим и заполним итоговую таблицу:

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

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

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

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

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

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

assumeNotNull(dropoff_nyct2010_gid) AS dropoff_nyct2010_gid,
toFloat32(ifNull(dropoff_ctlabel, '0')) AS dropoff_ctlabel,
assumeNotNull(dropoff_borocode) AS dropoff_borocode,
CAST(assumeNotNull(dropoff_boroname) AS Enum8('Manhattan' = 1, 'Queens' = 4, 'Brooklyn' = 3, '' = 0, 'Bronx' = 2, 'Staten Island' = 5)) AS dropoff_boroname,
toFixedString(ifNull(dropoff_ct2010, '000000'), 6) AS dropoff_ct2010,
toFixedString(ifNull(dropoff_boroct2010, '0000000'), 7) AS dropoff_boroct2010,
CAST(assumeNotNull(ifNull(dropoff_cdeligibil, ' ')) AS Enum8(' ' = 0, 'E' = 1, 'I' = 2)) AS dropoff_cdeligibil,
toFixedString(ifNull(dropoff_ntacode, '0000'), 4) AS dropoff_ntacode,

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

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

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

FROM trips
```

Это занимает 3030 секунд со скоростью около 428 тысяч строк в секунду.
Для более короткого времени загрузки, можно создать таблицу с движком `Log` вместо `MergeTree`. В этом случае загрузка отработает быстрее, чем за 200 секунд.

Таблица заняла 126 Гб дискового пространства.

```
:) 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))─┐
│ 126.10 GiB                     │
```

```
| 126.18 GiB                    |
└───────────────────────────────┘
```

Между прочим, на MergeTree можно запустить запрос OPTIMIZE. Но это не обязательно, всё будет в порядке и без этого.

Скачивание готовых партиций

```
curl -O https://clickhouse-datasets.s3.yandex.net/trips_mergetree/partitions/trips_mergetree.tar
tar xvf trips_mergetree.tar -C /var/lib/clickhouse # путь к папке с данными ClickHouse
## убедитесь, что установлены корректные права доступа на файлы
sudo service clickhouse-server restart
clickhouse-client --query "SELECT COUNT(*) FROM datasets.trips_mergetree"
```

Info

Если вы собираетесь выполнять запросы, приведенные ниже, то к имени таблицы нужно добавить имя базы, `datasets.trips_mergetree`.

Результаты на одном сервере

Q1:

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

0.490 секунд.

Q2:

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

1.224 секунд.

Q3:

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

2.104 секунд.

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 секунд.

Использовался следующий сервер:

Два Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz, в сумме 16 физических ядер,
128 GiB RAM,
8x6 TB HD на программном RAID-5

Время выполнения — лучшее из трех запусков
На самом деле начиная со второго запуска, запросы читают данные из кеша страниц файловой системы. Никакого дальнейшего кеширования не происходит: данные зачитываются и обрабатываются при

каждом запуске.

Создание таблицы на 3 серверах:

На каждом сервере:

```
CREATE TABLE default.trips_mergetree_third ( trip_id UInt32, vendor_id Enum8('1' = 1, '2' = 2, 'CMT' = 3, 'VTS' = 4, 'DDS' = 5, 'B02512' = 10, 'B02598' = 11, 'B02617' = 12, 'B02682' = 13, 'B02764' = 14), pickup_date Date, pickup_datetime DateTime, dropoff_date Date, dropoff_datetime DateTime, store_and_fwd_flag UInt8, rate_code_id UInt8, pickup_longitude Float64, pickup_latitude Float64, dropoff_longitude Float64, dropoff_latitude Float64, passenger_count UInt8, trip_distance Float64, fare_amount Float32, extra Float32, mta_tax Float32, tip_amount Float32, tolls_amount Float32, ehail_fee Float32, improvement_surcharge Float32, total_amount Float32, payment_type_ Enum8('UNK' = 0, 'CSH' = 1, 'CRE' = 2, 'NOC' = 3, 'DIS' = 4), trip_type UInt8, pickup FixedString(25), dropoff FixedString(25), cab_type Enum8('yellow' = 1, 'green' = 2, 'uber' = 3), pickup_nyct2010_gid UInt8, pickup_ctlabel Float32, pickup_borocode UInt8, pickup_boroname Enum8('' = 0, 'Manhattan' = 1, 'Bronx' = 2, 'Brooklyn' = 3, 'Queens' = 4, 'Staten Island' = 5), pickup_ct2010 FixedString(6), pickup_boroct2010 FixedString(7), pickup_cdeligibil Enum8(' ' = 0, 'E' = 1, 'I' = 2), pickup_ntacode FixedString(4), pickup_ntaname Enum16('' = 0, 'Airport' = 1, 'Allerton-Pelham Gardens' = 2, 'Annadale-Huguenot-Prince\'s Bay-Eltingville' = 3, 'Arden Heights' = 4, 'Astoria' = 5, 'Auburndale' = 6, 'Baisley Park' = 7, 'Bath Beach' = 8, 'Battery Park City-Lower Manhattan' = 9, 'Bay Ridge' = 10, 'Bayside-Bayside Hills' = 11, 'Bedford' = 12, 'Bedford Park-Fordham North' = 13, 'Bellerose' = 14, 'Belmont' = 15, 'Bensonhurst East' = 16, 'Bensonhurst West' = 17, 'Borough Park' = 18, 'Breezy Point-Belle Harbor-Rockaway Park-Broad Channel' = 19, 'Briarwood-Jamaica Hills' = 20, 'Brighton Beach' = 21, 'Bronxdale' = 22, 'Brooklyn Heights-Cobble Hill' = 23, 'Brownsville' = 24, 'Bushwick North' = 25, 'Bushwick South' = 26, 'Cambria Heights' = 27, 'Canarsie' = 28, 'Carroll Gardens-Columbia Street-Red Hook' = 29, 'Central Harlem North-Polo Grounds' = 30, 'Central Harlem South' = 31, 'Charleston-Richmond Valley-Tottenville' = 32, 'Chinatown' = 33, 'Claremont-Bathgate' = 34, 'Clinton' = 35, 'Clinton Hill' = 36, 'Co-op City' = 37, 'College Point' = 38, 'Corona' = 39, 'Crotona Park East' = 40, 'Crown Heights North' = 41, 'Crown Heights South' = 42, 'Cypress Hills-City Line' = 43, 'DUMBO-Vinegar Hill-Downtown Brooklyn-Boerum Hill' = 44, 'Douglas Manor-Douglaston-Little Neck' = 45, 'Dyker Heights' = 46, 'East Concourse-Concourse Village' = 47, 'East Elmhurst' = 48, 'East Flatbush-Farragut' = 49, 'East Flushing' = 50, 'East Harlem North' = 51, 'East Harlem South' = 52, 'East New York' = 53, 'East New York (Pennsylvania Ave)' = 54, 'East Tremont' = 55, 'East Village' = 56, 'East Williamsburg' = 57, 'Eastchester-Edenwald-Baychester' = 58, 'Elmhurst' = 59, 'Elmhurst-Maspeth' = 60, 'Erasmus' = 61, 'Far Rockaway-Bayswater' = 62, 'Flatbush' = 63, 'Flatlands' = 64, 'Flushing' = 65, 'Fordham South' = 66, 'Forest Hills' = 67, 'Fort Greene' = 68, 'Fresh Meadows-Utopia' = 69, 'Ft. Totten-Bay Terrace-Clearview' = 70, 'Georgetown-Marine Park-Bergen Beach-Mill Basin' = 71, 'Glen Oaks-Floral Park-New Hyde Park' = 72, 'Glendale' = 73, 'Gramercy' = 74, 'Grasmere-Arrochar-Ft. Wadsworth' = 75, 'Gravesend' = 76, 'Great Kills' = 77, 'Greenpoint' = 78, 'Grymes Hill-Clifton-Fox Hills' = 79, 'Hamilton Heights' = 80, 'Hammels-Arverne-Edgemere' = 81, 'Highbridge' = 82, 'Hollis' = 83, 'Homecrest' = 84, 'Hudson Yards-Chelsea-Flatiron-Union Square' = 85, 'Hunters Point-Sunnyside-West Maspeth' = 86, 'Hunts Point' = 87, 'Jackson Heights' = 88, 'Jamaica' = 89, 'Jamaica Estates-Holliswood' = 90, 'Kensington-Ocean Parkway' = 91, 'Kew Gardens' = 92, 'Kew Gardens Hills' = 93, 'Kingsbridge Heights' = 94, 'Laurelton' = 95, 'Lenox Hill-Roosevelt Island' = 96, 'Lincoln Square' = 97, 'Lindenwood-Howard Beach' = 98, 'Longwood' = 99, 'Lower East Side' = 100, 'Madison' = 101, 'Manhattanville' = 102, 'Marble Hill-Inwood' = 103, 'Mariner\'s Harbor-Arlington-Port Ivory-Graniteville' = 104, 'Maspeth' = 105, 'Melrose South-Mott Haven North' = 106, 'Middle Village' = 107, 'Midtown-Midtown South' = 108, 'Midwood' = 109, 'Morningside Heights' = 110, 'Morrisania-Melrose' = 111, 'Mott Haven-Port Morris' = 112, 'Mount Hope' = 113, 'Murray Hill' = 114, 'Murray Hill-Kips Bay' = 115, 'New Brighton-Silver Lake' = 116, 'New Dorp-Midland Beach' = 117, 'New Springville-Bloomfield-Travis' = 118, 'North Corona' = 119, 'North Riverdale-Fieldston-Riverdale' = 120, 'North Side-South Side' = 121, 'Norwood' = 122, 'Oakland Gardens' = 123, 'Oakwood-Oakwood Beach' = 124, 'Ocean Hill' = 125, 'Ocean Parkway South' = 126, 'Old Astoria' = 127, 'Old Town-Dongan Hills-South Beach' = 128, 'Ozone Park' = 129, 'Park Slope-Gowanus' = 130, 'Parkchester' = 131, 'Pelham Bay-Country Club-City Island' = 132, 'Pelham Parkway' = 133, 'Pomonok-Flushing Heights-Hillcrest' = 134, 'Port Richmond' = 135, 'Prospect Heights' = 136, 'Prospect Lefferts Gardens-Wingate' = 137, 'Queens Village' = 138, 'Queensboro Hill' = 139, 'Queensbridge-Ravenswood-Long Island City' = 140, 'Rego Park' = 141, 'Richmond Hill' = 142, 'Ridgewood' = 143, 'Rikers Island' = 144, 'Rosedale' = 145, 'Rossville-Woodrow' = 146, 'Rugby-Remsen Village' = 147, 'Schuylerville-Throgs Neck-Edgewater Park' = 148, 'Seagate-Coney Island' = 149, 'Sheepshead Bay-Gerritsen Beach-Manhattan Beach' = 150, 'SoHo-TriBeCa-Civic Center-Little Italy' = 151, 'Soundview-Bruckner' = 152, 'Soundview-Castle Hill-Clason Point-Harding Park' = 153, 'South Jamaica' = 154, 'South Ozone Park' = 155, 'Springfield Gardens North' = 156, 'Springfield Gardens South-Brookville' = 157, 'Spuyten Duyvil-Kingsbridge' = 158, 'St. Albans' = 159, 'Stapleton-Rosebank' = 160, 'Starrett City' = 161, 'Steinway' = 162, 'Stuyvesant Heights' = 163, 'Stuyvesant Town-Cooper Village' = 164, 'Sunset Park East' = 165, 'Sunset Park West' = 166, 'Todt Hill-Emerson Hill-Heartland Village-Lighthouse Hill' = 167, 'Turtle Bay-East Midtown' = 168, 'University Heights-Morris Heights' = 169, 'Upper East Side-Carnegie Hill' = 170, 'Upper West Side' = 171, 'Van Cortlandt Village' = 172, 'Van Nest-Morris Park-Westchester Square' = 173, 'Washington Heights North' = 174, 'Washington Heights South' = 175, 'West Brighton' = 176, 'West Concourse' = 177, 'West Farms-Bronx River' = 178, 'West New Brighton-New Brighton-St. George' = 179, 'West Village' = 180, 'Westchester-Unionport' = 181, 'Westerleigh' = 182, 'Whitestone' = 183, 'Williamsbridge-Olinville' = 184, 'Williamsburg' = 185, 'Windsor Terrace' = 186, 'Woodhaven' = 187, 'Woodlawn-Wakefield' = 188, 'Woodside' = 189, 'Yorkville' = 190, 'park-cemetery-etc-Bronx' = 191, 'park-cemetery-etc-Brooklyn' = 192, 'park-cemetery-etc-Manhattan' = 193, 'park-cemetery-etc-Queens' = 194, 'park-cemetery-etc-Staten Island' = 195), pickup_puma UInt16, dropoff_nyct2010_gid UInt8, dropoff_ctlabel Float32, dropoff_borocode UInt8,
```

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

На исходном сервере:

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

Следующим запрос перераспределит данные:

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

Это занимает 2454 секунд.

На трёх серверах:

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

Никакого сюрприза, так как запросы масштабируются линейно.

Также у нас есть результаты с кластера из 140 серверов:

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

В этом случае, время выполнения запросов определяется в первую очередь сетевыми задержками.
Мы выполняли запросы с помощью клиента, расположенного в датацентре Яндекса в Мянтсяля (Финляндия), на кластер в России, что добавляет порядка 20 мс задержки.

Резюме

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

AMPLab Big Data Benchmark

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

Зарегистрируйте бесплатную учетную запись на https://aws.amazon.com - понадобится кредитная карта, email и номер телефона
Получите новый ключ доступа на https://console.aws.amazon.com/iam/home?nc2=h_m_sc#security_credential

Выполните следующее в консоли:

```
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 ..
```

Выполните следующие запросы к ClickHouse:

```
CREATE TABLE rankings_tiny
(
    pageURL String,
    pageRank UInt32,
    avgDuration UInt32
```

```
    avgDuration UInt32
) ENGINE = Log;

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

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

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

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

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

Возвращаемся в консоль:

```
 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
```

```
query="INSERT INTO rankings_1node FORMAT CSV"; done
for i in 1node/uservisits/*.deflate; do echo $i; zlib-flate -uncompress < $i | clickhouse-client --host=example-perftest01j --query="INSERT INTO uservisits_1node FORMAT CSV"; done
for i in 5nodes/rankings/*.deflate; do echo $i; zlib-flate -uncompress < $i | clickhouse-client --host=example-perftest01j --query="INSERT INTO rankings_5nodes_on_single FORMAT CSV"; done
for i in 5nodes/uservisits/*.deflate; do echo $i; zlib-flate -uncompress < $i | clickhouse-client --host=example-perftest01j --query="INSERT INTO uservisits_5nodes_on_single FORMAT CSV"; done
```

Запросы для получения выборок данных:

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

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

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

WikiStat

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

Создание таблицы:

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

Загрузка данных:

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

Терабайт логов кликов от Criteo

Терабайт логов кликов от Criteo

Скачайте данные с http://labs.criteo.com/downloads/download-terabyte-click-logs/

Создайте таблицу для импорта лога:

```
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
```

Загрузите данные:

```
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
```

Создайте таблицу для сконвертированных данных:

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

```
) ENGINE = MergeTree(date, intHash32(icat1), (date, intHash32(icat1)), 8192)
```

Преобразуем данные из сырого лога и положим во вторую таблицу:

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

DROP TABLE criteo_log;
```

Star Schema Benchmark

Compiling dbgen:

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

Generating data:

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

Creating tables in ClickHouse:

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

CREATE TABLE lineorder
(
    LO_ORDERKEY             UInt32,
    LO_LINENUMBER           UInt8,
    LO_CUSTKEY              UInt32,
    LO_PARTKEY              UInt32,
    LO_SUPPKEY              UInt32,
    LO_ORDERDATE            Date,
```

```
    LO_ORDERPRIORITY        LowCardinality(String),
    LO_SHIPPRIORITY         UInt8,
    LO_QUANTITY             UInt8,
    LO_EXTENDEDPRICE        UInt32,
    LO_ORDTOTALPRICE        UInt32,
    LO_DISCOUNT             UInt8,
    LO_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;
```

```
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;
```

Анонимизированные данные Яндекс.Метрики

Датасет состоит из двух таблиц, содержащих анонимизированные данные о хитах (`hits_v1`) и визитах (`visits_v1`) Яндекс.Метрики. Каждую из таблиц можно скачать в виде сжатого `.tsv.xz`-файла или в виде уже готовых партиций.

Получение таблиц из партиций

Скачивание и импортирование партиций hits:

```
curl -O https://clickhouse-datasets.s3.yandex.net/hits/partitions/hits_v1.tar
tar xvf hits_v1.tar -C /var/lib/clickhouse # путь к папке с данными ClickHouse
## убедитесь, что установлены корректные права доступа на файлы
sudo service clickhouse-server restart
clickhouse-client --query "SELECT COUNT(*) FROM datasets.hits_v1"
```

Скачивание и импортирование партиций visits:

```
curl -O https://clickhouse-datasets.s3.yandex.net/visits/partitions/visits_v1.tar
tar xvf visits_v1.tar -C /var/lib/clickhouse # путь к папке с данными ClickHouse
## убедитесь, что установлены корректные права доступа на файлы
sudo service clickhouse-server restart
clickhouse-client --query "SELECT COUNT(*) FROM datasets.visits_v1"
```

Получение таблиц из сжатых tsv-файлов

Скачивание и импортирование hits из сжатого tsv-файла

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

```
clickhouse-client --query "OPTIMIZE TABLE datasets.hits_v1 FINAL"
clickhouse-client --query "SELECT COUNT(*) FROM datasets.hits_v1"
```

Скачивание и импортирование visits из сжатого tsv-файла

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

Запросы

Примеры запросов к этим таблицам (они называются `test.hits` и `test.visits`) можно найти среди stateful тестов и в некоторых performance тестах ClickHouse.

Интерфейсы

ClickHouse предоставляет два сетевых интерфейса (оба могут быть дополнительно обернуты в TLS для дополнительной безопасности):

- HTTP, который задокументирован и прост для использования напрямую;

- Native TCP, который имеет меньше накладных расходов.

В большинстве случаев рекомендуется использовать подходящий инструмент или библиотеку, а не напрямую взаимодействовать с ClickHouse по сути. Официально поддерживаемые Яндексом:

- Консольный клиент;
- JDBC-драйвер;
- ODBC-драйвер.

Существует также широкий спектр сторонних библиотек для работы с ClickHouse:

- Клиентские библиотеки;
- Библиотеки для интеграции;
- Визуальные интерфейсы.

Клиент командной строки

Для работы из командной строки вы можете использовать `clickhouse-client`:

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

:)
```

Клиент поддерживает параметры командной строки и конфигурационные файлы. Подробнее читайте в разделе "Конфигурирование".

Использование

Клиент может быть использован в интерактивном и неинтерактивном (batch) режиме.
Чтобы использовать batch режим, укажите параметр query, или отправьте данные в stdin (проверяется, что stdin - не терминал), или и то, и другое.
Аналогично HTTP интерфейсу, при использовании одновременно параметра query и отправке данных в stdin, запрос составляется из конкатенации параметра query, перевода строки, и данных в stdin. Это удобно для больших INSERT запросов.

Примеры использования клиента для вставки данных:

```
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";
```

В batch режиме в качестве формата данных по умолчанию используется формат TabSeparated. Формат может быть указан в секции FORMAT запроса.

По умолчанию, в batch режиме вы можете выполнить только один запрос. Чтобы выполнить несколько запросов из "скрипта", используйте параметр --multiquery. Это работает для всех запросов кроме INSERT. Результаты запросов выводятся подряд без дополнительных разделителей.
Также, при необходимости выполнить много запросов, вы можете запускать clickhouse-client на каждый запрос. Заметим, что запуск программы clickhouse-client может занимать десятки миллисекунд.

В интерактивном режиме, вы получите командную строку, в которую можно вводить запросы.

Если не указано multiline (по умолчанию):
Чтобы выполнить запрос, нажмите Enter. Точка с запятой на конце запроса не обязательна. Чтобы ввести запрос, состоящий из нескольких строк, перед переводом строки, введите символ обратного слеша: `\` - тогда после нажатия Enter, вам предложат ввести следующую строку запроса.

Если указано multiline (многострочный режим):
Чтобы выполнить запрос, завершите его точкой с запятой и нажмите Enter. Если в конце введённой строки не было точки с запятой, то вам предложат ввести следующую строчку запроса.

Исполняется только один запрос, поэтому всё, что введено после точки с запятой, игнорируется.

Вместо или после точки с запятой может быть указано `\G`. Это обозначает использование формата Vertical. В этом формате каждое значение выводится на отдельной строке, что удобно для широких таблиц. Столь необычная функциональность добавлена для совместимости с MySQL CLI.

Командная строка сделана на основе readline (и history) (или libedit, или без какой-либо библиотеки, в зависимости от сборки) - то есть, в ней работают привычные сочетания клавиш, а также присутствует история.
История пишется в `~/.clickhouse-client-history`.

По умолчанию, в качестве формата, используется формат PrettyCompact (красивые таблички). Вы можете изменить формат с помощью секции FORMAT запроса, или с помощью указания `\G` на конце запроса, с помощью аргумента командной строки `--format` или `--vertical`, или с помощью конфигурационного файла клиента.

Чтобы выйти из клиента, нажмите Ctrl+D (или Ctrl+C), или наберите вместо запроса одно из:
"exit", "quit", "logout", "учше", "йгше", "дщпщге", "exit;", "quit;", "logout;", "учшеж", "йгшеж", "дщпщгеж", "q", "й", "q", "Q", ":q", "й", "Й", "Жй"

При выполнении запроса, клиент показывает:

1. Прогресс выполнение запроса, который обновляется не чаще, чем 10 раз в секунду (по умолчанию). При быстрых запросах, прогресс может не успеть отобразиться.
2. Отформатированный запрос после его парсинга - для отладки.
3. Результат в заданном формате.
4. Количество строк результата, прошедшее время, а также среднюю скорость выполнения запроса.

Вы можете прервать длинный запрос, нажав Ctrl+C. При этом вам всё равно придётся чуть-чуть подождать, пока сервер остановит запрос. На некоторых стадиях выполнения, запрос невозможно прервать. Если вы не дождётесь и нажмёте Ctrl+C второй раз, то клиент будет завершён.

Клиент командной строки позволяет передать внешние данные (внешние временные таблицы) для использования запроса. Подробнее смотрите раздел "Внешние данные для обработки запроса"

Конфигурирование

В `clickhouse-client` можно передавать различные параметры (все параметры имеют значения по умолчанию) с помощью:

- Командной строки.

Параметры командной строки переопределяют значения по умолчанию и параметры конфигурационных файлов.

- Конфигурационных файлов.

Параметры в конфигурационных файлах переопределяют значения по умолчанию.

Параметры командной строки

- `--host, -h` - имя сервера, по умолчанию - localhost. Вы можете использовать как имя, так и IPv4 или IPv6 адрес.

IPv6 адрес.
- `--port` - порт, к которому соединяться, по умолчанию - 9000. Замечу, что для HTTP и родного интерфейса используются разные порты.
- `--user, -u` - имя пользователя, по умолчанию - default.
- `--password` - пароль, по умолчанию - пустая строка.
- `--query, -q` - запрос для выполнения, при использовании в неинтерактивном режиме.
- `--database, -d` - выбрать текущую БД, по умолчанию - текущая БД из настроек сервера (по умолчанию - БД default).
- `--multiline, -m` - если указано - разрешить многострочные запросы, не отправлять запрос по нажатию Enter.
- `--multiquery, -n` - если указано - разрешить выполнять несколько запросов, разделённых точкой с запятой. Работает только в неинтерактивном режиме.
- `--format, -f` - использовать указанный формат по умолчанию для вывода результата.
- `--vertical, -E` - если указано, использовать формат Vertical по умолчанию для вывода результата. То же самое, что --format=Vertical. В этом формате каждое значение выводится на отдельной строке, что удобно для отображения широких таблиц.
- `--time, -t` - если указано, в неинтерактивном режиме вывести время выполнения запроса в stderr.
- `--stacktrace` - если указано, в случае исключения, выводить также его стек трейс.
- `--config-file` - имя конфигурационного файла.
- `--secure` - если указано, будет использован безопасный канал.

Конфигурационные файлы

`clickhouse-client` использует первый существующий файл из:

- Определенного параметром `--config-file`.
- `./clickhouse-client.xml`
- `~/.clickhouse-client/config.xml`
- `/etc/clickhouse-client/config.xml`

Пример конфигурационного файла:

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

Родной интерфейс (TCP)

Нативный протокол используется в клиенте командной строки, для взаимодействия между серверами во время обработки распределенных запросов, а также в других программах на C++. К сожалению, у родного протокола ClickHouse пока нет формальной спецификации, но в нем можно разобраться с использованием исходного кода ClickHouse (начиная с примерно этого места) и/или путем перехвата и анализа TCP трафика.

HTTP-интерфейс

HTTP интерфейс позволяет использовать ClickHouse на любой платформе, из любого языка программирования. У нас он используется для работы из Java и Perl, а также из shell-скриптов. В других отделах, HTTP интерфейс используется из Perl, Python и Go. HTTP интерфейс более ограничен по сравнению с родным интерфейсом, но является более совместимым.

По умолчанию, clickhouse-server слушает HTTP на порту 8123 (это можно изменить в конфиге).
Если запросить GET / без параметров, то вернётся строка "Ok." (с переводом строки на конце). Это может быть использовано в скриптах проверки живости.

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

Запрос отправляется в виде параметра URL query. Или POST-ом. Или начало запроса в параметре query, а продолжение POST-ом (зачем это нужно, будет объяснено ниже). Размер URL ограничен 16KB, это следует учитывать при отправке больших запросов.

В случае успеха, вам вернётся код ответа 200 и результат обработки запроса в теле ответа.
В случае ошибки, вам вернётся код ответа 500 и текст с описанием ошибки в теле ответа.

При использовании метода GET, выставляется настройка readonly. То есть, для запросов, модифицирующие данные, можно использовать только метод POST. Сам запрос при этом можно отправлять как в теле POST-а, так и в параметре URL.

Примеры:

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

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

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

1
```

Как видно, curl немного неудобен тем, что надо URL-эскейпить пробелы.
Хотя wget сам всё эскейпит, но его не рекомендуется использовать, так как он плохо работает по HTTP 1.1 при использовании keep-alive и Transfer-Encoding: chunked.

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

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

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

Если часть запроса отправляется в параметре, а часть POST-ом, то между этими двумя кусками данных ставится перевод строки.
Пример (так работать не будет):

```
$ 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
```

По умолчанию, данные возвращаются в формате TabSeparated (подробнее смотри раздел "Форматы").
Можно попросить любой другой формат - с помощью секции FORMAT запроса.

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

```
└──┘
```

Возможность передавать данные POST-ом нужна для INSERT-запросов. В этом случае вы можете написать начало запроса в параметре URL, а вставляемые данные передать POST-ом. Вставляемыми данными может быть, например, tab-separated дамп, полученный из MySQL. Таким образом, запрос INSERT заменяет LOAD DATA LOCAL INFILE из MySQL.

Примеры:
Создаём таблицу:

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

Используем привычный запрос INSERT для вставки данных:

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

Данные можно отправить отдельно от запроса:

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

Можно указать любой формат для данных. Формат Values - то же, что используется при записи INSERT INTO t VALUES:

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

Можно вставить данные из tab-separated дампа, указав соответствующий формат:

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

Прочитаем содержимое таблицы. Данные выводятся в произвольном порядке из-за параллельной обработки запроса:

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

Удаляем таблицу.

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

Для запросов, которые не возвращают таблицу с данными, в случае успеха, выдаётся пустое тело ответа.

Вы можете использовать внутренний формат сжатия Clickhouse при передаче данных. Формат сжатых данных нестандартный, и вам придётся использовать для работы с ним специальную программу `clickhouse-compressor` (устанавливается вместе с пакетом `clickhouse-client`). Для повышения эффективности вставки данных можно отключить проверку контрольной суммы на стороне сервера с помощью настройкиhttp_native_compression_disable_checksumming_on_decompress.

Если вы указали `compress=1` в URL, то сервер сжимает данные, которые он отправляет.
Если вы указали `decompress=1` в URL, сервер распаковывает те данные, которые вы передаёте методом `POST`.

Также можно использовать стандартное HTTP сжатие с помощью `gzip`. Чтобы отправить запрос `POST`, сжатый с помощью `gzip`, добавьте к запросу заголовок `Content-Encoding: gzip`.
Чтобы ClickHouse сжимал ответ на запрос с помощью `gzip`, необходимо добавить `Accept-Encoding: gzip` к заголовкам запроса, и включить настройку ClickHouse enable_http_compression. Вы можете настроить уровень сжатия данных с помощью настройки http_zlib_compression_level.

Это может быть использовано для уменьшения трафика по сети при передаче большого количества данных, а также для создания сразу сжатых дампов.

Примеры отправки данных со сжатием:

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

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

Примечание

Некоторые HTTP-клиенты могут по умолчанию распаковывать данные (`gzip` и `deflate`) с сервера в фоновом режиме и вы можете получить распакованные данные, даже если правильно используете настройки сжатия.

В параметре URL database может быть указана БД по умолчанию.

```
$ 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
```

По умолчанию используется БД, которая прописана в настройках сервера, как БД по умолчанию. По умолчанию, это - БД default. Также вы всегда можете указать БД через точку перед именем таблицы.

Имя пользователя и пароль могут быть указаны в одном из двух вариантов:

1. С использованием HTTP Basic Authentification. Пример:

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

2. В параметрах URL user и password. Пример:

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

Если имя пользователя не указано, то используется `default`. Если пароль не указан, то используется пустой пароль.
Также в параметрах URL вы можете указать любые настройки, которые будут использованы для обработки одного запроса, или целые профили настроек. Пример:
http://localhost:8123/?profile=web&max_rows_to_read=1000000000&query=SELECT+1

Подробнее смотрите в разделе Настройки.

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

Об остальных параметрах смотри раздел "SET".

Аналогично можно использовать ClickHouse-сессии в HTTP-протоколе. Для этого необходимо добавить к запросу GET параметр `session_id`. В качестве идентификатора сессии можно использовать произвольную строку. По умолчанию через 60 секунд бездействия сессия будет прервана. Можно изменить этот таймаут, изменяя настройку `default_session_timeout` в конфигурации сервера, или добавив к запросу GET параметр `session_timeout`. Статус сессии можно проверить с помощью параметра `session_check=1`. В рамках одной сессии одновременно может исполняться только один запрос.

Имеется возможность получать информацию о прогрессе выполнения запроса в залоголвках X-ClickHouse-Progress. Для этого нужно включить настройку send_progress_in_http_headers.

Запущенные запросы не останавливаются автоматически при разрыве HTTP соединения. Парсинг и форматирование данных производится на стороне сервера и использование сети может быть неэффективным.
Может быть передан необязательный параметр query_id - идентификатор запроса, произвольная строка. Подробнее смотрите раздел "Настройки, replace_running_query".

Может быть передан необязательный параметр quota_key - ключ квоты, произвольная строка. Подробнее смотрите раздел "Квоты".

HTTP интерфейс позволяет передать внешние данные (внешние временные таблицы) для использования запроса. Подробнее смотрите раздел "Внешние данные для обработки запроса"

Буферизация ответа

Существует возможность включить буферизацию ответа на стороне сервера. Для этого предусмотрены параметры URL `buffer_size` и `wait_end_of_query`.

`buffer_size` определяет количество байт результата которые будут буферизованы в памяти сервера. Если тело результата больше этого порога, то буфер будет переписан в HTTP канал, а оставшиеся данные будут отправляться в HTTP-канал напрямую.

Чтобы гарантировать буферизацию всего ответа необходимо выставить `wait_end_of_query=1`. В этом случае данные, не поместившиеся в памяти, будут буферизованы во временном файле сервера.

Пример:

```
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'
```

Буферизация позволяет избежать ситуации когда код ответа и HTTP-заголовки были отправлены клиенту, после чего возникла ошибка выполнения запроса. В такой ситуации сообщение об ошибке записывается в конце тела ответа, и на стороне клиента ошибка может быть обнаружена только на этапе парсинга.

Форматы входных и выходных данных

ClickHouse может принимать (`INSERT`) и отдавать (`SELECT`) данные в различных форматах.

Поддерживаемые форматы и возможность использовать их в запросах `INSERT` и `SELECT` перечислены в таблице ниже.

Формат	INSERT	SELECT
TabSeparated	✔	✔
TabSeparatedRaw	✗	✔
TabSeparatedWithNames	✔	✔
TabSeparatedWithNamesAndTypes	✔	✔
CSV	✔	✔
CSVWithNames	✔	✔
Values	✔	✔
Vertical	✗	✔
JSON	✗	✔
JSONCompact	✗	✔
JSONEachRow	✔	✔
TSKV	✔	✔
Pretty	✗	✔
PrettyCompact	✗	✔
PrettyCompactMonoBlock	✗	✔
PrettyNoEscapes	✗	✔
PrettySpace	✗	✔
Protobuf	✔	✔
RowBinary	✔	✔

Формат	INSERT	SELECT
Native	✔	✔
Null	✗	✔
XML	✗	✔
CapnProto	✔	✗

TabSeparated

В TabSeparated формате данные пишутся по строкам. Каждая строчка содержит значения, разделённые табами. После каждого значения идёт таб, кроме последнего значения в строке, после которого идёт перевод строки. Везде подразумеваются исключительно unix-переводы строк. Последняя строка также обязана содержать перевод строки на конце. Значения пишутся в текстовом виде, без обрамляющих кавычек, с экранированием служебных символов.

Этот формат также доступен под именем `TSV`.

Формат `TabSeparated` удобен для обработки данных произвольными программами и скриптами. Он используется по умолчанию в HTTP-интерфейсе, а также в batch-режиме клиента командной строки. Также формат позволяет переносить данные между разными СУБД. Например, вы можете получить дамп из MySQL и загрузить его в ClickHouse, или наоборот.

Формат `TabSeparated` поддерживает вывод тотальных значений (при использовании WITH TOTALS) и экстремальных значений (при настройке extremes выставленной в 1). В этих случаях, после основных данных выводятся тотальные значения, и экстремальные значения. Основной результат, тотальные значения и экстремальные значения, отделяются друг от друга пустой строкой. Пример:

```
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
```

Форматирование данных

Целые числа пишутся в десятичной форме. Числа могут содержать лишний символ "+" в начале (игнорируется при парсинге, а при форматировании не пишется). Неотрицательные числа не могут содержать знак отрицания. При чтении допустим парсинг пустой строки, как числа ноль, или (для знаковых типов) строки, состоящей из одного минуса, как числа ноль. Числа, не помещающиеся в соответствующий тип данных, могут парсится, как некоторое другое число, без сообщения об ошибке.

Числа с плавающей запятой пишутся в десятичной форме. При этом, десятичный разделитель - точка. Поддерживается экспоненциальная запись, а также inf, +inf, -inf, nan. Запись числа с плавающей запятой может начинаться или заканчиваться на десятичную точку.
При форматировании возможна потеря точности чисел с плавающей запятой.
При парсинге, допустимо чтение не обязательно наиболее близкого к десятичной записи машинно-представимого числа.

представляется числа.

Даты выводятся в формате YYYY-MM-DD, парсятся в том же формате, но с любыми символами в качестве разделителей.
Даты-с-временем выводятся в формате YYYY-MM-DD hh:mm:ss, парсятся в том же формате, но с любыми символами в качестве разделителей.
Всё это происходит в системном часовом поясе на момент старта клиента (если клиент занимается форматированием данных) или сервера. Для дат-с-временем не указывается, действует ли daylight saving time. То есть, если в дампе есть времена во время перевода стрелок назад, то дамп не соответствует данным однозначно, и при парсинге будет выбрано какое-либо из двух времён.
При парсинге, некорректные даты и даты-с-временем могут парситься с естественным переполнением или как нулевые даты/даты-с-временем без сообщения об ошибке.

В качестве исключения, поддерживается также парсинг даты-с-временем в формате unix timestamp, если он состоит ровно из 10 десятичных цифр. Результат не зависит от часового пояса. Различение форматов YYYY-MM-DD hh:mm:ss и NNNNNNNNNN делается автоматически.

Строки выводятся с экранированием спецсимволов с помощью обратного слеша. При выводе, используются следующие escape-последовательности: `\b`, `\f`, `\r`, `\n`, `\t`, `\0`, `\'`, `\\`. Парсер также поддерживает последовательности `\a`, `\v`, и `\xHH` (последовательности hex escape) и любые последовательности вида `\c`, где `c` — любой символ (такие последовательности преобразуются в `c`). Таким образом, при чтении поддерживаются форматы, где перевод строки может быть записан как `\n` и как `\` и перевод строки. Например, строка `Hello world`, где между словами вместо пробела стоит перевод строки, может быть считана в любом из следующих вариантов:

```
Hello\nworld

Hello\
world
```

Второй вариант поддерживается, так как его использует MySQL при записи tab-separated дампа.

Минимальный набор символов, которых вам необходимо экранировать при передаче в TabSeparated формате: таб, перевод строки (LF) и обратный слеш.

Экранируется лишь небольшой набор символов. Вы можете легко наткнуться на строковое значение, которое испортит ваш терминал при выводе в него.

Массивы форматируются в виде списка значений через запятую в квадратных скобках. Элементы массива - числа форматируются как обычно, а даты, даты-с-временем и строки - в одинарных кавычках с такими же правилами экранирования, как указано выше.

NULL форматируется как `\N`.

TabSeparatedRaw

Отличается от формата `TabSeparated` тем, что строки выводятся без экранирования.
Этот формат подходит только для вывода результата выполнения запроса, но не для парсинга (приёма данных для вставки в таблицу).

Этот формат также доступен под именем `TSVRaw`.

TabSeparatedWithNames

Отличается от формата `TabSeparated` тем, что в первой строке пишутся имена столбцов.
При парсинге, первая строка полностью игнорируется. Вы не можете использовать имена столбцов, чтобы указать их порядок расположения, или чтобы проверить их корректность.
(Поддержка обработки заголовка при парсинге может быть добавлена в будущем.)

Этот формат также доступен под именем `TSVWithNames`.

TabSeparatedWithNamesAndTypes

Отличается от формата `TabSeparated` тем, что в первой строке пишутся имена столбцов, а во второй - типы столбцов.
При парсинге, первая и вторая строка полностью игнорируется.

Этот формат также доступен под именем `TSVWithNamesAndTypes`.

TSKV

Похож на TabSeparated, но выводит значения в формате name=value. Имена экранируются так же, как строки в формате TabSeparated и, дополнительно, экранируется также символ =.

```
SearchPhrase=   count()=8267016
SearchPhrase=интерьер ванной комнаты    count()=2166
SearchPhrase=яндекс     count()=1655
SearchPhrase=весна 2014 мода    count()=1549
SearchPhrase=фриформ фото       count()=1480
SearchPhrase=анджелина джоли    count()=1245
SearchPhrase=омск       count()=1112
SearchPhrase=фото собак разных пород    count()=1091
SearchPhrase=дизайн штор        count()=1064
SearchPhrase=баку       count()=1000
```

NULL форматируется как `\N`.

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

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

При большом количестве маленьких столбцов, этот формат существенно неэффективен, и обычно нет причин его использовать. Он реализован, так как используется в некоторых отделах Яндекса.

Поддерживается как вывод, так и парсинг данных в этом формате. При парсинге, поддерживается расположение значений разных столбцов в произвольном порядке. Допустимо отсутствие некоторых значений - тогда они воспринимаются как равные значениям по умолчанию. В этом случае в качестве значений по умолчанию используются нули и пустые строки. Сложные значения, которые могут быть заданы в таблице не поддерживаются как значения по умолчанию.

При парсинге, в качестве дополнительного поля, может присутствовать `tskv` без знака равенства и без значения. Это поле игнорируется.

CSV

Формат Comma Separated Values (RFC).

При форматировании, строки выводятся в двойных кавычках. Двойная кавычка внутри строки выводится как две двойные кавычки подряд. Других правил экранирования нет. Даты и даты-с-временем выводятся в двойных кавычках. Числа выводятся без кавычек. Значения разделяются символом-разделителем, по умолчанию — `,`. Символ-разделитель определяется настройкой format_csv_delimiter. Строки разделяются unix переводом строки (LF). Массивы сериализуются в CSV следующим образом: сначала массив сериализуется в строку, как в формате TabSeparated, а затем полученная строка выводится в CSV в двойных кавычках. Кортежи в формате CSV сериализуются, как отдельные столбцы (то есть, теряется их вложенность в кортеж).

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

&astПо умолчанию — `,`. См. настройку format_csv_delimiter для дополнительной информации

Кавычка умолчанию `"`). См. настройку format_csv_delimiter для дополнительной информации.

При парсинге, все значения могут парситься как в кавычках, так и без кавычек. Поддерживаются как двойные, так и одинарные кавычки. Строки также могут быть без кавычек. В этом случае они парсятся до символа-разделителя или перевода строки (CR или LF). В нарушение RFC, в случае парсинга строк не в кавычках, начальные и конечные пробелы и табы игнорируются. В качестве перевода строки, поддерживаются как Unix (LF), так и Windows (CR LF) и Mac OS Classic (LF CR) варианты.

`NULL` форматируется в виде `\N`.

Формат CSV поддерживает вывод totals и extremes аналогично `TabSeparated`.

CSVWithNames

Выводит также заголовок, аналогично `TabSeparatedWithNames`.

JSON

Выводит данные в формате JSON. Кроме таблицы с данными, также выводятся имена и типы столбцов, и некоторая дополнительная информация - общее количество выведенных строк, а также количество строк, которое могло бы быть выведено, если бы не было LIMIT-а. Пример:

```
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":
```

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

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

        "rows": 5,

        "rows_before_limit_at_least": 141137
}
```

JSON совместим с JavaScript. Для этого, дополнительно экранируются некоторые символы: символ прямого слеша `/` экранируется в виде `\/`; альтернативные переводы строк `U+2028`, `U+2029`, на которых ломаются некоторые браузеры, экранируются в виде `\uXXXX`-последовательностей. Экранируются ASCII control characters: backspace, form feed, line feed, carriage return, horizontal tab в виде `\b`, `\f`, `\n`, `\r`, `\t` соответственно, а также остальные байты из диапазона 00-1F с помощью `\uXXXX`-последовательностей. Невалидные UTF-8 последовательности заменяются на replacement character и, таким образом, выводимый текст будет состоять из валидных UTF-8 последовательностей. Числа типа UInt64 и Int64, для совместимости с JavaScript, по умолчанию выводятся в двойных кавычках. Чтобы они выводились без кавычек, можно установить конфигурационный параметр output_format_json_quote_64bit_integers равным 0.

`rows` - общее количество выведенных строчек.

`rows_before_limit_at_least` - не менее скольких строчек получилось бы, если бы не было LIMIT-а. Выводится только если запрос содержит LIMIT.
В случае, если запрос содержит GROUP BY, rows_before_limit_at_least - точное число строк, которое получилось бы, если бы не было LIMIT-а.

`totals` - тотальные значения (при использовании WITH TOTALS).

`extremes` - экстремальные значения (при настройке extremes, выставленной в 1).

Этот формат подходит только для вывода результата выполнения запроса, но не для парсинга (приёма данных для вставки в таблицу).

ClickHouse поддерживает NULL, который при выводе JSON будет отображен как `null`.

Смотрите также формат JSONEachRow .

JSONCompact

Отличается от JSON только тем, что строчки данных выводятся в массивах, а не в object-ах.

Пример:

```
{
        "meta":
        [
```

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

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

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

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

        "rows": 5,

        "rows_before_limit_at_least": 141137
}
```

Этот формат подходит только для вывода результата выполнения запроса, но не для парсинга (приёма данных для вставки в таблицу).
Смотрите также формат `JSONEachRow`.

JSONEachRow

При использовании этого формата, ClickHouse выводит каждую запись как объект JSON (каждый объект отдельной строкой), при этом данные в целом — невалидный JSON.

```
{"SearchPhrase":"дизайн штор","count()":"1064"}
{"SearchPhrase":"баку","count()":"1000"}
{"SearchPhrase":"","count":"8267016"}
```

При вставке данных необходимо каждую запись передавать как отдельный объект JSON.

Вставка данных

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

ClickHouse допускает:

- Любой порядок пар ключ-значение в объекте.
- Пропуск отдельных значений.

ClickHouse игнорирует пробелы между элементами и запятые после объектов. Вы можете передать все объекты одной строкой. Вам не нужно разделять их переносами строк.

Обработка пропущенных значений

Обработка пропущенных значений

ClickHouse заменяет опущенные значения значениями по умолчанию для соответствующих data types.

Если указано `DEFAULT expr`, то ClickHouse использует различные правила подстановки в зависимости от настройки input_format_defaults_for_omitted_fields.

Рассмотрим следующую таблицу:

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

- Если `input_format_defaults_for_omitted_fields = 0`, то значение по умолчанию для `x` и `a` равняется `0` (поскольку это значение по умолчанию для типа данных `UInt32`.)
- Если `input_format_defaults_for_omitted_fields = 1`, то значение по умолчанию для `x` равно `0`, а значение по умолчанию `a` равно `x * 2`.

Предупреждение

Если `insert_sample_with_metadata = 1`, то при обработке запросов ClickHouse потребляет больше вычислительных ресурсов, чем если `insert_sample_with_metadata = 0`.

Выборка данных

Рассмотрим в качестве примера таблицу `UserActivity`:

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │         5 │      146 │   -1 │
│ 4324182021466249494 │         6 │      185 │    1 │
└─────────────────────┴───────────┴──────────┴──────┘
```

Запрос `SELECT * FROM UserActivity FORMAT JSONEachRow` возвращает:

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

В отличие от формата JSON, для `JSONEachRow` ClickHouse не заменяет невалидные UTF-8 последовательности. Значения экранируются так же, как и для формата `JSON`.

Примечание

В строках может выводиться произвольный набор байт. Используйте формат `JSONEachRow`, если вы уверены, что данные в таблице могут быть представлены в формате JSON без потери информации.

Native

Самый эффективный формат. Данные пишутся и читаются блоками в бинарном виде. Для каждого блока пишется количество строк, количество столбцов, имена и типы столбцов, а затем кусочки столбцов этого блока, один за другим. То есть, этот формат является "столбцовым" - не преобразует столбцы в строки. Именно этот формат используется в родном интерфейсе - при межсерверном взаимодействии, при использовании клиента командной строки, при работе клиентов, написанных на C++.

Вы можете использовать этот формат для быстрой генерации дампов, которые могут быть прочитаны только СУБД ClickHouse. Вряд ли имеет смысл работать с этим форматом самостоятельно.

Null

Ничего не выводит. При этом, запрос обрабатывается, а при использовании клиента командной строки,

данные ещё и передаются на клиент. Используется для тестов, в том числе, тестов производительности. Очевидно, формат подходит только для вывода, но не для парсинга.

Pretty

Выводит данные в виде Unicode-art табличек, также используя ANSI-escape последовательности для установки цветов в терминале.
Рисуется полная сетка таблицы и, таким образом, каждая строчка занимает две строки в терминале.
Каждый блок результата выводится в виде отдельной таблицы. Это нужно, чтобы можно было выводить блоки без буферизации результата (буферизация потребовалась бы, чтобы заранее вычислить видимую ширину всех значений.)

NULL выводится как `NULL`.

```
SELECT * FROM t_null
```

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

В форматах `Pretty*` строки выводятся без экранирования. Ниже приведен пример для формата PrettyCompact:

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

```
┌─Escaping_test────────────────────────┐
│ String with 'quotes' and   character │
└──────────────────────────────────────┘
```

Для защиты от вываливания слишком большого количества данных в терминал, выводится только первые 10 000 строк. Если строк больше или равно 10 000, то будет написано "Showed first 10 000."
Этот формат подходит только для вывода результата выполнения запроса, но не для парсинга (приёма данных для вставки в таблицу).

Формат `Pretty` поддерживает вывод тотальных значений (при использовании WITH TOTALS) и экстремальных значений (при настройке extremes выставленной в 1). В этих случаях, после основных данных выводятся тотальные значения, и экстремальные значения, в отдельных табличках. Пример (показан для формата PrettyCompact):

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

```
┌──EventDate─┬───────c─┐
│ 2014-03-17 │ 1406958 │
│ 2014-03-18 │ 1383658 │
│ 2014-03-19 │ 1405797 │
│ 2014-03-20 │ 1353623 │
│ 2014-03-21 │ 1245779 │
│ 2014-03-22 │ 1031592 │
│ 2014-03-23 │ 1046491 │
└────────────┴─────────┘

Totals:
┌──EventDate─┬───────c─┐
│ 0000-00-00 │ 8873898 │
```

```
└────────────┴─────────┘

Extremes:
┌─EventDate─┬───────c─┐
│ 2014-03-17 │ 1031592 │
│ 2014-03-23 │ 1406958 │
└────────────┴─────────┘
```

PrettyCompact

Отличается от Pretty тем, что не рисуется сетка между строками - результат более компактный.
Этот формат используется по умолчанию в клиенте командной строки в интерактивном режиме.

PrettyCompactMonoBlock

Отличается от PrettyCompact тем, что строки (до 10 000 штук) буферизуются и затем выводятся в виде одной таблицы, а не по блокам.

PrettyNoEscapes

Отличается от Pretty тем, что не используются ANSI-escape последовательности. Это нужно для отображения этого формата в браузере, а также при использовании утилиты командной строки watch.

Пример:

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

Для отображения в браузере, вы можете использовать HTTP интерфейс.

PrettyCompactNoEscapes

Аналогично.

PrettySpaceNoEscapes

Аналогично.

PrettySpace

Отличается от PrettyCompact тем, что вместо сетки используется пустое пространство (пробелы).

RowBinary

Форматирует и парсит данные по строкам, в бинарном виде. Строки и значения уложены подряд, без разделителей.
Формат менее эффективен, чем формат Native, так как является строковым.

Числа представлены в little endian формате фиксированной длины. Для примера, UInt64 занимает 8 байт.
DateTime представлены как UInt32, содержащий unix timestamp в качестве значения.
Date представлены как UInt16, содержащий количество дней, прошедших с 1970-01-01 в качестве значения.
String представлены как длина в формате varint (unsigned LEB128), а затем байты строки.
FixedString представлены просто как последовательность байт.

Array представлены как длина в формате varint (unsigned LEB128), а затем элементы массива, подряд.

Для поддержки NULL перед каждым значением типа [Nullable](../data_types/nullable.md

Values

Выводит каждую строку в скобках. Строки разделены запятыми. После последней строки запятой нет.
Значения внутри скобок также разделены запятыми. Числа выводятся в десятичном виде без кавычек.
Массивы выводятся в квадратных скобках. Строки, даты, даты с временем выводятся в кавычках.

Массивы выводятся в квадратных скобках. Строки, даты, даты-с-временем выводятся в кавычках. Правила экранирования и особенности парсинга аналогичны формату TabSeparated. При форматировании, лишние пробелы не ставятся, а при парсинге - допустимы и пропускаются (за исключением пробелов внутри значений типа массив, которые недопустимы). NULL представляется как `NULL`.

Минимальный набор символов, которых вам необходимо экранировать при передаче в Values формате: одинарная кавычка и обратный слеш.

Именно этот формат используется в запросе `INSERT INTO t VALUES ...`, но вы также можете использовать его для форматирования результатов запросов.

Vertical

Выводит каждое значение на отдельной строке, с указанием имени столбца. Формат удобно использовать для вывода одной-нескольких строк, если каждая строка состоит из большого количества столбцов.

NULL выводится как `NULL`.

Пример:

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

```
Row 1:
──────
x: 1
y: NULL
```

В формате `Vertical` строки выводятся без экранирования. Например:

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

```
Row 1:
──────
test: string with 'quotes' and   with some special
 characters
```

Этот формат подходит только для вывода результата выполнения запроса, но не для парсинга (приёма данных для вставки в таблицу).

XML

Формат XML подходит только для вывода данных, не для парсинга. Пример:

```
<?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>интерьер ванной комнаты</SearchPhrase>
                        <field>2166</field>
                </row>
                <row>
                        <SearchPhrase>яндекс</SearchPhrase>
                        <field>1655</field>
                </row>
                <row>
                        <SearchPhrase>весна 2014 мода</SearchPhrase>
                        <field>1549</field>
                </row>
                <row>
                        <SearchPhrase>фриформ фото</SearchPhrase>
                        <field>1480</field>
                </row>
                <row>
                        <SearchPhrase>анджелина джоли</SearchPhrase>
                        <field>1245</field>
                </row>
                <row>
                        <SearchPhrase>омск</SearchPhrase>
                        <field>1112</field>
                </row>
                <row>
                        <SearchPhrase>фото собак разных пород</SearchPhrase>
                        <field>1091</field>
                </row>
                <row>
                        <SearchPhrase>дизайн штор</SearchPhrase>
                        <field>1064</field>
                </row>
                <row>
                        <SearchPhrase>баку</SearchPhrase>
                        <field>1000</field>
                </row>
        </data>
        <rows>10</rows>
        <rows_before_limit_at_least>141137</rows_before_limit_at_least>
</result>
```

Если имя столбца не имеет некоторый допустимый вид, то в качестве имени элемента используется просто field. В остальном, структура XML повторяет структуру в формате JSON.
Как и для формата JSON, невалидные UTF-8 последовательности заменяются на replacement character `<` и, таким образом, выводимый текст будет состоять из валидных UTF-8 последовательностей.

В строковых значениях, экранируются символы `<` и `&` как `&lt;` и `&amp;`.

Массивы выводятся как `<array><elem>Hello</elem><elem>World</elem>...</array>`,
а кортежи как `<tuple><elem>Hello</elem><elem>World</elem>...</tuple>`.

CapnProto

Cap'n Proto - формат бинарных сообщений, похож на Protocol Buffers и Thrift, но не похож на JSON или MessagePack.

Сообщения Cap'n Proto строго типизированы и не самоописывающиеся, т.е. нуждаются во внешнем

описании схемы. Схема применяется "на лету" и кешируется между запросами.

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

Где `schema.capnp` выглядит следующим образом:

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

Десериализация эффективна и обычно не повышает нагрузку на систему.

См. также схема формата.

Protobuf

Protobuf - формат Protocol Buffers.

Формат нуждается во внешнем описании схемы. Схема кэшируется между запросами.
ClickHouse поддерживает как синтаксис `proto2`, так и `proto3`; все типы полей (repeated/optional/required) поддерживаются.

Пример использования формата:

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

или

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

Где файл `schemafile.proto` может выглядеть так:

```
syntax = "proto3";

message MessageType {
  string name = 1;
  string surname = 2;
  uint32 birthDate = 3;
  repeated string phoneNumbers = 4;
};
```

Соответствие между столбцами таблицы и полями сообщения `Protocol Buffers` устанавливается по имени, при этом игнорируется регистр букв и символы `_` (подчеркивание) и `.` (точка) считаются одинаковыми.
Если типы столбцов не соответствуют точно типам полей сообщения `Protocol Buffers`, производится необходимая конвертация.

Вложенные сообщения поддерживаются, например, для поля `z` в таком сообщении

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

```
  };
  XType x;
};
```

ClickHouse попытается найти столбец с именем `x.y.z` (или `x_y_z`, или `X.y_Z` и т.п.).
Вложенные сообщения удобно использовать в качестве соответствия для вложенной структуры данных.

Значения по умолчанию, определенные в схеме, например,

```
syntax = "proto2";

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

не применяются; вместо них используются определенные в таблице значения по умолчанию.

ClickHouse пишет и читает сообщения `Protocol Buffers` в формате `length-delimited`. Это означает, что перед каждым сообщением пишется его длина
в формате varint. См. также как читать и записывать сообщения Protocol Buffers в формате length-delimited в различных языках программирования.

Схема формата

Имя файла со схемой записывается в настройке `format_schema`. При использовании форматов `Cap'n Proto` и `Protobuf` требуется указать схему.
Схема представляет собой имя файла и имя типа в этом файле, разделенные двоеточием, например `schemafile.proto:MessageType`.
Если файл имеет стандартное расширение для данного формата (например `.proto` для `Protobuf`),
то можно его не указывать и записывать схему так `schemafile:MessageType`.

Если для ввода/вывода данных используется клиент в интерактивном режиме, то при записи схемы можно использовать абсолютный путь или записывать путь
относительно текущей директории на клиенте. Если клиент используется в batch режиме, то в записи схемы допускается только относительный путь, из соображений безопасности.

Если для ввода/вывода данных используется HTTP-интерфейс, то файл со схемой должен располагаться на сервере в каталоге,
указанном в параметре format_schema_path конфигурации сервера.

JDBC-драйвер

- Официальный драйвер.
- Драйвер от сторонней огранизации ClickHouse-Native-JDBC.

ODBC-драйвер

- Официальный драйвер.

Клиентские библиотеки от сторонних разработчиков

Disclaimer

Яндекс не поддерживает перечисленные ниже библиотеки и не проводит тщательного тестирования для проверки их качества.

- Python:
  - infi.clickhouse_orm
  - clickhouse-driver
  - clickhouse-client

    - aiochclient
- PHP
    - phpClickHouse
    - clickhouse-php-client
    - clickhouse-client
    - PhpClickHouseClient
- Go
    - clickhouse
    - go-clickhouse
    - mailrugo-clickhouse
    - golang-clickhouse
- NodeJs
    - clickhouse (NodeJs)
    - node-clickhouse
- Perl
    - perl-DBD-ClickHouse
    - HTTP-ClickHouse
    - AnyEvent-ClickHouse
- Ruby
    - clickhouse (Ruby)
- R
    - clickhouse-r
    - RClickhouse
- Java
    - clickhouse-client-java
- Scala
    - clickhouse-scala-client
- Kotlin
    - AORM
- C#
    - ClickHouse.Ado
    - ClickHouse.Net
- C++
    - clickhouse-cpp
- Elixir
    - clickhousex
- Nim
    - nim-clickhouse

Библиотеки для интеграции от сторонних разработчиков

Disclaimer

Яндекс не занимается поддержкой перечисленных ниже инструментов и библиотек и не проводит тщательного тестирования для проверки их качества.

Инфраструктурные продукты

- Реляционные системы управления базами данных
    - MySQL
        - ProxySQL
        - clickhouse-mysql-data-reader
        - horgh-replicator
    - PostgreSQL
        - clickhousedb_fdw
        - infi.clickhouse_fdw (использует infi.clickhouse_orm)

        - pg2ch
    - MSSQL
        - ClickHouseMightrator
- Очереди ообщений
    - Kafka
        - clickhouse_sinker (использует Go client)
- Хранилища объектов
    - S3
        - clickhouse-backup
- Оркестрация контейнеров
    - Kubernetes
        - clickhouse-operator
- Системы управления конфигурацией
    - puppet
        - innogames/clickhouse
        - mfedotov/clickhouse
- Мониторинг
    - Graphite
        - graphouse
        - carbon-clickhouse
    - Grafana
        - clickhouse-grafana
    - Prometheus
        - clickhouse_exporter
        - PromHouse
        - clickhouse_exporter (использует Go client)
    - Nagios
        - check_clickhouse
    - Zabbix
        - clickhouse-zabbix-template
    - Sematext
        - clickhouse интеграция
- Логирование
    - rsyslog
        - omclickhouse
    - fluentd
        - loghouse (для Kubernetes)
    - logagent
        - logagent output-plugin-clickhouse
- Гео
    - MaxMind
        - clickhouse-maxmind-geoip

Экосистемы вокруг языков программирования

- Python
    - SQLAlchemy
        - sqlalchemy-clickhouse (использует infi.clickhouse_orm)
    - pandas
        - pandahouse
- R
    - dplyr
        - RClickhouse (использует clickhouse-cpp)
- Java
    - Hadoop
        - clickhouse-hdfs-loader (использует JDBC)
- Scala

    - Akka
        - clickhouse-scala-client
- C#
    - ADO.NET
        - ClickHouse.Ado
        - ClickHouse.Net
        - ClickHouse.Net.Migrations
- Elixir
    - Ecto
        - clickhouse_ecto

Визуальные интерфейсы от сторонних разработчиков

С открытым исходным кодом

Tabix

Веб-интерфейс для ClickHouse в проекте Tabix.

Основные возможности:

- Работает с ClickHouse напрямую из браузера, без необходимости установки дополнительного ПО;
- Редактор запросов с подсветкой синтаксиса;
- Автодополнение команд;
- Инструменты графического анализа выполнения запросов;
- Цветовые схемы на выбор.

Документация Tabix.

HouseOps

HouseOps — UI/IDE для OSX, Linux и Windows.

Основные возможности:

- Построение запросов с подсветкой синтаксиса;
- Просмотр ответа в табличном или JSON представлении;
- Экспортирование результатов запроса в формате CSV или JSON;
- Список процессов с описанием;
- Режим записи;
- Возможность остановки (`KILL`) запроса;
- Граф базы данных. Показывает все таблицы и их столбцы с дополнительной информацией;
- Быстрый просмотр размера столбца;
- Конфигурирование сервера.

Планируется разработка следующих возможностей:

- Управление базами;
- Управление пользователями;
- Анализ данных в режиме реального времени;
- Мониторинг кластера;
- Управление кластером;
- Мониторинг реплицированных и Kafka таблиц.

LightHouse

LightHouse — это легковесный веб-интерфейс для ClickHouse.

Основные возможности:

- Список таблиц с фильтрацией и метаданными;

- Список таблиц с фильтрацией и метаданными;
- Предварительный просмотр таблицы с фильтрацией и сортировкой;
- Выполнение запросов только для чтения.

DBeaver

DBeaver - универсальный desktop клиент баз данных с поддержкой ClickHouse.

Основные возможности:

- Построение запросов с подсветкой синтаксиса;
- Просмотр таблиц;
- Автодополнение команд;
- Полнотекстовый поиск.

clickhouse-cli

clickhouse-cli - это альтернативный клиент командной строки для ClickHouse, написанный на Python 3.

Основные возможности:
- Автодополнение;
- Подсветка синтаксиса для запросов и вывода данных;
- Поддержка постраничного просмотра для результирующих данных;
- Дополнительные PostgreSQL-подобные команды.

Коммерческие

DataGrip

DataGrip — это IDE для баз данных о JetBrains с выделенной поддержкой ClickHouse. Он также встроен в другие инструменты на основе IntelliJ: PyCharm, IntelliJ IDEA, GoLand, PhpStorm и другие.

Основные возможности:

- Очень быстрое дополнение кода.
- Подсветка синтаксиса для SQL диалекта ClickHouse.
- Поддержка функций, специфичных для ClickHouse, например вложенных столбцов, движков таблиц.
- Редактор данных.
- Рефакторинги.
- Поиск и навигация.

Прокси-серверы от сторонних разработчиков

chproxy

chproxy - это http-прокси и балансировщик нагрузки для базы данных ClickHouse.

Основные возможности:

- Индивидуальная маршрутизация и кэширование ответов;
- Гибкие ограничения;
- Автоматическое продление SSL сертификатов.

Реализован на Go.

KittenHouse

KittenHouse предназначен для использования в качестве локального прокси-сервера между ClickHouse и вашим сервером приложений в случае, если буферизовать данные INSERT на стороне приложения не представляется возможным или не удобно.

Основные возможности:

- Буферизация данных в памяти и на диске;

- Маршрутизация по таблицам;
- Балансировка нагрузки и проверка работоспособности.

Реализован на Go.

ClickHouse-Bulk

ClickHouse-Bulk - простой сборщик вставок ClickHouse.

Особенности:

- Группировка запросов и отправка по порогу или интервалу;
- Несколько удаленных серверов;
- Базовая аутентификация.

Реализован на Go.

Типы данных

ClickHouse может сохранять в ячейках таблиц данные различных типов.

Раздел содержит описания поддерживаемых типов данных и специфику их использования и/или реализации, если таковые имеются.

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

Целые числа фиксированной длины, без знака или со знаком.

Диапазоны Int

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

Диапазоны Uint

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

Float32, Float64

Числа с плавающей запятой.

Типы эквивалентны типам языка C:

- `Float32` - `float`;
- `Float64` - `double`.

Рекомендуется хранить данные в целочисленноми виде всегда, когда это возможно. Например, переводите в целочисленные значения числа с фиксированной точностью, такие как денежные суммы или времена загрузки страниц в миллисекундах.

Особенности использования чисел с плавающей запятой

- При вычислениях с числами с плавающей запятой возможна ошибка округления.

```
SELECT 1 - 0.9
```

```
minus(1, 0.9)
```

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

- Результат вычисления зависит от метода вычисления (типа процессора и архитектуры вычислительной системы).
- При вычислениях с плавающей запятой возможно появление таких категорий числа как бесконечность (`Inf`) и "не число" (`NaN`). Это необходимо учитывать при обработке результатов вычислений.
- При чтении чисел с плавающей запятой из строк, в качестве результата может быть получено не обязательно ближайшее машинно-представимое число.

NaN и Inf

В отличие от стандартного SQL, ClickHouse поддерживает следующие категории чисел с плавающей запятой:

- `Inf` - бесконечность.

```
SELECT 0.5 / 0
```

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

- `-Inf` - отрицательная бесконечность;

```
SELECT -0.5 / 0
```

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

- `NaN` - не число.

```
SELECT 0 / 0
```

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

Смотрите правила сортировки `NaN` в разделе Секция ORDER BY.

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

Знаковые дробные числа с сохранением точности операций сложения, умножения и вычитания. Для деления осуществляется отбрасывание (не округление) знаков, не попадающих в младший десятичный разряд.

Параметры

- P - precision. Значение из диапазона [1 : 38]. Определяет, сколько десятичных знаков (с учетом дробной части) может содержать число.

- S - scale. Значение из диапазона [0 : P]. Определяет, сколько десятичных знаков содержится в дробной части числа.

В зависимости от параметра P Decimal(P, S) является синонимом:
- P из [1 : 9] - для Decimal32(S)
- P из [10 : 18] - для Decimal64(S)
- P из [19 : 38] - для Decimal128(S)

Диапазоны Decimal

- 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))

Например, Decimal32(4) содержит числа от -99999.9999 до 99999.9999 с шагом 0.0001.

Внутреннее представление

Внутри данные представляются как знаковые целые числа, соответсвующей разрядности. Реальные диапазоны, хранящиеся в ячейках памяти несколько больше заявленных. Заявленные диапазоны Decimal проверяются только при вводе числа из строкового представления.
Поскольку современные CPU не поддерживают 128-битные числа, операции над Decimal128 эмулируются программно. Decimal128 работает в разы медленней чем Decimal32/Decimal64.

Операции и типы результата

Результат операции между двумя Decimal расширяется до большего типа (независимо от порядка аргументов).

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

Для размера дробной части (scale) результата действуют следующие правила:

- сложение, вычитание: S = max(S1, S2).
- умножение: S = S1 + S2.
- деление: S = S1.

При операциях между Decimal и целыми числами результатом является Decimal, аналогичный аргументу.

Операции между Decimal и Float32/64 не определены. Для осуществления таких операций нужно явно привести один из агруметнов функциями: toDecimal32, toDecimal64, toDecimal128, или toFloat32, toFloat64. Это сделано из двух соображений. Во-первых, результат операции будет с потерей точности. Во-вторых, преобразование типа - дорогая операция, из-за ее наличия пользовательский запрос может работать в несколько раз дольше.

Часть функций над Decimal возвращают Float64 (например, var, stddev). Для некоторых из них промежуточные операции проходят в Decimal.
Для таких функций результат над одинаковыми данными во Float64 и Decimal может отличаться, несмотря на одинаковый тип результата.

Проверка переполнений

При выполнении операций над типом Decimal могут происходить целочисленные переполнения. Лишняя дробная часть отбрасывается (не округляется). Лишняя целочисленная часть приводит к исключению.

```
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.
```

Проверка переполнения приводит к замедлению операций. При уверенности, что типа результата хватит для его записи проверку переполнения можно отключить настройкой decimal_check_overflow. В этом случае при переполнении вернется неверное значение:

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

Переполнения происходят не только на арифметических операциях, но и на операциях сравнения. Отключать проверку стоит только при полной уверенности в корректности результата:

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

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

Булевы значения

Отдельного типа для булевых значений нет. Для них используется тип UInt8, в котором используются только значения 0 и 1.

String

Строки произвольной длины. Длина не ограничена. Значение может содержать произвольный набор байт, включая нулевые байты.
Таким образом, тип String заменяет типы VARCHAR, BLOB, CLOB и т. п. из других СУБД.

Кодировки

В ClickHouse нет понятия кодировок. Строки могут содержать произвольный набор байт, который хранится и выводится, как есть.
Если вам нужно хранить тексты, рекомендуется использовать кодировку UTF-8. По крайней мере, если у вас терминал работает в кодировке UTF-8 (это рекомендуется), вы сможете читать и писать свои значения без каких-либо преобразований.
Также, некоторые функции по работе со строками, имеют отдельные варианты, которые работают при допущении, что строка содержит набор байт, представляющий текст в кодировке UTF-8.
Например, функция length вычисляет длину строки в байтах, а функция lengthUTF8 - длину строки в кодовых точках Unicode, при допущении, что значение в кодировке UTF-8.

FixedString

Строка фиксированной длины `N` байт (не символов, не кодовых точек).

Чтобы объявить столбец типа `FixedString`, используйте следующий синтаксис:

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

Где `N` — натуральное число.

Тип `FixedString` эффективен, когда данные имеют длину ровно `N` байт. Во всех остальных случаях использование FixedString может привести к снижению эффективности.

Примеры значений, которые можно эффективно хранить в столбцах типа `FixedString`:

- Двоичное представление IP-адреса (`FixedString(16)` для IPv6).
- Коды языков (ru_RU, en_US ...).
- Коды валют (USD, RUB ...).
- Двоичное представление хэшей (`FixedString(16)` для MD5, `FixedString(32)` для SHA256).

Для хранения значений UUID используйте тип данных UUID.

При вставке данных, ClickHouse:

- Дополняет строку нулевыми байтами, если строка содержит меньше байтов, чем `N`.
- Генерирует исключение `Too large value for FixedString(N)`, если строка содержит более `N` байт.

При выборе данных ClickHouse не обрезает нулевые байты в конце строки. Если вы используете секцию `WHERE`, то необходимо добавлять нулевые байты вручную, чтобы ClickHouse смог сопоставить выражение из фильтра значению `FixedString`. Следующий пример показывает, как использовать секцию `WHERE` с `FixedString`.

Рассмотрим следующую таблицу с единственным столбцом типа `FixedString(2)`:

```
┌─name─┐
│ b    │
└──────┘
```

Запрос `SELECT * FROM FixedStringTable WHERE a = 'b'` не возвращает необходимых данных. Необходимо дополнить шаблон фильтра нулевыми байтами.

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

┌─a─┐
│ b │
└───┘

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

Это поведение отличается от поведения MySQL для типа `CHAR`, где строки дополняются пробелами, а пробелы перед выводом вырезаются.

Обратите внимание, что длина значения `FixedString(N)` постоянна. Функция length возвращает `N` даже если значение `FixedString(N)` заполнено только нулевыми байтами, однако функция empty в этом же случае возвращает `1`.

UUID

Универсальный уникальный идентификатор (UUID) - это 16-байтовое число, используемое для идентификации записей. Подробнее про UUID читайте на Википедии.

Пример UUID значения представлен ниже:

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

Если при вставке новой записи значение для UUID-колонки не указано, UUID идентификатор будет заполнен нулями:

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

Как сгенерировать UUID

Для генерации UUID-значений предназначена функция generateUUIDv4.

Примеры использования

Ниже представлены примеры работы с UUID.

Пример 1

Этот пример демонстрирует, как создать таблицу с UUID-колонкой и добавить в нее сгенерированный UUID.

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

Пример 2

В этом примере, при добавлении записи в таблицу значение для UUID-колонки не задано. UUID будет заполнен нулями.

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

Ограничения

Тип данных UUID можно использовать только с функциями, которые поддерживаются типом данных String (например, min, max, и count).

Тип данных UUID не поддерживается арифметическими операциями (например, abs) или агрегатными функциями, такими как sum и avg.

функциями, такими как sum и avg.

Date

Дата. Хранится в двух байтах в виде (беззнакового) числа дней, прошедших от 1970-01-01. Позволяет хранить значения от чуть больше, чем начала unix-эпохи до верхнего порога, определяющегося константой на этапе компиляции (сейчас - до 2106 года, последний полностью поддерживаемый год - 2105).
Минимальное значение выводится как 0000-00-00.

Дата хранится без учёта часового пояса.

DateTime

Дата-с-временем. Хранится в 4 байтах, в виде (беззнакового) unix timestamp. Позволяет хранить значения в том же интервале, что и для типа Date. Минимальное значение выводится как 0000-00-00 00:00:00.
Время хранится с точностью до одной секунды (без учёта секунд координации).

Часовые пояса

Дата-с-временем преобразуется из текстового (разбитого на составляющие) в бинарный вид и обратно, с использованием системного часового пояса на момент старта клиента или сервера. В текстовом виде, теряется информация о том, был ли произведён перевод стрелок.

По умолчанию клиент переключается на часовой пояс сервера при подключении. Это поведение можно изменить, включив у клиента параметр командной строки `--use_client_time_zone`.

Поддерживаются только часовые пояса, для которых для всего диапазона времён, с которым вы будете работать, не существовало моментов времени, в которые время отличалось от UTC на нецелое число часов (без учёта секунд координации).

То есть, при работе с датой в виде текста (например, при сохранении текстовых дампов), следует иметь ввиду о проблемах с неоднозначностью во время перевода стрелок назад, и о проблемах с соответствием данных, при смене часового пояса.

Enum8, Enum16

Включает в себя типы `Enum8` и `Enum16`. `Enum` сохраняет конечный набор пар `'строка' = целое число`. Все операции с данными типа `Enum` ClickHouse выполняет как с числами, однако пользователь при этом работает со строковыми константами. Это более эффективно с точки зрения производительности, чем работа с типом данных `String`.

- `Enum8` описывается парами `'String' = Int8`.
- `Enum16` описывается парами `'String' = Int16`.

Примеры применения

Создадим таблицу со столбцом типа `Enum8('hello' = 1, 'world' = 2)`.

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

В столбец `x` можно сохранять только значения, перечисленные при определении типа, т.е. `'hello'` или `'world'`. Если попытаться сохранить другое значение, ClickHouse сгенерирует исключение.

```
:) 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)
```

При запросе данных из таблицы ClickHouse выдаст строковые значения из `Enum`.

```
SELECT * FROM t_enum

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

Если необходимо увидеть цифровые эквиваленты строкам, то необходимо привести тип.

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

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

Чтобы создать значение типа Enum в запросе, также необходима функция `CAST`.

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

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

Общие правила и особенности использования

Для каждого из значений прописывается число в диапазоне `-128 .. 127` для `Enum8` или в диапазоне `-32768 .. 32767` для `Enum16`. Все строки должны быть разными, числа - тоже. Разрешена пустая строка. При указании такого типа (в определении таблицы), числа могут идти не подряд и в произвольном порядке. При этом, порядок не имеет значения.

Ни строка, ни цифровое значение в `Enum` не могут быть NULL.

`Enum` может быть передан в тип Nullable. Таким образом, если создать таблицу запросом

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

, то в ней можно будет хранить не только 'hello' и 'world', но и NULL.

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

В оперативке столбец типа Enum представлен так же, как Int8 или Int16 соответствующими числовыми значениями.
При чтении в текстовом виде, парсит значение как строку и ищет соответствующую строку из множества значений Enum-а. Если не находит - кидается исключение.
При записи в текстовом виде, записывает значение как соответствующую строку. Если в данных столбца есть мусор - числа не из допустимого множества, то кидается исключение. При чтении и записи в бинарном виде, оно осуществляется так же, как для типов данных Int8, Int16.
Неявное значение по умолчанию - это значение с минимальным номером.

При ORDER BY, GROUP BY, IN, DISTINCT и т. п., Enum-ы ведут себя так же, как соответствующие числа. Например, при ORDER BY они сортируются по числовым значениям. Функции сравнения на равенство и сравнения на отношение порядка двух Enum-ов работают с Enum-ами так же, как с числами.

Сравнивать Enum с числом нельзя. Можно сравнивать Enum с константной строкой - при этом, для строки ищется соответствующее значение Enum-а; если не находится - кидается исключение.
Поддерживается оператор IN, где слева стоит Enum, а справа - множество строк. В этом случае, строки рассматриваются как значения соответствующего Enum-а.

Большинство операций с числами и со строками не имеет смысла и не работают для Enum-ов: например, к Enum-у нельзя прибавить число.
Для Enum-а естественным образом определяется функция toString, которая возвращает его строковое значение.

Также для Enum-а определяются функции toT, где T - числовой тип. При совпадении T с типом столбца Enum-а, преобразование работает бесплатно.
При ALTER, есть возможность бесплатно изменить тип Enum-а, если меняется только множество значений. При этом, можно добавлять новые значения; можно удалять старые значения (это безопасно только если они ни разу не использовались, так как это не проверяется). В качестве "защиты от дурака", нельзя менять числовые значения у имеющихся строк - в этом случае, кидается исключение.

При ALTER, есть возможность поменять Enum8 на Enum16 и обратно - так же, как можно поменять Int8 на Int16.

Array(T)

Массив из элементов типа T.

T может любым, в том числе, массивом. Таким образом поддержаны многомерные массивы.

Создание массива

Массив можно создать с помощью функции:

```
array(T)
```

Также можно использовать квадратные скобки

```
[]
```

Пример создания массива:

```
:) 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.
```

Особенности работы с типами данных

При создании массива "на лету" ClickHouse автоматически определяет тип аргументов как наиболее узкий тип данных, в котором можно хранить все перечисленные аргументы. Если среди аргументов есть NULL или аргумент типа Nullable, то тип элементов массива — Nullable.

Если ClickHouse не смог подобрать тип данных, то он сгенерирует исключение. Это произойдёт, например, при попытке создать массив одновременно со строками и числами `SELECT array(1, 'a')`.

Примеры автоматического определения типа данных:

```
:) 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.
```

Если попытаться создать массив из несовместимых типов данных, то ClickHouse выбросит исключение:

```
:) 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...)

Промежуточное состояние агрегатной функции. Чтобы его получить, используются агрегатные функции

с суффиксом `-State`. Чтобы в дальнейшем получить агрегированные данные необходимо использовать те же агрегатные функции с суффиксом `-Merge`.

`AggregateFunction` — параметрический тип данных.

Параметры

- Имя агрегатной функции.

  Для параметрических агрегатных функций указываются также их параметры.

- Типы аргументов агрегатной функции.

Пример

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

uniq, anyIf (any+If) и quantiles — агрегатные функции, поддержанные в ClickHouse.

Особенности использования

Вставка данных

Для вставки данных используйте `INSERT SELECT` с агрегатными `-State`-функциями.

Примеры функций

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

В отличие от соответствующих функций `uniq` и `quantiles`, `-State`-функциями возвращают не готовое значение, а состояние. То есть, значение типа `AggregateFunction`.

В запросах `SELECT` значения типа `AggregateFunction` выводятся во всех форматах, которые поддерживает ClickHouse, в виде implementation-specific бинарных данных. Если с помощью `SELECT` выполнить дамп данных, например, в формат `TabSeparated`, то потом этот дамп можно загрузить обратно с помощью запроса `INSERT`.

Выборка данных

При выборке данных из таблицы `AggregatingMergeTree`, используйте `GROUP BY` и те же агрегатные функции, что и при вставке данных, но с суффиксом `-Merge`.

Агрегатная функция с суффиксом `-Merge` берёт множество состояний, объединяет их, и возвращает результат полной агрегации данных.

Например, следующие два запроса возвращают один и тот же результат:

```
SELECT uniq(UserID) FROM table

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

Пример использования

Смотрите в описании движка AggregatingMergeTree.

Tuple(T1, T2, ...)

Кортеж из элементов любого типа. Элементы кортежа могут быть одного или разных типов.

Кортежи нельзя хранить в таблицах (кроме таблиц типа Memory). Они используется для временной группировки столбцов. Столбцы могут группироваться при использовании выражения IN в запросе, а также для указания нескольких формальных параметров лямбда-функций. Подробнее смотрите разделы Операторы IN, Функции высшего порядка.

Кортежи могут быть результатом запроса. В этом случае, в текстовых форматах кроме JSON, значения выводятся в круглых скобках через запятую. В форматах JSON, кортежи выводятся в виде массивов (в квадратных скобках).

Создание кортежа

Кортеж можно создать с помощью функции

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

Пример создания кортежа:

```
:) 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.
```

Особенности работы с типами данных

При создании кортежа "на лету" ClickHouse автоматически определяет тип каждого аргументов как минимальный из типов, который может сохранить значение аргумента. Если аргумент — NULL, то тип элемента кортежа — Nullable.

Пример автоматического определения типа данных:

```
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)

Позволяет работать как со значением типа `TypeName` так и с отсутствием этого значения (NULL) в одной и той же переменной, в том числе хранить `NULL` в таблицах вместе со значения типа `TypeName`. Например, в столбце типа `Nullable(Int8)` можно хранить значения типа `Int8`, а в тех строках, где значения нет, будет храниться `NULL`.

В качестве `TypeName` нельзя использовать составные типы данных Array и Tuple. Составные типы данных могут содержать значения типа `Nullable`, например `Array(Nullable(Int8))`.

Поле типа `Nullable` нельзя включать в индексы.

`NULL` — значение по умолчанию для типа `Nullable`, если в конфигурации сервера ClickHouse не указано иное.

Особенности хранения

Для хранения значения типа `Nullable` ClickHouse использует:

- Отдельный файл с масками `NULL` (далее маска).
- Непосредственно файл со значениями.

Маска определяет, что лежит в ячейке данных: `NULL` или значение.

В случае, когда маска указывает, что в ячейке хранится `NULL`, в файле значений хранится значение по умолчанию для типа данных. Т.е. если, например, поле имеет тип `Nullable(Int8)`, то ячейка будет хранить значение по умолчанию для `Int8`. Эта особенность увеличивает размер хранилища.

Info

Почти всегда использование `Nullable` снижает производительность, учитывайте это при проектировании своих баз.

Пример использования

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

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

Ok.

0 rows in set. Elapsed: 0.012 sec.

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

INSERT INTO t_null VALUES

Ok.

1 rows in set. Elapsed: 0.007 sec.

:) SELECT x + y from t_null

SELECT x + y
FROM t_null

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

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

Вложенные структуры данных

Вложенные структуры данных

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

Вложенная структура данных - это как будто вложенная таблица. Параметры вложенной структуры данных - имена и типы столбцов, указываются так же, как у запроса CREATE. Каждой строке таблицы может соответствовать произвольное количество строк вложенной структуры данных.

Пример:

```
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)
```

В этом примере объявлена вложенная структура данных `Goals`, содержащая данные о достижении целей. Каждой строке таблицы visits может соответствовать от нуля до произвольного количества достижений целей.

Поддерживается только один уровень вложенности. Столбцы вложенных структур, содержащие массивы, эквивалентны многомерным массивам, поэтому их поддержка ограничена (не поддерживается хранение таких столбцов в таблицах с движком семейства MergeTree).

В большинстве случаев, при работе с вложенной структурой данных, указываются отдельные её столбцы. Для этого, имена столбцов указываются через точку. Эти столбцы представляют собой массивы соответствующих типов. Все столбцы-массивы одной вложенной структуры данных имеют одинаковые длины.

Пример:

```
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'] |
└──────────────────────────────┴──────────────────────────────────────────────────────────────────────────────────────────────────────
──────────────┘
```

Проще всего понимать вложенную структуру данных, как набор из нескольких столбцов-массивов одинаковых длин.

Единственное место, где в запросе SELECT можно указать имя целой вложенной структуры данных, а не отдельных столбцов - секция ARRAY JOIN. Подробнее см. раздел "Секция ARRAY JOIN". Пример:

```
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 |
└─────────┴─────────────────────┘
```

Вы не можете сделать SELECT целой вложенной структуры данных. Можно лишь явно перечислить отдельные столбцы - её составляющие.

При запросе INSERT, вы должны передать все составляющие столбцы-массивы вложенной структуры данных по-отдельности (как если бы это были отдельные столбцы-массивы). При вставке проверяется, что они имеют одинаковые длины.

При запросе DESCRIBE, столбцы вложенной структуры данных перечисляются так же по отдельности.

Работоспособность запроса ALTER для элементов вложенных структур данных, является сильно ограниченной.

Служебные типы данных

Значения служебных типов данных не могут сохраняться в таблицу и выводиться в качестве результата, а возникают как промежуточный результат выполнения запроса.

Expression

Используется для представления лямбда-выражений в функциях высшего порядка.

Set

Set

Используется для представления правой части выражения IN.

Nothing

Этот тип данных предназначен только для того, чтобы представлять NULL, т.е. отсутствие значения.

Невозможно создать значение типа `Nothing`, поэтому он используется там, где значение не подразумевается. Например, `NULL` записывается как `Nullable(Nothing)` (Nullable — это тип данных, позволяющий хранить `NULL` в таблицах). Также тип `Nothing` используется для обозначения пустых массивов:

```
:) 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 constrains on stored values.

IPv4

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

Basic Usage

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

DESCRIBE TABLE hits;
```

```
┌─name─┬─type───┬─default_type─┬─default_expression─┬─comment─┬─codec_expression─┐
│ url  │ String │              │                    │         │                  │
│ from │ IPv4   │              │                    │         │                  │
└──────┴────────┴──────────────┴────────────────────┴─────────┴──────────────────┘
```

OR you can use IPv4 domain as a key:

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

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

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

SELECT * FROM hits;
```

```
┌─url────────────────────────────────┬──────────from─┐
│ https://clickhouse.yandex/docs/en/ │ 116.106.34.242 │
│ https://wikipedia.org              │ 116.253.40.133 │
│ https://clickhouse.yandex          │ 183.247.232.58 │
└────────────────────────────────────┴────────────────┘
```

Values are stored in compact binary form:

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

```
┌─toTypeName(from)─┬─hex(from)─┐
│ IPv4             │ B7F7E83A  │
└──────────────────┴───────────┘
```

Domain values are not implicitly convertible to types other than `UInt32`.
If you want to convert `IPv4` value to a string, you have to do that explicitly with `IPv4NumToString()` function:

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

```
┌─toTypeName(IPv4NumToString(from))─┬─s───────────────┐
│ String                            │ 183.247.232.58 │
└───────────────────────────────────┴─────────────────┘
```

Or cast to a `UInt32` value:

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

```
┌─toTypeName(CAST(from, 'UInt32'))─┬──────────i─┐
│ UInt32                           │ 3086477370 │
└──────────────────────────────────┴────────────┘
```

IPv6

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

Basic Usage

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

DESCRIBE TABLE hits;
```

```
┌─name─┬─type───┬─default_type─┬─default_expression─┬─comment─┬─codec_expression─┐
│ url  │ String │              │                    │         │                  │
│ from │ IPv6   │              │                    │         │                  │
└──────┴────────┴──────────────┴────────────────────┴─────────┴──────────────────┘
```

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

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

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

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

SELECT * FROM hits;
```

```
┌─url──────────────────────────────────┬─from──────────────────────────┐
│ https://clickhouse.yandex            │ 2001:44c8:129:2632:33:0:252:2 │
│ https://clickhouse.yandex/docs/en/ │ 2a02:e980:1e::1               │
│ https://wikipedia.org                │ 2a02:aa08:e000:3100::2        │
└──────────────────────────────────────┴───────────────────────────────┘
```

Values are stored in compact binary form:

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

```
┌─toTypeName(from)─┬─hex(from)────────────────────────┐
│ IPv6             │ 200144C80129263200330000002520002 │
└──────────────────┴──────────────────────────────────┘
```

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───────┐
```

```
┌─toTypeName(CAST(from, 'FixedString(16)'))─┬─┐
│ FixedString(16)                           │ ??? │
└───────────────────────────────────────────┴─────┘
```

Движки таблиц

Движок таблицы (тип таблицы) определяет:

- Как и где хранятся данные, куда их писать и откуда читать.
- Какие запросы поддерживаются и каким образом.
- Конкурентный доступ к данным.
- Использование индексов, если есть.
- Возможно ли многопоточное выполнение запроса.
- Параметры репликации данных.

При чтении, движок обязан лишь выдать запрошенные столбцы, но в некоторых случаях движок может частично обрабатывать данные при ответе на запрос.

Для большинства серьёзных задач, следует использовать движки семейства `MergeTree`.

MergeTree

Движок `MergeTree`, а также другие движки этого семейства (`*MergeTree`) — это наиболее функциональные движки таблиц ClickHouse.

Основная идея, заложенная в основу движков семейства `MergeTree` следующая. Когда у вас есть огромное количество данных, которые должны быть вставлены в таблицу, вы должны быстро записать их по частям, а затем объединить части по некоторым правилам в фоновом режиме. Этот метод намного эффективнее, чем постоянная перезапись данных в хранилище при вставке.

Основные возможности:

- Хранит данные, отсортированные по первичному ключу.

  Это позволяет создавать разреженный индекс небольшого объёма, который позволяет быстрее находить данные.

- Позволяет оперировать партициями, если задан ключ партиционирования.

  ClickHouse поддерживает отдельные операции с партициями, которые работают эффективнее, чем общие операции с этим же результатом над этими же данными. Также, ClickHouse автоматически отсекает данные по партициям там, где ключ партиционирования указан в запросе. Это также увеличивает эффективность выполнения запросов.

- Поддерживает репликацию данных.

  Для этого используется семейство таблиц `ReplicatedMergeTree`. Подробнее читайте в разделе Репликация данных.

- Поддерживает сэмплирование данных.

  При необходимости можно задать способ сэмплирования данных в таблице.

Info

Движок Merge не относится к семейству `*MergeTree`.

Создание таблицы

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1] [TTL expr1],
```

```
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2] [TTL expr2],
    ...
    INDEX index_name1 expr1 TYPE type1(...) GRANULARITY value1,
    INDEX index_name2 expr2 TYPE type2(...) GRANULARITY value2
) ENGINE = MergeTree()
[PARTITION BY expr]
[ORDER BY expr]
[PRIMARY KEY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

Описание параметров запроса смотрите в описании запроса.

Секции запроса

- `ENGINE` — Имя и параметры движка. `ENGINE = MergeTree()`. `MergeTree` не имеет параметров.
- `PARTITION BY` — ключ партиционирования.

  Для партиционирования по месяцам используйте выражение `toYYYYMM(date_column)`, где `date_column` — столбец с датой типа Date. В этом случае имена партиций имеют формат `"YYYYMM"`.
- `ORDER BY` — ключ сортировки.

  Кортеж столбцов или произвольных выражений. Пример: `ORDER BY (CounterID, EventDate)`.
- `PRIMARY KEY` — первичный ключ, если он отличается от ключа сортировки.

  По умолчанию первичный ключ совпадает с ключом сортировки (который задаётся секцией `ORDER BY`.) Поэтому в большинстве случаев секцию `PRIMARY KEY` отдельно указывать не нужно.
- `SAMPLE BY` — выражение для сэмплирования.

  Если используется выражение для сэмплирования, то первичный ключ должен содержать его. Пример:
  `SAMPLE BY intHash32(UserID) ORDER BY (CounterID, EventDate, intHash32(UserID))`.
- `TTL` - выражение для задания времени хранения строк.

  Оно должно зависеть от стобца типа `Date` или `DateTime` и в качестве результата вычислять столбец типа `Date` или `DateTime`. Пример:
  `TTL date + INTERVAL 1 DAY`

  Подробнее смотрите в TTL для столбцов и таблиц
- `SETTINGS` — дополнительные параметры, регулирующие поведение `MergeTree`:
  - `index_granularity` — гранулярность индекса. Число строк данных между «засечками» индекса. По умолчанию — 8192. Список всех доступных параметров можно посмотреть в MergeTreeSettings.h.
  - `min_merge_bytes_to_use_direct_io` — минимальный объем данных, необходимый для прямого (небуферизованного) чтения/записи (direct I/O) на диск. При слиянии частей данных ClickHouse вычисляет общий объем хранения всех данных, подлежащих слиянию. Если общий объем хранения всех данных для чтения превышает `min_bytes_to_use_direct_io` байт, тогда ClickHouse использует флаг `O_DIRECT` при чтении данных с диска. Если `min_merge_bytes_to_use_direct_io = 0`, тогда прямой ввод-вывод отключен. Значение по умолчанию: `10 * 1024 * 1024 * 1024` байт.
  - `merge_with_ttl_timeout` - Минимальное время в секундах для повторного выполнения слияний с TTL. По умолчанию - 86400 (1 день).

Пример задания секций

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

BY intHash32(UserID) SETTINGS index_granularity=8192

В примере мы устанавливаем партиционирование по месяцам.

Также мы задаем выражение для сэмплирования в виде хэша по идентификатору посетителя. Это позволяет псевдослучайным образом перемешать данные в таблице для каждого `CounterID` и `EventDate`. Если при выборке данных задать секцию SAMPLE, то ClickHouse вернёт равномерно-псевдослучайную выборку данных для подмножества посетителей.

`index_granularity` можно было не указывать, поскольку 8192 — это значение по умолчанию.

▼ Устаревший способ создания таблицы

Attention

Не используйте этот способ в новых проектах и по возможности переведите старые проекты на способ, описанный выше.

```
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)
```

Параметры MergeTree()

- `date-column` — имя столбца с типом Date. На основе этого столбца ClickHouse автоматически создаёт партиции по месяцам. Имена партиций имеют формат `"YYYYMM"`.
- `sampling_expression` — выражение для сэмплирования.
- `(primary, key)` — первичный ключ. Тип — [Tuple()](../../data_types/tuple.md- `index_granularity` — гранулярность индекса. Число строк данных между «засечками» индекса. Для большинства задач подходит значение 8192.

Пример

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

Движок `MergeTree` сконфигурирован таким же образом, как и в примере выше для основного способа конфигурирования движка.

Хранение данных

Таблица состоит из *кусков* данных (data parts), отсортированных по первичному ключу.

При вставке в таблицу создаются отдельные куски данных, каждый из которых лексикографически отсортирован по первичному ключу. Например, если первичный ключ — `(CounterID, Date)`, то данные в куске будут лежать в порядке `CounterID`, а для каждого `CounterID` в порядке `Date`.

Данные, относящиеся к разным партициям, разбиваются на разные куски. В фоновом режиме ClickHouse выполняет слияния (merge) кусков данных для более эффективного хранения. Куски, относящиеся к разным партициям не объединяются. Механизм слияния не гарантирует, что все строки с одинаковым первичным ключом окажутся в одном куске.

Для каждого куска данных ClickHouse создаёт индексный файл, который содержит значение первичного ключа для каждой индексной строки («засечка»). Номера строк индекса определяются как `n * index_granularity`. Максимальное значение `n` равно целой части деления общего числа строк на `index_granularity`. Для каждого столбца "засечки" также записываются для тех же строк индекса, что и первичный ключ. Эти "засечки" позволяют находить данные непосредственно в столбцах.

Вы можете использовать одну большую таблицу, постоянно добавляя в неё данные пачками, именно для этого предназначен движок `MergeTree`.

Первичные ключи и индексы в запросах

Рассмотрим первичный ключ — `(CounterID, Date)`. В этом случае сортировку и индекс можно проиллюстрировать следующим образом:

```
Whole data:     [-------------------------------------------------------------------------]
CounterID:      [aaaaaaaaaaaaaaaaaabbbbcdeeeeeeeeeeeeefgggggggghhhhhhhhhiiiiiiiiikllllllll]
Date:           [1111111222222233331233211111222222333211111112122222223111112223311122333]
Marks:           |    |    |    |    |    |    |    |    |    |    |
                a,1  a,2  a,3  b,3  e,2  e,3  g,1  h,2  i,1  i,3  l,3
Marks numbers:   0    1    2    3    4    5    6    7    8    9    10
```

Если в запросе к данным указать:

- `CounterID IN ('a', 'h')`, то сервер читает данные в диапазонах засечек `[0, 3)` и `[6, 8)`.
- `CounterID IN ('a', 'h') AND Date = 3`, то сервер читает данные в диапазонах засечек `[1, 3)` и `[7, 8)`.
- `Date = 3`, то сервер читает данные в диапазоне засечек `[1, 10]`.

Примеры выше показывают, что использование индекса всегда эффективнее, чем full scan.

Разреженный индекс допускает чтение лишних строк. При чтении одного диапазона первичного ключа, может быть прочитано до `index_granularity * 2` лишних строк в каждом блоке данных. В большинстве случаев ClickHouse не теряет производительности при `index_granularity = 8192`.

Разреженность индекса позволяет работать даже с очень большим количеством строк в таблицах, поскольку такой индекс всегда помещается в оперативную память компьютера.

ClickHouse не требует уникального первичного ключа. Можно вставить много строк с одинаковым первичным ключом.

Выбор первичного ключа

Количество столбцов в первичном ключе не ограничено явным образом. В зависимости от структуры данных в первичный ключ можно включать больше или меньше столбцов. Это может:

- Увеличить эффективность индекса.

  Пусть первичный ключ — `(a, b)`, тогда добавление ещё одного столбца `c` повысит эффективность, если выполнены условия:

  - Есть запросы с условием на столбец `c`.
  - Часто встречаются достаточно длинные (в несколько раз больше `index_granularity`) диапазоны данных с одинаковыми значениями `(a, b)`. Иначе говоря, когда добавление ещё одного столбца позволит пропускать достаточно длинные диапазоны данных.
- Улучшить сжатие данных.

  ClickHouse сортирует данные по первичному ключу, поэтому чем выше однородность, тем лучше сжатие.

- Обеспечить дополнительную логику при слиянии кусков данных в движках CollapsingMergeTree и SummingMergeTree.

  В этом случае имеет смысл указать отдельный *ключ сортировки*, отличающийся от первичного ключа.

Длинный первичный ключ будет негативно влиять на производительность вставки и потребление памяти, однако на производительность ClickHouse при запросах `SELECT` лишние столбцы в первичном ключе не влияют.

Первичный ключ, отличный от ключа сортировки

Существует возможность задать первичный ключ (выражение, значения которого будут записаны в индексный файл для
каждой засечки), отличный от ключа сортировки (выражение, по которому будут упорядочены строки в кусках
данных). Кортеж выражения первичного ключа при этом должен быть префиксом кортежа выражения ключа
сортировки.

Данная возможность особенно полезна при использовании движков SummingMergeTree и AggregatingMergeTree. В типичном сценарии использования этих движков таблица содержит столбцы двух типов: *измерения* (dimensions) и *меры* (measures). Типичные запросы агрегируют значения столбцов-мер с произвольной группировкой и фильтрацией по измерениям. Так как `SummingMergeTree` и `AggregatingMergeTree` производят фоновую агрегацию строк с одинаковым значением ключа сортировки, приходится
добавлять в него все столбцы-измерения. В результате выражение ключа содержит большой список столбцов,
который приходится постоянно расширять при добавлении новых измерений.

В этом сценарии имеет смысл оставить в первичном ключе всего несколько столбцов, которые обеспечат эффективную фильтрацию по индексу, а остальные столбцы-измерения добавить в выражение ключа сортировки.

ALTER ключа сортировки — лёгкая операция, так как при одновременном добавлении нового столбца в таблицу и ключ сортировки не нужно изменять данные кусков (они остаются упорядоченными и по новому выражению ключа).

Использование индексов и партиций в запросах

Для запросов `SELECT` ClickHouse анализирует возможность использования индекса. Индекс может использоваться, если в секции `WHERE/PREWHERE`, в качестве одного из элементов конъюнкции, или целиком, есть выражение, представляющее операции сравнения на равенства, неравенства, а также `IN` или `LIKE` с фиксированным префиксом, над столбцами или выражениями, входящими в первичный ключ или ключ партиционирования, либо над некоторыми частично монотонными функциями от этих столбцов, а также логические связки над такими выражениями.

Таким образом, обеспечивается возможность быстро выполнять запросы по одному или многим диапазонам первичного ключа. Например, в указанном примере будут быстро работать запросы для конкретного счётчика; для конкретного счётчика и диапазона дат; для конкретного счётчика и даты, для нескольких счётчиков и диапазона дат и т. п.

Рассмотрим движок сконфигурированный следующим образом:

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

В этом случае в запросах:

```
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 будет использовать индекс по первичному ключу для отсечения не подходящих данных, а также ключ партиционирования по месяцам для отсечения партиций, которые находятся в не подходящих диапазонах дат.

подходящих диапазонах дат.

Запросы выше показывают, что индекс используется даже для сложных выражений. Чтение из таблицы организовано так, что использование индекса не может быть медленнее, чем full scan.

В примере ниже индекс не может использоваться.

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

Чтобы проверить, сможет ли ClickHouse использовать индекс при выполнении запроса, используйте настройки force_index_by_date и force_primary_key.

Ключ партиционирования по месяцам обеспечивает чтение только тех блоков данных, которые содержат даты из нужного диапазона. При этом блок данных может содержать данные за многие даты (до целого месяца). В пределах одного блока данные упорядочены по первичному ключу, который может не содержать дату в качестве первого столбца. В связи с этим, при использовании запроса с указанием условия только на дату, но не на префикс первичного ключа, будет читаться данных больше, чем за одну дату.

Дополнительные индексы (Экспериментальная функциональность)

Для использования требуется установить настройку `allow_experimental_data_skipping_indices` в 1. (запустить `SET allow_experimental_data_skipping_indices = 1`).

Объявление индексов при определении столбцов в запросе `CREATE`.

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

Для таблиц семейства `*MergeTree` можно задать дополнительные индексы в секции столбцов.

Индексы агрегируют для заданного выражения некоторые данные, а потом при `SELECT` запросе используют для пропуска блоков данных (пропускаемый блок состоит из гранул данных в количестве равном гранулярности данного индекса), на которых секция `WHERE` не может быть выполнена, тем самым уменьшая объем данных читаемых с диска.

Пример

```
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()
...
```

Эти индексы смогут использоваться для оптимизации следующих запросов

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

Доступные индексы

- `minmax`
  Хранит минимум и максимум выражения (если выражение - `tuple`, то для каждого элемента `tuple`), используя их для пропуска блоков аналогично первичному ключу.

- `set(max_rows)`

- set(max_rows)
  Хранит уникальные значения выражения на блоке в количестве не более max_rows (если max_rows = 0, то ограничений нет), используя их для пропуска блоков, оценивая выполнимость WHERE выражения на хранимых данных.

Примеры

```
INDEX b (u64 * length(str), i32 + f64 * 100, date, str) TYPE minmax GRANULARITY 4
INDEX b (u64 * length(str), i32 + f64 * 100, date, str) TYPE set(100) GRANULARITY 4
```

Конкурентный доступ к данным

Для конкурентного доступа к таблице используется мультиверсионность. То есть, при одновременном чтении и обновлении таблицы, данные будут читаться из набора кусочков, актуального на момент запроса. Длинных блокировок нет. Вставки никак не мешают чтениям.

Чтения из таблицы автоматически распараллеливаются.

TTL для столбцов и таблиц

Данные с истекшим TTL удаляются во время слияний.

Если TTL указан для столбца, то когда он истекает, значение заменяется на значение по умолчанию. Если все значения столбца обнулены в куске, то данные этого столбца удаляются с диска в куске. Если TTL указан для таблицы, то когда он истекает, удаляется строка.

Когда истекает TTL для какого-нибудь значения или строки в куске, назначается внеочередное слияние. Чтобы контролировать частоту слияний с TTL, вы можете задать настройку merge_with_ttl_timeout. Если ее значение слишком мало, то будет происходить слишком много внеочередных слияний и мало обычных, вследствие чего может ухудшиться производительность.

Репликация данных

Репликация поддерживается только для таблиц семейства MergeTree:

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

Репликация работает на уровне отдельных таблиц, а не всего сервера. То есть, на сервере могут быть расположены одновременно реплицируемые и не реплицируемые таблицы.

Репликация не зависит от шардирования. На каждом шарде репликация работает независимо.

Реплицируются сжатые данные запросов INSERT, ALTER (см. подробности в описании запроса ALTER).

Запросы CREATE, DROP, ATTACH, DETACH и RENAME выполняются на одном сервере и не реплицируются:

- Запрос CREATE TABLE создаёт новую реплицируемую таблицу на том сервере, где его выполнили. Если таблица уже существует на других серверах, запрос добавляет новую реплику.
- DROP TABLE удаляет реплику, расположенную на том сервере, где выполняется запрос.
- Запрос RENAME переименовывает таблицу на одной реплик. Другими словами, реплицируемые таблицы на разных репликах могут называться по-разному.

Чтобы использовать репликацию, укажите в конфигурационном файле адреса ZooKeeper кластера. Пример:

```
<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>
```

Используйте ZooKeeper версии 3.4.5 или более новый.

Можно указать любой имеющийся у вас ZooKeeper-кластер - система будет использовать в нём одну директорию для своих данных (директория указывается при создании реплицируемой таблицы).

Если в конфигурационном файле не настроен ZooKeeper, то вы не сможете создать реплицируемые таблицы, а уже имеющиеся реплицируемые таблицы будут доступны в режиме только на чтение.

При запросах `SELECT`, ZooKeeper не используется, т.е. репликация не влияет на производительность `SELECT` и запросы работают так же быстро, как и для нереплицируемых таблиц. При запросах к распределенным реплицированным таблицам поведение ClickHouse регулируется настройками max_replica_delay_for_distributed_queries and fallback_to_stale_replicas_for_distributed_queries.

При каждом запросе `INSERT`, делается около десятка записей в ZooKeeper в рамках нескольких транзакций. (Чтобы быть более точным, это для каждого вставленного блока данных; запрос INSERT содержит один блок или один блок на `max_insert_block_size = 1048576` строк.) Это приводит к некоторому увеличению задержек при `INSERT`, по сравнению с нереплицируемыми таблицами. Но если придерживаться обычных рекомендаций - вставлять данные пачками не более одного `INSERT` в секунду, то это не составляет проблем. На всём кластере ClickHouse, использующим для координации один кластер ZooKeeper, может быть в совокупности несколько сотен `INSERT` в секунду. Пропускная способность при вставке данных (количество строчек в секунду) такая же высокая, как для нереплицируемых таблиц.

Для очень больших кластеров, можно использовать разные кластеры ZooKeeper для разных шардов. Впрочем, на кластере Яндекс.Метрики (примерно 300 серверов) такой необходимости не возникает.

Репликация асинхронная, мульти-мастер. Запросы `INSERT` и `ALTER` можно направлять на любой доступный сервер. Данные вставятся на сервер, где выполнен запрос, а затем скопируются на остальные серверы. В связи с асинхронностью, только что вставленные данные появляются на остальных репликах с небольшой задержкой. Если часть реплик недоступна, данные на них запишутся тогда, когда они станут доступны. Если реплика доступна, то задержка составляет столько времени, сколько требуется для передачи блока сжатых данных по сети.

По умолчанию, запрос INSERT ждёт подтверждения записи только от одной реплики. Если данные были успешно записаны только на одну реплику, и сервер с этой репликой перестал существовать, то записанные данные будут потеряны. Вы можете включить подтверждение записи от нескольких реплик, используя настройку `insert_quorum`.

Каждый блок данных записывается атомарно. Запрос INSERT разбивается на блоки данных размером до `max_insert_block_size = 1048576` строк. То есть, если в запросе `INSERT` менее 1048576 строк, то он делается атомарно.

Блоки данных дедуплицируются. При многократной записи одного и того же блока данных (блоков данных одинакового размера, содержащих одни и те же строчки в одном и том же порядке), блок будет записан только один раз. Это сделано для того, чтобы в случае сбоя в сети, когда клиентское

приложение не может понять, были ли данные записаны в БД, можно было просто повторить запрос `INSERT`. При этом не имеет значения, на какую реплику будут отправлены INSERT-ы с одинаковыми данными. Запрос `INSERT` идемпотентный. Параметры дедуплицирования регулируются настройками сервера merge_tree

При репликации, по сети передаются только исходные вставляемые данные. Дальнейшие преобразования данных (слияния) координируются и делаются на всех репликах одинаковым образом. За счёт этого минимизируется использование сети, и благодаря этому, репликация хорошо работает при расположении реплик в разных датацентрах. (Стоит заметить, что дублирование данных в разных датацентрах, по сути, является основной задачей репликации).

Количество реплик одних и тех же данных может быть произвольным. В Яндекс.Метрике в продакшене используется двухкратная репликация. На каждом сервере используется RAID-5 или RAID-6, в некоторых случаях RAID-10. Это является сравнительно надёжным и удобным для эксплуатации решением.

Система следит за синхронностью данных на репликах и умеет восстанавливаться после сбоя. Восстановление после сбоя автоматическое (в случае небольших различий в данных) или полуавтоматическое (когда данные отличаются слишком сильно, что может свидетельствовать об ошибке конфигурации).

Создание реплицируемых таблиц

В начало имени движка таблицы добавляется `Replicated`. Например, `ReplicatedMergeTree`.

Параметры Replicated*MergeTree

- `zoo_path` — путь к таблице в ZooKeeper.
- `replica_name` — имя реплики в ZooKeeper.

Пример:

```
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)
```

▼ Пример в устаревшем синтаксисе

```
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)
```

Как видно в примере, эти параметры могут содержать подстановки в фигурных скобках. Подставляемые значения достаются из конфигурационного файла, из секции `macros`. Пример:

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

Путь к таблице в ZooKeeper должен быть разным для каждой реплицируемой таблицы. В том числе, для таблиц на разных шардах, должны быть разные пути.
В данном случае, путь состоит из следующих частей:

`/clickhouse/tables/` — общий префикс. Рекомендуется использовать именно его.

`{layer}-{shard}` — идентификатор шарда. В данном примере он состоит из двух частей, так как на кластере Яндекс.Метрики используется двухуровневое шардирование. Для большинства задач, оставьте только подстановку {shard}, которая будет раскрываться в идентификатор шарда.

`table_name` - имя узла для таблицы в ZooKeeper. Разумно делать его таким же, как имя таблицы. Оно указывается явно, так как, в отличие от имени таблицы, оно не меняется после запроса RENAME.
Подсказка: можно также указать имя базы данных перед `table_name`, например `db_name.table_name`

Имя реплики — то, что идентифицирует разные реплики одной и той же таблицы. Можно использовать для него имя сервера, как показано в примере. Впрочем, достаточно, чтобы имя было уникально лишь в пределах каждого шарда.

Можно не использовать подстановки, а указать соответствующие параметры явно. Это может быть удобным для тестирования и при настройке маленьких кластеров. Однако в этом случае нельзя пользоваться распределенными DDL-запросами (`ON CLUSTER`).

При работе с большими кластерами мы рекомендуем использовать подстановки, они уменьшают вероятность ошибки.

Выполните запрос `CREATE TABLE` на каждой реплике. Запрос создаёт новую реплицируемую таблицу, или добавляет новую реплику к имеющимся.

Если вы добавляете новую реплику после того, как таблица на других репликах уже содержит некоторые данные, то после выполнения запроса, данные на новую реплику будут скачаны с других реплик. То есть, новая реплика синхронизирует себя с остальными.

Для удаления реплики, выполните запрос `DROP TABLE`. При этом, удаляется только одна реплика — расположенная на том сервере, где вы выполняете запрос.

Восстановление после сбоя

Если при старте сервера, недоступен ZooKeeper, реплицируемые таблицы переходят в режим только для чтения. Система будет пытаться периодически установить соединение с ZooKeeper.

Если при `INSERT` недоступен ZooKeeper, или происходит ошибка при взаимодействии с ним, будет выкинуто исключение.

При подключении к ZooKeeper, система проверяет соответствие между имеющимся в локальной файловой системе набором данных и ожидаемым набором данных (информация о котором хранится в ZooKeeper). Если имеются небольшие несоответствия, то система устраняет их, синхронизируя данные с реплик.

Обнаруженные битые куски данных (с файлами несоответствующего размера) или неизвестные куски (куски, записанные в файловую систему, но информация о которых не была записана в ZooKeeper) переносятся в поддиректорию detached (не удаляются). Недостающие куски скачиваются с реплик.

Стоит заметить, что ClickHouse не делает самостоятельно никаких деструктивных действий типа автоматического удаления большого количества данных.

При старте сервера (или создании новой сессии с ZooKeeper), проверяется только количество и размеры всех файлов. Если у файлов совпадают размеры, но изменены байты где-то посередине, то это обнаруживается не сразу, а только при попытке их прочитать при каком-либо запросе `SELECT`. Запрос кинет исключение о несоответствующей чексумме или размере сжатого блока. В этом случае, куски данных добавляются в очередь на проверку, и при необходимости, скачиваются с реплик.

Если обнаруживается, что локальный набор данных слишком сильно отличается от ожидаемого, то

срабатывает защитный механизм. Сервер сообщает об этом в лог и отказывается запускаться. Это сделано, так как такой случай может свидетельствовать об ошибке конфигурации - например, если реплика одного шарда была случайно сконфигурирована, как реплика другого шарда. Тем не менее, пороги защитного механизма поставлены довольно низкими, и такая ситуация может возникнуть и при обычном восстановлении после сбоя. В этом случае, восстановление делается полуавтоматически - "по кнопке".

Для запуска восстановления, создайте в ZooKeeper узел `/path_to_table/replica_name/flags/force_restore_data` с любым содержимым или выполните команду для восстановления всех реплицируемых таблиц:

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

Затем запустите сервер. При старте, сервер удалит эти флаги и запустит восстановление.

Восстановление в случае потери всех данных

Если на одном из серверов исчезли все данные и метаданные, восстановление делается следующим образом:

1. Установите на сервер ClickHouse. Корректно пропишите подстановки в конфигурационном файле, отвечающие за идентификатор шарда и реплики, если вы их используете.
2. Если у вас были нереплицируемые таблицы, которые должны быть вручную продублированы на серверах, скопируйте их данные (в директории `/var/lib/clickhouse/data/db_name/table_name/`) с реплики.
3. Скопируйте с реплики определения таблиц, находящиеся в `/var/lib/clickhouse/metadata/`. Если в определениях таблиц, идентификатор шарда или реплики, прописаны в явном виде - исправьте их, чтобы они соответствовали данной реплике. (Альтернативный вариант - запустить сервер и сделать самостоятельно все запросы `ATTACH TABLE`, которые должны были бы быть в соответствующих .sql файлах в `/var/lib/clickhouse/metadata/`.)
4. Создайте в ZooKeeper узел `/path_to_table/replica_name/flags/force_restore_data` с любым содержимым или выполните команду для восстановления всех реплицируемых таблиц: `sudo -u clickhouse touch /var/lib/clickhouse/flags/force_restore_data`

Затем запустите сервер (перезапустите, если уже запущен). Данные будут скачаны с реплик.

В качестве альтернативного варианта восстановления, вы можете удалить из ZooKeeper информацию о потерянной реплике (`/path_to_table/replica_name`), и затем создать реплику заново, как написано в разделе Создание реплицированных таблиц .

Отсутствует ограничение на использование сетевой полосы при восстановлении. Имейте это ввиду, если восстанавливаете сразу много реплик.

Преобразование из MergeTree в ReplicatedMergeTree

Здесь и далее, под `MergeTree` подразумеваются все движки таблиц семейства `MergeTree`, так же для `ReplicatedMergeTree`.

Если у вас была таблица типа `MergeTree`, репликация которой делалась вручную, вы можете преобразовать её в реплицируемую таблицу. Это может понадобиться лишь в случаях, когда вы уже успели накопить большое количество данных в таблице типа `MergeTree`, а сейчас хотите включить репликацию.

Если на разных репликах данные отличаются, то сначала синхронизируйте их, либо удалите эти данные на всех репликах кроме одной.

Переименуйте имеющуюся MergeTree таблицу, затем создайте со старым именем таблицу типа `ReplicatedMergeTree`.
Перенесите данные из старой таблицы в поддиректорию detached в директории с данными новой таблицы (`/var/lib/clickhouse/data/db_name/table_name/`).
Затем добавьте эти куски данных в рабочий набор с помощью выполнения запросов `ALTER TABLE ATTACH PARTITION` на одной из реплик.

PARTITION на одной из реплик.

Преобразование из ReplicatedMergeTree в MergeTree

Создайте таблицу типа MergeTree с другим именем. Перенесите в её директорию с данными все данные из директории с данными таблицы типа `ReplicatedMergeTree`. Затем удалите таблицу типа `ReplicatedMergeTree` и перезапустите сервер.

Если вы хотите избавиться от таблицы `ReplicatedMergeTree`, не запуская сервер, то

- удалите соответствующий файл `.sql` в директории с метаданными (`/var/lib/clickhouse/metadata/`);
- удалите соответствующий путь в ZooKeeper (`/path_to_table/replica_name`);

После этого, вы можете запустить сервер, создать таблицу типа `MergeTree`, перенести данные в её директорию, и перезапустить сервер.

Восстановление в случае потери или повреждения метаданных на ZooKeeper кластере

Если данные в ZooKeeper оказались утеряны или повреждены, то вы можете сохранить данные, переместив их в нереплицируемую таблицу, как описано в пункте выше.

Произвольный ключ партиционирования

Партиционирование данных доступно для таблиц семейства MergeTree (включая реплицированные таблицы). Таблицы MaterializedView, созданные на основе таблиц MergeTree, также поддерживают партиционирование.

Партиция – это набор записей в таблице, объединенных по какому-либо критерию. Например, партиция может быть по месяцу, по дню или по типу события. Данные для разных партиций хранятся отдельно. Это позволяет оптимизировать работу с данными, так как при обработке запросов будет использоваться только необходимое подмножество из всевозможных данных. Например, при получении данных за определенный месяц, ClickHouse будет считывать данные только за этот месяц.

Ключ партиционирования задается при создании таблицы, в секции `PARTITION BY expr`. Ключ может представлять собой произвольное выражение из столбцов таблицы. Например, чтобы задать партиционирования по месяцам, можно использовать выражение `toYYYYMM(date_column)`:

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

Ключом партиционирования также может быть кортеж из выражений (аналогично первичному ключу). Например:

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

В этом примере задано партиционирование по типам событий, произошедших в течение текущей недели.

Каждая партиция состоит из отдельных фрагментов или так называемых *кусков данных*. Каждый кусок отсортирован по первичному ключу. При вставке данных в таблицу каждая отдельная запись

сохраняется в виде отдельного куска. Через некоторое время после вставки (обычно до 10 минут), ClickHouse выполняет в фоновом режиме слияние данных — в результате куски для одной и той же партиции будут объединены в более крупный кусок.

Info

Не рекомендуется делать слишком гранулированное партиционирование – то есть задавать партиции по столбцу, в котором будет слишком большой разброс значений (речь идет о порядке более тысячи партиций). Это приведет к скоплению большого числа файлов и файловых дескрипторов в системе, что может значительно снизить производительность запросов `SELECT`.

Чтобы получить набор кусков и партиций таблицы, можно воспользоваться системной таблицей system.parts. В качестве примера рассмотрим таблицу `visits`, в которой задано партиционирование по месяцам. Выполним `SELECT` для таблицы `system.parts`:

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

Столбец `partition` содержит имена всех партиций таблицы. Таблица `visits` из нашего примера содержит две партиции: `201901` и `201902`. Используйте значения из этого столбца в запросах ALTER ... PARTITION.

Столбец `name` содержит названия кусков партиций. Значения из этого столбца можно использовать в запросах ALTER ATTACH PART.

Столбец `active` отображает состояние куска. `1` означает, что кусок активен; `0` – неактивен. К неактивным можно отнести куски, оставшиеся после слияния данных. Поврежденные куски также отображаются как неактивные. Неактивные куски удаляются приблизительно через 10 минут после того, как было выполнено слияние.

Рассмотрим детальнее имя первого куска `201901_1_3_1`:

- `201901` имя партиции;
- `1` – минимальный номер блока данных;
- `3` – максимальный номер блока данных;
- `1` – уровень куска (глубина дерева слияний, которыми этот кусок образован).

Info

Названия кусков для таблиц старого типа образуются следующим образом: `20190117_20190123_2_2_0` (минимальная дата _ максимальная дата _ номер минимального блока _ номер максимального блока _ уровень).

Как видно из примера выше, таблица содержит несколько отдельных кусков для одной и той же партиции (например, куски `201901_1_3_1` и `201901_1_9_2` принадлежат партиции `201901`). Это означает, что эти куски еще не были объединены – в файловой системе они хранятся отдельно. После того как будет выполнено автоматическое слияние данных (выполняется примерно спустя 10 минут после вставки

данных), исходные куски будут объединены в один более крупный кусок и помечены как неактивные.

Вы можете запустить внеочередное слияние данных с помощью запроса OPTIMIZE. Пример:

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

Неактивные куски будут удалены примерно через 10 минут после слияния.

Другой способ посмотреть набор кусков и партиций – зайти в директорию с данными таблицы: `/var/lib/clickhouse/data/<database>/<table>/`. Например:

```
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
```

'201901_1_1_0', '201901_1_7_1' и т. д. – это директории кусков партиции. Каждый кусок содержит данные только для соответствующего месяца (таблица в данном примере содержит партиционирование по месяцам).

Директория `detached` содержит куски, отсоединенные от таблицы с помощью запроса DETACH. Поврежденные куски также попадают в эту директорию – они не удаляются с сервера.

Сервер не использует куски из директории `detached`. Вы можете в любое время добавлять, удалять, модифицировать данные в директории detached - сервер не будет об этом знать, пока вы не сделаете запрос ATTACH.

Следует иметь в виду, что при работающем сервере нельзя вручную изменять набор кусков на файловой системе, так как сервер не будет знать об этом.
Для нереплицируемых таблиц, вы можете это делать при остановленном сервере, однако это не рекомендуется.
Для реплицируемых таблиц, набор кусков нельзя менять в любом случае.

ClickHouse позволяет производить различные манипуляции с кусками: удалять, копировать из одной таблицы в другую или создавать их резервные копии. Подробнее см. в разделе Манипуляции с партициями и кусками.

ReplacingMergeTree

Движок отличается от MergeTree тем, что выполняет удаление дублирующихся записей с одинаковым значением первичного ключа (точнее, с одинаковым значением ключа сортировки).

значением первичного ключа (точнее, с одинаковым значением ключа сортировки).

Дедупликация данных производится лишь во время слияний. Слияние происходят в фоне в неизвестный момент времени, на который вы не можете ориентироваться. Некоторая часть данных может остаться необработанной. Хотя вы можете вызвать внеочередное слияние с помощью запроса `OPTIMIZE`, на это не стоит рассчитывать, так как запрос `OPTIMIZE` приводит к чтению и записи большого объёма данных.

Таким образом, `ReplacingMergeTree` подходит для фоновой чистки дублирующихся данных в целях экономии места, но не даёт гарантии отсутствия дубликатов.

Создание таблицы

```
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, ...]
```

Описание параметров запроса смотрите в описании запроса.

Параметры ReplacingMergeTree

- `ver` — столбец с версией, тип `UInt*`, `Date` или `DateTime`. Необязательный параметр.

  При слиянии, из всех строк с одинаковым значением первичного ключа `ReplacingMergeTree` оставляет только одну:

  - Последнюю в выборке, если `ver` не задан.
  - С максимальной версией, если `ver` задан.

Секции запроса

При создании таблицы `ReplacingMergeTree` используются те же секции, что и при создании таблицы `MergeTree`.

▼ Устаревший способ создания таблицы

Attention

Не используйте этот способ в новых проектах и по возможности переведите старые проекты на способ описанный выше.

```
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])
```

Все параметры, кроме `ver` имеют то же значение, что в и `MergeTree`.

- `ver` — столбец с версией. Необязательный параметр. Описание смотрите выше по тексту.

SummingMergeTree

Движок наследует функциональность MergeTree. Отличие заключается в том, что для таблиц `SummingMergeTree` при слиянии кусков данных ClickHouse все строки с одинаковым первичным ключом

(точнее, с одинаковым ключом сортировки) заменяет на одну, которая хранит только суммы значений из столбцов с цифровым типом данных. Если ключ сортировки подобран таким образом, что одному значению ключа соответствует много строк, это значительно уменьшает объем хранения и ускоряет последующую выборку данных.

Мы рекомендуем использовать движок в паре с `MergeTree`. В `MergeTree` храните полные данные, а `SummingMergeTree` используйте для хранения агрегированных данных, например, при подготовке отчетов. Такой подход позволит не утратить ценные данные из-за неправильно выбранного первичного ключа.

Создание таблицы

```
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, ...]
```

Описание параметров запроса смотрите в описании запроса.

Параметры SummingMergeTree

- `columns` — кортеж с именами столбцов, в которых будут суммироваться данные. Необязательный параметр.
  Столбцы должны иметь числовой тип и не должны входить в первичный ключ.

  Если `columns` не задан, то ClickHouse суммирует значения во всех столбцах с числовым типом данных, не входящих в первичный ключ.

Секции запроса

При создании таблицы `SummingMergeTree` используются те же секции запроса, что и при создании таблицы `MergeTree`.

▼ Устаревший способ создания таблицы

Attention

Не используйте этот способ в новых проектах и по возможности переведите старые проекты на способ описанный выше.

```
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])
```

Все параметры, кроме `columns` имеют то же значение, что в и `MergeTree`.

- `columns` — кортеж с именами столбцов для суммирования данных. Необязательный параметр.
  Описание смотрите выше по тексту.

Пример использования

Рассмотрим следующую таблицу:

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

Добавим в неё данные:

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

ClickHouse может не полностью просуммировать все строки (смотрите ниже по тексту), поэтому при запросе мы используем агрегатную функцию `sum` и секцию `GROUP BY`.

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

```
┌─key─┬─sum(value)─┐
│   2 │          1 │
│   1 │          3 │
└─────┴────────────┘
```

Обработка данных

При вставке данных в таблицу они сохраняются как есть. Периодически ClickHouse выполняет слияние вставленных кусков данных и именно в этот момент производится суммирование и замена многих строк с одинаковым первичным ключом на одну для каждого результирующего куска данных.

ClickHouse может слить куски данных таким образом, что не все строки с одинаковым первичным ключом окажутся в одном финальном куске, т.е. суммирование будет не полным. Поэтому, при выборке данных (`SELECT`) необходимо использовать агрегатную функцию sum() и секцию `GROUP BY` как описано в примере выше.

Общие правила суммирования

Суммируются значения в столбцах с числовым типом данных. Набор столбцов определяется параметром `columns`.

Если значения во всех столбцах для суммирования оказались нулевыми, то строчка удаляется.

Для столбцов, не входящих в первичный ключ и не суммирующихся, выбирается произвольное значение из имеющихся.

Значения для столбцов, входящих в первичный ключ, не суммируются.

Суммирование в столбцах AggregateFunction

Для столбцов типа AggregateFunction ClickHouse выполняет агрегацию согласно заданной функции, повторяя поведение движка AggregatingMergeTree.

Вложенные структуры

Таблица может иметь вложенные структуры данных, которые обрабатываются особым образом.

Если название вложенной таблицы заканчивается на `Map` и она содержит не менее двух столбцов, удовлетворяющих критериям:

- первый столбец - числовой `(*Int*, Date, DateTime)`, назовем его условно `key`,
- остальные столбцы - арифметические `(*Int*, Float32/64)`, условно `(values...)`,

то вложенная таблица воспринимается как отображение `key => (values...)` и при слиянии её строк выполняется слияние элементов двух множеств по `key` со сложением соответствующих `(values...)`.

Примеры:

```
[(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)]
```

При запросе данных используйте функцию sumMap(key, value) для агрегации `Map`.

Для вложенной структуры данных не нужно указывать её столбцы в кортеже столбцов для суммирования.

AggregatingMergeTree

Движок наследует функциональность MergeTree, изменяя логику слияния кусков данных. Все строки с одинаковым первичным ключом (точнее, с одинаковым ключом сортировки) ClickHouse заменяет на одну (в пределах одного куска данных), которая хранит объединение состояний агрегатных функций.

Таблицы типа `AggregatingMergeTree` могут использоваться для инкрементальной агрегации данных, в том числе, для агрегирующих материализованных представлений.

Движок обрабатывает все столбцы типа AggregateFunction.

Использование `AggregatingMergeTree` оправдано только в том случае, когда это уменьшает количество строк на порядки.

Создание таблицы

```
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, ...]
```

Описание параметров запроса смотрите в описании запроса.

Секции запроса

При создании таблицы `AggregatingMergeTree` используются те же секции, что и при создании таблицы `MergeTree`.

▼ Устаревший способ создания таблицы

Attention

Не используйте этот способ в новых проектах и по возможности переведите старые проекты на способ описанный выше.

```
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)
```

Все параметры имеют то же значение, что в и `MergeTree`.

SELECT/INSERT данных

Для вставки данных используйте `INSERT SELECT` с агрегатными `-State`-функциями.

При выборке данных из таблицы `AggregatingMergeTree`, используйте `GROUP BY` и те же агрегатные функции, что и при вставке данных, но с суффиксом `-Merge`.

В запросах `SELECT` значения типа `AggregateFunction` выводятся во всех форматах, которые поддерживает ClickHouse, в виде implementation-specific бинарных данных. Если с помощью `SELECT` выполнить дамп данных, например, в формат `TabSeparated`, то потом этот дамп можно загрузить обратно с помощью запроса `INSERT`.

Пример агрегирущего материализованного представления

Создаём материализованное представление типа `AggregatingMergeTree`, следящее за таблицей `test.visits`:

```
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;
```

Вставляем данные в таблицу `test.visits`:

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

Данные окажутся и в таблице и в представлении `test.basic`, которое выполнит агрегацию.

Чтобы получить агрегированные данные, выполним запрос вида `SELECT ... GROUP BY ...` из представления `test.basic`:

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

CollapsingMergeTree

Движок наследует функциональность от MergeTree и добавляет в алгоритм слияния кусков данных логику сворачивания (удаления) строк.

`CollapsingMergeTree` асинхронно удаляет (сворачивает) пары строк, если все поля в строке эквивалентны, за исключением специального поля `Sign`, которое может принимать значения `1` и `-1`. Строки без пары сохраняются. Подробнее смотрите раздел Сворачивание (удаление) строк.

Движок может значительно уменьшить объем хранения и, как следствие, повысить эффективность запросов `SELECT`.

Создание таблицы

```
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, ...]
```

Подробности про `CREATE TABLE` смотрите в описании запроса.

Параметры CollapsingMergeTree

- `sign` — Имя столбца с типом строки: `1` — строка состояния, `-1` — строка отмены состояния.

  Тип данных столбца — `Int8`.

Секции запроса
При создании таблицы `CollapsingMergeTree` используются те же секции запроса, что и при создании таблицы `MergeTree`.

▼ Устаревший способ создания таблицы

Attention

Не используйте этот способ в новых проектах и по возможности переведите старые проекты на способ описанный выше.

```
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)
```

Все параметры, кроме `ver` имеют то же значение, что и в `MergeTree`.

- `sign` — Имя столбца с типом строки: `1` — строка состояния, `-1` — строка отмены состояния.

  Тип данных столбца — `Int8`.

Сворачивание (удаление) строк

Данные

Рассмотрим ситуацию, когда необходимо сохранять постоянно изменяющиеся данные для какого-либо объекта. Кажется логичным иметь одну строку для объекта и обновлять её при любом изменении, однако операция обновления является дорогостоящей и медленной для СУБД, поскольку требует перезаписи данных в хранилище. Если необходимо быстро записать данные, обновление не допустимо, но можно записать изменения объекта последовательно как описано ниже.

Используйте специальный столбец `Sign`. Если `Sign = 1`, то это означает, что строка является состоянием объекта, назовём её строкой состояния. Если `Sign = -1`, то это означает отмену состояния объекта с теми же атрибутами, назовём её строкой отмены состояния.

Например, мы хотим рассчитать, сколько страниц проверили пользователи на каком-то сайте и как долго они там находились. В какой-то момент времени мы пишем следующую строку с состоянием действий

пользователя:

```
┌──────────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │      5 │    146 │   1 │
└─────────────────────────┴────────────┴──────────┴──────┘
```

Через некоторое время мы регистрируем изменение активности пользователя и записываем его следующими двумя строками.

```
┌──────────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │      5 │    146 │  -1 │
│ 4324182021466249494 │      6 │    185 │   1 │
└─────────────────────────┴────────────┴──────────┴──────┘
```

Первая строка отменяет предыдущее состояние объекта (пользователя). Она должен повторять все поля отменённого состояния за исключением `Sign`.

Вторая строка содержит текущее состояние.

Поскольку нам нужно только последнее состояние активности пользователя, строки

```
┌──────────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │      5 │    146 │   1 │
│ 4324182021466249494 │      5 │    146 │  -1 │
└─────────────────────────┴────────────┴──────────┴──────┘
```

можно удалить, сворачивая (удаляя) устаревшее состояние объекта. `CollapsingMergeTree` выполняет это при слиянии кусков данных.

Зачем нужны 2 строки для каждого изменения, читайте в параграфе Алгоритм.

Особенности подхода

1. Программа, которая записывает данные, должна помнить состояние объекта, чтобы иметь возможность отменить его. Строка отмены состояния должна быть копией предыдущей строки состояния с противоположным значением `Sign`. Это увеличивает начальный размер хранилища, но позволяет быстро записывать данные.
2. Длинные растущие массивы в Столбцах снижают эффективность работы движка за счёт нагрузки на запись. Чем проще данные, тем выше эффективность.
3. Результаты запроса `SELECT` сильно зависят от согласованности истории изменений объекта. Будьте точны при подготовке данных для вставки. Можно получить непредсказуемые результаты для несогласованных данных, например отрицательные значения для неотрицательных метрик, таких как глубина сеанса.

Алгоритм

Когда ClickHouse объединяет куски данных, каждая группа последовательных строк с одним и тем же первичным ключом уменьшается до не более чем двух строк, одна из которых имеет `Sign = 1` (строка состояния), а другая строка с `Sign = -1` (строка отмены состояния). Другими словами, записи сворачиваются.

Для каждого результирующего куска данных ClickHouse сохраняет:

1. Первую строку отмены состояния и последнюю строку состояния, если количество строк обоих видов совпадает.

2. Последнюю строку состояния, если строк состояния на одну больше, чем строк отмены состояния.

3. Первую строку отмены состояния, если их на одну больше, чем строк состояния.

4. Ни в одну из строк во всех остальных случаях.

   Слияние продолжается, но ClickHouse рассматривает эту ситуацию как логическую ошибку и записывает её в журнал сервера. Эта ошибка может возникать, если одни и те же данные вставлялись несколько раз.

Как видно, от сворачивания не должны меняться результаты расчётов статистик.
Изменения постепенно сворачиваются так, что остаются лишь последнее состояние почти каждого объекта.

Столбец `Sign` необходим, поскольку алгоритм слияния не гарантирует, что все строки с одинаковым первичным ключом будут находиться в одном результирующем куске данных и даже на одном физическом сервере. ClickHouse выполняет запросы `SELECT` несколькими потоками, и он не может предсказать порядок строк в результате. Если необходимо получить полностью свёрнутые данные из таблицы `CollapsingMergeTree`, то необходимо агрегирование.

Для завершения свертывания добавьте в запрос секцию `GROUP BY` и агрегатные функции, которые учитывают знак. Например, для расчета количества используйте `sum(Sign)` вместо `count()`. Чтобы вычислить сумму чего-либо, используйте `sum(Sign * x)` вместо `sum(x)`, и так далее, а также добавьте `HAVING sum(Sign) > 0` .

Таким образом можно вычислять агрегации `count`, `sum` и `avg`. Если объект имеет хотя бы одно не свёрнутое состояние, то может быть вычислена агрегация `uniq`. Агрегации `min` и `max` невозможно вычислить, поскольку `CollapsingMergeTree` не сохраняет историю значений свернутых состояний.

Если необходимо выбирать данные без агрегации (например, проверить наличие строк, последние значения которых удовлетворяют некоторым условиям), можно использовать модификатор `FINAL` для секции `FROM`. Это вариант существенно менее эффективен.

Пример использования

Example data:

```
┌──────────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │       5 │    146 │   1 │
│ 4324182021466249494 │       5 │    146 │  -1 │
│ 4324182021466249494 │       6 │    185 │   1 │
└─────────────────────┴───────────┴──────────┴──────┘
```

Создание таблицы:

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

Insertion of the data:

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

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

Мы используем два запроса `INSERT` для создания двух различных кусков данных. Если вставить данные одним запросом, ClickHouse создаёт один кусок данных и никогда не будет выполнять слияние.

Получение данных:

```
SELECT * FROM UAct
```

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │         5 │      146 │   -1 │
│ 4324182021466249494 │         6 │      185 │    1 │
└─────────────────────┴───────────┴──────────┴──────┘
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │         5 │      146 │    1 │
└─────────────────────┴───────────┴──────────┴──────┘
```

Что мы видим и где сворачивание?

Двумя запросами `INSERT`, мы создали два куска данных. Запрос `SELECT` был выполнен в 2 потока, и мы получили случайный порядок строк. Сворачивание не произошло, так как слияние кусков данных еще не произошло. ClickHouse объединяет куски данных в неизвестный момент времени, который мы не можем предсказать.

Таким образом, нам нужна агрегация:

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

Если нам не нужна агрегация, но мы хотим принудительно выполнить свёртку данных, можно использовать модификатор `FINAL` для секции `FROM`.

```
SELECT * FROM UAct FINAL
```

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │         6 │      185 │    1 │
└─────────────────────┴───────────┴──────────┴──────┘
```

Такой способ выбора данных очень неэффективен. Не используйте его для больших таблиц.

VersionedCollapsingMergeTree

Движок:

- Позволяет быстро записывать постоянно изменяющиеся состояния объектов.
- Удаляет старые состояния объектов в фоновом режиме. Это значительно сокращает объем хранения.

Подробнее читайте в разделе Collapsing

Подробнее читайте в разделе Collapsing.

Движок наследует функциональность от MergeTree и добавляет в алгоритм слияния кусков данных логику сворачивания (удаления) строк. `VersionedCollapsingMergeTree` предназначен для тех же задач, что и CollapsingMergeTree, но использует другой алгоритм свёртывания, который позволяет вставлять данные в любом порядке в несколько потоков. В частности, столбец `Version` помогает свернуть строки правильно, даже если они вставлены в неправильном порядке. `CollapsingMergeTree` требует строго последовательную вставку данных.

Создание таблицы

```
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, ...]
```

Подробности про `CREATE TABLE` смотрите в описании запроса.

Параметры движка

```
VersionedCollapsingMergeTree(sign, version)
```

- `sign` — Имя столбца с типом строки: `1` — строка состояния, `-1` — строка отмены состояния.

  Тип данных столбца должен быть `Int8`.

- `version` — имя столбца с версией состояния объекта.

  Тип данных столбца должен быть `UInt*`.

Секции запроса

При создании таблицы `VersionedCollapsingMergeTree` используются те же секции запроса, что и при создании таблицы `MergeTree` .

▼ Устаревший способ создания таблицы

Внимание

Не используйте этот метод в новых проектах. По возможности переключите старые проекты на метод, описанный выше.

```
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)
```

Все параметры, за исключением `sign` и `version` имеют то же значение, что и в `MergeTree`.

- `sign` — Имя столбца с типом строки: `1` — строка состояния, `-1` — строка отмены состояния.

  Тип данных столбца — `Int8`.

- `version` — имя столбца с версией состояния объекта.

  Тип данных столбца должен быть `UInt*`.

Сворачивание (удаление) строк

Данные

Рассмотрим ситуацию, когда необходимо сохранять постоянно изменяющиеся данные для какого-либо объекта. Разумно иметь одну строку для объекта и обновлять эту строку при каждом изменении. Однако операция обновления является дорогостоящей и медленной для СУБД, поскольку требует перезаписи данных в хранилище. Обновление неприемлемо, если требуется быстро записывать данные, но можно записывать изменения в объект последовательно следующим образом.

Используйте столбец `Sign` при записи строки. Если `Sign = 1`, то это означает, что строка является состоянием объекта, назовём её строкой состояния. Если `Sign = -1`, то это означает отмену состояния объекта с теми же атрибутами, назовём её строкой отмены состояния. Также используйте столбец `Version`, который должен идентифицировать каждое состояние объекта отдельным номером.

Например, мы хотим рассчитать, сколько страниц пользователи посетили на каком-либо сайте и как долго они там находились. В какой-то момент времени мы записываем следующую строку состояния пользовательской активности:

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┬─Version─┐
│ 4324182021466249494 │         5 │      146 │    1 │       1 │
└─────────────────────┴───────────┴──────────┴──────┴─────────┘
```

Через некоторое время мы регистрируем изменение активности пользователя и записываем его следующими двумя строками.

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┬─Version─┐
│ 4324182021466249494 │         5 │      146 │   -1 │       1 │
│ 4324182021466249494 │         6 │      185 │    1 │       2 │
└─────────────────────┴───────────┴──────────┴──────┴─────────┘
```

Первая строка отменяет предыдущее состояние объекта (пользователя). Она должна копировать все поля отменяемого состояния за исключением `Sign`.

Вторая строка содержит текущее состояние.

Поскольку нам нужно только последнее состояние активности пользователя, строки

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┬─Version─┐
│ 4324182021466249494 │         5 │      146 │    1 │       1 │
│ 4324182021466249494 │         5 │      146 │   -1 │       1 │
└─────────────────────┴───────────┴──────────┴──────┴─────────┘
```

можно удалить, сворачивая (удаляя) устаревшее состояние объекта. `VersionedCollapsingMergeTree` делает это при слиянии кусков данных.

Чтобы узнать, зачем нам нужны две строки для каждого изменения, см. раздел Алгоритм.

Примечания по использованию

1. Программа, которая записывает данные, должна помнить состояние объекта, чтобы иметь возможность отменить его. Строка отмены состояния должна быть копией предыдущей строки состояния с противоположным значением `Sign`. Это увеличивает начальный размер хранилища, но позволяет быстро записывать данные.

2. Длинные растущие массивы в столбцах снижают эффективность работы движка за счет нагрузки на запись. Чем проще данные, тем выше эффективность.
3. `SELECT` результаты сильно зависят от согласованности истории изменений объекта. Будьте точны при подготовке данных для вставки. Вы можете получить непредсказуемые результаты с несогласованными данными, такими как отрицательные значения для неотрицательных метрик, таких как глубина сеанса.

Алгоритм

Когда ClickHouse объединяет куски данных, он удаляет каждую пару строк, которые имеют один и тот же первичный ключ и версию и разный `Sign`. Порядок строк не имеет значения.

Когда ClickHouse вставляет данные, он упорядочивает строки по первичному ключу. Если столбец `Version` не находится в первичном ключе, ClickHouse добавляет его к первичному ключу неявно как последнее поле и использует для сортировки.

Выборка данных

ClickHouse не гарантирует, что все строки с одинаковым первичным ключом будут находиться в одном результирующем куске данных или даже на одном физическом сервере. Это справедливо как для записи данных, так и для последующего слияния кусков данных. Кроме того, ClickHouse обрабатывает запросы `SELECT` несколькими потоками, и не может предсказать порядок строк в конечной выборке. Это означает, что если необходимо получить полностью "свернутые" данные из таблицы `VersionedCollapsingMergeTree`, то требуется агрегирование.

Для завершения свертывания добавьте в запрос секцию `GROUP BY` и агрегатные функции, которые учитывают знак. Например, для расчета количества используйте `sum(Sign)` вместо `count()`. Чтобы вычислить сумму чего-либо, используйте `sum(Sign * x)` вместо `sum(x)`, а также добавьте `HAVING sum(Sign) > 0` .

Таким образом можно вычислять агрегации `count`, `sum` и `avg`. Агрегация `uniq` может вычисляться, если объект имеет хотя бы одно не свернутое состояние. Невозможно вычислить агрегации `min` и `max` поскольку `VersionedCollapsingMergeTree` не сохраняет историю значений для свернутых состояний.

Если необходимо выбирать данные без агрегации (например, проверить наличие строк, последние значения которых удовлетворяют некоторым условиям), можно использовать модификатор `FINAL` для секции `FROM`. Такой подход неэффективен и не должен использоваться с большими таблицами.

Пример использования

Данные для примера:

```
┌──────────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┬─Version─┐
│ 4321821021466249494 │       5 │    146 │   1 │     1 |
│ 4321821021466249494 │       5 │    146 │  -1 │     1 |
│ 4321821021466249494 │       6 │    185 │   1 │     2 |
└─────────────────────┴───────────┴──────────┴──────┴─────────┘
```

Создание таблицы:

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

Вставка данных:

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

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

Мы используем два запроса `INSERT` для создания двух различных кусков данных. Если мы вставляем данные с помощью одного запроса, ClickHouse создаёт один кусок данных и не будет выполнять слияние.

Получение данных:

```
SELECT * FROM UAct
```

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┬─Version─┐
│ 4324182021466249494 │         5 │      146 │    1 │       1 │
└─────────────────────┴───────────┴──────────┴──────┴─────────┘
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┬─Version─┐
│ 4324182021466249494 │         5 │      146 │   -1 │       1 │
│ 4324182021466249494 │         6 │      185 │    1 │       2 │
└─────────────────────┴───────────┴──────────┴──────┴─────────┘
```

Что мы видим и где сворачивание?
Мы создали два куска данных, используя два запроса `INSERT`. Запрос `SELECT` был выполнен в два потока, и результатом является случайный порядок строк.
Свертывание не произошло, поскольку части данных еще не были объединены. ClickHouse объединяет части данных в неизвестный момент времени, который мы не можем предсказать.

Поэтому нам нужна агрегация:

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

Если нам не нужна агрегация, но мы хотим принудительно выполнить свёртку данных, то можно использовать модификатор `FINAL` для секции `FROM`.

```
SELECT * FROM UAct FINAL
```

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┬─Version─┐
│ 4324182021466249494 │         6 │      185 │    1 │       2 │
└─────────────────────┴───────────┴──────────┴──────┴─────────┘
```

Это очень неэффективный способ выбора данных. Не используйте его для больших таблиц.

GraphiteMergeTree

Движок предназначен для прореживания и агрегирования/усреднения (rollup) данных Graphite. Он может быть интересен разработчикам, которые хотят использовать ClickHouse как хранилище данных для Graphite.

Если rollup не требуется, то для хранения данных Graphite можно использовать любой движок таблиц ClickHouse, в противном случае используйте `GraphiteMergeTree`. Движок уменьшает объем хранения и повышает эффективность запросов от Graphite.

Движок наследует свойства от MergeTree.

Создание таблицы

```
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, ...]
```

Описание параметров запроса смотрите в описании запроса.

Таблица для данных Graphite должна содержать следующие столбцы:

- Колонка с названием метрики (Graphite sensor). Тип данных: `String`.
- Столбец со временем измерения метрики. Тип данных `DateTime`.
- Столбец со значением метрики. Тип данных: любой числовой.
- Столбец с версией метрики. Тип данных: любой числовой.

  ClickHouse сохраняет строки с последней версией или последнюю записанную строку, если версии совпадают. Другие строки удаляются при слиянии кусков данных.

Имена этих столбцов должны быть заданы в конфигурации rollup.

Параметры GraphiteMergeTree

- `config_section` — имя раздела в конфигурационном файле, в котором находятся правила rollup.

Секции запроса

При создании таблицы `GraphiteMergeTree` используются те же секции запроса, что при создании таблицы `MergeTree`.

▼ Устаревший способ создания таблицы

Attention

Не используйте этот способ в новых проектах и по возможности переведите старые проекты на способ описанный выше.

```
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)
```

Все параметры, кроме `config_section` имеют то же значение, что в `MergeTree`.

- `config_section` — имя раздела в конфигурационном файле, в котором находятся правила rollup.

Конфигурация rollup

Настройки для прореживания данных задаются параметром graphite_rollup. Имя параметра может быть любым. Можно создать несколько конфигураций и использовать их для разных таблиц.

Структура конфигурации rollup:

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

Важно: порядок разделов `pattern` должен быть следующим:

1. Разделы *без* параметра `function` *или* `retention`.
2. Разделы *с* параметрами `function` *и* `retention`.
3. Раздел `default`.

При обработке строки ClickHouse проверяет правила в разделах `pattern`. Каждый из разделов `pattern` (включая `default`) может содержать параметр `function` для аггрегации, правила `retention` для прореживания или оба эти параметра. Если имя метрики соответствует шаблону `regexp`, то применяются правила из раздела (или разделов) `pattern`, в противном случае из раздела `default`.

Поля для разделов `pattern` и `default`:

- `regexp` – шаблон имени метрики.
- `age` – минимальный возраст данных в секундах.
- `precision` – точность определения возраста данных в секундах. Должен быть делителем для 86400 (количество секунд в дне).
- `function` – имя агрегирующей функции, которую следует применить к данным, чей возраст оказался в интервале `[age, age + precision]`.

`required-columns`:

- `path_column_name` — колонка с названием метрики (Graphite sensor)

- path_column_name — колонка с названием метрики (Graphite sensor).
- time_column_name — столбец со временем измерения метрики.
- value_column_name — столбец со значением метрики в момент времени, установленный в time_column_name.
- version_column_name — столбец с версией метрики.

Пример настройки:

```
<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>\.count$</regexp>
        <function>sum</function>
    </pattern>
    <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>
```

Семейство Log

Движки разработаны для сценариев, когда необходимо записывать много таблиц с небольшим объемом данных (менее 1 миллиона строк).

Движки семейства:

- StripeLog
- Log
- TinyLog

Общие свойства

Движки:

- Хранят данные на диске.

- Добавляют данные в конец файла при записи.
- Не поддерживают операции мутации.
- Не поддерживают индексы.

  Это означает, что запросы `SELECT` не эффективны для выборки диапазонов данных.
- Записывают данные не атомарно.

  Вы можете получить таблицу с повреждёнными данными, если что-то нарушит операцию записи (например, аварийное завершение работы сервера).

Отличия

Движки `Log` и `StripeLog` поддерживают:

- Блокировки для конкурентного доступа к данным.

  Во время выполнения запроса `INSERT` таблица заблокирована и другие запросы на чтение и запись данных ожидают снятия блокировки. При отсутствии запросов на запись данных можно одновременно выполнять любое количество запросов на чтение данных.
- Параллельное чтение данных.

  ClickHouse читает данные в несколько потоков. Каждый поток обрабатывает отдельный блок данных.

Движок `Log` сохраняет каждый столбец таблицы в отдельном файле. Движок `StripeLog` хранит все данные в одном файле. Таким образом, движок `StripeLog` использует меньше дескрипторов в операционной системе, а движок `Log` обеспечивает более эффективное считывание данных.

Движок `TinyLog` самый простой в семье и обеспечивает самые низкие функциональность и эффективность. Движок `TinyLog` не поддерживает ни параллельного чтения данных, ни конкурентного доступа к данным. Он хранит каждый столбец в отдельном файле. Движок читает данные медленнее, чем оба других движка с параллельным чтением, и использует почти столько же дескрипторов, сколько и движок `Log`. Его можно использовать в простых сценариях с низкой нагрузкой.

StripeLog

Движок относится к семейству движков Log. Смотрите общие свойства и различия движков в статье Семейство Log.

Движок разработан для сценариев, когда необходимо записывать много таблиц с небольшим объемом данных (менее 1 миллиона строк).

Создание таблицы

```
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
```

Смотрите подробное описание запроса CREATE TABLE.

Запись данных

Движок `StripeLog` хранит все столбцы в одном файле. При каждом запросе `INSERT`, ClickHouse добавляет блок данных в конец файла таблицы, записывая столбцы один за другим.

Для каждой таблицы ClickHouse записывает файлы:

Для каждой таблицы ClickHouse записывает файлы:

- `data.bin` — файл с данными.
- `index.mrk` — файл с метками. Метки содержат смещения для каждого столбца каждого вставленного блока данных.

Движок `StripeLog` не поддерживает запросы `ALTER UPDATE` и `ALTER DELETE`.

Чтение данных

Файл с метками позволяет ClickHouse распараллеливать чтение данных. Это означает, что запрос `SELECT` возвращает строки в непредсказуемом порядке. Используйте секцию `ORDER BY` для сортировки строк.

Пример использования

Создание таблицы:

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

Вставка данных:

```
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')
```

Мы использовали два запроса `INSERT` для создания двух блоков данных внутри файла `data.bin`.

ClickHouse использует несколько потоков при выборе данных. Каждый поток считывает отдельный блок данных и возвращает результирующие строки независимо по мере завершения. В результате порядок блоков строк в выходных данных в большинстве случаев не совпадает с порядком тех же блоков во входных данных. Например:

```
SELECT * FROM stripe_log_table
```

```
┌─────────────timestamp─┬─message_type─┬─message────────────────────┐
│ 2019-01-18 14:27:32 │ REGULAR      │ The second regular message │
│ 2019-01-18 14:34:53 │ WARNING      │ The first warning message  │
└─────────────────────┴──────────────┴────────────────────────────┘

┌─────────────timestamp─┬─message_type─┬─message───────────────────┐
│ 2019-01-18 14:23:43 │ REGULAR      │ The first regular message │
└─────────────────────┴──────────────┴───────────────────────────┘
```

Сортировка результатов (по умолчанию по возрастанию):

```
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

Движок относится к семейству движков Log. Смотрите общие свойства и различия движков в статье Семейство Log.

Отличается от TinyLog тем, что вместе с файлами столбцов лежит небольшой файл "засечек". Засечки пишутся на каждый блок данных и содержат смещение - с какого места нужно читать файл, чтобы пропустить заданное количество строк. Это позволяет читать данные из таблицы в несколько потоков. При конкурентном доступе к данным, чтения могут выполняться одновременно, а записи блокируют чтения и друг друга.
Движок Log не поддерживает индексы. Также, если при записи в таблицу произошёл сбой, то таблица станет битой, и чтения из неё будут возвращать ошибку. Движок Log подходит для временных данных, write-once таблиц, а также для тестовых и демонстрационных целей.

TinyLog

Движок относится к семейству движков Log. Смотрите общие свойства и различия движков в статье Семейство Log.

Самый простой движок таблиц, который хранит данные на диске.
Каждый столбец хранится в отдельном сжатом файле.
При записи, данные дописываются в конец файлов.

Конкурентный доступ к данным никак не ограничивается:

- если вы одновременно читаете из таблицы и в другом запросе пишете в неё, то чтение будет завершено с ошибкой;
- если вы одновременно пишете в таблицу в нескольких запросах, то данные будут битыми.

Типичный способ использования этой таблицы - это write-once: сначала один раз только пишем данные, а потом сколько угодно читаем.
Запросы выполняются в один поток. То есть, этот движок предназначен для сравнительно маленьких таблиц (рекомендуется до 1 000 000 строк).
Этот движок таблиц имеет смысл использовать лишь в случае, если у вас есть много маленьких таблиц, так как он проще, чем движок Log (требуется открывать меньше файлов).
Случай, когда у вас много маленьких таблиц, является гарантированно плохим по производительности, но может уже использоваться при работе с другой СУБД, и вам может оказаться удобнее перейти на использование таблиц типа TinyLog.
Индексы не поддерживаются.

В Яндекс.Метрике таблицы типа TinyLog используются для промежуточных данных, обрабатываемых маленькими пачками.

Kafka

Движок работает с Apache Kafka.

Kafka позволяет:

- Публиковать/подписываться на потоки данных.
- Организовать отказоустойчивое хранилище.
- Обрабатывать потоки по мере их появления.

Создание таблицы

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
```

```
    ...
) ENGINE = Kafka()
SETTINGS
    kafka_broker_list = 'host:port',
    kafka_topic_list = 'topic1,topic2,...',
    kafka_group_name = 'group_name',
    kafka_format = 'data_format'[,]
    [kafka_row_delimiter = 'delimiter_symbol',]
    [kafka_schema = '',]
    [kafka_num_consumers = N,]
    [kafka_skip_broken_messages = <0|1>]
```

Обязательные параметры:

- `kafka_broker_list` – перечень брокеров, разделенный запятыми (`localhost:9092`).
- `kafka_topic_list` – перечень необходимых топиков Kafka.
- `kafka_group_name` – группа потребителя Kafka. Отступы для чтения отслеживаются для каждой группы отдельно. Если необходимо, чтобы сообщения не повторялись на кластере, используйте везде одно имя группы.
- `kafka_format` – формат сообщений. Названия форматов должны быть теми же, что можно использовать в секции `FORMAT`, например, `JSONEachRow`. Подробнее читайте в разделе Форматы.

Опциональные параметры:

- `kafka_row_delimiter` – символ-разделитель записей (строк), которым завершается сообщение.
- `kafka_schema` – опциональный параметр, необходимый, если используется формат, требующий определения схемы. Например, Cap'n Proto требует путь к файлу со схемой и название корневого объекта `schema.capnp:Message`.
- `kafka_num_consumers` – количество потребителей (consumer) на таблицу. По умолчанию: `1`. Укажите больше потребителей, если пропускная способность одного потребителя недостаточна. Общее число потребителей не должно превышать количество партиций в топике, так как на одну партицию может быть назначено не более одного потребителя.
- `kafka_skip_broken_messages` – режим обработки сообщений Kafka. Если `kafka_skip_broken_messages = 1`, то движок отбрасывает сообщения Кафки, которые не получилось обработать. Одно сообщение в точности соответствует одной записи (строке).

Примеры

```
 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;
```

▼ Устаревший способ создания таблицы

Attention

Не используйте этот метод в новых проектах. По возможности переключите старые проекты на метод, описанный выше.

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

Описание

Полученные сообщения отслеживаются автоматически, поэтому из одной группы каждое сообщение считывается только один раз. Если необходимо получить данные дважды, то создайте копию таблицы с другим именем группы.

Группы пластичны и синхронизированы на кластере. Например, если есть 10 топиков и 5 копий таблицы в кластере, то в каждую копию попадет по 2 топика. Если количество копий изменится, то распределение топиков по копиям изменится автоматически. Подробно читайте об этом на http://kafka.apache.org/intro.

Чтение сообщения с помощью `SELECT` не слишком полезно (разве что для отладки), поскольку каждое сообщения может быть прочитано только один раз. Практичнее создавать потоки реального времени с помощью материализованных преставлений. Для этого:

1. Создайте потребителя Kafka с помощью движка и рассматривайте его как поток данных.
2. Создайте таблицу с необходимой структурой.
3. Создайте материализованное представление, которое преобразует данные от движка и помещает их в ранее созданную таблицу.

Когда к движку присоединяется материализованное представление (`MATERIALIZED VIEW`), оно начинает в фоновом режиме собирать данные. Это позволяет непрерывно получать сообщения от Kafka и преобразовывать их в необходимый формат с помощью `SELECT`.

Пример:

```
 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;
```

Для улучшения производительности полученные сообщения группируются в блоки размера max_insert_block_size. Если блок не удалось сформировать за stream_flush_interval_ms миллисекунд, то данные будут сброшены в таблицу независимо от полноты блока.

данные будут сброшены в таблицу независимо от полноты блока.

Чтобы остановить получение данных топика или изменить логику преобразования, отсоедините материализованное представление:

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

Если необходимо изменить целевую таблицу с помощью `ALTER`, то материализованное представление рекомендуется отключить, чтобы избежать несостыковки между целевой таблицей и данными от представления.

Конфигурация

Аналогично GraphiteMergeTree, движок Kafka поддерживает расширенную конфигурацию с помощью конфигурационного файла ClickHouse. Существует два конфигурационных ключа, которые можно использовать: глобальный (`kafka`) и по топикам (`kafka_topic_*`). Сначала применяется глобальная конфигурация, затем конфигурация по топикам (если она существует).

```
<!-- 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>
```

В документе librdkafka configuration reference можно увидеть список возможных опций конфигурации. Используйте подчеркивание (`_`) вместо точки в конфигурации ClickHouse. Например, `check.crcs=true` будет соответствовать `<check_crcs>true</check_crcs>`.

MySQL

Движок MySQL позволяет выполнять `SELECT` запросы над данными, хранящимися на удалённом MySQL сервере.

Формат вызова:

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

Параметры вызова

- `host:port` — Адрес сервера MySQL.
- `database` — Имя базы данных на сервере MySQL.
- `table` — Имя таблицы.
- `user` — Пользователь MySQL.
- `password` — Пароль пользователя.
- `replace_query` — Флаг, устанавливающий замену запроса `INSERT INTO` на `REPLACE INTO`. Если `replace_query=1`, то запрос заменяется.
- `on_duplicate_clause` — Добавляет выражение `ON DUPLICATE KEY 'on_duplicate_clause'` в запрос `INSERT`.

  Например, `INSERT INTO t (c1,c2) VALUES ('a', 2) ON DUPLICATE KEY UPDATE c2 = c2 + 1`, где `on_duplicate_clause` — `UPDATE c2 = c2 + 1`. Какие выражения `on_duplicate_clause` вы можете использовать с `ON DUPLICATE KEY`, смотрите в документации MySQL.

Чтобы указать `on_duplicate_clause` необходимо передать `0` в параметр `replace_query`. Если одновременно передать `replace_query = 1` и `on_duplicate_clause`, то ClickHouse сгенерирует исключение.

На данный момент простые условия `WHERE`, такие как `=, !=, >, >=, <, <=` будут выполняться на стороне сервера MySQL.

Остальные условия и ограничение выборки `LIMIT` будут выполнены в ClickHouse только после выполнения запроса к MySQL.

Движок `MySQL` не поддерживает тип данных Nullable, поэтому при чтении данных из таблиц MySQL `NULL` преобразуются в значения по умолчанию для заданного типа столбца, обычно это 0 или пустая строка.

Distributed

Движок Distributed не хранит данные самостоятельно, а позволяет обрабатывать запросы распределённо, на нескольких серверах.
Чтение автоматически распараллеливается. При чтении будут использованы индексы таблиц на удалённых серверах, если есть.
Движок Distributed принимает параметры: имя кластера в конфигурационном файле сервера, имя удалённой базы данных, имя удалённой таблицы, а также (не обязательно) ключ шардирования.
Пример:

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

данные будут читаться со всех серверов кластера logs, из таблицы default.hits, расположенной на каждом сервере кластера.
Данные не только читаются, но и частично (настолько, насколько это возможно) обрабатываются на удалённых серверах.
Например, при запросе с GROUP BY, данные будут агрегированы на удалённых серверах, промежуточные состояния агрегатных функций будут отправлены на запросивший сервер; затем данные будут доагрегированы.

Вместо имени базы данных может использоваться константное выражение, возвращающее строку. Например, currentDatabase().

logs - имя кластера в конфигурационном файле сервера.

Кластеры задаются следующим образом:

```
<remote_servers>
  <logs>
    <shard>
      <!-- Не обязательно. Вес шарда при записи данных. По умолчанию, 1. -->
      <weight>1</weight>
      <!-- Не обязательно. Записывать ли данные только на одну, любую из реплик. По умолчанию, false -
записывать данные на все реплики. -->
      <internal_replication>false</internal_replication>
      <replica>
        <host>example01-01-1</host>
        <port>9000</port>
      </replica>
      <replica>
        <host>example01-01-2</host>
        <port>9000</port>
      </replica>
    </shard>
    <shard>
      <weight>2</weight>
      <internal_replication>false</internal_replication>
      <replica>
```

```
            <host>example01-02-1</host>
            <port>9000</port>
        </replica>
        <replica>
            <host>example01-02-2</host>
            <port>9000</port>
        </replica>
    </shard>
  </logs>
</remote_servers>
```

Здесь задан кластер с именем logs, состоящий из двух шардов, каждый из которых состоит из двух реплик.
Шардами называются серверы, содержащие разные части данных (чтобы прочитать все данные, нужно идти на все шарды).
Репликами называются дублирующие серверы (чтобы прочитать данные, можно идти за данными на любую из реплик).

Имя кластера не должно содержать точки.

В качестве параметров для каждого сервера указываются `host`, `port` и, не обязательно, `user`, `password`, `secure`, `compression`:

- `host` - адрес удалённого сервера. Может быть указан домен, или IPv4 или IPv6 адрес. В случае указания домена, при старте сервера делается DNS запрос, и результат запоминается на всё время работы сервера. Если DNS запрос неуспешен, то сервер не запускается. Если вы изменяете DNS-запись, перезапустите сервер.
- `port` - TCP-порт для межсерверного взаимодействия (в конфиге - tcp_port, обычно 9000). Не перепутайте с http_port.
- `user` - имя пользователя для соединения с удалённым сервером. по умолчанию - default. Этот пользователь должен иметь доступ для соединения с указанным сервером. Доступы настраиваются в файле users.xml, подробнее смотрите в разделе "Права доступа".
- `password` - пароль для соединения с удалённым сервером, в открытом виде. по умолчанию - пустая строка.
- `secure` - Использовать шифрованое соединение ssl, Обычно используется с портом `port` = 9440. Сервер должен слушать порт 9440 с корректными настройками сертификатов.
- `compression` - Использовать сжатие данных. По умолчанию: true.

При указании реплик, для каждого из шардов, при чтении, будет выбрана одна из доступных реплик. Можно настроить алгоритм балансировки нагрузки (то есть, предпочтения, на какую из реплик идти) - см. настройку load_balancing.
Если соединение с сервером не установлено, то будет произведена попытка соединения с небольшим таймаутом. Если соединиться не удалось, то будет выбрана следующая реплика, и так для всех реплик. Если попытка соединения для всех реплик не удалась, то будут снова произведены попытки соединения по кругу, и так несколько раз.
Это работает в пользу отказоустойчивости, хотя и не обеспечивает полную отказоустойчивость: удалённый сервер может принять соединение, но не работать, или плохо работать.

Можно указать от одного шарда (в таком случае, обработку запроса стоит называть удалённой, а не распределённой) до произвольного количества шардов. В каждом шарде можно указать от одной до произвольного числа реплик. Можно указать разное число реплик для каждого шарда.

Вы можете прописать сколько угодно кластеров в конфигурации.

Для просмотра имеющихся кластеров, вы можете использовать системную таблицу system.clusters.

Движок Distributed позволяет работать с кластером, как с локальным сервером. При этом, кластер является неэластичным: вы должны прописать его конфигурацию в конфигурационный файл сервера (лучше всех серверов кластера).

Не поддерживаются Distributed таблицы, смотрящие на другие Distributed таблицы (за исключением

случаев, когда у Distributed таблицы всего один шард). Вместо этого, сделайте так, чтобы Distributed таблица смотрела на "конечные" таблицы.

Как видно, движок Distributed требует прописывания кластера в конфигурационный файл; кластера из конфигурационного файла обновляются налету, без перезапуска сервера. Если вам необходимо каждый раз отправлять запрос на неизвестный набор шардов и реплик, вы можете не создавать Distributed таблицу, а воспользоваться табличной функцией remote. Смотрите раздел "Табличные функции".

Есть два способа записывать данные на кластер:

Во первых, вы можете самостоятельно определять, на какие серверы какие данные записывать, и выполнять запись непосредственно на каждый шард. То есть, делать INSERT в те таблицы, на которые "смотрит" распределённая таблица.
Это наиболее гибкое решение - вы можете использовать любую схему шардирования, которая может быть нетривиальной из-за требований предметной области.
Также это является наиболее оптимальным решением, так как данные могут записываться на разные шарды полностью независимо.

Во вторых, вы можете делать INSERT в Distributed таблицу. В этом случае, таблица будет сама распределять вставляемые данные по серверам.
Для того, чтобы писать в Distributed таблицу, у неё должен быть задан ключ шардирования (последний параметр). Также, если шард всего-лишь один, то запись работает и без указания ключа шардирования (так как в этом случае он не имеет смысла).

У каждого шарда в конфигурационном файле может быть задан "вес" (weight). По умолчанию, вес равен единице. Данные будут распределяться по шардам в количестве, пропорциональном весу шарда. Например, если есть два шарда, и у первого выставлен вес 9, а у второго 10, то на первый будет отправляться 9 / 19 доля строк, а на второй - 10 / 19.

У каждого шарда в конфигурационном файле может быть указан параметр internal_replication.

Если он выставлен в true, то для записи будет выбираться первая живая реплика и данные будут писаться на неё. Этот вариант следует использовать, если Distributed таблица "смотрит" на реплицируемые таблицы. То есть, если таблица, в которую будут записаны данные, будет сама заниматься их репликацией.

Если он выставлен в false (по умолчанию), то данные будут записываться на все реплики. По сути, это означает, что Distributed таблица занимается репликацией данных самостоятельно. Это хуже, чем использование реплицируемых таблиц, так как не контролируется консистентность реплик, и они со временем будут содержать немного разные данные.

Для выбора шарда, на который отправляется строка данных, вычисляется выражение шардирования, и берётся его остаток от деления на суммарный вес шардов. Строка отправляется на шард, соответствующий полуинтервалу остатков от prev_weights до prev_weights + weight, где prev_weights - сумма весов шардов с меньшим номером, а weight - вес этого шарда. Например, если есть два шарда, и у первого выставлен вес 9, а у второго 10, то строка будет отправляться на первый шард для остатков из диапазона [0, 9), а на второй - для остатков из диапазона [9, 19).

Выражением шардирование может быть произвольное выражение от констант и столбцов таблицы, возвращающее целое число. Например, вы можете использовать выражение rand() для случайного распределения данных, или UserID - для распределения по остатку от деления идентификатора посетителя (тогда данные одного посетителя будут расположены на одном шарде, что упростит выполнение IN и JOIN по посетителям). Если распределение какого-либо столбца недостаточно равномерное, вы можете обернуть его в хэш функцию: intHash64(UserID).

Простой остаток от деления является довольно ограниченным решением для шардирования и подходит не для всех случаев. Он подходит для среднего и большого объёма данных (десятки серверов), но не для очень больших объёмов данных (сотни серверов и больше). В последнем случае, лучше использовать схему шардирования, продиктованную требованиями предметной области, и не использовать возможность записи в Distributed таблицы.

Запросы SELECT отправляются на все шарды, и работают независимо от того, каким образом данные распределены по шардам (они могут быть распределены полностью случайно). При добавлении нового шарда, можно не переносить на него старые данные, а записывать новые данные с большим весом - данные будут распределены слегка неравномерно, но запросы будут работать корректно и достаточно эффективно.

Беспокоиться о схеме шардирования имеет смысл в следующих случаях:
- используются запросы, требующие соединение данных (IN, JOIN) по определённому ключу - тогда если данные шардированы по этому ключу, то можно использовать локальные IN, JOIN вместо GLOBAL IN, GLOBAL JOIN, что кардинально более эффективно.
- используется большое количество серверов (сотни и больше) и большое количество маленьких запросов (запросы отдельных клиентов - сайтов, рекламодателей, партнёров) - тогда, для того, чтобы маленькие запросы не затрагивали весь кластер, имеет смысл располагать данные одного клиента на одном шарде, или (вариант, который используется в Яндекс.Метрике) сделать двухуровневое шардирование: разбить весь кластер на "слои", где слой может состоять из нескольких шардов; данные для одного клиента располагаются на одном слое, но в один слой можно по мере необходимости добавлять шарды, в рамках которых данные распределены произвольным образом; создаются распределённые таблицы на каждый слой и одна общая распределённая таблица для глобальных запросов.

Запись данных осуществляется полностью асинхронно. При INSERT-е в Distributed таблицу, блок данных всего лишь записывается в локальную файловую систему. Данные отправляются на удалённые серверы в фоне, при первой возможности. Вы должны проверять, успешно ли отправляются данные, проверяя список файлов (данные, ожидающие отправки) в директории таблицы:
/var/lib/clickhouse/data/database/table/.

Если после INSERT-а в Distributed таблицу, сервер перестал существовать или был грубо перезапущен (например, в следствие аппаратного сбоя), то записанные данные могут быть потеряны. Если в директории таблицы обнаружен повреждённый кусок данных, то он переносится в поддиректорию broken и больше не используется.

При выставлении опции max_parallel_replicas выполнение запроса распараллеливается по всем репликам внутри одного шарда. Подробнее смотрите раздел "Настройки, max_parallel_replicas".

Внешние данные для обработки запроса

ClickHouse позволяет отправить на сервер данные, необходимые для обработки одного запроса, вместе с запросом SELECT. Такие данные будут положены во временную таблицу (см. раздел "Временные таблицы") и смогут использоваться в запросе (например, в операторах IN).

Для примера, если у вас есть текстовый файл с важными идентификаторами посетителей, вы можете загрузить его на сервер вместе с запросом, в котором используется фильтрация по этому списку.

Если вам нужно будет выполнить более одного запроса с достаточно большими внешними данными - лучше не использовать эту функциональность, а загрузить данные в БД заранее.

Внешние данные могут быть загружены как с помощью клиента командной строки (в неинтерактивном режиме), так и через HTTP-интерфейс.

В клиенте командной строки, может быть указана секция параметров вида

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

Таких секций может быть несколько - по числу передаваемых таблиц.

--external - маркер начала секции.
--file - путь к файлу с дампом таблицы, или -, что обозначает stdin.
Из stdin может быть считана только одна таблица.

Следующие параметры не обязательные:
--name - имя таблицы. Если не указано - используется _data.
--format - формат данных в файле. Если не указано - используется TabSeparated.

Должен быть указан один из следующих параметров:
--types - список типов столбцов через запятую. Например, `UInt64,String`. Столбцы будут названы _1, _2, ...
--structure - структура таблицы, в форме `UserID UInt64`, `URL String`. Определяет имена и типы столбцов.

Файлы, указанные в file, будут разобраны форматом, указанным в format, с использованием типов данных, указанных в types или structure. Таблица будет загружена на сервер, и доступна там в качестве временной таблицы с именем name.

Примеры:

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

При использовании HTTP интерфейса, внешние данные передаются в формате multipart/form-data. Каждая таблица передаётся отдельным файлом. Имя таблицы берётся из имени файла. В query_string передаются параметры name_format, name_types, name_structure, где name - имя таблицы, которой соответствуют эти параметры. Смысл параметров такой же, как при использовании клиента командной строки.

Пример:

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

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

При распределённой обработке запроса, временные таблицы передаются на все удалённые серверы.

Dictionary

Движок `Dictionary` отображает данные словаря как таблицу ClickHouse.

Рассмотрим для примера словарь `products` со следующей конфигурацией:

```
<dictionaries>
<dictionary>
      <name>products</name>
      <source>
         <odbc>
            <table>products</table>
```

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

Запрос данных словаря:

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

В таком виде данные из словаря можно получить при помощи функций dictGet*.

Такое представление неудобно, когда нам необходимо получить данные в чистом виде, а также при выполнении операции `JOIN`. Для этих случаев можно использовать движок `Dictionary`, который отобразит данные словаря в таблицу.

Синтаксис:

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

Пример использования:

```
create table products (product_id UInt64, title String) Engine = Dictionary(products);
```

```
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.
```

Проверим что у нас в таблице?

```
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

Движок `Merge` (не путайте с движком `MergeTree`) не хранит данные самостоятельно, а позволяет читать одновременно из произвольного количества других таблиц.
Чтение автоматически распараллеливается. Запись в таблицу не поддерживается. При чтении будут использованы индексы тех таблиц, из которых реально идёт чтение, если они существуют.
Движок `Merge` принимает параметры: имя базы данных и регулярное выражение для таблиц.

Пример:

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

Данные будут читаться из таблиц в базе `hits`, имена которых соответствуют регулярному выражению '`^WatchLog`'.

Вместо имени базы данных может использоваться константное выражение, возвращающее строку. Например, `currentDatabase()`.

Регулярные выражения — re2 (поддерживает подмножество PCRE), регистрозависимые.
Смотрите замечание об экранировании в регулярных выражениях в разделе "match".

При выборе таблиц для чтения, сама `Merge`-таблица не будет выбрана, даже если попадает под регулярное выражение, чтобы не возникло циклов.
Впрочем, вы можете создать две `Merge`-таблицы, которые будут пытаться бесконечно читать данные друг друга, но делать этого не нужно.

Типичный способ использования движка `Merge` — работа с большим количеством таблиц типа `TinyLog`, как с одной.

Пример 2:

Пусть есть старая таблица `WatchLog_old`. Необходимо изменить партиционирование без перемещения данных в новую таблицу `WatchLog_new`. При этом в выборке должны участвовать данные обеих таблиц.

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

Виртуальные столбцы

Виртуальные столбцы — столбцы, предоставляемые движком таблиц независимо от определения таблицы. То есть, такие столбцы не указываются в `CREATE TABLE`, но доступны для `SELECT`.

Виртуальные столбцы отличаются от обычных следующими особенностями:

- они не указываются в определении таблицы;
- в них нельзя вставить данные при `INSERT`;
- при `INSERT` без указания списка столбцов виртуальные столбцы не учитываются;
- они не выбираются при использовании звёздочки (`SELECT *`);
- виртуальные столбцы не показываются в запросах `SHOW CREATE TABLE` и `DESC TABLE`;

Таблица типа `Merge` содержит виртуальный столбец `_table` типа `String`. (Если в таблице уже есть столбец `_table`, то виртуальный столбец называется `_table1`; если уже есть `_table1`, то `_table2` и т. п.) Он содержит имя таблицы, из которой были прочитаны данные.

Если секция `WHERE/PREWHERE` содержит (в качестве одного из элементов конъюнкции или в качестве всего выражения) условия на столбец `_table`, не зависящие от других столбцов таблицы, то эти условия используются как индекс: условия выполняются над множеством имён таблиц, из которых нужно читать данные, и чтение будет производиться только из тех таблиц, для которых условия сработали.

File(Format)

Управляет данными в одном файле на диске в указанном формате.

Примеры применения:

- Выгрузка данных из ClickHouse в файл.
- Преобразование данных из одного формата в другой.
- Обновление данных в ClickHouse редактированием файла на диске.

Использование движка в сервере ClickHouse

```
File(Format)
```

`Format` должен быть таким, который ClickHouse может использовать и в запросах `INSERT` и в запросах

`SELECT`. Полный список поддерживаемых форматов смотрите в разделе Форматы.

Сервер ClickHouse не позволяет указать путь к файлу, с которым будет работать `File`. Используется путь к хранилищу, определенный параметром path в конфигурации сервера.

При создании таблицы с помощью `File(Format)` сервер ClickHouse создает в хранилище каталог с именем таблицы, а после добавления в таблицу данных помещает туда файл `data.Format`.

Можно вручную создать в хранилище каталог таблицы, поместить туда файл, затем на сервере ClickHouse добавить (ATTACH) информацию о таблице, соответствующей имени каталога и прочитать из файла данные.

Warning

Будьте аккуратны с этой функциональностью, поскольку сервер ClickHouse не отслеживает внешние изменения данных. Если в файл будет производиться запись одновременно со стороны сервера ClickHouse и с внешней стороны, то результат непредсказуем.

Пример:

1. Создадим на сервере таблицу `file_engine_table`:

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

В конфигурации по умолчанию сервер ClickHouse создаст каталог `/var/lib/clickhouse/data/default/file_engine_table`.

2. Вручную создадим файл `/var/lib/clickhouse/data/default/file_engine_table/data.TabSeparated` с содержимым:

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

3. Запросим данные:

```
SELECT * FROM file_engine_table
```

```
┌─name─┬─value─┐
│ one  │     1 │
│ two  │     2 │
└──────┴───────┘
```

Использование движка в clickhouse-local

В clickhouse-local движок в качестве параметра принимает не только формат, но и путь к файлу. В том числе можно указать стандартные потоки ввода/вывода цифровым или буквенным обозначением `0` или `stdin`, `1` или `stdout`.

Пример:

```
$ 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"
```

Детали реализации

- Поддерживается многопоточное чтение и однопоточная запись.
- Не поддерживается:

- использование операций `ALTER` и `SELECT...SAMPLE`;
- индексы;
- репликация.

Null

При записи в таблицу типа Null, данные игнорируются. При чтении из таблицы типа Null, возвращается пустота.

Тем не менее, есть возможность создать материализованное представление над таблицей типа Null. Тогда данные, записываемые в таблицу, будут попадать в представление.

Set

Представляет собой множество, постоянно находящееся в оперативке. Предназначено для использования в правой части оператора IN (смотрите раздел "Операторы IN").

В таблицу можно вставлять данные INSERT-ом - будут добавлены новые элементы в множество, с игнорированием дубликатов.
Но из таблицы нельзя, непосредственно, делать SELECT. Единственная возможность чтения - использование в правой части оператора IN.

Данные постоянно находятся в оперативке. При INSERT-е, в директорию таблицы на диске, также пишутся блоки вставленных данных. При запуске сервера, эти данные считываются в оперативку. То есть, после перезапуска, данные остаются на месте.

При грубом перезапуске сервера, блок данных на диске может быть потерян или повреждён. В последнем случае, может потребоваться вручную удалить файл с повреждёнными данными.

Join

Представляет собой подготовленную структуру данных для JOIN-а, постоянно находящуюся в оперативке.

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

Параметры движка: `ANY|ALL` - строгость, `LEFT|INNER` - тип.
Эти параметры (задаются без кавычек) должны соответствовать тому JOIN-у, для которого будет использоваться таблица. k1, k2, ... - ключевые столбцы из секции USING, по которым будет делаться соединение.

Таблица не может использоваться для GLOBAL JOIN-ов.

В таблицу можно вставлять данные INSERT-ом, аналогично движку Set. В случае ANY, данные для дублирующихся ключей будут проигнорированы; в случае ALL - будут учитываться. Из таблицы нельзя, непосредственно, делать SELECT. Единственная возможность чтения - использование в качестве "правой" таблицы для JOIN.

Хранение данных на диске аналогично движку Set.

URL(URL, Format)

Управляет данными на удаленном HTTP/HTTPS сервере. Данный движок похож
на движок File.

Использование движка в сервере ClickHouse

`Format` должен быть таким, который ClickHouse может использовать в запросах
`SELECT` и, если есть необходимость, `INSERT`. Полный список поддерживаемых форматов смотрите в разделе Форматы.

`URL` должен соответствовать структуре Uniform Resource Locator. По указанному URL должен находится сервер
работающий по протоколу HTTP или HTTPS. При этом не должно требоваться никаких дополнительных заголовков для получения ответа от сервера.

Запросы `INSERT` и `SELECT` транслируются в `POST` и `GET` запросы
соответственно. Для обработки `POST`-запросов удаленный сервер должен поддерживать Chunked transfer encoding.

Пример:

1. Создадим на сервере таблицу `url_engine_table`:

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

2. Создадим простейший http-сервер стандартными средствами языка python3 и
запустим его:

```
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. Запросим данные:

```
SELECT * FROM url_engine_table
```

```
┌─word──┬─value─┐
│ Hello │     1 │
│ World │     2 │
└───────┴───────┘
```

Особенности использования

- Поддерживается многопоточное чтение и запись.
- Не поддерживается:
  - использование операций `ALTER` и `SELECT...SAMPLE`;
  - индексы;
  - репликация.

View

Используется для реализации представлений (подробнее см. запрос `CREATE VIEW`). Не хранит данные, а хранит только указанный запрос `SELECT`. При чтении из таблицы, выполняет его (с удалением из запроса

всех ненужных столбцов).

MaterializedView

Используется для реализации материализованных представлений (подробнее см. запрос CREATE TABLE). Для хранения данных, использует другой движок, который был указан при создании представления. При чтении из таблицы, просто использует этот движок.

Memory

Хранит данные в оперативке, в несжатом виде. Данные хранятся именно в таком виде, в каком они получаются при чтении. То есть, само чтение из этой таблицы полностью бесплатно.
Конкурентный доступ к данным синхронизируется. Блокировки короткие: чтения и записи не блокируют друг друга.
Индексы не поддерживаются. Чтение распараллеливается.
За счёт отсутствия чтения с диска, разжатия и десериализации данных удаётся достичь максимальной производительности (выше 10 ГБ/сек.) на простых запросах. (Стоит заметить, что во многих случаях, производительность движка MergeTree, почти такая же высокая.)
При перезапуске сервера данные из таблицы исчезают и таблица становится пустой.
Обычно, использование этого движка таблиц является неоправданным. Тем не менее, он может использоваться для тестов, а также в задачах, где важно достичь максимальной скорости на не очень большом количестве строк (примерно до 100 000 000).

Движок Memory используется системой для временных таблиц - внешних данных запроса (смотрите раздел "Внешние данные для обработки запроса"), для реализации `GLOBAL IN` (смотрите раздел "Операторы IN").

Buffer

Буферизует записываемые данные в оперативке, периодически сбрасывая их в другую таблицу. При чтении, производится чтение данных одновременно из буфера и из другой таблицы.

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

Параметры движка:
database, table - таблица, в которую сбрасывать данные. Вместо имени базы данных может использоваться константное выражение, возвращающее строку.
num_layers - уровень параллелизма. Физически таблица будет представлена в виде num_layers независимых буферов. Рекомендуемое значение - 16.
min_time, max_time, min_rows, max_rows, min_bytes, max_bytes - условия для сброса данных из буфера.

Данные сбрасываются из буфера и записываются в таблицу назначения, если выполнены все min-условия или хотя бы одно max-условие.
min_time, max_time - условие на время в секундах от момента первой записи в буфер;
min_rows, max_rows - условие на количество строк в буфере;
min_bytes, max_bytes - условие на количество байт в буфере.

При записи, данные вставляются в случайный из num_layers буферов. Или, если размер куска вставляемых данных достаточно большой (больше max_rows или max_bytes), то он записывается в таблицу назначения минуя буфер.

Условия для сброса данных учитываются отдельно для каждого из num_layers буферов. Например, если num_layers = 16 и max_bytes = 100000000, то максимальный расход оперативки будет 1.6 GB.

Пример:

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

Создаём таблицу merge.hits_buffer такой же структуры как merge.hits и движком Buffer. При записи в эту таблицу, данные буферизуются в оперативке и, в дальнейшем, записываются в таблицу merge.hits. Создаётся 16 буферов. Данные, имеющиеся в каждом из них будут сбрасываться, если прошло сто секунд, или записан миллион строк, или записано сто мегабайт данных; или если одновременно прошло десять секунд и записано десять тысяч строк и записано десять мегабайт данных. Для примера, если записана всего лишь одна строка, то через сто секунд она будет сброшена в любом случае. А если записано много строк, то они будут сброшены раньше.

При остановке сервера, при DROP TABLE или DETACH TABLE, данные из буфера тоже сбрасываются в таблицу назначения.

В качестве имени базы данных и имени таблицы можно указать пустые строки в одинарных кавычках. Это обозначает отсутствие таблицы назначения. В таком случае, при достижении условий на сброс данных, буфер будет просто очищаться. Это может быть полезным, чтобы хранить в оперативке некоторое окно данных.

При чтении из таблицы типа Buffer, будут обработаны данные, как находящиеся в буфере, так и данные из таблицы назначения (если такая есть).
Но следует иметь ввиду, что таблица Buffer не поддерживает индекс. То есть, данные в буфере будут просканированы полностью, что может быть медленно для буферов большого размера. (Для данных в подчинённой таблице, будет использоваться тот индекс, который она поддерживает.)

Если множество столбцов таблицы Buffer не совпадает с множеством столбцов подчинённой таблицы, то будут вставлено подмножество столбцов, которое присутствует в обеих таблицах.

Если у одного из столбцов таблицы Buffer и подчинённой таблицы не совпадает тип, то в лог сервера будет записано сообщение об ошибке и буфер будет очищен.
То же самое происходит, если подчинённая таблица не существует в момент сброса буфера.

Если есть необходимость выполнить ALTER для подчинённой таблицы и для таблицы Buffer, то рекомендуется удалить таблицу Buffer, затем выполнить ALTER подчинённой таблицы, а затем создать таблицу Buffer заново.

При нештатном перезапуске сервера, данные, находящиеся в буфере, будут потеряны.

Для таблиц типа Buffer неправильно работают FINAL и SAMPLE. Эти условия пробрасываются в таблицу назначения, но не используются для обработки данных в буфере. В связи с этим, рекомендуется использовать таблицу типа Buffer только для записи, а читать из таблицы назначения.

При добавлении данных в Buffer, один из буферов блокируется. Это приводит к задержкам, если одновременно делается чтение из таблицы.

Данные, вставляемые в таблицу Buffer, попадают в подчинённую таблицу в порядке, возможно отличающимся от порядка вставки, и блоками, возможно отличающимися от вставленных блоков. В связи с этим, трудно корректно использовать таблицу типа Buffer для записи в CollapsingMergeTree. Чтобы избежать проблемы, можно выставить num_layers в 1.

Если таблица назначения является реплицируемой, то при записи в таблицу Buffer будут потеряны некоторые ожидаемые свойства реплицируемых таблиц. Из-за произвольного изменения порядка строк и размеров блоков данных, перестаёт работать дедупликация данных, в результате чего исчезает возможность надёжной exactly once записи в реплицируемые таблицы.

В связи с этими недостатками, таблицы типа Buffer можно рекомендовать к применению лишь в очень редких случаях.

Таблицы типа Buffer используются в тех случаях, когда от большого количества серверов поступает слишком много INSERT-ов в единицу времени, и нет возможности заранее самостоятельно буферизовать данные перед вставкой, в результате чего, INSERT-ы не успевают выполняться.

Заметим, что даже для таблиц типа Buffer не имеет смысла вставлять данные по одной строке, так как

Заметим, что даже для таблиц типа Buffer не имеет смысла вставлять данные по одной строке, так как таким образом будет достигнута скорость всего лишь в несколько тысяч строк в секунду, тогда как при вставке более крупными блоками, достижимо более миллиона строк в секунду (смотрите раздел "Производительность").

JDBC

Позволяет ClickHouse подключаться к внешним базам данных с помощью JDBC.

Для реализации соединения по JDBC ClickHouse использует отдельную программу clickhouse-jdbc-bridge, которая должна запускаться как демон.

Движок поддерживает тип данных Nullable.

Создание таблицы

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

Параметры движка

- `dbms_uri` — URI внешней СУБД.

  Формат: `jdbc:<driver_name>://<host_name>:<port>/?user=<username>&password=<password>`.
  Пример для MySQL: `jdbc:mysql://localhost:3306/?user=root&password=root`.

- `external_database` — база данных во внешней СУБД.

- `external_table` — таблица в `external_database`.

Пример использования

Создадим таблицу в на сервере MySQL с помощью консольного клиента 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)
```

Создадим таблицу на сервере ClickHouse и получим из неё данные:

```
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─┐
```

```
┌─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.
```

Смотрите также

- Табличная функция JDBC.

ODBC

Allows ClickHouse to connect to external databases via ODBC.

To implement ODBC connection, ClickHouse uses the separate program `clickhouse-odbc-bridge`. ClickHouse starts this program automatically when it is required. The ODBC bridge program is installed by the same package with the ClickHouse server.

This engine supports the Nullable data type.

Creating a Table

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

Engine Parameters

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

Usage Example

Some examples of how to use external dictionaries through ODBC you can find in the ODBC section of external dictionaries configuration.

See Also

- ODBC table function.

Справка по SQL

- SELECT
- INSERT INTO
- CREATE
- ALTER
- Прочие виды запросов

SELECT запросы

`SELECT` осуществляет выборку данных.

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

Все секции, кроме списка выражений сразу после SELECT, являются необязательными.
Ниже секции будут описаны в порядке, почти соответствующем конвейеру выполнения запроса.

Если в запросе отсутствуют секции `DISTINCT`, `GROUP BY`, `ORDER BY`, подзапросы в `IN` и `JOIN`, то запрос будет обработан полностью потоково, с использованием O(1) количества оперативки.
Иначе запрос может съесть много оперативки, если не указаны подходящие ограничения `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`. Подробнее смотрите в разделе "Настройки". Присутствует возможность использовать внешнюю сортировку (с сохранением временных данных на диск) и внешнюю агрегацию. `Merge join` в системе нет.

Секция FROM

Если секция FROM отсутствует, то данные будут читаться из таблицы `system.one`.
Таблица system.one содержит ровно одну строку (то есть, эта таблица выполняет такую же роль, как таблица DUAL, которую можно найти в других СУБД).

В секции FROM указывается таблица, из которой будут читаться данные, либо подзапрос, либо табличная функция; дополнительно могут присутствовать ARRAY JOIN и обычный JOIN (смотрите ниже).

Вместо таблицы, может быть указан подзапрос SELECT в скобках.
В этом случае, конвейер обработки подзапроса будет встроен в конвейер обработки внешнего запроса.
В отличие от стандартного SQL, после подзапроса не нужно указывать его синоним. Для совместимости, присутствует возможность написать AS name после подзапроса, но указанное имя нигде не используется.

Вместо таблицы, может быть указана табличная функция. Подробнее смотрите раздел "Табличные функции".

Для выполнения запроса, из соответствующей таблицы, вынимаются все столбцы, перечисленные в запросе. Из подзапросов выкидываются столбцы, не нужные для внешнего запроса.
Если в запросе не перечислено ни одного столбца (например, SELECT count() FROM t), то из таблицы всё равно вынимается один какой-нибудь столбец (предпочитается самый маленький), для того, чтобы можно было хотя бы посчитать количество строк.

Модификатор FINAL может быть использован только при SELECT-е из таблицы типа CollapsingMergeTree. При указании FINAL, данные будут выбираться полностью "сколлапсированными". Стоит учитывать, что использование FINAL приводит к выбору кроме указанных в SELECT-е столбцов также столбцов, относящихся к первичному ключу. Также, запрос будет выполняться в один поток, и при выполнении запроса будет выполняться слияние данных. Это приводит к тому, что при использовании FINAL, запрос выполняется медленнее. В большинстве случаев, следует избегать использования FINAL. Подробнее смотрите раздел "Движок CollapsingMergeTree".

Секция SAMPLE

Секция `SAMPLE` позволяет выполнять запросы приближённо. Например, чтобы посчитать статистику по всем визитам, можно обработать 1/10 всех визитов и результат домножить на 10.

Сэмплирование имеет смысл, когда:

1. Точность результата не важна, например, для оценочных расчетов.
2. Возможности аппаратной части не позволяют соответствовать строгим критериям. Например, время ответа должно быть <100 мс. При этом точность расчета имеет более низкий приоритет.
3. Точность результата участвует в бизнес-модели сервиса. Например, пользователи с бесплатной подпиской на сервис могут получать отчеты с меньшей точностью, чем пользователи с премиум подпиской.

Внимание

Не стоит использовать сэмплирование в тех задачах, где важна точность расчетов. Например, при работе с финансовыми отчетами.

Свойства сэмплирования:

- Сэмплирование работает детерминированно. При многократном выполнении одного и того же запроса `SELECT .. SAMPLE`, результат всегда будет одинаковым.
- Сэмплирование поддерживает консистентность для разных таблиц. Имеется в виду, что для таблиц с одним и тем же ключом сэмплирования, подмножество данных в выборках будет одинаковым (выборки при этом должны быть сформированы для одинаковой доли данных). Например, выборка по идентификаторам посетителей выберет из разных таблиц строки с одинаковым подмножеством всех возможных идентификаторов. Это свойство позволяет использовать выборки в подзапросах в секции IN, а также объединять выборки с помощью JOIN.
- Сэмплирование позволяет читать меньше данных с диска. Обратите внимание, для этого необходимо корректно указать ключ сэмплирования. Подробнее см. в разделе Создание таблицы MergeTree.

Сэмплирование поддерживается только таблицами семейства MergeTree и только в том случае, если для таблиц был указан ключ сэмплирования (выражение, на основе которого должна производиться выборка). Подробнее см. в разделе Создание таблиц MergeTree.

Выражение `SAMPLE` в запросе можно задать следующими способами:

Способ задания SAMPLE	Описание
`SAMPLE k`	Здесь `k` – это дробное число в интервале от 0 до 1. Запрос будет выполнен по `k` доле данных. Например, если указано `SAMPLE 1/10`, то запрос будет выполнен для выборки из 1/10 данных. Подробнее
`SAMPLE n`	Здесь `n` – это достаточно большое целое число.

Запрос будет выполнен для выборки, состоящей из не менее чем n строк. Например, если указано SAMPLE 10000000, то запрос будет выполнен для не менее чем 10,000,000 строк. Подробнее SAMPLE k OFFSET m Здесь k и m – числа от 0 до 1. Запрос будет выполнен по k доле данных. При этом выборка будет сформирована со смещением на m долю. Подробнее

SAMPLE k

Здесь k – число в интервале от 0 до 1. Поддерживается как дробная, так и десятичная форма записи. Например, SAMPLE 1/2 или SAMPLE 0.5.

Если задано выражение SAMPLE k, запрос будет выполнен для k доли данных. Рассмотрим пример:

```
SELECT
    Title,
    count() * 10 AS PageViews
FROM hits_distributed
SAMPLE 0.1
WHERE
    CounterID = 34
GROUP BY Title
ORDER BY PageViews DESC LIMIT 1000
```

В этом примере запрос выполняется по выборке из 0.1 (10%) данных. Значения агрегатных функций не корректируются автоматически, поэтому чтобы получить приближённый результат, значение count() нужно вручную умножить на 10.

Выборка с указанием относительного коэффициента является "согласованной": для таблиц с одним и тем же ключом сэмплирования, выборка с одинаковой относительной долей всегда будет составлять одно и то же подмножество данных. То есть выборка из разных таблиц, на разных серверах, в разное время, формируется одинаковым образом.

SAMPLE n

Здесь n – это достаточно большое целое число. Например, SAMPLE 10000000.

Если задано выражение SAMPLE n, запрос будет выполнен для выборки из не менее n строк (но не значительно больше этого значения). Например, если задать SAMPLE 10000000, в выборку попадут не менее 10,000,000 строк.

Примечание

Следует иметь в виду, что n должно быть достаточно большим числом. Так как минимальной единицей данных для чтения является одна гранула (её размер задаётся настройкой index_granularity для таблицы), имеет смысл создавать выборки, размер которых существенно превосходит размер гранулы.

При выполнении SAMPLE n коэффициент сэмплирования заранее неизвестен (то есть нет информации о том, относительно какого количества данных будет сформирована выборка). Чтобы узнать коэффициент сэмплирования, используйте столбец _sample_factor.

Виртуальный столбец _sample_factor автоматически создается в тех таблицах, для которых задано выражение SAMPLE BY (подробнее см. в разделе Создание таблицы MergeTree). В столбце содержится коэффициент сэмплирования для таблицы – он рассчитывается динамически по мере добавления данных в таблицу. Ниже приведены примеры использования столбца _sample_factor.

Предположим, у нас есть таблица, в которой ведется статистика посещений сайта. Пример ниже показывает, как рассчитать суммарное число просмотров:

```
SELECT sum(PageViews * _sample_factor)
FROM visits
SAMPLE 10000000
```

Следующий пример показывает, как посчитать общее число визитов:

```
SELECT sum(_sample_factor)
FROM visits
SAMPLE 10000000
```

В примере ниже рассчитывается среднее время на сайте. Обратите внимание, при расчете средних значений, умножать результат на коэффициент сэмплирования не нужно.

```
SELECT avg(Duration)
FROM visits
SAMPLE 10000000
```

SAMPLE k OFFSET m

Здесь k и m – числа в интервале от 0 до 1. Например, SAMPLE 0.1 OFFSET 0.5. Поддерживается как дробная, так и десятичная форма записи.

При задании SAMPLE k OFFSET m, выборка будет сформирована из k доли данных со смещением на долю m. Примеры приведены ниже.

Пример 1

```
SAMPLE 1/10
```

В этом примере выборка будет сформирована по 1/10 доле всех данных:

[++----------]

Пример 2

```
SAMPLE 1/10 OFFSET 1/2
```

Здесь выборка, которая состоит из 1/10 доли данных, взята из второй половины данных.

[----------++--------]

Секция ARRAY JOIN

Позволяет выполнить JOIN с массивом или вложенной структурой данных. Смысл похож на функцию arrayJoin, но функциональность более широкая.

```
SELECT <expr_list>
FROM <left_subquery>
[LEFT] ARRAY JOIN <array>
[WHERE|PREWHERE <expr>]
...
```

В запросе может быть указано не более одной секции ARRAY JOIN.

При использовании ARRAY JOIN, порядок выполнения запроса оптимизируется. Несмотря на то что секция ARRAY JOIN всегда указывается перед выражением WHERE / PREWHERE, преобразование JOIN может быть выполнено как до выполнения выражения WHERE / PREWHERE (если результат необходим в этом выражении), так и после (чтобы уменьшить объем расчетов). Порядок обработки контролируется оптимизатором запросов.

Секция ARRAY JOIN поддерживает следующие формы записи:

- ARRAY JOIN — в этом случае результат JOIN не будет содержать пустые массивы;
- LEFT ARRAY JOIN — пустые массивы попадут в результат выполнения JOIN. В качестве значения для пустых массивов устанавливается значение по умолчанию. Обычно это 0, пустая строка или NULL, в зависимости от типа элементов массива.

Рассмотрим примеры использования ARRAY JOIN и LEFT ARRAY JOIN. Для начала создадим таблицу, содержащую столбец с типом Array, и добавим в него значение:

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

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

```
┌─s───────────┬─arr─────┐
│ Hello       │ [1,2]   │
│ World       │ [3,4,5] │
│ Goodbye     │ []      │
└─────────────┴─────────┘
```

В примере ниже используется ARRAY JOIN:

```
SELECT s, arr
FROM arrays_test
ARRAY JOIN arr;
```

```
┌─s─────┬─arr─┐
│ Hello │   1 │
│ Hello │   2 │
│ World │   3 │
│ World │   4 │
│ World │   5 │
└───────┴─────┘
```

Следующий пример использует LEFT ARRAY JOIN:

```
SELECT s, arr
FROM arrays_test
LEFT ARRAY JOIN arr;
```

```
┌─s───────────┬─arr─┐
│ Hello       │   1 │
│ Hello       │   2 │
│ World       │   3 │
│ World       │   4 │
│ World       │   5 │
│ Goodbye     │   0 │
└─────────────┴─────┘
```

Использование алиасов

Для массива в секции ARRAY JOIN может быть указан алиас. В этом случае, элемент массива будет доступен под этим алиасом, а сам массив — под исходным именем. Пример:

```
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 │
└───────┴─────────┴───┘
```

Используя алиасы, можно выполнять JOIN с внешними массивами:

```
SELECT s, arr_external
FROM arrays_test
ARRAY JOIN [1, 2, 3] AS arr_external;
```

```
┌─s─────────┬─arr_external─┐
│ Hello     │            1 │
│ Hello     │            2 │
│ Hello     │            3 │
│ World     │            1 │
│ World     │            2 │
│ World     │            3 │
│ Goodbye   │            1 │
│ Goodbye   │            2 │
│ Goodbye   │            3 │
└───────────┴──────────────┘
```

В секции ARRAY JOIN можно указать через запятую сразу несколько массивов. В этом случае, JOIN делается с ними одновременно (прямая сумма, а не прямое произведение). Обратите внимание, массивы должны быть одинаковых размеров. Примеры:

```
SELECT s, arr, a, num, mapped
FROM arrays_test
ARRAY JOIN arr AS a, arrayEnumerate(arr) AS num, arrayMap(x -> x + 1, arr) AS mapped;
```

```
┌─s─────┬─arr─────┬─a─┬─num─┬─mapped─┐
│ Hello │ [1,2]   │ 1 │   1 │      2 │
│ Hello │ [1,2]   │ 2 │   2 │      3 │
│ World │ [3,4,5] │ 3 │   1 │      4 │
│ World │ [3,4,5] │ 4 │   2 │      5 │
│ World │ [3,4,5] │ 5 │   3 │      6 │
└───────┴─────────┴───┴─────┴────────┘
```

В примере ниже используется функция arrayEnumerate:

```
SELECT s, arr, a, num, arrayEnumerate(arr)
FROM arrays_test
ARRAY JOIN arr AS a, arrayEnumerate(arr) AS num;
```

```
┌─s─────┬─arr─────┬─a─┬─num─┬─arrayEnumerate(arr)─┐
│ Hello │ [1,2]   │ 1 │   1 │ [1,2]               │
│ Hello │ [1,2]   │ 2 │   2 │ [1,2]               │
│ World │ [3,4,5] │ 3 │   1 │ [1,2,3]             │
│ World │ [3,4,5] │ 4 │   2 │ [1,2,3]             │
│ World │ [3,4,5] │ 5 │   3 │ [1,2,3]             │
└───────┴─────────┴───┴─────┴─────────────────────┘
```

ARRAY JOIN с вложенными структурами данных

ARRAY JOIN также работает с вложенными структурами данных. Пример:

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

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

```
┌─s───────┬─nest.x──┬─nest.y─────┐
│ Hello   │ [1,2]   │ [10,20]    │
│ World   │ [3,4,5] │ [30,40,50] │
│ Goodbye │ []      │ []         │
└─────────┴─────────┴────────────┘
```

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

```
┌─s─────┬─nest.x─┬─nest.y─┐
│ Hello │      1 │     10 │
│ Hello │      2 │     20 │
│ World │      3 │     30 │
│ World │      4 │     40 │
│ World │      5 │     50 │
└───────┴────────┴────────┘
```

При указании имени вложенной структуры данных в ARRAY JOIN, смысл такой же, как ARRAY JOIN со всеми элементами-массивами, из которых она состоит. Пример:

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

Такой вариант тоже имеет смысл:

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

Алиас для вложенной структуры данных можно использовать, чтобы выбрать как результат JOIN-а, так и исходный массив. Пример:

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

Пример использования функции arrayEnumerate:

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

Секция JOIN

Соединяет данные в привычном для SQL JOIN смысле.

Примечание

Не связана с функциональностью с ARRAY JOIN.

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

Вместо <left_subquery> и <right_subquery> можно указать имена таблиц. Это эквивалентно подзапросу SELECT * FROM table, за исключением особого случая таблицы с движком Join – массива, подготовленного для присоединения.

Поддерживаемые типы JOIN

- INNER JOIN (or JOIN)
- LEFT JOIN (or LEFT OUTER JOIN)
- RIGHT JOIN (or RIGHT OUTER JOIN)
- FULL JOIN (or FULL OUTER JOIN)
- CROSS JOIN (or ,)

Смотрите описание стандартного SQL JOIN.

Строгость ANY или ALL

Если указано ALL, то при наличии в "правой" таблице нескольких соответствующих строк, данные будут размножены по количеству этих строк. Это нормальное поведение JOIN как в стандартном SQL.
Если указано ANY, то при наличии в "правой" таблице нескольких соответствующих строк, будет присоединена только первая попавшаяся. Если известно, что в "правой" таблице есть не более одной подходящей строки, то результаты ANY и ALL совпадают.

Чтобы задать значение строгости по умолчанию, используйте сессионный параметр join_default_strictness.

GLOBAL JOIN

При использовании обычного JOIN , запрос отправляется на удалённые серверы. На каждом из них выполняются подзапросы для формирования "правой" таблицы, и с этой таблицей выполняется соединение. То есть, "правая" таблица формируется на каждом сервере отдельно.

При использовании GLOBAL ... JOIN, сначала сервер-инициатор запроса запускает подзапрос для вычисления правой таблицы. Эта временная таблица передаётся на каждый удалённый сервер, и на них выполняются запросы с использованием переданных временных данных.

Будьте аккуратны при использовании GLOBAL . За дополнительной информацией обращайтесь в раздел Распределенные подзапросы.

Советы по использованию

Из подзапроса удаляются все столбцы, ненужные для JOIN.

При запуске JOIN, отсутствует оптимизация порядка выполнения по отношению к другим стадиям запроса. Соединение (поиск в "правой" таблице) выполняется до фильтрации в WHERE и до агрегации. Чтобы явно задать порядок вычислений, рекомендуется выполнять JOIN подзапроса с подзапросом.

Пример:

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

У подзапросов нет возможности задать имена и нет возможности их использовать для того, чтобы сослаться на столбец из конкретного подзапроса.
Требуется, чтобы столбцы, указанные в USING, назывались одинаково в обоих подзапросах, а остальные столбцы - по-разному. Изменить имена столбцов в подзапросах можно с помощью синонимов (в примере используются синонимы hits и visits).

В секции USING указывается один или несколько столбцов для соединения, что обозначает условие на равенство этих столбцов. Список столбцов задаётся без скобок. Более сложные условия соединения не поддерживаются.

"Правая" таблица (результат подзапроса) располагается в оперативной памяти. Если её не хватает, вы не сможете выполнить JOIN.

Каждый раз для выполнения запроса с одинаковым JOIN, подзапрос выполняется заново — результат не кэшируется. Это можно избежать, используя специальный движок таблиц Join, представляющий собой подготовленное множество для соединения, которое всегда находится в оперативке.

В некоторых случаях более эффективно использовать IN вместо JOIN.
Среди разных типов JOIN, наиболее эффективен ANY LEFT JOIN, следующий по эффективности ANY INNER JOIN.
Наименее эффективны ALL LEFT JOIN и ALL INNER JOIN.

Если JOIN необходим для соединения с таблицами измерений (dimension tables - сравнительно небольшие таблицы, которые содержат свойства измерений - например, имена для рекламных кампаний), то использование JOIN может быть не очень удобным из-за громоздкости синтаксиса, а также из-за того, что правая таблица читается заново при каждом запросе. Специально для таких случаев существует функциональность "Внешние словари", которую следует использовать вместо JOIN. Дополнительные сведения смотрите в разделе Внешние словари.

Обработка пустых ячеек и NULL

При слиянии таблиц могут появляться пустые ячейки. То, каким образом ClickHouse заполняет эти ячейки, определяется настройкой join_use_nulls.

Если ключами JOIN выступают поля типа Nullable, то строки, где хотя бы один из ключей имеет значение NULL, не соединяются.

Секция WHERE

Позволяет задать выражение, которое ClickHouse использует для фильтрации данных перед всеми другими действиями в запросе кроме выражений, содержащихся в секции PREWHERE. Обычно, это выражение с логическими операторами.

Результат выражения должен иметь тип UInt8.

ClickHouse использует в выражении индексы, если это позволяет движок таблицы.

Если в секции необходимо проверить NULL, то используйте операторы IS NULL и IS NOT NULL, а также соответствующие функции isNull и isNotNull. В противном случае выражение будет считаться всегда не выполненным.

Пример проверки на NULL:

```
:) SELECT * FROM t_null WHERE y IS NULL

SELECT *
FROM t_null
WHERE isNull(y)

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

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

Секция PREWHERE

Имеет такой же смысл, как и секция WHERE. Отличие состоит в том, какие данные читаются из таблицы. При использовании PREWHERE, из таблицы сначала читаются только столбцы, необходимые для выполнения PREWHERE. Затем читаются остальные столбцы, нужные для выполнения запроса, но из них только те блоки, в которых выражение в PREWHERE истинное.

PREWHERE имеет смысл использовать, если есть условия фильтрации, которые использует меньшинство столбцов из тех, что есть в запросе, но достаточно сильно фильтрует данные. Таким образом, сокращается количество читаемых данных.

Например, полезно писать PREWHERE для запросов, которые вынимают много столбцов, но в которых фильтрация производится лишь по нескольким столбцам.

PREWHERE поддерживается только таблицами семейства *MergeTree.

В запросе могут быть одновременно указаны секции PREWHERE и WHERE. В этом случае, PREWHERE идёт перед WHERE.

Если настройка optimize_move_to_prewhere выставлена в 1, то при отсутствии PREWHERE, система будет автоматически переносить части выражений из WHERE в PREWHERE согласно некоторой эвристике.

Секция GROUP BY

Это одна из наиболее важных частей СУБД.

Секция GROUP BY, если есть, должна содержать список выражений. Каждое выражение далее будем называть "ключом".
При этом, все выражения в секциях SELECT, HAVING, ORDER BY, должны вычисляться из ключей или из агрегатных функций. То есть, каждый выбираемый из таблицы столбец, должен использоваться либо в ключах, либо внутри агрегатных функций.

Если запрос содержит столбцы таблицы только внутри агрегатных функций, то секция GROUP BY может не указываться, и подразумевается агрегация по пустому набору ключей.

Пример:

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

Но, в отличие от стандартного SQL, если в таблице нет строк (вообще нет или после фильтрации с помощью WHERE), в качестве результата возвращается пустой результат, а не результат из одной строки, содержащий "начальные" значения агрегатных функций.

В отличие от MySQL (и в соответствии со стандартом SQL), вы не можете получить какое-нибудь значение некоторого столбца, не входящего в ключ или агрегатную функцию (за исключением константных выражений). Для обхода этого вы можете воспользоваться агрегатной функцией any (получить первое попавшееся значение) или min/max.

Пример:

```
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 вычисляет для каждого встретившегося различного значения ключей, набор значений агрегатных функций.

Не поддерживается GROUP BY по столбцам-массивам.

Не поддерживается указание констант в качестве аргументов агрегатных функций. Пример: sum(1). Вместо этого, вы можете избавиться от констант. Пример: count().

Обработка NULL

При группировке, ClickHouse рассматривает NULL как значение, причём NULL=NULL.

Рассмотрим, что это значит на примере.

Пусть есть таблица:

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

В результате запроса `SELECT sum(x), y FROM t_null_big GROUP BY y` мы получим:

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

Видно, что `GROUP BY` для `y = NULL` просуммировал `x`, как будто `NULL` — это значение.

Если в `GROUP BY` передать несколько ключей, то в результате мы получим все комбинации выборки, как если бы `NULL` был конкретным значением.

Модификатор WITH TOTALS

Если указан модификатор WITH TOTALS, то будет посчитана ещё одна строчка, в которой в столбцах-ключах будут содержаться значения по умолчанию (нули, пустые строки), а в столбцах агрегатных функций - значения, посчитанные по всем строкам ("тотальные" значения).

Эта дополнительная строчка выводится в форматах JSON*, TabSeparated*, Pretty* отдельно от остальных строчек. В остальных форматах эта строчка не выводится.

В форматах JSON* строчка выводится отдельным полем totals. В форматах TabSeparated* строчка выводится после основного результата, и перед ней (после остальных данных) вставляется пустая строка. В форматах Pretty* строчка выводится отдельной табличкой после основного результата.

`WITH TOTALS` может выполняться по-разному при наличии HAVING. Поведение зависит от настройки totals_mode.
По умолчанию `totals_mode = 'before_having'`. В этом случае totals считается по всем строчкам, включая непрошедших через HAVING и max_rows_to_group_by.

Остальные варианты учитывают в totals только строчки, прошедшие через HAVING, и имеют разное поведение при наличии настройки `max_rows_to_group_by` и `group_by_overflow_mode = 'any'`.

`after_having_exclusive` - не учитывать строчки, не прошедшие `max_rows_to_group_by`. То есть в totals попадёт меньше или столько же строчек, чем если бы `max_rows_to_group_by` не было.

`after_having_inclusive` - учитывать в totals все строчки, не прошедшие max_rows_to_group_by. То есть в totals попадёт больше или столько же строчек, чем если бы `max_rows_to_group_by` не было.

`after_having_auto` - считать долю строчек, прошедших через HAVING. Если она больше некоторого значения (по умолчанию - 50%), то включить все строчки, не прошедшие max_rows_to_group_by в totals, иначе - не включить.

`totals_auto_threshold` - по умолчанию 0.5. Коэффициент для работы `after_having_auto`.

Если `max_rows_to_group_by` и `group_by_overflow_mode = 'any'` не используются, то все варианты вида `after_having` не отличаются, и вы можете использовать любой из них, например, `after_having_auto`.

Вы можете использовать WITH TOTALS в подзапросах, включая подзапросы в секции JOIN (в этом случае соответствующие тотальные значения будут соединены).

GROUP BY во внешней памяти

Существует возможность включить сброс временных данных на диск для ограничения потребления оперативной памяти при GROUP BY.
Настройка max_bytes_before_external_group_by - потребление оперативки, при котором временные данные GROUP BY сбрасываются в файловую систему. Если равно 0 (по умолчанию) - значит выключено.

При использовании max_bytes_before_external_group_by рекомендуется выставить max_memory_usage примерно в два раза больше. Это следует сделать, потому что агрегация выполняется в две стадии: чтение и формирование промежуточных данных (1) и слияние промежуточных данных (2). Сброс данных на файловую систему может производиться только на стадии 1. Если сброса временных данных не было, то на стадии 2 может потребляться до такого же объёма памяти, как на стадии 1.

Например, если у вас max_memory_usage было выставлено в 10000000000, и вы хотите использовать внешнюю агрегацию, то имеет смысл выставить max_bytes_before_external_group_by в 10000000000, а max_memory_usage в 20000000000. При срабатывании внешней агрегации (если был хотя бы один сброс временных данных в файловую систему) максимальное потребление оперативки будет лишь чуть-чуть больше max_bytes_before_external_group_by.

При распределённой обработке запроса внешняя агрегация производится на удалённых серверах. Для того чтобы на сервере-инициаторе запроса использовалось немного оперативки, нужно выставить настройку distributed_aggregation_memory_efficient в 1.

При слиянии данных, сброшенных на диск, а также при слиянии результатов с удалённых серверов, при включенной настройке distributed_aggregation_memory_efficient, потребляется до 1/256 * количество потоков от общего объёма оперативки.

При включенной внешней агрегации, если данных было меньше max_bytes_before_external_group_by (то есть сброса данных не было), то запрос работает так же быстро, как без внешней агрегации. Если же какие-то временные данные были сброшены, то время выполнения будет в несколько раз больше (примерно в три раза).

Если после GROUP BY у вас есть ORDER BY с небольшим LIMIT, то на ORDER BY не будет тратиться существенного количества оперативки.
Но если есть ORDER BY без LIMIT, то не забудьте включить внешнюю сортировку (max_bytes_before_external_sort).

Секция LIMIT N BY

LIMIT N BY COLUMNS выбирает топ N строк для каждой группы COLUMNS. LIMIT N BY не связан с LIMIT и они могут использоваться в одном запросе. Ключ для LIMIT N BY может содержать произвольное число колонок или выражений.

Пример:

```
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
```

Запрос выберет топ 5 рефереров для каждой пары domain, device_type, но не более 100 строк (LIMIT n BY + LIMIT).

LIMIT n BY работает с NULL как если бы это было конкретное значение. Т.е. в результате запроса пользователь получит все комбинации полей, указанных в BY.

Секция HAVING

Позволяет отфильтровать результат, полученный после GROUP BY, аналогично секции WHERE.
WHERE и HAVING отличаются тем, что WHERE выполняется до агрегации (GROUP BY), а HAVING - после.
Если агрегации не производится, то HAVING использовать нельзя.

Секция ORDER BY

Секция ORDER BY содержит список выражений, к каждому из которых также может быть приписано DESC или ASC (направление сортировки). Если ничего не приписано - это аналогично приписыванию ASC. ASC - сортировка по возрастанию, DESC - сортировка по убыванию. Обозначение направления сортировки действует на одно выражение, а не на весь список. Пример: `ORDER BY Visits DESC, SearchPhrase`

Для сортировки по значениям типа String есть возможность указать collation (сравнение). Пример: `ORDER BY SearchPhrase COLLATE 'tr'` - для сортировки по поисковой фразе, по возрастанию, с учётом турецкого алфавита, регистронезависимо, при допущении, что строки в кодировке UTF-8. COLLATE может быть указан или не указан для каждого выражения в ORDER BY независимо. Если есть ASC или DESC, то COLLATE указывается после них. При использовании COLLATE сортировка всегда регистронезависима.

Рекомендуется использовать COLLATE только для окончательной сортировки небольшого количества строк, так как производительность сортировки с указанием COLLATE меньше, чем обычной сортировки по байтам.

Строки, для которых список выражений, по которым производится сортировка, принимает одинаковые значения, выводятся в произвольном порядке, который может быть также недетерминированным (каждый раз разным).
Если секция ORDER BY отсутствует, то, аналогично, порядок, в котором идут строки, не определён, и может быть недетерминированным.

Порядок сортировки `NaN` и `NULL`:

- С модификатором `NULLS FIRST` — Сначала `NULL`, затем `NaN`, затем остальные значения.
- С модификатором `NULLS LAST` — Сначала значения, затем `NaN`, затем `NULL`.
- По умолчанию — Как с модификатором `NULLS LAST`.

Пример:

Для таблицы

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

Выполним запрос `SELECT * FROM t_null_nan ORDER BY y NULLS FIRST`, получим:

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

Если кроме ORDER BY указан также не слишком большой LIMIT, то расходуется меньше оперативки. Иначе расходуется количество памяти, пропорциональное количеству данных для сортировки. При распределённой обработке запроса, если отсутствует GROUP BY, сортировка частично делается на удалённых серверах, а на сервере-инициаторе запроса производится слияние результатов. Таким образом, при распределённой сортировке, может сортироваться объём данных, превышающий размер памяти на одном сервере.

Существует возможность выполнять сортировку во внешней памяти (с созданием временных файлов на диске), если оперативной памяти не хватает. Для этого предназначена настройка max_bytes_before_external_sort. Если она выставлена в 0 (по умолчанию), то внешняя сортировка выключена. Если она включена, то при достижении объёмом данных для сортировки указанного количества байт, накопленные данные будут отсортированы и сброшены во временный файл. После того, как все данные будут прочитаны, будет произведено слияние всех сортированных файлов и выдача результата. Файлы записываются в директорию /var/lib/clickhouse/tmp/ (по умолчанию, может быть изменено с помощью параметра tmp_path) в конфиге.

На выполнение запроса может расходоваться больше памяти, чем max_bytes_before_external_sort. Поэтому, значение этой настройки должно быть существенно меньше, чем max_memory_usage. Для примера, если на вашем сервере 128 GB оперативки, и вам нужно выполнить один запрос, то выставите max_memory_usage в 100 GB, а max_bytes_before_external_sort в 80 GB.

Внешняя сортировка работает существенно менее эффективно, чем сортировка в оперативке.

Секция SELECT

После вычислений, соответствующих всем перечисленным выше секциям, производится вычисление выражений, указанных в секции SELECT.
Вернее, вычисляются выражения, стоящие над агрегатными функциями, если есть агрегатные функции.
Сами агрегатные функции и то, что под ними, вычисляются при агрегации (GROUP BY).
Эти выражения работают так, как будто применяются к отдельным строкам результата.

Секция DISTINCT

Если указано DISTINCT, то из всех множеств полностью совпадающих строк результата, будет оставляться только одна строка.
Результат выполнения будет таким же, как если указано GROUP BY по всем указанным полям в SELECT и не указаны агрегатные функции. Но имеется несколько отличий от GROUP BY:

- DISTINCT может применяться совместно с GROUP BY;
- при отсутствии ORDER BY и наличии LIMIT, запрос прекратит выполнение сразу после того, как будет прочитано необходимое количество различных строк - в этом случае использование DISTINCT существенно более оптимально;
- блоки данных будут выдаваться по мере их обработки, не дожидаясь выполнения всего запроса.

DISTINCT не поддерживается, если в SELECT присутствует хотя бы один столбец типа массив.

DISTINCT работает с NULL как если бы NULL был конкретным значением, причём NULL=NULL. Т.е. в результате DISTINCT разные комбинации с NULL встретятся только по одному разу.

Секция LIMIT

LIMIT m позволяет выбрать из результата первые m строк.

LIMIT n, m позволяет выбрать из результата первые m строк после пропуска первых n строк. Синтаксис LIMIT m OFFSET n также поддерживается.

n и m должны быть неотрицательными целыми числами.

При отсутствии секции ORDER BY, однозначно сортирующей результат, результат может быть произвольным и может являться недетерминированным.

Секция UNION ALL

Произвольное количество запросов может быть объединено с помощью UNION ALL. Пример:

```
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
```

Поддерживается только UNION ALL. Обычный UNION (UNION DISTINCT) не поддерживается. Если вам нужен UNION DISTINCT, то вы можете написать SELECT DISTINCT из подзапроса, содержащего UNION ALL.

Запросы - части UNION ALL могут выполняться параллельно, и их результаты могут возвращаться вперемешку.

Структура результатов (количество и типы столбцов) у запросов должна совпадать. Но имена столбцов могут отличаться. В этом случае, имена столбцов для общего результата будут взяты из первого запроса. При объединении выполняется приведение типов. Например, если в двух объединяемых запросах одно и тоже поле имеет типы не-Nullable и Nullable от совместимого типа, то в результате UNION ALL получим поле типа Nullable.

Запросы - части UNION ALL нельзя заключить в скобки. ORDER BY и LIMIT применяются к отдельным запросам, а не к общему результату. Если вам нужно применить какое-либо преобразование к общему результату, то вы можете разместить все запросы с UNION ALL в подзапросе в секции FROM.

Секция INTO OUTFILE

При указании INTO OUTFILE filename (где filename - строковый литерал), результат запроса будет сохранён в файл filename.
В отличие от MySQL, файл создаётся на стороне клиента. Если файл с таким именем уже существует, это приведёт к ошибке.
Функциональность доступна в клиенте командной строки и clickhouse-local (попытка выполнить запрос с INTO OUTFILE через HTTP интерфейс приведёт к ошибке).

Формат вывода по умолчанию - TabSeparated, как и в неинтерактивном режиме клиента командной строки.

Секция FORMAT

При указании FORMAT format вы можете получить данные в любом указанном формате.
Это может использоваться для удобства или для создания дампов.
Подробнее смотрите раздел "Форматы".
Если секция FORMAT отсутствует, то используется формат по умолчанию, который зависит от используемого интерфейса для доступа к БД и от настроек. Для HTTP интерфейса, а также для клиента командной строки, используемого в batch-режиме, по умолчанию используется формат TabSeparated. Для клиента командной строки, используемого в интерактивном режиме, по умолчанию используется формат PrettyCompact (прикольные таблички, компактные).

При использовании клиента командной строки данные на клиент передаются во внутреннем эффективном формате. При этом клиент самостоятельно интерпретирует секцию FORMAT запроса и форматирует данные на своей стороне (снимая нагрузку на сеть и сервер).

Операторы IN

Операторы IN, NOT IN, GLOBAL IN, GLOBAL NOT IN рассматриваются отдельно, так как их функциональность достаточно богатая.

В качестве левой части оператора, может присутствовать как один столбец, так и кортеж.

Примеры:

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

Если слева стоит один столбец, входящий в индекс, а справа - множество констант, то при выполнении запроса, система воспользуется индексом.

Не перечисляйте слишком большое количество значений (миллионы) явно. Если множество большое - лучше загрузить его во временную таблицу (например, смотрите раздел "Внешние данные для обработки запроса"), и затем воспользоваться подзапросом.

В качестве правой части оператора может быть множество константных выражений, множество кортежей с константными выражениями (показано в примерах выше), а также имя таблицы или подзапрос SELECT в скобках.

Если в качестве правой части оператора указано имя таблицы (например, UserID IN users), то это эквивалентно подзапросу UserID IN (SELECT * FROM users). Это используется при работе с внешними данными, отправляемым вместе с запросом. Например, вместе с запросом может быть отправлено множество идентификаторов посетителей, загруженное во временную таблицу users, по которому следует выполнить фильтрацию.

Если в качестве правой части оператора, указано имя таблицы, имеющий движок Set (подготовленное множество, постоянно находящееся в оперативке), то множество не будет создаваться заново при каждом запросе.

В подзапросе может быть указано более одного столбца для фильтрации кортежей.
Пример:

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

Типы столбцов слева и справа оператора IN, должны совпадать.

Оператор IN и подзапрос могут встречаться в любой части запроса, в том числе в агрегатных и лямбда функциях.
Пример:

```
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 │
└────────────┴──────────┘
```

за каждый день после 17 марта считаем долю хитов, сделанных посетителями, которые заходили на сайт 17 марта.
Подзапрос в секции IN на одном сервере всегда выполняется только один раз. Зависимых подзапросов не существует.

Обработка NULL

При обработке запроса оператор IN будет считать, что результат операции с NULL всегда равен 0, независимо от того, находится NULL в правой или левой части оператора. Значения NULL не входят ни в какое множество, не соответствуют друг другу и не могут сравниваться.

Рассмотрим для примера таблицу t_null:

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

При выполнении запроса SELECT x FROM t_null WHERE y IN (NULL,3) получим следующий результат:

```
┌─x─┐
│ 2 │
└───┘
```

Видно, что строка, в которой y = NULL, выброшена из результатов запроса. Это произошло потому, что ClickHouse не может решить входит ли NULL в множество (NULL,3), возвращает результат операции 0, а SELECT выбрасывает эту строку из финальной выдачи.

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

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

Распределённые подзапросы

Существует два варианта IN-ов с подзапросами (аналогично для JOIN-ов): обычный IN / JOIN и GLOBAL IN / GLOBAL JOIN. Они отличаются способом выполнения при распределённой обработке запроса.

Attention

Помните, что алгоритмы, описанные ниже, могут работать иначе в зависимости от настройки distributed_product_mode.

При использовании обычного IN-а, запрос отправляется на удалённые серверы, и на каждом из них выполняются подзапросы в секциях IN / JOIN.

При использовании GLOBAL IN / GLOBAL JOIN-а, сначала выполняются все подзапросы для GLOBAL IN / GLOBAL JOIN-ов, и результаты складываются во временные таблицы. Затем эти временные таблицы передаются на каждый удалённый сервер, и на них выполняются запросы, с использованием этих переданных временных данных.

Если запрос не распределённый, используйте обычный IN / JOIN.

Следует быть внимательным при использовании подзапросов в секции IN / JOIN в случае распределённой обработки запроса.

Рассмотрим это на примерах. Пусть на каждом сервере кластера есть обычная таблица **local_table**. Пусть также есть таблица **distributed_table** типа **Distributed**, которая смотрит на все серверы кластера.

При запросе к распределённой таблице **distributed_table**, запрос будет отправлен на все удалённые серверы, и на них будет выполнен с использованием таблицы **local_table**.

Например, запрос

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

будет отправлен на все удалённые серверы в виде

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

, выполнен параллельно на каждом из них до стадии, позволяющей объединить промежуточные результаты; затем промежуточные результаты вернутся на сервер-инициатор запроса, будут на нём объединены, и финальный результат будет отправлен клиенту.

Теперь рассмотрим запрос с IN-ом:

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

- расчёт пересечения аудиторий двух сайтов.

Этот запрос будет отправлен на все удалённые серверы в виде

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

То есть, множество в секции IN будет собрано на каждом сервере независимо, только по тем данным, которые есть локально на каждом из серверов.

Это будет работать правильно и оптимально, если вы предусмотрели такой случай, и раскладываете данные по серверам кластера таким образом, чтобы данные одного UserID-а лежали только на одном сервере. В таком случае все необходимые данные будут присутствовать на каждом сервере локально. В противном случае результат будет посчитан неточно. Назовём этот вариант запроса "локальный IN".

Чтобы исправить работу запроса, когда данные размазаны по серверам кластера произвольным образом, можно было бы указать **distributed_table** внутри подзапроса. Запрос будет выглядеть так:

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

Этот запрос будет отправлен на все удалённые серверы в виде

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

На каждом удалённом сервере начнёт выполняться подзапрос. Так как в подзапросе используется распределённая таблица, то подзапрос будет, на каждом удалённом сервере, снова отправлен на каждый удалённый сервер, в виде

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

Например, если у вас кластер из 100 серверов, то выполнение всего запроса потребует 10 000 элементарных запросов, что, как правило, является неприемлемым.

В таких случаях всегда следует использовать GLOBAL IN вместо IN. Рассмотрим его работу для запроса

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

На сервере-инициаторе запроса будет выполнен подзапрос

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

, и результат будет сложен во временную таблицу в оперативке. Затем запрос будет отправлен на каждый удалённый сервер в виде

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

, и вместе с запросом, на каждый удалённый сервер будет отправлена временная таблица _data1 (имя временной таблицы - implementation defined).

Это гораздо более оптимально, чем при использовании обычного IN. Но при этом, следует помнить о нескольких вещах:

1. При создании временной таблицы данные не уникализируются. Чтобы уменьшить объём передаваемых по сети данных, укажите в подзапросе DISTINCT (для обычного IN-а этого делать не нужно).
2. Временная таблица будет передана на все удалённые серверы. Передача не учитывает топологию сети. Например, если 10 удалённых серверов расположены в удалённом относительно сервера-инициатора запроса датацентре, то по каналу в удалённый датацентр данные будет переданы 10 раз. Старайтесь не использовать большие множества при использовании GLOBAL IN.
3. При передаче данных на удалённые серверы не настраивается ограничение использования сетевой полосы. Вы можете перегрузить сеть.
4. Старайтесь распределять данные по серверам так, чтобы в GLOBAL IN-ах не было частой необходимости.
5. Если в GLOBAL IN есть частая необходимость, то спланируйте размещение кластера ClickHouse таким образом, чтобы в каждом датацентре была хотя бы одна реплика каждого шарда, и среди них была быстрая сеть - чтобы запрос целиком можно было бы выполнить, передавая данные в пределах одного датацентра.

В секции GLOBAL IN также имеет смысл указывать локальную таблицу - в случае, если эта локальная таблица есть только на сервере-инициаторе запроса, и вы хотите воспользоваться данными из неё на удалённых серверах.

Экстремальные значения

Вы можете получить в дополнение к результату также минимальные и максимальные значения по столбцам результата. Для этого выставите настройку **extremes** в 1. Минимумы и максимумы считаются для числовых типов, дат, дат-с-временем. Для остальных столбцов будут выведены значения по умолчанию.

Вычисляются дополнительные две строчки - минимумы и максимумы, соответственно. Эти дополнительные две строчки выводятся в форматах JSON*, TabSeparated*, Pretty* отдельно от остальных строчек. В остальных форматах они не выводится.

В форматах JSON* экстремальные значения выводятся отдельным полем extremes. В форматах TabSeparated* строчка выводится после основного результата и после totals, если есть. Перед ней (после остальных данных) вставляется пустая строка. В форматах Pretty* строчка выводится отдельной табличкой после основного результата и после totals, если есть.

Экстремальные значения считаются по строчкам, прошедшим через LIMIT. Но при этом, при использовании LIMIT offset, size, строчки до offset учитываются в extremes. В потоковых запросах, в результате может учитываться также небольшое количество строчек, прошедших LIMIT.

Замечания

В секциях GROUP BY, ORDER BY, в отличие от диалекта MySQL, и в соответствии со стандартным SQL, не поддерживаются позиционные аргументы.
Например, если вы напишите GROUP BY 1, 2 - то это будет воспринято, как группировка по константам (то есть, агрегация всех строк в одну).

Вы можете использовать синонимы (алиасы AS) в любом месте запроса.

В любом месте запроса, вместо выражения, может стоять звёздочка. При анализе запроса звёздочка раскрывается в список всех столбцов таблицы (за исключением MATERIALIZED и ALIAS столбцов). Есть лишь немного случаев, когда оправдано использовать звёздочку:

- при создании дампа таблицы;
- для таблиц, содержащих всего несколько столбцов - например, системных таблиц;
- для получения информации о том, какие столбцы есть в таблице; в этом случае, укажите LIMIT 1. Но лучше используйте запрос DESC TABLE;
- при наличии сильной фильтрации по небольшому количеству столбцов с помощью PREWHERE;
- в подзапросах (так как из подзапросов выкидываются столбцы, не нужные для внешнего запроса).

В других случаях использование звёздочки является издевательством над системой, так как вместо преимуществ столбцовой СУБД вы получаете недостатки. То есть использовать звёздочку не рекомендуется.

INSERT

Добавление данных.

Базовый формат запроса:

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

В запросе можно указать список столбцов для вставки [(c1, c2, c3)]. В этом случае, в остальные столбцы записываются:

- Значения, вычисляемые из DEFAULT выражений, указанных в определении таблицы.
- Нули и пустые строки, если DEFAULT не определены.

Если strict_insert_defaults=1, то столбцы, для которых не определены DEFAULT, необходимо перечислить в запросе.

В INSERT можно передавать данные любого формата, который поддерживает ClickHouse. Для этого формат необходимо указать в запросе в явном виде:

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

Например, следующий формат запроса идентичен базовому варианту INSERT ... VALUES:

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

ClickHouse отсекает все пробелы и один перенос строки (если он есть) перед данными. Рекомендуем при формировании запроса переносить данные на новую строку после операторов запроса (это важно, если данные начинаются с пробелов).

Пример:

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

С помощью консольного клиента или HTTP интерфейса можно вставлять данные отдельно от запроса. Как это сделать, читайте в разделе "Интерфейсы".

Вставка результатов SELECT

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

Соответствие столбцов определяется их позицией в секции SELECT. При этом, их имена в выражении SELECT и в таблице для INSERT, могут отличаться. При необходимости выполняется приведение типов данных, эквивалентное соответствующему оператору CAST.

Все форматы данных кроме Values не позволяют использовать в качестве значений выражения, такие как now(), 1 + 2 и подобные. Формат Values позволяет ограниченно использовать выражения, но это не рекомендуется, так как в этом случае для их выполнения используется неэффективный вариант кода.

Не поддерживаются другие запросы на модификацию части данных: UPDATE, DELETE, REPLACE, MERGE, UPSERT, INSERT UPDATE.
Вы можете удалять старые данные с помощью запроса ALTER TABLE ... DROP PARTITION.

Замечания о производительности

INSERT сортирует входящие данные по первичному ключу и разбивает их на партиции по месяцам. Если вы вставляете данные за разные месяцы вперемешку, то это может значительно снизить производительность запроса INSERT. Чтобы избежать этого:

- Добавляйте данные достаточно большими пачками. Например, по 100 000 строк.
- Группируйте данные по месяцам самостоятельно перед загрузкой в ClickHouse.

Снижения производительности не будет, если:

- Данные поступают в режиме реального времени.
- Вы загружаете данные, которые как правило отсортированы по времени.

CREATE DATABASE

Создание базы данных db_name.

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

База данных - это просто директория для таблиц.
Если написано IF NOT EXISTS, то запрос не будет возвращать ошибку, если база данных уже существует.

CREATE TABLE

Запрос CREATE TABLE может иметь несколько форм.

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

Создаёт таблицу с именем name в БД db или текущей БД, если db не указана, со структурой, указанной в скобках, и движком engine.
Структура таблицы представляет список описаний столбцов. Индексы, если поддерживаются движком, указываются в качестве параметров для движка таблицы.

Описание столбца, это name type, в простейшем случае. Пример: RegionID UInt32.
Также могут быть указаны выражения для значений по умолчанию - смотрите ниже.

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

Создаёт таблицу с такой же структурой, как другая таблица. Можно указать другой движок для таблицы. Если движок не указан, то будет выбран такой же движок, как у таблицы db2.name2.

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

Создаёт таблицу со структурой, как результат запроса SELECT, с движком engine, и заполняет её данными из SELECT-а.

Во всех случаях, если указано IF NOT EXISTS, то запрос не будет возвращать ошибку, если таблица уже существует. В этом случае, запрос будет ничего не делать.

После секции ENGINE в запросе могут использоваться и другие секции в зависимости от движка. Подробную документацию по созданию таблиц смотрите в описаниях движков.

Значения по умолчанию

В описании столбца, может быть указано выражение для значения по умолчанию, одного из следующих видов:
DEFAULT expr, MATERIALIZED expr, ALIAS expr.
Пример: URLDomain String DEFAULT domain(URL).

Если выражение для значения по умолчанию не указано, то в качестве значений по умолчанию будут использоваться нули для чисел, пустые строки для строк, пустые массивы для массивов, а также 0000-00-00 для дат и 0000-00-00 00:00:00 для дат с временем. NULL-ы не поддерживаются.

В случае, если указано выражение по умолчанию, то указание типа столбца не обязательно. При отсутствии явно указанного типа, будет использован тип выражения по умолчанию. Пример: EventDate DEFAULT toDate(EventTime) - для столбца EventDate будет использован тип Date.

При наличии явно указанного типа данных и выражения по умолчанию, это выражение будет приводиться к указанному типу с использованием функций приведения типа. Пример: Hits UInt32 DEFAULT 0 - имеет такой же смысл, как Hits UInt32 DEFAULT toUInt32(0).

В качестве выражения для умолчания, может быть указано произвольное выражение от констант и столбцов таблицы. При создании и изменении структуры таблицы, проверяется, что выражения не содержат циклов. При INSERT-е проверяется разрешимость выражений - что все столбцы, из которых их можно вычислить, переданы.

DEFAULT expr

Обычное значение по умолчанию. Если в запросе INSERT не указан соответствующий столбец, то он будет заполнен путём вычисления соответствующего выражения.

MATERIALIZED expr

Материализованное выражение. Такой столбец не может быть указан при INSERT, то есть, он всегда вычисляется.
При INSERT без указания списка столбцов, такие столбцы не рассматриваются.
Также этот столбец не подставляется при использовании звёздочки в запросе SELECT. Это необходимо, чтобы сохранить инвариант, что дамп, полученный путём SELECT *, можно вставить обратно в таблицу INSERT-ом без указания списка столбцов.

ALIAS expr

Синоним. Такой столбец вообще не хранится в таблице.
Его значения не могут быть вставлены в таблицу, он не подставляется при использовании звёздочки в запросе SELECT.
Он может быть использован в SELECT-ах - в таком случае, во время разбора запроса, алиас раскрывается.

При добавлении новых столбцов с помощью запроса ALTER, старые данные для этих столбцов не записываются. Вместо этого, при чтении старых данных, для которых отсутствуют значения новых столбцов, выполняется вычисление выражений по умолчанию на лету. При этом, если выполнение выражения требует использования других столбцов, не указанных в запросе, то эти столбцы будут дополнительно прочитаны, но только для тех блоков данных, для которых это необходимо.

Если добавить в таблицу новый столбец, а через некоторое время изменить его выражение по умолчанию, то используемые значения для старых данных (для данных, где значения не хранились на диске) поменяются. Также заметим, что при выполнении фоновых слияний, данные для столбцов, отсутствующих в одном из сливаемых кусков, записываются в объединённый кусок.

Отсутствует возможность задать значения по умолчанию для элементов вложенных структур данных.

Выражение для TTL

Может быть указано только для таблиц семейства MergeTree. Выражение для указания времени хранения значений. Оно должно зависеть от стобца типа Date или DateTime и в качестве результата вычислять столбец типа Date или DateTime. Пример:
TTL date + INTERVAL 1 DAY

Нельзя указывать TTL для ключевых столбцов. Подробнее смотрите в TTL для стоблцов и таблиц

Форматы сжатия для колонок

Помимо сжатия для колонок по умолчанию, определяемого в настройках сервера,
существует возможность указать формат сжатия индивидуально для каждой колонки.

Поддерживаемые форматы:

- NONE - сжатие отсутствует
- LZ4
- LZ4HC(level) - алгоритм сжатия LZ4_HC с указанным уровнем компрессии level.
  Возможный диапазон значений level: [3, 12]. Значение по умолчанию: 9. Чем выше уровень, тем лучше сжатие, но тратится больше времени. Рекомендованный диапазон [4, 9].
- ZSTD(level) - алгоритм сжатия ZSTD с указанным уровнем компрессии level. Возможный диапазон значений level: [1, 22]. Значение по умолчанию: 1.
  Чем выше уровень, тем лучше сжатие, но тратится больше времени.
- Delta(delta_bytes) - способ сжатия, при котором вместо числовых значений поля сохраняется разность между двумя соседними значениями. Значение delta_bytes - число байт для хранения дельты.
  Возможные значения: 1, 2, 4, 8. Значение по умолчанию: если sizeof(type) равен 1, 2, 4, 8 - sizeof(type), иначе - 1.

Пример использования:

```
CREATE TABLE codec_example
(
    dt Date CODEC(ZSTD), /* используется уровень сжатия по умолчанию */
    ts DateTime CODEC(LZ4HC),
    float_value Float32 CODEC(NONE),
    double_value Float64 CODEC(LZ4HC(9))
)
ENGINE = MergeTree
PARTITION BY tuple()
ORDER BY dt
```

Кодеки могут комбинироваться между собой. Если для колонки указана своя последовательность кодеков, то общий табличный кодек не применяется (должен быть указан в последовательности принудительно, если нужен). В примере ниже - оптимизация для хранения timeseries метрик.
Как правило, значения одной и той же метрики path не сильно различаются между собой, и выгоднее использовать дельта-компрессию вместо записи всего числа:

```
CREATE TABLE timeseries_example
(
    dt Date,
    ts DateTime,
    path String,
    value Float32 CODEC(Delta, ZSTD)
)
ENGINE = MergeTree
PARTITION BY dt
ORDER BY (path, ts)
```

Временные таблицы

ClickHouse поддерживает временные таблицы со следующими характеристиками:

- временные таблицы исчезают после завершения сессии; в том числе, при обрыве соединения;
- Временная таблица использует только модуль памяти.
- Невозможно указать базу данных для временной таблицы. Временные таблицы создается вне баз данных.
- если временная таблица имеет то же имя, что и некоторая другая, то, при упоминании в запросе без указания БД, будет использована временная таблица;
- при распределённой обработке запроса, используемые в запросе временные таблицы, передаются на удалённые серверы.

Чтобы создать временную таблицу, используйте следующий синтаксис:

```
CREATE TEMPORARY TABLE [IF NOT EXISTS] table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
)
```

В большинстве случаев, временные таблицы создаются не вручную, а при использовании внешних данных для запроса, или при распределённом (GLOBAL) IN. Подробнее см. соответствующие разделы

Распределенные DDL запросы (секция ON CLUSTER)

Запросы CREATE, DROP, ALTER, RENAME поддерживают возможность распределенного выполнения на кластере.
Например, следующий запрос создает распределенную (Distributed) таблицу all_hits на каждом хосте в cluster:

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

Для корректного выполнения таких запросов необходимо на каждом хосте иметь одинаковое определение кластера (для упрощения синхронизации конфигов можете использовать подстановки из ZooKeeper). Также необходимо подключение к ZooKeeper серверам.
Локальная версия запроса в конечном итоге будет выполнена на каждом хосте кластера, даже если некоторые хосты в данный момент не доступны. Гарантируется упорядоченность выполнения запросов в рамках одного хоста. Для реплицированных таблиц не поддерживаются запросы ALTER.

CREATE VIEW

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

Создаёт представление. Представления бывают двух видов - обычные и материализованные (MATERIALIZED).

Обычные представления не хранят никаких данных, а всего лишь производят чтение из другой таблицы. То есть, обычное представление - не более чем сохранённый запрос. При чтении из представления, этот сохранённый запрос, используется в качестве подзапроса в секции FROM.

Для примера, пусть вы создали представление:

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

и написали запрос:

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

Этот запрос полностью эквивалентен использованию подзапроса:

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

Материализованные (MATERIALIZED) представления хранят данные, преобразованные соответствующим запросом SELECT.

При создании материализованного представления, нужно обязательно указать ENGINE - движок таблицы для хранения данных.

Материализованное представление устроено следующим образом: при вставке данных в таблицу, указанную в SELECT-е, кусок вставляемых данных преобразуется этим запросом SELECT, и полученный результат вставляется в представление.

Если указано POPULATE, то при создании представления, в него будут вставлены имеющиеся данные таблицы, как если бы был сделан запрос CREATE TABLE ... AS SELECT ... . Иначе, представление будет содержать только данные, вставляемые в таблицу после создания представления. Не рекомендуется использовать POPULATE, так как вставляемые в таблицу данные во время создания представления, не попадут в него.

Запрос SELECT может содержать DISTINCT, GROUP BY, ORDER BY, LIMIT... Следует иметь ввиду, что соответствующие преобразования будут выполняться независимо, на каждый блок вставляемых данных. Например, при наличии GROUP BY, данные будут агрегироваться при вставке, но только в рамках одной пачки вставляемых данных. Далее, данные не будут доагрегированы. Исключение - использование ENGINE, производящего агрегацию данных самостоятельно, например, SummingMergeTree.

Недоработано выполнение запросов ALTER над материализованными представлениями, поэтому они могут быть неудобными для использования. Если материализованное представление использует конструкцию TO [db.]name, то можно выполнить DETACH представления, ALTER для целевой таблицы и последующий ATTACH ранее отсоединенного (DETACH) представления.

Представления выглядят так же, как обычные таблицы. Например, они перечисляются в результате запроса SHOW TABLES.

Отсутствует отдельный запрос для удаления представлений. Чтобы удалить представление, следует использовать DROP TABLE.

ALTER

Запрос ALTER поддерживается только для таблиц типа *MergeTree, а также Merge и Distributed. Запрос имеет несколько вариантов.

Манипуляции со столбцами

Изменение структуры таблицы.

```
ALTER TABLE [db].name [ON CLUSTER cluster] ADD|DROP|CLEAR|COMMENT|MODIFY COLUMN ...
```

В запросе указывается список из одного или более действий через запятую.
Каждое действие — операция над столбцом.

Существуют следующие действия:

- ADD COLUMN — добавляет столбец в таблицу;
- DROP COLUMN — удаляет столбец;
- CLEAR COLUMN — сбрасывает все значения в столбце для заданной партиции;
- COMMENT COLUMN — добавляет комментарий к столбцу;
- MODIFY COLUMN — изменяет тип столбца и/или выражение для значения по умолчанию.

Подробное описание для каждого действия приведено ниже.

ADD COLUMN

```
ADD COLUMN [IF NOT EXISTS] name [type] [default_expr] [AFTER name_after]
```

Добавляет в таблицу новый столбец с именем name, типом type и выражением для умолчания default_expr (смотрите раздел Значения по умолчанию).

Если указано IF NOT EXISTS, запрос не будет возвращать ошибку, если столбец уже существует. Если указано AFTER name_after (имя другого столбца), то столбец добавляется (в список столбцов таблицы) после указанного. Иначе, столбец добавляется в конец таблицы. Обратите внимание, ClickHouse не позволяет добавлять столбцы в начало таблицы. Для цепочки действий, name_after может быть именем столбца, который добавляется в одном из предыдущих действий.

Добавление столбца всего лишь меняет структуру таблицы, и не производит никаких действий с данными - соответствующие данные не появляются на диске после ALTER-а. При чтении из таблицы, если для какого-либо столбца отсутствуют данные, то он заполняется значениями по умолчанию (выполняя выражение по умолчанию, если такое есть, или нулями, пустыми строками). Также, столбец появляется на диске при слиянии кусков данных (см. MergeTree).

Такая схема позволяет добиться мгновенной работы запроса ALTER и отсутствия необходимости увеличивать объём старых данных.

Пример:

```
ALTER TABLE visits ADD COLUMN browser String AFTER user_id
```

DROP COLUMN

```
DROP COLUMN [IF EXISTS] name
```

Удаляет столбец с именем name. Если указано IF EXISTS, запрос не будет возвращать ошибку, если столбца не существует.

Запрос удаляет данные из файловой системы. Так как это представляет собой удаление целых файлов, запрос выполняется почти мгновенно.

Пример:

```
ALTER TABLE visits DROP COLUMN browser
```

CLEAR COLUMN

```
CLEAR COLUMN [IF EXISTS] name IN PARTITION partition_name
```

Сбрасывает все значения в столбце для заданной партиции. Если указано IF EXISTS, запрос не будет возвращать ошибку, если столбца не существует.

Как корректно задать имя партиции, см. в разделе Как задавать имя партиции в запросах ALTER.

Пример:

```
ALTER TABLE visits CLEAR COLUMN browser IN PARTITION tuple()
```

COMMENT COLUMN

```
COMMENT COLUMN [IF EXISTS] name 'Text comment'
```

Добавляет комментарий к таблице. Если указано IF EXISTS, запрос не будет возвращать ошибку, если столбца не существует.

Каждый столбец может содержать только один комментарий. При выполнении запроса существующий комментарий заменяется на новый.

Посмотреть комментарии можно в столбце comment_expression из запроса DESCRIBE TABLE.

Пример:

```
ALTER TABLE visits COMMENT COLUMN browser 'Столбец показывает, из каких браузеров пользователи заходили на сайт.'
```

MODIFY COLUMN

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

Изменяет тип столбца name на type и/или выражение для умолчания на default_expr. Если указано IF EXISTS, запрос не будет возвращать ошибку, если столбца не существует.

При изменении типа, значения преобразуются так, как если бы к ним была применена функция toType. Если изменяется только выражение для умолчания, запрос не делает никакой сложной работы и выполняется мгновенно.

Пример запроса:

```
ALTER TABLE visits MODIFY COLUMN browser Array(String)
```

Изменение типа столбца - это единственное действие, которое выполняет сложную работу - меняет содержимое файлов с данными. Для больших таблиц, выполнение может занять длительное время.

Выполнение производится в несколько стадий:

- подготовка временных (новых) файлов с изменёнными данными;
- переименование старых файлов;
- переименование временных (новых) файлов в старые;
- удаление старых файлов.

Из них, длительной является только первая стадия. Если на этой стадии возникнет сбой, то данные не поменяются.
Если на одной из следующих стадий возникнет сбой, то данные будет можно восстановить вручную. За исключением случаев, когда старые файлы удалены из файловой системы, а данные для новых файлов не доехали на диск и потеряны.

Запрос ALTER на изменение столбцов реплицируется. Соответствующие инструкции сохраняются в ZooKeeper, и затем каждая реплика их применяет. Все запросы ALTER выполняются в одном и том же порядке. Запрос ждёт выполнения соответствующих действий на всех репликах. Но при этом, запрос на изменение столбцов в реплицируемой таблице можно прервать, и все действия будут осуществлены асинхронно.

Ограничения запроса ALTER

Запрос ALTER позволяет создавать и удалять отдельные элементы (столбцы) вложенных структур данных, но не вложенные структуры данных целиком. Для добавления вложенной структуры данных, вы можете добавить столбцы с именем вида name.nested_name и типом Array(T) - вложенная структура данных полностью эквивалентна нескольким столбцам-массивам с именем, имеющим одинаковый префикс до точки.

Отсутствует возможность удалять столбцы, входящие в первичный ключ или ключ для сэмплирования (в общем, входящие в выражение ENGINE). Изменение типа у столбцов, входящих в первичный ключ возможно только в том случае, если это изменение не приводит к изменению данных (например, разрешено добавление значения в Enum или изменение типа с DateTime на UInt32).

Если возможностей запроса ALTER не хватает для нужного изменения таблицы, вы можете создать новую таблицу, скопировать туда данные с помощью запроса INSERT SELECT, затем поменять таблицы местами с помощью запроса RENAME, и удалить старую таблицу. В качестве альтернативы для запроса INSERT SELECT, можно использовать инструмент clickhouse-copier.

Запрос ALTER блокирует все чтения и записи для таблицы. То есть, если на момент запроса ALTER, выполнялся долгий SELECT, то запрос ALTER сначала дождётся его выполнения. И в это время, все новые запросы к той же таблице, будут ждать, пока завершится этот ALTER.

Для таблиц, которые не хранят данные самостоятельно (типа Merge и Distributed), ALTER всего лишь меняет структуру таблицы, но не меняет структуру подчинённых таблиц. Для примера, при ALTER-е таблицы типа Distributed, вам также потребуется выполнить запрос ALTER для таблиц на всех удалённых серверах.

Манипуляции с ключевыми выражениями таблиц

Поддерживается операция:

```
MODIFY ORDER BY new_expression
```

Работает только для таблиц семейства MergeTree (в том числе реплицированных). После выполнения запроса
ключ сортировки таблицы
заменяется на new_expression (выражение или кортеж выражений). Первичный ключ при этом остаётся прежним.

Операция затрагивает только метаданные. Чтобы сохранить свойство упорядоченности кусков данных по ключу
сортировки, разрешено добавлять в ключ только новые столбцы (т.е. столбцы, добавляемые командой ADD COLUMN
в том же запросе ALTER), у которых нет выражения по умолчанию.

Манипуляции с индексами

Добавить или удалить индекс можно с помощью операций

```
ALTER TABLE [db].name ADD INDEX name expression TYPE type GRANULARITY value [AFTER name]
ALTER TABLE [db].name DROP INDEX name
```

Поддерживается только таблицами семейства *MergeTree.

Команда ADD INDEX добавляет описание индексов в метаданные, а DROP INDEX удаляет индекс из метаданных и стирает файлы индекса с диска, поэтому они легковесные и работают мгновенно.

Если индекс появился в метаданных, то он начнет считаться в последующих слияниях и записях в таблицу, а не сразу после выполнения операции ALTER.

Запрос на изменение индексов реплицируется, сохраняя новые метаданные в ZooKeeper и применяя изменения на всех репликах.

Манипуляции с партициями и кусками

Для работы с партициями доступны следующие операции:

- DETACH PARTITION – перенести партицию в директорию detached;
- DROP PARTITION – удалить партицию;
- ATTACH PARTITION|PART – добавить партицию/кусок в таблицу из директории detached;
- REPLACE PARTITION – скопировать партицию из другой таблицы;

- CLEAR COLUMN IN PARTITION – удалить все значения в столбце для заданной партиции;
- FREEZE PARTITION – создать резервную копию партиции;
- FETCH PARTITION – скачать партицию с другого сервера.

DETACH PARTITION

```
ALTER TABLE table_name DETACH PARTITION partition_expr
```

Перемещает заданную партицию в директорию detached. Сервер не будет знать об этой партиции до тех пор, пока вы не выполните запрос ATTACH.

Пример:

```
ALTER TABLE visits DETACH PARTITION 201901
```

Подробнее о том, как корректно задать имя партиции, см. в разделе Как задавать имя партиции в запросах ALTER.

После того как запрос будет выполнен, вы сможете производить любые операции с данными в директории detached. Например, можно удалить их из файловой системы.

Запрос реплицируется — данные будут перенесены в директорию detached и забыты на всех репликах. Обратите внимание, запрос может быть отправлен только на реплику-лидер. Чтобы узнать, является ли реплика лидером, выполните запрос SELECT к системной таблице system.replicas. Либо можно выполнить запрос DETACH на всех репликах — тогда на всех репликах, кроме реплики-лидера, запрос вернет ошибку.

DROP PARTITION

```
ALTER TABLE table_name DROP PARTITION partition_expr
```

Удаляет партицию. Партиция помечается как неактивная и будет полностью удалена примерно через 10 минут.

Подробнее о том, как корректно задать имя партиции, см. в разделе Как задавать имя партиции в запросах ALTER.

Запрос реплицируется — данные будут удалены на всех репликах.

ATTACH PARTITION|PART

```
ALTER TABLE table_name ATTACH PARTITION|PART partition_expr
```

Добавляет данные в таблицу из директории detached. Можно добавить данные как для целой партиции, так и для отдельного куска. Примеры:

```
ALTER TABLE visits ATTACH PARTITION 201901;
ALTER TABLE visits ATTACH PART 201901_2_2_0;
```

Как корректно задать имя партиции или куска, см. в разделе Как задавать имя партиции в запросах ALTER.

Этот запрос реплицируется. Каждая реплика проверяет, есть ли данные в директории detached. Если данные есть, то запрос проверяет их целостность и соответствие данным на сервере-инициаторе запроса. В случае успеха данные добавляются в таблицу. В противном случае, реплика загружает данные с реплики-инициатора запроса или с другой реплики, на которой эти данные уже добавлены.

Это означает, что вы можете разместить данные в директории detached на одной реплике и с помощью запроса ALTER ... ATTACH добавить их в таблицу на всех репликах.

REPLACE PARTITION

```
ALTER TABLE table2 REPLACE PARTITION partition_expr FROM table1
```

Копирует партицию из таблицы table1 в таблицу table2. Данные из table1 не удаляются.

Следует иметь в виду:

- Таблицы должны иметь одинаковую структуру.
- Для таблиц должен быть задан одинаковый ключ партиционирования.

Подробнее о том, как корректно задать имя партиции, см. в разделе Как задавать имя партиции в запросах ALTER.

CLEAR COLUMN IN PARTITION

```
ALTER TABLE table_name CLEAR COLUMN column_name IN PARTITION partition_expr
```

Сбрасывает все значения в столбце для заданной партиции. Если для столбца определено значение по умолчанию (в секции DEFAULT), то будет выставлено это значение.

Пример:

```
ALTER TABLE visits CLEAR COLUMN hour in PARTITION 201902
```

FREEZE PARTITION

```
ALTER TABLE table_name FREEZE [PARTITION partition_expr]
```

Создаёт резервную копию для заданной партиции. Если выражение PARTITION опущено, резервные копии будут созданы для всех партиций.

Note

Создание резервной копии не требует остановки сервера.

Для таблиц старого стиля имя партиций можно задавать в виде префикса (например, '2019'). В этом случае резервные копии будут созданы для всех соответствующих партиций. Подробнее о том, как корректно задать имя партиции, см. в разделе Как задавать имя партиции в запросах ALTER.

Запрос делает следующее — для текущего состояния таблицы он формирует жесткие ссылки на данные в этой таблице. Ссылки размещаются в директории /var/lib/clickhouse/shadow/N/..., где:

- /var/lib/clickhouse/ — рабочая директория ClickHouse, заданная в конфигурационном файле;
- N — инкрементальный номер резервной копии.

Структура директорий внутри резервной копии такая же, как внутри /var/lib/clickhouse/. Запрос выполнит 'chmod' для всех файлов, запрещая запись в них.

Обратите внимание, запрос ALTER TABLE t FREEZE PARTITION не реплицируется. Он создает резервную копию только на локальном сервере. После создания резервной копии данные из /var/lib/clickhouse/shadow/ можно скопировать на удалённый сервер, а локальную копию удалить.

Резервная копия создается почти мгновенно (однако сначала запрос дожидается завершения всех запросов, которые выполняются для соответствующей таблицы).

ALTER TABLE t FREEZE PARTITION копирует только данные, но не метаданные таблицы. Чтобы сделать резервную копию метаданных таблицы, скопируйте файл /var/lib/clickhouse/metadata/database/table.sql

Чтобы восстановить данные из резервной копии, выполните следующее:

1. Создайте таблицу, если она ещё не существует. Запрос на создание можно взять из .sql файла (замените в нём ATTACH на CREATE).
2. Скопируйте данные из директории data/database/table/ внутри резервной копии в директорию /var/lib/clickhouse/data/database/table/detached/.
3. С помощью запросов ALTER TABLE t ATTACH PARTITION добавьте данные в таблицу.

Восстановление данных из резервной копии не требует остановки сервера.

Подробнее о резервном копировании и восстановлении данных читайте в разделе Резервное копирование данных.

FETCH PARTITION

```
ALTER TABLE table_name FETCH PARTITION partition_expr FROM 'path-in-zookeeper'
```

Загружает партицию с другого сервера. Этот запрос работает только для реплицированных таблиц.

Запрос выполняет следующее:

1. Загружает партицию с указанного шарда. Путь к шарду задается в секции FROM ('path-in-zookeeper'). Обратите внимание, нужно задавать путь к шарду в ZooKeeper.
2. Помещает загруженные данные в директорию detached таблицы table_name. Чтобы прикрепить эти данные к таблице, используйте запрос ATTACH PARTITION|PART.

Например:

```
ALTER TABLE users FETCH PARTITION 201902 FROM '/clickhouse/tables/01-01/visits';
ALTER TABLE users ATTACH PARTITION 201902;
```

Следует иметь в виду:

- Запрос ALTER TABLE t FETCH PARTITION не реплицируется. Он загружает партицию в директорию detached только на локальном сервере.
- Запрос ALTER TABLE t ATTACH реплицируется — он добавляет данные в таблицу сразу на всех репликах. На одной из реплик данные будут добавлены из директории detached, а на других — из соседних реплик.

Перед загрузкой данных система проверяет, существует ли партиция и совпадает ли её структура со структурой таблицы. При этом автоматически выбирается наиболее актуальная реплика среди всех живых реплик.

Несмотря на то что запрос называется ALTER TABLE, он не изменяет структуру таблицы и не изменяет сразу доступные данные в таблице.

Как задавать имя партиции в запросах ALTER

Чтобы задать нужную партицию в запросах ALTER ... PARTITION, можно использовать:

- Имя партиции. Посмотреть имя партиции можно в столбце partition системной таблицы system.parts. Например, ALTER TABLE visits DETACH PARTITION 201901.
- Произвольное выражение из столбцов исходной таблицы. Также поддерживаются константы и константные выражения. Например, ALTER TABLE visits DETACH PARTITION toYYYYMM(toDate('2019-01-25')).
- Строковый идентификатор партиции. Идентификатор партиции используется для именования кусков партиции на файловой системе и в ZooKeeper. В запросах ALTER идентификатор партиции нужно указывать в секции PARTITION ID, в одинарных кавычках. Например, ALTER TABLE visits DETACH PARTITION ID '201901'.
- Для запросов ATTACH PART: чтобы задать имя куска партиции, используйте значение из столбца name системной таблицы system.parts. Например, ALTER TABLE visits ATTACH PART 201901_1_1_0.

Использование кавычек в имени партиций зависит от типа данных столбца, по которому задано партиционирование. Например, для столбца с типом String имя партиции необходимо указывать в кавычках (одинарных). Для типов Date и Int* кавычки указывать не нужно.

Замечание: для таблиц старого стиля партицию можно указывать и как число 201901, и как строку '201901'. Синтаксис для таблиц нового типа более строг к типам (аналогично парсеру входного формата VALUES).

Правила, сформулированные выше, актуальны также для запросов OPTIMIZE. Чтобы указать единственную партицию непартиционированной таблицы, укажите PARTITION tuple(). Например:

```
OPTIMIZE TABLE table_not_partitioned PARTITION tuple() FINAL;
```

Примеры запросов ALTER ... PARTITION можно посмотреть в тестах: 00502_custom_partitioning_local и 00502_custom_partitioning_replicated_zookeeper.

Синхронность запросов ALTER

Для нереплицируемых таблиц, все запросы ALTER выполняются синхронно. Для реплицируемых таблиц, запрос всего лишь добавляет инструкцию по соответствующим действиям в ZooKeeper, а сами действия осуществляются при первой возможности. Но при этом, запрос может ждать завершения выполнения этих действий на всех репликах.

Для запросов ALTER ... ATTACH|DETACH|DROP можно настроить ожидание, с помощью настройки replication_alter_partitions_sync.
Возможные значения: 0 - не ждать, 1 - ждать выполнения только у себя (по умолчанию), 2 - ждать всех.

Мутации

Мутации - разновидность запроса ALTER, позволяющая изменять или удалять данные в таблице. В отличие от стандартных запросов DELETE и UPDATE, рассчитанных на точечное изменение данных, область применения мутаций - достаточно тяжёлые изменения, затрагивающие много строк в таблице.

Функциональность находится в состоянии beta и доступна начиная с версии 1.1.54388. Реализована поддержка *MergeTree таблиц (с репликацией и без).

Конвертировать существующие таблицы для работы с мутациями не нужно. Но после применения первой мутации формат данных таблицы становится несовместимым с предыдущими версиями и откатиться на предыдущую версию уже не получится.

На данный момент доступны команды:

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

Выражение filter_expr должно иметь тип UInt8. Запрос удаляет строки таблицы, для которых это выражение принимает ненулевое значение.

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

Команда доступна начиная с версии 18.12.14. Выражение filter_expr должно иметь тип UInt8. Запрос изменяет значение указанных столбцов на вычисленное значение соответствующих выражений в каждой строке, для которой filter_expr принимает ненулевое значение. Вычисленные значения преобразуются к типу столбца с помощью оператора CAST. Изменение столбцов, которые используются при вычислении первичного ключа или ключа партиционирования, не поддерживается.

В одном запросе можно указать несколько команд через запятую.

Для *MergeTree-таблиц мутации выполняются, перезаписывая данные по кускам (parts). При этом атомарности нет — куски заменяются на помутированные по мере выполнения и запрос SELECT, заданный во время выполнения мутации, увидит данные как из измененных кусков, так и из кусков, которые еще не были изменены.

Мутации линейно упорядочены между собой и накладываются на каждый кусок в порядке добавления. Мутации также упорядочены со вставками - гарантируется, что данные, вставленные в таблицу до начала выполнения запроса мутации, будут изменены, а данные, вставленные после окончания запроса мутации, изменены не будут. При этом мутации никак не блокируют вставки.

Запрос завершается немедленно после добавления информации о мутации (для реплицированных таблиц - в ZooKeeper, для нереплицированных - на файловую систему). Сама мутация выполняется асинхронно, используя настройки системного профиля. Следить за ходом её выполнения можно по таблице system.mutations. Добавленные мутации будут выполняться до конца даже в случае перезапуска серверов ClickHouse. Откатить мутацию после её добавления нельзя, но если мутация по какой-то причине не может выполниться до конца, её можно остановить с помощью запроса KILL MUTATION.

Записи о последних выполненных мутациях удаляются не сразу (количество сохраняемых мутаций определяется параметром движка таблиц finished_mutations_to_keep). Более старые записи удаляются.

Прочие виды запросов

ATTACH

Запрос полностью аналогичен запросу CREATE, но:

- вместо слова CREATE используется слово ATTACH;
- запрос не создаёт данные на диске, а предполагает, что данные уже лежат в соответствующих местах, и всего лишь добавляет информацию о таблице на сервер. После выполнения запроса ATTACH сервер будет знать о существовании таблицы.

Если таблица перед этим была отсоединена (DETACH), т.е. её структура известна, можно использовать сокращенную форму записи без определения структуры.

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

Этот запрос используется при старте сервера. Сервер хранит метаданные таблиц в виде файлов с запросами ATTACH, которые он просто исполняет при запуске (за исключением системных таблиц, которые явно создаются на сервере).

CHECK TABLE

Проверяет таблицу на повреждение данных.

```
CHECK TABLE [db.]name
```

Запрос CHECK TABLE сравнивает текущие размеры файлов (в которых хранятся данные из колонок) с ожидаемыми значениями. Если значения не совпадают, данные в таблице считаются поврежденными. Искажение возможно, например, из-за сбоя при записи данных.

Ответ содержит колонку result, содержащую одну строку с типом Boolean. Допустимые значения:

- 0 - данные в таблице повреждены;
- 1 - данные не повреждены.

Запрос CHECK TABLE поддерживается только для следующих движков:

- Log
- TinyLog
- StripeLog

В этих движках не предусмотрено автоматическое восстановление данных после сбоя. Поэтому используйте запрос CHECK TABLE, чтобы своевременно выявить повреждение данных.

Обратите внимание, высокая защита целостности данных обеспечивается в таблицах семейства MergeTree. Для избежания потери данных рекомендуется использовать именно эти таблицы.

Что делать, если данные повреждены

В этом случае можно скопировать оставшиеся неповрежденные данные в другую таблицу. Для этого:

1. Создайте новую таблицу с такой же структурой, как у поврежденной таблицы. Для этого выполните запрос CREATE TABLE <new_table_name> AS <damaged_table_name>.
2. Установите значение параметра max_threads в 1. Это нужно для того, чтобы выполнить следующий запрос в одном потоке. Установить значение параметра можно через запрос: SET max_threads = 1.
3. Выполните запрос INSERT INTO <new_table_name> SELECT * FROM <damaged_table_name>. В результате неповрежденные данные будут скопированы в другую таблицу. Обратите внимание, будут скопированы только те данные, которые следуют до поврежденного участка.
4. Перезапустите clickhouse-client, чтобы вернуть предыдущее значение параметра max_threads.

DESCRIBE TABLE

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

Возвращает описание столбцов таблицы.

Результат запроса содержит столбцы (все столбцы имеют тип String):

- name — имя столбца таблицы;
- type— тип столбца;
- default_type — в каком виде задано выражение для значения по умолчанию: DEFAULT, MATERIALIZED или ALIAS. Столбец содержит пустую строку, если значение по умолчанию не задано.
- default_expression — значение, заданное в секции DEFAULT;
- comment_expression — комментарий к столбцу.

Вложенные структуры данных выводятся в "развёрнутом" виде. То есть, каждый столбец - по отдельности, с именем через точку.

DETACH

Удаляет из сервера информацию о таблице name. Сервер перестаёт знать о существовании таблицы.

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

Но ни данные, ни метаданные таблицы не удаляются. При следующем запуске сервера, сервер прочитает метаданные и снова узнает о таблице.
Также, "отцепленную" таблицу можно прицепить заново запросом ATTACH (за исключением системных таблиц, для которых метаданные не хранятся).

Запроса DETACH DATABASE нет.

DROP

Запрос имеет два вида: DROP DATABASE и DROP TABLE.

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

Удаляет все таблицы внутри базы данных db, а затем саму базу данных db.
Если указано IF EXISTS - не выдавать ошибку, если база данных не существует.

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

Удаляет таблицу.
Если указано IF EXISTS - не выдавать ошибку, если таблица не существует или база данных не существует.

EXISTS

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

Возвращает один столбец типа UInt8, содержащий одно значение - 0, если таблицы или БД не существует и 1, если таблица в указанной БД существует.

KILL QUERY

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

Пытается принудительно остановить исполняющиеся в данный момент запросы.
Запросы для принудительной остановки выбираются из таблицы system.processes с помощью условия, указанного в секции WHERE запроса KILL.

Примеры

```
-- Принудительно останавливает все запросы с указанным query_id:
KILL QUERY WHERE query_id='2-857d-4a57-9ee0-327da5d60a90'

-- Синхронно останавливает все запросы пользователя 'username':
KILL QUERY WHERE user='username' SYNC
```

Readonly-пользователи могут останавливать только свои запросы.

По умолчанию используется асинхронный вариант запроса (ASYNC), который не дожидается подтверждения остановки запросов.

Синхронный вариант (SYNC) ожидает остановки всех запросов и построчно выводит информацию о процессах по ходу их остановки.
Ответ содержит колонку kill_status, которая может принимать следующие значения:

1. 'finished' - запрос был успешно остановлен;
2. 'waiting' - запросу отправлен сигнал завершения, ожидается его остановка;
3. остальные значения описывают причину невозможности остановки запроса.

Тестовый вариант запроса (TEST) только проверяет права пользователя и выводит список запросов для остановки.

KILL MUTATION

```
KILL MUTATION [ON CLUSTER cluster]
  WHERE <where expression to SELECT FROM system.mutations query>
  [TEST]
  [FORMAT format]
```

Пытается остановить выполняющиеся в данные момент мутации. Мутации для остановки выбираются из таблицы system.mutations с помощью условия, указанного в секции WHERE запроса KILL.

Тестовый вариант запроса (TEST) только проверяет права пользователя и выводит список запросов для остановки.

Примеры:

```
-- Останавливает все мутации одной таблицы:
KILL MUTATION WHERE database = 'default' AND table = 'table'

-- Останавливает конкретную мутацию:
KILL MUTATION WHERE database = 'default' AND table = 'table' AND mutation_id = 'mutation_3.txt'
```

Запрос полезен в случаях, когда мутация не может выполниться до конца (например, если функция в запросе мутации бросает исключение на данных таблицы).

Данные, уже изменённые мутацией, остаются в таблице (отката на старую версию данных не происходит).

OPTIMIZE

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

Просит движок таблицы сделать что-нибудь, что может привести к более оптимальной работе.
Поддерживается только движками *MergeTree, в котором выполнение этого запроса инициирует внеочередное слияние кусков данных.
Если указан PARTITION, то оптимизация будет производиться только для указаной партиции.
Если указан FINAL, то оптимизация будет производиться даже когда все данные уже лежат в одном куске.

!!! warning "Внимание"Запрос OPTIMIZE не может устранить причину появления ошибки "Too many parts".

RENAME

Переименовывает одну или несколько таблиц.

```
RENAME TABLE [db11.]name11 TO [db12.]name12, [db21.]name21 TO [db22.]name22, ... [ON CLUSTER cluster]
```

Все таблицы переименовываются под глобальной блокировкой. Переименовывание таблицы является лёгкой операцией. Если вы указали после TO другую базу данных, то таблица будет перенесена в эту базу данных. При этом, директории с базами данных должны быть расположены в одной файловой системе (иначе возвращается ошибка).

SET

```
SET param = value
```

Позволяет установить настройку param в значение value. Также можно одним запросом установить все настройки из заданного профиля настроек. Для этого укажите 'profile' в качестве имени настройки.
Подробнее смотрите в разделе "Настройки".
Настройка устанавливается на сессию, или на сервер (глобально), если указано GLOBAL.
При установке глобальных настроек, эти настройки не применяются к уже запущенной сессии, включая текущую сессию. Она будет использована только для новых сессий.

При перезапуске сервера теряются настройки, установленные с помощью SET.
Установить настройки, которые переживут перезапуск сервера, можно только с помощью конфигурационного файла сервера.

SHOW CREATE TABLE

```
SHOW CREATE [TEMPORARY] TABLE [db.]table [INTO OUTFILE filename] [FORMAT format]
```

Возвращает один столбец statement типа String, содержащий одно значение - запрос CREATE, с помощью которого создана указанная таблица.

SHOW DATABASES

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

Выводит список всех баз данных.
Запрос полностью аналогичен запросу SELECT name FROM system.databases [INTO OUTFILE filename] [FORMAT format].

Смотрите также раздел "Форматы".

SHOW PROCESSLIST

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

Выводит список запросов, выполняющихся в данный момент времени, кроме самих запросов SHOW PROCESSLIST.

Выдаёт таблицу, содержащую столбцы:

user - пользователь, под которым был задан запрос. Следует иметь ввиду, что при распределённой обработке запроса на удалённые серверы запросы отправляются под пользователем 'default'. И SHOW PROCESSLIST показывает имя пользователя для конкретного запроса, а не для запроса, который данный запрос инициировал.

address - имя хоста, с которого был отправлен запрос. При распределённой обработке запроса на удалённых серверах — это имя хоста-инициатора запроса. Чтобы проследить, откуда был задан распределённый запрос изначально, следует смотреть SHOW PROCESSLIST на сервере-инициаторе запроса.

elapsed - время выполнения запроса, в секундах. Запросы выводятся в порядке убывания времени выполнения.

rows_read, **bytes_read** - сколько было прочитано строк, байт несжатых данных при обработке запроса. При распределённой обработке запроса суммируются данные со всех удалённых серверов. Именно эти данные используются для ограничений и квот.

memory_usage - текущее потребление оперативки в байтах. Смотрите настройку 'max_memory_usage'.

query - сам запрос. В запросах INSERT данные для вставки не выводятся.

query_id - идентификатор запроса. Непустой, только если был явно задан пользователем. При распределённой обработке запроса идентификатор запроса не передаётся на удалённые серверы.

Этот запрос аналогичен запросу SELECT * FROM system.processes за тем исключением, что последний отображает список запросов, включая самого себя.

Полезный совет (выполните в консоли):

```
watch -n1 "clickhouse-client --query='SHOW PROCESSLIST'"
```

SHOW TABLES

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

Выводит список таблиц:

- из текущей базы данных или из базы db, если указано FROM db;
- всех, или имя которых соответствует шаблону pattern, если указано LIKE 'pattern';

Запрос полностью аналогичен запросу: SELECT name FROM system.tables WHERE database = 'db' [AND name LIKE 'pattern'] [INTO OUTFILE filename] [FORMAT format].

Смотрите также раздел "Оператор LIKE".

TRUNCATE

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

Удаляет все данные из таблицы. Если условие IF EXISTS не указано, запрос вернет ошибку, если таблицы не существует.

Запрос TRUNCATE не поддерживается для следующих движков: View, File, URL и Null.

USE

```
USE db
```

Позволяет установить текущую базу данных для сессии.
Текущая база данных используется для поиска таблиц, если база данных не указана в запросе явно через точку перед именем таблицы.
При использовании HTTP протокола запрос не может быть выполнен, так как понятия сессии не существует.

Функции

Функции бывают как минимум* двух видов - обычные функции (называются просто, функциями) и агрегатные функции. Это совершенно разные вещи. Обычные функции работают так, как будто применяются к каждой строке по отдельности (для каждой строки, результат вычисления функции не зависит от других строк). Агрегатные функции аккумулируют множество значений из разных строк (то есть, зависят от целого множества строк).

В этом разделе речь пойдёт об обычных функциях. Для агрегатных функций, смотрите раздел "Агрегатные функции".

* - есть ещё третий вид функций, к которым относится функция arrayJoin; также можно отдельно иметь ввиду табличные функции.*

Строгая типизация

В ClickHouse, в отличие от стандартного SQL, типизация является строгой. То есть, не производится неявных преобразований между типами. Все функции работают для определённого набора типов. Это значит, что иногда вам придётся использовать функции преобразования типов.

Склейка одинаковых выражений

Все выражения в запросе, имеющие одинаковые AST (одинаковую запись или одинаковый результат синтаксического разбора), считаются имеющими одинаковые значения. Такие выражения склеиваются и исполняются один раз. Одинаковые подзапросы тоже склеиваются.

Типы результата

Все функции возвращают одно (не несколько, не ноль) значение в качестве результата. Тип результата обычно определяется только типами аргументов, но не значениями аргументов. Исключение - функция tupleElement (оператор a.N), а также функция toFixedString.

Константы

Для простоты, некоторые функции могут работать только с константами в качестве некоторых аргументов. Например, правый аргумент оператора LIKE должен быть константой.
Почти все функции возвращают константу для константных аргументов. Исключение - функции генерации случайных чисел.
Функция now возвращает разные значения для запросов, выполненных в разное время, но результат считается константой, так как константность важна лишь в пределах одного запроса.
Константное выражение также считается константой (например, правую часть оператора LIKE можно сконструировать из нескольких констант).

Функции могут быть по-разному реализованы для константных и не константных аргументов (выполняется разный код). Но результат работы для константы и полноценного столбца, содержащего только одно такое же значение, должен совпадать.

Обработка NULL

Функции имеют следующие виды поведения:

- Если хотя бы один из аргументов функции — NULL, то результат функции тоже NULL.
- Специальное поведение, указанное в описании каждой функции отдельно. В исходном коде ClickHouse такие функции можно определить по свойству UseDefaultImplementationForNulls=false.

Неизменяемость

Функции не могут поменять значения своих аргументов - любые изменения возвращаются в качестве результата. Соответственно, от порядка записи функций в запросе, результат вычислений отдельных функций не зависит.

Обработка ошибок

Некоторые функции могут кидать исключения в случае ошибочных данных. В этом случае, выполнение запроса прерывается, и текст ошибки выводится клиенту. При распределённой обработке запроса, при возникновении исключения на одном из серверов, на другие серверы пытается отправиться просьба тоже прервать выполнение запроса.

Вычисление выражений-аргументов

В почти всех языках программирования, для некоторых операторов может не вычисляться один из аргументов. Обычно - для операторов &&, ||, ?:.
Но в ClickHouse, аргументы функций (операторов) вычисляются всегда. Это связано с тем, что вычисления производятся не по отдельности для каждой строки, а сразу для целых кусочков столбцов.

Выполнение функций при распределённой обработке запроса

При распределённой обработке запроса, как можно большая часть стадий выполнения запроса производится на удалённых серверах, а оставшиеся стадии (слияние промежуточных результатов и всё, что дальше) - на сервере-инициаторе запроса.

Это значит, что выполнение функций может производиться на разных серверах.
Например, в запросе SELECT f(sum(g(x))) FROM distributed_table GROUP BY h(y),
- если distributed_table имеет хотя бы два шарда, то функции g и h выполняются на удалённых серверах, а функция f - на сервере-инициаторе запроса;
- если distributed_table имеет только один шард, то все функции f, g, h выполняются на сервере этого шарда.

Обычно результат выполнения функции не зависит от того, на каком сервере её выполнить. Но иногда это довольно важно.
Например, функции, работающие со словарями, будут использовать словарь, присутствующий на том сервере, на котором они выполняются.
Другой пример - функция hostName вернёт имя сервера, на котором она выполняется, и это можно использовать для служебных целей - чтобы в запросе SELECT сделать GROUP BY по серверам.

Если функция в запросе выполняется на сервере-инициаторе запроса, а вам нужно, чтобы она выполнялась на удалённых серверах, вы можете обернуть её в агрегатную функцию any или добавить в ключ в GROUP BY.

Арифметические функции

Для всех арифметических функций, тип результата вычисляется, как минимальный числовой тип, который может вместить результат, если такой тип есть. Минимум берётся одновременно по числу бит, знаковости и "плавучести". Если бит не хватает, то берётся тип максимальной битности.

Пример:

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

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

Арифметические функции работают для любой пары типов из UInt8, UInt16, UInt32, UInt64, Int8, Int16, Int32, Int64, Float32, Float64.

Переполнение производится также, как в C++.

plus(a, b), оператор a + b

Вычисляет сумму чисел.
Также можно складывать целые числа с датой и датой-с-временем. В случае даты, прибавление целого числа означает прибавление соответствующего количества дней. В случае даты-с-временем - прибавление соответствующего количества секунд.

minus(a, b), оператор a - b

Вычисляет разность чисел. Результат всегда имеет знаковый тип.

Также можно вычитать целые числа из даты и даты-с-временем. Смысл аналогичен - смотрите выше для plus.

multiply(a, b), оператор a * b

Вычисляет произведение чисел.

divide(a, b), оператор a / b

Вычисляет частное чисел. Тип результата всегда является типом с плавающей запятой.
То есть, деление не целочисленное. Для целочисленного деления, используйте функцию intDiv.
При делении на ноль получится inf, -inf или nan.

intDiv(a, b)

Вычисляет частное чисел. Деление целочисленное, с округлением вниз (по абсолютному значению).
При делении на ноль или при делении минимального отрицательного числа на минус единицу, кидается исключение.

intDivOrZero(a, b)

Отличается от intDiv тем, что при делении на ноль или при делении минимального отрицательного числа на минус единицу, возвращается ноль.

modulo(a, b), оператор a % b

Вычисляет остаток от деления.
Если аргументы - числа с плавающей запятой, то они предварительно преобразуются в целые числа, путём отбрасывания дробной части.
Берётся остаток в том же смысле, как это делается в C++. По факту, для отрицательных чисел, используется truncated division.
При делении на ноль или при делении минимального отрицательного числа на минус единицу, кидается исключение.

negate(a), оператор -a

Вычисляет число, обратное по знаку. Результат всегда имеет знаковый тип.

abs(a)

Вычисляет абсолютное значение для числа a. То есть, если a < 0, то возвращает -a.
Для беззнаковых типов ничего не делает. Для чисел типа целых со знаком, возвращает число беззнакового типа.

gcd(a, b)

Вычисляет наибольший общий делитель чисел.
При делении на ноль или при делении минимального отрицательного числа на минус единицу, кидается исключение.

lcm(a, b)

Вычисляет наименьшее общее кратное чисел.
При делении на ноль или при делении минимального отрицательного числа на минус единицу, кидается исключение.

Функции сравнения

Функции сравнения возвращают всегда 0 или 1 (UInt8).

Сравнивать можно следующие типы:

- числа;
- строки и фиксированные строки;
- даты;
- даты-с-временем;

внутри каждой группы, но не из разных групп.

Например, вы не можете сравнить дату со строкой. Надо использовать функцию преобразования строки в дату или наоборот.

Строки сравниваются побайтово. Более короткая строка меньше всех строк, начинающихся с неё и содержащих ещё хотя бы один символ.

Замечание. До версии 1.1.54134 сравнение знаковых и беззнаковых целых чисел производилось также, как в C++. То есть, вы могли получить неверный результат в таких случаях: SELECT 9223372036854775807 > -1. С версии 1.1.54134 поведение изменилось и стало математически корректным.

equals, оператор a = b и a == b

notEquals, оператор a != b и a <> b

less, оператор <

greater, оператор >

lessOrEquals, оператор <=

greaterOrEquals, оператор >=

Логические функции

Логические функции принимают любые числовые типы, а возвращают число типа UInt8, равное 0 или 1.

Ноль в качестве аргумента считается "ложью", а любое ненулевое значение - "истиной".

and, оператор AND

or, оператор OR

not, оператор NOT

xor

Функции преобразования типов

toUInt8, toUInt16, toUInt32, toUInt64

toInt8, toInt16, toInt32, toInt64

toFloat32, toFloat64

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

toDate, toDateTime

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

Приводит строку или число value к типу Decimal указанной точности.
Параметр S (scale) определяет число десятичных знаков после запятой.

toString

Функции преобразования между числами, строками (но не фиксированными строками), датами и датами-с-временем.
Все эти функции принимают один аргумент.

При преобразовании в строку или из строки, производится форматирование или парсинг значения по тем же правилам, что и для формата TabSeparated (и почти всех остальных текстовых форматов). Если распарсить строку не удаётся - кидается исключение и выполнение запроса прерывается.

При преобразовании даты в число или наоборот, дате соответствует число дней от начала unix эпохи.
При преобразовании даты-с-временем в число или наоборот, дате-с-временем соответствует число секунд от начала unix эпохи.

Форматы даты и даты-с-временем для функций toDate/toDateTime определены следующим образом:

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

В качестве исключения, если делается преобразование из числа типа UInt32, Int32, UInt64, Int64 в Date, и если число больше или равно 65536, то число рассматривается как unix timestamp (а не как число дней) и округляется до даты. Это позволяет поддержать распространённый случай, когда пишут toDate(unix_timestamp), что иначе было бы ошибкой и требовало бы написания более громоздкого toDate(toDateTime(unix_timestamp))

Преобразование между датой и датой-с-временем производится естественным образом: добавлением нулевого времени или отбрасыванием времени.

Преобразование между числовыми типами производится по тем же правилам, что и присваивание между разными числовыми типами в C++.

Дополнительно, функция toString от аргумента типа DateTime может принимать второй аргумент String - имя тайм-зоны. Пример: Asia/Yekaterinburg В этом случае, форматирование времени производится согласно указанной тайм-зоне.

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

Также смотрите функцию toUnixTimestamp.

toFixedString(s, N)

Преобразует аргумент типа String в тип FixedString(N) (строку фиксированной длины N). N должно быть константой.
Если строка имеет меньше байт, чем N, то она дополняется нулевыми байтами справа. Если строка имеет больше байт, чем N - кидается исключение.

toStringCutToZero(s)

Принимает аргумент типа String или FixedString. Возвращает String, вырезая содержимое строки до первого найденного нулевого байта.

Пример:

```
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

Функции принимают строку и интерпретируют байты, расположенные в начале строки, как число в host order (little endian). Если строка имеет недостаточную длину, то функции работают так, как будто строка дополнена необходимым количеством нулевых байт. Если строка длиннее, чем нужно, то лишние байты игнорируются. Дата интерпретируется, как число дней с начала unix-эпохи, а дата-с-временем - как число секунд с начала unix-эпохи.

reinterpretAsString

Функция принимает число или дату или дату-с-временем и возвращает строку, содержащую байты, представляющие соответствующее значение в host order (little endian). При этом, отбрасываются нулевые байты с конца. Например, значение 255 типа UInt32 будет строкой длины 1 байт.

CAST(x, t)

Преобразует x в тип данных t.
Поддерживается также синтаксис CAST(x AS t).

Пример:

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

Преобразование в FixedString(N) работает только для аргументов типа String или FixedString(N).

Поддержано преобразование к типу Nullable и обратно. Пример:

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

Функции для работы с датами и временем

Поддержка часовых поясов

Все функции по работе с датой и временем, для которых это имеет смысл, могут принимать второй, необязательный аргумент - имя часового пояса. Пример: Asia/Yekaterinburg. В этом случае, они используют не локальный часовой пояс (по умолчанию), а указанный.

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

Поддерживаются только часовые пояса, отличающиеся от UTC на целое число часов.

toYear

Переводит дату или дату-с-временем в число типа UInt16, содержащее номер года (AD).

toMonth

Переводит дату или дату-с-временем в число типа UInt8, содержащее номер месяца (1-12).

toDayOfMonth

Переводит дату или дату-с-временем в число типа UInt8, содержащее номер дня в месяце (1-31).

toDayOfWeek

Переводит дату или дату-с-временем в число типа UInt8, содержащее номер дня в неделе (понедельник - 1, воскресенье - 7).

toHour

Переводит дату-с-временем в число типа UInt8, содержащее номер часа в сутках (0-23).
Функция исходит из допущения, что перевод стрелок вперёд, если осуществляется, то на час, в два часа ночи, а перевод стрелок назад, если осуществляется, то на час, в три часа ночи (что, в общем, не верно - даже в Москве два раза перевод стрелок был осуществлён в другое время).

toMinute

Переводит дату-с-временем в число типа UInt8, содержащее номер минуты в часе (0-59).

toSecond

Переводит дату-с-временем в число типа UInt8, содержащее номер секунды в минуте (0-59).
Секунды координации не учитываются.

toStartOfYear

Округляет дату или дату-с-временем вниз до первого дня года.
Возвращается дата.

toStartOfQuarter

Округляет дату или дату-с-временем вниз до первого дня квартала.
Первый день квартала - это одно из 1 января, 1 апреля, 1 июля, 1 октября.
Возвращается дата.

toStartOfMonth

Округляет дату или дату-с-временем вниз до первого дня месяца.
Возвращается дата.

Attention

Возвращаемое значение для некорректных дат зависит от реализации. ClickHouse может вернуть нулевую дату, выбросить исключение, или выполнить "естественное" перетекание дат между месяцами.

toMonday

Округляет дату или дату-с-временем вниз до ближайшего понедельника.
Возвращается дата.

toStartOfDay

Округляет дату-с-временем вниз до начала дня. Возвращается дата-с-временем.

toStartOfHour

Округляет дату-с-временем вниз до начала часа.

toStartOfMinute

Округляет дату-с-временем вниз до начала минуты.

toStartOfFiveMinute

Округляет дату-с-временем вниз до начала пятиминутного интервала.

toStartOfTenMinutes

Округляет дату-с-временем вниз до начала десятиминутного интервала.

toStartOfFifteenMinutes

Округляет дату-с-временем вниз до начала пятнадцатиминутного интервала.

toStartOfInterval(time_or_data, INTERVAL x unit [, time_zone])

Обобщение остальных функций toStartOf*. Например,
toStartOfInterval(t, INTERVAL 1 year) возвращает то же самое, что и toStartOfYear(t),
toStartOfInterval(t, INTERVAL 1 month) возвращает то же самое, что и toStartOfMonth(t),
toStartOfInterval(t, INTERVAL 1 day) возвращает то же самое, что и toStartOfDay(t),
toStartOfInterval(t, INTERVAL 15 minute) возвращает то же самое, что и toStartOfFifteenMinutes(t), и т.п.

toTime

Переводит дату-с-временем на некоторую фиксированную дату, сохраняя при этом время.

toRelativeYearNum

Переводит дату-с-временем или дату в номер года, начиная с некоторого фиксированного момента в прошлом.

toRelativeMonthNum

Переводит дату-с-временем или дату в номер месяца, начиная с некоторого фиксированного момента в прошлом.

toRelativeWeekNum

Переводит дату-с-временем или дату в номер недели, начиная с некоторого фиксированного момента в прошлом.

toRelativeDayNum

Переводит дату-с-временем или дату в номер дня, начиная с некоторого фиксированного момента в прошлом.

toRelativeHourNum

Переводит дату-с-временем в номер часа, начиная с некоторого фиксированного момента в прошлом.

toRelativeMinuteNum

Переводит дату-с-временем в номер минуты, начиная с некоторого фиксированного момента в прошлом.

toRelativeSecondNum

Переводит дату-с-временем в номер секунды, начиная с некоторого фиксированного момента в прошлом.

now

Принимает ноль аргументов и возвращает текущее время на один из моментов выполнения запроса.
Функция возвращает константу, даже если запрос выполнялся долго.

today

Принимает ноль аргументов и возвращает текущую дату на один из моментов выполнения запроса.
То же самое, что toDate(now())

yesterday

Принимает ноль аргументов и возвращает вчерашнюю дату на один из моментов выполнения запроса.
Делает то же самое, что today() - 1.

timeSlot

Округляет время до получаса.
Эта функция является специфичной для Яндекс.Метрики, так как пол часа - минимальное время, для которого, если соседние по времени хиты одного посетителя на одном счётчике отстоят друг от друга строго более, чем на это время, визит может быть разбит на два визита. То есть, кортежи (номер счётчика, идентификатор посетителя, тайм-слот) могут использоваться для поиска хитов, входящий в соответствующий визит.

timeSlots(StartTime, Duration,[, Size])

Для интервала времени, начинающегося в 'StartTime' и продолжающегося 'Duration' секунд, возвращает массив моментов времени, состоящий из округлений вниз до 'Size' точек в секундах из этого интервала.
'Size' - необязательный параметр, константный UInt32, по умолчанию равен 1800.

Например, timeSlots(toDateTime('2012-01-01 12:20:00'), toUInt32(600)) = [toDateTime('2012-01-01 12:00:00'), toDateTime('2012-01-01 12:30:00')].
Это нужно для поиска хитов, входящих в соответствующий визит.

formatDateTime(Time, Format[, Timezone])

Функция преобразования даты-с-временем в String согласно заданному шаблону. Важно - шаблон является константным выражением, т.е. невозможно использование разных шаблонов в одной колонке.

Поддерживаемые модификаторы в шаблоне Format:
(колонка "Пример" показана для времени 2018-01-02 22:33:44)

Модификатор	Описание	Пример
%C	номер года, поделённый на 100 (00-99)	20
%d	день месяца, с ведущим нулём (01-31)	02
%D	короткая запись %m/%d/%y	01/02/2018
%e	день месяца, с ведущим пробелом (1-31)	2
%F	короткая запись %Y-%m-%d	2018-01-02
%H	час в 24-часовом формате (00-23)	22
%I	час в 12-часовом формате (01-12)	10
%j	номер дня в году, с ведущими нулями (001-366)	002
%m	месяц, с ведущим нулём (01-12)	01
%M	минуты, с ведущим нулём (00-59)	33
%n	символ переноса строки ('\n')	
%p	обозначения AM или PM	PM
%R	короткая запись %H:%M	22:33
%S	секунды, с ведущими нулями (00-59)	44
%t	символ табуляции ('\t')	
%T	формат времени ISO 8601, одинаковый с %H:%M:%S	22:33:44
%u	номер дня недели согласно ISO 8601, понедельник - 1, воскресенье - 7	2
%V	номер недели согласно ISO 8601 (01-53)	01
%w	номер дня недели, начиная с воскресенья (0-6)	2
%y	год, последние 2 цифры (00-99)	18
%Y	год, 4 цифры	2018
%%	символ %	%

Функции для работы со строками

empty

Возвращает 1 для пустой строки, и 0 для непустой строки.
Тип результата - UInt8.
Строка считается непустой, если содержит хотя бы один байт, пусть даже это пробел или нулевой байт.
Функция также работает для массивов.

notEmpty

Возвращает 0 для пустой строки, и 1 для непустой строки.
Тип результата - UInt8.
Функция также работает для массивов.

length

Возвращает длину строки в байтах (не символах, не кодовых точках).
Тип результата - UInt64.
Функция также работает для массивов.

lengthUTF8

Возвращает длину строки в кодовых точках Unicode (не символах), при допущении, что строка содержит набор байт, являющийся текстом в кодировке UTF-8. Если допущение не выполнено - то возвращает какой-нибудь результат (не кидает исключение).
Тип результата - UInt64.

lower

Переводит ASCII-символы латиницы в строке в нижний регистр.

upper

Переводит ASCII-символы латиницы в строке в верхний регистр.

lowerUTF8

Переводит строку в нижний регистр, при допущении, что строка содержит набор байт, представляющий текст в кодировке UTF-8.
Не учитывает язык. То есть, для турецкого языка, результат может быть не совсем верным.
Если длина UTF-8 последовательности байт различна для верхнего и нижнего регистра кодовой точки, то для этой кодовой точки, результат работы может быть некорректным.
Если строка содержит набор байт, не являющийся UTF-8, то поведение не определено.

upperUTF8

Переводит строку в верхний регистр, при допущении, что строка содержит набор байт, представляющий текст в кодировке UTF-8.
Не учитывает язык. То есть, для турецкого языка, результат может быть не совсем верным.
Если длина UTF-8 последовательности байт различна для верхнего и нижнего регистра кодовой точки, то для этой кодовой точки, результат работы может быть некорректным.
Если строка содержит набор байт, не являющийся UTF-8, то поведение не определено.

isValidUTF8

Возвращает 1, если набор байт является корректным в кодировке UTF-8, 0 иначе.

toValidUTF8

Заменяет некорректные символы UTF-8 на символ `&#65533;` (U+FFFD). Все идущие подряд некорректные символы схлопываются в один заменяющий символ.

```
toValidUTF8( input_string )
```

Параметры:

- input_string — произвольный набор байтов, представленный как объект типа String.

Возвращаемое значение: Корректная строка UTF-8.

Пример

```
SELECT toValidUTF8('\x61\xF0\x80\x80\x80b')
```

```
┌─toValidUTF8('a����b')─┐
│ a�b                   │
└───────────────────────┘
```

reverse

Разворачивает строку (как последовательность байт).

reverseUTF8

Разворачивает последовательность кодовых точек Unicode, при допущении, что строка содержит набор байт, представляющий текст в кодировке UTF-8. Иначе - что-то делает (не кидает исключение).

format(pattern, s0, s1, ...)

Форматирует константный шаблон со строками, перечисленными в аргументах. pattern -- упрощенная версия шаблона в языке Python. Шаблон содержит "заменяющие поля", которые окружены фигурными скобками {}. Всё, что не содержится в скобках, интерпретируется как обычный текст и просто копируется. Если нужно использовать символ фигурной скобки, можно экранивать двойной скобкой {{ или }}. Имя полей могут быть числами (нумерация с нуля) или пустыми (тогда они интерпретируются как последовательные числа).

```
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, ...)

Склеивает строки, перечисленные в аргументах, без разделителей.

substring(s, offset, length)

Возвращает подстроку, начиная с байта по индексу offset, длины length байт. Индексация символов - начиная с единицы (как в стандартном SQL). Аргументы offset и length должны быть константами.

substringUTF8(s, offset, length)

Так же, как substring, но для кодовых точек Unicode. Работает при допущении, что строка содержит набор байт, представляющий текст в кодировке UTF-8. Если допущение не выполнено - то возвращает какой-нибудь результат (не кидает исключение).

appendTrailingCharIfAbsent(s, c)

Если строка s непустая и не содержит символ c на конце, то добавляет символ c в конец.

convertCharset(s, from, to)

Возвращает сконвертированную из кодировки from в кодировку to строку s.

base64Encode(s)

Производит кодирование строки s в base64-представление.

base64Decode(s)

Декодирует base64-представление s в исходную строку. При невозможности декодирования выбрасывает исключение

tryBase64Decode(s)

Функционал аналогичен base64Decode, но при невозможности декодирования возвращает пустую строку.

Функции поиска в строках

Во всех функциях, поиск регистрозависимый по-умолчанию. Существуют варианты функций для регистронезависимого поиска.

position(haystack, needle)

Поиск подстроки needle в строке haystack.
Возвращает позицию (в байтах) найденной подстроки, начиная с 1, или 0, если подстрока не найдена.

Для поиска без учета регистра используйте функцию positionCaseInsensitive.

positionUTF8(haystack, needle)

Так же, как position, но позиция возвращается в кодовых точках Unicode. Работает при допущении, что строка содержит набор байт, представляющий текст в кодировке UTF-8. Если допущение не выполнено -- то возвращает какой-нибудь результат (не кидает исключение).

Для поиска без учета регистра используйте функцию positionCaseInsensitiveUTF8.

multiSearchAllPositions(haystack, [$needle_1$, $needle_2$, ..., $needle_n$])

Так же, как и position, только возвращает Array первых вхождений.

Для поиска без учета регистра и/или в кодировке UTF-8 используйте функции multiSearchAllPositionsCaseInsensitive, multiSearchAllPositionsUTF8, multiSearchAllPositionsCaseInsensitiveUTF8.

multiSearchFirstPosition(haystack, [$needle_1$, $needle_2$, ..., $needle_n$])

Так же, как и position, только возвращает оффсет первого вхождения любого из needles.

Для поиска без учета регистра и/или в кодировке UTF-8 используйте функции multiSearchFirstPositionCaseInsensitive, multiSearchFirstPositionUTF8, multiSearchFirstPositionCaseInsensitiveUTF8.

multiSearchFirstIndex(haystack, [$needle_1$, $needle_2$, ..., $needle_n$])

Возвращает индекс i (нумерация с единицы) первой найденной строки $needle_i$ в строке haystack и 0 иначе.

Для поиска без учета регистра и/или в кодировке UTF-8 используйте функции multiSearchFirstIndexCaseInsensitive, multiSearchFirstIndexUTF8, multiSearchFirstIndexCaseInsensitiveUTF8.

multiSearchAny(haystack, [$needle_1$, $needle_2$, ..., $needle_n$])

Возвращает 1, если хотя бы одна подстрока $needle_i$ нашлась в строке haystack и 0 иначе.

Для поиска без учета регистра и/или в кодировке UTF-8 используйте функции multiSearchAnyCaseInsensitive, multiSearchAnyUTF8, multiSearchAnyCaseInsensitiveUTF8.

Примечание: во всех функциях multiSearch* количество needles должно быть меньше 2^8 из-за особенностей реализации.

match(haystack, pattern)

Проверка строки на соответствие регулярному выражению pattern. Регулярное выражение **re2**. Синтаксис регулярных выражений **re2** является более ограниченным по сравнению с регулярными выражениями **Perl** (подробнее).
Возвращает 0 (если не соответствует) или 1 (если соответствует).

Обратите внимание, что для экранирования в регулярном выражении, используется символ \ (обратный слеш). Этот же символ используется для экранирования в строковых литералах. Поэтому, чтобы экранировать символ в регулярном выражении, необходимо написать в строковом литерале \ (два обратных слеша).

Регулярное выражение работает со строкой как с набором байт. Регулярное выражение не может содержать нулевые байты.
Для шаблонов на поиск подстроки в строке, лучше используйте LIKE или position, так как они работают существенно быстрее.

multiMatchAny(haystack, [$pattern_1$, $pattern_2$, ..., $pattern_n$])

То же, что и match, но возвращает ноль, если ни одно регулярное выражение не подошло и один, если хотя бы одно. Используется библиотека hyperscan для соответствия регулярных выражений. Для шаблонов на поиск многих подстрок в строке, лучше используйте multiSearchAny, так как она работает существенно быстрее.

Примечание: длина любой строки из haystack должна быть меньше 2^{32} байт, иначе бросается исключение. Это ограничение связано с ограничением hyperscan API.

multiMatchAnyIndex(haystack, [$pattern_1$, $pattern_2$, ..., $pattern_n$])

То же, что и multiMatchAny, только возвращает любой индекс подходящего регулярного выражения.

multiFuzzyMatchAny(haystack, distance, [$pattern_1$, $pattern_2$, ..., $pattern_n$])

То же, что и multiMatchAny, но возвращает 1 если любой pattern соответствует haystack в пределах константного редакционного расстояния. Эта функция также находится в экспериментальном режиме и может быть очень медленной. За подробностями обращайтесь к документации hyperscan.

multiFuzzyMatchAnyIndex(haystack, distance, [$pattern_1$, $pattern_2$, ..., $pattern_n$])

То же, что и multiFuzzyMatchAny, только возвращает любой индекс подходящего регулярного выражения в пределах константного редакционного расстояния.

Примечание: multiFuzzyMatch* функции не поддерживают UTF-8 закодированные регулярные выражения, и такие выражения рассматриваются как байтовые из-за ограничения hyperscan.

Примечание: чтобы выключить все функции, использующие hyperscan, используйте настройку SET allow_hyperscan = 0;.

extract(haystack, pattern)

Извлечение фрагмента строки по регулярному выражению. Если haystack не соответствует регулярному выражению pattern, то возвращается пустая строка. Если регулярное выражение не содержит subpattern-ов, то вынимается фрагмент, который подпадает под всё регулярное выражение. Иначе вынимается фрагмент, который подпадает под первый subpattern.

extractAll(haystack, pattern)

Извлечение всех фрагментов строки по регулярному выражению. Если haystack не соответствует регулярному выражению pattern, то возвращается пустая строка. Возвращается массив строк, состоящий из всех соответствий регулярному выражению. В остальном, поведение аналогично функции extract (по прежнему, вынимается первый subpattern, или всё выражение, если subpattern-а нет).

like(haystack, pattern), оператор haystack LIKE pattern

Проверка строки на соответствие простому регулярному выражению.
Регулярное выражение может содержать метасимволы % и _.

% обозначает любое количество любых байт (в том числе, нулевое количество символов).

_ обозначает один любой байт.

Для экранирования метасимволов, используется символ \ (обратный слеш). Смотрите замечание об экранировании в описании функции match.

Для регулярных выражений вида %needle% действует более оптимальный код, который работает также быстро, как функция position.
Для остальных регулярных выражений, код аналогичен функции match.

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

То же, что like, но с отрицанием.

ngramDistance(haystack, needle)

Вычисление 4-граммного расстояния между haystack и needle: считается симметрическая разность между двумя мультимножествами 4-грамм и нормализается на сумму их мощностей. Возвращает число float от 0 до 1 -- чем ближе к нулю, тем больше строки похожи друг на друга. Если константный needle или haystack больше чем 32КБ, кидается исключение. Если некоторые строки из неконстантного haystack или needle больше 32КБ, расстояние всегда равно единице.

Для поиска без учета регистра и/или в формате UTF-8 используйте функции ngramDistanceCaseInsensitive, ngramDistanceUTF8, ngramDistanceCaseInsensitiveUTF8.

ngramSearch(haystack, needle)

То же, что и ngramDistance, но вычисляет несимметричную разность между needle и haystack -- количество n-грамм из needle минус количество общих n-грамм, нормированное на количество n-грамм из needle. Может быть использовано для приближенного поиска.

Для поиска без учета регистра и/или в формате UTF-8 используйте функции ngramSearchCaseInsensitive, ngramSearchUTF8, ngramSearchCaseInsensitiveUTF8.

Примечание: для случая UTF-8 мы используем триграммное расстояние. Вычисление n-граммного расстояния не совсем честное. Мы используем 2-х байтные хэши для хэширования n-грамм, а затем вычисляем (не)симметрическую разность между хэш таблицами -- могут возникнуть коллизии. В формате UTF-8 без учета регистра мы не используем честную функцию tolower -- мы обнуляем 5-й бит (нумерация с нуля) каждого байта кодовой точки, а также первый бит нулевого байта, если байтов больше 1 -- это работает для латиницы и почти для всех кириллических букв.

Функции поиска и замены в строках

replaceOne(haystack, pattern, replacement)

Замена первого вхождения, если такое есть, подстроки pattern в haystack на подстроку replacement.
Здесь и далее, pattern и replacement должны быть константами.

replaceAll(haystack, pattern, replacement)

Замена всех вхождений подстроки pattern в haystack на подстроку replacement.

replaceRegexpOne(haystack, pattern, replacement)

Замена по регулярному выражению pattern. Регулярное выражение re2.
Заменяется только первое вхождение, если есть.
В качестве replacement может быть указан шаблон для замен. Этот шаблон может включать в себя подстановки \0-\9.
Подстановка \0 - вхождение регулярного выражения целиком. Подстановки \1-\9 - соответствующие по номеру subpattern-ы.
Для указания символа \ в шаблоне, он должен быть экранирован с помощью символа \.
Также помните о том, что строковый литерал требует ещё одно экранирование.

Пример 1. Переведём дату в американский формат:

```
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
```

Пример 2. Размножить строку десять раз:

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

```
┌─res────────────────────────────────────────────────────────────────────────────────────────────────┐
│ Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello, World!Hello, World! │
└────────────────────────────────────────────────────────────────────────────────────────────────┘
```

replaceRegexpAll(haystack, pattern, replacement)

То же самое, но делается замена всех вхождений. Пример:

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

```
┌─res────────────────────────┐
│ HHeelllloo,,  WWoorrlldd!! │
└────────────────────────────┘
```

В качестве исключения, если регулярное выражение сработало на пустой подстроке, то замена делается не более одного раза.
Пример:

```
SELECT replaceRegexpAll('Hello, World!', '^', 'here: ') AS res
```

```
┌─res─────────────────┐
│ here: Hello, World! │
└─────────────────────┘
```

Условные функции

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

Возвращает then, если cond != 0 или else, если cond = 0.
cond должно иметь тип UInt8, а then и else должны иметь тип, для которого есть наименьший общий тип.

then и else могут быть NULL

multiIf

Позволяет более компактно записать оператор CASE в запросе.

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

Параметры

- cond_N — Условие, при выполнении которого функция вернёт then_N.

- then_N — Результат функции при выполнении.
- else — Результат функции, если ни одно из условий не выполнено.

Функция принимает 2N+1 параметров.

Возвращаемые значения

Функция возвращает одно из значений then_N или else, в зависимости от условий cond_N.

Пример

Рассмотрим таблицу

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

Выполним запрос SELECT multiIf(isNull(y), x, y < 3, y, NULL) FROM t_null. Результат:

```
┌─multiIf(isNull(y), x, less(y, 3), y, NULL)─┐
│                                          1 │
│                                       NULL │
└────────────────────────────────────────────┘
```

Математические функции

Все функции возвращают число типа Float64. Точность результата близка к максимально возможной, но результат может не совпадать с наиболее близким к соответствующему вещественному числу машинно представимым числом.

e()

Возвращает число типа Float64, близкое к числу e.

pi()

Возвращает число типа Float64, близкое к числу π.

exp(x)

Принимает числовой аргумент, возвращает число типа Float64, близкое к экспоненте от аргумента.

log(x)

Принимает числовой аргумент, возвращает число типа Float64, близкое к натуральному логарифму от аргумента.

exp2(x)

Принимает числовой аргумент, возвращает число типа Float64, близкое к 2 в степени x.

log2(x)

Принимает числовой аргумент, возвращает число типа Float64, близкое к двоичному логарифму от аргумента.

exp10(x)

Принимает числовой аргумент, возвращает число типа Float64, близкое к 10 в степени x.

log10(x)

Принимает числовой аргумент, возвращает число типа Float64, близкое к десятичному логарифму от аргумента.

sqrt(x)

Принимает числовой аргумент, возвращает число типа Float64, близкое к квадратному корню от аргумента.

cbrt(x)

Принимает числовой аргумент, возвращает число типа Float64, близкое к кубическому корню от аргумента.

erf(x)

Если x неотрицательно, то erf(x / σ√2) - вероятность того, что случайная величина, имеющая нормальное распределение со среднеквадратичным отклонением σ, принимает значение, отстоящее от мат. ожидания больше чем на x.

Пример (правило трёх сигм):

```
SELECT erf(3 / sqrt(2))
```

```
┌─erf(divide(3, sqrt(2)))─┐
│      0.9973002039367398 │
└─────────────────────────┘
```

erfc(x)

Принимает числовой аргумент, возвращает число типа Float64, близкое к 1 - erf(x), но без потери точности для больших x.

lgamma(x)

Логарифм от гамма функции.

tgamma(x)

Гамма функция.

sin(x)

Синус.

cos(x)

Косинус.

tan(x)

Тангенс.

asin(x)

Арксинус.

acos(x)

Арккосинус.

atan(x)

Арктангенс.

pow(x, y)

Принимает два числовых аргумента x и y. Возвращает число типа Float64, близкое к x в степени y.

Функции округления

floor(x[, N])

Возвращает наибольшее круглое число, которое меньше или равно, чем x.
Круглым называется число, кратное 1 / 10N или ближайшее к нему число соответствующего типа данных, если 1 / 10N не представимо точно.
N - целочисленная константа, не обязательный параметр. По умолчанию - ноль, что означает - округлять до целого числа.
N может быть отрицательным.

Примеры: floor(123.45, 1) = 123.4, floor(123.45, -1) = 120.

x - любой числовой тип. Результат - число того же типа.
Для целочисленных аргументов имеет смысл округление с отрицательным значением N (для неотрицательных N, функция ничего не делает).
В случае переполнения при округлении (например, floor(-128, -1)), возвращается implementation specific результат.

ceil(x[, N])

Возвращает наименьшее круглое число, которое больше или равно, чем x.
В остальном, аналогично функции floor, см. выше.

round(x[, N])

Округляет значение до указанного десятичного разряда.

Функция возвращает ближайшее значение указанного порядка. В случае, когда заданное число равноудалено от чисел необходимого порядка, функция возвращает то из них, которое имеет ближайшую чётную цифру (банковское округление).

```
round(expression [, decimal_places])
```

Параметры:

- expression — Число для округления. Может быть любым выражением, возвращающим числовой тип данных.
- decimal-places — Целое значение.
  - Если decimal-places > 0, то функция округляет значение справа от запятой.
  - Если decimal-places < 0 то функция округляет значение слева от запятой.
  - Если decimal-places = 0, то функция округляет значение до целого. В этом случае аргумент можно опустить.

Возвращаемое значение:

Округлённое значение того же типа, что и входящее.

Примеры

Пример использования

```
SELECT number / 2 AS x, round(x) FROM system.numbers LIMIT 3
```

```
┌───x─┬─round(divide(number, 2))─┐
│   0 │                        0 │
│ 0.5 │                        0 │
│   1 │                        1 │
└─────┴──────────────────────────┘
```

Примеры округления

Округление до ближайшего числа.

```
round(3.2, 0) = 3
round(4.1267, 2) = 4.13
round(22,-1) = 20
round(467,-2) = 500
round(-467,-2) = -500
```

Банковское округление.

```
round(3.5) = 4
round(4.5) = 4
round(3.55, 1) = 3.6
round(3.65, 1) = 3.6
```

roundToExp2(num)

Принимает число. Если число меньше единицы - возвращает 0. Иначе округляет число вниз до ближайшей (целой неотрицательной) степени двух.

roundDuration(num)

Принимает число. Если число меньше единицы - возвращает 0. Иначе округляет число вниз до чисел из набора: 1, 10, 30, 60, 120, 180, 240, 300, 600, 1200, 1800, 3600, 7200, 18000, 36000. Эта функция специфична для Яндекс.Метрики и предназначена для реализации отчёта по длительности визита.

roundAge(num)

Принимает число. Если число меньше 18 - возвращает 0. Иначе округляет число вниз до чисел из набора: 18, 25, 35, 45, 55. Эта функция специфична для Яндекс.Метрики и предназначена для реализации отчёта по возрасту посетителей.

Функции по работе с массивами

empty

Возвращает 1 для пустого массива, и 0 для непустого массива.
Тип результата - UInt8.
Функция также работает для строк.

notEmpty

Возвращает 0 для пустого массива, и 1 для непустого массива.
Тип результата - UInt8.
Функция также работает для строк.

length

Возвращает количество элементов в массиве.
Тип результата - UInt64.
Функция также работает для строк.

emptyArrayUInt8, emptyArrayUInt16, emptyArrayUInt32, emptyArrayUInt64

emptyArrayInt8, emptyArrayInt16, emptyArrayInt32, emptyArrayInt64

emptyArrayFloat32, emptyArrayFloat64

emptyArrayDate, emptyArrayDateTime

emptyArrayString

Принимает ноль аргументов и возвращает пустой массив соответствующего типа.

emptyArrayToSingle

Принимает пустой массив и возвращает массив из одного элемента, равного значению по умолчанию.

range(N)

Возвращает массив чисел от 0 до N-1.
На всякий случай, если на блок данных, создаются массивы суммарной длины больше 100 000 000 элементов, то кидается исключение.

array(x1, ...), оператор [x1, ...]

Создаёт массив из аргументов функции.
Аргументы должны быть константами и иметь типы, для которых есть наименьший общий тип. Должен быть передан хотя бы один аргумент, так как иначе непонятно, какого типа создавать массив. То есть, с помощью этой функции невозможно создать пустой массив (для этого используйте функции emptyArray*, описанные выше).
Возвращает результат типа Array(T), где T - наименьший общий тип от переданных аргументов.

arrayConcat

Объединяет массивы, переданные в качестве аргументов.

```
arrayConcat(arrays)
```

Параметры

- arrays – произвольное количество элементов типа Array
  Пример

```
SELECT arrayConcat([1, 2], [3, 4], [5, 6]) AS res
```

```
┌─res─────────────┐
│ [1,2,3,4,5,6] │
└─────────────────┘
```

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

Достаёт элемент с индексом n из массива arr. n должен быть любым целочисленным типом.
Индексы в массиве начинаются с единицы.
Поддерживаются отрицательные индексы. В этом случае, будет выбран соответствующий по номеру элемент с конца. Например, arr[-1] - последний элемент массива.

Если индекс выходит за границы массива, то возвращается некоторое значение по умолчанию (0 для чисел, пустая строка для строк и т. п.).

has(arr, elem)

Проверяет наличие элемента elem в массиве arr.
Возвращает 0, если элемента в массиве нет, или 1, если есть.

NULL обрабатывается как значение.

```
SELECT has([1, 2, NULL], NULL)

┌─has([1, 2, NULL], NULL)─┐
│                       1 │
└─────────────────────────┘
```

hasAll

Проверяет, является ли один массив подмножеством другого.

```
hasAll(set, subset)
```

Параметры

- set – массив любого типа с набором элементов.
- subset – массив любого типа со значениями, которые проверяются на вхождение в set.

Возвращаемые значения

- 1, если set содержит все элементы из subset.
- 0, в противном случае.

Особенности

- Пустой массив является подмножеством любого массива.
- NULL обрабатывается как значение.
- Порядок значений в обоих массивах не имеет значения.

Примеры

SELECT hasAll([], []) возвращает 1.

SELECT hasAll([1, Null], [Null]) возвращает 1.

SELECT hasAll([1.0, 2, 3, 4], [1, 3]) возвращает 1.

SELECT hasAll(['a', 'b'], ['a']) возвращает 1.

SELECT hasAll([1], ['a']) возвращает 0.

SELECT hasAll([[1, 2], [3, 4]], [[1, 2], [3, 5]]) возвращает 0.

hasAny

Проверяет, имеют ли два массива хотя бы один общий элемент.

```
hasAny(array1, array2)
```

Параметры

- array1 – массив любого типа с набором элементов.
- array2 – массив любого типа с набором элементов.

Возвращаемые значения

- 1, если array1 и array2 имеют хотя бы один одинаковый элемент.
- 0, в противном случае.

Особенности

- NULL обрабатывается как значение.
- Порядок значений в обоих массивах не имеет значения.

Примеры

SELECT hasAny([1], []) возвращает 0.

SELECT hasAny([Null], [Null, 1]) возвращает 1.

SELECT hasAny([-128, 1., 512], [1]) возвращает 1.

SELECT hasAny([[1, 2], [3, 4]], ['a', 'c']) возвращает 0.

SELECT hasAll([[1, 2], [3, 4]], [[1, 2], [1, 2]]) возвращает 1.

indexOf(arr, x)

Возвращает индекс первого элемента x (начиная с 1), если он есть в массиве, или 0, если его нет.

Пример:

```
:) SELECT indexOf([1,3,NULL,NULL],NULL)

SELECT indexOf([1, 3, NULL, NULL], NULL)

┌─indexOf([1, 3, NULL, NULL], NULL)─┐
│                                 3 │
└───────────────────────────────────┘
```

Элементы, равные NULL, обрабатываются как обычные значения.

countEqual(arr, x)

Возвращает количество элементов массива, равных x. Эквивалентно arrayCount(elem -> elem = x, arr).

NULL обрабатывается как значение.

Пример:

```
SELECT countEqual([1, 2, NULL, NULL], NULL)

┌─countEqual([1, 2, NULL, NULL], NULL)─┐
│                                    2 │
└──────────────────────────────────────┘
```

arrayEnumerate(arr)

Возвращает массив [1, 2, 3, ..., length(arr)]

Эта функция обычно используется совместно с ARRAY JOIN. Она позволяет, после применения ARRAY JOIN, посчитать что-либо только один раз для каждого массива. Пример:

```
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 │
└─────────┴───────┘
```

В этом примере, Reaches - число достижений целей (строк, получившихся после применения ARRAY JOIN), а Hits - число хитов (строк, которые были до ARRAY JOIN). В данном случае, тот же результат можно получить проще:

```
SELECT
    sum(length(GoalsReached)) AS Reaches,
    count() AS Hits
FROM test.hits
WHERE (CounterID = 160656) AND notEmpty(GoalsReached)
```

```
┌─Reaches─┬──Hits─┐
│   95606 │ 31406 │
└─────────┴───────┘
```

Также эта функция может быть использована в функциях высшего порядка. Например, с её помощью можно достать индексы массива для элементов, удовлетворяющих некоторому условию.

arrayEnumerateUniq(arr, ...)

Возвращает массив, такого же размера, как исходный, где для каждого элемента указано, какой он по счету среди элементов с таким же значением.
Например: arrayEnumerateUniq([10, 20, 10, 30]) = [1, 1, 2, 1].

Эта функция полезна при использовании ARRAY JOIN и агрегации по элементам массива.
Пример:

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

В этом примере, для каждого идентификатора цели, посчитано количество достижений целей (каждый элемент вложенной структуры данных Goals является достижением целей) и количество визитов. Если бы не было ARRAY JOIN, мы бы считали количество визитов как sum(Sign). Но в данном случае, строчки были размножены по вложенной структуре Goals, и чтобы после этого учесть каждый визит один раз, мы поставили условие на значение функции arrayEnumerateUniq(Goals.ID).

Функция arrayEnumerateUniq может принимать несколько аргументов - массивов одинаковых размеров. В этом случае, уникальность считается для кортежей элементов на одинаковых позициях всех массивов.

```
SELECT arrayEnumerateUniq([1, 1, 1, 2, 2, 2], [1, 1, 2, 1, 1, 2]) AS res
```

```
┌─res─────────┐
│ [1,2,1,1,2,1] │
└─────────────┘
```

Это нужно при использовании ARRAY JOIN с вложенной структурой данных и затем агрегации по нескольким элементам этой структуры.

arrayPopBack

Удаляет последний элемент из массива.

```
arrayPopBack(array)
```

Параметры

- array - Массив.

Пример

```
SELECT arrayPopBack([1, 2, 3]) AS res
```

```
┌─res───┐
│ [1,2] │
└───────┘
```

arrayPopFront

Удаляет первый элемент из массива.

```
arrayPopFront(array)
```

Параметры

- array - Массив.

Пример

```
SELECT arrayPopFront([1, 2, 3]) AS res
```

```
┌─res───┐
│ [2,3] │
└───────┘
```

arrayPushBack

Добавляет один элемент в конец массива.

```
arrayPushBack(array, single_value)
```

Параметры

- array - Массив.
- single_value - Одиночное значение. В массив с числам можно добавить только числа, в массив со строками только строки. При добавлении чисел ClickHouse автоматически приводит тип single_value к типу данных массива. Подробнее о типах данных в ClickHouse читайте в разделе "Типы данных". Может быть равно NULL. Функция добавит элемент NULL в массив, а тип элементов массива преобразует в Nullable.

Пример

```
SELECT arrayPushBack(['a'], 'b') AS res
```

```
┌─res───────┐
│ ['a','b'] │
└───────────┘
```

arrayPushFront

Добавляет один элемент в начало массива.

```
arrayPushFront(array, single_value)
```

Параметры

- array - Массив.
- single_value - Одиночное значение. В массив с числам можно добавить только числа, в массив со строками только строки. При добавлении чисел ClickHouse автоматически приводит тип single_value к типу данных массива. Подробнее о типах данных в ClickHouse читайте в разделе "Типы данных". Может быть равно NULL. Функция добавит элемент NULL в массив, а тип элементов массива преобразует в Nullable.

Пример

```
SELECT arrayPushBack(['b'], 'a') AS res
```

```
┌─res─────┐
│ ['a','b'] │
└─────────┘
```

arrayResize

Изменяет длину массива.

```
arrayResize(array, size[, extender])
```

Параметры

- array — массив.
- size — необходимая длина массива.
  - Если size меньше изначального размера массива, то массив обрезается справа.
  - Если size больше изначального размера массива, массив дополняется справа значениями extender или значениями по умолчанию для типа данных элементов массива.
- extender — значение для дополнения массива. Может быть NULL.

Возвращаемое значение:

Массив длины size.

Примеры вызовов

```
SELECT arrayResize([1], 3)

┌─arrayResize([1], 3)─┐
│ [1,0,0]             │
└─────────────────────┘
```

```
SELECT arrayResize([1], 3, NULL)

┌─arrayResize([1], 3, NULL)─┐
│ [1,NULL,NULL]             │
└───────────────────────────┘
```

arraySlice

Возвращает срез массива.

```
arraySlice(array, offset[, length])
```

Параметры

- array - Массив данных.
- offset - Отступ от края массива. Положительное значение - отступ слева, отрицательное значение - отступ справа. Отсчет элементов массива начинается с 1.
- length - Длина необходимого среза. Если указать отрицательное значение, то функция вернёт открытый срез [offset, array_length - length). Если не указать значение, то функция вернёт срез [offset, the_end_of_array].

Пример

```
SELECT arraySlice([1, 2, NULL, 4, 5], 2, 3) AS res
```

```
┌─res─────┐
│ [2,NULL,4] │
└─────────┘
```

Элементы массива равные NULL обрабатываются как обычные значения.

arraySort([func,] arr, ...)

Возвращает массив arr, отсортированный в восходящем порядке. Если задана функция func, то порядок сортировки определяется результатом применения этой функции на элементы массива arr. Если func принимает несколько аргументов, то в функцию arraySort нужно передавать несколько массивов, которые будут соответствовать аргументам функции func. Подробные примеры рассмотрены в конце описания arraySort.

Пример сортировки целочисленных значений:

```
SELECT arraySort([1, 3, 3, 0])
```

```
┌─arraySort([1, 3, 3, 0])─┐
│ [0,1,3,3]               │
└─────────────────────────┘
```

Пример сортировки строковых значений:

```
SELECT arraySort(['hello', 'world', '!'])
```

```
┌─arraySort(['hello', 'world', '!'])─┐
│ ['!','hello','world']              │
└────────────────────────────────────┘
```

Значения NULL, NaN и Inf сортируются по следующему принципу:

```
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]                     │
└───────────────────────────────────────────────────────────┘
```

- Значения -Inf идут в начале массива.
- Значения NULL идут в конце массива.
- Значения NaN идут перед NULL.
- Значения Inf идут перед NaN.

Функция arraySort является функцией высшего порядка — в качестве первого аргумента ей можно передать лямбда-функцию. В этом случае порядок сортировки определяется результатом применения лямбда-функции на элементы массива.

Рассмотрим пример:

```
SELECT arraySort((x) -> -x, [1, 2, 3]) as res;
```

```
┌─res─────┐
│ [3,2,1] │
└─────────┘
```

Для каждого элемента исходного массива лямбда-функция возвращает ключ сортировки, то есть [1 –> -1, 2 –> -2, 3 –> -3]. Так как arraySort сортирует элементы в порядке возрастания ключей, результат будет [3, 2, 1]. Как можно заметить, функция x –> -x устанавливает обратный порядок сортировки.

Лямбда-функция может принимать несколько аргументов. В этом случае, в функцию arraySort нужно передавать несколько массивов, которые будут соответствовать аргументам лямбда-функции (массивы должны быть одинаковой длины). Следует иметь в виду, что результат будет содержать элементы только из первого массива; элементы из всех последующих массивов будут задавать ключи сортировки. Например:

```
SELECT arraySort((x, y) -> y, ['hello', 'world'], [2, 1]) as res;
```

```
┌─res────────────────┐
│ ['world', 'hello'] │
└────────────────────┘
```

Элементы, указанные во втором массиве ([2,1]), определяют ключ сортировки для элементов из исходного массива (['hello', 'world']), то есть ['hello' –> 2, 'world' –> 1]. Так как лямбда-функция не использует x, элементы исходного массива не влияют на порядок сортировки. Таким образом, 'hello' будет вторым элементом в отсортированном массиве, а 'world' — первым.

Ниже приведены другие примеры.

```
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] │
└─────────┘
```

Примечание

Для улучшения эффективности сортировки применяется преобразование Шварца.

arrayReverseSort([func,] arr, ...)

Возвращает массив arr, отсортированный в нисходящем порядке. Если указана функция func, то массив arr сначала сортируется в порядке, который определяется функцией func, а затем отсортированный массив переворачивается. Если функция func принимает несколько аргументов, то в функцию arrayReverseSort необходимо передавать несколько массивов, которые будут соответствовать аргументам функции func. Подробные примеры рассмотрены в конце описания функции arrayReverseSort.

Пример сортировки целочисленных значений:

```
SELECT arrayReverseSort([1, 3, 3, 0]);
```

```
┌─arrayReverseSort([1, 3, 3, 0])─┐
│ [3,3,1,0]                      │
└────────────────────────────────┘
```

Пример сортировки строковых значений:

```
SELECT arrayReverseSort(['hello', 'world', '!']);
```

```
┌─arrayReverseSort(['hello', 'world', '!'])─┐
│ ['world','hello','!']                     │
└───────────────────────────────────────────┘
```

Значения NULL, NaN и Inf сортируются в следующем порядке:

```
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] │
└─────────────────────────────────────┘
```

- Значения Inf идут в начале массива.
- Значения NULL идут в конце массива.
- Значения NaN идут перед NULL.
- Значения -Inf идут перед NaN.

Функция arrayReverseSort является функцией высшего порядка. Вы можете передать ей в качестве первого аргумента лямбда-функцию. Например:

```
SELECT arrayReverseSort((x) -> -x, [1, 2, 3]) as res;
```

```
┌─res─────┐
│ [1,2,3] │
└─────────┘
```

В этом примере, порядок сортировки устанавливается следующим образом:

1. Сначала исходный массив ([1, 2, 3]) сортируется в том порядке, который определяется лямбда-функцией. Результатом будет массив [3, 2, 1].
2. Массив, который был получен на предыдущем шаге, переворачивается. То есть, получается массив [1, 2, 3].

Лямбда-функция может принимать на вход несколько аргументов. В этом случае, в функцию arrayReverseSort нужно передавать несколько массивов, которые будут соответствовать аргументам лямбда-функции (массивы должны быть одинаковой длины). Следует иметь в виду, что результат будет содержать элементы только из первого массива; элементы из всех последующих массивов будут определять ключи сортировки. Например:

```
SELECT arrayReverseSort((x, y) -> y, ['hello', 'world'], [2, 1]) as res;
```

```
┌─res───────────────┐
│ ['hello','world'] │
└───────────────────┘
```

В этом примере, массив сортируется следующим образом:

1. Сначала массив сортируется в том порядке, который определяется лямбда-функцией. Элементы, указанные во втором массиве ([2,1]), определяют ключи сортировки соответствующих элементов из исходного массива (['hello', 'world']). То есть, будет массив ['world', 'hello'].
2. Массив, который был отсортирован на предыдущем шаге, переворачивается. Получается массив ['hello', 'world'].

Ниже приведены ещё примеры.

```
SELECT arrayReverseSort((x, y) -> y, [0, 1, 2], ['c', 'b', 'a']) as res;
```

```
┌─res─────┐
│ [0,1,2] │
└─────────┘
```

```
SELECT arrayReverseSort((x, y) -> -y, [4, 3, 5], [1, 2, 3]) AS res;
```

```
┌─res─────┐
│ [4,3,5] │
└─────────┘
```

arrayUniq(arr, ...)

Если передан один аргумент, считает количество разных элементов в массиве.
Если передано несколько аргументов, считает количество разных кортежей из элементов на соответствующих позициях в нескольких массивах.

Если необходимо получить список уникальных элементов массива, можно воспользоваться arrayReduce('groupUniqArray', arr).

arrayJoin(arr)

Особенная функция. Смотрите раздел "Функция arrayJoin".

Функции разбиения и слияния строк и массивов

splitByChar(separator, s)

Разбивает строку на подстроки, используя в качестве разделителя separator.
separator должен быть константной строкой из ровно одного символа.
Возвращается массив выделенных подстрок. Могут выделяться пустые подстроки, если разделитель идёт в начале или в конце строки, или если идёт более одного разделителя подряд.

splitByString(separator, s)

То же самое, но использует строку из нескольких символов в качестве разделителя. Строка должна быть непустой.

arrayStringConcat(arr[, separator])

Склеивает строки, перечисленные в массиве, с разделителем separator.
separator - необязательный параметр, константная строка, по умолчанию равен пустой строке.
Возвращается строка.

alphaTokens(s)

Выделяет подстроки из подряд идущих байт из диапазонов a-z и A-Z.
Возвращается массив выделенных подстрок.

Пример:

```
SELECT alphaTokens('abca1abc')

┌─alphaTokens('abca1abc')─┐
│ ['abca','abc']          │
└─────────────────────────┘
```

Битовые функции

Битовые функции работают для любой пары типов из UInt8, UInt16, UInt32, UInt64, Int8, Int16, Int32, Int64, Float32, Float64.

Тип результата - целое число, битность которого равна максимальной битности аргументов. Если хотя бы один аргумент знаковый, то результат - знаковое число. Если аргумент - число с плавающей запятой - оно приводится к Int64.

bitAnd(a, b)

bitOr(a, b)

bitXor(a, b)

bitNot(a)

bitShiftLeft(a, b)

bitShiftRight(a, b)

Функции хэширования

Функции хэширования могут использоваться для детерминированного псевдослучайного разбрасывания элементов.

halfMD5

Вычисляет MD5 от строки. Затем берёт первые 8 байт от хэша и интерпретирует их как UInt64 в big endian.
Принимает аргумент типа String. Возвращает UInt64.
Функция работает достаточно медленно (5 миллионов коротких строк в секунду на одном процессорном ядре).
Если вам не нужен конкретно MD5, то используйте вместо этого функцию sipHash64.

MD5

Вычисляет MD5 от строки и возвращает полученный набор байт в виде FixedString(16).
Если вам не нужен конкретно MD5, а нужен неплохой криптографический 128-битный хэш, то используйте вместо этого функцию sipHash128.
Если вы хотите получить такой же результат, как выдаёт утилита md5sum, напишите lower(hex(MD5(s))).

sipHash64

Вычисляет SipHash от строки.
Принимает аргумент типа String. Возвращает UInt64.
SipHash - криптографическая хэш-функция. Работает быстрее чем MD5 не менее чем в 3 раза.
Подробнее смотрите по ссылке: https://131002.net/siphash/

sipHash128

Вычисляет SipHash от строки.
Принимает аргумент типа String. Возвращает FixedString(16).
Отличается от sipHash64 тем, что финальный xor-folding состояния делается только до 128 бит.

cityHash64

Вычисляет CityHash64 от строки или похожую хэш-функцию для произвольного количества аргументов произвольного типа.
Если аргумент имеет тип String, то используется CityHash. Это быстрая некриптографическая хэш-функция неплохого качества для строк.
Если аргумент имеет другой тип, то используется implementation specific быстрая некриптографическая хэш-функция неплохого качества.
Если передано несколько аргументов, то функция вычисляется по тем же правилам, с помощью комбинации по цепочке с использованием комбинатора из CityHash.
Например, так вы можете вычислить чексумму всей таблицы с точностью до порядка строк: `SELECT sum(cityHash64(*)) FROM table`.

intHash32

Вычисляет 32-битный хэш-код от целого числа любого типа.
Это сравнительно быстрая некриптографическая хэш-функция среднего качества для чисел.

intHash64

Вычисляет 64-битный хэш-код от целого числа любого типа.
Работает быстрее, чем intHash32. Качество среднее.

SHA1

SHA224

SHA256

Вычисляет SHA-1, SHA-224, SHA-256 от строки и возвращает полученный набор байт в виде FixedString(20), FixedString(28), FixedString(32).
Функция работает достаточно медленно (SHA-1 - примерно 5 миллионов коротких строк в секунду на одном процессорном ядре, SHA-224 и SHA-256 - примерно 2.2 миллионов).
Рекомендуется использовать эти функции лишь в тех случаях, когда вам нужна конкретная хэш-функция и вы не можете её выбрать.
Даже в этих случаях, рекомендуется применять функцию оффлайн - заранее вычисляя значения при вставке в таблицу, вместо того, чтобы применять её при SELECT-ах.

URLHash(url[, N])

Быстрая некриптографическая хэш-функция неплохого качества для строки, полученной из URL путём некоторой нормализации.
URLHash(s) - вычислить хэш от строки без одного завершающего символа /, ? или # на конце, если такой там есть.
URLHash(s, N) - вычислить хэш от строки до N-го уровня в иерархии URL, без одного завершающего символа /, ? или # на конце, если такой там есть.
Уровни аналогичные URLHierarchy. Функция специфична для Яндекс.Метрики.

Функции генерации псевдослучайных чисел

Используются некриптографические генераторы псевдослучайных чисел.

Все функции принимают ноль аргументов или один аргумент.
В случае, если передан аргумент - он может быть любого типа, и его значение никак не используется.
Этот аргумент нужен только для того, чтобы предотвратить склейку одинаковых выражений - чтобы две разные записи одной функции возвращали разные столбцы, с разными случайными числами.

rand

Возвращает псевдослучайное число типа UInt32, равномерно распределённое среди всех чисел типа UInt32.
Используется linear congruential generator.

rand64

Возвращает псевдослучайное число типа UInt64, равномерно распределённое среди всех чисел типа UInt64.
Используется linear congruential generator.

Функции для работы с UUID

generateUUIDv4

Генерирует идентификатор UUID версии 4.

```
generateUUIDv4()
```

Возвращаемое значение

Значение типа UUID.

Пример использования

Этот пример демонстрирует, как создать таблицу с UUID-колонкой и добавить в нее сгенерированный UUID.

```
:) 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)

Преобразует значение типа String в тип UUID.

```
toUUID(String)
```

Возвращаемое значение

Значение типа UUID.

Пример использования

```
:) SELECT toUUID('61f0c404-5cb3-11e7-907b-a6006ad3dba0') AS uuid
```

```
┌─────────────────────────────────uuid─┐
│ 61f0c404-5cb3-11e7-907b-a6006ad3dba0 │
└──────────────────────────────────────┘
```

UUIDStringToNum

Принимает строку, содержащую 36 символов в формате xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx, и возвращает в виде набора байт в FixedString(16).

```
UUIDStringToNum(String)
```

Возвращаемое значение

FixedString(16)

Пример использования

```
:) SELECT
    '612f3c40-5d3b-217e-707b-6a546a3d7b29' AS uuid,
    UUIDStringToNum(uuid) AS bytes
```

```
┌─uuid─────────────────────────────────┬─bytes────────┐
│ 612f3c40-5d3b-217e-707b-6a546a3d7b29 │ a/<@];!~p{jTj={} │
└──────────────────────────────────────┴──────────────┘
```

UUIDNumToString

Принимает значение типа FixedString(16). Возвращает строку из 36 символов в текстовом виде.

```
UUIDNumToString(FixedString(16))
```

Возвращаемое значение

Значение типа String.

Пример использования

```
SELECT
    'a/<@];!~p{jTj={}' AS bytes,
    UUIDNumToString(toFixedString(bytes, 16)) AS uuid
```

```
┌─bytes────────┬─uuid─────────────────────────────────┐
│ a/<@];!~p{jTj={} │ 612f3c40-5d3b-217e-707b-6a546a3d7b29 │
└──────────────┴──────────────────────────────────────┘
```

См. также:

- dictGetUUID

- dictGetUUIDOrDefault

Функции кодирования

hex

Принимает аргументы типов: String, unsigned integer, Date, or DateTime. Возвращает строку, содержащую шестнадцатеричное представление аргумента. Используются заглавные буквы A-F. Не используются префиксы 0x и суффиксы h. Для строк просто все байты кодируются в виде двух шестнадцатеричных цифр. Числа выводятся в big endian ("человеческом") формате. Для чисел вырезаются старшие нули, но только по целым байтам. Например, hex(1) = '01'. Date кодируется как число дней с начала unix-эпохи. DateTime кодируются как число секунд с начала unix-эпохи.

unhex(str)

Принимает строку, содержащую произвольное количество шестнадцатеричных цифр, и возвращает строку, содержащую соответствующие байты. Поддерживаются как строчные, так и заглавные буквы A-F. Число шестнадцатеричных цифр не обязано быть чётным. Если оно нечётное - последняя цифра интерпретируется как младшая половинка байта 00-0F. Если строка-аргумент содержит что-либо кроме шестнадцатеричных цифр, то будет возвращён какой-либо implementation-defined результат (не кидается исключение).
Если вы хотите преобразовать результат в число, то вы можете использовать функции reverse и reinterpretAsType.

UUIDStringToNum(str)

Принимает строку, содержащую 36 символов в формате 123e4567-e89b-12d3-a456-426655440000, и возвращает в виде набора байт в FixedString(16).

UUIDNumToString(str)

Принимает значение типа FixedString(16). Возвращает строку из 36 символов в текстовом виде.

bitmaskToList(num)

Принимает целое число. Возвращает строку, содержащую список степеней двойки, в сумме дающих исходное число; по возрастанию, в текстовом виде, через запятую, без пробелов.

bitmaskToArray(num)

Принимает целое число. Возвращает массив чисел типа UInt64, содержащий степени двойки, в сумме дающих исходное число; числа в массиве идут по возрастанию.

Функции для работы с URL

Все функции работают не по RFC - то есть, максимально упрощены ради производительности.

Функции, извлекающие часть URL-а.

Если в URL-е нет ничего похожего, то возвращается пустая строка.

protocol

Возвращает протокол. Примеры: http, ftp, mailto, magnet...

domain

Возвращает домен.

domainWithoutWWW

Возвращает домен, удалив не более одного 'www.' с начала, если есть.

topLevelDomain

Возвращает домен верхнего уровня. Пример: .ru.

firstSignificantSubdomain

Возвращает "первый существенный поддомен". Это понятие является нестандартным и специфично для Яндекс.Метрики. Первый существенный поддомен - это домен второго уровня, если он не равен одному из com, net, org, co, или домен третьего уровня, иначе. Например, firstSignificantSubdomain('https://news.yandex.ru/') = 'yandex', firstSignificantSubdomain('https://news.yandex.com.tr/') = 'yandex'. Список "несущественных" доменов второго уровня и другие детали реализации могут изменяться в будущем.

cutToFirstSignificantSubdomain

Возвращает часть домена, включающую поддомены верхнего уровня до "первого существенного поддомена" (см. выше).

Например, cutToFirstSignificantSubdomain('https://news.yandex.com.tr/') = 'yandex.com.tr'.

path

Возвращает путь. Пример: /top/news.html Путь не включает в себя query string.

pathFull

То же самое, но включая query string и fragment. Пример: /top/news.html?page=2#comments

queryString

Возвращает query-string. Пример: page=1&lr=213. query-string не включает в себя начальный знак вопроса, а также # и всё, что после #.

fragment

Возвращает fragment identifier. fragment не включает в себя начальный символ решётки.

queryStringAndFragment

Возвращает query string и fragment identifier. Пример: страница=1#29390.

extractURLParameter(URL, name)

Возвращает значение параметра name в URL, если такой есть; или пустую строку, иначе; если параметров с таким именем много - вернуть первый попавшийся. Функция работает при допущении, что имя параметра закодировано в URL в точности таким же образом, что и в переданном аргументе.

extractURLParameters(URL)

Возвращает массив строк вида name=value, соответствующих параметрам URL. Значения никак не декодируются.

extractURLParameterNames(URL)

Возвращает массив строк вида name, соответствующих именам параметров URL. Значения никак не декодируются.

URLHierarchy(URL)

Возвращает массив, содержащий URL, обрезанный с конца по символам /, ? в пути и query-string. Подряд идущие символы-разделители считаются за один. Резка производится в позиции после всех подряд идущих символов-разделителей. Пример:

URLPathHierarchy(URL)

То же самое, но без протокола и хоста в результате. Элемент / (корень) не включается. Пример: Функция используется для реализации древовидных отчётов по URL в Яндекс.Метрике.

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

decodeURLComponent(URL)

Возвращает декодированный URL.
Пример:

```
SELECT decodeURLComponent('http://127.0.0.1:8123/?query=SELECT%201%3B') AS DecodedURL;
```

```
┌─DecodedURL─────────────────────────────┐
│ http://127.0.0.1:8123/?query=SELECT 1; │
└────────────────────────────────────────┘
```

Функции, удаляющие часть из URL-а

Если в URL-е нет ничего похожего, то URL остаётся без изменений.

cutWWW

Удаляет не более одного 'www.' с начала домена URL-а, если есть.

cutQueryString

Удаляет query string. Знак вопроса тоже удаляется.

cutFragment

Удаляет fragment identifier. Символ решётки тоже удаляется.

cutQueryStringAndFragment

Удаляет query string и fragment identifier. Знак вопроса и символ решётки тоже удаляются.

cutURLParameter(URL, name)

Удаляет параметр URL с именем name, если такой есть. Функция работает при допущении, что имя параметра закодировано в URL в точности таким же образом, что и в переданном аргументе.

Функции для работы с IP-адресами

IPv4NumToString(num)

Принимает число типа UInt32. Интерпретирует его, как IPv4-адрес в big endian. Возвращает строку, содержащую соответствующий IPv4-адрес в формате A.B.C.D (числа в десятичной форме через точки).

IPv4StringToNum(s)

Функция, обратная к IPv4NumToString. Если IPv4 адрес в неправильном формате, то возвращает 0.

IPv4NumToStringClassC(num)

Похоже на IPv4NumToString, но вместо последнего октета используется xxx.

Пример:

```
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 │
└────────────────┴───────┘
```

В связи с тем, что использование xxx весьма необычно, это может быть изменено в дальнейшем, и вам не следует полагаться на конкретный вид этого фрагмента.

IPv6NumToString(x)

Принимает значение типа FixedString(16), содержащее IPv6-адрес в бинарном виде. Возвращает строку, содержащую этот адрес в текстовом виде.
IPv6-mapped IPv4 адреса выводится в формате ::ffff:111.222.33.44. Примеры:

```
SELECT IPv6NumToString(toFixedString(unhex('2A0206B8000000000000000000000011'), 16)) AS addr
```

```
┌─addr─────────┐
│ 2a02:6b8::11 │
└──────────────┘
```

```
SELECT
    IPv6NumToString(ClientIP6 AS k),
    count() AS c
FROM hits_all
WHERE EventDate = today() AND substring(ClientIP6, 1, 12) != unhex('000000000000000000FFFF')
GROUP BY k
ORDER BY c DESC
LIMIT 10
```

```
┌─IPv6NumToString(ClientIP6)──────────────┬─────c─┐
│ 2a02:2168:aaa:bbbb::2                   │ 24695 │
│ 2a02:2698:abcd:abcd:abcd:abcd:8888:5555 │ 22408 │
│ 2a02:6b8:0:fff::ff                      │ 16389 │
│ 2a01:4f8:111:6666::2                    │ 16016 │
│ 2a02:2168:888:222::1                    │ 15896 │
│ 2a01:7e00::ffff:ffff:ffff:222           │ 14774 │
│ 2a02:8109:eee:ee:eeee:eeee:eeee:eeee    │ 14443 │
│ 2a02:810b:8888:888:8888:8888:8888:8888  │ 14345 │
│ 2a02:6b8:0:444:4444:4444:4444:4444      │ 14279 │
│ 2a01:7e00::ffff:ffff:ffff:ffff          │ 13880 │
└─────────────────────────────────────────┴───────┘
```

```
SELECT
    IPv6NumToString(ClientIP6 AS k),
    count() AS c
FROM hits_all
WHERE EventDate = today()
GROUP BY k
ORDER BY c DESC
LIMIT 10
```

```
┌─IPv6NumToString(ClientIP6)─┬──────c─┐
│ ::ffff:94.26.111.111       │ 747440 │
│ ::ffff:37.143.222.4        │ 529483 │
│ ::ffff:5.166.111.99        │ 317707 │
│ ::ffff:46.38.11.77         │ 263086 │
│ ::ffff:79.105.111.111      │ 186611 │
│ ::ffff:93.92.111.88        │ 176773 │
│ ::ffff:84.53.111.33        │ 158709 │
│ ::ffff:217.118.11.22       │ 154004 │
│ ::ffff:217.118.11.33       │ 148449 │
│ ::ffff:217.118.11.44       │ 148243 │
└────────────────────────────┴────────┘
```

IPv6StringToNum(s)

Функция, обратная к IPv6NumToString. Если IPv6 адрес в неправильном формате, то возвращает строку из нулевых байт.
HEX может быть в любом регистре.

Функции для работы с JSON.

В Яндекс.Метрике пользователями передаётся JSON в качестве параметров визитов. Для работы с таким JSON-ом, реализованы некоторые функции. (Хотя в большинстве случаев, JSON-ы дополнительно обрабатываются заранее, и полученные значения кладутся в отдельные столбцы в уже обработанном виде.) Все эти функции исходят из сильных допущений о том, каким может быть JSON, и при этом стараются почти ничего не делать.

Делаются следующие допущения:

1. Имя поля (аргумент функции) должно быть константой;
2. Считается, что имя поля в JSON-е закодировано некоторым каноническим образом. Например, visitParamHas('{"abc":"def"}', 'abc') = 1, но visitParamHas('{"\\u0061\\u0062\\u0063":"def"}', 'abc') = 0
3. Поля ищутся на любом уровне вложенности, без разбора. Если есть несколько подходящих полей - берётся первое.
4. В JSON-е нет пробельных символов вне строковых литералов.

visitParamHas(params, name)

Проверить наличие поля с именем name.

visitParamExtractUInt(params, name)

Распарсить UInt64 из значения поля с именем name. Если поле строковое - попытаться распарсить число из начала строки. Если такого поля нет, или если оно есть, но содержит не число, то вернуть 0.

visitParamExtractInt(params, name)

Аналогично для Int64.

visitParamExtractFloat(params, name)

Аналогично для Float64.

visitParamExtractBool(params, name)

Распарсить значение true/false. Результат - UInt8.

visitParamExtractRaw(params, name)

Вернуть значение поля, включая разделители.

Примеры:

```
visitParamExtractRaw('{"abc":"\\n\\u0000"}', 'abc') = '"\\n\\u0000"'
visitParamExtractRaw('{"abc":{"def":[1,2,3]}}', 'abc') = '{"def":[1,2,3]}'
```

visitParamExtractString(params, name)

Распарсить строку в двойных кавычках. У значения убирается экранирование. Если убрать экранированные символы не удалось, то возвращается пустая строка.

Примеры:

```
visitParamExtractString('{"abc":"\\n\\u0000"}', 'abc') = '\n\0'
visitParamExtractString('{"abc":"\\u263a"}', 'abc') = '☺'
visitParamExtractString('{"abc":"\\u263"}', 'abc') = ''
visitParamExtractString('{"abc":"hello}', 'abc') = ''
```

На данный момент, не поддерживаются записанные в формате \uXXXX\uYYYY кодовые точки не из basic multilingual plane (они переводятся не в UTF-8, а в CESU-8).

Функции высшего порядка

Оператор ->, функция lambda(params, expr)

Позволяет описать лямбда-функцию для передачи в функцию высшего порядка. Слева от стрелочки стоит формальный параметр - произвольный идентификатор, или несколько формальных параметров - произвольные идентификаторы в кортеже. Справа от стрелочки стоит выражение, в котором могут использоваться эти формальные параметры, а также любые столбцы таблицы.

Примеры: x -> 2 * x, str -> str != Referer.

Функции высшего порядка, в качестве своего функционального аргумента могут принимать только лямбда-функции.

В функции высшего порядка может быть передана лямбда-функция, принимающая несколько аргументов. В этом случае, в функцию высшего порядка передаётся несколько массивов одинаковых длин, которым эти аргументы будут соответствовать.

Для всех функций кроме arrayMap, arrayFilter, первый аргумент (лямбда-функция) может отсутствовать. В этом случае, подразумевается тождественное отображение.

arrayMap(func, arr1, ...)

Вернуть массив, полученный из исходного применением функции func к каждому элементу массива arr.

arrayFilter(func, arr1, ...)

Вернуть массив, содержащий только те элементы массива arr1, для которых функция func возвращает не 0.

Примеры:

```
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, ...)

Вернуть количество элементов массива arr, для которых функция func возвращает не 0. Если func не указана - вернуть количество ненулевых элементов массива.

arrayExists([func,] arr1, ...)

Вернуть 1, если существует хотя бы один элемент массива arr, для которого функция func возвращает не 0. Иначе вернуть 0.

arrayAll([func,] arr1, ...)

Вернуть 1, если для всех элементов массива arr, функция func возвращает не 0. Иначе вернуть 0.

arraySum([func,] arr1, ...)

Вернуть сумму значений функции func. Если функция не указана - просто вернуть сумму элементов массива.

arrayFirst(func, arr1, ...)

Вернуть первый элемент массива arr1, для которого функция func возвращает не 0.

arrayFirstIndex(func, arr1, ...)

Вернуть индекс первого элемента массива arr1, для которого функция func возвращает не 0.

arrayCumSum([func,] arr1, ...)

Возвращает массив из частичных сумм элементов исходного массива (сумма с накоплением). Если указана функция func, то значения элементов массива преобразуются этой функцией перед суммированием.

Пример:

```
SELECT arrayCumSum([1, 1, 1, 1]) AS res
```

```
┌─res──────────┐
│ [1, 2, 3, 4] │
└──────────────┘
```

arraySort([func,] arr1, ...)

Возвращает отсортированный в восходящем порядке массив arr1. Если задана функция func, то порядок сортировки определяется результатом применения функции func на элементы массива (массивов).

Для улучшения эффективности сортировки применяется Преобразование Шварца.

Пример:

```
SELECT arraySort((x, y) -> y, ['hello', 'world'], [2, 1]);
```

```
┌─res────────────────┐
│ ['world', 'hello'] │
└────────────────────┘
```

NULL и NaN будут последними в массиве (при этом NaN будет перед NULL). Например:

```
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, ...)

Возвращает отсортированный в нисходящем порядке массив arr1. Если задана функция func, то порядок сортировки определяется результатом применения функции func на элементы массива (массивов).

NULL и NaN будут последними в массиве (при этом NaN будет перед NULL). Например:

```
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]                          │
└──────────────────────────────────────────────────────┘
```

Функции для работы с внешними словарями

Информация о подключении и настройке внешних словарей смотрите в разделе Внешние словари.

dictGetUInt8, dictGetUInt16, dictGetUInt32, dictGetUInt64
dictGetInt8, dictGetInt16, dictGetInt32, dictGetInt64
dictGetFloat32, dictGetFloat64
dictGetDate, dictGetDateTime
dictGetUUID
dictGetString

dictGetT('dict_name', 'attr_name', id)
- получить из словаря dict_name значение атрибута attr_name по ключу id.
dict_name и attr_name - константные строки.
id должен иметь тип UInt64.
Если ключа id нет в словаре - вернуть значение по умолчанию, заданное в описании словаря.

dictGetTOrDefault

dictGetT('dict_name', 'attr_name', id, default)

Аналогично функциям dictGetT, но значение по умолчанию берётся из последнего аргумента функции.

dictIsIn

dictIsIn('dict_name', child_id, ancestor_id)
- для иерархического словаря dict_name - узнать, находится ли ключ child_id внутри ancestor_id (или совпадает с ancestor_id). Возвращает UInt8.

dictGetHierarchy

dictGetHierarchy('dict_name', id)
- для иерархического словаря dict_name - вернуть массив ключей словаря, начиная с id и продолжая цепочкой родительских элементов. Возвращает Array(UInt64).

dictHas

dictHas('dict_name', id)
- проверить наличие ключа в словаре. Возвращает значение типа UInt8, равное 0, если ключа нет и 1, если ключ есть.

Функции для работы со словарями Яндекс.Метрики

Чтобы указанные ниже функции работали, в конфиге сервера должны быть указаны пути и адреса для получения всех словарей Яндекс.Метрики. Словари загружаются при первом вызове любой из этих функций. Если справочники не удаётся загрузить - будет выкинуто исключение.

О том, как создать справочники, смотрите в разделе "Словари".

Множественные геобазы

ClickHouse поддерживает работу одновременно с несколькими альтернативными геобазами (иерархиями регионов), для того чтобы можно было поддержать разные точки зрения о принадлежности регионов странам.

В конфиге clickhouse-server указывается файл с иерархией регионов:
<path_to_regions_hierarchy_file>/opt/geo/regions_hierarchy.txt</path_to_regions_hierarchy_file>

Кроме указанного файла, рядом ищутся файлы, к имени которых (до расширения) добавлен символ _ и какой угодно суффикс.
Например, также найдётся файл /opt/geo/regions_hierarchy_ua.txt, если такой есть.

ua называется ключом словаря. Для словаря без суффикса, ключ является пустой строкой.

Все словари перезагружаются в рантайме (раз в количество секунд, заданное в конфигурационном параметре builtin_dictionaries_reload_interval, по умолчанию - раз в час), но перечень доступных словарей определяется один раз, при старте сервера.

Во все функции по работе с регионами, в конце добавлен один необязательный аргумент - ключ словаря. Далее он обозначен как geobase.
Пример:

```
regionToCountry(RegionID) - использует словарь по умолчанию: /opt/geo/regions_hierarchy.txt;
regionToCountry(RegionID, '') - использует словарь по умолчанию: /opt/geo/regions_hierarchy.txt;
regionToCountry(RegionID, 'ua') - использует словарь для ключа ua: /opt/geo/regions_hierarchy_ua.txt;
```

regionToCity(id[, geobase])

Принимает число типа UInt32 - идентификатор региона из геобазы Яндекса. Если регион является городом или входит в некоторый город, то возвращает идентификатор региона - соответствующего города. Иначе возвращает 0.

regionToArea(id[, geobase])

Переводит регион в область (тип в геобазе - 5). В остальном, аналогично функции regionToCity.

```
SELECT DISTINCT regionToName(regionToArea(toUInt32(number), 'ua'))
FROM system.numbers
LIMIT 15
```

```
┌─regionToName(regionToArea(toUInt32(number), \'ua\'))─┐
│                                      │
│ Москва и Московская область                    │
│ Санкт-Петербург и Ленинградская область            │
│ Белгородская область                           │
│ Ивановская область                             │
│ Калужская область                              │
│ Костромская область                            │
│ Курская область                                │
│ Липецкая область                               │
│ Орловская область                              │
│ Рязанская область                              │
│ Смоленская область                             │
│ Тамбовская область                             │
│ Тверская область                               │
│ Тульская область                               │
└──────────────────────────────────────────────────┘
```

regionToDistrict(id[, geobase])

Переводит регион в федеральный округ (тип в геобазе - 4). В остальном, аналогично функции regionToCity.

```
SELECT DISTINCT regionToName(regionToDistrict(toUInt32(number), 'ua'))
FROM system.numbers
LIMIT 15
```

```
┌─regionToName(regionToDistrict(toUInt32(number), \'ua\'))─┐
│                                                          │
│ Центральный федеральный округ                            │
│ Северо-Западный федеральный округ                        │
│ Южный федеральный округ                                  │
│ Северо-Кавказский федеральный округ                      │
│ Приволжский федеральный округ                            │
│ Уральский федеральный округ                              │
│ Сибирский федеральный округ                              │
│ Дальневосточный федеральный округ                        │
│ Шотландия                                                │
│ Фарерские острова                                        │
│ Фламандский регион                                       │
│ Брюссельский столичный регион                            │
│ Валлония                                                 │
│ Федерация Боснии и Герцеговины                           │
└──────────────────────────────────────────────────────────┘
```

regionToCountry(id[, geobase])

Переводит регион в страну. В остальном, аналогично функции regionToCity.
Пример: `regionToCountry(toUInt32(213)) = 225` - преобразовали Москву (213) в Россию (225).

regionToContinent(id[, geobase])

Переводит регион в континент. В остальном, аналогично функции regionToCity.
Пример: `regionToContinent(toUInt32(213)) = 10001` - преобразовали Москву (213) в Евразию (10001).

regionToPopulation(id[, geobase])

Получает население для региона.
Население может быть прописано в файлах с геобазой. Смотрите в разделе "Встроенные словари".
Если для региона не прописано население, возвращается 0.
В геобазе Яндекса, население может быть прописано для дочерних регионов, но не прописано для родительских.

regionIn(lhs, rhs[, geobase])

Проверяет принадлежность региона lhs региону rhs. Возвращает число типа UInt8, равное 1, если принадлежит и 0, если не принадлежит.
Отношение рефлексивное - любой регион принадлежит также самому себе.

regionHierarchy(id[, geobase])

Принимает число типа UInt32 - идентификатор региона из геобазы Яндекса. Возвращает массив идентификаторов регионов, состоящий из переданного региона и всех родителей по цепочке.
Пример: `regionHierarchy(toUInt32(213)) = [213,1,3,225,10001,10000]`.

regionToName(id[, lang])

Принимает число типа UInt32 - идентификатор региона из геобазы Яндекса. Вторым аргументом может быть передана строка - название языка. Поддерживаются языки ru, en, ua, uk, by, kz, tr. Если второй аргумент отсутствует - используется язык ru. Если язык не поддерживается - кидается исключение. Возвращает строку - название региона на соответствующем языке. Если региона с указанным идентификатором не существует - возвращается пустая строка.

ua и uk обозначают одно и то же - украинский язык.

Функции для реализации оператора IN.

in, notIn, globalIn, globalNotIn

Смотрите раздел Операторы IN.

tuple(x, y, ...), оператор (x, y, ...)

Функция, позволяющая сгруппировать несколько столбцов.
Для столбцов, имеющих типы T1, T2, ... возвращает кортеж типа Tuple(T1, T2, ...), содержащий эти столбцы. Выполнение функции ничего не стоит.
Кортежи обычно используются как промежуточное значение в качестве аргумента операторов IN, или для создания списка формальных параметров лямбда-функций. Кортежи не могут быть записаны в таблицу.

tupleElement(tuple, n), оператор x.N

Функция, позволяющая достать столбец из кортежа.
N - индекс столбца начиная с 1. N должно быть константой. N должно быть целым строго положительным числом не большим размера кортежа.
Выполнение функции ничего не стоит.

Функция arrayJoin

Это совсем необычная функция.

Обычные функции не изменяют множество строк, а лишь изменяют значения в каждой строке (map).
Агрегатные функции выполняют свёртку множества строк (fold, reduce).
Функция arrayJoin выполняет размножение каждой строки в множество строк (unfold).

Функция принимает в качестве аргумента массив, и размножает исходную строку в несколько строк - по числу элементов массива.
Все значения в столбцах просто копируются, кроме значения в столбце с применением этой функции - он заменяется на соответствующее значение массива.

В запросе может быть использовано несколько функций arrayJoin. В этом случае, соответствующее преобразование делается несколько раз.

Обратите внимание на синтаксис ARRAY JOIN в запросе SELECT, который предоставляет более широкие возможности.

Пример:

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

Функции для работы с географическими координатами

greatCircleDistance

Вычисляет расстояние между двумя точками на поверхности Земли по формуле большого круга.

```
greatCircleDistance(lon1Deg, lat1Deg, lon2Deg, lat2Deg)
```

Входные параметры

- lon1Deg — долгота первой точки в градусах. Диапазон — [-180°, 180°].

- lat1Deg — широта первой точки в градусах. Диапазон — [-90°, 90°].
- lon2Deg — долгота второй точки в градусах. Диапазон — [-180°, 180°].
- lat2Deg — широта второй точки в градусах. Диапазон — [-90°, 90°].

Положительные значения соответствуют северной широте и восточной долготе, отрицательные — южной широте и западной долготе.

Возвращаемое значение

Расстояние между двумя точками на поверхности Земли в метрах.

Генерирует исключение, когда значения входных параметров выходят за границы диапазонов.

Пример

```
SELECT greatCircleDistance(55.755831, 37.617673, -55.755831, -37.617673)
```

```
┌─greatCircleDistance(55.755831, 37.617673, -55.755831, -37.617673)─┐
│                                                 14132374.194975413 │
└────────────────────────────────────────────────────────────────────┘
```

pointInEllipses

Проверяет, принадлежит ли точка хотя бы одному из эллипсов.

```
pointInEllipses(x, y, x₀, y₀, a₀, b₀,...,xₙ, yₙ, aₙ, bₙ)
```

Входные параметры

- x, y — координаты точки на плоскости.
- x_i, y_i — координаты центра i-го эллипса.
- a_i, b_i — полуоси i-го эллипса в метрах.

Входных параметров должно быть $2+4 \cdot n$, где n — количество эллипсов.

Возвращаемые значения

1, если точка внутри хотя бы одного из эллипсов, 0, если нет.

Пример

```
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

Проверяет, принадлежит ли точка многоугольнику на плоскости.

```
pointInPolygon((x, y), [(a, b), (c, d) ...], ...)
```

Входные значения

- (x, y) — координаты точки на плоскости. Тип данных — [Tuple](../../data_types/tuple.md- [(a, b), (c, d) ...] — вершины многоугольника. Тип данных — Array. Каждая вершина представлена парой координат (a, b). Вершины следует указывать в порядке обхода по или против часовой стрелки. Минимальное количество вершин — 3. Многоугольник должен быть константным.
- [(a, b), (c, d) ...] — вершины многоугольника. Тип данных — Array. Каждая вершина представлена парой координат (a, b). Вершины следует указывать в порядке обхода по или против часовой стрелки. Минимальное количество вершин — 3. Многоугольник должен быть константным.

- функция поддерживает также многоугольники с дырками (вырезанными кусками). Для этого случая, добавьте многоугольники, описывающие вырезанные куски, дополнительными аргументами функции. Функция не поддерживает неодносвязные многоугольники.

Возвращаемые значения

1, если точка внутри многоугольника, 0, если нет.
Если точка находится на границе многоугольника, функция может возвращать как 0, так и 1.

Пример

```
SELECT pointInPolygon((3., 3.), [(6, 0), (8, 4), (5, 8), (0, 2)]) AS res
```

```
┌─res─┐
│   1 │
└─────┘
```

Функции для работы с Nullable-агрументами

isNull

Проверяет является ли аргумент NULL.

```
isNull(x)
```

Параметры

- x — значение с не составным типом данных.

Возвращаемое значение

- 1, если x — NULL.
- 0, если x — не NULL.

Пример

Входная таблица

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

Запрос

```
:) 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

Проверяет не является ли аргумент NULL.

```
isNotNull(x)
```

Параметры

- x — значение с не составным типом данных.

Возвращаемое значение

- 0, если x — NULL.
- 1, если x — не NULL.

Пример

Входная таблица

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

Запрос

```
:) 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

Последовательно слева-направо проверяет являются ли переданные аргументы NULL и возвращает первый не NULL.

```
coalesce(x,...)
```

Параметры

- Произвольное количество параметров не составного типа. Все параметры должны быть совместимы по типу данных.

Возвращаемые значения

- Первый не NULL аргумент.
- NULL, если все аргументы — NULL.

Пример

Рассмотрим адресную книгу, в которой может быть указано несколько способов связи с клиентом.

```
┌─name─────┬─mail─┬─phone─────┬─icq─┐
│ client 1 │ NULL │ 123-45-67 │ 123 │
│ client 2 │ NULL │ NULL      │ NULL │
└──────────┴──────┴───────────┴─────┘
```

Поля mail и phone имеют тип String, а поле icq — UInt32, его необходимо будет преобразовать в String.

Получим из адресной книги первый доступный способ связаться с клиентом:

```
:) 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

Возвращает альтернативное значение, если основной аргумент — NULL.

```
ifNull(x,alt)
```

Параметры

- x — значение для проверки на NULL,
- alt — значение, которое функция вернёт, если x — NULL.

Возвращаемые значения

- Значение x, если x — не NULL.
- Значение alt, если x — NULL.

Пример

```
SELECT ifNull('a', 'b')

┌─ifNull('a', 'b')─┐
│ a                │
└──────────────────┘
```

```
SELECT ifNull(NULL, 'b')

┌─ifNull(NULL, 'b')─┐
│ b                 │
└───────────────────┘
```

nullIf

Возвращает NULL, если аргументы равны.

```
nullIf(x, y)
```

Параметры

x, y — значения для сравнивания. Они должны быть совместимых типов, иначе ClickHouse сгенерирует исключение.

Возвращаемые значения

- NULL, если аргументы равны.
- Значение x, если аргументы не равны.

Пример

```
SELECT nullIf(1, 1)

┌─nullIf(1, 1)─┐
│         NULL │
└──────────────┘
```

```
SELECT nullIf(1, 2)

┌─nullIf(1, 2)─┐
│            1 │
└──────────────┘
```

assumeNotNull

Приводит значение типа Nullable к не Nullable, если значение не NULL.

```
assumeNotNull(x)
```

Параметры

- x — исходное значение.

Возвращаемые значения

- Исходное значение с не Nullable типом, если оно — не NULL.
- Значение по умолчанию для не Nullable типа, если исходное значение — NULL.

Пример

Рассмотрим таблицу t_null.

```
SHOW CREATE TABLE t_null

┌─statement─────────────────────────────────────────────────────────────────┐
│ CREATE TABLE default.t_null ( x Int8,  y Nullable(Int8)) ENGINE = TinyLog │
└───────────────────────────────────────────────────────────────────────────┘
```

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

Применим функцию assumeNotNull к столбцу y.

```
SELECT assumeNotNull(y) FROM t_null

┌─assumeNotNull(y)─┐
│                0 │
│                3 │
└──────────────────┘
```

```
SELECT toTypeName(assumeNotNull(y)) FROM t_null

┌─toTypeName(assumeNotNull(y))─┐
│ Int8                         │
│ Int8                         │
└──────────────────────────────┘
```

toNullable

Преобразует тип аргумента к Nullable.

```
toNullable(x)
```

Параметры

- x — значение произвольного не составного типа.

Возвращаемое значение

- Входное значение с типом не Nullable.

Пример

```
SELECT toTypeName(10)

┌─toTypeName(10)─┐
│ UInt8          │
└────────────────┘

SELECT toTypeName(toNullable(10))

┌─toTypeName(toNullable(10))─┐
│ Nullable(UInt8)            │
└────────────────────────────┘
```

Прочие функции

hostName()

Возвращает строку - имя хоста, на котором эта функция была выполнена. При распределённой обработке запроса, это будет имя хоста удалённого сервера, если функция выполняется на удалённом сервере.

visibleWidth(x)

Вычисляет приблизительную ширину при выводе значения в текстовом (tab-separated) виде на консоль. Функция используется системой для реализации Pretty форматов.

NULL представляется как строка, соответствующая отображению NULL в форматах Pretty.

```
SELECT visibleWidth(NULL)

┌─visibleWidth(NULL)─┐
│                  4 │
└────────────────────┘
```

toTypeName(x)

Возвращает строку, содержащую имя типа переданного аргумента.

Если на вход функции передать NULL, то она вернёт тип Nullable(Nothing), что соответствует внутреннему представлению NULL в ClickHouse.

blockSize()

Получить размер блока.
В ClickHouse выполнение запроса всегда идёт по блокам (наборам кусочков столбцов). Функция позволяет получить размер блока, для которого её вызвали.

materialize(x)

Превращает константу в полноценный столбец, содержащий только одно значение.
В ClickHouse полноценные столбцы и константы представлены в памяти по-разному. Функции по-разному работают для аргументов-констант и обычных аргументов (выполняется разный код), хотя результат почти всегда должен быть одинаковым. Эта функция предназначена для отладки такого поведения.

ignore(...)

Принимает любые аргументы, в т.ч. NULL, всегда возвращает 0.
При этом, аргумент всё равно вычисляется. Это может использоваться для бенчмарков.

sleep(seconds)

Спит seconds секунд на каждый блок данных. Можно указать как целое число, так и число с плавающей запятой.

currentDatabase()

Возвращает имя текущей базы данных.
Эта функция может использоваться в параметрах движка таблицы в запросе CREATE TABLE там, где нужно указать базу данных.

isFinite(x)

Принимает Float32 или Float64 и возвращает UInt8, равный 1, если аргумент не бесконечный и не NaN, иначе 0.

isInfinite(x)

Принимает Float32 или Float64 и возвращает UInt8, равный 1, если аргумент бесконечный, иначе 0. Отметим, что в случае NaN возвращается 0.

isNaN(x)

Принимает Float32 или Float64 и возвращает UInt8, равный 1, если аргумент является NaN, иначе 0.

hasColumnInTable(['hostname'[, 'username'[, 'password']],] 'database', 'table', 'column')

Принимает константные строки - имя базы данных, имя таблицы и название столбца. Возвращает константное выражение типа UInt8, равное 1,
если есть столбец, иначе 0. Если задан параметр hostname, проверка будет выполнена на удалённом сервере.
Функция кидает исключение, если таблица не существует.
Для элементов вложенной структуры данных функция проверяет существование столбца. Для самой же вложенной структуры данных функция возвращает 0.

bar

Позволяет построить unicode-art диаграмму.

`bar(x, min, max, width)` рисует полосу ширины пропорциональной `(x - min)` и равной `width` символов при `x = max`.

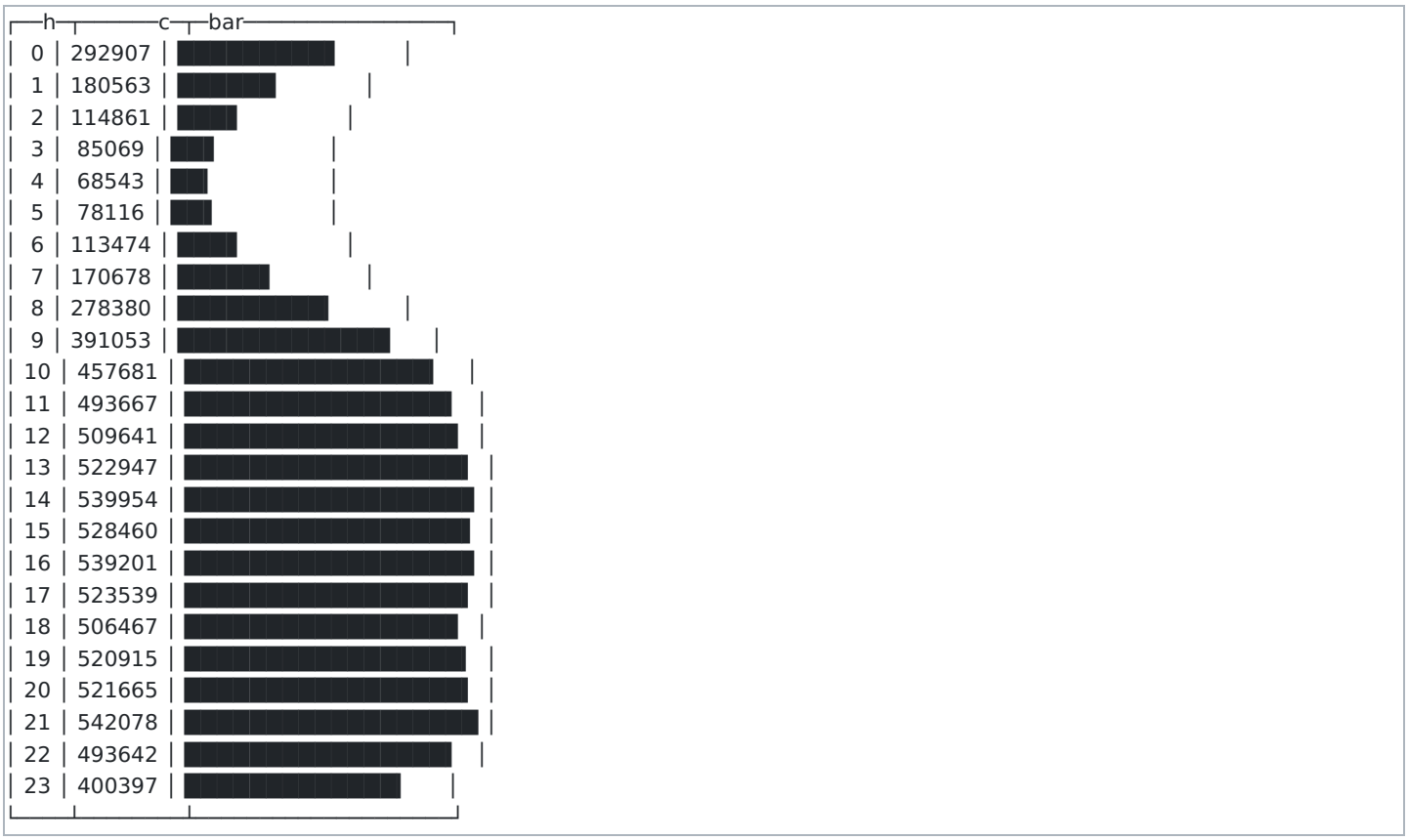
Параметры:

- `x` — Величина для отображения.
- `min, max` — Целочисленные константы, значение должно помещаться в `Int64`.
- `width` — Константа, положительное число, может быть дробным.

Полоса рисуется с точностью до одной восьмой символа.

Пример:

```
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

Преобразовать значение согласно явно указанному отображению одних элементов на другие.
Имеется два варианта функции:

1. transform(x, array_from, array_to, default)

x - что преобразовывать.

array_from - константный массив значений для преобразования.

array_to - константный массив значений, в которые должны быть преобразованы значения из from.

default - какое значение использовать, если x не равен ни одному из значений во from.

array_from и array_to - массивы одинаковых размеров.

Типы:

transform(T, Array(T), Array(U), U) -> U

T и U - могут быть числовыми, строковыми, или Date или DateTime типами.
При этом, где обозначена одна и та же буква (T или U), могут быть, в случае числовых типов, не совпадающие типы, а типы, для которых есть общий тип.
Например, первый аргумент может иметь тип Int64, а второй - Array(UInt16).

Если значение x равно одному из элементов массива array_from, то возвращает соответствующий (такой же по номеру) элемент массива array_to; иначе возвращает default. Если имеется несколько совпадающих элементов в array_from, то возвращает какой-нибудь из соответствующих.

Пример:

```
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)

Отличается от первого варианта отсутствующим аргументом default.
Если значение x равно одному из элементов массива array_from, то возвращает соответствующий (такой же по номеру) элемент массива array_to; иначе возвращает x.

Типы:

transform(T, Array(T), Array(T)) -> T

Пример:

```
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)

Принимает размер (число байт). Возвращает округленный размер с суффиксом (KiB, MiB и т.д.) в виде строки.

Пример:

```
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)

Возвращает наименьшее значение из a и b.

greatest(a, b)

Возвращает наибольшее значение из a и b.

uptime()

Возвращает аптайм сервера в секундах.

version()

Возвращает версию сервера в виде строки.

rowNumberInAllBlocks()

Возвращает порядковый номер строки в блоке данных. Функция учитывает только задействованные блоки данных.

runningDifference(x)

Считает разницу между последовательными значениями строк в блоке данных.
Возвращает 0 для первой строки и разницу с предыдущей строкой для каждой последующей строки.

Результат функции зависит от затронутых блоков данных и порядка данных в блоке.
Если сделать подзапрос с ORDER BY и вызывать функцию извне подзапроса, можно будет получить ожидаемый результат.

Пример:

```
SELECT
    EventID,
    EventTime,
    runningDifference(EventTime) AS delta
FROM
(
    SELECT
        EventID,
        EventTime
    FROM events
    WHERE EventDate = '2016-11-24'
    ORDER BY EventTime ASC
    LIMIT 5
)
```

```
┌─EventID─┬─────────────EventTime─┬─delta─┐
│    1106 │ 2016-11-24 00:00:04 │     0 │
│    1107 │ 2016-11-24 00:00:05 │     1 │
│    1108 │ 2016-11-24 00:00:05 │     0 │
│    1109 │ 2016-11-24 00:00:09 │     4 │
│    1110 │ 2016-11-24 00:00:10 │     1 │
└─────────┴─────────────────────┴───────┘
```

MACNumToString(num)

Принимает число типа UInt64. Интерпретирует его, как MAC-адрес в big endian. Возвращает строку, содержащую соответствующий MAC-адрес в формате AA:BB:CC:DD:EE:FF (числа в шестнадцатеричной форме через двоеточие).

MACStringToNum(s)

Функция, обратная к MACNumToString. Если MAC адрес в неправильном формате, то возвращает 0.

MACStringToOUI(s)

Принимает MAC адрес в формате AA:BB:CC:DD:EE:FF (числа в шестнадцатеричной форме через двоеточие). Возвращает первые три октета как число в формате UInt64. Если MAC адрес в неправильном формате, то возвращает 0.

getSizeOfEnumType

Возвращает количество полей в Enum.

```
getSizeOfEnumType(value)
```

Параметры

- value — Значение типа Enum.

Возвращаемые значения

- Количество полей входного значения типа Enum.
- Исключение, если тип не Enum.

Пример

```
SELECT getSizeOfEnumType( CAST('a' AS Enum8('a' = 1, 'b' = 2) ) ) AS x

┌─x─┐
│ 2 │
└───┘
```

toColumnTypeName

Возвращает имя класса, которым представлен тип данных столбца в оперативной памяти.

```
toColumnTypeName(value)
```

Параметры

- value — Значение произвольного типа.

Возвращаемые значения

- Строка с именем класса, который используется для представления типа данных value в оперативной памяти.

Пример разницы между toTypeName и toColumnTypeName

```
:) select toTypeName(cast('2018-01-01 01:02:03' AS DateTime))

SELECT toTypeName(CAST('2018-01-01 01:02:03', 'DateTime'))

┌─toTypeName(CAST('2018-01-01 01:02:03', 'DateTime'))─┐
│ DateTime                                            │
└─────────────────────────────────────────────────────┘

1 rows in set. Elapsed: 0.008 sec.

:) select toColumnTypeName(cast('2018-01-01 01:02:03' AS DateTime))

SELECT toColumnTypeName(CAST('2018-01-01 01:02:03', 'DateTime'))

┌─toColumnTypeName(CAST('2018-01-01 01:02:03', 'DateTime'))─┐
│ Const(UInt32)                                             │
└───────────────────────────────────────────────────────────┘
```

В примере видно, что тип данных DateTime хранится в памяти как Const(UInt32).

dumpColumnStructure

Выводит развернутое описание структур данных в оперативной памяти

```
dumpColumnStructure(value)
```

Параметры

- value — Значение произвольного типа.

Возвращаемые значения

- Строка с описанием структуры, которая используется для представления типа данных value в оперативной памяти.

Пример

```
SELECT dumpColumnStructure(CAST('2018-01-01 01:02:03', 'DateTime'))

┌─dumpColumnStructure(CAST('2018-01-01 01:02:03', 'DateTime'))─┐
│ DateTime, Const(size = 1, UInt32(size = 1))                  │
└──────────────────────────────────────────────────────────────┘
```

defaultValueOfArgumentType

Выводит значение по умолчанию для типа данных.

Не учитывает значения по умолчанию для столбцов, заданные пользователем.

```
defaultValueOfArgumentType(expression)
```

Параметры

- expression — Значение произвольного типа или выражение, результатом которого является значение произвольного типа.

Возвращаемые значения

- 0 для чисел;
- Пустая строка для строк;
- NULL для Nullable.

Пример

```
:) SELECT defaultValueOfArgumentType( CAST(1 AS Int8) )

SELECT defaultValueOfArgumentType(CAST(1, 'Int8'))

┌─defaultValueOfArgumentType(CAST(1, 'Int8'))─┐
│                                           0 │
└─────────────────────────────────────────────┘

1 rows in set. Elapsed: 0.002 sec.

:) SELECT defaultValueOfArgumentType( CAST(1 AS Nullable(Int8) ) )

SELECT defaultValueOfArgumentType(CAST(1, 'Nullable(Int8)'))

┌─defaultValueOfArgumentType(CAST(1, 'Nullable(Int8)'))─┐
│                                                  NULL │
└───────────────────────────────────────────────────────┘

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

indexHint

Выводит данные, попавшие в диапазон, выбранный по индексу без фильтрации по указанному в качестве аргумента выражению.

Переданное в функцию выражение не вычисляется, но при этом ClickHouse применяет к этому выражению индекс таким же образом, как если бы выражение участвовало в запросе без indexHint.

Возвращаемое значение

- 1.

Пример

Рассмотрим таблицу с тестовыми данными ontime.

```
SELECT count() FROM ontime

┌─count()─┐
│ 4276457 │
└─────────┘
```

В таблице есть индексы по полям (FlightDate, (Year, FlightDate)).

Выполним выборку по дате следующим образом:

```
:) 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.)
```

В этой выборке индекс не используется и ClickHouse обработал всю таблицу (Processed 4.28 million rows). Для подключения индекса выберем конкретную дату и выполним следующий запрос:

```
:) SELECT FlightDate AS k, count() FROM ontime WHERE k = '2017-09-15' GROUP BY k ORDER BY k

SELECT
    FlightDate AS k,
    count()
FROM ontime
WHERE k = '2017-09-15'
GROUP BY k
ORDER BY k ASC

┌──────────k─┬─count()─┐
│ 2017-09-15 │   16428 │
└────────────┴─────────┘

1 rows in set. Elapsed: 0.014 sec. Processed 32.74 thousand rows, 65.49 KB (2.31 million rows/s., 4.63 MB/s.)
```

В последней строке выдачи видно, что благодаря использованию индекса, ClickHouse обработал значительно меньшее количество строк (Processed 32.74 thousand rows).

Теперь передадим выражение k = '2017-09-15' в функцию indexHint:

```
:) 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.)
```

В ответе на запрос видно, что ClickHouse применил индекс таким же образом, что и в предыдущий раз (Processed 32.74 thousand rows). Однако по результирующему набору строк видно, что выражение k = '2017-09-15' не использовалось при формировании результата.

Поскольку индекс в ClickHouse разреженный, то при чтении диапазона в ответ попадают "лишние" данные, в данном случае соседние даты. Функция indexHint позволяет их увидеть.

replicate

Создает массив, заполненный одним значением.

Используется для внутренней реализации arrayJoin.

```
replicate(x, arr)
```

Параметры

- arr — Исходный массив. ClickHouse создаёт новый массив такой же длины как исходный и заполняет его значением x.
- x — Значение, которым будет заполнен результирующий массив.

Выходное значение

- Массив, заполненный значением x.

Пример

```
SELECT replicate(1, ['a', 'b', 'c'])

┌─replicate(1, ['a', 'b', 'c'])─┐
│ [1,1,1]                       │
└───────────────────────────────┘
```

Агрегатные функции

Агрегатные функции работают в привычном для специалистов по базам данных смысле.

ClickHouse поддерживает также:

- Параметрические агрегатные функции, которые помимо столбцов принимаю и другие параметры.

- Комбинаторы, которые изменяют поведение агрегатных фунций.

Обработка NULL

При агрегации все NULL пропускаются.

Примеры

Рассмотрим таблицу:

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

Выполним суммирование значений в столбце y:

```
:) 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.
```

Функция sum работает с NULL как с 0. В частности, это означает, что если на вход в функцию подать выборку, где все значения NULL, то результат будет 0, а не NULL.

Теперь с помощью фукции groupArray сформируем массив из стобца y:

```
:) 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 не включает NULL в результирующий массив.

Справочник функций

count()

Считает количество строк. Принимает ноль аргументов, возвращает UInt64.
Не поддерживается синтаксис COUNT (DISTINCT x). Для этого существует агрегатная функцияuniq.

Запрос вида SELECT count() FROM table не оптимизируется, так как количество записей в таблице нигде не хранится отдельно. Из таблицы будет выбран какой-нибудь достаточно маленький столбец, и будет посчитано количество значений в нём.

any(x)

Выбирает первое попавшееся значение.
Порядок выполнения запроса может быть произвольным и даже каждый раз разным, поэтому результат данной функции недетерминирован.
Для получения детерминированного результата, можно использовать функции min или max вместо any.

В некоторых случаях, вы всё-таки можете рассчитывать на порядок выполнения запроса. Это - случаи, когда SELECT идёт из подзапроса, в котором используется ORDER BY.

При наличии в запросе SELECT секции GROUP BY или хотя бы одной агрегатной функции, ClickHouse (в отличие от, например, MySQL) требует, чтобы все выражения в секциях SELECT, HAVING, ORDER BY вычислялись из ключей или из агрегатных функций. То есть, каждый выбираемый из таблицы столбец, должен использоваться либо в ключах, либо внутри агрегатных функций. Чтобы получить поведение, как в MySQL, вы можете поместить остальные столбцы в агрегатную функцию any.

anyHeavy(x)

Выбирает часто встречающееся значение с помощью алгоритма "heavy hitters". Если существует значение, которое встречается чаще, чем в половине случаев, в каждом потоке выполнения запроса, то возвращается данное значение. В общем случае, результат недетерминирован.

```
anyHeavy(column)
```

Аргументы

- column – Имя столбца.

Пример

Возьмём набор данных OnTime и выберем произвольное часто встречающееся значение в столбце AirlineID.

```
SELECT anyHeavy(AirlineID) AS res
FROM ontime
```

```
┌─res─┐
│ 19690 │
└─────┘
```

anyLast(x)

Выбирает последнее попавшееся значение.
Результат так же недетерминирован, как и для функции any.

groupBitAnd

Применяет побитовое И для последовательности чисел.

```
groupBitAnd(expr)
```

Параметры

expr – Выражение, результат которого имеет тип UInt*.

Возвращаемое значение

Значение типа UInt*.

Пример

Тестовые данные:

```
binary     decimal
00101100 = 44
00011100 = 28
00001101 = 13
01010101 = 85
```

Запрос:

```
SELECT groupBitAnd(num) FROM t
```

Где num – столбец с тестовыми данными.

Результат:

```
binary     decimal
00000100 = 4
```

groupBitOr

Применяет побитовое ИЛИ для последовательности чисел.

```
groupBitOr (expr)
```

Параметры

expr – Выражение, результат которого имеет тип UInt*.

Возвращаемое значение

Значение типа UInt*.

Пример

Тестовые данные:

```
binary     decimal
00101100 = 44
00011100 = 28
00001101 = 13
01010101 = 85
```

Запрос:

```
SELECT groupBitOr(num) FROM t
```

Где num – столбец с тестовыми данными.

Результат:

```
binary     decimal
01111101 = 125
```

groupBitXor

Применяет побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ для последовательности чисел.

```
groupBitXor(expr)
```

Параметры

expr – Выражение, результат которого имеет тип UInt*.

Возвращаемое значение

Значение типа UInt*.

Пример

Тестовые данные:

```
binary     decimal
00101100 = 44
00011100 = 28
00001101 = 13
01010101 = 85
```

Запрос:

```
SELECT groupBitXor(num) FROM t
```

Где num – столбец с тестовыми данными.

Результат:

```
binary     decimal
01101000 = 104
```

min(x)

Вычисляет минимум.

max(x)

Вычисляет максимум.

argMin(arg, val)

Вычисляет значение arg при минимальном значении val. Если есть несколько разных значений arg для минимальных значений val, то выдаётся первое попавшееся из таких значений.

Пример:

```
┌─user─────┬─salary─┐
│ director │   5000 │
│ manager  │   3000 │
│ worker   │   1000 │
└──────────┴────────┘

SELECT argMin(user, salary) FROM salary

┌─argMin(user, salary)─┐
│ worker               │
└──────────────────────┘
```

argMax(arg, val)

Вычисляет значение arg при максимальном значении val. Если есть несколько разных значений arg для максимальных значений val, то выдаётся первое попавшееся из таких значений.

sum(x)

Вычисляет сумму.
Работает только для чисел.

sumWithOverflow(x)

Вычисляет сумму чисел, используя для результата тот же тип данных, что и для входных параметров. Если сумма выйдет за максимальное значение для заданного типа данных, то функция вернёт ошибку.

Работает только для чисел.

sumMap(key, value)

Производит суммирование массива 'value' по соответствующим ключам заданным в массиве 'key'.
Количество элементов в 'key' и 'value' должно быть одинаковым для каждой строки, для которой происходит суммирование.
Возвращает кортеж из двух массивов - ключи в отсортированном порядке и значения, просуммированные по соответствующим ключам.

Пример:

```
CREATE TABLE sum_map(
    date Date,
    timeslot DateTime,
    statusMap Nested(
        status UInt16,
        requests UInt64
    )
) ENGINE = Log;
INSERT INTO sum_map VALUES
    ('2000-01-01', '2000-01-01 00:00:00', [1, 2, 3], [10, 10, 10]),
    ('2000-01-01', '2000-01-01 00:00:00', [3, 4, 5], [10, 10, 10]),
    ('2000-01-01', '2000-01-01 00:01:00', [4, 5, 6], [10, 10, 10]),
    ('2000-01-01', '2000-01-01 00:01:00', [6, 7, 8], [10, 10, 10]);
SELECT
    timeslot,
    sumMap(statusMap.status, statusMap.requests)
FROM sum_map
GROUP BY timeslot
```

```
┌────────────timeslot─┬─sumMap(statusMap.status, statusMap.requests)─┐
│ 2000-01-01 00:00:00 │ ([1,2,3,4,5],[10,10,20,10,10])               │
│ 2000-01-01 00:01:00 │ ([4,5,6,7,8],[10,10,20,10,10])               │
└─────────────────────┴──────────────────────────────────────────────┘
```

avg(x)

Вычисляет среднее.
Работает только для чисел.
Результат всегда Float64.

uniq(x)

Приближённо вычисляет количество различных значений аргумента. Работает для чисел, строк, дат, дат-с-временем, для нескольких аргументов и аргументов-кортежей.

Используется алгоритм типа adaptive sampling: в качестве состояния вычислений используется выборка значений хэшей элементов, размером до 65536.
Алгоритм является очень точным для множеств небольшой кардинальности (до 65536) и очень эффективным по CPU (при расчёте не слишком большого количества таких функций, использование uniq почти так же быстро, как использование других агрегатных функций).

Результат детерминирован (не зависит от порядка выполнения запроса).

Функция обеспечивает высокую точность даже для множеств с высокой кардинальностью (более 10 миллиардов элементов). Рекомендуется для использования по умолчанию.

uniqCombined(HLL_precision)(x)

Приближённо вычисляет количество различных значений аргумента. Работает для чисел, строк, дат, дат-с-временем, для нескольких аргументов и аргументов-кортежей.

Используется комбинация трёх алгоритмов: массив, хэш-таблица и HyperLogLog с таблицей коррекции погрешности. Для небольшого количества различных значений используется массив; при увеличении количества значений, используется хэш таблица, до тех пор, пока её размер меньше размера HyperLogLog структуры. При дальнейшем увеличении количества значений, используется HyperLogLog структура, имеющая фиксированный размер в памяти.

Параметр HLL_precision - логарифм по основанию 2 от количества ячеек в HyperLogLog. Параметер можно не указывать (для этого, опустите первую пару скобок). По-умолчанию - 17. При использовании параметра по-умолчанию, расход памяти в несколько раз меньше, чем у функции uniq, а точность в несколько раз выше. Скорость работы чуть ниже, чем у функции uniq, но иногда может быть даже выше - в случае распределённых запросов, в которых по сети передаётся большое количество состояний агрегации. Каждая ячейка имеет размер 6 бит, что даёт 96 KiB для размера HyperLogLog структуры.

Результат детерминирован (не зависит от порядка выполнения запроса).

Функция uniqCombined является хорошим выбором по умолчанию для подсчёта количества различных значений, но стоит иметь ввиду что для множеств большой кардинальности (200 миллионов различных элементов и больше) ошибка оценки становится существенно больше расчётной из-за недостаточно хорошего выбора хэш-функции.

uniqHLL12(x)

Приближённо вычисляет количество различных значений аргумента, используя алгоритм HyperLogLog. Используется 212 5-битовых ячеек. Размер состояния чуть больше 2.5 КБ. Результат не очень точный (ошибка до ~10%) для небольших множеств (<10K элементов). Однако, для множеств с большой кардинальностью (10K - 100M) результат имеет ошибку до ~1.6%. Начиная со 100M, ошибка оценки увеличивается и для множеств огромной кардинальности (1B+ элементов) результат будет очень неточным.

Результат детерминирован (не зависит от порядка выполнения запроса).

Мы не рекомендуем использовать эту функцию. В большинстве случаев, используйте функцию uniq или uniqCombined.

uniqExact(x)

Вычисляет количество различных значений аргумента, точно.
Не стоит бояться приближённых расчётов. Поэтому, используйте лучше функцию uniq.
Функцию uniqExact следует использовать, если вам точно нужен точный результат.

Функция uniqExact расходует больше оперативки, чем функция uniq, так как размер состояния неограниченно растёт по мере роста количества различных значений.

groupArray(x), groupArray(max_size)(x)

Составляет массив из значений аргумента.
Значения в массив могут быть добавлены в любом (недетерминированном) порядке.

Вторая версия (с параметром max_size) ограничивает размер результирующего массива max_size элементами.
Например, groupArray(1)(x) эквивалентно [any(x)].

В некоторых случаях, вы всё же можете рассчитывать на порядок выполнения запроса. Это — случаи, когда SELECT идёт из подзапроса, в котором используется ORDER BY.

groupArrayInsertAt(x)

Вставляет в массив значение в заданную позицию.

Принимает на вход значение и позицию. Если на одну и ту же позицию вставляется несколько значений, в результирующем массиве может оказаться любое (первое в случае однопоточного выполнения). Если в позицию не вставляется ни одного значения, то позиции присваивается значение по умолчанию.

Опциональные параметры:

- Значение по умолчанию для подстановки на пустые позиции.
- Длина результирующего массива. Например, если вы хотите получать массивы одинакового размера для всех агрегатных ключей. При использовании этого параметра значение по умолчанию задавать обязательно.

groupUniqArray(x)

Составляет массив из различных значений аргумента. Расход оперативки такой же, как у функции uniqExact.

quantile(level)(x)

Приближённо вычисляет квантиль уровня level. level - константа, число с плавающей запятой от 0 до 1. Рекомендуется использовать значения level в диапазоне [0.01, 0.99].
Не используйте значение 'level' равное 0 или 1 – используйте функции 'min' и 'max' для этих случаев.

В этой функции, равно как и во всех функциях для расчёта квантилей, параметр level может быть не указан. В таком случае, он принимается равным 0.5 - то есть, функция будет вычислять медиану.

Работает для чисел, дат, дат-с-временем.
Для чисел возвращает Float64, для дат - дату, для дат-с-временем - дату-с-временем.

Используется reservoir sampling с размером резервуара до 8192.
При необходимости, результат выдаётся с линейной аппроксимацией из двух соседних значений.
Этот алгоритм обеспечивает весьма низкую точность расчёта. Смотрите также функции quantileTiming, quantileTDigest, quantileExact.

Результат зависит от порядка выполнения запроса, и является недетерминированным.

При использовании нескольких функций quantile (и аналогичных) с разными уровнями в запросе, внутренние состояния не объединяются (то есть, запрос работает менее эффективно, чем мог бы). В этом случае, используйте функцию quantiles (и аналогичные).

quantileDeterministic(level)(x, determinator)

Работает аналогично функции quantile, но, в отличие от неё, результат является детерминированным и не зависит от порядка выполнения запроса.

Для этого, функция принимает второй аргумент - «детерминатор». Это некоторое число, хэш от которого используется вместо генератора случайных чисел в алгоритме reservoir sampling. Для правильной работы функции, одно и то же значение детерминатора не должно встречаться слишком часто. В качестве детерминатора вы можете использовать идентификатор события, идентификатор посетителя и т. п.

Не используйте эту функцию для расчёта таймингов. Для этого есть более подходящая функция - quantileTiming.

quantileTiming(level)(x)

Вычисляет квантиль уровня level с фиксированной точностью.
Работает для чисел. Предназначена для расчёта квантилей от времени загрузки страницы в миллисекундах.

Если значение больше 30000 (соответствует времени загрузки страницы большем 30 секундам) - результат приравнивается к 30000.

Если всего значений не больше примерно 5670, то вычисление точное.

Иначе:

- если время меньше 1024 мс., то вычисление точное.
- иначе вычисление идёт с округлением до числа, кратного 16 мс.

При передаче в функцию отрицательных значений, поведение не определено.

Возвращаемое значение имеет тип Float32. Когда в функцию не было передано ни одного значения (при использовании quantileTimingIf), возвращается nan. Это сделано, чтобы отличать такие случаи от нулей. Смотрите замечание о сортировке NaN-ов в разделе «Секция ORDER BY».

Результат детерминирован (не зависит от порядка выполнения запроса).

Для своей задачи (расчёт квантилей времени загрузки страниц), использование этой функции эффективнее и результат точнее, чем для функции quantile.

quantileTimingWeighted(level)(x, weight)

Отличается от функции quantileTiming наличием второго аргумента - «веса». Вес - неотрицательное целое число.
Результат считается так же, как если бы в функцию quantileTiming значение x было передано weight количество раз.

quantileExact(level)(x)

Вычисляет квантиль уровня level точно. Для этого, все переданные значения складываются в массив, который затем частично сортируется. Поэтому, функция потребляет O(n) памяти, где n - количество переданных значений. Впрочем, для случая маленького количества значений, функция весьма эффективна.

quantileExactWeighted(level)(x, weight)

Вычисляет квантиль уровня level точно. При этом, каждое значение учитывается с весом weight - как будто оно присутствует weight раз. Аргументы функции можно рассматривать как гистограммы, где значению x соответствует «столбик» гистограммы высоты weight, а саму функцию можно рассматривать как суммирование гистограмм.

В качестве алгоритма используется хэш-таблица. Из-за этого, в случае, если передаваемые значения часто повторяются, функция потребляет меньше оперативки, чем quantileExact. Вы можете использовать эту функцию вместо quantileExact, указав в качестве веса число 1.

quantileTDigest(level)(x)

Вычисляет квантиль уровня level приближенно, с использованием алгоритма t-digest. Максимальная погрешность составляет 1%. Расход памяти на состояние пропорционален логарифму от количества переданных значений.

Производительность функции ниже quantile, quantileTiming. По соотношению размера состояния и точности, функция существенно лучше, чем quantile.

Результат зависит от порядка выполнения запроса, и является недетерминированным.

median(x)

Для всех quantile-функций, также присутствуют соответствующие median-функции: median, medianDeterministic, medianTiming, medianTimingWeighted, medianExact, medianExactWeighted, medianTDigest. Они являются синонимами и их поведение ничем не отличается.

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

Для всех quantile-функций, также присутствуют соответствующие quantiles-функции: quantiles, quantilesDeterministic, quantilesTiming, quantilesTimingWeighted, quantilesExact, quantilesExactWeighted, quantilesTDigest. Эти функции за один проход вычисляют все квантили перечисленных уровней и возвращают массив вычисленных значений.

varSamp(x)

Вычисляет величину $\Sigma((x - \bar{x})^2) / (n - 1)$, где n - размер выборки, $\bar{x}$- среднее значение x.

Она представляет собой несмещённую оценку дисперсии случайной величины, если переданные в функцию значения являются выборкой этой случайной величины.

Возвращает Float64. В случае, когда $n <= 1$, возвращается $+\infty$.

varPop(x)

Вычисляет величину $\Sigma((x - \bar{x})^2) / n$, где n - размер выборки, $\bar{x}$- среднее значение x.

То есть, дисперсию для множества значений. Возвращает Float64.

stddevSamp(x)

Результат равен квадратному корню от varSamp(x).

stddevPop(x)

Результат равен квадратному корню от varPop(x).

topK(N)(column)

Возвращает массив наиболее часто встречающихся значений в указанном столбце. Результирующий массив упорядочен по убыванию частоты значения (не по самим значениям).

Реализует Filtered Space-Saving алгоритм для анализа TopK, на основе reduce-and-combine алгоритма из методики Parallel Space Saving.

```
topK(N)(column)
```

Функция не дает гарантированного результата. В некоторых ситуациях могут возникать ошибки, и функция возвращает частые, но не наиболее частые значения.

Рекомендуем использовать значения $N < 10$, при больших N снижается производительность. Максимально возможное значение $N = 65536$.

Аргументы

- 'N' - Количество значений.
- 'x' – Столбец.

Пример

Возьмём набор данных OnTime и выберем 3 наиболее часто встречающихся значения в столбце AirlineID.

```
SELECT topK(3)(AirlineID) AS res
FROM ontime
```

```
┌─res─────────────────┐
│ [19393,19790,19805] │
└─────────────────────┘
```

covarSamp(x, y)

Вычисляет величину $\Sigma((x - \bar{x})(y - \bar{y})) / (n - 1)$.

Возвращает Float64. В случае, когда $n <= 1$, возвращается $+\infty$.

covarPop(x, y)

Вычисляет величину Σ((x - x̄)(y - ȳ)) / n.

corr(x, y)

Вычисляет коэффициент корреляции Пирсона: Σ((x - x̄)(y - ȳ)) / sqrt(Σ((x - x̄)^2) * Σ((y - ȳ)^2)).

Комбинаторы агрегатных функций

К имени агрегатной функции может быть приписан некоторый суффикс. При этом, работа агрегатной функции некоторым образом модифицируется.

-If

К имени любой агрегатной функции может быть приписан суффикс -If. В этом случае, агрегатная функция принимает ещё один дополнительный аргумент - условие (типа UInt8). Агрегатная функция будет обрабатывать только те строки, для которых условие сработало. Если условие ни разу не сработало - возвращается некоторое значение по умолчанию (обычно - нули, пустые строки).

Примеры: sumIf(column, cond), countIf(cond), avgIf(x, cond), quantilesTimingIf(level1, level2)(x, cond), argMinIf(arg, val, cond) и т. п.

С помощью условных агрегатных функций, вы можете вычислить агрегаты сразу для нескольких условий, не используя подзапросы и JOIN-ы.
Например, в Яндекс.Метрике, условные агрегатные функции используются для реализации функциональности сравнения сегментов.

-Array

К имени любой агрегатной функции может быть приписан суффикс -Array. В этом случае, агрегатная функция вместо аргументов типов T принимает аргументы типов Array(T) (массивы). Если агрегатная функция принимает несколько аргументов, то это должны быть массивы одинаковых длин. При обработке массивов, агрегатная функция работает, как исходная агрегатная функция по всем элементам массивов.

Пример 1: sumArray(arr) - просуммировать все элементы всех массивов arr. В данном примере можно было бы написать проще: sum(arraySum(arr)).

Пример 2: uniqArray(arr) - посчитать количество уникальных элементов всех массивов arr. Это можно было бы сделать проще: uniq(arrayJoin(arr)), но не всегда есть возможность добавить arrayJoin в запрос.

Комбинаторы -If и -Array можно сочетать. При этом, должен сначала идти Array, а потом If. Примеры: uniqArrayIf(arr, cond), quantilesTimingArrayIf(level1, level2)(arr, cond). Из-за такого порядка получается, что аргумент cond не должен быть массивом.

-State

В случае применения этого комбинатора, агрегатная функция возвращает не готовое значение (например, в случае функции uniq — количество уникальных значений), а промежуточное состояние агрегации (например, в случае функции uniq — хэш-таблицу для расчёта количества уникальных значений), которое имеет тип AggregateFunction(...) и может использоваться для дальнейшей обработки или может быть сохранено в таблицу для последующей доагрегации - смотрите разделы «AggregatingMergeTree» и «функции для работы с промежуточными состояниями агрегации».

-Merge

В случае применения этого комбинатора, агрегатная функция будет принимать в качестве аргумента промежуточное состояние агрегации, доагрегировать (объединять вместе) эти состояния, и возвращать готовое значение.

-MergeState.

Выполняет слияние промежуточных состояний агрегации, аналогично комбинатору -Merge, но возвращает не готовое значение, а промежуточное состояние агрегации, аналогично комбинатору -State.

-ForEach

Преобразует агрегатную функцию для таблиц в агрегатную функцию для массивов, которая применяет агрегирование для соответствующих элементов массивов и возвращает массив результатов. Например, sumForEach для массивов [1, 2], [3, 4, 5] и [6, 7] даст результат [10, 13, 5], сложив соответственные элементы массивов.

Параметрические агрегатные функции

Некоторые агрегатные функции могут принимать не только столбцы-аргументы (по которым производится свёртка), но и набор параметров - констант для инициализации. Синтаксис - две пары круглых скобок вместо одной. Первая - для параметров, вторая - для аргументов.

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

Сопоставление с образцом для цепочки событий.

pattern - строка, содержащая шаблон для сопоставления. Шаблон похож на регулярное выражение.

time - время события, тип DateTime

cond1, cond2 ... - от одного до 32 аргументов типа UInt8 - признаков, было ли выполнено некоторое условие для события.

Функция собирает в оперативке последовательность событий. Затем производит проверку на соответствие этой последовательности шаблону.
Возвращает UInt8 - 0, если шаблон не подходит и 1, если шаблон подходит.

Пример: sequenceMatch('(?1).*(?2)')(EventTime, URL LIKE '%company%', URL LIKE '%cart%')

- была ли цепочка событий, в которой посещение страницы с адресом, содержащим company было раньше по времени посещения страницы с адресом, содержащим cart.

Это вырожденный пример. Его можно записать с помощью других агрегатных функций:

```
minIf(EventTime, URL LIKE '%company%') < maxIf(EventTime, URL LIKE '%cart%').
```

Но в более сложных случаях, такого решения нет.

Синтаксис шаблонов:

(?1) - ссылка на условие (вместо 1 - любой номер);

.* - произвольное количество любых событий;

(?t>=1800) - условие на время;

за указанное время допускается любое количество любых событий;

вместо >= могут использоваться операторы <, >, <=;

вместо 1800 может быть любое число;

События, произошедшие в одну секунду, могут оказаться в цепочке в произвольном порядке. От этого может зависеть результат работы функции.

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

Аналогично функции sequenceMatch, но возвращает не факт наличия цепочки событий, а UInt64 - количество найденных цепочек.
Цепочки ищутся без перекрытия. То есть, следующая цепочка может начаться только после окончания предыдущей.

windowFunnel(window)(timestamp, cond1, cond2, cond3, ...)

Отыскивает цепочки событий в скользящем окне по времени и вычисляет максимальное количество произошедших событий из цепочки.

```
windowFunnel(window)(timestamp, cond1, cond2, cond3, ...)
```

Параметры

- window — ширина скользящего окна по времени в секундах.
- timestamp — имя столбца, содержащего отметки времени. Тип данных DateTime или UInt32.
- cond1, cond2... — условия или данные, описывающие цепочку событий. Тип данных — UInt8. Значения могут быть 0 или 1.

Алгоритм

- Функция отыскивает данные, на которых срабатывает первое условие из цепочки, и присваивает счетчику событий значение 1. С этого же момента начинается отсчет времени скользящего окна.
- Если в пределах окна последовательно попадаются события из цепочки, то счетчик увеличивается. Если последовательность событий нарушается, то счетчик не растёт.
- Если в данных оказалось несколько цепочек разной степени завершенности, то функция выдаст только размер самой длинной цепочки.

Возвращаемое значение

- Целое число. Максимальное количество последовательно сработавших условий из цепочки в пределах скользящего окна по времени. Исследуются все цепочки в выборке.

Пример

Определим, успевает ли пользователь за час выбрать телефон в интернет-магазине и купить его.

Зададим следующую цепочку событий:

1. Пользователь вошел в личный кабинет магазина (eventID=1001).
2. Пользователь ищет телефон (eventID = 1003, product = 'phone').
3. Пользователь сделал заказ (eventID = 1009).

Чтобы узнать, как далеко пользователь user_id смог пройти по цепочке за час в январе 2017-го года, составим запрос:

```
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
```

В результате мы можем получить 0, 1, 2 или 3 в зависимости от действий пользователя.

uniqUpTo(N)(x)

Вычисляет количество различных значений аргумента, если оно меньше или равно N.
В случае, если количество различных значений аргумента больше N, возвращает N + 1.

Рекомендуется использовать для маленьких N - до 10. Максимальное значение N - 100.

Для состояния агрегатной функции используется количество оперативки равное 1 + N * размер одного значения байт.
Для строк запоминается некриптографический хэш, имеющий размер 8 байт. То есть, для строк вычисление приближённое.

Функция также работает для нескольких аргументов.

Работает максимально быстро за исключением патологических случаев, когда используется большое значение N и количество уникальных значений чуть меньше N.

Пример применения:

```
Задача: показывать в отчёте только поисковые фразы, по которым было хотя бы 5 уникальных посетителей.
Решение: пишем в запросе GROUP BY SearchPhrase HAVING uniqUpTo(4)(UserID) >= 5
```

Табличные функции

Табличные функции могут указываться в секции FROM вместо имени БД и таблицы.
Табличные функции можно использовать только если не выставлена настройка readonly.
Табличные функции не имеют отношения к другим функциям.

file

Создаёт таблицу из файла.

```
file(path, format, structure)
```

Входные параметры

- path — относительный путь до файла от user_files_path.
- format — формат файла.
- structure — структура таблицы. Формат 'colunmn1_name column1_ype, column2_name column2_type, ...'.

Возвращаемое значение

Таблица с указанной структурой, предназначенная для чтения или записи данных в указанном файле.

Пример

Настройка user_files_path и содержимое файла 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
```

Таблица из test.csv и выборка первых двух строк из неё:

```
SELECT *
FROM file('test.csv', 'CSV', 'column1 UInt32, column2 UInt32, column3 UInt32')
LIMIT 2
```

```
┌─column1─┬─column2─┬─column3─┐
│      1 │      2 │      3 │
│      3 │      2 │      1 │
└─────────┴─────────┴─────────┘
```

merge

merge(db_name, 'tables_regexp') - создаёт временную таблицу типа Merge. Подробнее смотрите раздел "Движки таблиц, Merge".

Структура таблицы берётся из первой попавшейся таблицы, подходящей под регулярное выражение.

numbers

numbers(N) - возвращает таблицу с единственным столбцом number (UInt64), содержащим натуральные числа от 0 до N-1.
numbers(N, M) - возвращает таблицу с единственным столбцом number (UInt64), содержащим натуральные числа от N to (N + M - 1).

Так же как и таблица system.numbers может использоваться для тестов и генерации последовательных значений. Функция numbers(N, M) работает более эффективно, чем выборка из system.numbers.

Следующие запросы эквивалентны:

```
SELECT * FROM numbers(10);
SELECT * FROM numbers(0,10);
SELECT * FROM system.numbers LIMIT 10;
```

Примеры:

```
-- генерация последовательности всех дат от 2010-01-01 до 2010-12-31
select toDate('2010-01-01') + number as d FROM numbers(365);
```

remote, remoteSecure

Позволяет обратиться к удалённым серверам без создания таблицы типа Distributed.

Сигнатуры:

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

addresses_expr - выражение, генерирующее адреса удалённых серверов. Это может быть просто один адрес сервера. Адрес сервера - это хост:порт, или только хост. Хост может быть указан в виде имени сервера, или в виде IPv4 или IPv6 адреса. IPv6 адрес указывается в квадратных скобках. Порт - TCP-порт удалённого сервера. Если порт не указан, используется tcp_port из конфигурационного файла сервера (по умолчанию - 9000).

Важно

С IPv6-адресом обязательно нужно указывать порт.

Примеры:

```
example01-01-1
example01-01-1:9000
localhost
127.0.0.1
[::]:9000
[2a02:6b8:0:1111::11]:9000
```

Адреса можно указать через запятую, в этом случае ClickHouse обработает запрос как распределённый, т.е. отправит его по всем указанным адресам как на шарды с разными данными.

Пример:

```
example01-01-1,example01-02-1
```

Часть выражения может быть указана в фигурных скобках. Предыдущий пример может быть записан следующим образом:

```
example01-0{1,2}-1
```

В фигурных скобках может быть указан диапазон (неотрицательных целых) чисел через две точки. В этом случае, диапазон раскрывается в множество значений, генерирующих адреса шардов. Если запись первого числа начинается с нуля, то значения формируются с таким же выравниванием нулями. Предыдущий пример может быть записан следующим образом:

```
example01-{01..02}-1
```

При наличии нескольких пар фигурных скобок, генерируется прямое произведение соответствующих множеств.

Адреса или их фрагменты в фигурных скобках можно указать через символ |. В этом случае, соответствующие множества адресов понимаются как реплики - запрос будет отправлен на первую живую реплику. При этом, реплики перебираются в порядке, согласно текущей настройке load_balancing.

Пример:

```
example01-{01..02}-{1|2}
```

В этом примере указано два шарда, в каждом из которых имеется две реплики.

Количество генерируемых адресов ограничено константой - сейчас это 1000 штук.

Использование табличной функции remote менее оптимально, чем создание таблицы типа Distributed, так как в этом случае, соединения с серверами устанавливаются заново при каждом запросе, в случае задания имён хостов, делается резолвинг имён, а также не ведётся подсчёт ошибок при работе с разными репликами. При обработке большого количества запросов, всегда создавайте Distributed таблицу заранее, не используйте табличную функцию remote.

Табличная функция remote может быть полезна для следующих случаях:

- обращение на конкретный сервер в целях сравнения данных, отладки и тестирования;
- запросы между разными кластерами ClickHouse в целях исследований;
- нечастых распределённых запросов, задаваемых вручную;
- распределённых запросов, где набор серверов определяется каждый раз заново.

Если пользователь не задан,то используется default.
Если пароль не задан, то используется пустой пароль.

remoteSecure - аналогично функции remote, но с соединением по шифрованому каналу. Порт по умолчанию - tcp_port_secure из конфига или 9440.

url

url(URL, format, structure) - возвращает таблицу со столбцами, указанными в structure, созданную из данных находящихся по URL в формате format.

URL - адрес, по которому сервер принимает GET и/или POST запросы по протоколу HTTP или HTTPS.

format - формат данных.

structure - структура таблицы в форме 'UserID UInt64, Name String'. Определяет имена и типы столбцов.

Пример

```
-- получение 3-х строк таблицы, состоящей из двух колонк типа String и UInt32 от сервера, отдающего данные в формате CSV
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 — Flag that sets query substitution INSERT INTO to REPLACE INTO. If replace_query=1, the query is replaced.
- on_duplicate_clause — The ON DUPLICATE KEY on_duplicate_clause expression that is added to the INSERT query.

  Example: INSERT INTO t (c1,c2) VALUES ('a', 2) ON DUPLICATE KEY UPDATE c2 = c2 + 1, where on_duplicate_clause is UPDATE c2 = c2 + 1. See MySQL documentation to find which on_duplicate_clause you can use with ON DUPLICATE KEY clause.

  To specify on_duplicate_clause you need to pass 0 to the replace_query parameter. If you simultaneously pass replace_query = 1 and on_duplicate_clause, ClickHouse generates an exception.

At this time, simple WHERE clauses and elements of AND chain in WHERE clause such as =, !=, >, >=, <, <=, LIKE, IN with compatible functions are also pushed down for execution on the MySQL server.

The rest of the conditions and the LIMIT constraint are executed in ClickHouse only after the query to MySQL finishes.

Returned Value

A table object with the same columns as the original MySQL table.

Usage Example

Table in MySQL:

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

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

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

Selection of the data from ClickHouse:

```
SELECT * FROM mysql('localhost:3306', 'test', 'test', 'bayonet', '123')
```

```
┌─int_id─┬─int_nullable─┬─float─┬─float_nullable─┐
│      1 │         NULL │     2 │           NULL │
└────────┴──────────────┴───────┴────────────────┘
```

See Also

- The 'MySQL' table engine
- Using MySQL as a source of extenal dictionary

jdbc

jdbc(jdbc_connection_uri, schema, table) - возвращает таблицу, соединение с которой происходит через JDBC-драйвер.

Для работы этой табличной функциии требуется отдельно запускать приложение clickhouse-jdbc-bridge. Данная функция поддерживает Nullable типы (на основании DDL таблицы к которой происходит запрос).

Пример

```
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 driver.

```
odbc(connection_settings, external_database, external_table)
```

Function Parameters

- connection_settings — Name of the section with connection settings in the odbc.ini file.
- external_database — Database in an external DBMS.
- external_table — A name of the table in external_database.

To implement ODBC connection, ClickHouse uses the separate program clickhouse-odbc-bridge. ClickHouse starts this program automatically when it is required. The ODBC bridge program is installed by the same package with the ClickHouse server.

This function supports the Nullable data type (based on DDL of remote table that is queried).

Examples

```
select * from odbc('DSN=connection_settings_name', 'external_database_name', 'external_table_name')
```

Some examples of how to use external dictionaries through ODBC you can find in the ODBC section of external dictionaries configuration.

Словари

Словарь — это отображение (ключ -> атрибуты), которое удобно использовать для различного вида справочников.

ClickHouse поддерживает специальные функции для работы со словарями, которые можно использовать в запросах. Проще и эффективнее использовать словари с помощью функций, чем JOIN с таблицами-справочниками.

В словаре нельзя хранить значения NULL.

ClickHouse поддерживает:

- Встроенные словари со специфическим набором функций.
- Подключаемые (внешние) словари с набором функций.

Внешние словари

Существует возможность подключать собственные словари из различных источников данных. Источником данных для словаря может быть локальный текстовый/исполняемый файл, HTTP(s) ресурс или другая СУБД. Подробнее смотрите в разделе "Источники внешних словарей".

ClickHouse:

- Полностью или частично хранит словари в оперативной памяти.
- Периодически обновляет их и динамически подгружает отсутствующие значения. Т.е. словари можно подгружать динамически.

Конфигурация внешних словарей находится в одном или нескольких файлах. Путь к конфигурации указывается в параметре dictionaries_config.

Словари могут загружаться при старте сервера или при первом использовании, в зависимости от настройки dictionaries_lazy_load.

Конфигурационный файл словарей имеет вид:

```
<yandex>
  <comment>Необязательный элемент с любым содержимым. Игнорируется сервером ClickHouse.</comment>

  <!--Необязательный элемент, имя файла с подстановками-->
  <include_from>/etc/metrika.xml</include_from>


  <dictionary>
    <!-- Конфигурация словаря -->
  </dictionary>

  ...

  <dictionary>
    <!-- Конфигурация словаря -->
  </dictionary>
</yandex>
```

В одном файле можно сконфигурировать произвольное количество словарей. Формат файла сохраняется даже если словарь один (т.е. `<yandex><dictionary> <!--configuration--> </dictionary></yandex>`).

можете преобразовывать значения по небольшому словарю, описав его в запросе SELECT (см. функцию transform). Эта функциональность не связана с внешними словарями.

Смотрите также:

- Настройка внешнего словаря
- Хранение словарей в памяти
- Обновление словарей
- Источники внешних словарей
- Ключ и поля словаря
- Функции для работы с внешними словарями

Настройка внешнего словаря

Конфигурация словаря имеет следующую структуру:

```
<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 - Идентификатор, под которым словарь будет доступен для использования. Используйте символы [a-zA-Z0-9_\-].
- source - Источник словаря.
- layout - Размещение словаря в памяти.
- structure - Структура словаря. Ключ и атрибуты, которые можно получить по ключу.
- lifetime - Периодичность обновления словарей.

Хранение словарей в памяти

Словари можно размещать в памяти множеством способов.

Рекомендуем flat, hashed и complex_key_hashed. Скорость обработки словарей при этом максимальна.

Размещение с кэшированием не рекомендуется использовать из-за потенциально низкой производительности и сложностей в подборе оптимальных параметров. Читайте об этом подробнее в разделе "cache".

Повысить производительнось словарей можно следующими способами:

- Вызывать функцию для работы со словарём после GROUP BY.
- Помечать извлекаемые атрибуты как инъективные. Атрибут называется инъективным, если разным ключам соответствуют разные значения атрибута. Тогда при использовании в GROUP BY функции, достающей значение атрибута по ключу, эта функция автоматически выносится из GROUP BY.

При ошибках работы со словарями ClickHouse генерирует исключения. Например, в следующих ситуациях:

- При обращении к словарю, который не удалось загрузить.
- При ошибке запроса к cached-словарю.

Список внешних словарей и их статус можно посмотреть в таблице system.dictionaries.

Общий вид конфигурации:

```
<yandex>
  <dictionary>
    ...
    <layout>
      <layout_type>
        <!-- layout settings -->
      </layout_type>
    </layout>
    ...
  </dictionary>
</yandex>
```

Способы размещения словарей в памяти

- flat
- hashed
- cache
- range_hashed
- complex_key_hashed
- complex_key_cache
- ip_trie

flat

Словарь полностью хранится в оперативной памяти в виде плоских массивов. Объем памяти, занимаемой словарем? пропорционален размеру самого большого (по размеру) ключа.

Ключ словаря имеет тип UInt64 и его величина ограничена 500 000. Если при создании словаря обнаружен ключ больше, то ClickHouse бросает исключение и не создает словарь.

Поддерживаются все виды источников. При обновлении, данные (из файла, из таблицы) читаются целиком.

Это метод обеспечивает максимальную производительность среди всех доступных способов размещения словаря.

Пример конфигурации:

```
<layout>
  <flat />
</layout>
```

hashed

Словарь полностью хранится в оперативной памяти в виде хэш-таблиц. Словарь может содержать произвольное количество элементов с произвольными идентификаторами. На практике, количество ключей может достигать десятков миллионов элементов.

Поддерживаются все виды источников. При обновлении, данные (из файла, из таблицы) читаются целиком.

Пример конфигурации:

```
<layout>
  <hashed />
</layout>
```

complex_key_hashed

Тип размещения предназначен для использования с составными ключами. Аналогичен hashed.

Пример конфигурации:

```
<layout>
  <complex_key_hashed />
</layout>
```

range_hashed

Словарь хранится в оперативной памяти в виде хэш-таблицы с упорядоченным массивом диапазонов и соответствующих им значений.

Этот способ размещения работает также как и hashed и позволяет дополнительно к ключу использовать дипазоны по дате/времени, если они указаны в словаре.

Пример: таблица содержит скидки для каждого рекламодателя в виде:

```
+---------------+---------------------+-------------------+--------+
| 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   |
+---------------+---------------------+-------------------+--------+
```

Чтобы использовать выборку по диапазонам дат, необходимо в structure определить элементы range_min, range_max.

Пример:

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

Для работы с такими словарями в функцию dictGetT необходимо передавать дополнительный аргумент - дату: :

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

Функция возвращает значение для заданных id и диапазона дат, в который входит переданная дата.

Особенности алгоритма:

- Если не найден id или для найденного id не найден диапазон, то возвращается значение по умолчанию для словаря.
- Если есть перекрывающиеся диапазоны, то можно использовать любой подходящий.
- Если граница диапазона NULL или некорректная дата (1900-01-01, 2039-01-01), то диапазон считается открытым. Диапазон может быть открытым с обеих сторон.

Пример конфигурации:

```
<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

Словарь хранится в кэше, состоящем из фиксированного количества ячеек. Ячейки содержат часто используемые элементы.

При поиске в словаре сначала просматривается кэш. На каждый блок данных, все не найденные в кэше или устаревшие ключи запрашиваются у источника с помощью SELECT attrs... FROM db.table WHERE id IN (k1, k2, ...). Затем, полученные данные записываются в кэш.

Для cache-словарей может быть задано время устаревания lifetime данных в кэше. Если от загрузки данных в ячейке прошло больше времени, чем lifetime, то значение не используется, и будет запрошено заново при следующей необходимости его использовать.

Это наименее эффективный из всех способов размещения словарей. Скорость работы кэша очень сильно зависит от правильности настройки и сценария использования. Словарь типа cache показывает высокую производительность лишь при достаточно больших hit rate-ах (рекомендуется 99% и выше). Посмотреть средний hit rate можно в таблице system.dictionaries.

Чтобы увеличить производительность кэша, используйте подзапрос с LIMIT, а снаружи вызывайте функцию со словарём.

Поддерживаются источники: MySQL, ClickHouse, executable, HTTP.

Пример настройки:

```
<layout>
  <cache>
    <!-- Размер кэша в количестве ячеек. Округляется вверх до степени двух. -->
    <size_in_cells>1000000000</size_in_cells>
  </cache>
</layout>
```

Укажите достаточно большой размер кэша. Количество ячеек следует подобрать экспериментальным путём:

1. Выставить некоторое значение.
2. Запросами добиться полной заполненности кэша.
3. Оценить потребление оперативной памяти с помощью таблицы system.dictionaries.
4. Увеличивать/уменьшать количество ячеек до получения требуемого расхода оперативной памяти.

Warning

Не используйте в качестве источника ClickHouse, поскольку он медленно обрабатывает запросы со случайным чтением.

complex_key_cache

Тип размещения предназначен для использования с составными ключами. Аналогичен cache.

ip_trie

Тип размещения предназначен для сопоставления префиксов сети (IP адресов) с метаданными, такими как ASN.

Пример: таблица содержит префиксы сети и соответствующие им номера AS и коды стран:

```
 +-----------------+-------+--------+
| 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     |
+-----------------+-------+--------+
```

При использовании такого макета структура должна иметь составной ключ.

Пример:

```
<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>
   ...
```

Этот ключ должен иметь только один атрибут типа String, содержащий допустимый префикс IP. Другие типы еще не поддерживаются.

Для запросов необходимо использовать те же функции (dictGetT с кортежем), что и для словарей с составными ключами:

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

Функция принимает либо UInt32 для IPv4, либо FixedString(16) для IPv6:

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

Никакие другие типы не поддерживаются. Функция возвращает атрибут для префикса, соответствующего данному IP-адресу. Если есть перекрывающиеся префиксы, возвращается наиболее специфический.

Данные хранятся в побитовом дереве (trie), он должены полностью помещаться в оперативной памяти.

Обновление словарей

ClickHouse периодически обновляет словари. Интервал обновления для полностью загружаемых словарей и интервал инвалидации для кэшируемых словарей определяется в теге <lifetime> в секундах.

Обновление словарей (кроме загрузки при первом использовании) не блокирует запросы - во время обновления используется старая версия словаря. Если при обновлении возникнет ошибка, то ошибка пишется в лог сервера, а запросы продолжат использовать старую версию словарей.

Пример настройки:

```
<dictionary>
  ...
  <lifetime>300</lifetime>
  ...
</dictionary>
```

Настройка <lifetime>0</lifetime> запрещает обновление словарей.

Можно задать интервал, внутри которого ClickHouse равномерно-случайно выберет время для обновления. Это необходимо для распределения нагрузки на источник словаря при обновлении на большом количестве серверов.

Пример настройки:

```
<dictionary>
  ...
  <lifetime>
    <min>300</min>
    <max>360</max>
  </lifetime>
  ...
</dictionary>
```

При обновлении словарей сервер ClickHouse применяет различную логику в зависимости от типа источника:

- У текстового файла проверяется время модификации. Если время изменилось по отношению к запомненному ранее, то словарь обновляется.
- Для таблиц типа MyISAM, время модификации проверяется запросом SHOW TABLE STATUS.
- Словари из других источников по умолчанию обновляются каждый раз.

Для источников MySQL (InnoDB), ODBC и ClickHouse можно настроить запрос, который позволит обновлять словари только в случае их фактического изменения, а не каждый раз. Чтобы это сделать необходимо выполнить следующие условия/действия:

- В таблице словаря должно быть поле, которое гарантированно изменяется при обновлении данных в источнике.
- В настройках источника указывается запрос, который получает изменяющееся поле. Результат запроса сервер ClickHouse интерпретирует как строку и если эта строка изменилась по отношению к предыдущему состоянию, то словарь обновляется. Запрос следует указывать в поле <invalidate_query> настроек источника.

Пример настройки:

```
<dictionary>
  ...
  <odbc>
    ...
    <invalidate_query>SELECT update_time FROM dictionary_source where id = 1</invalidate_query>
  </odbc>
  ...
</dictionary>
```

Источники внешних словарей

Внешний словарь можно подключить из множества источников.

Общий вид конфигурации:

```
<yandex>
  <dictionary>
    ...
    <source>
      <source_type>
        <!-- Source configuration -->
      </source_type>
    </source>
    ...
  </dictionary>
  ...
</yandex>
```

Источник настраивается в разделе source.

Типы источников (source_type):

- Локальный файл
- Исполняемый файл
- HTTP(s)
- СУБД:
  - ODBC
  - MySQL
  - ClickHouse
  - MongoDB

Локальный файл

Пример настройки:

```
<source>
  <file>
    <path>/opt/dictionaries/os.tsv</path>
    <format>TabSeparated</format>
  </file>
</source>
```

Поля настройки:

- path - Абсолютный путь к файлу.
- format - Формат файла. Поддерживаются все форматы, описанные в разделе "Форматы".

Исполняемый файл

Работа с исполняемым файлом зависит от размещения словаря в памяти. Если тип размещения словаря cache и complex_key_cache, то ClickHouse запрашивает необходимые ключи, отправляя запрос в STDIN исполняемого файла.

Пример настройки:

```
<source>
  <executable>
    <command>cat /opt/dictionaries/os.tsv</command>
    <format>TabSeparated</format>
  </executable>
</source>
```

Поля настройки:

- command - Абсолютный путь к исполняемому файлу или имя файла (если каталог программы прописан в PATH).
- format - Формат файла. Поддерживаются все форматы, описанные в разделе "Форматы".

HTTP(s)

Работа с HTTP(s) сервером зависит от размещения словаря в памяти. Если тип размещения словаря cache и complex_key_cache, то ClickHouse запрашивает необходимые ключи, отправляя запрос методом POST.

Пример настройки:

```
<source>
  <http>
    <url>http://[::1]/os.tsv</url>
    <format>TabSeparated</format>
  </http>
</source>
```

Чтобы ClickHouse смог обратиться к HTTPS-ресурсу, необходимо настроить openSSL в конфигурации сервера.

Поля настройки:

- url - URL источника.
- format - Формат файла. Поддерживаются все форматы, описанные в разделе "Форматы".

ODBC

Этим способом можно подключить любую базу данных, имеющую ODBC драйвер.

Пример настройки:

```
<odbc>
  <db>DatabaseName</db>
  <table>ShemaName.TableName</table>
  <connection_string>DSN=some_parameters</connection_string>
  <invalidate_query>SQL_QUERY</invalidate_query>
</odbc>
```

Поля настройки:

- db - имя базы данных. Не указывать, если имя базы задано в параметрах. <connection_string>.
- table - имя таблицы и схемы, если она есть.
- connection_string - строка соединения.
- invalidate_query - запрос для проверки статуса словаря. Необязательный параметр. Читайте подробнее в разделе Обновление словарей.

ClickHouse получает от ODBC-драйвера информацию о квотировании и квотирует настройки в запросах к драйверу, поэтому имя таблицы нужно указывать в соответствии с регистром имени таблицы в базе данных.

Выявленная уязвимость в функционировании ODBC словарей

Attention

При соединении с базой данных через ODBC можно заменить параметр соединения Servername. В этом случае, значения USERNAME и PASSWORD из odbc.ini отправляются на удаленный сервер и могут быть скомпроментированы.

Пример небезопасного использования

Сконфигурируем unixODBC для работы с PostgreSQL. Содержимое /etc/odbc.ini:

```
[gregtest]
Driver = /usr/lib/psqlodbca.so
Servername = localhost
PORT = 5432
DATABASE = test_db
##OPTION = 3
USERNAME = test
PASSWORD = test
```

Если выполнить запрос вида:

```
SELECT * FROM odbc('DSN=gregtest;Servername=some-server.com', 'test_db');
```

то ODBC драйвер отправит значения USERNAME и PASSWORD из odbc.ini на some-server.com.

Пример подключения PostgreSQL

ОС Ubuntu.

Установка unixODBC и ODBC-драйвера для PostgreSQL: :

```
sudo apt-get install -y unixodbc odbcinst odbc-postgresql
```

Настройка /etc/odbc.ini (или ~/.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        =
```

Конфигурация словаря в ClickHouse:

```
<yandex>
  <dictionary>
    <name>table_name</name>
    <source>
      <odbc>
        <!-- в connection_string можно указывать следующие параметры: -->
        <!-- 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>
```

Может понадобиться в odbc.ini указать полный путь до библиотеки с драйвером DRIVER=/usr/local/lib/psqlodbcw.so.

Пример подключения MS SQL Server

ОС Ubuntu.

Установка драйвера: :

```
sudo apt-get install tdsodbc freetds-bin sqsh
```

Настройка драйвера: :

```
$ 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
```

Настройка словаря в 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>
```

СУБД

MySQL

Пример настройки:

```
<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>
```

Поля настройки:

- port - порт сервера MySQL. Можно указать для всех реплик или для каждой в отдельности (внутри <replica>).
- user - имя пользователя MySQL. Можно указать для всех реплик или для каждой в отдельности (внутри <replica>).
- password - пароль пользователя MySQL. Можно указать для всех реплик или для каждой в отдельности (внутри <replica>).
- replica - блок конфигурации реплики. Блоков может быть несколько.
    - replica/host - хост MySQL.

  * replica/priority - приоритет реплики. При попытке соединения ClickHouse обходит реплики в соответствии с приоритетом. Чем меньше цифра, тем выше приоритет.
  - db - имя базы данных.
  - table - имя таблицы.
  - where - условие выбора. Необязательный параметр.
  - invalidate_query - запрос для проверки статуса словаря. Необязательный параметр. Читайте подробнее в разделе Обновление словарей.

MySQL можно подключить на локальном хосте через сокеты, для этого необходимо задать host и socket.

Пример настройки:

```
<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

Пример настройки:

```
<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>
```

Поля настройки:

- host - хост ClickHouse. Если host локальный, то запрос выполняется без сетевого взаимодействия. Чтобы повысить отказоустойчивость решения, можно создать таблицу типа Distributed и прописать её в дальнейших настройках.
- port - порт сервера ClickHouse.
- user - имя пользователя ClickHouse.
- password - пароль пользователя ClickHouse.
- db - имя базы данных.
- table - имя таблицы.
- where - условие выбора. Может отсутствовать.
- invalidate_query - запрос для проверки статуса словаря. Необязательный параметр. Читайте подробнее в разделе Обновление словарей.

MongoDB

Пример настройки:

```
<source>
    <mongodb>
        <host>localhost</host>
        <port>27017</port>
        <user></user>
        <password></password>
        <db>test</db>
        <collection>dictionary_source</collection>
    </mongodb>
</source>
```

Поля настройки:

- host - хост MongoDB.
- port - порт сервера MongoDB.
- user - имя пользователя MongoDB.
- password - пароль пользователя MongoDB.
- db - имя базы данных.
- collection - имя коллекции.

Ключ и поля словаря

Секция <structure> описывает ключ словаря и поля, доступные для запросов.

Общий вид структуры:

```
<dictionary>
  <structure>
    <id>
      <name>Id</name>
    </id>

    <attribute>
      <!-- Attribute parameters -->
    </attribute>

    ...

  </structure>
</dictionary>
```

В структуре описываются столбцы:

- <id> - ключевой столбец.
- <attribute> - столбец данных. Столбцов может быть много.

Ключ

ClickHouse поддерживает следующие виды ключей:

- Числовой ключ. Формат UInt64. Описывается в теге <id>.
- Составной ключ. Набор значений разного типа. Описывается в теге <key>.

Структура может содержать либо <id> либо <key>.

Обратите внимание

Ключ не надо дополнительно описывать в атрибутах.

Числовой ключ

Формат: UInt64.

Пример конфигурации:

```
<id>
  <name>Id</name>
</id>
```

Поля конфигурации:

- name - имя столбца с ключами.

Составной ключ

Ключом может быть кортеж (tuple) из полей произвольных типов. layout в этом случае должен быть complex_key_hashed или complex_key_cache.

Совет

Составной ключ может состоять из одного элемента. Это даёт возможность использовать в качестве ключа, например, строку.

Структура ключа задаётся в элементе <key>. Поля ключа задаются в том же формате, что и атрибуты словаря. Пример:

```
<structure>
    <key>
        <attribute>
            <name>field1</name>
            <type>String</type>
        </attribute>
        <attribute>
            <name>field2</name>
            <type>UInt32</type>
        </attribute>
        ...
    </key>
...
```

При запросе в функции dictGet* в качестве ключа передаётся кортеж. Пример: dictGetString('dict_name', 'attr_name', tuple('string for field1', num_for_field2)).

Атрибуты

Пример конфигурации:

```
<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>
```

Поля конфигурации:

- name - Имя столбца.
- type - Тип столбца. Задает способ интерпретации данных в источнике. Например, в случае MySQL, в таблице-источнике поле может быть TEXT, VARCHAR, BLOB, но загружено может быть как String.
- null_value - Значение по умолчанию для несуществующего элемента. В примере - пустая строка.
- expression - Атрибут может быть выражением. Тег не обязательный.
- hierarchical - Поддержка иерархии. Отображение в идентификатор родителя. По умолчанию, false.
- injective - Признак инъективности отображения id -> attribute. Если true, то можно оптимизировать GROUP BY. По умолчанию, false.
- is_object_id - Признак того, что запрос выполняется к документу MongoDB по ObjectID.

Встроенные словари

ClickHouse содержит встроенную возможность работы с геобазой.

Это позволяет:

- для идентификатора региона получить его имя на нужном языке;
- по идентификатору региона получить идентификатор города, области, федерального округа, страны, континента;
- проверить, что один регион входит в другой;
- получить цепочку родительских регионов.

Все функции поддерживают "транслокальность", то есть возможность использовать одновременно разные точки зрения на принадлежность регионов. Подробнее смотрите в разделе "Функции для работы со словарями Яндекс.Метрики".

В пакете по умолчанию, встроенные словари выключены.
Для включения, раскомментируйте параметры path_to_regions_hierarchy_file и path_to_regions_names_files в конфигурационном файле сервера.

Геобаза загружается из текстовых файлов.

Положите файлы regions_hierarchy*.txt в директорию path_to_regions_hierarchy_file. Этот конфигурационный параметр должен содержать путь к файлу regions_hierarchy.txt (иерархия регионов по умолчанию), а другие файлы (regions_hierarchy_ua.txt) должны находиться рядом в той же директории.

Положите файлы regions_names_*.txt в директорию path_to_regions_names_files.

Также вы можете создать эти файлы самостоятельно. Формат файлов такой:

regions_hierarchy*.txt: TabSeparated (без заголовка), столбцы:

- идентификатор региона (UInt32);
- идентификатор родительского региона (UInt32);
- тип региона (UInt8): 1 - континент, 3 - страна, 4 - федеральный округ, 5 - область, 6 - город; остальные типы не имеют значения;
- население (UInt32) - не обязательный столбец.

regions_names_*.txt: TabSeparated (без заголовка), столбцы:

- идентификатор региона (UInt32);
- имя региона (String) - не может содержать табы или переводы строк, даже экранированные.

Для хранения в оперативке используется плоский массив. Поэтому, идентификаторы не должны быть больше миллиона.

Словари могут обновляться без перезапуска сервера. Но набор доступных словарей не обновляется.
Для обновления проверяется время модификации файлов; если файл изменился, то словарь будет обновлён.
Периодичность проверки настраивается конфигурационным параметром builtin_dictionaries_reload_interval.
Обновление словарей (кроме загрузки при первом использовании) не блокирует запросы - во время обновления запросы используют старую версию словарей. Если при обновлении возникнет ошибка, то ошибка пишется в лог сервера, а запросы продолжат использовать старую версию словарей.

Рекомендуется периодически обновлять словари с геобазой. При обновлении, генерируйте новые файлы, записывая их в отдельное место, а только когда всё готово - переименовывайте в файлы, которые использует сервер.

Также имеются функции для работы с идентификаторами операционных систем и поисковых систем Яндекс.Метрики, пользоваться которыми не нужно.

Операторы

Все операторы преобразуются в соответствующие функции на этапе парсинга запроса, с учётом их приоритетов и ассоциативности.
Далее будут перечислены группы операторов в порядке их приоритета (чем выше, тем раньше оператор связывается со своими аргументами).

Операторы доступа

a[N] - доступ к элементу массива, функция arrayElement(a, N).

a.N - доступ к элементу кортежа, функция tupleElement(a, N).

Оператор числового отрицания

-a - функция negate(a).

Операторы умножения и деления

a * b - функция multiply(a, b)

a / b - функция divide(a, b)

a % b - функция modulo(a, b)

Операторы сложения и вычитания

a + b - функция plus(a, b)

a - b - функция minus(a, b)

Операторы сравнения

a = b - функция equals(a, b)

a == b - функция equals(a, b)

a != b - функция notEquals(a, b)

a <> b - функция notEquals(a, b)

a <= b - функция lessOrEquals(a, b)

a >= b - функция greaterOrEquals(a, b)

a < b - функция less(a, b)

a > b - функция greater(a, b)

a LIKE s - функция like(a, b)

a NOT LIKE s - функция notLike(a, b)

a BETWEEN b AND c - равнозначно a >= b AND a <= c

a NOT BETWEEN b AND c - равнозначно a < b OR a > c

Операторы для работы с множествами

Смотрите раздел Операторы IN.

a IN ... - функция in(a, b)

a NOT IN ... - функция notIn(a, b)

a GLOBAL IN ... - функция globalIn(a, b)

a GLOBAL NOT IN ... - функция globalNotIn(a, b)

Оператор для работы с датами и временем

```
EXTRACT(part FROM date);
```

Позволяет извлечь отдельные части из переданной даты. Например, можно получить месяц из даты, или минуты из времени.

В параметре part указывается, какой фрагмент даты нужно получить. Доступные значения:

- DAY — День. Возможные значения: 1–31.
- MONTH — Номер месяца. Возможные значения: 1–12.
- YEAR — Год.
- SECOND — Секунда. Возможные значения: 0–59.
- MINUTE — Минута. Возможные значения: 0–59.

- HOUR — Час. Возможные значения: 0–23.

Эти значения могут быть указаны также в нижнем регистре (day, month).

В параметре date указывается исходная дата. Поддерживаются типы Date и DateTime.

Примеры:

```
SELECT EXTRACT(DAY FROM toDate('2017-06-15'));
SELECT EXTRACT(MONTH FROM toDate('2017-06-15'));
SELECT EXTRACT(YEAR FROM toDate('2017-06-15'));
```

В следующем примере создадим таблицу и добавим в неё значение с типом DateTime.

```
CREATE TABLE test.Orders
(
    OrderId UInt64,
    OrderName String,
    OrderDate DateTime
)
ENGINE = Log;
```

```
INSERT INTO test.Orders VALUES (1, 'Jarlsberg Cheese', toDateTime('2008-10-11 13:23:44'));
```

```
SELECT
    toYear(OrderDate) AS OrderYear,
    toMonth(OrderDate) AS OrderMonth,
    toDayOfMonth(OrderDate) AS OrderDay,
    toHour(OrderDate) AS OrderHour,
    toMinute(OrderDate) AS OrderMinute,
    toSecond(OrderDate) AS OrderSecond
FROM test.Orders;

┌─OrderYear─┬─OrderMonth─┬─OrderDay─┬─OrderHour─┬─OrderMinute─┬─OrderSecond─┐
│     2008 │       10 │      11 │       13 │        23 │        44 │
└──────────┴──────────┴─────────┴──────────┴─────────────┴─────────────┘
```

Больше примеров приведено в тестах.

Оператор логического отрицания

NOT a - функция not(a)

Оператор логического 'И'

a AND b - функция and(a, b)

Оператор логического 'ИЛИ'

a OR b - функция or(a, b)

Условный оператор

a ? b : c - функция if(a, b, c)

Примечание:

Условный оператор сначала вычисляет значения b и c, затем проверяет выполнение условия a, и только после этого возвращает соответствующее значение. Если в качестве b или c выступает функция arrayJoin(), то размножение каждой строки произойдет вне зависимости от условия a.

Условное выражение

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

В случае указания x - функция transform(x, [a, ...], [b, ...], c). Иначе — multilf(a, b, ..., c).
При отсутствии секции ELSE c, значением по умолчанию будет NULL.

Примечание

Функция transform не умеет работать с NULL.

Оператор склеивания строк

s1 || s2 - функция concat(s1, s2)

Оператор создания лямбда-выражения

x -> expr - функция lambda(x, expr)

Следующие операторы не имеют приоритета, так как представляют собой скобки:

Оператор создания массива

[x1, ...] - функция array(x1, ...)

Оператор создания кортежа

(x1, x2, ...) - функция tuple(x2, x2, ...)

Ассоциативность

Все бинарные операторы имеют левую ассоциативность. Например, 1 + 2 + 3 преобразуется в plus(plus(1, 2), 3).
Иногда это работает не так, как ожидается. Например, SELECT 4 > 3 > 2 выдаст 0.

Для эффективности, реализованы функции and и or, принимающие произвольное количество аргументов. Соответствующие цепочки операторов AND и OR, преобразуются в один вызов этих функций.

Проверка на NULL

ClickHouse поддерживает операторы IS NULL и IS NOT NULL.

IS NULL

- Для значений типа Nullable оператор IS NULL возвращает:
  - 1, если значение — NULL.
  - 0 в обратном случае.
- Для прочих значений оператор IS NULL всегда возвращает 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

- Для значений типа Nullable оператор IS NOT NULL возвращает:
  - 0, если значение — NULL.
  - 1, в обратном случае.
- Для прочих значений оператор IS NOT NULL всегда возвращает 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.
```

Синтаксис

В системе есть два вида парсеров: полноценный парсер SQL (recursive descent parser) и парсер форматов данных (быстрый потоковый парсер).
Во всех случаях кроме запроса INSERT, используется только полноценный парсер SQL.
В запросе INSERT используется оба парсера:

```
INSERT INTO t VALUES (1, 'Hello, world'), (2, 'abc'), (3, 'def')
```

Фрагмент INSERT INTO t VALUES парсится полноценным парсером, а данные (1, 'Hello, world'), (2, 'abc'), (3, 'def') - быстрым потоковым парсером.
Данные могут иметь любой формат. При получении запроса, сервер заранее считывает в оперативку не более max_query_size байт запроса (по умолчанию, 1МБ), а всё остальное обрабатывается потоково.
Таким образом, в системе нет проблем с большими INSERT запросами, как в MySQL.

При использовании формата Values в INSERT запросе может сложиться иллюзия, что данные парсятся также, как выражения в запросе SELECT, но это не так. Формат Values гораздо более ограничен.

Далее пойдёт речь о полноценном парсере. О парсерах форматов, смотри раздел "Форматы".

Пробелы

Между синтаксическими конструкциями (в том числе, в начале и конце запроса) может быть расположено произвольное количество пробельных символов. К пробельным символам относятся пробел, таб, перевод строки, CR, form feed.

Комментарии

Поддерживаются комментарии в SQL-стиле и C-стиле.
Комментарии в SQL-стиле: от -- до конца строки. Пробел после -- может не ставиться.
Комментарии в C-стиле: от /* до */. Такие комментарии могут быть многострочными. Пробелы тоже не обязательны.

Ключевые слова

Ключевые слова (например, SELECT) регистронезависимы. Всё остальное (имена столбцов, функций и т. п.), в отличие от стандарта SQL, регистрозависимо.

Ключевые слова не зарезервированы (а всего лишь парсятся как ключевые слова в соответствующем контексте). Если вы используете идентификаторы, совпадающие с ключевыми словами, заключите их в кавычки. Например, запрос SELECT "FROM" FROM table_name валиден, если таблица table_name имеет столбец с именем "FROM".

Идентификаторы

Идентификаторы:

- Имена кластеров, баз данных, таблиц, разделов и столбцов;
- Функции;
- Типы данных;
- Синонимы выражений.

Некоторые идентификаторы нужно указывать в кавычках (например, идентификаторы с пробелами). Прочие идентификаторы можно указывать без кавычек. Рекомендуется использовать идентификаторы, не требующие кавычек.

Идентификаторы не требующие кавычек соответствуют регулярному выражению ^[a-zA-Z_][0-9a-zA-Z_]*$ и не могут совпадать с ключевыми словами. Примеры: x, _1, X_y__Z123_.

Если вы хотите использовать идентификаторы, совпадающие с ключевыми словами, или использовать в идентификаторах символы, не входящие в регулярное выражение, заключите их в двойные или обратные кавычки, например, "id", `id`.

Литералы

Существуют: числовые, строковые, составные литералы и NULL.

Числовые

Числовой литерал пытается распарситься:

- Сначала как знаковое 64-разрядное число, функцией strtoull.
- Если не получилось, то как беззнаковое 64-разрядное число, функцией strtoll.
- Если не получилось, то как число с плавающей запятой, функцией strtod.
- Иначе — ошибка.

Соответствующее значение будет иметь тип минимального размера, который вмещает значение. Например, 1 парсится как UInt8, а 256 как UInt16. Подробнее о типах данных читайте в разделе Типы данных.

Примеры: 1, 18446744073709551615, 0xDEADBEEF, 01, 0.1, 1e100, -1e-100, inf, nan.

Строковые

Поддерживаются только строковые литералы в одинарных кавычках. Символы внутри могут быть экранированы с помощью обратного слеша. Следующие escape-последовательности имеют соответствующее специальное значение: \b, \f, \r, \n, \t, \0, \a, \v, \xHH. Во всех остальных случаях, последовательности вида \c, где c — любой символ, преобразуется в c . Таким образом, могут быть использованы последовательности \' и \\. Значение будет иметь тип String.

Минимальный набор символов, которых вам необходимо экранировать в строковых литералах: ' и \. Одинарная кавычка может быть экранирована одинарной кавычкой, литералы 'It\'s' и 'It''s' эквивалентны.

Составные

Поддерживаются конструкции для массивов: [1, 2, 3] и кортежей: (1, 'Hello, world!', 2).
На самом деле, это вовсе не литералы, а выражение с оператором создания массива и оператором создания кортежа, соответственно.
Массив должен состоять хотя бы из одного элемента, а кортеж - хотя бы из двух.
Кортежи носят служебное значение для использования в секции IN запроса SELECT. Кортежи могут быть получены как результат запроса, но они не могут быть сохранены в базе данных (за исключением таблицы Memory.)

NULL

Обозначает, что значение отсутствует.

Чтобы в поле таблицы можно было хранить NULL, оно должно быть типа Nullable.

В зависимости от формата данных (входных или выходных) NULL может иметь различное представление. Подробнее смотрите в документации для форматов данных.

При обработке NULL есть множество особенностей. Например, если хотя бы один из аргументов операции сравнения — NULL, то результатом такой операции тоже будет NULL. Этим же свойством обладают операции умножения, сложения и пр. Подробнее читайте в документации на каждую операцию.

В запросах можно проверить NULL с помощью операторов IS NULL и IS NOT NULL, а также соответствующих функций isNull и isNotNull.

Функции

Функции записываются как идентификатор со списком аргументов (возможно, пустым) в скобках. В отличие от стандартного SQL, даже в случае пустого списка аргументов, скобки обязательны. Пример: now().
Бывают обычные и агрегатные функции (смотрите раздел "Агрегатные функции"). Некоторые агрегатные функции могут содержать два списка аргументов в круглых скобках. Пример: quantile(0.9)(x). Такие агрегатные функции называются "параметрическими", а первый список аргументов называется "параметрами". Синтаксис агрегатных функций без параметров ничем не отличается от обычных функций.

Операторы

Операторы преобразуются в соответствующие им функции во время парсинга запроса, с учётом их приоритета и ассоциативности.
Например, выражение 1 + 2 * 3 + 4 преобразуется в plus(plus(1, multiply(2, 3)), 4).

Типы данных и движки таблиц

Типы данных и движки таблиц в запросе CREATE записываются также, как идентификаторы или также как функции. То есть, могут содержать или не содержать список аргументов в круглых скобках. Подробнее смотрите разделы "Типы данных", "Движки таблиц", "CREATE".

Синонимы выражений

Синоним — это пользовательское имя выражения в запросе.

```
expr AS alias
```

- AS — ключевое слово для определения синонимов. Можно определить синоним для имени таблицы или столбца в секции SELECT без использования ключевого слова AS .

  Например, SELECT table_name_alias.column_name FROM table_name table_name_alias.

  В функции CAST, ключевое слово AS имеет другое значение. Смотрите описание функции.

- expr — любое выражение, которое поддерживает ClickHouse.

  Например, SELECT column_name * 2 AS double FROM some_table.

- alias — имя для выражения. Синонимы должны соответствовать синтаксису идентификаторов.

  Например, SELECT "table t".column_name FROM table_name AS "table t".

Примечания по использованию

Синонимы являются глобальными для запроса или подзапроса, и вы можете определить синоним в любой части запроса для любого выражения. Например, SELECT (1 AS n) + 2, n.

Синонимы не передаются в подзапросы и между подзапросами. Например, при выполнении запроса SELECT (SELECT sum(b.a) + num FROM b) - a.a AS num FROM a ClickHouse сгенерирует исключение Unknown identifier: num.

Если синоним определен для результирующих столбцов в секции SELECT вложенного запроса, то эти столбцы отображаются во внешнем запросе. Например, SELECT n + m FROM (SELECT 1 AS n, 2 AS m).

Будьте осторожны с синонимами, совпадающими с именами столбцов или таблиц. Рассмотрим следующий пример:

```
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.
```

В этом примере мы объявили таблицу t со столбцом b. Затем, при выборе данных, мы определили синоним sum(b) AS b. Поскольку синонимы глобальные, то ClickHouse заменил литерал b в выражении argMax(a, b) выражением sum(b). Эта замена вызвала исключение.

Звёздочка

В запросе SELECT, вместо выражения может стоять звёздочка. Подробнее смотрите раздел "SELECT".

Выражения

Выражение представляет собой функцию, идентификатор, литерал, применение оператора, выражение в скобках, подзапрос, звёздочку. А также может содержать синоним.
Список выражений - одно выражение или несколько выражений через запятую.
Функции и операторы, в свою очередь, в качестве аргументов, могут иметь произвольные выражения.

Эксплуатация

Руководство по эксплуатации ClickHouse состоит из следующих основных разделов:

- Требования
- Мониторинг
- Решение проблем
- Советы по эксплуатации
- Процедура обновления
- Права доступа
- Резервное копирование
- Конфигурационные файлы
- Квоты
- Системные таблицы
- Конфигурационные параметры сервера
- Настройки
- Утилиты

Требования

Процессор

В случае установки из готовых deb-пакетов используйте процессоры с архитектурой x86_64 и поддержкой инструкций SSE 4.2. Для запуска ClickHouse на процессорах без поддержки SSE 4.2 или на процессорах с архитектурой AArch64 и PowerPC64LE необходимо собирать ClickHouse из исходников.

ClickHouse реализует параллельную обработку данных и использует все доступные аппаратные ресурсы. При выборе процессора учитывайте, что ClickHouse работает более эффективно в конфигурациях с большим количеством ядер, но с более низкой тактовой частотой, чем в конфигурациях с меньшим количеством ядер и более высокой тактовой частотой. Например, 16 ядер с 2600 MHz предпочтительнее, чем 8 ядер с 3600 MHz.

Рекомендуется использовать технологии **Turbo Boost** и **hyper-threading**. Их использование существенно улучшает производительность при типичной нагрузке.

RAM

Мы рекомендуем использовать как минимум 4 ГБ оперативной памяти, чтобы иметь возможность выполнять нетривиальные запросы. Сервер ClickHouse может работать с гораздо меньшим объёмом RAM, память требуется для обработки запросов.

Необходимый объем RAM зависит от:

- Сложности запросов.
- Объёма данных, обрабатываемых в запросах.

Для расчета объема RAM необходимо оценить размер промежуточных данных для операций GROUP BY, DISTINCT, JOIN а также других операций, которыми вы пользуетесь.

ClickHouse может использовать внешнюю память для промежуточных данных. Подробнее смотрите в разделе GROUP BY во внешней памяти.

Файл подкачки

Отключайте файл подкачки в продуктовых средах.

Подсистема хранения

Для установки ClickHouse необходимо 2ГБ свободного места на диске.

Объём дискового пространства, необходимый для хранения ваших данных, необходимо рассчитывать отдельно. Расчёт должен включать:

- Приблизительную оценку объёма данных.

  Можно взять образец данных и получить из него средний размер строки. Затем умножьте полученное значение на количество строк, которое вы планируете хранить.

- Оценку коэффициента сжатия данных.

  Чтобы оценить коэффициент сжатия данных, загрузите некоторую выборку данных в ClickHouse и сравните действительный размер данных с размером сохранённой таблицы. Например, данные типа clickstream обычно сжимаются в 6-10 раз.

Для оценки объёма хранилища, примените коэффициент сжатия к размеру данных. Если вы планируете хранить данные в нескольких репликах, то необходимо полученный объём умножить на количество реплик.

Сеть

По возможности, используйте сети 10G и более высокого класса.

Пропускная способность сети критически важна для обработки распределенных запросов с большим количеством промежуточных данных. Также, скорость сети влияет на задержки в процессах репликации.

Программное обеспечение

ClickHouse разработан для семейства операционных систем Linux. Рекомендуемый дистрибутив Linux — Ubuntu. В системе должен быть установлен пакет tzdata.

ClickHouse может работать и в других семействах операционных систем. Подробнее смотрите разделе документации Начало работы.

Мониторинг

Вы можете отслеживать:

- Использование аппаратных ресурсов.
- Метрики сервера ClickHouse.

Использование ресурсов

ClickHouse не отслеживает состояние аппаратных ресурсов самостоятельно.

Рекомендуем контролировать:

- Загрузку и температуру процессоров.

  Можно использовать dmesg, turbostat или другие инструменты.

- Использование системы хранения, оперативной памяти и сети.

Метрики сервера ClickHouse.

Сервер ClickHouse имеет встроенные инструменты мониторинга.

Для отслеживания событий на сервере используйте логи. Подробнее смотрите в разделе конфигурационного файла logger.

ClickHouse собирает:

- Различные метрики того, как сервер использует вычислительные ресурсы.
- Общую статистику обработки запросов.

Метрики находятся в таблицах system.metrics, system.events и system.asynchronous_metrics.

Можно настроить экспорт метрик из ClickHouse в Graphite. Смотрите секцию graphite конфигурационного файла ClickHouse. Перед настройкой экспорта метрик необходимо настроить Graphite, как указано в официальном руководстве.

Также, можно отслеживать доступность сервера через HTTP API. Отправьте HTTP GET к ресурсу /. Если сервер доступен, он отвечает 200 OK.

Для мониторинга серверов в кластерной конфигурации необходимо установить параметр max_replica_delay_for_distributed_queries и использовать HTTP ресурс /replicas-delay. Если реплика доступна и не отстаёт от других реплик, то запрос к /replicas-delay возвращает 200 OK. Если реплика отстаёт, то она возвращает информацию о размере отставания.

Устранение неисправностей

- Установка дистрибутива
- Соединение с сервером
- Обработка запросов
- Скорость обработки запросов

Установка дистрибутива

Не получается скачать deb-пакеты из репозитория ClickHouse с помощью apt-get

- Проверьте настройки брандмауэра.

- Если по какой-либо причине вы не можете получить доступ к репозиторию, скачайте пакеты как описано в разделе Начало работы и установите их вручную командой sudo dpkg -i <packages>. Также, необходим пакет tzdata.

Соединение с сервером

Возможные проблемы:

- Сервер не запущен.
- Неожиданные или неправильные параметры конфигурации.

Сервер не запущен

Проверьте, запущен ли сервер

Команда:

```
sudo service clickhouse-server status
```

Если сервер не запущен, запустите его с помощью команды:

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

Проверьте журналы

Основной лог clickhouse-server по умолчанию — /var/log/clickhouse-server/clickhouse-server.log.

В случае успешного запуска вы должны увидеть строки, содержащие:

- <Information> Application: starting up. — сервер запускается.
- <Information> Application: Ready for connections. — сервер запущен и готов принимать соединения.

Если clickhouse-server не запустился из-за ошибки конфигурации вы увидите <Error> строку с описанием ошибки. Например:

```
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
```

Если вы не видите ошибки в конце файла, просмотрите весь файл начиная со строки:

```
<Information> Application: starting up.
```

При попытке запустить второй экземпляр clickhouse-server журнал выглядит следующим образом:

```
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
```

Проверьте логи system.d

Если из логов clickhouse-server вы не получили необходимой информации или логов нет, то вы можете посмотреть логи system.d командой:

```
sudo journalctl -u clickhouse-server
```

Запустите clickhouse-server в интерактивном режиме

```
sudo -u clickhouse /usr/bin/clickhouse-server --config-file /etc/clickhouse-server/config.xml
```

Эта команда запускает сервер как интерактивное приложение со стандартными параметрами скрипта автозапуска. В этом режиме clickhouse-server выводит сообщения в консоль.

Параметры конфигурации

Проверьте:

- Настройки Docker.

  При запуске ClickHouse в Docker в сети IPv6 убедитесь, что установлено network=host.

- Параметры endpoint.

  Проверьте настройки listen_host и tcp_port.

  По умолчанию, сервер ClickHouse принимает только локальные подключения.

- Настройки протокола HTTP.

  Проверьте настройки протокола для HTTP API.

- Параметры безопасного подключения.

  Проверьте:

  - Настройку tcp_port_secure.
  - Параметры для SSL-сертификатов.

  Используйте правильные параметры при подключении. Например, используйте параметр port_secure при использовании clickhouse_client.

- Настройки пользователей.

  Возможно, вы используете неверное имя пользователя или пароль.

Обработка запросов

Если ClickHouse не может обработать запрос, он отправляет клиенту описание ошибки. В clickhouse-client вы получаете описание ошибки в консоли. При использовании интерфейса HTTP, ClickHouse отправляет описание ошибки в теле ответа. Например:

```
$ 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
```

Если вы запускаете clickhouse-client с параметром stack-trace, то ClickHouse возвращает описание ошибки и соответствующий стек вызовов функций на сервере.

Может появиться сообщение о разрыве соединения. В этом случае необходимо повторить запрос. Если соединение прерывается каждый раз при выполнении запроса, следует проверить журналы сервера на наличие ошибок.

Скорость обработки запросов

Если вы видите, что ClickHouse работает слишком медленно, необходимо профилировать загрузку ресурсов сервера и сети для ваших запросов.

Для профилирования запросов можно использовать утилиту clickhouse-benchmark. Она показывает количество запросов, обработанных за секунду, количество строк, обработанных за секунду и перцентили времени обработки запросов.

Советы по эксплуатации

CPU scaling governor

Всегда используйте performance scaling governor. ondemand scaling governor работает намного хуже при постоянно высоком спросе.

```
echo 'performance' | sudo tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
```

Ограничение CPU

Процессоры могут перегреваться. С помощью dmesg можно увидеть, если тактовая частота процессора была ограничена из-за перегрева.
Также ограничение может устанавливаться снаружи на уровне датацентра. С помощью turbostat можно за этим наблюдать под нагрузкой.

Оперативная память

Для небольших объемов данных (до ~200 Гб в сжатом виде) лучше всего использовать столько памяти не меньше, чем объем данных.
Для больших объемов данных, при выполнении интерактивных (онлайн) запросов, стоит использовать разумный объем оперативной памяти (128 Гб или более) для того, чтобы горячее подмножество данных поместилось в кеше страниц.
Даже для объемов данных в ~50 Тб на сервер, использование 128 Гб оперативной памяти намного лучше для производительности выполнения запросов, чем 64 Гб.

Не выключайте overcommit. Значение cat /proc/sys/vm/overcommit_memory должно быть 0 or 1. Выполните:

```
echo 0 | sudo tee /proc/sys/vm/overcommit_memory
```

Huge pages

Механизм прозрачных huge pages нужно отключить. Он мешает работе аллокаторов памяти, что приводит к значительной деградации производительности.

```
echo 'never' | sudo tee /sys/kernel/mm/transparent_hugepage/enabled
```

С помощью perf top можно наблюдать за временем, проведенном в ядре операционной системы для управления памятью.
Постоянные huge pages так же не нужно аллоцировать.

Подсистема хранения

Если ваш бюджет позволяет использовать SSD, используйте SSD.
В противном случае используйте HDD. SATA HDDs 7200 RPM подойдут.

Предпочитайте много серверов с локальными жесткими дисками вместо меньшего числа серверов с подключенными дисковыми полками.
Но для хранения архивов с редкими запросами полки всё же подходят.

RAID

При использовании HDD можно объединить их RAID-10, RAID-5, RAID-6 или RAID-50.
Лучше использовать программный RAID в Linux (mdadm). Лучше не использовать LVM.
При создании RAID-10, нужно выбрать far расположение.
Если бюджет позволяет, лучше выбрать RAID-10.

На более чем 4 дисках вместо RAID-5 нужно использовать RAID-6 (предпочтительнее) или RAID-50.
При использовании RAID-5, RAID-6 или RAID-50, нужно всегда увеличивать stripe_cache_size, так как значение по умолчанию выбрано не самым удачным образом.

```
echo 4096 | sudo tee /sys/block/md2/md/stripe_cache_size
```

Точное число стоит вычислять из числа устройств и размер блока по формуле: `2 * num_devices * chunk_size_in_bytes / 4096`.

Размер блока в 1024 Кб подходит для всех конфигураций RAID.
Никогда не указывайте слишком маленький или слишком большой размер блока.

На SSD можно использовать RAID-0.
Вне зависимости от использования RAID, всегда используйте репликацию для безопасности данных.

Включите NCQ с длинной очередью. Для HDD стоит выбрать планировщик CFQ, а для SSD — noop. Не стоит уменьшать настройку readahead.
На HDD стоит включать кеш записи.

Файловая система

Ext4 самый проверенный вариант. Укажите опции монтирования `noatime,nobarrier`.
XFS также подходит, но не так тщательно протестирована в сочетании с ClickHouse.
Большинство других файловых систем также должны нормально работать. Файловые системы с отложенной аллокацией работают лучше.

Ядро Linux

Не используйте слишком старое ядро Linux.

Сеть

При использовании IPv6, стоит увеличить размер кеша маршрутов.
Ядра Linux до 3.2 имели массу проблем в реализации IPv6.

Предпочитайте как минимум 10 Гбит сеть. 1 Гбит также будет работать, но намного хуже для починки реплик с десятками терабайт данных или для обработки распределенных запросов с большим объемом промежуточных данных.

ZooKeeper

Вероятно вы уже используете ZooKeeper для других целей. Можно использовать ту же инсталляцию ZooKeeper, если она не сильно перегружена.

Лучше использовать свежую версию ZooKeeper, как минимум 3.4.9. Версия в стабильных дистрибутивах Linux может быть устаревшей.

Никогда не используете написанные вручную скрипты для переноса данных между разными ZooKeeper кластерами, потому что результат будет некорректный для sequential нод. Никогда не используйте утилиту "zkcopy", по той же причине: https://github.com/ksprojects/zkcopy/issues/15

Если вы хотите разделить существующий ZooKeeper кластер на два, правильный способ - увеличить количество его реплик, а затем переконфигурировать его как два независимых кластера.

Не запускайте ZooKeeper на тех же серверах, что и ClickHouse. Потому что ZooKeeper очень чувствителен к задержкам, а ClickHouse может использовать все доступные системные ресурсы.

С настройками по умолчанию, ZooKeeper является бомбой замедленного действия:

> Сервер ZooKeeper не будет удалять файлы со старыми снепшоты и логами при использовании конфигурации по умолчанию (см. autopurge), это является ответственностью оператора.

Эту бомбу нужно обезвредить.

Далее описана конфигурация ZooKeeper (3.5.1), используемая в боевом окружении Яндекс.Метрики на момент 20 мая 2017 года:

zoo.cfg:

```
## http://hadoop.apache.org/zookeeper/docs/current/zookeeperAdmin.html

## The number of milliseconds of each tick
tickTime=2000
## The number of ticks that the initial
## synchronization phase can take
initLimit=30000
## The number of ticks that can pass between
## sending a request and getting an acknowledgement
syncLimit=10

maxClientCnxns=2000

maxSessionTimeout=60000000
## the directory where the snapshot is stored.
dataDir=/opt/zookeeper/{{ cluster['name'] }}/data
## Place the dataLogDir to a separate physical disc for better performance
dataLogDir=/opt/zookeeper/{{ cluster['name'] }}/logs

autopurge.snapRetainCount=10
autopurge.purgeInterval=1



## To avoid seeks ZooKeeper allocates space in the transaction log file in
## blocks of preAllocSize kilobytes. The default block size is 64M. One reason
## for changing the size of the blocks is to reduce the block size if snapshots
## are taken more often. (Also, see snapCount).
preAllocSize=131072

## Clients can submit requests faster than ZooKeeper can process them,
## especially if there are a lot of clients. To prevent ZooKeeper from running
## out of memory due to queued requests, ZooKeeper will throttle clients so that
## there is no more than globalOutstandingLimit outstanding requests in the
## system. The default limit is 1,000.ZooKeeper logs transactions to a
## transaction log. After snapCount transactions are written to a log file a
## snapshot is started and a new transaction log file is started. The default
## snapCount is 10,000.
snapCount=3000000

## If this option is defined, requests will be will logged to a trace file named
## traceFile.year.month.day.
##traceFile=

## Leader accepts client connections. Default value is "yes". The leader machine
## coordinates updates. For higher update throughput at thes slight expense of
## read throughput the leader can be configured to not accept clients and focus
## on coordination.
leaderServes=yes

standaloneEnabled=false
dynamicConfigFile=/etc/zookeeper-{{ cluster['name'] }}/conf/zoo.cfg.dynamic
```

Версия Java:

```
Java(TM) SE Runtime Environment (build 1.8.0_25-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
```

Параметры JVM:

```
NAME=zookeeper-{{ cluster['name'] }}
ZOOCFGDIR=/etc/$NAME/conf

## TODO this is really ugly
## How to find out, which jars are needed?
## seems, that log4j requires the log4j.properties file to be in the classpath
CLASSPATH="$ZOOCFGDIR:/usr/build/classes:/usr/build/lib/*.jar:/usr/share/zookeeper/zookeeper-3.5.1-
metrika.jar:/usr/share/zookeeper/slf4j-log4j12-1.7.5.jar:/usr/share/zookeeper/slf4j-api-1.7.5.jar:/usr/share/zookeeper/servlet-api-
2.5-20081211.jar:/usr/share/zookeeper/netty-3.7.0.Final.jar:/usr/share/zookeeper/log4j-1.2.16.jar:/usr/share/zookeeper/jline-
2.11.jar:/usr/share/zookeeper/jetty-util-6.1.26.jar:/usr/share/zookeeper/jetty-
6.1.26.jar:/usr/share/zookeeper/javacc.jar:/usr/share/zookeeper/jackson-mapper-asl-1.9.11.jar:/usr/share/zookeeper/jackson-core-
asl-1.9.11.jar:/usr/share/zookeeper/commons-cli-1.2.jar:/usr/src/java/lib/*.jar:/usr/etc/zookeeper"

ZOOCFG="$ZOOCFGDIR/zoo.cfg"
ZOO_LOG_DIR=/var/log/$NAME
USER=zookeeper
GROUP=zookeeper
PIDDIR=/var/run/$NAME
PIDFILE=$PIDDIR/$NAME.pid
SCRIPTNAME=/etc/init.d/$NAME
JAVA=/usr/bin/java
ZOOMAIN="org.apache.zookeeper.server.quorum.QuorumPeerMain"
ZOO_LOG4J_PROP="INFO,ROLLINGFILE"
JMXLOCALONLY=false
JAVA_OPTS="-Xms{{ cluster.get('xms','128M') }} \
    -Xmx{{ cluster.get('xmx','1G') }} \
    -Xloggc:/var/log/$NAME/zookeeper-gc.log \
    -XX:+UseGCLogFileRotation \
    -XX:NumberOfGCLogFiles=16 \
    -XX:GCLogFileSize=16M \
    -verbose:gc \
    -XX:+PrintGCTimeStamps \
    -XX:+PrintGCDateStamps \
    -XX:+PrintGCDetails
    -XX:+PrintTenuringDistribution \
    -XX:+PrintGCApplicationStoppedTime \
    -XX:+PrintGCApplicationConcurrentTime \
    -XX:+PrintSafepointStatistics \
    -XX:+UseParNewGC \
    -XX:+UseConcMarkSweepGC \
-XX:+CMSParallelRemarkEnabled"
```

Salt init:

```
description "zookeeper-{{ cluster['name'] }} centralized coordination service"

start on runlevel [2345]
stop on runlevel [!2345]

respawn

limit nofile 8192 8192

pre-start script
    [ -r "/etc/zookeeper-{{ cluster['name'] }}/conf/environment" ] || exit 0
    . /etc/zookeeper-{{ cluster['name'] }}/conf/environment
    [ -d $ZOO_LOG_DIR ] || mkdir -p $ZOO_LOG_DIR
    chown $USER:$GROUP $ZOO_LOG_DIR
end script

script
    . /etc/zookeeper-{{ cluster['name'] }}/conf/environment
    [ -r /etc/default/zookeeper ] && . /etc/default/zookeeper
    if [ -z "$JMXDISABLE" ]; then
        JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote -
Dcom.sun.management.jmxremote.local.only=$JMXLOCALONLY"
    fi
    exec start-stop-daemon --start -c $USER --exec $JAVA --name zookeeper-{{ cluster['name'] }} \
        -- -cp $CLASSPATH $JAVA_OPTS -Dzookeeper.log.dir=${ZOO_LOG_DIR} \
        -Dzookeeper.root.logger=${ZOO_LOG4J_PROP} $ZOOMAIN $ZOOCFG
end script
```

Обновление ClickHouse

Если ClickHouse установлен с помощью deb-пакетов, выполните следующие команды на сервере:

```
sudo apt-get update
sudo apt-get install clickhouse-client clickhouse-server
sudo service clickhouse-server restart
```

Если ClickHouse установлен не из рекомендуемых deb-пакетов, используйте соответствующий метод обновления.

ClickHouse не поддерживает распределенное обновление. Операция должна выполняться последовательно на каждом отдельном сервере. Не обновляйте все серверы в кластере одновременно, иначе кластер становится недоступным в течение некоторого времени.

Права доступа

Пользователи и права доступа настраиваются в конфиге пользователей. Обычно это users.xml.

Пользователи прописаны в секции users. Рассмотрим фрагмент файла users.xml:

```
<!-- Пользователи и ACL. -->
<users>
   <!-- Если имя пользователя не указано, используется пользователь default. -->
   <default>
      <!-- Password could be specified in plaintext or in SHA256 (in hex format).

           If you want to specify password in plaintext (not recommended), place it in 'password' element.
           Example: <password>qwerty</password>.
           Password could be empty.

           If you want to specify SHA256, place it in 'password_sha256_hex' element.
           Example:
<password_sha256_hex>65e84be33532fb784c48129675f9eff3a682b27168c0ea744b2cf58ee02337c5</password_sha256_hex>

           How to generate decent password:
           Execute: PASSWORD=$(base64 < /dev/urandom | head -c8); echo "$PASSWORD"; echo -n "$PASSWORD" | sha256sum | tr
d '-'
           In first line will be password and in second - corresponding SHA256.
      -->
      <password></password>

      <!-- Список сетей, из которых разрешён доступ.
           Каждый элемент списка имеет одну из следующих форм:
           <ip> IP-адрес или маска подсети. Например, 198.51.100.0/24 или 2001:DB8::/32.
           <host> Имя хоста. Например: example01. Для проверки делается DNS-запрос, и все полученные адреса
сравниваются с адресом клиента.
           <host_regexp> Регулярное выражение для имён хостов. Например, ^example\d\d-\d\d-\d\.yandex\.ru$
              Для проверки, для адреса клиента делается DNS PTR-запрос и к результату применяется регулярное выражение.
              Потом для результата PTR-запроса делается снова DNS-запрос, и все полученные адреса сравниваются с адресом
клиента.
              Настоятельно рекомендуется, чтобы регулярное выражение заканчивалось на \.yandex\.ru$.

           Если вы устанавливаете ClickHouse самостоятельно, укажите здесь:
              <networks>
                    <ip>::/0</ip>
              </networks>
      -->
      <networks incl="networks" />

      <!-- Профиль настроек, использующийся для пользователя. -->
      <profile>default</profile>

      <!-- Квота, использующаяся для пользователя. -->
      <quota>default</quota>
   </default>

   <!-- Для запросов из пользовательского интерфейса Метрики через API для данных по отдельным счётчикам. -->
   <web>
      <password></password>
      <networks incl="networks" />
      <profile>web</profile>
      <quota>default</quota>
      <allow_databases>
        <database>test</database>
      </allow_databases>
   </web>
```

Здесь видно объявление двух пользователей - default и web. Пользователя web мы добавили самостоятельно.

Пользователь default выбирается в случаях, когда имя пользователя не передаётся. Также пользователь default может использоваться при распределённой обработке запроса - если в конфигурации кластера для сервера не указаны user и password. (см. раздел о движке Distributed).

Пользователь, который используется для обмена информацией между серверами, объединенными в кластер, не должен иметь существенных ограничений или квот - иначе распределённые запросы сломаются.

Пароль указывается либо в открытом виде (не рекомендуется), либо в виде SHA-256. Хэш не содержит соль. В связи с этим, не следует рассматривать такие пароли, как защиту от потенциального злоумышленника. Скорее, они нужны для защиты от сотрудников.

Указывается список сетей, из которых разрешён доступ. В этом примере, список сетей для обеих пользователей, загружается из отдельного файла (/etc/metrika.xml), содержащего подстановку networks. Вот его фрагмент:

```
<yandex>
    ...
    <networks>
        <ip>::/64</ip>
        <ip>203.0.113.0/24</ip>
        <ip>2001:DB8::/32</ip>
        ...
    </networks>
</yandex>
```

Можно было бы указать этот список сетей непосредственно в users.xml, или в файле в директории users.d (подробнее смотрите раздел "Конфигурационные файлы").

В конфиге приведён комментарий, указывающий, как можно открыть доступ отовсюду.

Для продакшен использования, указывайте только элементы вида ip (IP-адреса и их маски), так как использование host и host_regexp может вызывать лишние задержки.

Далее указывается используемый профиль настроек пользователя (смотрите раздел "Профили настроек"). Вы можете указать профиль по умолчанию - default. Профиль может называться как угодно; один и тот же профиль может быть указан для разных пользователей. Наиболее важная вещь, которую вы можете прописать в профиле настроек readonly=1, что обеспечивает доступ только на чтение.
Затем указывается используемая квота (смотрите раздел "Квоты"). Вы можете указать квоту по умолчанию — default. Она настроена в конфиге по умолчанию так, что только считает использование ресурсов, но никак их не ограничивает. Квота может называться как угодно. Одна и та же квота может быть указана для разных пользователей, в этом случае подсчёт использования ресурсов делается для каждого пользователя по отдельности.

Также, в необязательном разделе <allow_databases> можно указать перечень баз, к которым у пользователя будет доступ. По умолчанию пользователю доступны все базы. Можно указать базу данных default, в этом случае пользователь получит доступ к базе данных по умолчанию.

Доступ к БД system всегда считается разрешённым (так как эта БД используется для выполнения запросов).

Пользователь может получить список всех БД и таблиц в них с помощью запросов SHOW или системных таблиц, даже если у него нет доступа к отдельным БД.

Доступ к БД не связан с настройкой readonly. Невозможно дать полный доступ к одной БД и readonly к другой.

Резервное копирование данных

Репликация обеспечивает защиту от аппаратных сбоев, но не защищает от человеческих ошибок: случайного удаления данных, удаления не той таблицы, которую надо было, или таблицы на не том кластере, а также программных ошибок, которые приводят к неправильной обработке данных или их повреждению. Во многих случаях подобные ошибки влияют на все реплики. ClickHouse имеет встроенные средства защиты для предотвращения некоторых типов ошибок — например, по умолчанию не получится удалить таблицы *MergeTree, содержащие более 50 Гб данных, одной командой. Однако эти средства защиты не охватывают все возможные случаи и могут быть обойдены.

Для того чтобы эффективно уменьшить возможные человеческие ошибки, следует тщательно подготовить стратегию резервного копирования и восстановления данных **заранее**.

Каждая компания имеет различные доступные ресурсы и бизнес-требования, поэтому нет универсального решения для резервного копирования и восстановления ClickHouse, которое будет подходить в каждой ситуации. То, что работает для одного гигабайта данных, скорее всего, не будет работать для десятков петабайт. Существует множество возможных подходов со своими плюсами и минусами, которые будут рассмотрены ниже. Рекомендуется использовать несколько подходов вместо одного, чтобы компенсировать их различные недостатки.

Примечание

Имейте в виду, что если вы создали резервную копию чего-то и никогда не пытались восстановить её, скорее всего, восстановление не будет работать должным образом, когда вам это действительно понадобится (или, по крайней мере, это займет больше времени, чем будет приемлемо для бизнеса). Поэтому, какой бы подход к резервному копированию вы ни выбрали, обязательно автоматизируйте процесс восстановления и регулярно запускайте его на резервном кластере ClickHouse.

Дублирование данных

Часто данные, которые поступают в ClickHouse, доставляются через некоторую отказоустойчивую очередь, например Apache Kafka. В этом случае можно настроить дополнительный набор подписчиков, которые будут считывать один и тот же поток данных во время записи в ClickHouse и хранить его в холодном хранилище. Большинство компаний уже имеют некоторые рекомендуемые по умолчанию холодные хранилища, которые могут быть хранилищем объектов или распределенной файловой системой, например HDFS.

Снимки файловой системы

Некоторые локальные файловые системы позволяют делать снимки (например, ZFS), но они могут быть не лучшим выбором для обслуживания живых запросов. Возможным решением является создание дополнительных реплик с такой файловой системой и исключение их из Distributed таблиц, используемых для запросов SELECT. Снимки на таких репликах будут недоступны для запросов, изменяющих данные. В качестве бонуса, эти реплики могут иметь особые конфигурации оборудования с большим количеством дисков, подключенных к серверу, что будет экономически эффективным.

clickhouse-copier

clickhouse-copier — это универсальный инструмент, который изначально был создан для перешардирования таблиц с петабайтами данных. Его также можно использовать для резервного копирования и восстановления, поскольку он надёжно копирует данные между таблицами и кластерами ClickHouse.

Для небольших объемов данных можно применять INSERT INTO ... SELECT ... в удалённые таблицы.

Манипуляции с партициями

ClickHouse позволяет использовать запрос ALTER TABLE ... FREEZE PARTITION ... для создания локальной копии партиций таблицы. Это реализуется с помощью жестких ссылок (hardlinks) на каталог /var/lib/clickhouse/shadow/, поэтому такая копия обычно не занимает дополнительное место на диске для старых данных. Созданные копии файлов не обрабатываются сервером ClickHouse, поэтому вы можете просто оставить их там: у вас будет простая резервная копия, которая не требует дополнительной внешней системы, однако при аппаратных проблемах вы можете утратить и актуальные данные и сохраненную копию. По этой причине, лучше удаленно скопировать их в другое место, а затем удалить локальную копию. Распределенные файловые системы и хранилища объектов по-прежнему являются хорошими вариантами для этого, однако можно использовать и обычные присоединенные файловые серверы с достаточно большой емкостью (в этом случае передача будет происходить через сетевую файловую систему или, возможно, rsync).

Дополнительные сведения о запросах, связанных с манипуляциями партициями, см. в разделе ALTER.

Для автоматизации этого подхода доступен инструмент от сторонних разработчиков: clickhouse-backup.

Конфигурационные файлы

Основной конфигурационный файл сервера - config.xml. Он расположен в директории /etc/clickhouse-server/.

Отдельные настройки могут быть переопределены в файлах *.xml и *.conf из директории config.d рядом с конфигом.

У элементов этих конфигурационных файлов могут быть указаны атрибуты replace или remove.

Если ни один не указан - объединить содержимое элементов рекурсивно с заменой значений совпадающих детей.

Если указано replace - заменить весь элемент на указанный.

Если указано remove - удалить элемент.

Также в конфиге могут быть указаны "подстановки". Если у элемента присутствует атрибут incl, то в качестве значения будет использована соответствующая подстановка из файла. По умолчанию, путь к файлу с подстановками - /etc/metrika.xml. Он может быть изменён в конфигурации сервера в элементе include_from. Значения подстановок указываются в элементах /yandex/имя_подстановки этого файла. Если подстановка, заданная в incl отсутствует, то в лог попадает соответствующая запись. Чтобы ClickHouse не писал в лог об отсутствии подстановки, необходимо указать атрибут optional="true" (например, настройка macros).

Подстановки могут также выполняться из ZooKeeper. Для этого укажите у элемента атрибут from_zk = "/path/to/node". Значение элемента заменится на содержимое узла /path/to/node в ZooKeeper. В ZooKeeper-узел также можно положить целое XML-поддерево, оно будет целиком вставлено в исходный элемент.

В config.xml может быть указан отдельный конфиг с настройками пользователей, профилей и квот. Относительный путь к нему указывается в элементе users_config. По умолчанию - users.xml. Если users_config не указан, то настройки пользователей, профилей и квот, указываются непосредственно в config.xml.

Для users_config могут также существовать переопределения в файлах из директории users_config.d (например, users.d) и подстановки. Например, можно иметь по отдельному конфигурационному файлу для каждого пользователя:

```
$ 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>
```

Для каждого конфигурационного файла, сервер при запуске генерирует также файлы file-preprocessed.xml. Эти файлы содержат все выполненные подстановки и переопределения, и предназначены для информационных целей. Если в конфигурационных файлах были использованы ZooKeeper-подстановки, но при старте сервера ZooKeeper недоступен, то сервер загрузит конфигурацию из preprocessed-файла.

Сервер следит за изменениями конфигурационных файлов, а также файлов и ZooKeeper-узлов, которые были использованы при выполнении подстановок и переопределений, и перезагружает настройки пользователей и кластеров на лету. То есть, можно изменять кластера, пользователей и их настройки без перезапуска сервера.

Квоты

Квоты позволяют ограничить использование ресурсов за некоторый интервал времени, или просто подсчитывать использование ресурсов.
Квоты настраиваются в конфиге пользователей. Обычно это users.xml.

В системе есть возможность ограничить сложность одного запроса. Для этого смотрите раздел "Ограничения на сложность запроса".

В отличие от них, квоты:

- ограничивают не один запрос, а множество запросов, которые могут быть выполнены за интервал времени;
- при распределённой обработке запроса, учитывают ресурсы, потраченные на всех удалённых серверах.

Рассмотрим фрагмент файла users.xml, описывающего квоты.

```
<!-- Квоты. -->
<quotas>
  <!-- Имя квоты. -->
  <default>
    <!-- Ограничения за интервал времени. Можно задать много интервалов с разными ограничениями. -->
    <interval>
      <!-- Длина интервала. -->
      <duration>3600</duration>

      <!-- Без ограничений. Просто считать соответствующие данные за указанный интервал. -->
      <queries>0</queries>
      <errors>0</errors>
      <result_rows>0</result_rows>
      <read_rows>0</read_rows>
      <execution_time>0</execution_time>
    </interval>
  </default>
```

Видно, что квота по умолчанию просто считает использование ресурсов за каждый час, но не ограничивает их.
Подсчитанное использование ресурсов за каждый интервал, выводится в лог сервера после каждого запроса.

```
<statbox>
   <!-- Ограничения за интервал времени. Можно задать много интервалов с разными ограничениями. -->
   <interval>
      <!-- Длина интервала. -->
      <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>
```

Для квоты с именем statbox заданы ограничения за каждый час и за каждые 24 часа (86 400 секунд). Интервал времени считается начиная от некоторого implementation defined фиксированного момента времени. То есть, интервал длины 24 часа начинается не обязательно в полночь.

Когда интервал заканчивается, все накопленные значения сбрасываются. То есть, в следующий час, расчёт квоты за час, начинается заново.

Рассмотрим величины, которые можно ограничить:

queries - общее количество запросов;

errors - количество запросов, при выполнении которых было выкинуто исключение;

result_rows - суммарное количество строк, отданных в виде результата;

read_rows - суммарное количество исходных строк, прочитанных из таблиц, для выполнения запроса, на всех удалённых серверах;

execution_time - суммарное время выполнения запросов, в секундах (wall time);

Если за хотя бы один интервал, ограничение превышено, то кидается исключение с текстом о том, какая величина превышена, за какой интервал, и когда начнётся новый интервал (когда снова можно будет задавать запросы).

Для квоты может быть включена возможность указывать "ключ квоты", чтобы производить учёт ресурсов для многих ключей независимо. Рассмотрим это на примере:

```
<!-- Для глобального конструктора отчётов. -->
<web_global>
    <!-- keyed - значит в параметре запроса передаётся "ключ" quota_key,
            и квота считается по отдельности для каждого значения ключа.
        Например, в качестве ключа может передаваться логин пользователя в Метрике,
            и тогда квота будет считаться для каждого логина по отдельности.
        Имеет смысл использовать только если quota_key передаётся не пользователем, а программой.

        Также можно написать <keyed_by_ip /> - тогда в качестве ключа квоты используется IP-адрес.
        (но стоит учесть, что пользователь может достаточно легко менять IPv6-адрес)
    -->
    <keyed />
```

Квота прописывается для пользователей в секции users конфига. Смотрите раздел "Права доступа".

При распределённой обработке запроса, накопленные величины хранятся на сервере-инициаторе запроса. То есть, если пользователь пойдёт на другой сервер - там квота будет действовать "с нуля".

При перезапуске сервера, квоты сбрасываются.

Системные таблицы

Системные таблицы используются для реализации части функциональности системы, а также предоставляют доступ к информации о работе системы.
Вы не можете удалить системную таблицу (хотя можете сделать DETACH).
Для системных таблиц нет файлов с данными на диске и файлов с метаданными. Сервер создаёт все системные таблицы при старте.
В системные таблицы нельзя записывать данные - можно только читать.
Системные таблицы расположены в базе данных system.

system.asynchronous_metrics

Содержат метрики, используемые для профилирования и мониторинга.
Обычно отражают количество событий, происходящих в данный момент в системе, или ресурсов, суммарно потребляемых системой.
Пример: количество запросов типа SELECT, исполняемых в текущий момент; количество потребляемой памяти.
system.asynchronous_metrics и system.metrics отличаются набором и способом вычисления метрик.

system.clusters

Содержит информацию о доступных в конфигурационном файле кластерах и серверах, которые в них входят.
Столбцы:

```
cluster String      - имя кластера
shard_num UInt32    - номер шарда в кластере, начиная с 1
shard_weight UInt32 - относительный вес шарда при записи данных
replica_num UInt32  - номер реплики в шарде, начиная с 1
host_name String    - имя хоста, как прописано в конфиге
host_address String - IP-адрес хоста, полученный из DNS
port UInt16         - порт, на который обращаться для соединения с сервером
user String         - имя пользователя, которого использовать для соединения с сервером
```

system.columns

Содержит информацию о столбцах всех таблиц.
С помощью этой таблицы можно получить информацию аналогично запросу DESCRIBE TABLE, но для многих таблиц сразу.

database String - имя базы данных, в которой находится таблица
table String - имя таблицы
name String - имя столбца
type String - тип столбца
default_type String - тип (DEFAULT, MATERIALIZED, ALIAS) выражения для значения по умолчанию, или пустая строка, если оно не описано
default_expression String - выражение для значения по умолчанию, или пустая строка, если оно не описано

system.databases

Таблица содержит один столбец name типа String - имя базы данных.
Для каждой базы данных, о которой знает сервер, будет присутствовать соответствующая запись в таблице.
Эта системная таблица используется для реализации запроса SHOW DATABASES.

system.dictionaries

Содержит информацию о внешних словарях.

Столбцы:

- name String — Имя словаря.
- type String — Тип словаря: Flat, Hashed, Cache.
- origin String — Путь к конфигурационному файлу, в котором описан словарь.
- attribute.names Array(String) — Массив имён атрибутов, предоставляемых словарём.
- attribute.types Array(String) — Соответствующий массив типов атрибутов, предоставляемых словарём.
- has_hierarchy UInt8 — Является ли словарь иерархическим.
- bytes_allocated UInt64 — Количество оперативной памяти, которое использует словарь.
- hit_rate Float64 — Для cache-словарей - доля использований, для которых значение было в кэше.
- element_count UInt64 — Количество хранящихся в словаре элементов.
- load_factor Float64 — Доля заполненности словаря (для hashed словаря - доля заполнения хэш-таблицы).
- creation_time DateTime — Время создания или последней успешной перезагрузки словаря.
- last_exception String — Текст ошибки, возникшей при создании или перезагрузке словаря, если словарь не удалось создать.
- source String - Текст, описывающий источник данных для словаря.

Заметим, что количество оперативной памяти, которое использует словарь, не является пропорциональным количеству элементов, хранящихся в словаре. Так, для flat и cached словарей, все ячейки памяти выделяются заранее, независимо от реальной заполненности словаря.

system.events

Содержит информацию о количестве произошедших в системе событий, для профилирования и мониторинга.
Пример: количество обработанных запросов типа SELECT.
Столбцы: event String - имя события, value UInt64 - количество.

system.functions

Содержит информацию об обычных и агрегатных функциях.

Столбцы:

- name (String) – Имя функции.
- is_aggregate (UInt8) – Признак, является ли функция агрегатной.

system.graphite_retentions

Содержит информацию о том, какие параметры graphite_rollup используются в таблицах с движками *GraphiteMergeTree.

Столбцы:
- config_name (String) - Имя параметра, используемого для graphite_rollup.
- regexp (String) - Шаблон имени метрики.
- function (String) - Имя агрегирующей функции.
- age (UInt64) - Минимальный возраст данных в секундах.
- precision (UInt64) - Точность определения возраста данных в секундах.
- priority (UInt16) - Приоритет раздела pattern.
- is_default (UInt8) - Является ли раздел pattern дефолтным.
- Tables.database (Array(String)) - Массив имён баз данных таблиц, использующих параметр config_name.
- Tables.table (Array(String)) - Массив имён таблиц, использующих параметр config_name.

system.merges

Содержит информацию о производящихся прямо сейчас слияниях и мутациях кусков для таблиц семейства MergeTree.

Столбцы:

- database String — Имя базы данных, в которой находится таблица.
- table String — Имя таблицы.
- elapsed Float64 — Время в секундах, прошедшее от начала выполнения слияния.
- progress Float64 — Доля выполненной работы от 0 до 1.
- num_parts UInt64 — Количество сливаемых кусков.
- result_part_name String — Имя куска, который будет образован в результате слияния.
- is_mutation UInt8 - Является ли данный процесс мутацией куска.
- total_size_bytes_compressed UInt64 — Суммарный размер сжатых данных сливаемых кусков.
- total_size_marks UInt64 — Суммарное количество засечек в сливаемых кусках.
- bytes_read_uncompressed UInt64 — Количество прочитанных байт, разжатых.
- rows_read UInt64 — Количество прочитанных строк.
- bytes_written_uncompressed UInt64 — Количество записанных байт, несжатых.
- rows_written UInt64 — Количество записанных строк.

system.metrics

system.numbers

Таблица содержит один столбец с именем number типа UInt64, содержащим почти все натуральные числа, начиная с нуля.
Эту таблицу можно использовать для тестов, а также если вам нужно сделать перебор.
Чтения из этой таблицы не распараллеливаются.

system.numbers_mt

То же самое, что и system.numbers, но чтение распараллеливается. Числа могут возвращаться в произвольном порядке.
Используется для тестов.

system.one

Таблица содержит одну строку с одним столбцом dummy типа UInt8, содержащим значение 0.
Эта таблица используется, если в SELECT запросе не указана секция FROM.
То есть, это - аналог таблицы DUAL, которую можно найти в других СУБД.

system.parts

Содержит информацию о кусках таблиц семейства MergeTree.

Каждая строка описывает один кусок данных.

Столбцы:

- partition (String) - Имя партиции. Что такое партиция можно узнать из описания запроса ALTER.

  Форматы:

  - YYYYMM для автоматической схемы партиционирования по месяцам.
  - any_string при партиционировании вручную.
- name (String) - имя куска;

- active (UInt8) - признак активности. Если кусок активен, то он используется таблицей, в противном случает он будет удален. Неактивные куски остаются после слияний;
- marks (UInt64) - количество засечек. Чтобы получить примерное количество строк в куске, умножьте marks на гранулированность индекса (обычно 8192);
- marks_size (UInt64) - размер файла с засечками;
- rows (UInt64) - количество строк;
- bytes (UInt64) - количество байт в сжатом виде;
- modification_time (DateTime) - время модификации директории с куском. Обычно соответствует времени создания куска;
- remove_time (DateTime) - время, когда кусок стал неактивным;
- refcount (UInt32) - количество мест, в котором кусок используется. Значение больше 2 говорит о том, что кусок участвует в запросах или в слияниях;
- min_date (Date) - минимальное значение ключа даты в куске;
- max_date (Date) - максимальное значение ключа даты в куске;
- min_block_number (UInt64) - минимальное число кусков, из которых состоит текущий после слияния;
- max_block_number (UInt64) - максимальное число кусков, из которых состоит текущий после слияния;
- level (UInt32) - глубина дерева слияний. Если слияний не было, то level=0;
- primary_key_bytes_in_memory (UInt64) - объем памяти (в байтах), занимаемой значениями первичных ключей;
- primary_key_bytes_in_memory_allocated (UInt64) - выделенный с резервом объем памяти (в байтах) для размещения первичных ключей;
- database (String) - имя базы данных;
- table (String) - имя таблицы;
- engine (String) - имя движка таблицы, без параметров.

system.part_log

Системная таблица system.part_log создается только в том случае, если задана серверная настройка part_log.

Содержит информацию о всех событиях, произошедших с кусками данных таблиц семейства MergeTree (например, события добавления, удаления или слияния данных).

Столбцы:

- event_type (Enum) — тип события. Столбец может содержать одно из следующих значений: NEW_PART — вставка нового куска; MERGE_PARTS — слияние кусков; DOWNLOAD_PART — загрузка с реплики; REMOVE_PART — удаление или отсоединение из таблицы с помощью DETACH PARTITION; MUTATE_PART — изменение куска.
- event_date (Date) — дата события;
- event_time (DateTime) — время события;
- duration_ms (UInt64) — длительность;
- database (String) — имя базы данных, в которой находится кусок;
- table (String) — имя таблицы, в которой находится кусок;
- part_name (String) — имя куска;
- partition_id (String) — идентификатор партиции, в которую был добавлен кусок. В столбце будет значение 'all', если таблица партициируется по выражению tuple();
- rows (UInt64) — число строк в куске;
- size_in_bytes (UInt64) — размер куска данных в байтах;

- merged_from (Array(String)) — массив имён кусков, из которых образован текущий кусок в результате слияния (также столбец заполняется в случае скачивания уже смерженного куска);
- bytes_uncompressed (UInt64) — количество прочитанных разжатых байт;
- read_rows (UInt64) — сколько было прочитано строк при слиянии кусков;
- read_bytes (UInt64) — сколько было прочитано байт при слиянии кусков;
- error (UInt16) — код ошибки, возникшей при текущем событии;
- exception (String) — текст ошибки.

Системная таблица system.part_log будет создана после первой вставки данных в таблицу MergeTree.

system.processes

Эта системная таблица используется для реализации запроса SHOW PROCESSLIST.
Столбцы:

user String - имя пользователя, который задал запрос. При распределённой обработке запроса, относится к пользователю, с помощью которого сервер-инициатор запроса отправил запрос на данный сервер, а не к имени пользователя, который задал распределённый запрос на сервер-инициатор запроса. address String - IP-адрес, с которого задан запрос. При распределённой обработке запроса, аналогично. elapsed Float64 - время в секундах, прошедшее от начала выполнения запроса. rows_read UInt64 - количество прочитанных из таблиц строк. При распределённой обработке запроса, на сервере-инициаторе запроса, представляет собой сумму по всем удалённым серверам. bytes_read UInt64 - количество прочитанных из таблиц байт, в несжатом виде. При распределённой обработке запроса, на сервере-инициаторе запроса, представляет собой сумму по всем удалённым серверам. total_rows_approx UInt64 - приблизительная оценка общего количества строк, которые должны быть прочитаны. При распределённой обработке запроса, на сервере-инициаторе запроса, представляет собой сумму по всем удалённым серверам. Может обновляться в процессе выполнения запроса, когда становятся известны новые источники для обработки. memory_usage UInt64 - потребление памяти запросом. Может не учитывать некоторые виды выделенной памяти. query String - текст запроса. В случае INSERT - без данных для INSERT-а. query_id String - идентификатор запроса, если был задан.

system.query_log

Содержит логи выполняемых запросов — дату запуска, длительность выполнения, текст возникших ошибок и другую информацию.

Внимание

Таблица не содержит данные, передаваемые в запросах INSERT.

Таблица system.query_log создаётся только в том случае, если задана серверная настройка query_log. Эта настройка определяет правила логирования. Например, с какой периодичностью логи будут записываться в таблицу. Также в этой настройке можно изменить название таблицы.

Чтобы включить логирование запросов, необходимо установить параметр log_queries в 1. Подробнее см. в разделе Настройки.

Логируются следующие запросы:

1. Запросы, которые были вызваны непосредственно клиентом.
2. Дочерние запросы, которые были вызваны другими запросами (при распределенном выполнении запросов). Для дочерних запросов, информация о родительских запросах содержится в столбцах initial_*.

Столбцы:

- type (UInt8) — тип события, которое возникло при выполнении запроса. Возможные значения:
  - 1 — запуск запроса произошел успешно;
  - 2 — запрос выполнен успешно;
  - 3 — при выполнении запроса возникла ошибка;
  - 4 — перед запуском запроса возникла ошибка.
- event_date (Date) — дата возникновения события;
- event_time (DateTime) — время возникновения события;
- query_start_time (DateTime) — время запуска запроса;
- query_duration_ms (UInt64) — длительность выполнения запроса;
- read_rows (UInt64) — количество прочитанных строк;
- read_bytes (UInt64) — количество прочитанных байт;
- written_rows (UInt64) — количество записанных строк, для запросов INSERT. Для остальных запросов столбец принимает значение 0.
- written_bytes (UInt64) — количество записанных байт, для запросов INSERT. Для остальных запросов столбец принимает значение 0.
- result_rows (UInt64) — количество строк, выведенных в результате;
- result_bytes (UInt64) — количество байт, выведенных в результате;
- memory_usage (FixedString(16)) — потребление памяти запросом;
- query (String) — строка запроса;
- exception (String) — сообщение об ошибке;
- stack_trace (String) — стектрейс (список методов, которые были вызваны до возникновения ошибки). Пустая строка, если запрос завершился успешно;
- is_initial_query (UInt8) — тип запроса:
  - 1 — запрос был запущен клиентом;
  - 0 — запрос был вызван другим запросом (при распределенном выполнении запросов);
- user (String) — имя пользователя, запустившего запрос;
- query_id (String) — идентификатор запроса;
- address (FixedString(16)) — имя хоста, с которого был отправлен запрос;
- port (UInt16) — порт удалённого сервера, принимающего запрос;
- initial_user (String) — имя пользователя, вызвавшего родительский запрос (для распределенного выполнения запросов);
- initial_query_id (String) — идентификатор родительского запроса, породившего исходный запрос;
- initial_address (FixedString(16)) — имя хоста, с которого был вызван родительский запрос;
- initial_port (UInt16) — порт удалённого сервера, принимающего родительский запрос;
- interface (UInt8) — используемый интерфейс. Возможные значения:
  - 1 — TCP.
  - 2 — HTTP.
- os_user (String) — операционная система на клиенте;
- client_hostname (String) — имя хоста, к которому подключен клиент clickhouse-client;
- client_name (String) — имя клиента clickhouse-client;
- client_revision (UInt32) — ревизия clickhouse-client;
- client_version_major (UInt32) — мажорная версия clickhouse-client;
- client_version_minor (UInt32) — минорная версия clickhouse-client;
- client_version_patch (UInt32) — patch-компонент версии clickhouse-client;
- http_method (UInt8) — используемый HTTP-метод. Возможные значения:
  - 0 — запрос был вызван из TCP интерфейса;
  - 1 — метод GET;
  - 2 — метод POST.
- http_user_agent (String) — содержимое заголовка UserAgent;
- quota_key (String) — ключ квоты, заданный в настройке quotas;
- revision (UInt32) — ревизия сервера ClickHouse;
- thread_numbers (Array(UInt32)) — номера потоков, участвующих в выполнении запроса;

- ProfileEvents.Names (Array(String)) — счётчики, измеряющие метрики:
  - время, потраченное на чтение и запись по сети;
  - чтение и запись на диск;
  - количество сетевых ошибок;
  - время, затраченное на ожидание, при ограниченной пропускной способности сети.
- ProfileEvents.Values (Array(UInt64)) — значения счётчиков, перечисленных в ProfileEvents.Names.
- Settings.Names (Array(String)) — настройки, которые были изменены при выполнении запроса. Чтобы включить отслеживание изменений настроек, установите параметр log_query_settings в 1.
- Settings.Values (Array(String)) — значения настроек, перечисленных в Settings.Names.

Каждый запрос создаёт в таблице query_log одно или два события, в зависимости от состояния этого запроса:

1. При успешном выполнении запроса, в таблице создаётся два события с типами 1 и 2 (см. столбец type).
2. Если в ходе выполнения запроса возникла ошибка, в таблице создаётся два события с типами 1 и 4.
3. Если ошибка возникла до начала выполнения запроса, создаётся одно событие с типом 3.

По умолчанию, логи записываются в таблицу с периодичностью в 7,5 секунд. Частоту записи логов можно регулировать настройкой query_log (см. параметр flush_interval_milliseconds). Чтобы принудительно пробросить логи из буфера памяти в таблицу, используйте запрос SYSTEM FLUSH LOGS.

При ручном удалении таблицы, она будет повторно создана на лету. Логи, которые содержались в таблице до её удаления, не сохраняются.

Примечание

Срок хранения логов в таблице неограничен — они не удаляются автоматически. Об удалении неактуальных логов вам нужно позаботиться самостоятельно.

Вы можете задать произвольный ключ партиционирования для таблицы system.query_log, в настройке query_log (см. параметр partition_by).

system.replicas

Содержит информацию и статус для реплицируемых таблиц, расположенных на локальном сервере.
Эту таблицу можно использовать для мониторинга. Таблица содержит по строчке для каждой Replicated*-таблицы.

Пример:

```
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
```

Столбцы:

database: имя БД
table: имя таблицы
engine: имя движка таблицы

is_leader: является ли реплика лидером

В один момент времени, не более одной из реплик является лидером. Лидер отвечает за выбор фоновых слияний, которые следует произвести.
Замечу, что запись можно осуществлять на любую реплику (доступную и имеющую сессию в ZK), независимо от лидерства.

is_readonly: находится ли реплика в режиме "только для чтения"
Этот режим включается, если в конфиге нет секции с ZK; если при переинициализации сессии в ZK произошла неизвестная ошибка; во время переинициализации сессии с ZK.

is_session_expired: истекла ли сессия с ZK.
В основном, то же самое, что и is_readonly.

future_parts: количество кусков с данными, которые появятся в результате INSERT-ов или слияний, которых ещё предстоит сделать

parts_to_check: количество кусков с данными в очереди на проверку
Кусок помещается в очередь на проверку, если есть подозрение, что он может быть битым.

zookeeper_path: путь к данным таблицы в ZK
replica_name: имя реплики в ZK; разные реплики одной таблицы имеют разное имя
replica_path: путь к данным реплики в ZK. То же самое, что конкатенация zookeeper_path/replicas/replica_path.

columns_version: номер версии структуры таблицы
Обозначает, сколько раз был сделан ALTER. Если на репликах разные версии, значит некоторые реплики сделали ещё не все ALTER-ы.

queue_size: размер очереди действий, которых предстоит сделать
К действиям относятся вставки блоков данных, слияния, и некоторые другие действия.
Как правило, совпадает с future_parts.

inserts_in_queue: количество вставок блоков данных, которых предстоит сделать
Обычно вставки должны быстро реплицироваться. Если величина большая - значит что-то не так.

merges_in_queue: количество слияний, которых предстоит сделать
Бывают длинные слияния - то есть, это значение может быть больше нуля продолжительное время.

Следующие 4 столбца имеют ненулевое значение только если активна сессия с ZK.

log_max_index: максимальный номер записи в общем логе действий
log_pointer: максимальный номер записи из общего лога действий, которую реплика скопировала в свою очередь для выполнения, плюс единица
Если log_pointer сильно меньше log_max_index, значит что-то не так.

total_replicas: общее число известных реплик этой таблицы
active_replicas: число реплик этой таблицы, имеющих сессию в ZK; то есть, число работающих реплик

Если запрашивать все столбцы, то таблица может работать слегка медленно, так как на каждую строчку делается несколько чтений из ZK.
Если не запрашивать последние 4 столбца (log_max_index, log_pointer, total_replicas, active_replicas), то таблица работает быстро.

Например, так можно проверить, что всё хорошо:

```
SELECT
    database,
    table,
    is_leader,
    is_readonly,
    is_session_expired,
    future_parts,
    parts_to_check,
    columns_version,
    queue_size,
    inserts_in_queue,
    merges_in_queue,
    log_max_index,
    log_pointer,
    total_replicas,
    active_replicas
FROM system.replicas
WHERE
       is_readonly
    OR is_session_expired
    OR future_parts > 20
    OR parts_to_check > 10
    OR queue_size > 20
    OR inserts_in_queue > 10
    OR log_max_index - log_pointer > 10
    OR total_replicas < 2
    OR active_replicas < total_replicas
```

Если этот запрос ничего не возвращает - значит всё хорошо.

system.settings

Содержит информацию о настройках, используемых в данный момент.
То есть, используемых для выполнения запроса, с помощью которого вы читаете из таблицы system.settings.

Столбцы:

```
name String   - имя настройки
value String  - значение настройки
changed UInt8 - была ли настройка явно задана в конфиге или изменена явным образом
```

Пример:

```
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

Таблица содержит столбцы database, name, engine типа String.
Также таблица содержит три виртуальных столбца: metadata_modification_time типа DateTime, create_table_query и engine_full типа String.
Для каждой таблицы, о которой знает сервер, будет присутствовать соответствующая запись в таблице system.tables.
Эта системная таблица используется для реализации запросов SHOW TABLES.

system.zookeeper

Таблицы не существует, если ZooKeeper не сконфигурирован. Позволяет читать данные из ZooKeeper кластера, описанного в конфигурации.
В запросе обязательно в секции WHERE должно присутствовать условие на равенство path - путь в ZooKeeper, для детей которого вы хотите получить данные.

Запрос `SELECT * FROM system.zookeeper WHERE path = '/clickhouse'` выведет данные по всем детям узла `/clickhouse`.
Чтобы вывести данные по всем узлам в корне, напишите path = '/'.
Если узла, указанного в path не существует, то будет брошено исключение.

Столбцы:

- `name String` — Имя узла.
- `path String` — Путь к узлу.
- `value String` — Значение узла.
- `dataLength Int32` — Размер значения.
- `numChildren Int32` — Количество детей.
- `czxid Int64` — Идентификатор транзакции, в которой узел был создан.
- `mzxid Int64` — Идентификатор транзакции, в которой узел был последний раз изменён.
- `pzxid Int64` — Идентификатор транзакции, последний раз удаливший или добавивший детей.
- `ctime DateTime` — Время создания узла.
- `mtime DateTime` — Время последней модификации узла.
- `version Int32` — Версия узла - количество раз, когда узел был изменён.
- `cversion Int32` — Количество добавлений или удалений детей.
- `aversion Int32` — Количество изменений ACL.
- `ephemeralOwner Int64` — Для эфемерных узлов - идентификатор сессии, которая владеет этим узлом.

Пример:

```
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

Таблица содержит информацию о ходе выполнения мутаций MergeTree-таблиц. Каждой команде мутации соответствует одна строка. В таблице есть следующие столбцы:

database, **table** - имя БД и таблицы, к которой была применена мутация.

mutation_id - ID запроса. Для реплицированных таблиц эти ID соответствуют именам записей в директории <table_path_in_zookeeper>/mutations/ в ZooKeeper, для нереплицированных - именам файлов в директории с данными таблицы.

command - Команда мутации (часть запроса после ALTER TABLE [db.]table).

create_time - Время создания мутации.

block_numbers.partition_id, **block_numbers.number** - Nested-столбец. Для мутаций реплицированных таблиц для каждой партиции содержит номер блока, полученный этой мутацией (в каждой партиции будут изменены только куски, содержащие блоки с номерами, меньшими номера, полученного мутацией в этой партиции). Для нереплицированных таблиц нумерация блоков сквозная по партициям, поэтому столбец содержит одну запись с единственным номером блока, полученным мутацией.

parts_to_do - Количество кусков таблицы, которые ещё предстоит изменить.

is_done - Завершена ли мутация. Замечание: даже если parts_to_do = 0, для реплицированной таблицы возможна ситуация, когда мутация ещё не завершена из-за долго выполняющейся вставки, которая добавляет данные, которые нужно будет мутировать.

Если во время мутации какого-либо куска возникли проблемы, заполняются следующие столбцы:

latest_failed_part - Имя последнего куска, мутация которого не удалась.

latest_fail_time - Время последней неудачной мутации куска.

latest_fail_reason - Ошибка, возникшая при последней неудачной мутации куска.

Конфигурационные параметры сервера

Раздел содержит описания настроек сервера, которые не могут изменяться на уровне сессии или запроса.

Рассмотренные настройки хранятся в файле config.xml сервера ClickHouse.

Прочие настройки описаны в разделе "Настройки".

Перед изучением настроек ознакомьтесь с разделом Конфигурационные файлы, обратите внимание на использование подстановок (атрибуты incl и optional).

Конфигурационные параметры сервера

builtin_dictionaries_reload_interval

Интервал (в секундах) перезагрузки встроенных словарей.

ClickHouse перезагружает встроенные словари с заданным интервалом. Это позволяет править словари "на лету" без перезапуска сервера.

Значение по умолчанию - 3600.

Пример

```
<builtin_dictionaries_reload_interval>3600</builtin_dictionaries_reload_interval>
```

compression

Настройки компрессии данных.

Внимание

Лучше не использовать, если вы только начали работать с ClickHouse.

Общий вид конфигурации:

```
<compression>
    <case>
      <parameters/>
    </case>
    ...
</compression>
```

Можно сконфигурировать несколько разделов <case>.

Поля блока <case>:

- min_part_size - Минимальный размер части таблицы.
- min_part_size_ratio - Отношение размера минимальной части таблицы к полному размеру таблицы.
- method - Метод сжатия. Возможные значения: lz4, zstd (экспериментальный).

ClickHouse проверит условия min_part_size и min_part_size_ratio и выполнит те блоки case, для которых условия совпали. Если ни один <case> не подходит, то ClickHouse применит алгоритм сжатия lz4.

Пример

```
<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

База данных по умолчанию.

Перечень баз данных можно получить запросом SHOW DATABASES.

Пример

```
<default_database>default</default_database>
```

default_profile

Профиль настроек по умолчанию.

Профили настроек находятся в файле, указанном в параметре user_config.

Пример

```
<default_profile>default</default_profile>
```

dictionaries_config

Путь к конфигурации внешних словарей.

Путь:

- Указывается абсолютным или относительно конфигурационного файла сервера.
- Может содержать wildcard-ы * и ?.

Смотрите также "Внешние словари".

Пример

```
<dictionaries_config>*_dictionary.xml</dictionaries_config>
```

dictionaries_lazy_load

Отложенная загрузка словарей.

Если true, то каждый словарь создаётся при первом использовании. Если словарь не удалось создать, то вызов функции, использующей словарь, сгенерирует исключение.

Если false, то все словари создаются при старте сервера, и в случае ошибки сервер завершает работу.

По умолчанию - true.

Пример

```
<dictionaries_lazy_load>true</dictionaries_lazy_load>
```

format_schema_path

Путь к каталогу со схемами для входных данных. Например со схемами для формата CapnProto.

Пример

```
<!-- Directory containing schema files for various input formats. -->
<format_schema_path>format_schemas/</format_schema_path>
```

graphite

Отправка даных в Graphite.

Настройки:

- host - Сервер Graphite.
- port - Порт сервера Graphite.
- interval - Период отправки в секундах.
- timeout - Таймаут отправки данных в секундах.
- root_path - Префикс для ключей.
- metrics - Отправка данных из таблицы :ref:system_tables-system.metrics.
- events - Отправка данных из таблицы :ref:system_tables-system.events.
- asynchronous_metrics - Отправка данных из таблицы :ref:system_tables-system.asynchronous_metrics.

Можно определить несколько секций <graphite>, например, для передачи различных данных с различной частотой.

Пример

```
<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

Настройка прореживания данных для Graphite.

Подробнее читайте в разделе GraphiteMergeTree.

Пример

```
<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

Порт для обращений к серверу по протоколу HTTP(s).

Если указан https_port, то требуется конфигурирование openSSL.

Если указан http_port, то настройка openSSL игнорируется, даже если она задана.

Пример

```
<https>0000</https>
```

http_server_default_response

Страница, показываемая по умолчанию, при обращении к HTTP(s) серверу ClickHouse.

Пример

Показывает https://tabix.io/ при обращенинии к 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

Путь к файлу с подстановками.

Подробности смотрите в разделе "Конфигурационный файлы".

Пример

```
<include_from>/etc/metrica.xml</include_from>
```

interserver_http_port

Порт для обмена между серверами ClickHouse.

Пример

```
<interserver_http_port>9009</interserver_http_port>
```

interserver_http_host

Имя хоста, которое могут использовать другие серверы для обращения к этому.

Если не указано, то определяется аналогично команде hostname -f.

Удобно использовать, чтобы отвязаться от конкретного сетевого интерфейса.

Пример

```
<interserver_http_host>example.yandex.ru</interserver_http_host>
```

keep_alive_timeout

Время в секундах, в течение которого ClickHouse ожидает входящих запросов прежде, чем закрыть соединение.

Пример

```
<keep_alive_timeout>3</keep_alive_timeout>
```

listen_host

Ограничение по хостам, с которых может прийти запрос. Если необходимо, чтобы сервер отвечал всем, то надо указать ::.

Примеры:

```
<listen_host>::1</listen_host>
<listen_host>127.0.0.1</listen_host>
```

logger

Настройки логгирования.

Ключи:

- level - Уровень логгирования. Допустимые значения: trace, debug, information, warning, error.
- log - Файл лога. Содержит все записи согласно level.
- errorlog - Файл лога ошибок.
- size - Размер файла. Действует для log и errorlog. Как только файл достиг размера size, ClickHouse архивирует и переименовывает его, а на его месте создает новый файл лога.
- count - Количество заархивированных файлов логов, которые сохраняет ClickHouse.

Пример

```
<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>
```

Также, существует поддержка записи в syslog. Пример конфига:

```
<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>
```

Ключи:
- user_syslog - обязательная настройка, если требуется запись в syslog
- address - хост[:порт] демона syslogd. Если не указан, используется локальный
- hostname - опционально, имя хоста, с которого отсылаются логи
- facility - категория syslog,
записанная в верхнем регистре, с префиксом "LOG_": (LOG_USER, LOG_DAEMON, LOG_LOCAL3 и прочие).
Значения по умолчанию: при указанном address - LOG_USER, иначе - LOG_DAEMON
- format - формат сообщений. Возможные значения - bsd и syslog

macros

Подстановки параметров реплицируемых таблиц.

Можно не указывать, если реплицируемых таблицы не используются.

Подробнее смотрите в разделе "Создание реплицируемых таблиц".

Пример

```
<macros incl="macros" optional="true" />
```

mark_cache_size

Приблизительный размер (в байтах) кеша "засечек", используемых движками таблиц семейства MergeTree.

Кеш общий для сервера, память выделяется по мере необходимости. Кеш не может быть меньше, чем 5368709120.

Пример

```
<mark_cache_size>5368709120</mark_cache_size>
```

max_concurrent_queries

Максимальное количество одновременно обрабатываемых запросов.

Пример

```
<max_concurrent_queries>100</max_concurrent_queries>
```

max_connections

Максимальное количество входящих соединений.

Пример

```
<max_connections>4096</max_connections>
```

max_open_files

Максимальное количество открытых файлов.

По умолчанию - maximum.

Рекомендуется использовать в Mac OS X, поскольу функция getrlimit() возвращает некорректное значение.

Пример

```
<max_open_files>262144</max_open_files>
```

max_table_size_to_drop

Ограничение на удаление таблиц.

Если размер таблицы семейства MergeTree превышает max_table_size_to_drop (в байтах), то ее нельзя удалить запросом DROP.

Если таблицу все же необходимо удалить, не перезапуская при этом сервер ClickHouse, то необходимо создать файл <clickhouse-path>/flags/force_drop_table и выполнить запрос DROP.

Значение по умолчанию - 50GB.

Значение 0 означает, что можно удалять все таблицы без ограничений.

Пример

```
<max_table_size_to_drop>0</max_table_size_to_drop>
```

merge_tree

Тонкая настройка таблиц семейства MergeTree.

Подробнее смотрите в заголовочном файле MergeTreeSettings.h.

Пример

```
<merge_tree>
    <max_suspicious_broken_parts>5</max_suspicious_broken_parts>
</merge_tree>
```

openSSL

Настройки клиента/сервера SSL.

Поддержку SSL обеспечивает библиотека libpoco. Описание интерфейса находится в файле SSLManager.h

Ключи настроек сервера/клиента:

- privateKeyFile - Путь к файлу с секретным ключем сертификата в формате PEM. Файл может содержать ключ и сертификат одновременно.
- certificateFile - Путь к файлу сертификата клиента/сервера в формате PEM. Можно не указывать, если privateKeyFile содержит сертификат.
- caConfig - Путь к файлу или каталогу, которые содержат доверенные корневые сертификаты.
- verificationMode - Способ проверки сертификатов узла. Подробности находятся в описании класса Context. Допустимые значения: none, relaxed, strict, once.
- verificationDepth - Максимальная длина верификационой цепи. Верификация завершится ошибкой, если длина цепи сертификатов превысит установленное значение.
- loadDefaultCAFile - Признак того, что будут использоваться встроенные CA-сертификаты для OpenSSL. Допустимые значения: true, false. |
- cipherList - Поддерживаемые OpenSSL-шифры. Например, ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH.
- cacheSessions - Включение/выключение кеширования сессии. Использовать обязательно вместе с sessionIdContext. Допустимые значения: true, false.
- sessionIdContext - Уникальный набор произвольных символов, которые сервер добавляет к каждому сгенерированному идентификатору. Длина строки не должна превышать SSL_MAX_SSL_SESSION_ID_LENGTH. Рекомендуется к использованию всегда, поскольку позволяет избежать проблем как в случае, если сервер кеширует сессию, так и если клиент затребовал кеширование. По умолчанию ${application.name}.
- sessionCacheSize - Максимальное количество сессий, которые кэширует сервер. По умолчанию - 1024*20. 0 - неограниченное количество сессий.
- sessionTimeout - Время кеширования сессии на севрере.
- extendedVerification - Автоматическая расширенная проверка сертификатов после завершении сессии. Допустимые значения: true, false.
- requireTLSv1 - Требование соединения TLSv1. Допустимые значения: true, false.
- requireTLSv1_1 - Требование соединения TLSv1.1. Допустимые значения: true, false.
- requireTLSv1_2 - Требование соединения TLSv1.2. Допустимые значения: true, false.
- fips - Активация режима OpenSSL FIPS. Поддерживается, если версия OpenSSL, с которой собрана библиотека поддерживает fips.
- privateKeyPassphraseHandler - Класс (подкласс PrivateKeyPassphraseHandler)запрашивающий кодовую фразу доступа к секретному ключу. Например, <privateKeyPassphraseHandler>, <name>KeyFileHandler</name>, <options><password>test</password></options>, </privateKeyPassphraseHandler>.
- invalidCertificateHandler - Класс (подкласс CertificateHandler) для подтверждения невалидных сертификатов. Например, <invalidCertificateHandler> <name>ConsoleCertificateHandler</name> </invalidCertificateHandler>.
- disableProtocols - Запрещенные к искользованию протоколы.
- preferServerCiphers - Предпочтение серверных шифров на клиенте.

Пример настройки:

```
<openSSL>
    <server>
        <!-- openssl req -subj "/CN=localhost" -new -newkey rsa:2048 -days 365 -nodes -x509 -keyout /etc/clickhouse-
server/server.key -out /etc/clickhouse-server/server.crt -->
        <certificateFile>/etc/clickhouse-server/server.crt</certificateFile>
        <privateKeyFile>/etc/clickhouse-server/server.key</privateKeyFile>
        <!-- openssl dhparam -out /etc/clickhouse-server/dhparam.pem 4096 -->
        <dhParamsFile>/etc/clickhouse-server/dhparam.pem</dhParamsFile>
        <verificationMode>none</verificationMode>
        <loadDefaultCAFile>true</loadDefaultCAFile>
        <cacheSessions>true</cacheSessions>
        <disableProtocols>sslv2,sslv3</disableProtocols>
        <preferServerCiphers>true</preferServerCiphers>
    </server>
    <client>
        <loadDefaultCAFile>true</loadDefaultCAFile>
        <cacheSessions>true</cacheSessions>
        <disableProtocols>sslv2,sslv3</disableProtocols>
        <preferServerCiphers>true</preferServerCiphers>
        <!-- Use for self-signed: <verificationMode>none</verificationMode> -->
        <invalidCertificateHandler>
            <!-- Use for self-signed: <name>AcceptCertificateHandler</name> -->
            <name>RejectCertificateHandler</name>
        </invalidCertificateHandler>
    </client>
</openSSL>
```

part_log

Логгирование событий, связанных с данными типа MergeTree. Например, события добавления или мержа данных. Лог можно использовать для симуляции алгоритмов слияния, чтобы сравнивать их характеристики. Также, можно визуализировать процесс слияния.

Запросы логгируются не в отдельный файл, а в таблицу system.part_log. Вы можете изменить название этой таблицы в параметре table (см. ниже).

При настройке логгирования используются следующие параметры:

- database — имя базы данных;
- table — имя таблицы;
- partition_by — устанавливает произвольный ключ партиционирования;
- flush_interval_milliseconds — период сброса данных из буфера в памяти в таблицу.

Пример

```
<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

Путь к каталогу с данными.

Обратите внимание

Завершающий слеш обязателен.

Пример

```
<path>/var/lib/clickhouse/</path>
```

query_log

Настройка логирования запросов, принятых с настройкой log_queries=1.

Запросы логируются не в отдельный файл, а в системную таблицу system.query_log. Вы можете изменить название этой таблицы в параметре table (см. ниже).

При настройке логирования используются следующие параметры:

- database — имя базы данных;
- table — имя таблицы, куда будет записываться лог;
- partition_by — произвольный ключ партиционирования для таблицы с логами;
- flush_interval_milliseconds — период сброса данных из буфера в памяти в таблицу.

Если таблица не существует, то ClickHouse создаст её. Если структура журнала запросов изменилась при обновлении сервера ClickHouse, то таблица со старой структурой переименовывается, а новая таблица создается автоматически.

Пример

```
<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

Конфигурация кластеров, которые использует движок таблиц Distributed.

Пример настройки смотрите в разделе "Движки таблиц/Distributed".

Пример

```
<remote_servers incl="clickhouse_remote_servers" />
```

Значение атрибута incl смотрите в разделе "Конфигурационные файлы".

timezone

Временная зона сервера.

Указывается идентификатором IANA в виде часового пояса UTC или географического положения (например, Africa/Abidjan).

Временная зона необходима при преобразованиях между форматами String и DateTime, которые возникают при выводе полей DateTime в текстовый формат (на экран или в файл) и при получении DateTime из строки. Также, временная зона используется в функциях, которые работают со временем и датой, если они не получили временную зону в параметрах вызова.

Пример

```
<timezone>Europe/Moscow</timezone>
```

tcp_port

Порт для взаимодействия с клиентами по протоколу TCP.

Пример

```
<tcp_port>9000</tcp_port>
```

tcp_port_secure

TCP порт для защищённого обмена данными с клиентами. Используйте с настройкой OpenSSL.

Возможные значения

Положительное целое число.

Значение по умолчанию

```
<tcp_port_secure>9440</tcp_port_secure>
```

tmp_path

Путь ко временным данным для обработки больших запросов.

Обратите внимание

Завершающий слеш обязателен.

Пример

```
<tmp_path>/var/lib/clickhouse/tmp/</tmp_path>
```

uncompressed_cache_size

Размер кеша (в байтах) для несжатых данных, используемых движками таблиц семейства MergeTree.

Кеш единый для сервера. Память выделяется по-требованию. Кеш используется в том случае, если включена опция use_uncompressed_cache.

Несжатый кеш выгодно использовать для очень коротких запросов в отдельных случаях.

Пример

```
<uncompressed_cache_size>8589934592</uncompressed_cache_size>
```

user_files_path

Каталог с пользовательскими файлами. Используется в табличной функции file().

Пример

```
<user_files_path>/var/lib/clickhouse/user_files/</user_files_path>
```

users_config

Путь к файлу, который содержит:

- Конфигурации пользователей.
- Права доступа.
- Профили настроек.
- Настройки квот.

Пример

```
<users_config>users.xml</users_config>
```

zookeeper

Конфигурация серверов ZooKeeper.

Содержит параметры для взаимодействия ClickHouse с кластером ZooKeeper.

ClickHouse использует ZooKeeper для хранения метаданных о репликах при использовании реплицированных таблиц.

Параметр можно не указывать, если реплицированные таблицы не используются.

Содержит следующие параметры:

- node — нода ZooKeeper. Может содержать несколько узлов.

  Пример:

  xml <node index="1"> <host>example_host</host> <port>2181</port> </node>

  Аттрибут index не используется в ClickHouse. Он присутствует в примере в связи с тем, что на серверах могут быть установлены другие программы, которые могут считывать тот же конфигурационный файл.

- session_timeout_ms — максимальный таймаут для сессии в миллисекундах (default: 30000).
- operation_timeout_ms — максимальный таймаут для операции в миллисекундах (default: 10000).
- root — корневая znode, которая используется сервером ClickHouse для всех остальных znode. Опционально.
- identity — пара usename:password для авторизации в кластере ZooKeeper. Опционально.

Пример

```
<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>
```

См. также:

- Репликация.
- ZooKeeper Programmer's Guide

use_minimalistic_part_header_in_zookeeper

Способ хранения заголовков кусков данных в ZooKeeper.

Параметр применяется только к семейству таблиц MergeTree. Его можно установить:

- Глобально в разделе merge_tree файла config.xml.

  ClickHouse использует этот параметр для всех таблиц на сервере. Вы можете изменить настройку в любое время. Существующие таблицы изменяют свое поведение при изменении параметра.

- Для каждой отдельной таблицы.

  При создании таблицы укажите соответствующую настройку движка. Поведение существующей таблицы с установленным параметром не изменяется даже при изменении глобального параметра.

Возможные значения

- 0 — функциональность выключена.
- 1 — функциональность включена.

Если use_minimalistic_part_header_in_zookeeper = 1, то реплицированные таблицы хранят заголовки кусков данных в компактном виде, используя только одну znode. Если таблица содержит много столбцов, этот метод хранения значительно уменьшает объем данных, хранящихся в Zookeeper.

Внимание

После того как вы установили use_minimalistic_part_header_in_zookeeper = 1, невозможно откатить ClickHouse до версии, которая не поддерживает этот параметр. Будьте осторожны при обновлении ClickHouse на серверах в кластере. Не обновляйте все серверы сразу. Безопаснее проверять новые версии ClickHouse в тестовой среде или только на некоторых серверах кластера.

Заголовки частей данных, ранее сохранённые с этим параметром, не могут быть восстановлены в их предыдущем (некомпактном) представлении.

Значение по умолчанию: 0.

Настройки

Все настройки, описанные ниже, могут быть заданы несколькими способами.
Настройки задаются послойно, т.е. каждый следующий слой перезаписывает предыдущие настройки.

Способы задания настроек, упорядоченные по приоритету:

- Настройки в конфигурационном файле сервера users.xml.

  Устанавливаются в элементе <profiles>.

- Настройки для сессии.

  Из консольного клиента ClickHouse в интерактивном режиме отправьте запрос SET setting=value. Аналогично можно использовать ClickHouse-сессии в HTTP-протоколе, для этого необходимо указывать HTTP-праметр session_id.

- Настройки для запроса.

  - При запуске консольного клиента ClickHouse в неинтерактивном режиме установите параметр запуска --setting=value.
  - При использовании HTTP API передавайте cgi-параметры (URL?setting_1=value&setting_2=value...).

Настройки, которые можно задать только в конфигурационном файле сервера, в разделе не рассматриваются.

Разрешения для запросов

Запросы в ClickHouse можно разделить на несколько типов:

1. Запросы на чтение данных: SELECT, SHOW, DESCRIBE, EXISTS.
2. Запросы за запись данных: INSERT, OPTIMIZE.
3. Запросы на изменение настроек: SET, USE.
4. Запросы DDL: CREATE, ALTER, RENAME, ATTACH, DETACH, DROP TRUNCATE.
5. KILL QUERY.

Разрешения пользователя по типу запроса регулируются параметрами:

- readonly — ограничивает разрешения для всех типов запросов, кроме DDL.
- allow_ddl — ограничивает разрешения для DDL запросов.

KILL QUERY выполняется с любыми настройками.

readonly

Ограничивает разрешения для запросов на чтение данных, запись данных и изменение параметров.

Разделение запросов по типам смотрите по тексту выше по тексту.

Возможные значения

- 0 — разрешены все запросы.
- 1 — разрешены только запросы на чтение данных.
- 2 — разрешены запросы на чтение данных и изменение настроек.

После установки readonly = 1пользователь не может изменить настройки readonly и allow_ddl в текущей сессии.

При использовании метода GET в HTTP интерфейсе, readonly = 1 устанавливается автоматически. Для изменения данных используйте метод POST.

Значение по умолчанию

0

allow_ddl

Разрешает/запрещает DDL запросы.

Разделение запросов по типам смотрите по тексту выше по тексту.

Возможные значения

- 0 — DDL запросы не разрешены.
- 1 — DDL запросы разрешены.

Если allow_ddl = 0, то невозможно выполнить SET allow_ddl = 1 для текущей сессии.

Значение по умолчанию

1

Ограничения на сложность запроса

Ограничения на сложность запроса - часть настроек.
Используются, чтобы обеспечить более безопасное исполнение запросов из пользовательского интерфейса.
Почти все ограничения действуют только на SELECT-ы.
При распределённой обработке запроса, ограничения действуют на каждом сервере по-отдельности.

Ограничения проверяются на каждый блок обработанных данных, а не на каждую строку. В связи с этим, ограничения могут быть превышены на размер блока.

Ограничения вида "максимальное количество чего-нибудь" могут принимать значение 0, которое обозначает "не ограничено".
Для большинства ограничений также присутствует настройка вида overflow_mode - что делать, когда ограничение превышено.
Оно может принимать одно из двух значений: throw или break; а для ограничения на агрегацию (group_by_overflow_mode) есть ещё значение any.

throw - кинуть исключение (по умолчанию).

break - прервать выполнение запроса и вернуть неполный результат, как будто исходные данные закончились.

any (только для group_by_overflow_mode) - продолжить агрегацию по ключам, которые успели войти в набор, но не добавлять новые ключи в набор.

max_memory_usage

Максимальный возможный объем оперативной памяти для выполнения запроса на одном сервере.

В конфигурационном файле по умолчанию, ограничение равно 10 ГБ.

Настройка не учитывает объём свободной памяти или общий объём памяти на машине.
Ограничение действует на один запрос, в пределах одного сервера.
Текущее потребление памяти для каждого запроса можно посмотреть с помощью SHOW PROCESSLIST.
Также отслеживается и выводится в лог пиковое потребление памяти для каждого запроса.

Потребление памяти не отслеживается для состояний некоторых агрегатных функций.

Потребление памяти не полностью учитывается для состояний агрегатных функций min, max, any, anyLast, argMin, argMax от аргументов String и Array.

Потребление памяти ограничивается также параметрами max_memory_usage_for_user и max_memory_usage_for_all_queries.

max_memory_usage_for_user

Максимальный возможный объем оперативной памяти для запросов пользователя на одном сервере.

Значения по умолчанию определены в файле Settings.h. По умолчанию размер не ограничен (max_memory_usage_for_user = 0).

Смотрите также описание настройки max_memory_usage.

max_memory_usage_for_all_queries

Максимальный возможный объем оперативной памяти для всех запросов на одном сервере.

Значения по умолчанию определены в файле Settings.h. По умолчанию размер не ограничен (max_memory_usage_for_all_queries = 0).

Смотрите также описание настройки max_memory_usage.

max_rows_to_read

Следующие ограничения могут проверяться на каждый блок (а не на каждую строку). То есть, ограничения могут быть немного нарушены.
При выполнении запроса в несколько потоков, следующие ограничения действуют в каждом потоке по-отдельности.

Максимальное количество строчек, которое можно прочитать из таблицы при выполнении запроса.

max_bytes_to_read

Максимальное количество байт (несжатых данных), которое можно прочитать из таблицы при выполнении запроса.

read_overflow_mode

Что делать, когда количество прочитанных данных превысило одно из ограничений: throw или break. По умолчанию: throw.

max_rows_to_group_by

Максимальное количество уникальных ключей, получаемых в процессе агрегации. Позволяет ограничить потребление оперативки при агрегации.

group_by_overflow_mode

Что делать, когда количество уникальных ключей при агрегации превысило ограничение: throw, break или any. По умолчанию: throw.
Использование значения any позволяет выполнить GROUP BY приближённо. Качество такого приближённого вычисления сильно зависит от статистических свойств данных.

max_rows_to_sort

Максимальное количество строк до сортировки. Позволяет ограничить потребление оперативки при сортировке.

max_bytes_to_sort

Максимальное количество байт до сортировки.

sort_overflow_mode

Что делать, если количество строк, полученное перед сортировкой, превысило одно из ограничений: throw или break. По умолчанию: throw.

max_result_rows

Ограничение на количество строк результата. Проверяются также для подзапросов и на удалённых серверах при выполнении части распределённого запроса.

max_result_bytes

Ограничение на количество байт результата. Аналогично.

result_overflow_mode

Что делать, если объём результата превысил одно из ограничений: throw или break. По умолчанию: throw.
Использование break по смыслу похоже на LIMIT.

max_execution_time

Максимальное время выполнения запроса в секундах.
На данный момент не проверяется при одной из стадий сортировки а также при слиянии и финализации агрегатных функций.

timeout_overflow_mode

Что делать, если запрос выполняется дольше max_execution_time: throw или break. По умолчанию: throw.

min_execution_speed

Минимальная скорость выполнения запроса в строчках в секунду. Проверяется на каждый блок данных по истечении timeout_before_checking_execution_speed. Если скорость выполнения запроса оказывается меньше, то кидается исключение.

min_execution_speed_bytes

Минимальная скорость выполнения запроса в строках на байт. Он проверяется для каждого блока данных после timeout_before_checking_execution_speed. Если скорость выполнения запроса меньше, исключение.

max_execution_speed

Максимальная скорость выполнения запроса в строках в секунду. Он проверяется для каждого блока данных после timeout_before_checking_execution_speed. Если скорость выполнения запроса выше, скорость будет снижена.

max_execution_speed_bytes

Максимальная скорость выполнения запроса в байтах в секунду. Он проверяется для каждого блока данных после timeout_before_checking_execution_speed. Если скорость выполнения запроса выше, скорость будет снижена.

timeout_before_checking_execution_speed

Проверять, что скорость выполнения запроса не слишком низкая (не меньше min_execution_speed), после прошествия указанного времени в секундах.

max_columns_to_read

Максимальное количество столбцов, которых можно читать из таблицы в одном запросе. Если запрос требует чтения большего количества столбцов - кинуть исключение.

max_temporary_columns

Максимальное количество временных столбцов, которых необходимо одновременно держать в оперативке, в процессе выполнения запроса, включая константные столбцы. Если временных столбцов оказалось больше - кидается исключение.

max_temporary_non_const_columns

То же самое, что и max_temporary_columns, но без учёта столбцов-констант.
Стоит заметить, что столбцы-константы довольно часто образуются в процессе выполнения запроса, но расходуют примерно нулевое количество вычислительных ресурсов.

max_subquery_depth

Максимальная вложенность подзапросов. Если подзапросы более глубокие - кидается исключение. По умолчанию: 100.

max_pipeline_depth

Максимальная глубина конвейера выполнения запроса. Соответствует количеству преобразований, которое проходит каждый блок данных в процессе выполнения запроса. Считается в пределах одного сервера. Если глубина конвейера больше - кидается исключение. По умолчанию: 1000.

max_ast_depth

Максимальная вложенность синтаксического дерева запроса. Если превышена - кидается исключение.
На данный момент, проверяются не во время парсинга а уже после парсинга запроса. То есть, во время парсинга может быть создано слишком глубокое синтаксическое дерево, но запрос не будет выполнен. По умолчанию: 1000.

max_ast_elements

Максимальное количество элементов синтаксического дерева запроса. Если превышено - кидается исключение.
Аналогично, проверяется уже после парсинга запроса. По умолчанию: 10 000.

max_rows_in_set

Максимальное количество строчек для множества в секции IN, создаваемого из подзапроса.

max_bytes_in_set

Максимальное количество байт (несжатых данных), занимаемое множеством в секции IN, создаваемым из подзапроса.

set_overflow_mode

Что делать, когда количество данных превысило одно из ограничений: throw или break. По умолчанию: throw.

max_rows_in_distinct

Максимальное количество различных строчек при использовании DISTINCT.

max_bytes_in_distinct

Максимальное количество байт, занимаемых хэш-таблицей, при использовании DISTINCT.

distinct_overflow_mode

Что делать, когда количество данных превысило одно из ограничений: throw или break. По умолчанию: throw.

max_rows_to_transfer

Максимальное количество строчек, которых можно передать на удалённый сервер или сохранить во временную таблицу, при использовании GLOBAL IN.

max_bytes_to_transfer

Максимальное количество байт (несжатых данных), которых можно передать на удалённый сервер или сохранить во временную таблицу, при использовании GLOBAL IN.

transfer_overflow_mode

Что делать, когда количество данных превысило одно из ограничений: throw или break. По умолчанию: throw.

Настройки

distributed_product_mode

Изменяет поведение распределенных подзапросов.

ClickHouse применяет настройку в тех случаях, когда запрос содержит произведение распределённых таблиц, т.е. когда запрос к распределенной таблице содержит не-GLOBAL подзапрос к также распределенной таблице.

Условия применения:

- Только подзапросы для IN, JOIN.
- Только если в секции FROM используется распределённая таблица, содержащая более одного шарда.
- Если подзапрос касается распределенной таблицы, содержащей более одного шарда,
- Не используется в случае табличной функции remote.

Возможные значения:

- deny — значение по умолчанию. Запрещает использование таких подзапросов (При попытке использование вернет исключение "Double-distributed IN/JOIN subqueries is denied");
- local — заменяет базу данных и таблицу в подзапросе на локальные для конечного сервера (шарда), оставив обычный IN / JOIN.
- global — заменяет запрос IN / JOIN на GLOBAL IN / GLOBAL JOIN.
- allow — разрешает использование таких подзапросов.

enable_optimize_predicate_expression

Включает пробрасывание предикатов в подзапросы для запросов SELECT.

Пробрасывание предикатов может существенно уменьшить сетевой трафик для распределенных запросов.

Возможные значения:

- 0 — выключена.
- 1 — включена.

Значение по умолчанию: 0.

Использование

Рассмотрим следующие запросы:

1. `SELECT count() FROM test_table WHERE date = '2018-10-10'`
2. `SELECT count() FROM (SELECT * FROM test_table) WHERE date = '2018-10-10'`

Если `enable_optimize_predicate_expression = 1`, то время выполнения запросов одинаковое, так как ClickHouse применяет `WHERE` к подзапросу сразу при его обработке.

Если `enable_optimize_predicate_expression = 0`, то время выполнения второго запроса намного больше, потому что секция `WHERE` применяется к данным уже после завершения подзапроса.

fallback_to_stale_replicas_for_distributed_queries

Форсирует запрос в устаревшую реплику в случае, если актуальные данные недоступны. Смотрите "Репликация".

Из устаревших реплик таблицы ClickHouse выбирает наиболее актуальную.

Используется при выполнении `SELECT` из распределенной таблицы, которая указывает на реплицированные таблицы.

По умолчанию - 1 (включена).

force_index_by_date

Запрещает выполнение запросов, если использовать индекс по дате невозможно.

Работает с таблицами семейства MergeTree.

При `force_index_by_date=1` ClickHouse проверяет, есть ли в запросе условие на ключ даты, которое может использоваться для отсечения диапазонов данных. Если подходящего условия нет - кидается исключение. При этом не проверяется, действительно ли условие уменьшает объём данных для чтения. Например, условие `Date != '2000-01-01'` подходит даже в том случае, когда соответствует всем данным в таблице (т.е. для выполнения запроса требуется full scan). Подробнее про диапазоны данных в таблицах MergeTree читайте в разделе "MergeTree".

force_primary_key

Запрещает выполнение запросов, если использовать индекс по первичному ключу невозможно.

Работает с таблицами семейства MergeTree.

При `force_primary_key=1` ClickHouse проверяет, есть ли в запросе условие на первичный ключ, которое может использоваться для отсечения диапазонов данных. Если подходящего условия нет - кидается исключение. При этом не проверяется, действительно ли условие уменьшает объём данных для чтения. Подробнее про диапазоны данных в таблицах MergeTree читайте в разделе "MergeTree".

fsync_metadata

Включает или отключает fsync при записи `.sql` файлов. По умолчанию включено.

Имеет смысл выключать, если на сервере миллионы мелких таблиц-чанков, которые постоянно создаются и уничтожаются.

enable_http_compression

Включает или отключает сжатие данных в ответе на HTTP-запрос.

Для получения дополнительной информации, читайте Описание интерфейса HTTP.

Возможные значения:

- 0 — выключена.
- 1 — включена.

Значение по умолчанию: 0.

http_zlib_compression_level

Задает уровень сжатия данных в ответе на HTTP-запрос, если enable_http_compression = 1.

Возможные значения: числа от 1 до 9.

Значение по умолчанию: 3.

http_native_compression_disable_checksumming_on_decompress

Включает или отключает проверку контрольной суммы при распаковке данных HTTP POST от клиента. Используется только для собственного (Navite) формата сжатия ClickHouse (ни gzip, ни deflate).

Для получения дополнительной информации, читайте Описание интерфейса HTTP.

Возможные значения:

- 0 — выключена.
- 1 — включена.

Значение по умолчанию: 0.

input_format_allow_errors_num

Устанавливает максимальное количество допустимых ошибок при чтении из текстовых форматов (CSV, TSV и т.п.).

Значение по умолчанию: 0.

Используйте обязательно в паре с input_format_allow_errors_ratio. Для пропуска ошибок, значения обеих настроек должны быть больше 0.

Если при чтении строки возникла ошибка, но при этом счетчик ошибок меньше input_format_allow_errors_num, то ClickHouse игнорирует строку и переходит к следующей.

В случае превышения input_format_allow_errors_num ClickHouse генерирует исключение.

input_format_allow_errors_ratio

Устанавливает максимальную долю допустимых ошибок при чтении из текстовых форматов (CSV, TSV и т.п.).
Доля ошибок задаётся в виде числа с плавающей запятой от 0 до 1.

Значение по умолчанию: 0.

Используйте обязательно в паре с input_format_allow_errors_num. Для пропуска ошибок, значения обеих настроек должны быть больше 0.

Если при чтении строки возникла ошибка, но при этом текущая доля ошибок меньше input_format_allow_errors_ratio, то ClickHouse игнорирует строку и переходит к следующей.

В случае превышения input_format_allow_errors_ratio ClickHouse генерирует исключение.

input_format_values_interpret_expressions

Включает или отключает парсер SQL, если потоковый парсер не может проанализировать данные. Этот параметр используется только для формата Values при вводе данных. Дополнительные сведения о парсерах читайте в разделе Синтаксис.

Возможные значения:

- 0 — выключена.

  В этом случае необходимо вставлять форматированные данные. Смотрите раздел Форматы.

- 1 — включена.

  В этом случае вы можете использовать выражение SQL в качестве значения, но вставка данных намного медленнее. Если вы вставляете только форматированные данные, ClickHouse ведет себя так, как будто значение параметра равно 0.

Значение по умолчанию: 1.

Пример использования

Вставим значение типа DateTime при разных значения настройки.

```
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.
```

Последний запрос эквивалентен следующему:

```
SET input_format_values_interpret_expressions = 0;
INSERT INTO datetime_t SELECT now()

Ok.
```

input_format_defaults_for_omitted_fields

Включает/выключает расширенный обмен данными между клиентом ClickHouse и сервером ClickHouse. Параметр применяется для запросов INSERT.

При выполнении запросаINSERT, клиент ClickHouse подготавливает данные и отправляет их на сервер для записи. При подготовке данных клиент получает структуру таблицы от сервера. В некоторых случаях клиенту требуется больше информации, чем сервер отправляет по умолчанию. Включите расширенный обмен данными с помощью настройки input_format_defaults_for_omitted_fields = 1.

Если расширенный обмен данными включен, сервер отправляет дополнительные метаданные вместе со структурой таблицы. Состав метаданных зависит от операции.

Операции, для которых может потребоваться включить расширенный обмен данными:

- Вставка данных в формате JSONEachRow.

Для всех остальных операций ClickHouse не применяет этот параметр.

Примечание

Функциональность расширенного обмена данными потребляет дополнительные вычислительные ресурсы на сервере и может снизить производительность.

Возможные значения

- 0 — выключена.
- 1 — включена.

Значение по умолчанию: 0.

join_default_strictness

Устанавливает строгость по умолчанию для JOIN.

Возможные значения

- ALL — если в правой таблице несколько совпадающих строк, данные умножаются на количество этих строк. Это нормальное поведение JOIN как в стандартном SQL.
- ANY — если в правой таблице несколько соответствующих строк, то соединяется только первая найденная. Если в "правой" таблице есть не более одной подходящей строки, то результаты ANY и ALL совпадают.
- Пустая строка — если ALL или ANY не указаны в запросе, то ClickHouse генерирует исключение.

Значение по умолчанию: ALL

join_use_nulls

Устанавливает тип поведения JOIN. При объединении таблиц могут появиться пустые ячейки. ClickHouse заполняет их по-разному в зависимости от настроек.

Возможные значения

- 0 — пустые ячейки заполняются значением по умолчанию соответствующего типа поля.
- 1 — JOIN ведет себя как в стандартном SQL. Тип соответствующего поля преобразуется в Nullable, а пустые ячейки заполняются значениями NULL.

Значение по умолчанию: 0.

max_block_size

Данные в ClickHouse обрабатываются по блокам (наборам кусочков столбцов). Внутренние циклы обработки для одного блока достаточно эффективны, но есть заметные издержки на каждый блок. Настройка max_block_size — это рекомендация, какой размер блока (в количестве строк) загружать из таблиц. Размер блока не должен быть слишком маленьким, чтобы затраты на каждый блок были заметны, но не слишком велики, чтобы запрос с LIMIT, который завершается после первого блока, обрабатывался быстро. Цель состоит в том, чтобы не использовалось слишком много оперативки при вынимании большого количества столбцов в несколько потоков; чтобы оставалась хоть какая-нибудь кэш-локальность.

Значение по умолчанию: 65,536.

Из таблицы не всегда загружаются блоки размера max_block_size. Если ясно, что нужно прочитать меньше данных, то будет считан блок меньшего размера.

preferred_block_size_bytes

Служит для тех же целей что и max_block_size, но задает рекомедуемый размер блоков в байтах, выбирая адаптивное количество строк в блоке.
При этом размер блока не может быть более max_block_size строк.
По умолчанию: 1,000,000. Работает только при чтении из MergeTree-движков.

merge_tree_uniform_read_distribution

При чтении из таблиц MergeTree* ClickHouse использует несколько потоков. Этот параметр включает/выключает равномерное распределение заданий по рабочим потокам. Алгоритм равномерного распределения стремится сделать время выполнения всех потоков примерно равным для одного запроса SELECT.

Возможные значения

- 0 — не использовать равномерное распределение заданий на чтение.
- 1 — использовать равномерное распределение заданий на чтение.

Значение по умолчанию: 1.

merge_tree_min_rows_for_concurrent_read

Если количество строк, считываемых из файла таблицы MergeTree* превышает merge_tree_min_rows_for_concurrent_read, то ClickHouse пытается выполнить одновременное чтение из этого файла в несколько потоков.

Возможные значения

Любое положительное целое число.

Значение по умолчанию: 163840.

merge_tree_min_rows_for_seek

Если расстояние между двумя блоками данных для чтения в одном файле меньше, чем merge_tree_min_rows_for_seek строк, то ClickHouse не перескакивает через блоки, а считывает данные последовательно.

Возможные значения

Любое положительное целое число.

Значение по умолчанию: 0.

merge_tree_coarse_index_granularity

При поиске данных ClickHouse проверяет засечки данных в файле индекса. Если ClickHouse обнаруживает, что требуемые ключи находятся в некотором диапазоне, он делит этот диапазон на merge_tree_coarse_index_granularity поддиапазонов и выполняет в них рекурсивный поиск нужных ключей.

Возможные значения

Любое положительное целое число.

Значение по умолчанию: 8.

merge_tree_max_rows_to_use_cache

Если требуется прочитать более, чем merge_tree_max_rows_to_use_cache строк в одном запросе, ClickHouse не используют кэш несжатых блоков. Настройка сервера uncompressed_cache_size определяет размер кэша несжатых блоков.

Возможные значения

Любое положительное целое число.

Значение по умолчанию: 1048576.

min_bytes_to_use_direct_io

Минимальный объем данных, необходимый для прямого (небуферизованного) чтения/записи (direct I/O) на диск.

ClickHouse использует этот параметр при чтении данных из таблиц. Если общий объем хранения всех данных для чтения превышает min_bytes_to_use_direct_io байт, тогда ClickHouse использует флаг O_DIRECT при чтении данных с диска.

Возможные значения

- 0 — прямой ввод-вывод отключен.
- Положительное целое число.

Значение по умолчанию: 0.

log_queries

Установка логгирования запроса.

Запросы, переданные в ClickHouse с этой установкой, логгируются согласно правилам конфигурационного параметра сервера query_log.

Пример :

```
log_queries=1
```

max_insert_block_size

Формировать блоки указанного размера, при вставке в таблицу.
Эта настройка действует только в тех случаях, когда сервер сам формирует такие блоки.
Например, при INSERT-е через HTTP интерфейс, сервер парсит формат данных, и формирует блоки указанного размера.
А при использовании clickhouse-client, клиент сам парсит данные, и настройка max_insert_block_size на сервере не влияет на размер вставляемых блоков.
При использовании INSERT SELECT, настройка так же не имеет смысла, так как данные будут вставляться теми блоками, которые вышли после SELECT-а.

Значение по умолчанию: 1,048,576.

Это значение намного больше, чем max_block_size. Это сделано, потому что некоторые движки таблиц (*MergeTree) будут на каждый вставляемый блок формировать кусок данных на диске, что является довольно большой сущностью. Также, в таблицах типа *MergeTree, данные сортируются при вставке, и достаточно большой размер блока позволяет отсортировать больше данных в оперативке.

max_replica_delay_for_distributed_queries

Отключает отстающие реплики при распределенных запросах. Смотрите "Репликация".

Устанавливает время в секундах. Если оставание реплики больше установленного значения, то реплика не используется.

Значение по умолчанию: 300.

Используется при выполнении SELECT из распределенной таблицы, которая указывает на реплицированные таблицы.

max_threads

Максимальное количество потоков обработки запроса без учёта потоков для чтения данных с удалённых серверов (смотрите параметр max_distributed_connections).

Этот параметр относится к потокам, которые выполняют параллельно одни стадии конвейера выполнения запроса.
Например, при чтении из таблицы, если есть возможность вычислять выражения с функциями, фильтровать с помощью WHERE и предварительно агрегировать для GROUP BY параллельно, используя хотя бы количество потоков max_threads, то используются max_threads.

Значение по умолчанию: 2.

Если на сервере обычно исполняется менее одного запроса SELECT одновременно, то выставите этот параметр в значение чуть меньше количества реальных процессорных ядер.

Для запросов, которые быстро завершаются из-за LIMIT-а, имеет смысл выставить max_threads поменьше. Например, если нужное количество записей находится в каждом блоке, то при max_threads = 8 будет считано 8 блоков, хотя достаточно было прочитать один.

Чем меньше max_threads, тем меньше будет использоваться оперативки.

max_compress_block_size

Максимальный размер блоков не сжатых данных перед сжатием при записи в таблицу. По умолчанию - 1 048 576 (1 MiB). При уменьшении размера, незначительно уменьшается коэффициент сжатия, незначительно возрастает скорость сжатия и разжатия за счёт кэш-локальности, и уменьшается потребление оперативки. Как правило, не имеет смысла менять эту настройку.

Не путайте блоки для сжатия (кусок памяти, состоящий из байт) и блоки для обработки запроса (пачка строк из таблицы).

min_compress_block_size

Для таблиц типа "MergeTree". В целях уменьшения задержек при обработке запросов, блок сжимается при записи следующей засечки, если его размер не меньше min_compress_block_size. По умолчанию - 65 536.

Реальный размер блока, если несжатых данных меньше max_compress_block_size, будет не меньше этого значения и не меньше объёма данных на одну засечку.

Рассмотрим пример. Пусть index_granularity, указанная при создании таблицы - 8192.

Пусть мы записываем столбец типа UInt32 (4 байта на значение). При записи 8192 строк, будет всего 32 КБ данных. Так как min_compress_block_size = 65 536, сжатый блок будет сформирован на каждые две засечки.

Пусть мы записываем столбец URL типа String (средний размер - 60 байт на значение). При записи 8192 строк, будет, в среднем, чуть меньше 500 КБ данных. Так как это больше 65 536 строк, то сжатый блок будет сформирован на каждую засечку. В этом случае, при чтении с диска данных из диапазона в одну засечку, не будет разжато лишних данных.

Как правило, не имеет смысла менять эту настройку.

max_query_size

Максимальный кусок запроса, который будет считан в оперативку для разбора парсером языка SQL. Запрос INSERT также содержит данные для INSERT-а, которые обрабатываются отдельным, потоковым парсером (расходующим O(1) оперативки), и не учитываются в этом ограничении.

Значение по умолчанию: 256 Кб.

interactive_delay

Интервал в микросекундах для проверки, не запрошена ли остановка выполнения запроса, и отправки прогресса.

Значение по умолчанию: 100,000 (проверять остановку запроса и отправлять прогресс десять раз в секунду).

connect_timeout, receive_timeout, send_timeout

Таймауты в секундах на сокет, по которому идёт общение с клиентом.

Значение по умолчанию: 10, 300, 300.

poll_interval

Блокироваться в цикле ожидания запроса в сервере на указанное количество секунд.

Значение по умолчанию: 10.

max_distributed_connections

Максимальное количество одновременных соединений с удалёнными серверами при распределённой обработке одного запроса к одной таблице типа Distributed. Рекомендуется выставлять не меньше, чем количество серверов в кластере.

Значение по умолчанию: 1024.

Следующие параметры имеют значение только на момент создания таблицы типа Distributed (и при запуске сервера), поэтому их не имеет смысла менять в рантайме.

distributed_connections_pool_size

Максимальное количество одновременных соединений с удалёнными серверами при распределённой обработке всех запросов к одной таблице типа Distributed. Рекомендуется выставлять не меньше, чем количество серверов в кластере.

Значение по умолчанию: 1024.

connect_timeout_with_failover_ms

Таймаут в миллисекундах на соединение с удалённым сервером, для движка таблиц Distributed, если используются секции shard и replica в описании кластера.
В случае неуспеха, делается несколько попыток соединений с разными репликами.

Значение по умолчанию: 50.

connections_with_failover_max_tries

Максимальное количество попыток соединения с каждой репликой, для движка таблиц Distributed.

Значение по умолчанию: 3.

extremes

Считать ли экстремальные значения (минимумы и максимумы по столбцам результата запроса).
Принимает 0 или 1. По умолчанию - 0 (выключено).
Подробнее смотрите раздел "Экстремальные значения".

use_uncompressed_cache

Использовать ли кэш разжатых блоков. Принимает 0 или 1. По умолчанию - 0 (выключено).

Использование кэша несжатых блоков (только для таблиц семейства MergeTree) может существенно сократить задержку и увеличить пропускную способность при работе с большим количеством коротких запросов. Включите эту настройку для пользователей, от которых идут частые короткие запросы. Также обратите внимание на конфигурационный параметр uncompressed_cache_size (настраивается только в конфигурационном файле) - размер кэша разжатых блоков. По умолчанию - 8 GiB. Кэш разжатых блоков заполняется по мере надобности, а наиболее невостребованные данные автоматически удаляются.

Для запросов, читающих хоть немного приличный объём данных (миллион строк и больше), кэш разжатых блоков автоматически выключается, чтобы оставить место для действительно мелких запросов. Поэтому, можно держать настройку use_uncompressed_cache всегда выставленной в 1.

replace_running_query

При использовании интерфейса HTTP может быть передан параметр query_id. Это любая строка, которая служит идентификатором запроса.
Если в этот момент, уже существует запрос от того же пользователя с тем же query_id, то поведение определяется параметром replace_running_query.

0 - (по умолчанию) кинуть исключение (не давать выполнить запрос, если запрос с таким же query_id уже выполняется);

1 - отменить старый запрос и начать выполнять новый.

Эта настройка, выставленная в 1, используется в Яндекс.Метрике для реализации suggest-а значений для условий сегментации. После ввода очередного символа, если старый запрос ещё не выполнился, его следует отменить.

schema

Параметр применяется в том случае, когда используются форматы, требующие определения схемы, например Cap'n Proto. Значение параметра зависит от формата.

stream_flush_interval_ms

Работает для таблиц со стриммингом в случае тайм-аута, или когда поток генерирует max_insert_block_size строк.

Значение по умолчанию - 7500.

Чем меньше значение, тем чаще данные сбрасываются в таблицу. Установка слишком низкого значения приводит к снижению производительности.

load_balancing

На какие реплики (среди живых реплик) предпочитать отправлять запрос (при первой попытке) при распределённой обработке запроса.

random (по умолчанию)

Для каждой реплики считается количество ошибок. Запрос отправляется на реплику с минимальным числом ошибок, а если таких несколько, то на случайную из них.
Недостатки: не учитывается близость серверов; если на репликах оказались разные данные, то вы будете получать так же разные данные.

nearest_hostname

Для каждой реплики считается количество ошибок. Каждые 5 минут, число ошибок целочисленно делится на 2. Таким образом, обеспечивается расчёт числа ошибок за недавнее время с экспоненциальным сглаживанием. Если есть одна реплика с минимальным числом ошибок (то есть, на других репликах недавно были ошибки) - запрос отправляется на неё. Если есть несколько реплик с одинаковым минимальным числом ошибок, то запрос отправляется на реплику, имя хоста которой в конфигурационном файле минимально отличается от имени хоста сервера (по количеству отличающихся символов на одинаковых позициях, до минимальной длины обеих имён хостов).

Для примера, example01-01-1 и example01-01-2.yandex.ru отличаются в одной позиции, а example01-01-1 и example01-02-2 - в двух.
Этот метод может показаться примитивным, но он не требует внешних данных о топологии сети и не сравнивает IP-адреса, что было бы сложно для наших IPv6-адресов.

Таким образом, если есть равнозначные реплики, предпочитается ближайшая по имени.
Также можно сделать предположение, что при отправке запроса на один и тот же сервер, в случае отсутствия сбоев, распределённый запрос будет идти тоже на одни и те же серверы. То есть, даже если на репликах расположены разные данные, запрос будет возвращать в основном одинаковые результаты.

in_order

Реплики перебираются в таком порядке, в каком они указаны. Количество ошибок не имеет значения.
Этот способ подходит для тех случаев, когда вы точно знаете, какая реплика предпочтительнее.

totals_mode

Каким образом вычислять TOTALS при наличии HAVING, а также при наличии max_rows_to_group_by и group_by_overflow_mode = 'any'.
Смотрите раздел "Модификатор WITH TOTALS".

totals_auto_threshold

Порог для totals_mode = 'auto'.
Смотрите раздел "Модификатор WITH TOTALS".

max_parallel_replicas

Максимальное количество используемых реплик каждого шарда при выполнении запроса.
Для консистентности (чтобы получить разные части одного и того же разбиения), эта опция работает только при заданном ключе сэмплирования.
Отставание реплик не контролируется.

compile

Включить компиляцию запросов. По умолчанию - 0 (выключено).

Компиляция предусмотрена только для части конвейера обработки запроса - для первой стадии агрегации (GROUP BY).
В случае, если эта часть конвейера была скомпилирована, запрос может работать быстрее, за счёт разворачивания коротких циклов и инлайнинга вызовов агрегатных функций. Максимальный прирост производительности (до четырёх раз в редких случаях) достигается на запросах с несколькими простыми агрегатными функциями. Как правило, прирост производительности незначителен. В очень редких случаях возможно замедление выполнения запроса.

min_count_to_compile

После скольких раз, когда скомпилированный кусок кода мог пригодиться, выполнить его компиляцию. По умолчанию - 3.
Для тестирования можно установить значение 0: компиляция выполняется синхронно, и запрос ожидает окончания процесса компиляции перед продолжением выполнения. Во всех остальных случаях используйте значения, начинающиеся с 1. Как правило, компиляция занимает по времени около 5-10 секунд.
В случае, если значение равно 1 или больше, компиляция выполняется асинхронно, в отдельном потоке. При готовности результата, он сразу же будет использован, в том числе, уже выполняющимися в данный момент запросами.

Скомпилированный код требуется для каждого разного сочетания используемых в запросе агрегатных функций и вида ключей в GROUP BY.
Результаты компиляции сохраняются в директории build в виде .so файлов. Количество результатов компиляции не ограничено, так как они не занимают много места. При перезапуске сервера, старые результаты будут использованы, за исключением случая обновления сервера - тогда старые результаты удаляются.

input_format_skip_unknown_fields

Если значение равно true, то при выполнении INSERT входные данные из столбцов с неизвестными именами будут пропущены. В противном случае эта ситуация создаст исключение.
Работает для форматов JSONEachRow и TSKV.

output_format_json_quote_64bit_integers

Если значение истинно, то при использовании JSON* форматов UInt64 и Int64 числа выводятся в кавычках (из соображений совместимости с большинством реализаций JavaScript), иначе - без кавычек.

format_csv_delimiter

Символ, интерпретируемый как разделитель в данных формата CSV. По умолчанию — ,.

insert_quorum

Включает кворумную запись.

- Если insert_quorum < 2, то кворумная запись выключена.
- Если insert_quorum >= 2, то кворумная запись включена.

Значение по умолчанию: 0.

Кворумная запись

INSERT завершается успешно только в том случае, когда ClickHouse смог без ошибки записать данные в insert_quorum реплик за время insert_quorum_timeout. Если по любой причине количество реплик с успешной записью не достигнет insert_quorum, то запись считается не состоявшейся и ClickHouse удалит вставленный блок из всех реплик, куда уже успел записать данные.

Все реплики в кворуме консистентны, т.е. содержат данные всех более ранних запросов INSERT. Последовательность INSERT линеаризуется.

При чтении данных, записанных с insert_quorum можно использовать настройку select_sequential_consistency.

ClickHouse генерирует исключение

- Если количество доступных реплик на момент запроса меньше insert_quorum.
- При попытке записать данные в момент, когда предыдущий блок ещё не вставлен в insert_quorum реплик. Эта ситуация может возникнуть, если пользователь вызвал INSERT прежде, чем завершился предыдущий с insert_quorum.

См. также параметры:

- insert_quorum_timeout
- select_sequential_consistency

insert_quorum_timeout

Время ожидания кворумной записи в секундах. Если время прошло, а запись так не состоялась, то ClickHouse сгенерирует исключение и клиент должен повторить запрос на запись того же блока на эту же или любую другую реплику.

Значение по умолчанию: 60 секунд.

См. также параметры:

- insert_quorum
- select_sequential_consistency

select_sequential_consistency

Включает или выключает последовательную консистентность для запросов SELECT.

Возможные значения:

- 0 — выключена.
- 1 — включена.

Значение по умолчанию: 0.

Использование

Когда последовательная консистентность включена, то ClickHouse позволит клиенту выполнить запрос SELECT только к тем репликам, которые содержат данные всех предыдущих запросов INSERT, выполненных с insert_quorum. Если клиент обратится к неполной реплике, то ClickHouse сгенерирует исключение. В запросе SELECT не будут участвовать данные, которые ещё не были записаны на кворум реплик.

Смотрите также

- insert_quorum
- insert_quorum_timeout

Профили настроек

Профили настроек - это множество настроек, сгруппированных под одним именем. Для каждого пользователя ClickHouse указывается некоторый профиль.
Все настройки профиля можно применить, установив настройку profile.

Пример:

Установить профиль web.

```
SET profile = 'web'
```

Профили настроек объявляются в конфигурационном файле пользователей. Обычно это users.xml.

Пример:

```
<!-- Settings profiles -->
<profiles>
   <!-- Default settings -->
   <default>
      <!-- The maximum number of threads when running a single query. -->
      <max_threads>8</max_threads>
   </default>

   <!-- Settings for quries from the user interface -->
   <web>
      <max_rows_to_read>1000000000</max_rows_to_read>
      <max_bytes_to_read>100000000000</max_bytes_to_read>

      <max_rows_to_group_by>1000000</max_rows_to_group_by>
      <group_by_overflow_mode>any</group_by_overflow_mode>

      <max_rows_to_sort>1000000</max_rows_to_sort>
      <max_bytes_to_sort>1000000000</max_bytes_to_sort>

      <max_result_rows>100000</max_result_rows>
      <max_result_bytes>100000000</max_result_bytes>
      <result_overflow_mode>break</result_overflow_mode>

      <max_execution_time>600</max_execution_time>
      <min_execution_speed>1000000</min_execution_speed>
      <timeout_before_checking_execution_speed>15</timeout_before_checking_execution_speed>

      <max_columns_to_read>25</max_columns_to_read>
      <max_temporary_columns>100</max_temporary_columns>
      <max_temporary_non_const_columns>50</max_temporary_non_const_columns>

      <max_subquery_depth>2</max_subquery_depth>
      <max_pipeline_depth>25</max_pipeline_depth>
      <max_ast_depth>50</max_ast_depth>
      <max_ast_elements>100</max_ast_elements>

      <readonly>1</readonly>
   </web>
</profiles>
```

В примере задано два профиля: default и web. Профиль default имеет специальное значение - он всегда обязан присутствовать и применяется при запуске сервера. То есть, профиль default содержит настройки по умолчанию. Профиль web - обычный профиль, который может быть установлен с помощью запроса SET или с помощью параметра URL при запросе по HTTP.

Профили настроек могут наследоваться от друг-друга - это реализуется указанием настройки profile перед остальными настройками, перечисленными в профиле.

Утилиты ClickHouse

- clickhouse-local
- clickhouse-copier - копирует (и перешардирует) данные с одного кластера на другой.

clickhouse-copier

Копирует данные из таблиц одного кластера в таблицы другого (или этого же) кластера.

Можно запустить несколько clickhouse-copier для разных серверах для выполнения одного и того же задания. Для синхронизации между процессами используется ZooKeeper.

После запуска, clickhouse-copier:

- Соединяется с ZooKeeper и получает:
  - Задания на копирование.
  - Состояние заданий на копирование.
- Выполняет задания.

  Каждый запущенный процесс выбирает "ближайший" шард исходного кластера и копирует данные в кластер назначения, при необходимости перешардируя их.

clickhouse-copier отслеживает изменения в ZooKeeper и применяет их "на лету".

Для снижения сетевого трафика рекомендуем запускать clickhouse-copier на том же сервере, где находятся исходные данные.

Запуск clickhouse-copier

Утилиту следует запускать вручную следующим образом:

```
clickhouse-copier copier --daemon --config zookeeper.xml --task-path /task/path --base-dir /path/to/dir
```

Параметры запуска:

- daemon - запускает clickhouse-copier в режиме демона.
- config - путь к файлу zookeeper.xml с параметрами соединения с ZooKeeper.
- task-path - путь к ноде ZooKeeper. Нода используется для синхронизации между процессами clickhouse-copier и для хранения заданий. Задания хранятся в $task-path/description.
- task-file - необязательный путь к файлу с описанием конфигурация заданий для загрузки в ZooKeeper.
- task-upload-force - Загрузить task-file в ZooKeeper даже если уже было загружено.
- base-dir - путь к логам и вспомогательным файлам. При запуске clickhouse-copier создает в $base-dir подкаталоги clickhouse-copier_YYYYMMHHSS_<PID>. Если параметр не указан, то каталоги будут создаваться в каталоге, где clickhouse-copier был запущен.

Формат 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>
```

Конфигурация заданий на копирование

```
<yandex>
  <!-- Configuration of clusters as in an ordinary server config -->
  <remote_servers>
    <source_cluster>
      <shard>
        <internal_replication>false</internal_replication>
          <replica>
            <host>127.0.0.1</host>
            <port>9000</port>
          </replica>
      </shard>
      ...
    </source_cluster>
```

```
    <destination_cluster>
    ...
    </destination_cluster>
  </remote_servers>

  <!-- How many simultaneously active workers are possible. If you run more workers superfluous workers will sleep. -->
  <max_workers>2</max_workers>

  <!-- Setting used to fetch (pull) data from source cluster tables -->
  <settings_pull>
    <readonly>1</readonly>
  </settings_pull>

  <!-- Setting used to insert (push) data to destination cluster tables -->
  <settings_push>
    <readonly>0</readonly>
  </settings_push>

  <!-- Common setting for fetch (pull) and insert (push) operations. Also, copier process context uses it.
     They are overlaid by <settings_pull/> and <settings_push/> respectively. -->
  <settings>
    <connect_timeout>3</connect_timeout>
    <!-- Sync insert is set forcibly, leave it here just in case. -->
    <insert_distributed_sync>1</insert_distributed_sync>
  </settings>

  <!-- Copying tasks description.
     You could specify several table task in the same task description (in the same ZooKeeper node), they will be performed
     sequentially.
  -->
  <tables>
    <!-- A table task, copies one table. -->
    <table_hits>
      <!-- Source cluster name (from <remote_servers/> section) and tables in it that should be copied -->
      <cluster_pull>source_cluster</cluster_pull>
      <database_pull>test</database_pull>
      <table_pull>hits</table_pull>

      <!-- Destination cluster name and tables in which the data should be inserted -->
      <cluster_push>destination_cluster</cluster_push>
      <database_push>test</database_push>
      <table_push>hits2</table_push>

      <!-- Engine of destination tables.
         If destination tables have not be created, workers create them using columns definition from source tables and engine
         definition from here.

         NOTE: If the first worker starts insert data and detects that destination partition is not empty then the partition will
         be dropped and refilled, take it into account if you already have some data in destination tables. You could directly
         specify partitions that should be copied in <enabled_partitions/>, they should be in quoted format like partition column
of
         system.parts table.
      -->
      <engine>
      ENGINE=ReplicatedMergeTree('/clickhouse/tables/{cluster}/{shard}/hits2', '{replica}')
      PARTITION BY toMonday(date)
      ORDER BY (CounterID, EventDate)
      </engine>

      <!-- Sharding key used to insert data to destination cluster -->
      <sharding_key>jumpConsistentHash(intHash64(UserID), 2)</sharding_key>

      <!-- Optional expression that filter data while pull them from source servers -->
      <where_condition>CounterID != 0</where_condition>
```

```
        <!-- This section specifies partitions that should be copied, other partition will be ignored.
             Partition names should have the same format as
             partition column of system.parts table (i.e. a quoted text).
             Since partition key of source and destination cluster could be different,
             these partition names specify destination partitions.

             NOTE: In spite of this section is optional (if it is not specified, all partitions will be copied),
             it is strictly recommended to specify them explicitly.
             If you already have some ready paritions on destination cluster they
             will be removed at the start of the copying since they will be interpeted
             as unfinished data from the previous copying!!!
        -->
        <enabled_partitions>
           <partition>'2018-02-26'</partition>
           <partition>'2018-03-05'</partition>
           ...
        </enabled_partitions>
      </table_hits>

      <!-- Next table to copy. It is not copied until previous table is copying. -->
      </table_visits>
      ...
      </table_visits>
      ...
    </tables>
</yandex>
```

clickhouse-copier отслеживает изменения /task/path/description и применяет их "на лету". Если вы поменяете, например, значение max_workers, то количество процессов, выполняющих задания, также изменится.

clickhouse-local

Принимает на вход данные, которые можно представить в табличном виде и выполняет над ними операции, заданные на языке запросов ClickHouse.

clickhouse-local использует движок сервера ClickHouse, т.е. поддерживает все форматы данных и движки таблиц, с которыми работает ClickHouse, при этом для выполнения операций не требуется запущенный сервер.

clickhouse-local при настройке по умолчанию не имеет доступа к данным, которыми управляет сервер ClickHouse, установленный на этом же хосте, однако можно подключить конфигурацию сервера с помощью ключа --config-file.

Warning

Мы не рекомендуем подключать серверную конфигурацию к clickhouse-local, поскольку данные можно легко повредить неосторожными действиями.

Вызов программы

Основной формат вызова:

```
clickhouse-local --structure "table_structure" --input-format "format_of_incoming_data" -q "query"
```

Ключи команды:

- -S, --structure — структура таблицы, в которую будут помещены входящие данные.
- -if, --input-format — формат входящих данных. По умолчанию — TSV.
- -f, --file — путь к файлу с данными. По умолчанию — stdin.
- -q, --query — запросы на выполнение. Разделитель запросов — ;.
- -N, --table — имя таблицы, в которую будут помещены входящие данные. По умолчанию - table.
- -of, --format, --output-format — формат выходных данных. По умолчанию — TSV.

- --stacktrace — вывод отладочной информации при исключениях.
- --verbose — подробный вывод при выполнении запроса.
- -s — отключает вывод системных логов в stderr.
- --config-file — путь к файлу конфигурации. По умолчанию clickhouse-local запускается с пустой конфигурацией. Конфигурационный файл имеет тот же формат, что и для сервера ClickHouse и в нём можно использовать все конфигурационные параметры сервера. Обычно подключение конфигурации не требуется, если требуется установить отдельный параметр, то это можно сделать ключом с именем параметра.
- --help — вывод справочной информации о clickhouse-local.

Примеры вызова

```
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
```

Вызов выше эквивалентен следующему:

```
$ 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
```

А теперь давайте выведем на экран объем оперативной памяти, занимаемой пользователями (Unix):

```
$ 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 │
├──────────┼──────────┤
...
```

Общие вопросы

Почему бы не использовать системы типа MapReduce?

Системами типа MapReduce будем называть системы распределённых вычислений, в которых операция reduce сделана на основе распределённой сортировки. Наиболее распространённым opensource решением данного класса является Apache Hadoop, а в Яндексе используется внутрення разработка — YT.

Такие системы не подходят для онлайн запросов в силу слишком большой latency. То есть, не могут быть использованы в качестве бэкенда для веб-интерфейса.
Такие системы не подходят для обновления данных в реальном времени.
Распределённая сортировка не является оптимальным способом выполнения операции reduce, если результат выполнения операции и все промежуточные результаты, при их наличии, помещаются в оперативку на одном сервере, как обычно бывает в запросах, выполняющихся в режиме онлайн. В таком случае, оптимальным способом выполнения операции reduce является хэш-таблица. Частым способом оптимизации map-reduce задач является предагрегация (частичный reduce) с использованием хэш-таблицы в оперативной памяти. Эта оптимизация делается пользователем в ручном режиме.
Распределённая сортировка является основной причиной тормозов при выполнении несложных map-reduce задач.

Большинство реализаций MapReduce позволяют выполнять произвольный код на кластере. Но для OLAP задач лучше подходит декларативный язык запросов, который позволяет быстро проводить исследования. Для примера, для Hadoop существует Hive и Pig. Также смотрите Cloudera Impala, Shark (устаревший) для Spark, а также Spark SQL, Presto, Apache Drill. Впрочем, производительность при выполнении таких задач является сильно неоптимальной по сравнению со специализированными системами, а сравнительно высокая latency не позволяет использовать эти системы в качестве бэкенда для веб-интерфейса.

ClickHouse Development

Overview of ClickHouse Architecture

ClickHouse is a true column-oriented DBMS. Data is stored by columns, and during the execution of arrays (vectors or chunks of columns). Whenever possible, operations are dispatched on arrays, rather than on individual values. This is called "vectorized query execution," and it helps lower the cost of actual data processing.

> This idea is nothing new. It dates back to the APL programming language and its descendants: A +, J, K, and Q. Array programming is used in scientific data processing. Neither is this idea something new in relational databases: for example, it is used in the Vectorwise system.

There are two different approaches for speeding up the query processing: vectorized query execution and runtime code generation. In the latter, the code is generated for every kind of query on the fly, removing all indirection and dynamic dispatch. Neither of these approaches is strictly better than the other. Runtime code generation can be better when it fuses many operations together, thus fully utilizing CPU execution units and the pipeline. Vectorized query execution can be less practical, because it involves temporary vectors that must be written to the cache and read back. If the temporary data does not fit in the L2 cache, this becomes an issue. But vectorized query execution more easily utilizes the SIMD capabilities of the CPU. A research paper written by our friends shows that it is better to combine both approaches. ClickHouse uses vectorized query execution and has limited initial support for runtime code generation.

Columns

To represent columns in memory (actually, chunks of columns), the IColumn interface is used. This interface provides helper methods for implementation of various relational operators. Almost all operations are immutable: they do not modify the original column, but create a new modified one. For example, the IColumn :: filter method accepts a filter byte mask. It is used for the WHERE and HAVING relational operators. Additional examples: the IColumn :: permute method to support ORDER BY, the IColumn :: cut method to support LIMIT, and so on.

Various IColumn implementations (ColumnUInt8, ColumnString and so on) are responsible for the memory layout of columns. Memory layout is usually a contiguous array. For the integer type of columns it is just one contiguous array, like std :: vector. For String and Array columns, it is two vectors: one for all array elements, placed contiguously, and a second one for offsets to the beginning of each array. There is also ColumnConst that stores just one value in memory, but looks like a column.

Field

Nevertheless, it is possible to work with individual values as well. To represent an individual value, the Field is used. Field is just a discriminated union of UInt64, Int64, Float64, String and Array. IColumn has the operator[] method to get the n-th value as a Field, and the insert method to append a Field to the end of a column. These methods are not very efficient, because they require dealing with temporary Field objects representing an individual value. There are more efficient methods, such as insertFrom, insertRangeFrom, and so on.

Field doesn't have enough information about a specific data type for a table. For example, UInt8, UInt16, UInt32, and UInt64 are all represented as UInt64 in a Field.

Leaky Abstractions

IColumn has methods for common relational transformations of data, but they don't meet all needs. For example, ColumnUInt64 doesn't have a method to calculate the sum of two columns, and ColumnString doesn't have a method to run a substring search. These countless routines are implemented outside of IColumn.

Various functions on columns can be implemented in a generic, non-efficient way using IColumn methods to extract Field values, or in a specialized way using knowledge of inner memory layout of data in a specific IColumn implementation. To do this, functions are cast to a specific IColumn type and deal with internal representation directly. For example, ColumnUInt64 has the getData method that returns a reference to an internal array, then a separate routine reads or fills that array directly. In fact, we have "leaky abstractions" to allow efficient specializations of various routines.

Data Types

IDataType is responsible for serialization and deserialization: for reading and writing chunks of columns or individual values in binary or text form.
IDataType directly corresponds to data types in tables. For example, there are DataTypeUInt32, DataTypeDateTime, DataTypeString and so on.

IDataType and IColumn are only loosely related to each other. Different data types can be represented in memory by the same IColumn implementations. For example, DataTypeUInt32 and DataTypeDateTime are both represented by ColumnUInt32 or ColumnConstUInt32. In addition, the same data type can be represented by different IColumn implementations. For example, DataTypeUInt8 can be represented by ColumnUInt8 or ColumnConstUInt8.

IDataType only stores metadata. For instance, DataTypeUInt8 doesn't store anything at all (except vptr) and DataTypeFixedString stores just N (the size of fixed-size strings).

IDataType has helper methods for various data formats. Examples are methods to serialize a value with possible quoting, to serialize a value for JSON, and to serialize a value as part of XML format. There is no direct correspondence to data formats. For example, the different data formats Pretty and TabSeparated can use the same serializeTextEscaped helper method from the IDataType interface.

Block

A Block is a container that represents a subset (chunk) of a table in memory. It is just a set of triples: (IColumn, IDataType, column name). During query execution, data is processed by Blocks. If we have a Block, we have data (in the IColumn object), we have information about its type (in IDataType) that tells us how to deal with that column, and we have the column name (either the original column name from the table, or some artificial name assigned for getting temporary results of calculations).

When we calculate some function over columns in a block, we add another column with its result to the block, and we don't touch columns for arguments of the function because operations are immutable. Later, unneeded columns can be removed from the block, but not modified. This is convenient for elimination of common subexpressions.

Blocks are created for every processed chunk of data. Note that for the same type of calculation, the column names and types remain the same for different blocks, and only column data changes. It is better to split block data from the block header, because small block sizes will have a high overhead of temporary strings for copying shared_ptrs and column names.

Block Streams

Block streams are for processing data. We use streams of blocks to read data from somewhere, perform data transformations, or write data to somewhere. IBlockInputStream has the read method to fetch the next block while available. IBlockOutputStream has the write method to push the block somewhere.

Streams are responsible for:

1. Reading or writing to a table. The table just returns a stream for reading or writing blocks.
2. Implementing data formats. For example, if you want to output data to a terminal in Pretty format, you create a block output stream where you push blocks, and it formats them.

3. Performing data transformations. Let's say you have IBlockInputStream and want to create a filtered stream. You create FilterBlockInputStream and initialize it with your stream. Then when you pull a block from FilterBlockInputStream, it pulls a block from your stream, filters it, and returns the filtered block to you. Query execution pipelines are represented this way.

There are more sophisticated transformations. For example, when you pull from AggregatingBlockInputStream, it reads all data from its source, aggregates it, and then returns a stream of aggregated data for you. Another example: UnionBlockInputStream accepts many input sources in the constructor and also a number of threads. It launches multiple threads and reads from multiple sources in parallel.

> Block streams use the "pull" approach to control flow: when you pull a block from the first stream, it consequently pulls the required blocks from nested streams, and the entire execution pipeline will work. Neither "pull" nor "push" is the best solution, because control flow is implicit, and that limits implementation of various features like simultaneous execution of multiple queries (merging many pipelines together). This limitation could be overcome with coroutines or just running extra threads that wait for each other. We may have more possibilities if we make control flow explicit: if we locate the logic for passing data from one calculation unit to another outside of those calculation units. Read this article for more thoughts.

We should note that the query execution pipeline creates temporary data at each step. We try to keep block size small enough so that temporary data fits in the CPU cache. With that assumption, writing and reading temporary data is almost free in comparison with other calculations. We could consider an alternative, which is to fuse many operations in the pipeline together, to make the pipeline as short as possible and remove much of the temporary data. This could be an advantage, but it also has drawbacks. For example, a split pipeline makes it easy to implement caching intermediate data, stealing intermediate data from similar queries running at the same time, and merging pipelines for similar queries.

Formats

Data formats are implemented with block streams. There are "presentational" formats only suitable for output of data to the client, such as Pretty format, which provides only IBlockOutputStream. And there are input/output formats, such as TabSeparated or JSONEachRow.

There are also row streams: IRowInputStream and IRowOutputStream. They allow you to pull/push data by individual rows, not by blocks. And they are only needed to simplify implementation of row-oriented formats. The wrappers BlockInputStreamFromRowInputStream and BlockOutputStreamFromRowOutputStream allow you to convert row-oriented streams to regular block-oriented streams.

I/O

For byte-oriented input/output, there are ReadBuffer and WriteBuffer abstract classes. They are used instead of C++ iostreams. Don't worry: every mature C++ project is using something other than iostreams for good reasons.

ReadBuffer and WriteBuffer are just a contiguous buffer and a cursor pointing to the position in that buffer. Implementations may own or not own the memory for the buffer. There is a virtual method to fill the buffer with the following data (for ReadBuffer) or to flush the buffer somewhere (for WriteBuffer). The virtual methods are rarely called.

Implementations of ReadBuffer/WriteBuffer are used for working with files and file descriptors and network sockets, for implementing compression (CompressedWriteBuffer is initialized with another WriteBuffer and performs compression before writing data to it), and for other purposes – the names ConcatReadBuffer, LimitReadBuffer, and HashingWriteBuffer speak for themselves.

Read/WriteBuffers only deal with bytes. To help with formatted input/output (for instance, to write a number in decimal format), there are functions from ReadHelpers and WriteHelpers header files.

Let's look at what happens when you want to write a result set in JSON format to stdout. You have a result set ready to be fetched from IBlockInputStream. You create WriteBufferFromFileDescriptor(STDOUT_FILENO) to write bytes to stdout. You create JSONRowOutputStream, initialized with that WriteBuffer, to write rows in JSON to stdout. You create BlockOutputStreamFromRowOutputStream on top of it, to represent it as IBlockOutputStream. Then you call copyData to transfer data from IBlockInputStream to IBlockOutputStream, and everything works. Internally, JSONRowOutputStream will write various JSON delimiters and call the IDataType::serializeTextJSON method with a reference to IColumn and the row number as arguments. Consequently, IDataType::serializeTextJSON will call a method from WriteHelpers.h: for example, writeText for numeric types and writeJSONString for DataTypeString.

Tables

Tables are represented by the IStorage interface. Different implementations of that interface are different table engines. Examples are StorageMergeTree, StorageMemory, and so on. Instances of these classes are just tables.

The most important IStorage methods are read and write. There are also alter, rename, drop, and so on. The read method accepts the following arguments: the set of columns to read from a table, the AST query to consider, and the desired number of streams to return. It returns one or multiple IBlockInputStream objects and information about the stage of data processing that was completed inside a table engine during query execution.

In most cases, the read method is only responsible for reading the specified columns from a table, not for any further data processing. All further data processing is done by the query interpreter and is outside the responsibility of IStorage.

But there are notable exceptions:

- The AST query is passed to the read method and the table engine can use it to derive index usage and to read less data from a table.
- Sometimes the table engine can process data itself to a specific stage. For example, StorageDistributed can send a query to remote servers, ask them to process data to a stage where data from different remote servers can be merged, and return that preprocessed data.
  The query interpreter then finishes processing the data.

The table's read method can return multiple IBlockInputStream objects to allow parallel data processing. These multiple block input streams can read from a table in parallel. Then you can wrap these streams with various transformations (such as expression evaluation or filtering) that can be calculated independently and create a UnionBlockInputStream on top of them, to read from multiple streams in parallel.

There are also TableFunctions. These are functions that return a temporary IStorage object to use in the FROM clause of a query.

To get a quick idea of how to implement your own table engine, look at something simple, like StorageMemory or StorageTinyLog.

> As the result of the read method, IStorage returns QueryProcessingStage – information about what parts of the query were already calculated inside storage. Currently we have only very coarse granularity for that information. There is no way for the storage to say "I have already processed this part of the expression in WHERE, for this range of data". We need to work on that.

Parsers

A query is parsed by a hand-written recursive descent parser. For example, ParserSelectQuery just recursively calls the underlying parsers for various parts of the query. Parsers create an AST. The AST is represented by nodes, which are instances of IAST.

> Parser generators are not used for historical reasons.

Interpreters

Interpreters are responsible for creating the query execution pipeline from an AST. There are simple interpreters, such as InterpreterExistsQueryand InterpreterDropQuery, or the more sophisticated InterpreterSelectQuery. The query execution pipeline is a combination of block input or output streams. For example, the result of interpreting the SELECT query is the IBlockInputStream to read the result set from; the result of the INSERT query is the IBlockOutputStream to write data for insertion to; and the result of interpreting the INSERT SELECT query is the IBlockInputStream that returns an empty result set on the first read, but that copies data from SELECT to INSERT at the same time.

InterpreterSelectQuery uses ExpressionAnalyzer and ExpressionActions machinery for query analysis and transformations. This is where most rule-based query optimizations are done. ExpressionAnalyzer is quite messy and should be rewritten: various query transformations and optimizations should be extracted to separate classes to allow modular transformations or query.

Functions

There are ordinary functions and aggregate functions. For aggregate functions, see the next section.

Ordinary functions don't change the number of rows – they work as if they are processing each row independently. In fact, functions are not called for individual rows, but for Block's of data to implement vectorized query execution.

There are some miscellaneous functions, like blockSize, rowNumberInBlock, and runningAccumulate, that exploit block processing and violate the independence of rows.

ClickHouse has strong typing, so implicit type conversion doesn't occur. If a function doesn't support a specific combination of types, an exception will be thrown. But functions can work (be overloaded) for many different combinations of types. For example, the plus function (to implement the + operator) works for any combination of numeric types: UInt8 + Float32, UInt16 + Int8, and so on. Also, some variadic functions can accept any number of arguments, such as the concat function.

Implementing a function may be slightly inconvenient because a function explicitly dispatches supported data types and supported IColumns. For example, the plus function has code generated by instantiation of a C++ template for each combination of numeric types, and for constant or non-constant left and right arguments.

> This is a nice place to implement runtime code generation to avoid template code bloat. Also, it will make it possible to add fused functions like fused multiply-add, or to make multiple comparisons in one loop iteration.

Due to vectorized query execution, functions are not short-circuit. For example, if you write WHERE f(x) AND g(y), both sides will be calculated, even for rows, when f(x) is zero (except when f(x) is a zero constant expression). But if selectivity of the f(x) condition is high, and calculation of f(x) is much cheaper than g(y), it's better to implement multi-pass calculation: first calculate f(x), then filter columns by the result, and then calculate g(y) only for smaller, filtered chunks of data.

Aggregate Functions

Aggregate functions are stateful functions. They accumulate passed values into some state, and allow you to get results from that state. They are managed with the IAggregateFunction interface. States can be rather simple (the state for AggregateFunctionCount is just a single UInt64 value) or quite complex (the state of AggregateFunctionUniqCombined is a combination of a linear array, a hash table and a HyperLogLog probabilistic data structure).

To deal with multiple states while executing a high-cardinality GROUP BY query, states are allocated in Arena (a memory pool), or they could be allocated in any suitable piece of memory. States can have a non-trivial constructor and destructor: for example, complex aggregation states can allocate additional memory themselves. This requires some attention to creating and destroying states and properly passing their ownership, to keep track of who and when will destroy states.

Aggregation states can be serialized and deserialized to pass over the network during distributed query execution or to write them on disk where there is not enough RAM. They can even be stored in a table with the DataTypeAggregateFunction to allow incremental aggregation of data.

> The serialized data format for aggregate function states is not versioned right now. This is ok if aggregate states are only stored temporarily. But we have the AggregatingMergeTree table engine for incremental aggregation, and people are already using it in production. This is why we should add support for backward compatibility when changing the serialized format for any aggregate function in the future.

Server

The server implements several different interfaces:

- An HTTP interface for any foreign clients.
- A TCP interface for the native ClickHouse client and for cross-server communication during distributed query execution.
- An interface for transferring data for replication.

Internally, it is just a basic multithreaded server without coroutines, fibers, etc. Since the server is not designed to process a high rate of simple queries but is intended to process a relatively low rate of complex queries, each of them can process a vast amount of data for analytics.

The server initializes the Context class with the necessary environment for query execution: the list of available databases, users and access rights, settings, clusters, the process list, the query log, and so on. This environment is used by interpreters.

We maintain full backward and forward compatibility for the server TCP protocol: old clients can talk to new servers and new clients can talk to old servers. But we don't want to maintain it eternally, and we are removing support for old versions after about one year.

> For all external applications, we recommend using the HTTP interface because it is simple and easy to use. The TCP protocol is more tightly linked to internal data structures: it uses an internal format for passing blocks of data and it uses custom framing for compressed data. We haven't released a C library for that protocol because it requires linking most of the ClickHouse codebase, which is not practical.

Distributed Query Execution

Servers in a cluster setup are mostly independent. You can create a Distributed table on one or all servers in a cluster. The Distributed table does not store data itself – it only provides a "view" to all local tables on multiple nodes of a cluster. When you SELECT from a Distributed table, it rewrites that query, chooses remote nodes according to load balancing settings, and sends the query to them. The Distributed table requests remote servers to process a query just up to a stage where intermediate results from different servers can be merged. Then it receives the intermediate results and merges them. The distributed table tries to distribute as much work as possible to remote servers, and does not send much intermediate data over the network.

> Things become more complicated when you have subqueries in IN or JOIN clauses and each of them uses a Distributed table. We have different strategies for execution of these queries.

There is no global query plan for distributed query execution. Each node has its own local query plan for its part of the job. We only have simple one-pass distributed query execution: we send queries for remote nodes and then merge the results. But this is not feasible for difficult queries with high cardinality GROUP BYs or with a large amount of temporary data for JOIN: in such cases, we need to "reshuffle" data between servers, which requires additional coordination. ClickHouse does not support that kind of query execution, and we need to work on it.

Merge Tree

MergeTree is a family of storage engines that supports indexing by primary key. The primary key can be an arbitary tuple of columns or expressions. Data in a MergeTree table is stored in "parts". Each part stores data in the primary key order (data is ordered lexicographically by the primary key tuple). All the table columns are stored in separate column.bin files in these parts. The files consist of compressed blocks. Each block is usually from 64 KB to 1 MB of uncompressed data, depending on the average value size. The blocks consist of column values placed contiguously one after the other. Column values are in the same order for each column (the order is defined by the primary key), so when you iterate by many columns, you get values for the corresponding rows.

The primary key itself is "sparse". It doesn't address each single row, but only some ranges of data. A separate primary.idx file has the value of the primary key for each N-th row, where N is called index_granularity (usually, N = 8192). Also, for each column, we have column.mrk files with "marks," which are offsets to each N-th row in the data file. Each mark is a pair: the offset in the file to the beginning of the compressed block, and the offset in the decompressed block to the beginning of data. Usually compressed blocks are aligned by marks, and the offset in the decompressed block is zero. Data for primary.idx always resides in memory and data for column.mrk files is cached.

When we are going to read something from a part in MergeTree, we look at primary.idx data and locate ranges that could possibly contain requested data, then look at column.mrk data and calculate offsets for where to start reading those ranges. Because of sparseness, excess data may be read. ClickHouse is not suitable for a high load of simple point queries, because the entire range with index_granularity rows must be read for each key, and the entire compressed block must be decompressed for each column. We made the index sparse because we must be able to maintain trillions of rows per single server without noticeable memory consumption for the index. Also, because the primary key is sparse, it is not unique: it cannot check the existence of the key in the table at INSERT time. You could have many rows with the same key in a table.

When you INSERT a bunch of data into MergeTree, that bunch is sorted by primary key order and forms a new part. To keep the number of parts relatively low, there are background threads that periodically select some parts and merge them to a single sorted part. That's why it is called MergeTree. Of course, merging leads to "write amplification". All parts are immutable: they are only created and deleted, but not modified. When SELECT is run, it holds a snapshot of the table (a set of parts). After merging, we also keep old parts for some time to make recovery after failure easier, so if we see that some merged part is probably broken, we can replace it with its source parts.

MergeTree is not an LSM tree because it doesn't contain "memtable" and "log": inserted data is written directly to the filesystem. This makes it suitable only to INSERT data in batches, not by individual row and not very frequently – about once per second is ok, but a thousand times a second is not. We did it this way for simplicity's sake, and because we are already inserting data in batches in our applications.

> MergeTree tables can only have one (primary) index: there aren't any secondary indices. It would be nice to allow multiple physical representations under one logical table, for example, to store data in more than one physical order or even to allow representations with pre-aggregated data along with original data.

There are MergeTree engines that are doing additional work during background merges. Examples are CollapsingMergeTree and AggregatingMergeTree. This could be treated as special support for updates. Keep in mind that these are not real updates because users usually have no control over the time when background merges will be executed, and data in a MergeTree table is almost always stored in more than one part, not in completely merged form.

Replication

Replication in ClickHouse is implemented on a per-table basis. You could have some replicated and some non-replicated tables on the same server. You could also have tables replicated in different ways, such as one table with two-factor replication and another with three-factor.

Replication is implemented in the ReplicatedMergeTree storage engine. The path in ZooKeeper is specified as a parameter for the storage engine. All tables with the same path in ZooKeeper become replicas of each other: they synchronize their data and maintain consistency. Replicas can be added and removed dynamically simply by creating or dropping a table.

Replication uses an asynchronous multi-master scheme. You can insert data into any replica that has a session with ZooKeeper, and data is replicated to all other replicas asynchronously. Because ClickHouse doesn't support UPDATEs, replication is conflict-free. As there is no quorum acknowledgment of inserts, just-inserted data might be lost if one node fails.

Metadata for replication is stored in ZooKeeper. There is a replication log that lists what actions to do. Actions are: get part; merge parts; drop partition, etc. Each replica copies the replication log to its queue and then executes the actions from the queue. For example, on insertion, the "get part" action is created in the log, and every replica downloads that part. Merges are coordinated between replicas to get byte-identical results. All parts are merged in the same way on all replicas. To achieve this, one replica is elected as the leader, and that replica initiates merges and writes "merge parts" actions to the log.

Replication is physical: only compressed parts are transferred between nodes, not queries. To lower the network cost (to avoid network amplification), merges are processed on each replica independently in most cases. Large merged parts are sent over the network only in cases of significant replication lag.

In addition, each replica stores its state in ZooKeeper as the set of parts and its checksums. When the state on the local filesystem diverges from the reference state in ZooKeeper, the replica restores its consistency by downloading missing and broken parts from other replicas. When there is some unexpected or broken data in the local filesystem, ClickHouse does not remove it, but moves it to a separate directory and forgets it.

The ClickHouse cluster consists of independent shards, and each shard consists of replicas. The cluster is not elastic, so after adding a new shard, data is not rebalanced between shards automatically. Instead, the cluster load will be uneven. This implementation gives you more control, and it is fine for relatively small clusters such as tens of nodes. But for clusters with hundreds of nodes that we are using in production, this approach becomes a significant drawback. We should implement a table engine that will span its data across the cluster with dynamically replicated regions that could be split and balanced between clusters automatically.

How to Build ClickHouse Release Package

Install Git and Pbuilder

```
sudo apt-get update
sudo apt-get install git pbuilder debhelper lsb-release fakeroot sudo debian-archive-keyring debian-keyring
```

Checkout ClickHouse Sources

```
git clone --recursive --branch stable https://github.com/yandex/ClickHouse.git
cd ClickHouse
```

Run Release Script

```
./release
```

How to Build ClickHouse for Development

The following tutorial is based on the Ubuntu Linux system.
With appropriate changes, it should also work on any other Linux distribution.
Only x86_64 with SSE 4.2 is supported. Support for AArch64 is experimental.

To test for SSE 4.2, do

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

Install Git and CMake

```
sudo apt-get install git cmake ninja-build
```

Or cmake3 instead of cmake on older systems.

Install GCC 7

There are several ways to do this.

Install from a PPA Package

```
sudo apt-get install software-properties-common
sudo apt-add-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
sudo apt-get install gcc-7 g++-7
```

Install from Sources

Look at ci/build-gcc-from-sources.sh

Use GCC 7 for Builds

```
export CC=gcc-7
export CXX=g++-7
```

Install Required Libraries from Packages

```
sudo apt-get install libicu-dev libreadline-dev
```

Checkout ClickHouse Sources

```
git clone --recursive git@github.com:yandex/ClickHouse.git
## or: git clone --recursive https://github.com/yandex/ClickHouse.git

cd ClickHouse
```

For the latest stable version, switch to the stable branch.

Build ClickHouse

```
mkdir build
cd build
cmake ..
ninja
cd ..
```

To create an executable, run ninja clickhouse.
This will create the dbms/programs/clickhouse executable, which can be used with client or server arguments.

How to Build ClickHouse on Mac OS X

Build should work on Mac OS X 10.12.

Install Homebrew

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Install Required Compilers, Tools, and Libraries

```
brew install cmake ninja gcc icu4c openssl libtool gettext readline
```

Checkout ClickHouse Sources

```
git clone --recursive git@github.com:yandex/ClickHouse.git
## or: git clone --recursive https://github.com/yandex/ClickHouse.git

cd ClickHouse
```

For the latest stable version, switch to the stable branch.

Build ClickHouse

```
mkdir build
cd build
cmake .. -DCMAKE_CXX_COMPILER=`which g++-8` -DCMAKE_C_COMPILER=`which gcc-8`
ninja
cd ..
```

Caveats

If you intend to run clickhouse-server, make sure to increase the system's maxfiles variable.

Note

You'll need to use sudo.

To do so, create the following file:

/Library/LaunchDaemons/limit.maxfiles.plist:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
       "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>Label</key>
    <string>limit.maxfiles</string>
    <key>ProgramArguments</key>
    <array>
      <string>launchctl</string>
      <string>limit</string>
      <string>maxfiles</string>
      <string>524288</string>
      <string>524288</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
    <key>ServiceIPC</key>
    <false/>
  </dict>
</plist>
```

Execute the following command:

```
$ sudo chown root:wheel /Library/LaunchDaemons/limit.maxfiles.plist
```

Reboot.

To check if it's working, you can use ulimit -n command.

Как писать код на C++

Общее

1. Этот текст носит рекомендательный характер.

2. Если вы редактируете код, то имеет смысл писать так, как уже написано.

3. Стиль нужен для единообразия. Единообразие нужно, чтобы было проще (удобнее) читать код. А также, чтобы было легче осуществлять поиск по коду.

4. Многие правила продиктованы не какими либо разумными соображениями, а сложившейся практикой.

Форматирование

1. Большую часть форматирования сделает автоматически clang-format.

2. Отступы — 4 пробела. Настройте среду разработки так, чтобы таб добавлял четыре пробела.

3. Открывающая и закрывающие фигурные скобки на отдельной строке.

```
inline void readBoolText(bool & x, ReadBuffer & buf)
{
    char tmp = '0';
    readChar(tmp, buf);
    x = tmp != '0';
}
```

4. Если всё тело функции — один statement, то его можно разместить на одной строке. При этом, вокруг фигурных скобок ставятся пробелы (кроме пробела на конце строки).

```
inline size_t mask() const                { return buf_size() - 1; }
inline size_t place(HashValue x) const    { return x & mask(); }
```

5. Для функций. Пробелы вокруг скобок не ставятся.

```
void reinsert(const Value & x)
```

```
memcpy(&buf[place_value], &x, sizeof(x));
```

6. В выражениях if, for, while и т.д. перед открывающей скобкой ставится пробел (в отличие от вызовов функций).

```
for (size_t i = 0; i < rows; i += storage.index_granularity)
```

7. Вокруг бинарных операторов (+, -, *, /, %, ...), а также тернарного оператора ?: ставятся пробелы.

```
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. Если ставится перенос строки, то оператор пишется на новой строке, и перед ним увеличивается отступ.

```
if (elapsed_ns)
    message << " ("
        << rows_read_on_server * 1000000000 / elapsed_ns << " rows/s., "
        << bytes_read_on_server * 1000.0 / elapsed_ns << " MB/s.) ";
```

9. Внутри строки можно, выполнять выравнивание с помощью пробелов.

```
dst.ClickLogID         = click.LogID;
dst.ClickEventID       = click.EventID;
dst.ClickGoodEvent     = click.GoodEvent;
```

10. Вокруг операторов ., -> не ставятся пробелы.

При необходимости, оператор может быть перенесён на новую строку. В этом случае, перед ним увеличивается отступ.

11. Унарные операторы --, ++, *, &, ... не отделяются от аргумента пробелом.

12. После запятой ставится пробел, а перед — нет. Аналогично для точки с запятой внутри выражения for.

13. Оператор [] не отделяется пробелами.

14. В выражении template <...>, между template и < ставится пробел, а после < и до > не ставится.

```
template <typename TKey, typename TValue>
struct AggregatedStatElement
{}
```

15. В классах и структурах, public, private, protected пишется на том же уровне, что и class/struct, а остальной код с отступом.

```
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. Если на весь файл один namespace и кроме него ничего существенного нет, то отступ внутри namespace не нужен.

17. Если блок для выражения if, for, while, ... состоит из одного statement, то фигурные скобки не обязательны. Вместо этого поместите statement на отдельную строку. Это правило справедливо и для вложенных if, for, while, ...

Если внутренний statement содержит фигурные скобки или else, то внешний блок следует писать в фигурных скобках.

```
/// Finish write.
for (auto & stream : streams)
    stream.second->finalize();
```

18. Не должно быть пробелов на концах строк.

19. Исходники в кодировке UTF-8.

20. В строковых литералах можно использовать не-ASCII.

```
<< ", " << (timer.elapsed() / chunks_stats.hits) << " μsec/hit.";
```

21. Не пишите несколько выражений в одной строке.

22. Внутри функций группируйте блоки кода, отделяя их не более, чем одной пустой строкой.

23. Функции, классы, и т. п. отделяются друг от друга одной или двумя пустыми строками.

24. const (относящийся к значению) пишется до имени типа.

```
//correct
const char * pos
const std::string & s
//incorrect
char const * pos
```

25. При объявлении указателя или ссылки, символы * и & отделяются пробелами с обеих сторон.

```
//correct
const char * pos
//incorrect
const char* pos
const char *pos
```

26. При использовании шаблонных типов, пишите using (кроме, возможно, простейших случаев).

То есть, параметры шаблона указываются только в using и затем не повторяются в коде.

using может быть объявлен локально, например, внутри функции.

```
//correct
using FileStreams = std::map<std::string, std::shared_ptr<Stream>>;
FileStreams streams;
//incorrect
std::map<std::string, std::shared_ptr<Stream>> streams;
```

27. Нельзя объявлять несколько переменных разных типов в одном выражении.

```
//incorrect
int x, *y;
```

28. C-style cast не используется.

```
//incorrect
std::cerr << (int)c <<; std::endl;
//correct
std::cerr << static_cast<int>(c) << std::endl;
```

29. В классах и структурах, группируйте отдельно методы и отдельно члены, внутри каждой области видимости.

30. Для не очень большого класса/структуры, можно не отделять объявления методов от реализации.

Аналогично для маленьких методов в любых классах/структурах.

Для шаблонных классов/структур, лучше не отделять объявления методов от реализации (так как иначе они всё равно должны быть определены в той же единице трансляции).

31. Не обязательно умещать код по ширине в 80 символов. Можно в 140.

32. Всегда используйте префиксный инкремент/декремент, если постфиксный не нужен.

```
for (Names::const_iterator it = column_names.begin(); it != column_names.end(); ++it)
```

Комментарии

1. Необходимо обязательно писать комментарии во всех нетривиальных местах.

Это очень важно. При написании комментария, можно успеть понять, что код не нужен вообще, или что всё сделано неверно.

```
/** 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. Комментарии могут быть сколь угодно подробными.

3. Комментарии пишутся до соответствующего кода. В редких случаях после, на той же строке.

```
/** 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. Комментарии следует писать только на английском языке.

5. При написании библиотеки, разместите подробный комментарий о том, что это такое, в самом главном заголовочном файле.

6. Нельзя писать комментарии, которые не дают дополнительной информации. В частности, нельзя писать пустые комментарии вроде этого:

```
/*
* 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:
*/
```

Пример взят с ресурса http://home.tamk.fi/~jaalto/course/coding-style/doc/unmaintainable-code/.

7. Нельзя писать мусорные комментарии (автор, дата создания...) в начале каждого файла.

8. Однострочные комментарии начинаются с трёх слешей: /// , многострочные с /**. Такие комментарии считаются «документирующими».

Замечание: такие комментарии могут использоваться для генерации документации с помощью Doxygen. Но, фактически, Doxygen не используется, так как для навигации по коду гораздо удобне использовать возможности IDE.

9. В начале и конце многострочного комментария, не должно быть пустых строк (кроме строки, на которой закрывается многострочный комментарий).

10. Для закомментированных кусков кода, используются обычные, не "документирующие" комментарии.

11. Удаляйте закомментированные куски кода перед коммитом.

12. Не нужно писать нецензурную брань в комментариях или коде.

13. Не пишите прописными буквами. Не используйте излишнее количество знаков препинания.

```
/// WHAT THE FAIL???
```

14. Не составляйте из комментариев строки-разделители.

```
///**************************************************
```

15. Не нужно писать в комментарии диалог (лучше сказать устно).

```
/// Why did you do this stuff?
```

16. Не нужно писать комментарий в конце блока о том, что представлял собой этот блок.

```
/// for
```

Имена

1. В именах переменных и членов класса используйте маленькие буквами с подчёркиванием.

```
size_t max_block_size;
```

2. Имена функций (методов) camelCase с маленькой буквы.

```
std::string getName() const override { return "Memory"; }
```

3. Имена классов (структур) - CamelCase с большой буквы. Префиксы кроме I для интерфейсов - не используются.

```
class StorageMemory : public IStorage
```

4. using называются также, как классы, либо с _t на конце.

5. Имена типов — параметров шаблонов: в простых случаях - T; T, U; T1, T2.

В более сложных случаях - либо также, как имена классов, либо можно добавить в начало букву T.

```
template <typename TKey, typename TValue>
struct AggregatedStatElement
```

6. Имена констант — параметров шаблонов: либо также, как имена переменных, либо N в простом случае.

```
template <bool without_www>
struct ExtractDomain
```

7. Для абстрактных классов (интерфейсов) можно добавить в начало имени букву I.

```
class IBlockInputStream
```

8. Если переменная используется достаточно локально, то можно использовать короткое имя.

В остальных случаях используйте имя, описывающее смысл.

```
bool info_successfully_loaded = false;
```

9. В именах define и глобальных констант используется ALL_CAPS с подчёркиванием.

```
##define MAX_SRC_TABLE_NAMES_TO_STORE 1000
```

10. Имена файлов с кодом называйте по стилю соответственно тому, что в них находится.

Если в файле находится один класс, назовите файл, как класс (CamelCase).

Если в файле находится одна функция, назовите файл, как функцию (camelCase).

11. Если имя содержит сокращение, то:

- для имён переменных, всё сокращение пишется маленькими буквами mysql_connection (не mySQL_connection).
- для имён классов и функций, сохраняются большие буквы в сокращении MySQLConnection (не MySqlConnection).

12. Параметры конструктора, использующиеся сразу же для инициализации соответствующих членов класса, следует назвать также, как и члены класса, добавив подчёркивание в конец.

```
FileQueueProcessor(
    const std::string & path_,
    const std::string & prefix_,
    std::shared_ptr<FileHandler> handler_)
    : path(path_),
    prefix(prefix_),
    handler(handler_),
    log(&Logger::get("FileQueueProcessor"))
{
}
```

Также можно называть параметры конструктора так же, как и члены класса (не добавлять подчёркивание), но только если этот параметр не используется в теле конструктора.

13. Именование локальных переменных и членов класса никак не отличается (никакие префиксы не нужны).

```
timer (not m_timer)
```

14. Константы в enum — CamelCase с большой буквы. Также допустим ALL_CAPS. Если enum не локален, то используйте enum class.

```
enum class CompressionMethod
{
    QuickLZ = 0,
    LZ4     = 1,
};
```

15. Все имена - по английски. Транслит с русского использовать нельзя.

```
не Stroka
```

16. Сокращения (из нескольких букв разных слов) в именах можно использовать только если они являются общепринятыми (если для сокращения можно найти расшифровку в английской википедии или сделав поисковый запрос).

```
`AST`, `SQL`.

Не `NVDH` (что-то неведомое)
```

Сокращения в виде обрезанного слова можно использовать, только если такое сокращение является широко используемым.

Впрочем, сокращения также можно использовать, если расшифровка находится рядом в комментарии.

17. Имена файлов с исходниками на C++ должны иметь расширение только .cpp. Заголовочные файлы - только .h.

Как писать код

1. Управление памятью.

Ручное освобождение памяти (delete) можно использовать только в библиотечном коде.

В свою очередь, в библиотечном коде, оператор delete можно использовать только в деструкторах.

В прикладном коде следует делать так, что память освобождается каким-либо объектом, который владеет ей.

Примеры:

- проще всего разместить объект на стеке, или сделать его членом другого класса.
- для большого количества маленьких объектов используйте контейнеры.
- для автоматического освобождения маленького количества объектов, выделенных на куче, используйте shared_ptr/unique_ptr.

2. Управление ресурсами.

Используйте RAII и см. пункт выше.

3. Обработка ошибок.

Используйте исключения. В большинстве случаев, нужно только кидать исключения, а ловить - не нужно (потому что RAII).

В программах офлайн обработки данных, зачастую, можно не ловить исключения.

В серверах, обрабатывающих пользовательские запросы, как правило, достаточно ловить исключения на самом верху обработчика соединения.

В функциях потока, следует ловить и запоминать все исключения, чтобы выкинуть их в основном потоке после join.

```
/// Если вычислений ещё не было - вычислим первый блок синхронно
if (!started)
{
    calculate();
    started = true;
}
else    /// Если вычисления уже идут - подождём результата
    pool.wait();

if (exception)
    exception->rethrow();
```

Ни в коем случае не «проглатывайте» исключения без разбора. Ни в коем случае, не превращайте все исключения без разбора в сообщения в логе.

```
//Not correct
catch (...) {}
```

Если вам нужно проигнорировать какие-то исключения, то игнорируйте только конкретные, а остальные кидайте обратно.

```
catch (const DB::Exception & e)
{
    if (e.code() == ErrorCodes::UNKNOWN_AGGREGATE_FUNCTION)
        return nullptr;
    else
        throw;
}
```

При использовании функций, использующих коды возврата или errno, проверяйте результат и кидайте исключение.

```
if (0 != close(fd))
    throwFromErrno("Cannot close file " + file_name, ErrorCodes::CANNOT_CLOSE_FILE);
```

assert не используются.

4. Типы исключений.

В прикладном коде не требуется использовать сложную иерархию исключений. Желательно, чтобы текст исключения был понятен системному администратору.

5. Исключения, вылетающие из деструкторов.

Использовать не рекомендуется, но допустимо.

Используйте следующие варианты:

- Сделайте функцию (done() или finalize()), которая позволяет заранее выполнить всю работу, в процессе которой может возникнуть исключение. Если эта функция была вызвана, то затем в деструкторе не должно возникать исключений.
- Слишком сложную работу (например, отправку данных по сети) можно вообще не делать в деструкторе, рассчитывая, что пользователь заранее позовёт метод для завершения работы.
- Если в деструкторе возникло исключение, желательно не "проглатывать" его, а вывести информацию в лог (если в этом месте доступен логгер).

- В простых программах, если соответствующие исключения не ловятся, и приводят к завершению работы с записью информации в лог, можно не беспокоиться об исключениях, вылетающих из деструкторов, так как вызов std::terminate (в случае noexcept по умолчанию в C++11), является приемлимым способом обработки исключения.

6. Отдельные блоки кода.

Внутри одной функции, можно создать отдельный блок кода, для того, чтобы сделать некоторые переменные локальными в нём, и для того, чтобы соответствующие деструкторы были вызваны при выходе из блока.

```
Block block = data.in->read();

{
    std::lock_guard<std::mutex> lock(mutex);
    data.ready = true;
    data.block = block;
}

ready_any.set();
```

7. Многопоточность.

В программах офлайн обработки данных:

- сначала добейтесь более-менее максимальной производительности на одном процессорном ядре, потом можно распараллеливать код, но только если есть необходимость.

В программах - серверах:

- используйте пул потоков для обработки запросов. На данный момент, у нас не было задач, в которых была бы необходимость использовать userspace context switching.

Fork для распараллеливания не используется.

8. Синхронизация потоков.

Часто можно сделать так, чтобы отдельные потоки писали данные в разные ячейки памяти (лучше в разные кэш-линии), и не использовать синхронизацию потоков (кроме joinAll).

Если синхронизация нужна, то в большинстве случаев, достаточно использовать mutex под lock_guard.

В остальных случаях, используйте системные примитивы синхронизации. Не используйте busy wait.

Атомарные операции можно использовать только в простейших случаях.

Не нужно писать самостоятельно lock-free структуры данных, если вы не являетесь экспертом.

9. Ссылки и указатели.

В большинстве случаев, предпочитайте ссылки.

10. const.

Используйте константные ссылки, указатели на константу, const_iterator, константные методы.

Считайте, что const — вариант написания «по умолчанию», а отсутствие const только при необходимости.

Для переменных, передающихся по значению, использовать const обычно не имеет смысла.

11. unsigned.

Используйте unsigned, если нужно.

12. Числовые типы.

Используйте типы UInt8, UInt16, UInt32, UInt64, Int8, Int16, Int32, Int64, а также size_t, ssize_t, ptrdiff_t.

Не используйте для чисел типы signed/unsigned long, long long, short, signed/unsigned char, char.

13. Передача аргументов.

Сложные значения передавайте по ссылке (включая std::string).

Если функция захватывает владение объектом, созданным на куче, то сделайте типом аргумента shared_ptr или unique_ptr.

14. Возврат значений.

В большинстве случаев, просто возвращайте значение с помощью return. Не пишите [return std::move(res)] {.strike}.

Если внутри функции создаётся объект на куче и отдаётся наружу, то возвращайте shared_ptr или unique_ptr.

В некоторых редких случаях, может потребоваться возвращать значение через аргумент функции. В этом случае, аргументом будет ссылка.

```
using AggregateFunctionPtr = std::shared_ptr<IAggregateFunction>;

/** Позволяет создать агрегатную функцию по её имени.
  */
class AggregateFunctionFactory
{
public:
    AggregateFunctionFactory();
    AggregateFunctionPtr get(const String & name, const DataTypes & argument_types) const;
```

15. namespace.

Для прикладного кода отдельный namespace использовать не нужно.

Для маленьких библиотек - не требуется.

Для не совсем маленьких библиотек - поместите всё в namespace.

Внутри библиотеки в .h файле можно использовать namespace detail для деталей реализации, не нужных прикладному коду.

В .cpp файле можно использовать static или анонимный namespace для скрытия символов.

Также, namespace можно использовать для enum, чтобы соответствующие имена не попали во внешний namespace (но лучше использовать enum class).

16. Отложенная инициализация.

Обычно, если для инициализации требуются аргументы, то не пишите конструктор по умолчанию.

Если потом вам потребовалась отложенная инициализация, то вы можете дописать конструктор по умолчанию (который создаст объект с некорректным состоянием). Или, для небольшого количества объектов, можно использовать shared_ptr/unique_ptr.

```
Loader(DB::Connection * connection_, const std::string & query, size_t max_block_size_);

/// Для отложенной инициализации
Loader() {}
```

17. Виртуальные функции.

Если класс не предназначен для полиморфного использования, то не нужно делать функции виртуальными зря. Это относится и к деструктору.

18. Кодировки.

Везде используется UTF-8. Используется std::string, char *. Не используется std::wstring, wchar_t.

19. Логгирование.

См. примеры везде в коде.

Перед коммитом, удалите всё бессмысленное и отладочное логгирование, и другие виды отладочного вывода.

Не должно быть логгирования на каждую итерацию внутреннего цикла, даже уровня Trace.

При любом уровне логгирования, логи должно быть возможно читать.

Логгирование следует использовать, в основном, только в прикладном коде.

Сообщения в логе должны быть написаны на английском языке.

Желательно, чтобы лог был понятен системному администратору.

Не нужно писать ругательства в лог.

В логе используется кодировка UTF-8. Изредка можно использовать в логе не-ASCII символы.

20. Ввод-вывод.

Во внутренних циклах (в критичных по производительности участках программы) нельзя использовать iostreams (в том числе, ни в коем случае не используйте stringstream).

Вместо этого используйте библиотеку DB/IO.

21. Дата и время.

См. библиотеку DateLUT.

22. include.

В заголовочном файле используется только #pragma once, а include guards писать не нужно.

23. using.

using namespace не используется. Можно использовать using что-то конкретное. Лучше локально, внутри класса или функции.

24. Не нужно использовать trailing return type для функций, если в этом нет необходимости.

```
[auto f() -&gt; void;]{.strike}
```

25. Объявление и инициализация переменных.

```
//right way
std::string s = "Hello";
std::string s{"Hello"};

//wrong way
auto s = std::string{"Hello"};
```

26. Для виртуальных функций, пишите virtual в базовом классе, а в классах-наследниках, пишите override и не пишите virtual.

Неиспользуемые возможности языка C++

1. Виртуальное наследование не используется.

2. Спецификаторы исключений из C++03 не используются.

Платформа

1. Мы пишем код под конкретную платформу.

Хотя, при прочих равных условиях, предпочитается более-менее кроссплатформенный или легко портируемый код.

2. Язык - C++17.

3. Компилятор - gcc. На данный момент (декабрь 2017), код собирается версией 7.2. (Также код может быть собран clang 5)

Используется стандартная библиотека (реализация libstdc++ или libc++).

4. ОС - Linux Ubuntu, не более старая, чем Precise.

5. Код пишется под процессор с архитектурой x86_64.

Набор инструкций минимальный из поддержаных нашими серверами. Сейчас это - SSE4.2.

6. Используются флаги компиляции -Wall -Wextra -Werror.

7. Используется статическая линковка со всеми библиотеками кроме тех, которые трудно подключить статически (см. вывод команды ldd).

8. Код разрабатывается и отлаживается с релизными параметрами сборки.

Инструментарий

1. Хорошая среда разработки - KDevelop.

2. Для отладки используется gdb, valgrind (memcheck), strace, -fsanitize=..., tcmalloc_minimal_debug.

3. Для профилирования используется Linux Perf, valgrind (callgrind), strace -cf.

4. Исходники в Git.

5. Сборка с помощью CMake.

6. Программы выкладываются с помощью deb пакетов.

7. Коммиты в master не должны ломать сборку проекта.

А работоспособность собранных программ гарантируется только для отдельных ревизий.

8. Коммитьте как можно чаще, в том числе и нерабочий код.

Для этого следует использовать бранчи.

Если ваш код в ветке master ещё не собирается, исключите его из сборки перед push, также вы будете должны его доработать или удалить в течение нескольких дней.

9. Для нетривиальных изменений, используются бранчи. Следует загружать бранчи на сервер.

10. Ненужный код удаляется из исходников.

Библиотеки

1. Используются стандартная библиотека C++14 (допустимо использовать экспериментальные расширения) а также фреймворки boost, Poco.

2. При необходимости, можно использовать любые известные библиотеки, доступные в ОС из пакетов.

Если есть хорошее готовое решение, то оно используется, даже если для этого придётся установить ещё одну библиотеку.

(Но будьте готовы к тому, что иногда вам придётся выкидывать плохие библиотеки из кода.)

3. Если в пакетах нет нужной библиотеки, или её версия достаточно старая, или если она собрана не так, как нужно, то можно использовать библиотеку, устанавливаемую не из пакетов.

4. Если библиотека достаточно маленькая и у неё нет своей системы сборки, то следует включить её файлы в проект, в директорию contrib.

5. Предпочтение всегда отдаётся уже использующимся библиотекам.

Общее

1. Пишите как можно меньше кода.

2. Пробуйте самое простое решение.

3. Не нужно писать код, если вы ещё не знаете, что будет делать ваша программа, и как будет работать её внутренний цикл.

4. В простейших случаях, используйте using вместо классов/структур.

5. Если есть возможность - не пишите конструкторы копирования, операторы присваивания, деструктор (кроме виртуального, если класс содержит хотя бы одну виртуальную функцию), move-конструкторы и move-присваивания. То есть, чтобы соответствущие функции, генерируемые компилятором, работали правильно. Можно использовать default.

6. Приветствуется упрощение и уменьшение объёма кода.

Дополнительно

1. Явное указание std:: для типов из stddef.h.

Рекомендуется не указывать. То есть, рекомендуется писать size_t вместо std::size_t, это короче.

При желании, можно дописать std::, этот вариант допустим.

2. Явное указание std:: для функций из стандартной библиотеки C.

Не рекомендуется. То есть, пишите memcpy вместо std::memcpy.

Причина - существуют похожие нестандартные функции, например, memmem. Мы можем использовать и изредка используем эти функции. Эти функции отсутствуют в namespace std.

Если вы везде напишете std::memcpy вместо memcpy, то будет неудобно смотреться memmem без std::.

Тем не менее, указывать std:: тоже допустимо, если так больше нравится.

3. Использование функций из C при наличии аналогов в стандартной библиотеке C++.

Допустимо, если это использование эффективнее.

Для примера, для копирования длинных кусков памяти, используйте memcpy вместо std::copy.

4. Перенос длинных аргументов функций.

Допустимо использовать любой стиль переноса, похожий на приведённые ниже:

```
function(
  T1 x1,
  T2 x2)
```

```
function(
  size_t left, size_t right,
  const & RangesInDataParts ranges,
  size_t limit)
```

```
function(size_t left, size_t right,
  const & RangesInDataParts ranges,
  size_t limit)
```

```
function(size_t left, size_t right,
    const & RangesInDataParts ranges,
    size_t limit)
```

```
function(
    size_t left,
    size_t right,
    const & RangesInDataParts ranges,
    size_t limit)
```

ClickHouse Testing

Functional Tests

Functional tests are the most simple and convenient to use. Most of ClickHouse features can be tested with functional tests and they are mandatory to use for every change in ClickHouse code that can be tested that way.

Each functional test sends one or multiple queries to the running ClickHouse server and compares the result with reference.

Tests are located in dbms/tests/queries directory. There are two subdirectories: stateless and stateful. Stateless tests run queries without any preloaded test data - they often create small synthetic datasets on the fly, within the test itself. Stateful tests require preloaded test data from Yandex.Metrica and not available to general public. We tend to use only stateless tests and avoid adding new stateful tests.

Each test can be one of two types: .sql and .sh. .sql test is the simple SQL script that is piped to clickhouse-client --multiquery --testmode. .sh test is a script that is run by itself.

To run all tests, use dbms/tests/clickhouse-test tool. Look --help for the list of possible options. You can simply run all tests or run subset of tests filtered by substring in test name: ./clickhouse-test substring.

The most simple way to invoke functional tests is to copy clickhouse-client to /usr/bin/, run clickhouse-server and then run ./clickhouse-test from its own directory.

To add new test, create a .sql or .sh file in dbms/tests/queries/0_stateless directory, check it manually and then generate .reference file in the following way: clickhouse-client -n --testmode < 00000_test.sql > 00000_test.reference or ./00000_test.sh > ./00000_test.reference.

Tests should use (create, drop, etc) only tables in test database that is assumed to be created beforehand; also tests can use temporary tables.

If you want to use distributed queries in functional tests, you can leverage remote table function with 127.0.0.{1..2} addresses for the server to query itself; or you can use predefined test clusters in server configuration file like test_shard_localhost.

Some tests are marked with zookeeper, shard or long in their names. zookeeper is for tests that are using ZooKeeper. shard is for tests that requires server to listen 127.0.0.*; distributed or global have the same meaning. long is for tests that run slightly longer that one second. You can disable these groups of tests using --no-zookeeper, --no-shard and --no-long options, respectively.

Known bugs

If we know some bugs that can be easily reproduced by functional tests, we place prepared functional tests in dbms/tests/queries/bugs directory. These tests will be moved to dbms/tests/queries/0_stateless when bugs are fixed.

Integration Tests

Integration tests allow to test ClickHouse in clustered configuration and ClickHouse interaction with other servers like MySQL, Postgres, MongoDB. They are useful to emulate network splits, packet drops, etc. These tests are run under Docker and create multiple containers with various software.

See dbms/tests/integration/README.md on how to run these tests.

Note that integration of ClickHouse with third-party drivers is not tested. Also we currently don't have integration tests with our JDBC and ODBC drivers.

Unit Tests

Unit tests are useful when you want to test not the ClickHouse as a whole, but a single isolated library or class. You can enable or disable build of tests with ENABLE_TESTS CMake option. Unit tests (and other test programs) are located in tests subdirectories across the code. To run unit tests, type ninja test. Some tests use gtest, but some are just programs that return non-zero exit code on test failure.

It's not necessarily to have unit tests if the code is already covered by functional tests (and functional tests are usually much more simple to use).

Performance Tests

Performance tests allow to measure and compare performance of some isolated part of ClickHouse on synthetic queries. Tests are located at dbms/tests/performance. Each test is represented by .xml file with description of test case. Tests are run with clickhouse performance-test tool (that is embedded in clickhouse binary). See --help for invocation.

Each test run one or miltiple queries (possibly with combinations of parameters) in a loop with some conditions for stop (like "maximum execution speed is not changing in three seconds") and measure some metrics about query performance (like "maximum execution speed"). Some tests can contain preconditions on preloaded test dataset.

If you want to improve performance of ClickHouse in some scenario, and if improvements can be observed on simple queries, it is highly recommended to write a performance test. It always makes sense to use perf top or other perf tools during your tests.

Test Tools And Scripts

Some programs in tests directory are not prepared tests, but are test tools. For example, for Lexer there is a tool dbms/src/Parsers/tests/lexer that just do tokenization of stdin and writes colorized result to stdout. You can use these kind of tools as a code examples and for exploration and manual testing.

You can also place pair of files .sh and .reference along with the tool to run it on some predefined input - then script result can be compared to .reference file. These kind of tests are not automated.

Miscellanous Tests

There are tests for external dictionaries located at dbms/tests/external_dictionaries and for machine learned models in dbms/tests/external_models. These tests are not updated and must be transferred to integration tests.

There is separate test for quorum inserts. This test run ClickHouse cluster on separate servers and emulate various failure cases: network split, packet drop (between ClickHouse nodes, between ClickHouse and ZooKeeper, between ClickHouse server and client, etc.), kill -9, kill -STOP and kill -CONT , like Jepsen. Then the test checks that all acknowledged inserts was written and all rejected inserts was not.

Quorum test was written by separate team before ClickHouse was open-sourced. This team no longer work with ClickHouse. Test was accidentially written in Java. For these reasons, quorum test must be rewritten and moved to integration tests.

Manual Testing

When you develop a new feature, it is reasonable to also test it manually. You can do it with the following steps:

Build ClickHouse. Run ClickHouse from the terminal: change directory to dbms/src/programs/clickhouse-server and run it with ./clickhouse-server. It will use configuration (config.xml, users.xml and files within config.d and users.d directories) from the current directory by default. To connect to ClickHouse server, run dbms/src/programs/clickhouse-client/clickhouse-client.

Note that all clickhouse tools (server, client, etc) are just symlinks to a single binary named clickhouse. You can find this binary at dbms/src/programs/clickhouse. All tools can also be invoked as clickhouse tool instead of clickhouse-tool.

Alternatively you can install ClickHouse package: either stable release from Yandex repository or you can build package for yourself with ./release in ClickHouse sources root. Then start the server with sudo service clickhouse-server start (or stop to stop the server). Look for logs at /etc/clickhouse-server/clickhouse-server.log.

When ClickHouse is already installed on your system, you can build a new clickhouse binary and replace the existing binary:

```
sudo service clickhouse-server stop
sudo cp ./clickhouse /usr/bin/
sudo service clickhouse-server start
```

Also you can stop system clickhouse-server and run your own with the same configuration but with logging to terminal:

```
sudo service clickhouse-server stop
sudo -u clickhouse /usr/bin/clickhouse server --config-file /etc/clickhouse-server/config.xml
```

Example with gdb:

```
sudo -u clickhouse gdb --args /usr/bin/clickhouse server --config-file /etc/clickhouse-server/config.xml
```

If the system clickhouse-server is already running and you don't want to stop it, you can change port numbers in your config.xml (or override them in a file in config.d directory), provide appropriate data path, and run it.

clickhouse binary has almost no dependencies and works across wide range of Linux distributions. To quick and dirty test your changes on a server, you can simply scp your fresh built clickhouse binary to your server and then run it as in examples above.

Testing Environment

Before publishing release as stable we deploy it on testing environment. Testing environment is a cluster that process 1/39 part of Yandex.Metrica data. We share our testing environment with Yandex.Metrica team. ClickHouse is upgraded without downtime on top of existing data. We look at first that data is processed successfully without lagging from realtime, the replication continue to work and there is no issues visible to Yandex.Metrica team. First check can be done in the following way:

```
SELECT hostName() AS h, any(version()), any(uptime()), max(UTCEventTime), count() FROM remote('example01-01-{1..3}t', merge, hits) WHERE EventDate >= today() - 2 GROUP BY h ORDER BY h;
```

In some cases we also deploy to testing environment of our friend teams in Yandex: Market, Cloud, etc. Also we have some hardware servers that are used for development purposes.

Load Testing

After deploying to testing environment we run load testing with queries from production cluster. This is done manually.

Make sure you have enabled query_log on your production cluster.

Collect query log for a day or more:

```
clickhouse-client --query="SELECT DISTINCT query FROM system.query_log WHERE event_date = today() AND query LIKE '%ym:%' AND query NOT LIKE '%system.query_log%' AND type = 2 AND is_initial_query" > queries.tsv
```

This is a way complicated example. `type = 2` will filter queries that are executed successfully. `query LIKE '%ym:%'` is to select relevant queries from Yandex.Metrica. `is_initial_query` is to select only queries that are initiated by client, not by ClickHouse itself (as parts of distributed query processing).

`scp` this log to your testing cluster and run it as following:

```
clickhouse benchmark --concurrency 16 < queries.tsv
```

(probably you also want to specify a `--user`)

Then leave it for a night or weekend and go take a rest.

You should check that `clickhouse-server` doesn't crash, memory footprint is bounded and performance not degrading over time.

Precise query execution timings are not recorded and not compared due to high variability of queries and environment.

Build Tests

Build tests allow to check that build is not broken on various alternative configurations and on some foreign systems. Tests are located at `ci` directory. They run build from source inside Docker, Vagrant, and sometimes with `qemu-user-static` inside Docker. These tests are under development and test runs are not automated.

Motivation:

Normally we release and run all tests on a single variant of ClickHouse build. But there are alternative build variants that are not thoroughly tested. Examples:

- build on FreeBSD;
- build on Debian with libraries from system packages;
- build with shared linking of libraries;
- build on AArch64 platform;
- build on PowerPc platform.

For example, build with system packages is bad practice, because we cannot guarantee what exact version of packages a system will have. But this is really needed by Debian maintainers. For this reason we at least have to support this variant of build. Another example: shared linking is a common source of trouble, but it is needed for some enthusiasts.

Though we cannot run all tests on all variant of builds, we want to check at least that various build variants are not broken. For this purpose we use build tests.

Testing For Protocol Compatibility

When we extend ClickHouse network protocol, we test manually that old clickhouse-client works with new clickhouse-server and new clickhouse-client works with old clickhouse-server (simply by running binaries from corresponding packages).

Help From The Compiler

Main ClickHouse code (that is located in `dbms` directory) is built with `-Wall -Wextra -Werror` and with some additional enabled warnings. Although these options are not enabled for third-party libraries.

Clang has even more useful warnings - you can look for them with `-Weverything` and pick something to default build.

For production builds, gcc is used (it still generates slightly more efficient code than clang). For development, clang is usually more convenient to use. You can build on your own machine with debug mode (to save battery of your laptop), but please note that compiler is able to generate more warnings with `-O3` due to better control flow and inter-procedure analysis. When building with clang, `libc++` is used instead of `libstdc++` and when building with debug mode, debug version of `libc++` is used that allows to catch more errors at runtime.

Sanitizers

Address sanitizer.
We run functional and integration tests under ASan on per-commit basis.

Valgrind (Memcheck).
We run functional tests under Valgrind overnight. It takes multiple hours. Currently there is one known false positive in re2 library, see this article.

Undefined behaviour sanitizer.
We run functional and integration tests under ASan on per-commit basis.

Thread sanitizer.
We run functional tests under TSan on per-commit basis. We still don't run integration tests under TSan on per-commit basis.

Memory sanitizer.
Currently we still don't use MSan.

Debug allocator.
Debug version of jemalloc is used for debug build.

Fuzzing

We use simple fuzz test to generate random SQL queries and to check that the server doesn't die. Fuzz testing is performed with Address sanitizer. You can find it in 00746_sql_fuzzy.pl. This test should be run continuously (overnight and longer).

As of December 2018, we still don't use isolated fuzz testing of library code.

Security Audit

People from Yandex Cloud department do some basic overview of ClickHouse capabilities from the security standpoint.

Static Analyzers

We run PVS-Studio on per-commit basis. We have evaluated clang-tidy, Coverity, cppcheck, PVS-Studio, tscancode. You will find instructions for usage in dbms/tests/instructions/ directory. Also you can read the article in russian.

If you use CLion as an IDE, you can leverage some clang-tidy checks out of the box.

Hardening

FORTIFY_SOURCE is used by default. It is almost useless, but still makes sense in rare cases and we don't disable it.

Code Style

Code style rules are described here.

To check for some common style violations, you can use utils/check-style script.

To force proper style of your code, you can use clang-format. File .clang-format is located at the sources root. It mostly corresponding with our actual code style. But it's not recommended to apply clang-format to existing files because it makes formatting worse. You can use clang-format-diff tool that you can find in clang source repository.

Alternatively you can try uncrustify tool to reformat your code. Configuration is in uncrustify.cfg in the sources root. It is less tested than clang-format.

CLion has its own code formatter that has to be tuned for our code style.

Metrica B2B Tests

Each ClickHouse release is tested with Yandex Metrica and AppMetrica engines. Testing and stable versions of ClickHouse are deployed on VMs and run with a small copy of Metrica engine that is processing fixed sample of input data. Then results of two instances of Metrica engine are compared together.

These tests are automated by separate team. Due to high number of moving parts, tests are fail most of the time by completely unrelated reasons, that are very difficult to figure out. Most likely these tests have negative value for us. Nevertheless these tests was proved to be useful in about one or two times out of hundreds.

Test Coverage

As of July 2018 we don't track test coverage.

Test Automation

We run tests with Yandex internal CI and job automation system named "Sandbox".

Build jobs and tests are run in Sandbox on per commit basis. Resulting packages and test results are published in GitHub and can be downloaded by direct links. Artifacts are stored eternally. When you send a pull request on GitHub, we tag it as "can be tested" and our CI system will build ClickHouse packages (release, debug, with address sanitizer, etc) for you.

We don't use Travis CI due to the limit on time and computational power.
We don't use Jenkins. It was used before and now we are happy we are not using Jenkins.

Используемые сторонние библиотеки

Библиотека	**Лицензия**
base64	BSD 2-Clause License
boost	Boost Software License 1.0
brotli	MIT
capnproto	MIT
cctz	Apache License 2.0
double-conversion	BSD 3-Clause License
FastMemcpy	MIT
googletest	BSD 3-Clause License
hyperscan	BSD 3-Clause License
libbtrie	BSD 2-Clause License
libcxxabi	BSD + MIT
libdivide	Zlib License
libgsasl	LGPL v2.1
libhdfs3	Apache License 2.0
libmetrohash	Apache License 2.0
libpcg-random	Apache License 2.0
libressl	OpenSSL License
librdkafka	BSD 2-Clause License
libwidechar_width	CC0 1.0 Universal
llvm	BSD 3-Clause License
lz4	BSD 2-Clause License
mariadb-connector-c	LGPL v2.1
murmurhash	Public Domain
pdqsort	Zlib License
poco	Boost Software License - Version 1.0
protobuf	BSD 3-Clause License
re2	BSD 3-Clause License
UnixODBC	LGPL v2.1

Библиотека	Лицензия
zlib-ng	Zlib License
zstd	BSD 3-Clause License

ClickHouse release 19.6.2.11, 2019-05-13

Новые возможности

- TTL выражения, позволяющие настроить время жизни и автоматическую очистку данных в таблице или в отдельных её столбцах. #4212 (Anton Popov)
- Добавлена поддержка алгоритма сжатия brotli в HTTP ответах (Accept-Encoding: br). Для тела POST запросов, эта возможность уже существовала. #4388 (Mikhail)
- Добавлена функция isValidUTF8 для проверки, содержит ли строка валидные данные в кодировке UTF-8. #4934 (Danila Kutenin)
- Добавлены новое правило балансировки (load_balancing) first_or_random по которому запросы посылаются на первый заданый хост и если он недоступен - на случайные хосты шарда. Полезно для топологий с кросс-репликацией. #5012 (nvartolomei)

Эксперемнтальные возможности

- Добавлена настройка index_granularity_bytes (адаптивная гранулярность индекса) для таблиц семейства MergeTree* . #4826 (alesapin)

Улучшения

- Добавлена поддержка для не константных и отрицательных значений аргументов смещения и длины для функции substringUTF8. #4989 (alexey-milovidov)
- Отключение push-down в правую таблицы в left join, левую таблицу в right join, и в обе таблицы в full join. Это исправляет неправильные JOIN результаты в некоторых случаях. #4846 (Ivan)
- clickhouse-copier: Автоматическая загрузка конфигурации задачи в zookeeper из --task-file опции #4876 (proller)
- Добавлены подсказки с учётом опечаток для имён движков таблиц и табличных функций. #4891 (Danila Kutenin)
- Поддержка выражений select * и select tablename.* для множественных join без подзапросов #4898 (Artem Zuikov)
- Сообщения об ошибках об отсутствующих столбцах стали более понятными. #4915 (Artem Zuikov)

Улучшение производительности

- Существенное ускорение ASOF JOIN #4924 (Martijn Bakker)

Обратно несовместимые изменения

- HTTP заголовок Query-Id переименован в X-ClickHouse-Query-Id для соответствия. #4972 (Mikhail)

Исправления ошибок

- Исправлены возможные разыменования нулевого указателя в clickhouse-copier. #4900 (proller)
- Исправлены ошибки в запросах с JOIN + ARRAY JOIN #4938 (Artem Zuikov)
- Исправлено зависание на старте сервера если внешний словарь зависит от другого словаря через использование таблицы из БД с движком Dictionary. #4962 (Vitaly Baranov)
- При использовании distributed_product_mode = 'local' корректно работает использование столбцов локальных таблиц в where/having/order by/... через табличные алиасы. Выкидывает исключение если таблица не имеет алиас. Доступ к столбцам без алиасов пока не возможен. #4986 (Artem Zuikov)
- Исправлен потенциально некорректный результат для SELECT DISTINCT с JOIN #5001 (Artem Zuikov)

Улучшения сборки/тестирования/пакетирования

- Исправлена неработоспособность тестов, если clickhouse-server запущен на удалённом хосте #4713 (Vasily Nemkov)
- clickhouse-test: Отключена раскраска результата, если команда запускается не в терминале. #4937 (alesapin)

- clickhouse-test: Возможность использования не только базы данных test #5008 (proller)
- Исправлены ошибки при запуске тестов под UBSan #5037 (Vitaly Baranov)
- Добавлен аллокатор Yandex LFAlloc для аллоцирования MarkCache и UncompressedCache данных разными способами для более надежного отлавливания проездов по памяти #4995 (Danila Kutenin)
- Утилита для упрощения бэкпортирования изменений в старые релизы и составления changelogs. #4949 (Ivan)

ClickHouse release 19.5.4.22, 2019-05-13

Исправления ошибок

- Исправлены возможные падения в bitmap* функциях #5220 #5228 (Andy Yang)
- Исправлен очень редкий data race condition который мог произойти при выполнении запроса с UNION ALL включающего минимум два SELECT из таблиц system.columns, system.tables, system.parts, system.parts_tables или таблиц семейства Merge и одновременно выполняющихся запросов ALTER столбцов соответствующих таблиц. #5189 (alexey-milovidov)
- Исправлена ошибка Set for IN is not created yet in case of using single LowCardinality column in the left part of IN. Эта ошибка возникала когда LowCardinality столбец была частью primary key. #5031 #5154 (Nikolai Kochetov)
- Исправление функции retention: только первое соответствующее условие добавлялось в состояние данных. Сейчас все условия которые удовлетворяют в строке данных добавляются в состояние. #5119 (小路)

ClickHouse release 19.5.3.8, 2019-04-18

Исправления ошибок

- Исправлен тип настройки max_partitions_per_insert_block с булевого на UInt64. #5028 (Mohammad Hossein Sekhavat)

ClickHouse release 19.5.2.6, 2019-04-15

Новые возможности

- Добавлены функции для работы с несколькими регулярными выражениями с помощью библиотеки Hyperscan. (multiMatchAny, multiMatchAnyIndex, multiFuzzyMatchAny, multiFuzzyMatchAnyIndex). #4780, #4841 (Danila Kutenin)
- Добавлена функция multiSearchFirstPosition. #4780 (Danila Kutenin)
- Реализована возможность указания построчного ограничения доступа к таблицам. #4792 (Ivan)
- Добавлен новый тип вторичного индекса на базе фильтра Блума (используется в функциях equal, in и like). #4499 (Nikita Vasilev)
- Добавлен ASOF JOIN которые позволяет джойнить строки по наиболее близкому известному значению. #4774 #4867 #4863 #4875 (Martijn Bakker, Artem Zuikov)
- Теперь запрос COMMA JOIN переписывается CROSS JOIN. И затем оба переписываются в INNER JOIN, если это возможно. #4661 (Artem Zuikov)

Улучшения

- Функции topK и topKWeighted теперь поддерживают произвольный loadFactor (исправляет issue #4252). #4634 (Kirill Danshin)
- Добавлена возможность использования настройки parallel_replicas_count > 1 для таблиц без семплирования (ранее настройка просто игнорировалась). #4637 (Alexey Elymanov)
- Поддержан запрос CREATE OR REPLACE VIEW. Позволяет создать VIEW или изменить запрос в одном выражении. #4654 (Boris Granveaud)
- Движок таблиц Buffer теперь поддерживает PREWHERE. #4671 (Yangkuan Liu)
- Теперь реплицируемые таблицы могу стартовать в readonly режиме даже при отсутствии zookeeper. #4691 (alesapin)
- Исправлено мигание прогресс-бара в clickhouse-client. Проблема была наиболее заметна при использовании FORMAT Null в потоковых запросах. #4811 (alexey-milovidov)

- Добавлена возможность отключения функций, использующих библиотеку hyperscan, для пользователей, чтобы ограничить возможное неконтролируемое потребление ресурсов. #4816 (alexey-milovidov)
- Добавлено логирование номера версии во все исключения. #4824 (proller)
- Добавлено ограничение на размер строк и количество параметров в функции multiMatch. Теперь они принимают строки умещающиеся в unsigned int. #4834 (Danila Kutenin)
- Улучшено использование памяти и обработка ошибок в Hyperscan. #4866 (Danila Kutenin)
- Теперь системная таблица system.graphite_detentions заполняется из конфигурационного файла для таблиц семейства *GraphiteMergeTree. #4584 (Mikhail f. Shiryaev)
- Функция trigramDistance переименована в функцию ngramDistance. Добавлено несколько функций с CaseInsensitive и UTF. #4602 (Danila Kutenin)
- Улучшено вычисление вторичных индексов. #4640 (Nikita Vasilev)
- Теперь обычные колонки, а также колонки DEFAULT, MATERIALIZED и ALIAS хранятся в одном списке (исправляет issue #2867). #4707 (Alex Zatelepin)

Исправления ошибок

- В случае невозможности выделить память вместо вызова std::terminate бросается исключение std::bad_alloc. #4665 (alexey-milovidov)
- Исправлены ошибки чтения capnproto из буфера. Иногда файлы не загружались по HTTP. #4674 (Vladislav)
- Исправлена ошибка Unknown log entry type: 0 после запроса OPTIMIZE TABLE FINAL. #4683 (Amos Bird)
- При передаче неправильных аргументов в hasAny и hasAll могла происходить ошибка сегментирования. #4698 (alexey-milovidov)
- Исправлен дедлок, который мог происходить при запросе DROP DATABASE dictionary. #4701 (alexey-milovidov)
- Исправлено неопределенное поведение в функциях median и quantile. #4702 (hcz)
- Исправлено определение уровня сжатия при указании настройки network_compression_method в нижнем регистре. Было сломано в v19.1. #4706 (proller)
- Настройка <timezone>UTC</timezone> больше не игнорируется (исправляет issue #4658). #4718 (proller)
- Исправлено поведение функции histogram с Distributed таблицами. #4741 (olegkv)
- Исправлено срабатывание thread-санитайзера с ошибкой destroy of a locked mutex. #4742 (alexey-milovidov)
- Исправлено срабатывание thread-санитайзера при завершении сервера, вызванное гонкой при использовании системных логов. Также исправлена потенциальная ошибка use-after-free при завершении сервера в котором был включен part_log. #4758 (alexey-milovidov)
- Исправлена перепроверка кусков в ReplicatedMergeTreeAlterThread при появлении ошибок. #4772 (Nikolai Kochetov)
- Исправлена работа арифметических операций с промежуточными состояниями агрегатных функций для константных аргументов (таких как результаты подзапросов). #4776 (alexey-milovidov)
- Теперь имена колонок всегда экранируются в файлах с метаинформацией. В противном случае было невозможно создать таблицу с колонкой с именем index. #4782 (alexey-milovidov)
- Исправлено падение в запросе ALTER ... MODIFY ORDER BY к Distributed таблице. #4790 (TCeason)
- Исправлена ошибка сегментирования при запросах с JOIN ON и включенной настройкой enable_optimize_predicate_expression. #4794 (Winter Zhang)
- Исправлено добавление лишней строки после чтения protobuf-сообщения из таблицы с движком Kafka. #4808 (Vitaly Baranov)
- Исправлено падение при запросе с JOIN ON с не nullable и nullable колонкой. Также исправлено поведение при появлении NULLs среди ключей справа вANY JOIN + join_use_nulls. #4815 (Artem Zuikov)
- Исправлена ошибка сегментирования в clickhouse-copier. #4835 (proller)
- Исправлена гонка при SELECT запросе из system.tables если таблица была конкурентно переименована или к ней был применен ALTER запрос. #4836 (alexey-milovidov)
- Исправлена гонка при скачивании куска, который уже является устаревшим. #4839 (alexey-milovidov)
- Исправлена редкая гонка при RENAME запросах к таблицам семейства MergeTree. #4844 (alexey-milovidov)
- Исправлена ошибка сегментирования в функции arrayIntersect. Ошибка возникала при вызове функции с константными и не константными аргументами. #4847 (Lixiang Qian)

- Исправлена редкая ошибка при чтении из колонки типа Array(LowCardinality), которая возникала, если в колонке содержалось большее количество подряд идущих пустых массивов. #4850 (Nikolai Kochetov)
- Исправлено паление в запроса с FULL/RIGHT JOIN когда объединение происходило по nullable и не nullable колонке. #4855 (Artem Zuikov)
- Исправлена ошибка No message received, возникавшая при скачивании кусков между репликами. #4856 (alesapin)
- Исправлена ошибка в функции arrayIntersect приводившая к неправильным результатам в случае нескольких повторяющихся значений в массиве. #4871 (Nikolai Kochetov)
- Исправлена гонка при конкурентных ALTER COLUMN запросах, которая могла приводить к падению сервера (исправляет issue #3421). #4592 (Alex Zatelepin)
- Исправлен некорректный результат в FULL/RIGHT JOIN запросах с константной колонкой. #4723 (Artem Zuikov)
- Исправлено появление дубликатов в GLOBAL JOIN со звездочкой. #4705 (Artem Zuikov)
- Исправлено определение параметров кодеков в запросах ALTER MODIFY, если тип колонки не был указан. #4883 (alesapin)
- Функции cutQueryStringAndFragment() и queryStringAndFragment() теперь работают корректно, когда URL содержит фрагмент, но не содержит запроса. #4894 (Vitaly Baranov)
- Исправлена редкая ошибка, возникавшая при установке настройки min_bytes_to_use_direct_io больше нуля. Она возникла при необходимости сдвинутся в файле, который уже прочитан до конца. #4897 (alesapin)
- Исправлено неправильное определение типов аргументов для агрегатных функций с LowCardinality аргументами (исправляет #4919). #4922 (Nikolai Kochetov)
- Исправлена неверная квалификация имён в GLOBAL JOIN. #4969 (Artem Zuikov)
- Исправлен результат функции toISOWeek для 1970 года. #4988 (alexey-milovidov)
- Исправлено дублирование DROP, TRUNCATE и OPTIMIZE запросов, когда они выполнялись ON CLUSTER для семейства таблиц ReplicatedMergeTree*. #4991 (alesapin)

Обратно несовместимые изменения

- Настройка insert_sample_with_metadata переименована в input_format_defaults_for_omitted_fields. #4771 (Artem Zuikov)
- Добавлена настройка max_partitions_per_insert_block (со значением по умолчанию 100). Если вставляемый блок содержит большое количество партиций, то бросается исключение. Лимит можно убрать выставив настройку в 0 (не рекомендуется). #4845 (alexey-milovidov)
- Функции мультипоиска были переименованы (multiPosition в multiSearchAllPositions, multiSearch в multiSearchAny, firstMatch в multiSearchFirstIndex). #4780 (Danila Kutenin)

Улучшение производительности

- Оптимизирован поиска с помощью алгоритма Volnitsky с помощью инлайнинга. Это дает около 5-10% улучшения производительности поиска для запросов ищущих множество слов или много одинаковых биграмм. #4862 (Danila Kutenin)
- Исправлено снижение производительности при выставлении настройки use_uncompressed_cache больше нуля для запросов, данные которых целиком лежат в кеше. #4913 (alesapin)

Улучшения сборки/тестирования/пакетирования

- Более строгие настройки для debug-сборок: более гранулярные маппинги памяти и использование ASLR; добавлена защита памяти для кеша засечек и индекса. Это позволяет найти больше ошибок порчи памяти, которые не обнаруживают address-санитайзер и thread-санитайзер. #4632 (alexey-milovidov)
- Добавлены настройки ENABLE_PROTOBUF, ENABLE_PARQUET и ENABLE_BROTLI которые позволяют отключить соответствующие компоненты. #4669 (Silviu Caragea)
- Теперь при зависании запросов во время работы тестов будет показан список запросов и стек-трейсы всех потоков. #4675 (alesapin)
- Добавлены ретраи при ошибке Connection loss в clickhouse-test. #4682 (alesapin)
- Добавлена возможность сборки под FreeBSD в packager-скрипт. #4712 #4748 (alesapin)
- Теперь при установке предлагается установить пароль для пользователя 'default'. #4725 (proller)

- Убраны предупреждения из библиотеки rdkafka при сборке. #4740 (alexey-milovidov)
- Добавлена возможность сборки без поддержки ssl. #4750 (proller)
- Добавлена возможность запускать докер-образ с clickhouse-server из под любого пользователя. #4753 (Mikhail f. Shiryaev)
- Boost обновлен до 1.69. #4793 (proller)
- Отключено использование mremap при сборке с thread-санитайзером, что приводило к ложным срабатываниям. Исправлены ошибки thread-санитайзера в stateful-тестах. #4859 (alexey-milovidov)
- Добавлен тест проверяющий использование схемы форматов для HTTP-интерфейса. #4864 (Vitaly Baranov)

ClickHouse release 19.4.4.33, 2019-04-17

Исправление ошибок

- В случае невозможности выделить память вместо вызова std::terminate бросается исключение std::bad_alloc. #4665 (alexey-milovidov)
- Исправлены ошибки чтения capnproto из буфера. Иногда файлы не загружались по HTTP. #4674 (Vladislav)
- Исправлена ошибка Unknown log entry type: 0 после запроса OPTIMIZE TABLE FINAL. #4683 (Amos Bird)
- При передаче неправильных аргументов в hasAny и hasAll могла происходить ошибка сегментирования. #4698 (alexey-milovidov)
- Исправлен дедлок, который мог происходить при запросе DROP DATABASE dictionary. #4701 (alexey-milovidov)
- Исправлено неопределенное поведение в функциях median и quantile. #4702 (hcz)
- Исправлено определение уровня сжатия при указании настройки network_compression_method в нижнем регистре. Было сломано в v19.1. #4706 (proller)
- Настройка <timezone>UTC</timezone> больше не игнорируется (исправляет issue #4658). #4718 (proller)
- Исправлено поведение функции histogram с Distributed таблицами. #4741 (olegkv)
- Исправлено срабатывание thread-санитайзера с ошибкой destroy of a locked mutex. #4742 (alexey-milovidov)
- Исправлено срабатывание thread-санитайзера при завершении сервера, вызванное гонкой при использовании системных логов. Также исправлена потенциальная ошибка use-after-free при завершении сервера в котором был включен part_log. #4758 (alexey-milovidov)
- Исправлена перепроверка кусков в ReplicatedMergeTreeAlterThread при появлении ошибок. #4772 (Nikolai Kochetov)
- Исправлена работа арифметических операций с промежуточными состояниями агрегатных функций для константных аргументов (таких как результаты подзапросов). #4776 (alexey-milovidov)
- Теперь имена колонок всегда экранируются в файлах с метаинформацией. В противном случае было невозможно создать таблицу с колонкой с именем index. #4782 (alexey-milovidov)
- Исправлено падение в запросе ALTER ... MODIFY ORDER BY к Distributed таблице. #4790 (TCeason)
- Исправлена ошибка сегментирования при запросах с JOIN ON и включенной настройкой enable_optimize_predicate_expression. #4794 (Winter Zhang)
- Исправлено добавление лишней строки после чтения protobuf-сообщения из таблицы с движком Kafka. #4808 (Vitaly Baranov)
- Исправлена ошибка сегментирования в clickhouse-copier. #4835 (proller)
- Исправлена гонка при SELECT запросе из system.tables если таблица была конкурентно переименована или к ней был применен ALTER запрос. #4836 (alexey-milovidov)
- Исправлена гонка при скачивании куска, который уже является устаревшим. #4839 (alexey-milovidov)
- Исправлена редкая гонка при RENAME запросах к таблицам семейства MergeTree. #4844 (alexey-milovidov)
- Исправлена ошибка сегментирования в функции arrayIntersect. Ошибка возникала при вызове функции с константными и не константными аргументами. #4847 (Lixiang Qian)
- Исправлена редкая ошибка при чтении из колонки типа Array(LowCardinality), которая возникала, если в колонке содержалось большее количество подряд идущих пустых массивов. #4850 (Nikolai Kochetov)
- Исправлена ошибка No message received, возникавшая при скачивании кусков между репликами. #4856 (alesapin)

- Исправлена ошибка в функции arrayIntersect приводившая к неправильным результатам в случае нескольких повторяющихся значений в массиве. #4871 (Nikolai Kochetov)
- Исправлена гонка при конкурентных ALTER COLUMN запросах, которая могла приводить к падению сервера (исправляет issue #3421). #4592 (Alex Zatelepin)
- Исправлено определение параметров кодеков в запросах ALTER MODIFY, если тип колонки не был указан. #4883 (alesapin)
- Функции cutQueryStringAndFragment() и queryStringAndFragment() теперь работают корректно, когда URL содержит фрагмент, но не содержит запроса. #4894 (Vitaly Baranov)
- Исправлена редкая ошибка, возникавшая при установке настройки min_bytes_to_use_direct_io больше нуля. Она возникла при необходимости сдвинутся в файле, который уже прочитан до конца. #4897 (alesapin)
- Исправлено неправильное определение типов аргументов для агрегатных функций с LowCardinality аргументами (исправляет #4919). #4922 (Nikolai Kochetov)
- Исправлен результат функции toISOWeek для 1970 года. #4988 (alexey-milovidov)
- Исправлено дублирование DROP, TRUNCATE и OPTIMIZE запросов, когда они выполнялись ON CLUSTER для семейства таблиц ReplicatedMergeTree*. #4991 (alesapin)

Улучшения

- Теперь обычные колонки, а также колонки DEFAULT, MATERIALIZED и ALIAS хранятся в одном списке (исправляет issue #2867). #4707 (Alex Zatelepin)

ClickHouse release 19.4.3.11, 2019-04-02

Исправление ошибок

- Исправлено паление в запроса с FULL/RIGHT JOIN когда объединение происходило по nullable и не nullable колонке. #4855 (Artem Zuikov)
- Исправлена ошибка сегментирования в clickhouse-copier. #4835 (proller)

Улучшения сборки/тестирования/пакетирования

- Добавлена возможность запускать докер-образ с clickhouse-server из под любого пользователя. #4753 (Mikhail f. Shiryaev)

ClickHouse release 19.4.2.7, 2019-03-30

Исправление ошибок

- Исправлена редкая ошибка при чтении из колонки типа Array(LowCardinality), которая возникала, если в колонке содержалось большее количество подряд идущих пустых массивов. #4850 (Nikolai Kochetov)

ClickHouse release 19.4.1.3, 2019-03-19

Исправление ошибок

- Исправлено поведение удаленных запросов, которые одновременно содержали LIMIT BY и LIMIT. Раньше для таких запросов LIMIT мог быть выполнен до LIMIT BY, что приводило к перефильтрации. #4708 (Constantin S. Pan)

ClickHouse release 19.4.0.49, 2019-03-09

Новые возможности

- Добавлена полная поддержка формата Protobuf (чтение и запись, вложенные структуры данных). #4174 #4493 (Vitaly Baranov)
- Добавлены функции для работы с битовыми масками с использованием библиотеки Roaring Bitmaps. #4207 (Andy Yang) #4568 (Vitaly Baranov)
- Поддержка формата Parquet #4448 (proller)
- Вычисление расстояния между строками с помощью подсчёта N-грам - для приближённого сравнения строк. Алгоритм похож на q-gram metrics в языке R. #4466 (Danila Kutenin)

- Движок таблиц GraphiteMergeTree поддерживает отдельные шаблоны для правил агрегации и для правил времени хранения. #4426 (Mikhail f. Shiryaev)
- Добавлены настройки max_execution_speed и max_execution_speed_bytes для того, чтобы ограничить потребление ресурсов запросами. Добавлена настройка min_execution_speed_bytes в дополнение к min_execution_speed. #4430 (Winter Zhang)
- Добавлена функция flatten - конвертация многомерных массивов в плоский массив. #4555 #4409 (alexey-milovidov, kzon)
- Добавлены функции arrayEnumerateDenseRanked и arrayEnumerateUniqRanked (похожа на arrayEnumerateUniq но позволяет указать глубину, на которую следует смотреть в многомерные массивы). #4475 (proller) #4601 (alexey-milovidov)
- Добавлена поддержка множества JOIN в одном запросе без подзапросов, с некоторыми ограничениями: без звёздочки и без алиасов сложных выражений в ON/WHERE/GROUP BY/... #4462 (Artem Zuikov)

Исправления ошибок

- Этот релиз также содержит все исправления из 19.3 и 19.1.
- Исправлена ошибка во вторичных индексах (экспериментальная возможность): порядок гранул при INSERT был неверным. #4407 (Nikita Vasilev)
- Исправлена работа вторичного индекса (экспериментальная возможность) типа set для столбцов типа Nullable и LowCardinality. Ранее их использование вызывало ошибку Data type must be deserialized with multiple streams при запросе SELECT. #4594 (Nikolai Kochetov)
- Правильное запоминание времени последнего обновления при полной перезагрузке словарей типа executable. #4551 (Tema Novikov)
- Исправлена неработоспособность прогресс-бара, возникшая в версии 19.3 #4627 (filimonov)
- Исправлены неправильные значения MemoryTracker, если кусок памяти был уменьшен в размере, в очень редких случаях. #4619 (alexey-milovidov)
- Исправлено undefined behaviour в ThreadPool #4612 (alexey-milovidov)
- Исправлено очень редкое падение с сообщением mutex lock failed: Invalid argument, которое могло произойти, если таблица типа MergeTree удалялась одновременно с SELECT. #4608 (Alex Zatelepin)
- Совместимость ODBC драйвера с типом данных LowCardinality #4381 (proller)
- Исправление ошибки AIOcontextPool: Found io_event with unknown id 0 под ОС FreeBSD #4438 (urgordeadbeef)
- Таблица system.part_log создавалась независимо от того, была ли она объявлена в конфигурации. #4483 (alexey-milovidov)
- Исправлено undefined behaviour в функции dictIsIn для словарей типа cache. #4515 (alesapin)
- Исправлен deadlock в случае, если запрос SELECT блокирует одну и ту же таблицу несколько раз (например - из разных потоков, либо при выполнении разных подзапросов) и одновременно с этим производится DDL запрос. #4535 (Alex Zatelepin)
- Настройка compile_expressions выключена по-умолчанию до тех пор, пока мы не зафиксируем исходники используемой библиотеки LLVM и не будем проверять её под ASan (сейчас библиотека LLVM берётся из системы). #4579 (alesapin)
- Исправлено падение по std::terminate, если invalidate_query для внешних словарей с источником clickhouse вернул неправильный результат (пустой; более чем одну строку; более чем один столбец). Исправлена ошибка, из-за которой запрос invalidate_query производился каждые пять секунд, независимо от указанного lifetime. #4583 (alexey-milovidov)
- Исправлен deadlock в случае, если запрос invalidate_query для внешнего словаря с источником clickhouse использовал таблицу system.dictionaries или базу данных типа Dictionary (редкий случай). #4599 (alexey-milovidov)
- Исправлена работа CROSS JOIN с пустым WHERE #4598 (Artem Zuikov)
- Исправлен segfault в функции replicate с константным аргументом. #4603 (alexey-milovidov)
- Исправлена работа predicate pushdown (настройка enable_optimize_predicate_expression) с лямбда-функциями. #4408 (Winter Zhang)
- Множественные исправления для множества JOIN в одном запросе. #4595 (Artem Zuikov)

Улучшения

- Поддержка алиасов в секции JOIN ON для правой таблицы #4412 (Artem Zuikov)

- Используются правильные алиасы в случае множественных JOIN с подзапросами. #4474 (Artem Zuikov)
- Исправлена логика работы predicate pushdown (настройка enable_optimize_predicate_expression) для JOIN. #4387 (Ivan)

Улучшения производительности

- Улучшена эвристика оптимизации "перенос в PREWHERE". #4405 (alexey-milovidov)
- Используются настоящие lookup таблицы вместо хэш-таблиц в случае 8 и 16 битных ключей. Интерфейс хэш-таблиц обобщён, чтобы поддерживать этот случай. #4536 (Amos Bird)
- Улучшена производительность сравнения строк. #4564 (alexey-milovidov)
- Очередь DDL операций (для запросов ON CLUSTER) очищается в отдельном потоке, чтобы не замедлять основную работу. #4502 (Alex Zatelepin)
- Даже если настройка min_bytes_to_use_direct_io выставлена в 1, не каждый файл открывался в режиме O_DIRECT, потому что размер файлов иногда недооценивался на размер одного сжатого блока. #4526 (alexey-milovidov)

Улучшения сборки/тестирования/пакетирования

- Добавлена поддержка компилятора clang-9 #4604 (alexey-milovidov)
- Исправлены неправильные __asm__ инструкции #4621 (Konstantin Podshumok)
- Добавлена поддержка задания настроек выполнения запросов для clickhouse-performance-test из командной строки. #4437 (alesapin)
- Тесты словарей перенесены в интеграционные тесты. #4477 (alesapin)
- В набор автоматизированных тестов производительности добавлены запросы, находящиеся в разделе "benchmark" на официальном сайте. #4496 (alexey-milovidov)
- Исправления сборки в случае использования внешних библиотек lz4 и xxhash. #4495 (Orivej Desh)
- Исправлен undefined behaviour, если функция quantileTiming была вызвана с отрицательным или нецелым аргументом (обнаружено с помощью fuzz test под undefined behaviour sanitizer). #4506 (alexey-milovidov)
- Исправлены опечатки в коде. #4531 (sdk2)
- Исправлена сборка под Mac. #4371 (Vitaly Baranov)
- Исправлена сборка под FreeBSD и для некоторых необычных конфигурациях сборки. #4444 (proller)

ClickHouse release 19.3.7, 2019-03-12

Исправления ошибок

- Исправлена ошибка в #3920. Ошибка проявлялась в виде случайных повреждений кэша (сообщения Unknown codec family code, Cannot seek through file) и segfault. Ошибка впервые возникла в 19.1 и присутствует во всех версиях до 19.1.10 и 19.3.6. #4623 (alexey-milovidov)

ClickHouse release 19.3.6, 2019-03-02

Исправления ошибок

- Если в пуле потоков было более 1000 потоков, то при выходе из потока, вызывается std::terminate. Azat Khuzhin #4485 #4505 (alexey-milovidov)
- Теперь возможно создавать таблицы ReplicatedMergeTree* с комментариями столбцов без указания DEFAULT, а также с CODEC но без COMMENT и DEFAULT. Исправлено сравнение CODEC друг с другом. #4523 (alesapin)
- Исправлено падение при JOIN по массивам и кортежам. #4552 (Artem Zuikov)
- Исправлено падение clickhouse-copier с сообщением ThreadStatus not created. #4540 (Artem Zuikov)
- Исправлено зависание сервера при завершении работы в случае использования распределённых DDL. #4472 (Alex Zatelepin)
- В сообщениях об ошибке при парсинге текстовых форматов, выдавались неправильные номера столбцов, в случае, если номер больше 10. #4484 (alexey-milovidov)

Улучшения сборки/тестирования/пакетирования

- Исправлена сборка с включенным AVX. #4527 (alexey-milovidov)

- Исправлена поддержка расширенных метрик выполнения запроса в случае, если ClickHouse был собран на системе с новым ядром Linux, а запускается на системе с существенно более старым ядром. #4541 (nvartolomei)
- Продолжение работы в случае невозможности применить настройку core_dump.size_limit с выводом предупреждения. #4473 (proller)
- Удалено inline для void readBinary(...) в Field.cpp. #4530 (hcz)

ClickHouse release 19.3.5, 2019-02-21

Исправления ошибок:

- Исправлена ошибка обработки длинных http-запросов на вставку на стороне сервера. #4454 (alesapin)
- Исправлена обратная несовместимость со старыми версиями, появившаяся из-за некорректной реализации настройки send_logs_level. #4445 (alexey-milovidov)
- Исправлена обратная несовместимость табличной функции remote, появившаяся из-за добавления комментариев колонок. #4446 (alexey-milovidov)

ClickHouse release 19.3.4, 2019-02-16

Улучшения:

- При выполнении запроса ATTACH TABLE при проверке ограничений на используемую память теперь не учитывается память, занимаемая индексом таблицы. Это позволяет избежать ситуации, когда невозможно сделать ATTACH TABLE после соответствующего DETACH TABLE. #4396 (alexey-milovidov)
- Немного увеличены ограничения на максимальный размер строки и массива, полученные от ZooKeeper. Это позволяет продолжать работу после увеличения настройки ZooKeeper CLIENT_JVMFLAGS=-Djute.maxbuffer=.... #4398 (alexey-milovidov)
- Теперь реплику, отключенную на длительный период, можно восстановить, даже если в её очереди скопилось огромное число записей. #4399 (alexey-milovidov)
- Для вторичных индексов типа set добавлен обязательный параметр (максимальное число хранимых значений). #4386 (Nikita Vasilev)

Исправления ошибок:

- Исправлен неверный результат запроса с модификатором WITH ROLLUP при группировке по единственному столбцу типа LowCardinality. #4384 (Nikolai Kochetov)
- Исправлена ошибка во вторичном индексе типа set (гранулы, в которых было больше, чем max_rows строк, игнорировались). #4386 (Nikita Vasilev)
- Исправлена подстановка alias-ов в запросах с подзапросом, содержащим этот же alias (#4110). #4351 (Artem Zuikov)

Улучшения сборки/тестирования/пакетирования:

- Множество исправлений для сборки под FreeBSD. #4397 (proller)
- Возможность запускать clickhouse-server для stateless тестов из docker-образа. #4347 (Vasily Nemkov)

ClickHouse release 19.3.3, 2019-02-13

Новые возможности:

- Добавлен запрос KILL MUTATION, который позволяет удалять мутации, которые по какой-то причине не могут выполниться. В таблицу system.mutations для облегчения диагностики добавлены столбцы latest_failed_part, latest_fail_time, latest_fail_reason. #4287 (Alex Zatelepin)
- Добавлена агрегатная функция entropy, которая вычисляет энтропию Шеннона. #4238 (Quid37)
- Добавлена обобщённая реализация функции arrayWithConstant. #4322 (alexey-milovidov)
- Добавлен оператор сравнения NOT BETWEEN. #4228 (Dmitry Naumov)
- Добавлена функция sumMapFiltered - вариант sumMap, позволяющий указать набор ключей, по которым будет производиться суммирование. #4129 (Léo Ercolanelli)
- Добавлена функция sumMapWithOverflow. #4151 (Léo Ercolanelli)
- Добавлена поддержка Nullable типов в табличной функции mysql. #4198 (Emmanuel Donin de Rosière)
- Добавлена поддержка произвольных константных выражений в секции LIMIT. #4246 (k3box)

- Добавлена агрегатная функция topKWeighted - вариант topK, позволяющий задавать (целый неотрицательный) вес добавляемого значения. #4245 (Andrew Golman)
- Движок Join теперь поддерживает настройку join_any_take_last_row, которая позволяет перезаписывать значения для существующих ключей. #3973 (Amos Bird)
- Добавлена функция toStartOfInterval. #4304 (Vitaly Baranov)
- Добавлена функция toStartOfTenMinutes. #4298 (Vitaly Baranov)
- Добавлен формат RowBinaryWithNamesAndTypes. #4200 (Oleg V. Kozlyuk)
- Добавлены типы IPv4 и IPv6. Более эффективная реализация функций IPv*. #3669 (Vasily Nemkov)
- Добавлен выходной формат Protobuf. #4005 #4158 (Vitaly Baranov)
- В HTTP-интерфейсе добавлена поддержка алгоритма сжатия brotli для вставляемых данных. #4235 (Mikhail)
- Клиент командной строки теперь подсказывает правильное имя, если пользователь опечатался в названии функции. #4239 (Danila Kutenin)
- В HTTP-ответ сервера добавлен заголовок Query-Id. #4231 (Mikhail)

Экспериментальные возможности:

- Добавлена поддержка вторичных индексов типа minmax и set для таблиц семейства MergeTree (позволяют быстро пропускать целые блоки данных). #4143 (Nikita Vasilev)
- Добавлена поддержка преобразования CROSS JOIN в INNER JOIN, если это возможно. #4221 #4266 (Artem Zuikov)

Исправления ошибок:

- Исправлена ошибка Not found column для случая дублирующихся столбцов в секции JOIN ON. #4279 (Artem Zuikov)
- Команда START REPLICATED SENDS теперь действительно включает посылку кусков данных при репликации. #4229 (nvartolomei)
- Исправлена агрегация столбцов типа Array(LowCardinality). #4055 (KochetovNicolai)
- Исправлена ошибка, приводившая к тому, что при исполнении запроса INSERT ... SELECT ... FROM file(...) терялась первая строчка файла, если он был в формате CSVWithNames или TSVWIthNames. #4297 (alexey-milovidov)
- Исправлено падение при перезагрузке внешнего словаря, если словарь недоступен. Ошибка возникла в 19.1.6. #4188 (proller)
- Исправлен неверный результат ALL JOIN, если в правой таблице присутствуют дубликаты ключа join. #4184 (Artem Zuikov)
- Исправлено падение сервера при включённой опции use_uncompressed_cache, а также исключение о неправильном размере разжатых данных. #4186 (alesapin)
- Исправлена ошибка, приводящая к неправильному результату сравнения больших (не помещающихся в Int16) дат при включённой настройке compile_expressions. #4341 (alesapin)
- Исправлен бесконечный цикл при запросе из табличной функции numbers(0). #4280 (alexey-milovidov)
- Временно отключён pushdown предикатов в подзапрос, если он содержит ORDER BY. #3890 (Winter Zhang)
- Исправлена ошибка Illegal instruction при использовании функций для работы с base64 на старых CPU. Ошибка проявлялась только, если ClickHouse был скомпилирован с gcc-8. #4275 (alexey-milovidov)
- Исправлена ошибка No message received при запросах к PostgreSQL через ODBC-драйвер и TLS-соединение, исправлен segfault при использовании MySQL через ODBC-драйвер. #4170 (alexey-milovidov)
- Исправлен неверный результат при использовании значений типа Date или DateTime в ветвях условного оператора (функции if). Функция if теперь работает для произвольного типа значений в ветвях. #4243 (alexey-milovidov)
- Словари с источником из локального ClickHouse теперь исполняются локально, а не используя TCP-соединение. #4166 (alexey-milovidov)
- Исправлено зависание запросов к таблице с движком File после того, как SELECT из этой таблицы завершился с ошибкой No such file or directory. #4161 (alexey-milovidov)
- Исправлена ошибка, из-за которой при запросе к таблице system.tables могло возникать исключение table doesn't exist. #4313 (alexey-milovidov)

- Исправлена ошибка, приводившая к падению clickhouse-client в интерактивном режиме, если успеть выйти из него во время загрузки подсказок командной строки. #4317 (alexey-milovidov)
- Исправлена ошибка, приводившая к неверным результатам исполнения мутаций, содержащих оператор IN. #4099 (Alex Zatelepin)
- Исправлена ошибка, из-за которой, если была создана база данных с движком Dictionary, все словари загружались при старте сервера, а словари с источником из локального ClickHouse не могли загрузиться. #4255 (alexey-milovidov)
- Исправлено повторное создание таблиц с системными логами (system.query_log, system.part_log) при остановке сервера. #4254 (alexey-milovidov)
- Исправлен вывод типа возвращаемого значения, а также использование блокировок в функции joinGet. #4153 (Amos Bird)
- Исправлено падение сервера при использовании настройки allow_experimental_multiple_joins_emulation. 52de2c (Artem Zuikov)
- Исправлено некорректное сравнение значений типа Date и DateTime. #4237 (valexey)
- Исправлена ошибка, проявлявшаяся при fuzz-тестировании с undefined behaviour-санитайзером: добавлена проверка типов параметров для семейства функций quantile*Weighted. #4145 (alexey-milovidov)
- Исправлена редкая ошибка, из-за которой при удалении старых кусков данных может возникать ошибка File not found. #4378 (alexey-milovidov)
- Исправлена установка пакета при отсутствующем файле /etc/clickhouse-server/config.xml. #4343 (proller)

Улучшения сборки/тестирования/пакетирования:

- При установке Debian-пакета символическая ссылка /etc/clickhouse-server/preprocessed теперь создаётся, учитывая пути, прописанные в конфигурационном файле. #4205 (proller)
- Исправления сборки под FreeBSD. #4225 (proller)
- Добавлена возможность создавать, заполнять и удалять таблицы в тестах производительности. #4220 (alesapin)
- Добавлен скрипт для поиска дублирующихся include-директив в исходных файлах. #4326 (alexey-milovidov)
- В тестах производительности добавлена возможность запускать запросы по номеру. #4264 (alesapin)
- Пакет с debug-символами добавлен в список рекомендованных для основного пакета. #4274 (alexey-milovidov)
- Рефакторинг утилиты performance-test. Улучшено логирование и обработка сигналов. #4171 (alesapin)
- Задокументирован анонимизированный датасет Яндекс.Метрики. #4164 (alesapin)
- Добавлен инструмент для конвертирования кусков данных таблиц, созданных с использованием старого синтаксиса с помесячным партиционированием, в новый формат. #4195 (Alex Zatelepin)
- Добавлена документация для двух датасетов, загруженных в s3. #4144 (alesapin)
- Добавлен инструмент, собирающий changelog из описаний pull request-ов. #4169 #4173 (KochetovNicolai) (KochetovNicolai)
- Добавлен puppet-модуль для Clickhouse. #4182 (Maxim Fedotov)
- Добавлена документация для нескольких недокументированных функций. #4168 (Winter Zhang)
- Исправления сборки под ARM. #4210#4306 #4291 (proller) (proller)
- Добавлена возможность запускать тесты словарей из ctest. #4189 (proller)
- Теперь директорией с SSL-сертификатами по умолчанию является /etc/ssl. #4167 (alexey-milovidov)
- Добавлена проверка доступности SSE и AVX-инструкций на старте. #4234 (Igr)
- Init-скрипт теперь дожидается, пока сервер запустится. #4281 (proller)

Обратно несовместимые изменения:

- Удалена настройка allow_experimental_low_cardinality_type. Семейство типов данных LowCardinality готово для использования в production. #4323 (alexey-milovidov)
- Размер кэша засечек и кэша разжатых блоков теперь уменьшается в зависимости от доступного объёма памяти. #4240 (Lopatin Konstantin)
- Для запроса CREATE TABLE добавлено ключевое слово INDEX. Имя столбца index теперь надо оборачивать в двойные или обратные кавычки: `index`. #4143 (Nikita Vasilev)

- Функция sumMap теперь возвращает тип с большей областью значений вместо переполнения. Если необходимо старое поведение, следует использовать добавленную функцию sumMapWithOverflow. #4151 (Léo Ercolanelli)

Улучшения производительности:

- Для запросов без секции LIMIT вместо std::sort теперь используется pdqsort. #4236 (Evgenii Pravda)
- Теперь сервер переиспользует потоки для выполнения запросов из глобального пула потоков. В краевых случаях это влияет на производительность. #4150 (alexey-milovidov)

Улучшения:

- Теперь, если в нативном протоколе послать запрос INSERT INTO tbl VALUES (.... (с данными в запросе), отдельно посылать разобранные данные для вставки не нужно. #4301 (alesapin)
- Добавлена поддержка AIO для FreeBSD. #4305 (urgordeadbeef)
- Запрос SELECT * FROM a JOIN b USING a, b теперь возвращает столбцы a и b только из левой таблицы. #4141 (Artem Zuikov)
- Добавлена опция командной строки -C для клиента, которая работает так же, как и опция -c. #4232 (syominsergey)
- Если для опции --password клиента командной строки не указано значение, пароль запрашивается из стандартного входа. #4230 (BSD_Conqueror)
- Добавлена подсветка метасимволов в строковых литералах, содержащих выражения для оператора LIKE и регулярные выражения. #4327 (alexey-milovidov)
- Добавлена отмена HTTP-запроса, если сокет клиента отваливается. #4213 (nvartolomei)
- Теперь сервер время от времени посылает пакеты Progress для поддержания соединения. #4215 (Ivan)
- Немного улучшено сообщение о причине, почему запрос OPTIMIZE не может быть исполнен (если включена настройка optimize_throw_if_noop). #4294 (alexey-milovidov)
- Добавлена поддержка опции --version для clickhouse-server. #4251 (Lopatin Konstantin)
- Добавлена поддержка опции --help/-h для clickhouse-server. #4233 (Yuriy Baranov)
- Добавлена поддержка скалярных подзапросов, возвращающих состояние агрегатной функции. #4348 (Nikolai Kochetov)
- Уменьшено время ожидания завершения сервера и завершения запросов ALTER. #4372 (alexey-milovidov)
- Добавлена информация о значении настройки replicated_can_become_leader в таблицу system.replicas. Добавлено логирование в случае, если реплика не собирается стать лидером. #4379 (Alex Zatelepin)

ClickHouse release 19.1.14, 2019-03-14

- Исправлена ошибка Column ... queried more than once, которая могла произойти в случае включенной настройки asterisk_left_columns_only в случае использования GLOBAL JOIN а также SELECT * (редкий случай). Эта ошибка изначально отсутствует в версиях 19.3 и более новых. 6bac7d8d (Artem Zuikov)

ClickHouse release 19.1.13, 2019-03-12

Этот релиз содержит такие же исправления ошибок, как и 19.3.7.

ClickHouse release 19.1.10, 2019-03-03

Этот релиз содержит такие же исправления ошибок, как и 19.3.6.

ClickHouse release 19.1.9, 2019-02-21

Исправления ошибок:

- Исправлена обратная несовместимость со старыми версиями, появившаяся из-за некорректной реализации настройки send_logs_level. #4445 (alexey-milovidov)
- Исправлена обратная несовместимость табличной функции remote, появившаяся из-за добавления комментариев колонок. #4446 (alexey-milovidov)

ClickHouse release 19.1.8, 2019-02-16

Исправления ошибок:

- Исправлена установка пакета при отсутствующем файле /etc/clickhouse-server/config.xml. #4343 (proller)

ClickHouse release 19.1.7, 2019-02-15

Исправления ошибок:

- Исправлен вывод типа возвращаемого значения, а также использование блокировок в функции joinGet. #4153 (Amos Bird)
- Исправлено повторное создание таблиц с системными логами (system.query_log, system.part_log) при остановке сервера. #4254 (alexey-milovidov)
- Исправлена ошибка, из-за которой, если была создана база данных с движком Dictionary, все словари загружались при старте сервера, а словари с источником из локального ClickHouse не могли загрузиться. #4255 (alexey-milovidov)
- Исправлена ошибка, приводившая к неверным результатам исполнения мутаций, содержащих оператор IN. #4099 (Alex Zatelepin)
- Исправлена ошибка, приводившая к падению clickhouse-client в интерактивном режиме, если успеть выйти из него во время загрузки подсказок командной строки. #4317 (alexey-milovidov)
- Исправлена ошибка, из-за которой при запросе к таблице system.tables могло возникать исключение table doesn't exist. #4313 (alexey-milovidov)
- Исправлено зависание запросов к таблице с движком File после того, как SELECT из этой таблицы завершился с ошибкой No such file or directory. #4161 (alexey-milovidov)
- Словари с источником из локального ClickHouse теперь исполняются локально, а не используя TCP-соединение. #4166 (alexey-milovidov)
- Исправлена ошибка No message received при запросах к PostgreSQL через ODBC-драйвер и TLS-соединение, исправлен segfault при использовании MySQL через ODBC-драйвер. #4170 (alexey-milovidov)
- Временно отключён pushdown предикатов в подзапрос, если он содержит ORDER BY. #3890 (Winter Zhang)
- Исправлен бесконечный цикл при запросе из табличной функции numbers(0). #4280 (alexey-milovidov)
- Исправлена ошибка, приводящая к неправильному результату сравнения больших (не помещающихся в Int16) дат при включённой настройке compile_expressions. #4341 (alesapin)
- Исправлено падение сервера при включённой опции uncompressed_cache, а также исключение о неправильном размере разжатых данных. #4186 (alesapin)
- Исправлен неверный результат ALL JOIN, если в правой таблице присутствуют дубликаты ключа join. #4184 (Artem Zuikov)
- Исправлена ошибка, приводившая к тому, что при исполнении запроса INSERT ... SELECT ... FROM file(...) терялась первая строчка файла, если он был в формате CSVWithNames или TSVWIthNames. #4297 (alexey-milovidov)
- Исправлена агрегация столбцов типа Array(LowCardinality). #4055 (KochetovNicolai)
- При установке Debian-пакета символическая ссылка /etc/clickhouse-server/preprocessed теперь создаётся, учитывая пути, прописанные в конфигурационном файле. #4205 (proller)
- Исправлена ошибка, проявлявшаяся при fuzz-тестировании с undefined behaviour-санитайзером: добавлена проверка типов параметров для семейства функций quantile*Weighted. #4145 (alexey-milovidov)
- Команда START REPLICATED SENDS теперь действительно включает посылку кусков данных при репликации. #4229 (nvartolomei)
- Исправлена ошибка Not found column для случая дублирующихся столбцов в секции JOIN ON. #4279 (Artem Zuikov)
- Теперь директорией с SSL-сертификатами по умолчанию является /etc/ssl. #4167 (alexey-milovidov)
- Исправлено падение при перезагрузке внешнего словаря, если словарь недоступен. Ошибка возникла в 19.1.6. #4188 (proller)
- Исправлено некорректное сравнение значений типа Date и DateTime. #4237 (valexey)

- Исправлен неверный результат при использовании значений типа Date или DateTime в ветвях условного оператора (функции if). Функция if теперь работает для произвольного типа значений в ветвях. #4243 (alexey-milovidov)

ClickHouse release 19.1.6, 2019-01-24

Новые возможности:

- Задание формата сжатия для отдельных столбцов. #3899 #4111 (alesapin, Winter Zhang, Anatoly)
- Формат сжатия Delta. #4052 (alesapin)
- Изменение формата сжатия запросом ALTER. #4054 (alesapin)
- Добавлены функции left, right, trim, ltrim, rtrim, timestampadd, timestampsub для совместимости со стандартом SQL. #3826 (Ivan Blinkov)
- Поддержка записи в движок HDFS и табличную функцию hdfs. #4084 (alesapin)
- Добавлены функции поиска набора константных строк в тексте: multiPosition, multiSearch ,firstMatch также с суффиксами -UTF8, -CaseInsensitive, и -CaseInsensitiveUTF8. #4053 (Danila Kutenin)
- Пропуск неиспользуемых шардов в случае, если запрос SELECT содержит фильтрацию по ключу шардирования (настройка optimize_skip_unused_shards). #3851 (Gleb Kanterov, Ivan)
- Пропуск строк в случае ошибки парсинга для движка Kafka (настройка kafka_skip_broken_messages). #4094 (Ivan)
- Поддержка применения мультиклассовых моделей CatBoost. Функция modelEvaluate возвращает кортеж в случае использования мультиклассовой модели. libcatboostmodel.so should be built with #607. #3959 (KochetovNicolai)
- Добавлены функции filesystemAvailable, filesystemFree, filesystemCapacity. #4097 (Boris Granveaud)
- Добавлены функции хеширования xxHash64 и xxHash32. #3905 (filimonov)
- Добавлена функция хеширования gccMurmurHash (GCC flavoured Murmur hash), использующая те же hash seed, что и gcc #4000 (sundyli)
- Добавлены функции хеширования javaHash, hiveHash. #3811 (shangshujie365)
- Добавлена функция remoteSecure. Функция работает аналогично remote, но использует безопасное соединение. #4088 (proller)

Экспериментальные возможности:

- Эмуляция запросов с несколькими секциями JOIN (настройка allow_experimental_multiple_joins_emulation). #3946 (Artem Zuikov)

Исправления ошибок:

- Ограничен размер кеша скомпилированных выражений в случае, если не указана настройка compiled_expression_cache_size для экономии потребляемой памяти. #4041 (alesapin)
- Исправлена проблема зависания потоков, выполняющих запрос ALTER для таблиц семейства Replicated, а также потоков, обновляющих конфигурацию из ZooKeeper. #2947 #3891 #3934 (Alex Zatelepin)
- Исправлен race condition в случае выполнения распределенной задачи запроса ALTER. Race condition приводил к состоянию, когда более чем одна реплика пыталась выполнить задачу, в результате чего все такие реплики, кроме одной, падали с ошибкой обращения к ZooKeeper. #3904 (Alex Zatelepin)
- Исправлена проблема обновления настройки from_zk. Настройка, указанная в файле конфигурации, не обновлялась в случае, если запрос к ZooKeeper падал по timeout. #2947 #3947 (Alex Zatelepin)
- Исправлена ошибка в вычислении сетевого префикса при указании IPv4 маски подсети. #3945 (alesapin)
- Исправлено падение (std::terminate) в редком сценарии, когда новый поток не мог быть создан из-за нехватки ресурсов. #3956 (alexey-milovidov)
- Исправлено падение табличной функции remote в случае, когда не удавалось получить структуру таблицы из-за ограничений пользователя. #4009 (alesapin)
- Исправлена утечка сетевых сокетов. Сокеты создавались в пуле и никогда не закрывались. При создании потока, создавались новые сокеты в случае, если все доступные использовались. #4017 (Alex Zatelepin)
- Исправлена проблема закрывания /proc/self/fd раньше, чем все файловые дескрипторы были прочитаны из /proc после создания процесса odbc-bridge. #4120 (alesapin)

- Исправлен баг в монотонном преобразовании String в UInt в случае использования String в первичном ключе. #3870 (Winter Zhang)
- Исправлен баг в вычислении монотонности функции преобразования типа целых значений. #3921 (alexey-milovidov)
- Исправлено падение в функциях arrayEnumerateUniq, arrayEnumerateDense при передаче невалидных аргументов. #3909 (alexey-milovidov)
- Исправлен undefined behavior в StorageMerge. #3910 (Amos Bird)
- Исправлено падение в функциях addDays, subtractDays. #3913 (alexey-milovidov)
- Исправлена проблема, в результате которой функции round, floor, trunc, ceil могли возвращать неверный результат для отрицательных целочисленных аргументов с большим значением. #3914 (alexey-milovidov)
- Исправлена проблема, в результате которой 'kill query sync' приводил к падению сервера. #3916 (muVulDeePecker)
- Исправлен баг, приводящий к большой задержке в случае пустой очереди репликации. #3928 #3932 (alesapin)
- Исправлено избыточное использование памяти в случае вставки в таблицу с LowCardinality в первичном ключе. #3955 (KochetovNicolai)
- Исправлена сериализация пустых массивов типа LowCardinality для формата Native. #3907 #4011 (KochetovNicolai)
- Исправлен неверный результат в случае использования distinct для числового столбца LowCardinality. #3895 #4012 (KochetovNicolai)
- Исправлена компиляция вычисления агрегатных функций для ключа LowCardinality (для случая, когда включена настройка compile). #3886 (KochetovNicolai)
- Исправлена передача пользователя и пароля для запросов с реплик. #3957 (alesapin) (小路)
- Исправлен очень редкий race condition возникающий при перечислении таблиц из базы данных типа Dictionary во время перезагрузки словарей. #3970 (alexey-milovidov)
- Исправлен неверный результат в случае использования HAVING с ROLLUP или CUBE. #3756 #3837 (Sam Chou)
- Исправлена проблема с алиасами столбцов для запросов с JOIN ON над распределенными таблицами. #3980 (Winter Zhang)
- Исправлена ошибка в реализации функции quantileTDigest (нашел Artem Vakhrushev). Эта ошибка никогда не происходит в ClickHouse и актуальна только для тех, кто использует кодовую базу ClickHouse напрямую в качестве библиотеки. #3935 (alexey-milovidov)

Улучшения:

- Добавлена поддержка IF NOT EXISTS в выражении ALTER TABLE ADD COLUMN, IF EXISTS в выражении DROP/MODIFY/CLEAR/COMMENT COLUMN. #3900 (Boris Granveaud)
- Функция parseDateTimeBestEffort теперь поддерживает форматы DD.MM.YYYY, DD.MM.YY, DD-MM-YYYY, DD-Mon-YYYY, DD/Month/YYYY и аналогичные. #3922 (alexey-milovidov)
- CapnProtoInputStream теперь поддерживает jagged структуры. #4063 (Odin Hultgren Van Der Horst)
- Улучшение usability: добавлена проверка, что сервер запущен от пользователя, совпадающего с владельцем директории данных. Запрещен запуск от пользователя root в случае, если root не владеет директорией с данными. #3785 (sergey-v-galtsev)
- Улучшена логика проверки столбцов, необходимых для JOIN, на стадии анализа запроса. #3930 (Artem Zuikov)
- Уменьшено число поддерживаемых соединений в случае большого числа распределенных таблиц. #3726 (Winter Zhang)
- Добавлена поддержка строки с totals для запроса с WITH TOTALS через ODBC драйвер. #3836 (Maksim Koritckiy)
- Поддержано использование Enum в качестве чисел в функции if. #3875 (Ivan)
- Добавлена настройка low_cardinality_allow_in_native_format. Если она выключена, то тип LowCadrinality не используется в формате Native. #3879 (KochetovNicolai)
- Удалены некоторые избыточные объекты из кеша скомпилированных выражений для уменьшения потребления памяти. #4042 (alesapin)
- Добавлена проверка того, что в запрос SET send_logs_level = 'value' передается верное значение. #3873 (Sabyanin Maxim)

- Добавлена проверка типов для функций преобразования типов. #3896 (Winter Zhang)

Улучшения производительности:

- Добавлена настройка use_minimalistic_part_header_in_zookeeper для движка MergeTree. Если настройка включена, Replicated таблицы будут хранить метаданные куска в компактном виде (в соответствующем znode для этого куска). Это может значительно уменьшить размер для ZooKeeper snapshot (особенно для таблиц с большим числом столбцов). После включения данной настройки будет невозможно сделать откат к версии, которая эту настройку не поддерживает. #3960 (Alex Zatelepin)
- Добавлена реализация функций sequenceMatch и sequenceCount на основе конечного автомата в случае, если последовательность событий не содержит условия на время. #4004 (Léo Ercolanelli)
- Улучшена производительность сериализации целых чисел. #3968 (Amos Bird)
- Добавлен zero left padding для PODArray. Теперь элемент с индексом -1 является валидным нулевым значением. Эта особенность используется для удаления условного выражения при вычислении оффсетов массивов. #3920 (Amos Bird)
- Откат версии jemalloc, приводящей к деградации производительности. #4018 (alexey-milovidov)

Обратно несовместимые изменения:

- Удалена недокументированная возможность ALTER MODIFY PRIMARY KEY, замененная выражением ALTER MODIFY ORDER BY. #3887 (Alex Zatelepin)
- Удалена функция shardByHash. #3833 (alexey-milovidov)
- Запрещено использование скалярных подзапросов с результатом, имеющим тип AggregateFunction. #3865 (Ivan)

Улучшения сборки/тестирования/пакетирования:

- Добавлена поддержка сборки под PowerPC (ppc64le). #4132 (Danila Kutenin)
- Функциональные stateful тесты запускаются на публично доступных данных. #3969 (alexey-milovidov)
- Исправлена ошибка, при которой сервер не мог запуститься с сообщением bash: /usr/bin/clickhouse-extract-from-config: Operation not permitted при использовании Docker или systemd-nspawn. #4136 (alexey-milovidov)
- Обновлена библиотека rdkafka до версии v1.0.0-RC5. Использована cppkafka на замену интерфейса языка C. #4025 (Ivan)
- Обновлена библиотека mariadb-client. Исправлена проблема, обнаруженная с использованием UBSan. #3924 (alexey-milovidov)
- Исправления для сборок с UBSan. #3926 #3021 #3948 (alexey-milovidov)
- Добавлены покоммитные запуски тестов с UBSan сборкой.
- Добавлены покоммитные запуски тестов со статическим анализатором PVS-Studio.
- Исправлены проблемы, найденные с использованием PVS-Studio. #4013 (alexey-milovidov)
- Исправлены проблемы совместимости glibc. #4100 (alexey-milovidov)
- Docker образы перемещены на Ubuntu 18.10, добавлена совместимость с glibc >= 2.28 #3965 (alesapin)
- Добавлена переменная окружения CLICKHOUSE_DO_NOT_CHOWN, позволяющая не делать shown директории для Docker образа сервера. #3967 (alesapin)
- Включены большинство предупреждений из -Weverything для clang. Включено -Wpedantic. #3986 (alexey-milovidov)
- Добавлены некоторые предупреждения, специфичные только для clang 8. #3993 (alexey-milovidov)
- При использовании динамической линковки используется libLLVM вместо библиотеки LLVM. #3989 (Orivej Desh)
- Добавлены переменные окружения для параметров TSan, UBSan, ASan в тестовом Docker образе. #4072 (alesapin)
- Debian пакет clickhouse-server будет рекомендовать пакет libcap2-bin для того, чтобы использовать утилиту setcap для настроек. Данный пакет опционален. #4093 (alexey-milovidov)
- Уменьшено время сборки, убраны ненужные включения заголовочных файлов. #3898 (proller)
- Добавлены тесты производительности для функций хеширования. #3918 (filimonov)
- Исправлены циклические зависимости библиотек. #3958 (proller)
- Улучшена компиляция при малом объеме памяти. #4030 (proller)

- Добавлен тестовый скрипт для воспроизведения деградации производительности в jemalloc. #4036 (alexey-milovidov)
- Исправления опечаток в комментариях и строковых литералах. #4122 (maiha)
- Исправления опечаток в комментариях. #4089 (Evgenii Pravda)

ClickHouse release 18.16.1, 2018-12-21

Исправления ошибок:

- Исправлена проблема, приводившая к невозможности обновить словари с источником ODBC. #3825, #3829
- JIT-компиляция агрегатных функций теперь работает с LowCardinality столбцами. #3838

Улучшения:

- Добавлена настройка low_cardinality_allow_in_native_format (по умолчанию включена). Если её выключить, столбцы LowCardinality в Native формате будут преобразовываться в соответствующий обычный тип при SELECT и из этого типа при INSERT. #3879

Улучшения сборки:

- Исправления сборки под macOS и ARM.

ClickHouse release 18.16.0, 2018-12-14

Новые возможности:

- Вычисление DEFAULT выражений для отсутствующих полей при загрузке данных в полуструктурированных форматах (JSONEachRow, TSKV) (требуется включить настройку запроса insert_sample_with_metadata). #3555
- Для запроса ALTER TABLE добавлено действие MODIFY ORDER BY для изменения ключа сортировки при одновременном добавлении или удалении столбца таблицы. Это полезно для таблиц семейства MergeTree, выполняющих дополнительную работу при слияниях, согласно этому ключу сортировки, как например, SummingMergeTree, AggregatingMergeTree и т. п. #3581 #3755
- Для таблиц семейства MergeTree появилась возможность указать различный ключ сортировки (ORDER BY) и индекс (PRIMARY KEY). Ключ сортировки может быть длиннее, чем индекс. #3581
- Добавлена табличная функция hdfs и движок таблиц HDFS для импорта и экспорта данных в HDFS. chenxing-xc
- Добавлены функции для работы с base64: base64Encode, base64Decode, tryBase64Decode. Alexander Krasheninnikov
- Для агрегатной функции uniqCombined появилась возможность настраивать точность работы с помощью параметра (выбирать количество ячеек HyperLogLog). #3406
- Добавлена таблица system.contributors, содержащая имена всех, кто делал коммиты в ClickHouse. #3452
- Добавлена возможность не указывать партицию для запроса ALTER TABLE ... FREEZE для бэкапа сразу всех партиций. #3514
- Добавлены функции dictGet, dictGetOrDefault без указания типа возвращаемого значения. Тип определяется автоматически из описания словаря. Amos Bird
- Возможность указания комментария для столбца в описании таблицы и изменения его с помощью ALTER. #3377
- Возможность чтения из таблицы типа Join в случае простых ключей. Amos Bird
- Возможность указания настроек join_use_nulls, max_rows_in_join, max_bytes_in_join, join_overflow_mode при создании таблицы типа Join. Amos Bird
- Добавлена функция joinGet, позволяющая использовать таблицы типа Join как словарь. Amos Bird
- Добавлены столбцы partition_key, sorting_key, primary_key, sampling_key в таблицу system.tables, позволяющие получить информацию о ключах таблицы. #3609
- Добавлены столбцы is_in_partition_key, is_in_sorting_key, is_in_primary_key, is_in_sampling_key в таблицу system.columns. #3609
- Добавлены столбцы min_time, max_time в таблицу system.parts. Эти столбцы заполняются, если ключ партиционирования является выражением от столбцов типа DateTime. Emmanuel Donin de Rosière

Исправления ошибок:

- Исправления и улучшения производительности для типа данных LowCardinality. GROUP BY по LowCardinality(Nullable(...)). Получение extremes значений. Выполнение функций высшего порядка. LEFT ARRAY JOIN. Распределённый GROUP BY. Функции, возвращающие Array. Выполнение ORDER BY. Запись в Distributed таблицы (nicelulu). Обратная совместимость для запросов INSERT от старых клиентов, реализующих Native протокол. Поддержка LowCardinality для JOIN. Производительность при работе в один поток. #3823 #3803 #3799 #3769 #3744 #3681 #3651 #3649 #3641 #3632 #3568 #3523 #3518
- Исправлена работа настройки select_sequential_consistency. Ранее, при включенной настройке, после начала записи в новую партицию, мог возвращаться неполный результат. #2863
- Корректное указание базы данных при выполнении DDL запросов ON CLUSTER, а также при выполнении ALTER UPDATE/DELETE. #3772 #3460
- Корректное указание базы данных для подзапросов внутри VIEW. #3521
- Исправлена работа PREWHERE с FINAL для VersionedCollapsingMergeTree. 7167bfd7
- Возможность с помощью запроса KILL QUERY отмены запросов, которые ещё не начали выполняться из-за ожидания блокировки таблицы. #3517
- Исправлены расчёты с датой и временем в случае, если стрелки часов были переведены назад в полночь (это происходит в Иране, а также было Москве с 1981 по 1983 год). Ранее это приводило к тому, что стрелки часов переводились на сутки раньше, чем нужно, а также приводило к некорректному форматированию даты-с-временем в текстовом виде. #3819
- Исправлена работа некоторых случаев VIEW и подзапросов без указания базы данных. Winter Zhang
- Исправлен race condition при одновременном чтении из MATERIALIZED VIEW и удалением MATERIALIZED VIEW из-за отсутствия блокировки внутренней таблицы MATERIALIZED VIEW. #3404 #3694
- Исправлена ошибка Lock handler cannot be nullptr. #3689
- Исправления выполнения запросов при включенной настройке compile_expressions (включена по-умолчанию) - убрана свёртка недетерминированных константных выражений, как например, функции now. #3457
- Исправлено падение при указании неконстантного аргумента scale в функциях toDecimal32/64/128.
- Исправлена ошибка при попытке вставки в формате Values массива с NULL элементами в столбец типа Array без Nullable (в случае input_format_values_interpret_expressions = 1). #3487 #3503
- Исправлено непрерывное логгирование ошибок в DDLWorker, если ZooKeeper недоступен. 8f50c620
- Исправлен тип возвращаемого значения для функций quantile* от аргументов типа Date и DateTime. #3580
- Исправлена работа секции WITH, если она задаёт простой алиас без выражений. #3570
- Исправлена обработка запросов с именованными подзапросами и квалифицированными именами столбцов при включенной настройке enable_optimize_predicate_expression. Winter Zhang
- Исправлена ошибка Attempt to attach to nullptr thread group при работе материализованных представлений. Marek Vavruša
- Исправлено падение при передаче некоторых некорректных аргументов в функцию arrayReverse. 73e3a7b6
- Исправлен buffer overflow в функции extractURLParameter. Увеличена производительность. Добавлена корректная обработка строк, содержащих нулевые байты. 141e9799
- Исправлен buffer overflow в функциях lowerUTF8, upperUTF8. Удалена возможность выполнения этих функций над аргументами типа FixedString. #3662
- Исправлен редкий race condition при удалении таблиц типа MergeTree. #3680
- Исправлен race condition при чтении из таблиц типа Buffer и одновременном ALTER либо DROP таблиц назначения. #3719
- Исправлен segfault в случае превышения ограничения max_temporary_non_const_columns. #3788

Улучшения:

- Обработанные конфигурационные файлы записываются сервером не в /etc/clickhouse-server/ директорию, а в директорию preprocessed_configs внутри path. Это позволяет оставить директорию /etc/clickhouse-server/ недоступной для записи пользователем clickhouse, что повышает безопасность. #2443

- Настройка min_merge_bytes_to_use_direct_io выставлена по-умолчанию в 10 GiB. Слияния, образующие крупные куски таблиц семейства MergeTree, будут производиться в режиме O_DIRECT, что исключает вымывание кэша. #3504
- Ускорен запуск сервера в случае наличия очень большого количества таблиц. #3398
- Добавлен пул соединений и HTTP Keep-Alive для соединения между репликами. #3594
- В случае ошибки синтаксиса запроса, в HTTP интерфейсе возвращается код 400 Bad Request (ранее возвращался код 500). 31bc680a
- Для настройки join_default_strictness выбрано значение по-умолчанию ALL для совместимости. 120e2cbe
- Убрано логгирование в stderr из библиотеки re2 в случае некорректных или сложных регулярных выражений. #3723
- Для движка таблиц Kafka: проверка наличия подписок перед началом чтения из Kafka; настройка таблицы kafka_max_block_size. Marek Vavruša
- Функции cityHash64, farmHash64, metroHash64, sipHash64, halfMD5, murmurHash2_32, murmurHash2_64, murmurHash3_32, murmurHash3_64 теперь работают для произвольного количества аргументов, а также для аргументов-кортежей. #3451 #3519
- Функция arrayReverse теперь работает с любыми типами массивов. 73e3a7b6
- Добавлен опциональный параметр - размер слота для функции timeSlots. Kirill Shvakov
- Для FULL и RIGHT JOIN учитывается настройка max_block_size для потока неприсоединённых данных из правой таблицы. Amos Bird
- В clickhouse-benchmark и clickhouse-performance-test добавлен параметр командной строки --secure для включения TLS. #3688 #3690
- Преобразование типов в случае, если структура таблицы типа Buffer не соответствует структуре таблицы назначения. Vitaly Baranov
- Добавлена настройка tcp_keep_alive_timeout для включения keep-alive пакетов после неактивности в течение указанного интервала времени. #3441
- Убрано излишнее квотирование значений ключа партиции в таблице system.parts, если он состоит из одного столбца. #3652
- Функция деления с остатком работает для типов данных Date и DateTime. #3385
- Добавлены синонимы функций POWER, LN, LCASE, UCASE, REPLACE, LOCATE, SUBSTR, MID. #3774 #3763 Некоторые имена функций сделаны регистронезависимыми для совместимости со стандартом SQL. Добавлен синтаксический сахар SUBSTRING(expr FROM start FOR length) для совместимости с SQL. #3804
- Добавлена возможность фиксации (mlock) страниц памяти, соответствующих исполняемому коду clickhouse-server для предотвращения вытеснения их из памяти. Возможность выключена по-умолчанию. #3553
- Увеличена производительность чтения с O_DIRECT (с включенной опцией min_bytes_to_use_direct_io). #3405
- Улучшена производительность работы функции dictGet...OrDefault в случае константного аргумента-ключа и неконстантного аргумента-default. Amos Bird
- В функции firstSignificantSubdomain добавлена обработка доменов gov, mil, edu. Igor Hatarist Увеличена производительность работы. #3628
- Возможность указания произвольных переменных окружения для запуска clickhouse-server посредством SYS-V init.d-скрипта с помощью указания CLICKHOUSE_PROGRAM_ENV в /etc/default/clickhouse. Pavlo Bashynskyi
- Правильный код возврата init-скрипта clickhouse-server. #3516
- В таблицу system.metrics добавлена метрика VersionInteger, а в system.build_options добавлена строчка VERSION_INTEGER, содержащая версию ClickHouse в числовом представлении, вида 18016000. #3644
- Удалена возможность сравнения типа Date с числом, чтобы избежать потенциальных ошибок вида date = 2018-12-17, где ошибочно не указаны кавычки вокруг даты. #3687
- Исправлено поведение функций с состоянием типа rowNumberInAllBlocks - раньше они выдавали число на единицу больше вследствие их запуска во время анализа запроса. Amos Bird
- При невозможности удалить файл force_restore_data, выводится сообщение об ошибке. Amos Bird

Улучшение сборки:

- Обновлена библиотека jemalloc, что исправляет потенциальную утечку памяти. Amos Bird
- Для debug сборок включено по-умолчанию профилирование jemalloc. 2cc82f5c

- Добавлена возможность запуска интеграционных тестов, при наличии установленным в системе лишь Docker. #3650
- Добавлен fuzz тест выражений в SELECT запросах. #3442
- Добавлен покоммитный стресс-тест, выполняющий функциональные тесты параллельно и в произвольном порядке, позволяющий обнаружить больше race conditions. #3438
- Улучшение способа запуска clickhouse-server в Docker образе. Elghazal Ahmed
- Для Docker образа добавлена поддержка инициализации базы данных с помощью файлов в директории /docker-entrypoint-initdb.d. Konstantin Lebedev
- Исправления для сборки под ARM. #3709

Обратно несовместимые изменения:

- Удалена возможность сравнения типа Date с числом, необходимо вместо toDate('2018-12-18') = 17883, использовать явное приведение типов = toDate(17883) #3687

ClickHouse release 18.14.19, 2018-12-19

Исправления ошибок:

- Исправлена проблема, приводившая к невозможности обновить словари с источником ODBC. #3825, #3829
- Исправлен segfault в случае превышения ограничения max_temporary_non_const_columns. #3788
- Корректное указание базы данных при выполнении DDL запросов ON CLUSTER. #3460

Улучшения сборки:

- Исправления сборки под ARM.

ClickHouse release 18.14.18, 2018-12-04

Исправления ошибок:

- Исправлена ошибка в функции dictGet... для словарей типа range, если один из аргументов константный, а другой - нет. #3751
- Исправлена ошибка, приводящая к выводу сообщений netlink: '...': attribute type 1 has an invalid length в логе ядра Linux, проявляющаяся на достаточно новых ядрах Linux. #3749
- Исправлен segfault при выполнении функции empty от аргумента типа FixedString. Daniel, Dao Quang Minh
- Исправлена избыточная аллокация памяти при большом значении настройки max_query_size (кусок памяти размера max_query_size выделялся сразу). #3720

Улучшения процесса сборки ClickHouse:

- Исправлена сборка с использованием библиотек LLVM/Clang версии 7 из пакетов ОС (эти библиотеки используются для динамической компиляции запросов). #3582

ClickHouse release 18.14.17, 2018-11-30

Исправления ошибок:

- Исправлена ситуация, при которой ODBC Bridge продолжал работу после завершения работы сервера ClickHouse. Теперь ODBC Bridge всегда завершает работу вместе с сервером. #3642
- Исправлена синхронная вставка в Distributed таблицу в случае явного указания неполного списка столбцов или списка столбцов в измененном порядке. #3673
- Исправлен редкий race condition, который мог привести к падению сервера при удалении MergeTree-таблиц. #3680
- Исправлен deadlock при выполнении запроса, возникающий если создание новых потоков выполнения невозможно из-за ошибки Resource temporarily unavailable. #3643
- Исправлена ошибка парсинга ENGINE при создании таблицы с синтаксисом AS table в случае, когда AS table указывался после ENGINE, что приводило к игнорированию указанного движка. #3692

ClickHouse release 18.14.15, 2018-11-21

Исправления ошибок:

- При чтении столбцов типа Array(String), размер требуемого куска памяти оценивался слишком большим, что приводило к исключению "Memory limit exceeded" при выполнении запроса. Ошибка появилась в версии 18.12.13. #3589

ClickHouse release 18.14.14, 2018-11-20

Исправления ошибок:

- Исправлена работа запросов ON CLUSTER в случае, когда в конфигурации кластера включено шифрование (флаг <secure>). #3599

Улучшения процесса сборки ClickHouse:

- Исправлены проблемы сборки (llvm-7 из системы, macos) #3582

ClickHouse release 18.14.13, 2018-11-08

Исправления ошибок:

- Исправлена ошибка Block structure mismatch in MergingSorted stream. #3162
- Исправлена работа запросов ON CLUSTER в случае, когда в конфигурации кластера включено шифрование (флаг <secure>). #3465
- Исправлена ошибка при использовании SAMPLE, PREWHERE и столбцов-алиасов. #3543
- Исправлена редкая ошибка unknown compression method при использовании настройки min_bytes_to_use_direct_io. 3544

Улучшения производительности:

- Исправлена деградация производительности запросов с GROUP BY столбцов типа Int16, Date на процессорах AMD EPYC. Игорь Лапко
- Исправлена деградация производительности при обработке длинных строк. #3530

Улучшения процесса сборки ClickHouse:

- Доработки для упрощения сборки в Arcadia. #3475, #3535

ClickHouse release 18.14.12, 2018-11-02

Исправления ошибок:

- Исправлена ошибка при join-запросе двух неименованных подзапросов. #3505
- Исправлена генерация пустой WHERE-части при запросах к внешним базам. hotid
- Исправлена ошибка использования неправильной настройки таймаута в ODBC-словарях. Marek Vavruša

ClickHouse release 18.14.11, 2018-10-29

Исправления ошибок:

- Исправлена ошибка Block structure mismatch in UNION stream: different number of columns в запросах с LIMIT. #2156
- Исправлены ошибки при слиянии данных в таблицах, содержащих массивы внутри Nested структур. #3397
- Исправлен неправильный результат запросов при выключенной настройке merge_tree_uniform_read_distribution (включена по умолчанию). #3429
- Исправлена ошибка при вставке в Distributed таблицу в формате Native. #3411

ClickHouse release 18.14.10, 2018-10-23

- Настройка compile_expressions (JIT компиляция выражений) выключена по умолчанию. #3410
- Настройка enable_optimize_predicate_expression выключена по умолчанию.

ClickHouse release 18.14.9, 2018-10-16

Новые возможности:

- Модификатор WITH CUBE для GROUP BY (также доступен синтаксис: GROUP BY CUBE(...)). #3172
- Добавлена функция formatDateTime. Alexandr Krasheninnikov
- Добавлен движок таблиц JDBC и табличная функция jdbc (для работы требуется установка clickhouse-jdbc-bridge). Alexandr Krasheninnikov
- Добавлены функции для работы с ISO номером недели: toISOWeek, toISOYear, toStartOfISOYear, а также toDayOfYear. #3146
- Добавлена возможность использования столбцов типа Nullable для таблиц типа MySQL, ODBC. #3362
- Возможность чтения вложенных структур данных как вложенных объектов в формате JSONEachRow. Добавлена настройка input_format_import_nested_json. Veloman Yunkan
- Возможность параллельной обработки многих MATERIALIZED VIEW при вставке данных. Настройка parallel_view_processing. Marek Vavruša
- Добавлен запрос SYSTEM FLUSH LOGS (форсированный сброс логов в системные таблицы, такие как например, query_log) #3321
- Возможность использования предопределённых макросов database и table в объявлении Replicated таблиц. #3251
- Добавлена возможность чтения значения типа Decimal в инженерной нотации (с указанием десятичной экспоненты). #3153

Экспериментальные возможности:

- Оптимизация GROUP BY для типов данных LowCardinality #3138
- Оптимизации вычисления выражений для типов данных LowCardinality #3200

Улучшения:

- Существенно уменьшено потребление памяти для запросов с ORDER BY и LIMIT. Настройка max_bytes_before_remerge_sort. #3205
- При отсутствии указания типа JOIN (LEFT, INNER, ...), подразумевается INNER JOIN. #3147
- Корректная работа квалифицированных звёздочек в запросах с JOIN. Winter Zhang
- Движок таблиц ODBC корректно выбирает способ квотирования идентификаторов в SQL диалекте удалённой СУБД. Alexandr Krasheninnikov
- Настройка compile_expressions (JIT компиляция выражений) включена по-умолчанию.
- Исправлено поведение при одновременном DROP DATABASE/TABLE IF EXISTS и CREATE DATABASE/TABLE IF NOT EXISTS. Ранее запрос CREATE DATABASE ... IF NOT EXISTS мог выдавать сообщение об ошибке вида "File ... already exists", а запросы CREATE TABLE ... IF NOT EXISTS и DROP TABLE IF EXISTS могли выдавать сообщение Table ... is creating or attaching right now. #3101
- Выражения LIKE и IN с константной правой частью пробрасываются на удалённый сервер при запросах из таблиц типа MySQL и ODBC. #3182
- Сравнения с константными выражениями в секции WHERE пробрасываются на удалённый сервер при запросах из таблиц типа MySQL и ODBC. Ранее пробрасывались только сравнения с константами. #3182
- Корректное вычисление ширины строк в терминале для Pretty форматов, в том числе для строк с иероглифами. Amos Bird.
- Возможность указания ON CLUSTER для запросов ALTER UPDATE.
- Увеличена производительность чтения данных в формате JSONEachRow. #3332
- Добавлены синонимы функций LENGTH, CHARACTER_LENGTH для совместимости. Функция CONCAT стала регистронезависимой. #3306
- Добавлен синоним TIMESTAMP для типа DateTime. #3390
- В логах сервера всегда присутствует место для query_id, даже если строчка лога не относится к запросу. Это сделано для более простого парсинга текстовых логов сервера сторонними инструментами.
- Логгирование потребления памяти запросом при превышении очередной отметки целого числа гигабайт. #3205

- Добавлен режим совместимости для случая, когда клиентская библиотека, работающая по Native протоколу, по ошибке отправляет меньшее количество столбцов, чем сервер ожидает для запроса INSERT. Такой сценарий был возможен при использовании библиотеки clickhouse-cpp. Ранее этот сценарий приводил к падению сервера. #3171
- В clickhouse-copier, в задаваемом пользователем выражении WHERE, появилась возможность использовать алиас partition_key (для дополнительной фильтрации по партициям исходной таблицы). Это полезно, если схема партиционирования изменяется при копировании, но изменяется незначительно. #3166
- Рабочий поток движка Kafka перенесён в фоновый пул потоков для того, чтобы автоматически уменьшать скорость чтения данных при большой нагрузке. Marek Vavruša.
- Поддержка чтения значений типа Tuple и Nested структур как struct в формате Cap'n'Proto Marek Vavruša.
- В список доменов верхнего уровня для функции firstSignificantSubdomain добавлен домен biz decaseal.
- В конфигурации внешних словарей, пустое значение null_value интерпретируется, как значение типа данных по-умолчанию. #3330
- Поддержка функций intDiv, intDivOrZero для Decimal. b48402e8
- Поддержка типов Date, DateTime, UUID, Decimal в качестве ключа для агрегатной функции sumMap. #3281
- Поддержка типа данных Decimal во внешних словарях. #3324
- Поддержка типа данных Decimal в таблицах типа SummingMergeTree. #3348
- Добавлена специализация для UUID в функции if. #3366
- Уменьшено количество системных вызовов open, close при чтении из таблиц семейства MergeTree #3283.
- Возможность выполнения запроса TRUNCATE TABLE на любой реплике (запрос пробрасывается на реплику-лидера). Kirill Shvakov

Исправление ошибок:

- Исправлена ошибка в работе таблиц типа Dictionary для словарей типа range_hashed. Ошибка возникла в версии 18.12.17. #1702
- Исправлена ошибка при загрузке словарей типа range_hashed (сообщение Unsupported type Nullable(...)). Ошибка возникла в версии 18.12.17. #3362
- Исправлена некорректная работа функции pointInPolygon из-за накопления погрешности при вычислениях для полигонов с большим количеством близко расположенных вершин. #3331 #3341
- Если после слияния кусков данных, у результирующего куска чексумма отличается от результата того же слияния на другой реплике, то результат слияния удаляется, и вместо этого кусок скачивается с другой реплики (это правильное поведение). Но после скачивания куска, он не мог добавиться в рабочий набор из-за ошибки, что кусок уже существует (так как кусок после слияния удалялся не сразу, а с задержкой). Это приводило к циклическим попыткам скачивания одних и тех же данных. #3194
- Исправлен некорректный учёт общего потребления оперативной памяти запросами (что приводило к неправильной работе настройки max_memory_usage_for_all_queries и неправильному значению метрики MemoryTracking). Ошибка возникла в версии 18.12.13. Marek Vavruša
- Исправлена работоспособность запросов CREATE TABLE ... ON CLUSTER ... AS SELECT ...Ошибка возникла в версии 18.12.13. #3247
- Исправлена лишняя подготовка структуры данных для JOIN на сервере-инициаторе запроса, если JOIN выполняется только на удалённых серверах. #3340
- Исправлены ошибки в движке Kafka: неработоспособность после исключения при начале чтения данных; блокировка при завершении Marek Vavruša.
- Для таблиц Kafka не передавался опциональный параметр schema (схема формата Cap'n'Proto). Vojtech Splichal
- Если ансамбль серверов ZooKeeper содержит серверы, которые принимают соединение, но сразу же разрывают его вместо ответа на рукопожатие, то ClickHouse выбирает для соединения другой сервер. Ранее в этом случае возникала ошибка Cannot read all data. Bytes read: 0. Bytes expected: 4.и сервер не мог стартовать. 8218cf3a
- Если ансамбль серверов ZooKeeper содержит серверы, для которых DNS запрос возвращает ошибку, то такие серверы пропускаются. 17b8e209

- Исправлено преобразование типов между Date и DateTime при вставке данных в формате VALUES (в случае, когда input_format_values_interpret_expressions = 1). Ранее преобразование производилось между числовым значением количества дней с начала unix эпохи и unix timestamp, что приводило к неожиданным результатам. #3229
- Исправление преобразования типов между Decimal и целыми числами. #3211
- Исправлены ошибки в работе настройки enable_optimize_predicate_expression. Winter Zhang
- Исправлена ошибка парсинга формата CSV с числами с плавающей запятой, если используется разделитель CSV не по-умолчанию, такой как например, ; #3155.
- Исправлена функция arrayCumSumNonNegative (она не накапливает отрицательные значения, если аккумулятор становится меньше нуля). Aleksey Studnev
- Исправлена работа Merge таблицы поверх Distributed таблиц при использовании PREWHERE. #3165
- Исправления ошибок в запросе ALTER UPDATE.
- Исправления ошибок в табличной функции odbc, которые возникли в версии 18.12. #3197
- Исправлена работа агрегатных функций с комбинаторами StateArray. #3188
- Исправлено падение при делении значения типа Decimal на ноль. 69dd6609
- Исправлен вывод типов для операций с использованием аргументов типа Decimal и целых чисел. #3224
- Исправлен segfault при GROUP BY по Decimal128. 3359ba06
- Настройка log_query_threads (логгирование информации о каждом потоке исполнения запроса) теперь имеет эффект только если настройка log_queries (логгирование информации о запросах) выставлена в 1. Так как настройка log_query_threads включена по-умолчанию, ранее информация о потоках логгировалась даже если логгирование запросов выключено. #3241
- Исправлена ошибка в распределённой работе агрегатной функции quantiles (сообщение об ошибке вида Not found column quantile...). 292a8855
- Исправлена проблема совместимости при одновременной работе на кластере серверов версии 18.12.17 и более старых, приводящая к тому, что при распределённых запросах с GROUP BY по ключам одновременно фиксированной и не фиксированной длины, при условии, что количество данных в процессе агрегации большое, могли возвращаться не до конца агрегированные данные (одни и те же ключи агрегации в двух разных строках). #3254
- Исправлена обработка подстановок в clickhouse-performance-test, если запрос содержит только часть из объявленных в тесте подстановок. #3263
- Исправлена ошибка при использовании FINAL совместно с PREWHERE. #3298
- Исправлена ошибка при использовании PREWHERE над столбцами, добавленными при ALTER. #3298
- Добавлена проверка отсутствия arrayJoin для DEFAULT, MATERIALIZED выражений. Ранее наличие arrayJoin приводило к ошибке при вставке данных. #3337
- Добавлена проверка отсутствия arrayJoin в секции PREWHERE. Ранее это приводило к сообщениям вида Size ... doesn't match или Unknown compression method при выполнении запросов. #3357
- Исправлен segfault, который мог возникать в редких случаях после оптимизации - замены цепочек AND из равенства выражения константам на соответствующее выражение IN. liuyimin-bytedance.
- Мелкие исправления clickhouse-benchmark: ранее информация о клиенте не передавалась на сервер; более корректный подсчёт числа выполненных запросов при завершении работы и для ограничения числа итераций. #3351 #3352

Обратно несовместимые изменения:

- Удалена настройка allow_experimental_decimal_type. Тип данных Decimal доступен для использования по-умолчанию. #3329

ClickHouse release 18.12.17, 2018-09-16

Новые возможности:

- invalidate_query (возможность задать запрос для проверки необходимости обновления внешнего словаря) реализована для источника clickhouse. #3126
- Добавлена возможность использования типов данных UInt*, Int*, DateTime (наравне с типом Date) в качестве ключа внешнего словаря типа range_hashed, определяющего границы диапазонов. Возможность использования NULL в качестве обозначения открытого диапазона. Vasily Nemkov
- Для типа Decimal добавлена поддержка агрегатных функций var*, stddev*. #3129

- Для типа Decimal добавлена поддержка математических функций (exp, sin и т. п.) #3129
- В таблицу system.part_log добавлен столбец partition_id. #3089

Исправление ошибок:

- Исправлена работа Merge таблицы поверх Distributed таблиц. Winter Zhang
- Исправлена несовместимость (лишняя зависимость от версии glibc), приводящая к невозможности запуска ClickHouse на Ubuntu Precise и более старых. Несовместимость возникла в версии 18.12.13. #3130
- Исправлены ошибки в работе настройки enable_optimize_predicate_expression. Winter Zhang
- Исправлено незначительное нарушение обратной совместимости, проявляющееся при одновременной работе на кластере реплик версий до 18.12.13 и создании новой реплики таблицы на сервере более новой версии (выдаётся сообщение Can not clone replica, because the ... updated to new ClickHouse version, что полностью логично, но не должно было происходить). #3122

Обратно несовместимые изменения:

- Настройка enable_optimize_predicate_expression включена по-умолчанию, что конечно очень оптимистично. При возникновении ошибок анализа запроса, связанных с поиском имён столбцов, следует выставить enable_optimize_predicate_expression в 0. Winter Zhang

ClickHouse release 18.12.14, 2018-09-13

Новые возможности:

- Добавлена поддержка запросов ALTER UPDATE. #3035
- Добавлена настройка allow_ddl, упраляющая доступом пользователя к DDL-запросам. #3104
- Добавлена настройка min_merge_bytes_to_use_direct_io для движков семейства MergeTree, позволяющая задать порог на суммарный размер слияния после которого работа с файлами кусков будет происходить с O_DIRECT. #3117
- В системную таблицу system.merges добавлен столбец partition_id. #3099

Улучшения

- Если в процессе мутации кусок остался неизменённым, он не будет скачан репликами. #3103
- При работе с clickhouse-client добавлено автодополнение для имён настроек. #3106

Исправление ошибок

- Добавлена проверка размеров массивов, которые являются элементами полей типа Nested, при вставке. #3118
- Исправлена ошибка обновления внешних словарей с источником ODBC и форматом хранения hashed. Ошибка возникла в версии 18.12.13.
- Исправлено падение при создании временной таблицы таблицы из запроса с условием IN. Winter Zhang
- Исправлена ошибка в работе агрегатных функций для массивов, элементами которых может быть NULL. Winter Zhang

ClickHouse release 18.12.13, 2018-09-10

Новые возможности:

- Добавлен тип данных DECIMAL(digits, scale) (Decimal32(scale), Decimal64(scale), Decimal128(scale)). Возможность доступна под настройкой allow_experimental_decimal_type. #2846 #2970 #3008 #3047
- Модификатор WITH ROLLUP для GROUP BY (также доступен синтаксис: GROUP BY ROLLUP(...)). #2948
- В запросах с JOIN, звёздочка раскрывается в список столбцов всех таблиц, в соответствии со стандартом SQL. Вернуть старое поведение можно, выставив настройку (уровня пользователя) asterisk_left_columns_only в значение 1. Winter Zhang
- Добавлена поддержка JOIN с табличной функцией. Winter Zhang
- Автодополнение по нажатию Tab в clickhouse-client. Sergey Shcherbin
- Нажатие Ctrl+C в clickhouse-client очищает запрос, если он был введён. #2877

- Добавлена настройка join_default_strictness (значения '', 'any', 'all'). Её использование позволяет не указывать ANY или ALL для JOIN. #2982
- В каждой строчке лога сервера, относящейся к обработке запроса, выводится идентификатор запроса. #2482
- Возможность получения логов выполнения запроса в clickhouse-client (настройка send_logs_level). При распределённой обработке запроса, логи отправляются каскадно со всех серверов. #2482
- В таблицах system.query_log и system.processes (SHOW PROCESSLIST) появилась информация о всех изменённых настройках при выполнении запроса (вложенная структура данных Settings). Добавлена настройка log_query_settings. #2482
- В таблицах system.query_log и system.processes появилась информация о номерах потоков, участвующих в исполнении запроса (столбец thread_numbers). #2482
- Добавлены счётчики ProfileEvents, измеряющие время, потраченное на чтение и запись по сети; чтение и запись на диск; количество сетевых ошибок; время потраченное на ожидании при ограничении сетевой полосы. #2482
- Добавлены счётчики ProfileEvents, содержащие системные метрики из rusage (позволяющие получить информацию об использовании CPU в userspace и ядре, page faults, context switches) а также метрики taskstats (позволяющие получить информацию о времени ожидания IO, CPU, а также количество прочитанных и записанных данных с учётом и без учёта page cache). #2482
- Счётчики ProfileEvents учитываются не только глобально, но и на каждый запрос, а также на каждый поток выполнения запроса, что позволяет детально профилировать потребление ресурсов отдельными запросами. #2482
- Добавлена таблица system.query_thread_log, содержащая информацию о каждом потоке выполнения запроса. Добавлена настройка log_query_threads. #2482
- В таблицах system.metrics и system.events появилась встроенная документация. #3016
- Добавлена функция arrayEnumerateDense. Amos Bird
- Добавлены функции arrayCumSumNonNegative и arrayDifference. Aleksey Studnev
- Добавлена агрегатная функция retention. Sundy Li
- Возможность сложения (слияния) состояний агрегатных функций с помощью оператора плюс, а также умножения состояний агрегатных функций на целую неотрицательную константу. #3062 #3034
- В таблицах семейства MergeTree добавлен виртуальный столбец _partition_id. #3089

Экспериментальные возможности:

- Добавлен тип данных LowCardinality(T). Тип данных автоматически создаёт локальный словарь значений и позволяет обрабатывать данные без распаковки словаря. #2830
- Добавлен кэш JIT-скомпилированных функций, а также счётчик числа использований перед компиляцией. Возможность JIT-компиляции выражений включается настройкой compile_expressions. #2990 #3077

Улучшения:

- Исправлена проблема неограниченного накопления лога репликации в случае наличия заброшенных реплик. Добавлен режим эффективного восстановления реплик после длительного отставания.
- Увеличена производительность при выполнении GROUP BY в случае, если есть несколько полей агрегации, одно из которых строковое, а другие - фиксированной длины.
- Увеличена производительность при использовании PREWHERE и при неявном переносе выражений в PREWHERE.
- Увеличена производительность парсинга текстовых форматов (CSV, TSV). Amos Bird #2980
- Увеличена производительность чтения строк и массивов в бинарных форматах. Amos Bird
- Увеличена производительность и уменьшено потребление памяти в запросах к таблицам system.tables и system.columns в случае наличия очень большого количества таблиц на одном сервере. #2953
- Исправлена проблема низкой производительности в случае наличия большого потока запросов, для которых возвращается ошибка (в perf top видна функция _dl_addr, при этом сервер использует мало CPU). #2938
- Прокидывание условий внутрь View (при включенной настройке enable_optimize_predicate_expression) Winter Zhang
- Доработки недостающей функциональности для типа данных UUID. #3074 #2985

- Тип данных UUID поддержан в словарях The-Alchemist. #2822
- Функция visitParamExtractRaw корректно работает с вложенными структурами. Winter Zhang
- При использовании настройки input_format_skip_unknown_fields корректно работает пропуск значений-объектов в формате JSONEachRow. BlahGeek
- Для выражения CASE с условиями, появилась возможность не указывать ELSE, что эквивалентно ELSE NULL. #2920
- Возможность конфигурирования operation timeout при работе с ZooKeeper. urykhy
- Возможность указания смещения для LIMIT n, m в виде LIMIT n OFFSET m. #2840
- Возможность использования синтаксиса SELECT TOP n в качестве альтернативы для LIMIT. #2840
- Увеличен размер очереди записи в системные таблицы, что позволяет уменьшить количество ситуаций SystemLog queue is full.
- В агрегатной функции windowFunnel добавлена поддержка событий, подходящих под несколько условий. Amos Bird
- Возможность использования дублирующихся столбцов в секции USING для JOIN. #3006
- Для форматов Pretty введено ограничение выравнивания столбцов по ширине. Настройка output_format_pretty_max_column_pad_width. В случае более широкого значения, оно всё ещё будет выведено целиком, но остальные ячейки таблицы не будут излишне широкими. #3003
- В табличной функции odbc добавлена возможность указания имени базы данных/схемы. Amos Bird
- Добавлена возможность использования имени пользователя, заданного в конфигурационном файле clickhouse-client. Vladimir Kozbin
- Счётчик ZooKeeperExceptions разделён на три счётчика ZooKeeperUserExceptions, ZooKeeperHardwareExceptions, ZooKeeperOtherExceptions.
- Запросы ALTER DELETE работают для материализованных представлений.
- Добавлена рандомизация во времени периодического запуска cleanup thread для таблиц типа ReplicatedMergeTree, чтобы избежать периодических всплесков нагрузки в случае очень большого количества таблиц типа ReplicatedMergeTree.
- Поддержка запроса ATTACH TABLE ... ON CLUSTER. #3025

Исправление ошибок:

- Исправлена ошибка в работе таблиц типа Dictionary (кидается исключение Size of offsets doesn't match size of column или Unknown compression method). Ошибка появилась в версии 18.10.3. #2913
- Исправлена ошибка при мерже данных таблиц типа CollapsingMergeTree, если один из кусков данных пустой (такие куски, в свою очередь, образуются при слиянии или при ALTER DELETE в случае удаления всех данных), и для слияния был выбран алгоритм vertical. #3049
- Исправлен race condition при DROP или TRUNCATE таблиц типа Memory при одновременном SELECT, который мог приводить к падениям сервера. Ошибка появилась в версии 1.1.54388. #3038
- Исправлена возможность потери данных при вставке в Replicated таблицы в случае получения ошибки Session expired (потеря данных может быть обнаружена по метрике ReplicatedDataLoss). Ошибка возникла в версии 1.1.54378. #2939 #2949 #2964
- Исправлен segfault при JOIN ... ON. #3000
- Исправлена ошибка поиска имён столбцов в случае, если выражение WHERE состоит целиком из квалифицированного имени столбца, как например WHERE table.column. #2994
- Исправлена ошибка вида "Not found column" при выполнении распределённых запросов в случае, если с удалённого сервера запрашивается единственный столбец, представляющий собой выражение IN с подзапросом. #3087
- Исправлена ошибка Block structure mismatch in UNION stream: different number of columns, возникающая при распределённых запросах, если один из шардов локальный, а другой - нет, и если при этом срабатывает оптимизация переноса в PREWHERE. #2226 #3037 #3055 #3065 #3073 #3090 #3093
- Исправлена работа функции pointInPolygon для некоторого случая невыпуклых полигонов. #2910
- Исправлен некорректный результат при сравнении nan с целыми числами. #3024
- Исправлена ошибка в библиотеке zlib-ng, которая могла приводить к segfault в редких случаях. #2854
- Исправлена утечка памяти при вставке в таблицу со столбцами типа AggregateFunction, если состояние агрегатной функции нетривиальное (выделяет память отдельно), и если в одном запросе на вставку получается несколько маленьких блоков. #3084
- Исправлен race condition при одновременном создании и удалении одной и той же таблицы типа Buffer или MergeTree.

- Исправлена возможность segfault при сравнении кортежей из некоторых нетривиальных типов, таких как, например, кортежей. #2989
- Исправлена возможность segfault при выполнении некоторых запросов ON CLUSTER. Winter Zhang
- Исправлена ошибка в функции arrayDistinct в случае Nullable элементов массивов. #2845 #2937
- Возможность enable_optimize_predicate_expression корректно поддерживает случаи с SELECT *. Winter Zhang
- Исправлена возможность segfault при переинициализации сессии с ZooKeeper. #2917
- Исправлена возможность блокировки при взаимодействии с ZooKeeper.
- Исправлен некорректный код суммирования вложенных структур данных в SummingMergeTree.
- При выделении памяти для состояний агрегатных функций, корректно учитывается выравнивание, что позволяет использовать при реализации состояний агрегатных функций операции, для которых выравнивание является необходимым. chenxing-xc

Исправления безопасности:

- Безопасная работа с ODBC источниками данных. Взаимодействие с ODBC драйверами выполняется через отдельный процесс clickhouse-odbc-bridge. Ошибки в сторонних ODBC драйверах теперь не приводят к проблемам со стабильностью сервера или уязвимостям. #2828 #2879 #2886 #2893 #2921
- Исправлена некорректная валидация пути к файлу в табличной функции catBoostPool. #2894
- Содержимое системных таблиц (tables, databases, parts, columns, parts_columns, merges, mutations, replicas, replication_queue) фильтруется согласно конфигурации доступа к базам данных для пользователя (allow_databases) Winter Zhang

Обратно несовместимые изменения:

- В запросах с JOIN, звёздочка раскрывается в список столбцов всех таблиц, в соответствии со стандартом SQL. Вернуть старое поведение можно, выставив настройку (уровня пользователя) asterisk_left_columns_only в значение 1.

Изменения сборки:

- Добавлен покоммитный запуск большинства интеграционных тестов.
- Добавлен покоммитный запуск проверки стиля кода.
- Корректный выбор реализации memcpy при сборке на CentOS7 / Fedora. Etienne Champetier
- При сборке с помощью clang добавлены некоторые warnings из -Weverything в дополнение к обычным -Wall -Wextra -Werror. #2957
- При debug сборке используется debug вариант jemalloc.
- Абстрагирован интерфейс библиотеки для взаимодействия с ZooKeeper. #2950

ClickHouse release 18.10.3, 2018-08-13

Новые возможности:

- Возможность использования HTTPS для репликации. #2760
- Добавлены функции murmurHash2_64, murmurHash3_32, murmurHash3_64, murmurHash3_128 в дополнение к имеющемуся murmurHash2_32. #2791
- Поддержка Nullable типов в ODBC драйвере ClickHouse (формат вывода ODBCDriver2) #2834
- Поддержка UUID в ключевых столбцах.

Улучшения:

- Удаление кластеров без перезагрузки сервера при их удалении из конфигурационных файлов. #2777
- Удаление внешних словарей без перезагрузки сервера при их удалении из конфигурационных файлов. #2779
- Добавлена поддержка SETTINGS для движка таблиц Kafka. Alexander Marshalov
- Доработки для типа данных UUID (не полностью) Šimon Podlipský. #2618
- Поддежка пустых кусков после мержей в движках SummingMergeTree, CollapsingMergeTree and VersionedCollapsingMergeTree. #2815
- Удаление старых записей о полностью выполнившихся мутациях (ALTER DELETE) #2784
- Добавлена таблица system.merge_tree_settings. Kirill Shvakov

- В таблицу system.tables добавлены столбцы зависимостей: dependencies_database и dependencies_table. Winter Zhang
- Добавлена опция конфига max_partition_size_to_drop. #2782
- Добавлена настройка output_format_json_escape_forward_slashes. Alexander Bocharov
- Добавлена настройка max_fetch_partition_retries_count. #2831
- Добавлена настройка prefer_localhost_replica, позволяющая отключить предпочтение локальной реплики и хождение на локальную реплику без межпроцессного взаимодействия. #2832
- Агрегатная функция quantileExact возвращает nan в случае агрегации по пустому множеству Float32/Float64 типов. Sundy Li

Исправление ошибок:

- Убрано излишнее экранирование параметров connection string для ODBC, котрое приводило к невозможности соединения. Ошибка возникла в версии 18.6.0.
- Исправлена логика обработки команд на REPLACE PARTITION в очереди репликации. Неправильная логика могла приводить к тому, что при наличии двух REPLACE одной и той же партиции, один из них оставался в очереди репликации и не мог выполниться. #2814
- Исправлена ошибка при мерже, если все куски были пустыми (такие куски, в свою очередь, образуются при слиянии или при ALTER DELETE в случае удаления всех данных). Ошибка появилась в версии 18.1.0. #2930
- Исправлена ошибка при параллельной записи в таблицы типа Set или Join. Amos Bird
- Исправлена ошибка Block structure mismatch in UNION stream: different number of columns, возникающая при запросах с UNION ALL внутри подзапроса, в случае, если один из SELECT запросов содержит дублирующиеся имена столбцов. Winter Zhang
- Исправлена утечка памяти в случае исключения при соединении с MySQL сервером.
- Исправлен некорректный код возврата clickhouse-client в случае ошибочного запроса
- Исправлен некорректная работа materialized views, содержащих DISTINCT. #2795

Обратно несовместимые изменения

- Убрана поддержка запросов CHECK TABLE для Distributed таблиц.

Изменения сборки:

- Заменен аллокатор, теперь используется jemalloc вместо tcmalloc. На некоторых сценариях ускорение достигает 20%. В то же время, существуют запросы, замедлившиеся до 20%. Потребление памяти на некоторых сценариях примерно на 10% меньше и более стабильно. При высококонкурентной нагрузке, потребление CPU в userspace и в system незначительно вырастает. #2773
- Использование libressl из submodule. #1983 #2807
- Использование unixodbc из submodule. #2789
- Использование mariadb-connector-c из submodule. #2785
- В репозиторий добавлены файлы функциональных тестов, рассчитывающих на наличие тестовых данных (пока без самих тестовых данных).

ClickHouse release 18.6.0, 2018-08-02

Новые возможности:

- Добавлена поддержка ON выражений для JOIN ON синтаксиса:
  JOIN ON Expr([table.]column, ...) = Expr([table.]column, ...) [AND Expr([table.]column, ...) = Expr([table.]column, ...) ...]
  Выражение должно представлять из себя цепочку равенств, объединенных оператором AND. Каждая часть равенства может являться произвольным выражением над столбцами одной из таблиц. Поддержана возможность использования fully qualified имен столбцов (table.name, database.table.name, table_alias.name, subquery_alias.name) для правой таблицы. #2742
- Добавлена возможность включить HTTPS для репликации. #2760

Улучшения:

- Сервер передаёт на клиент также patch-компонент своей версии. Данные о patch компоненте версии добавлены в system.processes и query_log. #2646

ClickHouse release 18.5.1, 2018-07-31

Новые возможности:

- Добавлена функция хеширования murmurHash2_32. #2756.

Улучшения:

- Добавлена возможность указывать значения в конфигурационных файлах из переменных окружения с помощью атрибута from_env. #2741.
- Добавлены регистронезависимые версии функций coalesce, ifNull, nullIf. #2752.

Исправление ошибок:

- Исправлена возможная ошибка при старте реплики. #2759.

ClickHouse release 18.4.0, 2018-07-28

Новые возможности:

- Добавлены системные таблицы formats, data_type_families, aggregate_function_combinators, table_functions, table_engines, collations #2721.
- Добавлена возможность использования табличной функции вместо таблицы в качестве аргумента табличной функции remote и cluster #2708.
- Поддержка HTTP Basic аутентификации в протоколе репликации #2727.
- В функции has добавлена возможность поиска в массиве значений типа Enum по числовому значению Maxim Khrisanfov.
- Поддержка добавления произвольных разделителей сообщений в процессе чтения из Kafka Amos Bird.

Улучшения:

- Запрос ALTER TABLE t DELETE WHERE не перезаписывает куски данных, которые не были затронуты условием WHERE #2694.
- Настройка use_minimalistic_checksums_in_zookeeper таблиц семейства ReplicatedMergeTree включена по-умолчанию. Эта настройка была добавлена в версии 1.1.54378, 2018-04-16. Установка версий, более старых, чем 1.1.54378, становится невозможной.
- Поддерживается запуск запросов KILL и OPTIMIZE с указанием ON CLUSTER Winter Zhang.

Исправление ошибок:

- Исправлена ошибка Column ... is not under aggregate function and not in GROUP BYв случае агрегации по выражению с оператором IN. Ошибка появилась в версии 18.1.0. (bbdd780b)
- Исправлена ошибка в агрегатной функции windowFunnel Winter Zhang.
- Исправлена ошибка в агрегатной функции anyHeavy (a2101df2)
- Исправлено падение сервера при использовании функции countArray().

Обратно несовместимые изменения:

- Список параметров для таблиц Kafka был изменён с Kafka(kafka_broker_list, kafka_topic_list, kafka_group_name, kafka_format[, kafka_schema, kafka_num_consumers]) на Kafka(kafka_broker_list, kafka_topic_list, kafka_group_name, kafka_format[, kafka_row_delimiter, kafka_schema, kafka_num_consumers]). Если вы использовали параметры kafka_schema или kafka_num_consumers, вам необходимо вручную отредактировать файлы с метаданными path/metadata/database/table.sql, добавив параметр kafka_row_delimiter со значением '' в соответствующее место.

ClickHouse release 18.1.0, 2018-07-23

Новые возможности:

- Поддержка запроса ALTER TABLE t DELETE WHERE для нереплицированных MergeTree-таблиц (#2634).
- Поддержка произвольных типов для семейства агрегатных функций uniq* (#2010).
- Поддержка произвольных типов в операторах сравнения (#2026).

- Возможность в users.xml указывать маску подсети в формате 10.0.0.1/255.255.255.0. Это необходимо для использования "дырявых" масок IPv6 сетей (#2637).
- Добавлена функция arrayDistinct (#2670).
- Движок SummingMergeTree теперь может работать со столбцами типа AggregateFunction (Constantin S. Pan).

Улучшения:

- Изменена схема версионирования релизов. Теперь первый компонент содержит год релиза (A.D.; по московскому времени; из номера вычитается 2000), второй - номер крупных изменений (увеличивается для большинства релизов), третий - патч-версия. Релизы по-прежнему обратно совместимы, если другое не указано в changelog.
- Ускорено преобразование чисел с плавающей точкой в строку (Amos Bird).
- Теперь, если при вставке из-за ошибок парсинга пропущено некоторое количество строк (такое возможно про включённых настройках input_allow_errors_num, input_allow_errors_ratio), это количество пишется в лог сервера (Leonardo Cecchi).

Исправление ошибок:

- Исправлена работа команды TRUNCATE для временных таблиц (Amos Bird).
- Исправлен редкий deadlock в клиентской библиотеке ZooKeeper, который возникал при сетевой ошибке во время вычитывания ответа (c315200).
- Исправлена ошибка при CAST в Nullable типы (#1322).
- Исправлен неправильный результат функции maxIntersection() в случае совпадения границ отрезков (Michael Furmur).
- Исправлено неверное преобразование цепочки OR-выражений в аргументе функции (chenxing-xc).
- Исправлена деградация производительности запросов, содержащих выражение IN (подзапрос) внутри другого подзапроса (#2571).
- Исправлена несовместимость серверов разных версий при распределённых запросах, использующих функцию CAST не в верхнем регистре (fe8c4d6).
- Добавлено недостающее квотирование идентификаторов при запросах к внешним СУБД (#2635).

Обратно несовместимые изменения:

- Не работает преобразование строки, содержащей число ноль, в DateTime. Пример: SELECT toDateTime('0'). По той же причине не работает DateTime DEFAULT '0' в таблицах, а также <null_value>0</null_value> в словарях. Решение: заменить 0 на 0000-00-00 00:00:00.

ClickHouse release 1.1.54394, 2018-07-12

Новые возможности:

- Добавлена агрегатная функция histogram (Михаил Сурин).
- Возможность использования OPTIMIZE TABLE ... FINAL без указания партиции для ReplicatedMergeTree (Amos Bird).

Исправление ошибок:

- Исправлена ошибка - выставление слишком маленького таймаута у сокетов (одна секунда) для чтения и записи при отправке и скачивании реплицируемых данных, что приводило к невозможности скачать куски достаточно большого размера при наличии некоторой нагрузки на сеть или диск (попытки скачивания кусков циклически повторяются). Ошибка возникла в версии 1.1.54388.
- Исправлена работа при использовании chroot в ZooKeeper, в случае вставки дублирующихся блоков данных в таблицу.
- Исправлена работа функции has для случая массива с Nullable элементами (#2115).
- Исправлена работа таблицы system.tables при её использовании в распределённых запросах; столбцы metadata_modification_time и engine_full сделаны невиртуальными; исправлена ошибка в случае, если из таблицы были запрошены только эти столбцы.
- Исправлена работа пустой таблицы типа TinyLog после вставки в неё пустого блока данных (#2563).
- Таблица system.zookeeper работает в случае, если значение узла в ZooKeeper равно NULL.

ClickHouse release 1.1.54390, 2018-07-06

Новые возможности:

- Возможность отправки запроса в формате multipart/form-data (в поле query), что полезно, если при этом также отправляются внешние данные для обработки запроса (Ольга Хвостикова).
- Добавлена возможность включить или отключить обработку одинарных или двойных кавычек при чтении данных в формате CSV. Это задаётся настройками format_csv_allow_single_quotes и format_csv_allow_double_quotes (Amos Bird)
- Возможность использования OPTIMIZE TABLE ... FINAL без указания партиции для не реплицированных вариантовMergeTree (Amos Bird).

Улучшения:

- Увеличена производительность, уменьшено потребление памяти, добавлен корректный учёт потребления памяти, при использовании оператора IN в случае, когда для его работы может использоваться индекс таблицы (#2584).
- Убраны избыточные проверки чексумм при добавлении куска. Это важно в случае большого количества реплик, так как в этом случае суммарное количество проверок было равно N^2.
- Добавлена поддержка аргументов типа Array(Tuple(...)) для функции arrayEnumerateUniq (#2573).
- Добавлена поддержка Nullable для функции runningDifference. (#2594)
- Увеличена производительность анализа запроса в случае очень большого количества выражений (#2572).
- Более быстрый выбор кусков для слияния в таблицах типа ReplicatedMergeTree. Более быстрое восстановление сессии с ZooKeeper. (#2597).
- Файл format_version.txt для таблиц семейства MergeTree создаётся заново при его отсутствии, что имеет смысл в случае запуска ClickHouse после копирования структуры директорий без файлов (Ciprian Hacman).

Исправление ошибок:

- Исправлена ошибка при работе с ZooKeeper, которая могла приводить к невозможности восстановления сессии и readonly состояниям таблиц до перезапуска сервера.
- Исправлена ошибка при работе с ZooKeeper, которая могла приводить к неудалению старых узлов при разрыве сессии.
- Исправлена ошибка в функции quantileTDigest для Float аргументов (ошибка появилась в версии 1.1.54388) (Михаил Сурин).
- Исправлена ошибка работы индекса таблиц типа MergeTree, если в условии, столбец первичного ключа расположен внутри функции преобразования типов между знаковым и беззнаковым целым одного размера (#2603).
- Исправлен segfault, если в конфигурационном файле нет macros, но они используются (#2570).
- Исправлено переключение на базу данных по-умолчанию при переподключении клиента (#2583).
- Исправлена ошибка в случае отключенной настройки use_index_for_in_with_subqueries.

Исправления безопасности:

- При соединениях с MySQL удалена возможность отправки файлов (LOAD DATA LOCAL INFILE).

ClickHouse release 1.1.54388, 2018-06-28

Новые возможности:

- Добавлена поддержка запроса ALTER TABLE t DELETE WHERE для реплицированных таблиц и таблица system.mutations.
- Добавлена поддержка запроса ALTER TABLE t [REPLACE|ATTACH] PARTITION для *MergeTree-таблиц.
- Добавлена поддержка запроса TRUNCATE TABLE (Winter Zhang)
- Добавлено несколько новых SYSTEM-запросов для реплицированных таблиц (RESTART REPLICAS, SYNC REPLICA, [STOP|START] [MERGES|FETCHES|REPLICATED SENDS|REPLICATION QUEUES]).
- Добавлена возможность записи в таблицу с движком MySQL и соответствующую табличную функцию (sundy-li).

- Добавлена табличная функция url() и движок таблиц URL (Александр Сапин).
- Добавлена агрегатная функция windowFunnel (sundy-li).
- Добавлены функции startsWith и endsWith для строк (Вадим Плахтинский).
- В табличной функции numbers() добавлена возможность указывать offset (Winter Zhang).
- Добавлена возможность интерактивного ввода пароля в clickhouse-client.
- Добавлена возможность отправки логов сервера в syslog (Александр Крашенинников).
- Добавлена поддержка логирования в словарях с источником shared library (Александр Сапин).
- Добавлена поддержка произвольного разделителя в формате CSV (Иван Жуков)
- Добавлена настройка date_time_input_format. Если переключить эту настройку в значение 'best_effort', значения DateTime будут читаться в широком диапазоне форматов.
- Добавлена утилита clickhouse-obfuscator для обфускации данных. Пример использования: публикация данных, используемых в тестах производительности.

Экспериментальные возможности:

- Добавлена возможность вычислять аргументы функции and только там, где они нужны (Анастасия Царькова)
- Добавлена возможность JIT-компиляции в нативный код некоторых выражений (pyos).

Исправление ошибок:

- Исправлено появление дублей в запросе с DISTINCT и ORDER BY.
- Запросы с ARRAY JOIN и arrayFilter раньше возвращали некорректный результат.
- Исправлена ошибка при чтении столбца-массива из Nested-структуры (#2066).
- Исправлена ошибка при анализе запросов с секцией HAVING вида HAVING tuple IN (...).
- Исправлена ошибка при анализе запросов с рекурсивными алиасами.
- Исправлена ошибка при чтении из ReplacingMergeTree с условием в PREWHERE, фильтрующим все строки (#2525).
- Настройки профиля пользователя не применялись при использовании сессий в HTTP-интерфейсе.
- Исправлено применение настроек из параметров командной строки в программе clickhouse-local.
- Клиентская библиотека ZooKeeper теперь использует таймаут сессии, полученный от сервера.
- Исправлена ошибка в клиентской библиотеке ZooKeeper, из-за которой ожидание ответа от сервера могло длиться дольше таймаута.
- Исправлено отсечение ненужных кусков при запросе с условием на столбцы ключа партиционирования (#2342).
- После CLEAR COLUMN IN PARTITION в соответствующей партиции теперь возможны слияния (#2315).
- Исправлено соответствие типов в табличной функции ODBC (sundy-li).
- Исправлено некорректное сравнение типов DateTime с таймзоной и без неё (Александр Бочаров).
- Исправлен синтаксический разбор и форматирование оператора CAST.
- Исправлена вставка в материализованное представление в случае, если движок таблицы представления - Distributed (Babacar Diassé).
- Исправлен race condition при записи данных из движка Kafka в материализованные представления (Yangkuan Liu).
- Исправлена SSRF в табличной функции remote().
- Исправлен выход из clickhouse-client в multiline-режиме (#2510).

Улучшения:

- Фоновые задачи в реплицированных таблицах теперь выполняются не в отдельных потоках, а в пуле потоков (Silviu Caragea)
- Улучшена производительность разжатия LZ4.
- Ускорен анализ запроса с большим числом JOIN-ов и подзапросов.
- DNS-кэш теперь автоматически обновляется при большом числе сетевых ошибок.
- Вставка в таблицу теперь не происходит, если вставка в одно из её материализованных представлений невозможна из-за того, что в нём много кусков.
- Исправлено несоответствие в значениях счётчиков событий Query, SelectQuery, InsertQuery.
- Разрешены выражения вида tuple IN (SELECT tuple), если типы кортежей совпадают.

- Сервер с реплицированными таблицами теперь может стартовать, даже если не сконфигурирован ZooKeeper.
- При расчёте количества доступных ядер CPU теперь учитываются ограничения cgroups (Atri Sharma).
- Добавлен chown директорий конфигов в конфигурационном файле systemd (Михаил Ширяев).

Изменения сборки:

- Добавлена возможность сборки компилятором gcc8.
- Добавлена возможность сборки llvm из submodule.
- Используемая версия библиотеки librdkafka обновлена до v0.11.4.
- Добавлена возможность использования библиотеки libcpuid из системы, используемая версия библиотеки обновлена до 0.4.0.
- Исправлена сборка с использованием библиотеки vectorclass (Babacar Diassé).
- Cmake теперь по умолчанию генерирует файлы для ninja (как при использовании `-G Ninja`).
- Добавлена возможность использования библиотеки libtinfo вместо libtermcap (Георгий Кондратьев).
- Исправлен конфликт заголовочных файлов в Fedora Rawhide (#2520).

Обратно несовместимые изменения:

- Убран escaping в форматах `Vertical` и `Pretty*`, удалён формат `VerticalRaw`.
- Если в распределённых запросах одновременно участвуют серверы версии 1.1.54388 или новее и более старые, то при использовании выражения `cast(x, 'Type')`, записанного без указания `AS`, если слово `cast` указано не в верхнем регистре, возникает ошибка вида `Not found column cast(0, 'UInt8') in block`. Решение: обновить сервер на всём кластере.

ClickHouse release 1.1.54385, 2018-06-01

Исправление ошибок:

- Исправлена ошибка, которая в некоторых случаях приводила к блокировке операций с ZooKeeper.

ClickHouse release 1.1.54383, 2018-05-22

Исправление ошибок:

- Исправлена деградация скорости выполнения очереди репликации при большом количестве реплик

ClickHouse release 1.1.54381, 2018-05-14

Исправление ошибок:

- Исправлена ошибка, приводящая к "утеканию" метаданных в ZooKeeper при потере соединения с сервером ZooKeeper.

ClickHouse release 1.1.54380, 2018-04-21

Новые возможности:

- Добавлена табличная функция `file(path, format, structure)`. Пример, читающий байты из `/dev/urandom`: `ln -s /dev/urandom /var/lib/clickhouse/user_files/random clickhouse-client -q "SELECT * FROM file('random', 'RowBinary', 'd UInt8') LIMIT 10"`.

Улучшения:

- Добавлена возможность оборачивать подзапросы скобками `()` для повышения читаемости запросов. Например: `(SELECT 1) UNION ALL (SELECT 1)`.
- Простые запросы `SELECT` из таблицы `system.processes` не учитываются в ограничении `max_concurrent_queries`.

Исправление ошибок:

- Исправлена неправильная работа оператора `IN` в `MATERIALIZED VIEW`.
- Исправлена неправильная работа индекса по ключу партиционирования в выражениях типа `partition_key_column IN (...)`.

- Исправлена невозможность выполнить OPTIMIZE запрос на лидирующей реплике после выполнения RENAME таблицы.
- Исправлены ошибки авторизации возникающие при выполнении запросов OPTIMIZE и ALTER на нелидирующей реплике.
- Исправлены зависания запросов KILL QUERY.
- Исправлена ошибка в клиентской библиотеке ZooKeeper, которая при использовании непустого префикса chroot в конфигурации приводила к потере watch'ей, остановке очереди distributed DDL запросов и замедлению репликации.

Обратно несовместимые изменения:

- Убрана поддержка выражений типа (a, b) IN (SELECT (a, b)) (можно использовать эквивалентные выражение (a, b) IN (SELECT a, b)). Раньше такие запросы могли приводить к недетерминированной фильтрации в WHERE.

ClickHouse release 1.1.54378, 2018-04-16

Новые возможности:

- Возможность изменения уровня логгирования без перезагрузки сервера.
- Добавлен запрос SHOW CREATE DATABASE.
- Возможность передать query_id в clickhouse-client (elBroom).
- Добавлена настройка max_network_bandwidth_for_all_users.
- Добавлена поддержка ALTER TABLE ... PARTITION ... для MATERIALIZED VIEW.
- Добавлена информация о размере кусков данных в несжатом виде в системные таблицы.
- Поддержка межсерверного шифрования для distributed таблиц (<secure>1</secure> в конфигурации реплики в <remote_servers>).
- Добавлена настройка уровня таблицы семейства ReplicatedMergeTree для уменьшения объема данных, хранимых в zookeeper: use_minimalistic_checksums_in_zookeeper = 1
- Возможность настройки приглашения clickhouse-client. По-умолчанию добавлен вывод имени сервера в приглашение. Возможность изменить отображаемое имя сервера. Отправка его в HTTP заголовке X-ClickHouse-Display-Name (Kirill Shvakov).
- Возможность указания нескольких topics через запятую для движка Kafka (Tobias Adamson)
- При остановке запроса по причине KILL QUERY или replace_running_query, клиент получает исключение Query was cancelled вместо неполного результата.

Улучшения:

- Запросы вида ALTER TABLE ... DROP/DETACH PARTITION выполняются впереди очереди репликации.
- Возможность использовать SELECT ... FINAL и OPTIMIZE ... FINAL даже в случае, если данные в таблице представлены одним куском.
- Пересоздание таблицы query_log налету в случае если было произведено её удаление вручную (Kirill Shvakov).
- Ускорение функции lengthUTF8 (zhang2014).
- Улучшена производительность синхронной вставки в Distributed таблицы (insert_distributed_sync = 1) в случае очень большого количества шардов.
- Сервер принимает настройки send_timeout и receive_timeout от клиента и применяет их на своей стороне для соединения с клиентом (в переставленном порядке: send_timeout у сокета на стороне сервера выставляется в значение receive_timeout принятое от клиента, и наоборот).
- Более надёжное восстановление после сбоев при асинхронной вставке в Distributed таблицы.
- Возвращаемый тип функции countEqual изменён с UInt32 на UInt64 (谢磊)

Исправление ошибок:

- Исправлена ошибка с IN где левая часть выражения Nullable.
- Исправлен неправильный результат при использовании кортежей с IN в случае, если часть компоненнтов кортежа есть в индексе таблицы.
- Исправлена работа ограничения max_execution_time с распределенными запросами.
- Исправлены ошибки при вычислении размеров составных столбцов в таблице system.columns.

- Исправлена ошибка при создании временной таблицы CREATE TEMPORARY TABLE IF NOT EXISTS
- Исправлены ошибки в StorageKafka (#2075)
- Исправлены падения сервера от некорректных аргументов некоторых аггрегатных функций.
- Исправлена ошибка, из-за которой запрос DETACH DATABASE мог не приводить к остановке фоновых задач таблицы типа ReplicatedMergeTree.
- Исправлена проблема с появлением Too many parts в агрегирующих материализованных представлениях (#2084).
- Исправлена рекурсивная обработка подстановок в конфиге, если после одной подстановки, требуется другая подстановка на том же уровне.
- Исправлена ошибка с неправильным синтаксисом в файле с метаданными при создании VIEW, использующих запрос с UNION ALL.
- Исправлена работа SummingMergeTree в случае суммирования вложенных структур данных с составным ключом.
- Исправлена возможность возникновения race condition при выборе лидера таблиц ReplicatedMergeTree.

Изменения сборки:

- Поддержка ninja вместо make при сборке. ninja используется по-умолчанию при сборке релизов.
- Переименованы пакеты clickhouse-server-base в clickhouse-common-static; clickhouse-server-common в clickhouse-server; clickhouse-common-dbg в clickhouse-common-static-dbg. Для установки используйте clickhouse-server clickhouse-client. Для совместимости, пакеты со старыми именами продолжают загружаться в репозиторий.

Обратно несовместимые изменения:

- Удалена специальная интерпретация выражения IN, если слева указан массив. Ранее выражение вида arr IN (set) воспринималось как "хотя бы один элемент arr принадлежит множеству set". Для получения такого же поведения в новой версии, напишите arrayExists(x -> x IN (set), arr).
- Отключено ошибочное использование опции сокета SO_REUSEPORT (которая по ошибке включена по-умолчанию в библиотеке Poco). Стоит обратить внимание, что на Linux системах теперь не имеет смысла указывать одновременно адреса :: и 0.0.0.0 для listen - следует использовать лишь адрес ::, который (с настройками ядра по-умолчанию) позволяет слушать соединения как по IPv4 так и по IPv6. Также вы можете вернуть поведение старых версий, указав в конфиге <listen_reuse_port>1</listen_reuse_port>.

ClickHouse release 1.1.54370, 2018-03-16

Новые возможности:

- Добавлена системная таблица system.macros и автоматическое обновление макросов при изменении конфигурационного файла.
- Добавлен запрос SYSTEM RELOAD CONFIG.
- Добавлена агрегатная функция maxIntersections(left_col, right_col), возвращающая максимальное количество одновременно пересекающихся интервалов [left; right]. Функция maxIntersectionsPosition(left, right) возвращает начало такого "максимального" интервала. (Michael Furmur).

Улучшения:

- При вставке данных в Replicated-таблицу делается меньше обращений к ZooKeeper (также из лога ZooKeeper исчезло большинство user-level ошибок).
- Добавлена возможность создавать алиасы для множеств. Пример: WITH (1, 2, 3) AS set SELECT number IN set FROM system.numbers LIMIT 10.

Исправление ошибок:

- Исправлена ошибка Illegal PREWHERE при чтении из Merge-таблицы над Distributed-таблицами.
- Добавлены исправления, позволяющие запускать clickhouse-server в IPv4-only docker-контейнерах.
- Исправлен race condition при чтении из системной таблицы system.parts_columns
- Убрана двойная буферизация при синхронной вставке в Distributed-таблицу, которая могла приводить к timeout-ам соединений.

- Исправлена ошибка, вызывающая чрезмерно долгое ожидание недоступной реплики перед началом выполнения SELECT.
- Исправлено некорректное отображение дат в таблице system.parts.
- Исправлена ошибка, приводящая к невозможности вставить данные в Replicated-таблицу, если в конфигурации кластера ZooKeeper задан непустой chroot.
- Исправлен алгоритм вертикального мержа при пустом ключе ORDER BY таблицы.
- Возвращена возможность использовать словари в запросах к удаленным таблицам, даже если этих словарей нет на сервере-инициаторе. Данная функциональность была потеряна в версии 1.1.54362.
- Восстановлено поведение, при котором в запросах типа SELECT * FROM remote('server2', default.table) WHERE col IN (SELECT col2 FROM default.table) в правой части IN должна использоваться удаленная таблица default.table, а не локальная. данное поведение было нарушено в версии 1.1.54358.
- Устранено ненужное Error-level логирование Not found column ... in block.

Релиз ClickHouse 1.1.54362, 2018-03-11

Новые возможности:

- Агрегация без GROUP BY по пустому множеству (как например, SELECT count(*) FROM table WHERE 0) теперь возвращает результат из одной строки с нулевыми значениями агрегатных функций, в соответствии со стандартом SQL. Вы можете вернуть старое поведение (возвращать пустой результат), выставив настройку empty_result_for_aggregation_by_empty_set в значение 1.
- Добавлено приведение типов при UNION ALL. Допустимо использование столбцов с разными алиасами в соответствующих позициях SELECT в UNION ALL, что соответствует стандарту SQL.
- Поддержка произвольных выражений в секции LIMIT BY. Ранее было возможно лишь использование столбцов - результата SELECT.
- Использование индекса таблиц семейства MergeTree при наличии условия IN на кортеж от выражений от столбцов первичного ключа. Пример WHERE (UserID, EventDate) IN ((123, '2000-01-01'), ...) (Anastasiya Tsarkova).
- Добавлен инструмент clickhouse-copier для межкластерного копирования и перешардирования данных (бета).
- Добавлены функции консистентного хэширования yandexConsistentHash, jumpConsistentHash, sumburConsistentHash. Их можно использовать в качестве ключа шардирования для того, чтобы уменьшить объём сетевого трафика при последующих перешардированиях.
- Добавлены функции arrayAny, arrayAll, hasAny, hasAll, arrayIntersect, arrayResize.
- Добавлена функция arrayCumSum (Javi Santana).
- Добавлена функция parseDateTimeBestEffort, parseDateTimeBestEffortOrZero, parseDateTimeBestEffortOrNull, позволяющая прочитать DateTime из строки, содержащей текст в широком множестве возможных форматов.
- Возможность частичной перезагрузки данных внешних словарей при их обновлении (загрузка лишь записей со значением заданного поля большим, чем при предыдущей загрузке) (Arsen Hakobyan).
- Добавлена табличная функция cluster. Пример: cluster(cluster_name, db, table). Табличная функция remote может принимать имя кластера в качестве первого аргумента, если оно указано в виде идентификатора.
- Возможность использования табличных функций remote, cluster в INSERT запросах.
- Добавлены виртуальные столбцы create_table_query, engine_full в таблице system.tables. Столбец metadata_modification_time сделан виртуальным.
- Добавлены столбцы data_path, metadata_path в таблицы system.tables иsystem.databases, а также столбец path в таблицы system.parts и system.parts_columns.
- Добавлена дополнительная информация о слияниях в таблице system.part_log.
- Возможность использования произвольного ключа партиционирования для таблицы system.query_log (Kirill Shvakov).
- Запрос SHOW TABLES теперь показывает также и временные таблицы. Добавлены временные таблицы и столбец is_temporary в таблице system.tables (zhang2014).
- Добавлен запрос DROP TEMPORARY TABLE, EXISTS TEMPORARY TABLE (zhang2014).
- Поддержка SHOW CREATE TABLE для временных таблиц (zhang2014).
- Добавлен конфигурационный параметр system_profile для настроек, используемых внутренними процессами.

- Поддержка загрузки object_id в качестве атрибута в словарях с источником MongoDB (Павел Литвиненко).
- Возможность читать null как значение по-умолчанию при загрузке данных для внешнего словаря с источником MongoDB (Павел Литвиненко).
- Возможность чтения значения типа DateTime в формате Values из unix timestamp без одинарных кавычек.
- Поддержан failover в табличной функции remote для случая, когда на части реплик отсутствует запрошенная таблица.
- Возможность переопределять параметры конфигурации в параметрах командной строки при запуске clickhouse-server, пример: clickhouse-server -- --logger.level=information.
- Реализована функция empty от аргумента типа FixedString: функция возвращает 1, если строка состоит полностью из нулевых байт (zhang2014).
- Добавлен конфигурационный параметр listen_try, позволяющий слушать хотя бы один из listen адресов и не завершать работу, если некоторые адреса не удаётся слушать (полезно для систем с выключенной поддержкой IPv4 или IPv6).
- Добавлен движок таблиц VersionedCollapsingMergeTree.
- Поддержка строк и произвольных числовых типов для источника словарей library.
- Возможность использования таблиц семейства MergeTree без первичного ключа (для этого необходимо указать ORDER BY tuple()).
- Добавлена возможность выполнить преобразование (CAST) Nullable типа в не Nullable тип, если аргумент не является NULL.
- Возможность выполнения RENAME TABLE для VIEW.
- Добавлена функция throwIf.
- Добавлена настройка odbc_default_field_size, позволяющая расширить максимальный размер значения, загружаемого из ODBC источника (по-умолчанию - 1024).
- В таблицу system.processes и в SHOW PROCESSLIST добавлены столбцы is_cancelled и peak_memory_usage.

Улучшения:

- Ограничения на результат и квоты на результат теперь не применяются к промежуточным данным для запросов INSERT SELECT и для подзапросов в SELECT.
- Уменьшено количество ложных срабатываний при проверке состояния Replicated таблиц при запуске сервера, приводивших к необходимости выставления флага force_restore_data.
- Добавлена настройка allow_distributed_ddl.
- Запрещено использование недетерминированных функций в выражениях для ключей таблиц семейства MergeTree.
- Файлы с подстановками из config.d директорий загружаются в алфавитном порядке.
- Увеличена производительность функции arrayElement в случае константного многомерного массива с пустым массивом в качестве одного из элементов. Пример: [[1], []][x].
- Увеличена скорость запуска сервера при использовании конфигурационных файлов с очень большими подстановками (например, очень большими списками IP-сетей).
- При выполнении запроса, табличные функции выполняются один раз. Ранее табличные функции remote, mysql дважды делали одинаковый запрос на получение структуры таблицы с удалённого сервера.
- Используется генератор документации MkDocs.
- При попытке удалить столбец таблицы, от которого зависят DEFAULT/MATERIALIZED выражения других столбцов, кидается исключение (zhang2014).
- Добавлена возможность парсинга пустой строки в текстовых форматах как числа 0 для Float типов данных. Эта возможность присутствовала раньше, но была потеряна в релизе 1.1.54342.
- Значения типа Enum можно использовать в функциях min, max, sum и некоторых других - в этих случаях используются соответствующие числовые значения. Эта возможность присутствовала ранее, но была потеряна в релизе 1.1.54337.
- Добавлено ограничение max_expanded_ast_elements действующее на размер AST после рекурсивного раскрытия алиасов.

Исправление ошибок:

- Исправлены случаи ошибочного удаления ненужных столбцов из подзапросов, а также отсутствие удаления ненужных столбцов из подзапросов, содержащих UNION ALL.
- Исправлена ошибка в слияниях для таблиц типа ReplacingMergeTree.
- Исправлена работа синхронного режима вставки в Distributed таблицы (insert_distributed_sync = 1).
- Исправлены segfault при некоторых случаях использования FULL и RIGHT JOIN с дублирующимися столбцами в подзапросах.
- Исправлены segfault, которые могут возникать при использовании функциональности replace_running_query и KILL QUERY.
- Исправлен порядок столбцов source и last_exception в таблице system.dictionaries.
- Исправлена ошибка - запрос DROP DATABASE не удалял файл с метаданными.
- Исправлен запрос DROP DATABASE для базы данных типа Dictionary.
- Исправлена неоправданно низкая точность работы функций uniqHLL12 и uniqCombined для кардинальностей больше 100 млн. элементов (Alex Bocharov).
- Исправлено вычисление неявных значений по-умолчанию при необходимости одновременного вычисления явных выражений по-умолчанию в запросах INSERT (zhang2014).
- Исправлен редкий случай, в котором запрос к таблице типа MergeTree мог не завершаться (chenxing-xc).
- Исправлено падение при выполнении запроса CHECK для Distributed таблиц, если все шарды локальные (chenxing.xc).
- Исправлена незначительная регрессия производительности при работе функций, использующих регулярные выражения.
- Исправлена регрессия производительности при создании многомерных массивов от сложных выражений.
- Исправлена ошибка, из-за которой в .sql файл с метаданными может записываться лишняя секция FORMAT.
- Исправлена ошибка, приводящая к тому, что ограничение max_table_size_to_drop действует при попытке удаления MATERIALIZED VIEW, смотрящего на явно указанную таблицу.
- Исправлена несовместимость со старыми клиентами (на старые клиенты могли отправляться данные с типом DateTime('timezone'), который они не понимают).
- Исправлена ошибка при чтении столбцов-элементов Nested структур, которые были добавлены с помощью ALTER, но являются пустыми для старых партиций, когда условия на такие столбцы переносятся в PREWHERE.
- Исправлена ошибка при фильтрации таблиц по условию на виртуальных столбец _table в запросах к таблицам типа Merge.
- Исправлена ошибка при использовании ALIAS столбцов в Distributed таблицах.
- Исправлена ошибка, приводящая к невозможности динамической компиляции запросов с агрегатными функциями из семейства quantile.
- Исправлен race condition в конвейере выполнения запроса, который мог проявляться в очень редких случаях при использовании Merge таблиц над большим количеством таблиц, а также при использовании GLOBAL подзапросов.
- Исправлено падение при передаче массивов разных размеров в функцию arrayReduce при использовании агрегатных функций от нескольких аргументов.
- Запрещено использование запросов с UNION ALL в MATERIALIZED VIEW.
- Исправлена ошибка, которая может возникать при инициализации системной таблицы part_log при старте сервера (по-умолчанию part_log выключен).

Обратно несовместимые изменения:

- Удалена настройка distributed_ddl_allow_replicated_alter. Соответствующее поведение включено по-умолчанию.
- Удалена настройка strict_insert_defaults. Если вы использовали эту функциональность, напишите на clickhouse-feedback@yandex-team.com.
- Удалён движок таблиц UnsortedMergeTree.

Релиз ClickHouse 1.1.54343, 2018-02-05

- Добавлена возможность использовать макросы при задании имени кластера в распределенных DLL запросах и создании Distributed-таблиц: CREATE TABLE distr ON CLUSTER '{cluster}' (...) ENGINE = Distributed('{cluster}', 'db', 'table').
- Теперь при вычислении запросов вида SELECT ... FROM table WHERE expr IN (subquery) используется индекс таблицы table.
- Улучшена обработка дубликатов при вставке в Replicated-таблицы, теперь они не приводят к излишнему замедлению выполнения очереди репликации.

Релиз ClickHouse 1.1.54342, 2018-01-22

Релиз содержит исправление к предыдущему релизу 1.1.54337: * Исправлена регрессия в версии 1.1.54337: если пользователь по-умолчанию имеет readonly доступ, то сервер отказывался стартовать с сообщением Cannot create database in readonly mode. * Исправлена регрессия в версии 1.1.54337: на системах под управлением systemd, логи по ошибке всегда записываются в syslog; watchdog скрипт по ошибке использует init.d. * Исправлена регрессия в версии 1.1.54337: неправильная конфигурация по-умолчанию в Docker образе. * Исправлена недетерминированная работа GraphiteMergeTree (в логах видно по сообщениям Data after merge is not byte-identical to data on another replicas). * Исправлена ошибка, в связи с которой запрос OPTIMIZE к Replicated таблицам мог приводить к неконсистентным мержам (в логах видно по сообщениям Part ... intersects previous part). * Таблицы типа Buffer теперь работают при наличии MATERIALIZED столбцов в таблице назначения (by zhang2014). * Исправлена одна из ошибок в реализации NULL.

Релиз ClickHouse 1.1.54337, 2018-01-18

Новые возможности:

- Добавлена поддержка хранения многомерных массивов и кортежей (тип данных Tuple) в таблицах.
- Поддержка табличных функций для запросов DESCRIBE и INSERT. Поддержка подзапроса в запросе DESCRIBE. Примеры: DESC TABLE remote('host', default.hits); DESC TABLE (SELECT 1); INSERT INTO TABLE FUNCTION remote('host', default.hits). Возможность писать INSERT INTO TABLE вместо INSERT INTO.
- Улучшена поддержка часовых поясов. В типе DateTime может быть указана таймзона, которая используется для парсинга и отображения данных в текстовом виде. Пример: DateTime('Europe/Moscow'). При указании таймзоны в функциях работы с DateTime, тип возвращаемого значения будет запоминать таймзону, для того, чтобы значение отображалось ожидаемым образом.
- Добавлены функции toTimeZone, timeDiff, toQuarter, toRelativeQuarterNum. В функцию toRelativeHour/Minute/Second можно передать аргумент типа Date. Имя функции now воспринимается без учёта регистра.
- Добавлена функция toStartOfFifteenMinutes (Kirill Shvakov).
- Добавлена программа clickhouse format для переформатирования запросов.
- Добавлен конфигурационный параметр format_schema_path (Marek Vavruša). Он используется для задания схемы для формата Cap'n'Proto. Файлы со схемой могут использоваться только из указанной директории.
- Добавлена поддержка incl и conf.d подстановок для конфигурации словарей и моделей (Pavel Yakunin).
- В таблице system.settings появилось описание большинства настроек (Kirill Shvakov).
- Добавлена таблица system.parts_columns, содержащая информацию о размерах столбцов в каждом куске данных MergeTree таблиц.
- Добавлена таблица system.models, содержащая информацию о загруженных моделях CatBoost.
- Добавлены табличные функции mysql и odbc и соответствующие движки таблиц MySQL, ODBC для обращения к удалённым базам данных. Функциональность в состоянии "бета".
- Для функции groupArray разрешено использование аргументов типа AggregateFunction (можно создать массив из состояний агрегатных функций).
- Удалены ограничения на использование разных комбинаций комбинаторов агрегатных функций. Для примера, вы можете использовать как функцию avgForEachIf, так и avgIfForEach, которые имеют разный смысл.
- Комбинатор агрегатных функций -ForEach расширен для случая агрегатных функций с более чем одним аргументом.

- Добавлена поддержка агрегатных функций от Nullable аргументов, для случаев, когда функция всегда возвращает не Nullable результат (реализовано с участием Silviu Caragea). Пример: groupArray, groupUniqArray, topK.
- Добавлен параметр командной строки max_client_network_bandwidth для clickhouse-client (Kirill Shvakov).
- Пользователям с доступом readonly = 2 разрешено работать с временными таблицами (CREATE, DROP, INSERT...) (Kirill Shvakov).
- Добавлена возможность указания количества consumers для Kafka. Расширена возможность конфигурации движка Kafka (Marek Vavruša).
- Добавлены функции intExp2, intExp10.
- Добавлена агрегатная функция sumKahan.
- Добавлены функции to*Number*OrNull, где *Number* - числовой тип.
- Добавлена поддержка секции WITH для запроса INSERT SELECT (автор: zhang2014).
- Добавлены настройки http_connection_timeout, http_send_timeout, http_receive_timeout. Настройки используются, в том числе, при скачивании кусков для репликации. Изменение этих настроек позволяет сделать более быстрый failover в случае перегруженной сети.
- Добавлена поддержка ALTER для таблиц типа Null (Anastasiya Tsarkova).
- Функция reinterpretAsString расширена на все типы данных, значения которых хранятся в памяти непрерывно.
- Для программы clickhouse-local добавлена опция --silent для подавления вывода информации о выполнении запроса в stderr.
- Добавлена поддержка чтения Date в текстовом виде в формате, где месяц и день месяца могут быть указаны одной цифрой вместо двух (Amos Bird).

Увеличение производительности:

- Увеличена производительность агрегатных функций min, max, any, anyLast, anyHeavy, argMin, argMax от строковых аргументов.
- Увеличена производительность функций isInfinite, isFinite, isNaN, roundToExp2.
- Увеличена производительность форматирования в текстовом виде и парсинга из текста значений типа Date и DateTime.
- Увеличена производительность и точность парсинга чисел с плавающей запятой.
- Уменьшено потребление памяти при JOIN, если левая и правая часть содержали столбцы с одинаковым именем, не входящие в USING.
- Увеличена производительность агрегатных функций varSamp, varPop, stddevSamp, stddevPop, covarSamp, covarPop, corr за счёт уменьшения стойкости к вычислительной погрешности. Старые версии функций добавлены под именами varSampStable, varPopStable, stddevSampStable, stddevPopStable, covarSampStable, covarPopStable, corrStable.

Исправления ошибок:

- Исправлена работа дедупликации блоков после DROP или DETATH PARTITION. Раньше удаление партиции и вставка тех же самых данных заново не работала, так как вставленные заново блоки считались дубликатами.
- Исправлена ошибка, в связи с которой может неправильно обрабатываться WHERE для запросов на создание MATERIALIZED VIEW с указанием POPULATE.
- Исправлена ошибка в работе параметра root_path в конфигурации zookeeper_servers.
- Исправлен неожиданный результат при передаче аргумента типа Date в функцию toStartOfDay.
- Исправлена работа функции addMonths, subtractMonths, арифметика с INTERVAL n MONTH, если в результате получается предыдущий год.
- Добавлена недостающая поддержка типа данных UUID для DISTINCT, JOIN, в агрегатных функциях uniq и во внешних словарях (Иванов Евгений). Поддержка UUID всё ещё остаётся не полной.
- Исправлено поведение SummingMergeTree для строк, в которых все значения после суммирования равны нулю.
- Многочисленные доработки для движка таблиц Kafka (Marek Vavruša).
- Исправлена некорректная работа движка таблиц Join (Amos Bird).
- Исправлена работа аллокатора под FreeBSD и OS X.
- Функция extractAll теперь может доставать пустые вхождения.

- Исправлена ошибка, не позволяющая подключить при сборке libressl вместо openssl.
- Исправлена работа CREATE TABLE AS SELECT из временной таблицы.
- Исправлена неатомарность обновления очереди репликации. Эта проблема могла приводить к рассинхронизации реплик и чинилась при перезапуске.
- Исправлено переполнение в функциях gcd, lcm, modulo (оператор %) (Maks Skorokhod).
- Файлы -preprocessed теперь создаются после изменения umask (umask может быть задан в конфигурационном файле).
- Исправлена ошибка фоновой проверки кусков (MergeTreePartChecker) при использовании партиционирования по произвольному ключу.
- Исправлен парсинг кортежей (значений типа Tuple) в текстовых форматах.
- Исправлены сообщения о неподходящих типах аргументов для функций multiIf, array и некоторых других.
- Переработана поддержка Nullable типов. Исправлены ошибки, которые могут приводить к падению сервера. Исправлено подавляющее большинство других ошибок, связанных с поддержкой NULL: неправильное приведение типов при INSERT SELECT, недостаточная поддержка Nullable в HAVING и в PREWHERE, режим join_use_nulls, Nullable типы в операторе OR и т. п.
- Исправлена работа с внутренними свойствами типов данных, что позволило исправить проблемы следующего вида: ошибочное суммирование полей типа Enum в SummingMergeTree; значения типа Enum ошибочно выводятся с выравниванием по правому краю в таблицах в Pretty форматах, и т. п.
- Более строгие проверки для допустимых комбинаций составных столбцов - это позволило исправить ошибок, которые могли приводить к падениям.
- Исправлено переполнение при задании очень большого значения параметра для типа FixedString.
- Исправлена работа агрегатной функции topK для generic случая.
- Добавлена отсутствующая проверка на совпадение размеров массивов для n-арных вариантов агрегатных функций с комбинатором -Array.
- Исправлена работа --pager для clickhouse-client (автор: ks1322).
- Исправлена точность работы функции exp10.
- Исправлено поведение функции visitParamExtract согласно документации.
- Исправлено падение при объявлении некорректных типов данных.
- Исправлена работа DISTINCT при условии, что все столбцы константные.
- Исправлено форматирование запроса в случае наличия функции tupleElement со сложным константным выражением в качестве номера элемента.
- Исправлена работа Dictionary таблиц для словарей типа range_hashed.
- Исправлена ошибка, приводящая к появлению лишних строк при FULL и RIGHT JOIN (Amos Bird).
- Исправлено падение сервера в случае создания и удаления временных файлов в config.d директориях в момент перечитывания конфигурации.
- Исправлена работа запроса SYSTEM DROP DNS CACHE: ранее сброс DNS кэша не приводил к повторному резолвингу имён хостов кластера.
- Исправлено поведение MATERIALIZED VIEW после DETACH TABLE таблицы, на которую он смотрит (Marek Vavruša).

Улучшения сборки:

- Для сборки используется pbuilder. Сборка максимально независима от окружения на сборочной машине.
- Для разных версий систем выкладывается один и тот же пакет, который совместим с широким диапазоном Linux систем.
- Добавлен пакет clickhouse-test, который может быть использован для запуска функциональных тестов.
- Добавлена выкладка в репозиторий архива с исходниками. Этот архив может быть использован для воспроизведения сборки без использования GitHub.
- Добавлена частичная интеграция с Travis CI. В связи с ограничениями на время сборки в Travis, запускается только ограниченный набор тестов на Debug сборке.
- Добавлена поддержка Cap'n'Proto в сборку по-умолчанию.
- Документация переведена с Restructured Text на Markdown.
- Добавлена поддержка systemd (Vladimir Smirnov). В связи с несовместимостью с некоторыми образами, она выключена по-умолчанию и может быть включена вручную.

- Для динамической компиляции запросов, clang и lld встроены внутрь clickhouse. Они также могут быть вызваны с помощью clickhouse clang и clickhouse lld.
- Удалено использование расширений GNU из кода и включена опция -Wextra. При сборке с помощью clang по-умолчанию используется libc++ вместо libstdc++.
- Выделены библиотеки clickhouse_parsers и clickhouse_common_io для более быстрой сборки утилит.

Обратно несовместимые изменения:

- Формат засечек (marks) для таблиц типа Log, содержащих Nullable столбцы, изменён обратно-несовместимым образом. В случае наличия таких таблиц, вы можете преобразовать их в TinyLog до запуска новой версии сервера. Для этого в соответствующем таблице файле .sql в директории metadata, замените ENGINE = Log на ENGINE = TinyLog. Если в таблице нет Nullable столбцов или тип таблицы не Log, то ничего делать не нужно.
- Удалена настройка experimental_allow_extended_storage_definition_syntax. Соответствующая функциональность включена по-умолчанию.
- Функция runningIncome переименована в runningDifferenceStartingWithFirstValue во избежание путаницы.
- Удалена возможность написания FROM ARRAY JOIN arr без указания таблицы после FROM (Amos Bird).
- Удалён формат BlockTabSeparated, использовавшийся лишь для демонстрационных целей.
- Изменён формат состояния агрегатных функций varSamp, varPop, stddevSamp, stddevPop, covarSamp, covarPop, corr. Если вы использовали эти состояния для хранения в таблицах (тип данных AggregateFunction от этих функций или материализованные представления, хранящие эти состояния), напишите на clickhouse-feedback@yandex-team.com.
- В предыдущих версиях существовала недокументированная возможность: в типе данных AggregateFunction можно было не указывать параметры для агрегатной функции, которая зависит от параметров. Пример: AggregateFunction(quantiles, UInt64) вместо AggregateFunction(quantiles(0.5, 0.9), UInt64). Эта возможность потеряна. Не смотря на то, что возможность не документирована, мы собираемся вернуть её в ближайших релизах.
- Значения типа данных Enum не могут быть переданы в агрегатные функции min/max. Возможность будет возвращена обратно в следующем релизе.

На что обратить внимание при обновлении:

- При обновлении кластера, на время, когда на одних репликах работает новая версия сервера, а на других - старая, репликация будет приостановлена и в логе появятся сообщения вида unknown parameter 'shard'. Репликация продолжится после обновления всех реплик кластера.
- Если на серверах кластера работают разные версии ClickHouse, то возможен неправильный результат распределённых запросов, использующих функции varSamp, varPop, stddevSamp, stddevPop, covarSamp, covarPop, corr. Необходимо обновить все серверы кластера.

Релиз ClickHouse 1.1.54327, 2017-12-21

Релиз содержит исправление к предыдущему релизу 1.1.54318: * Исправлена проблема с возможным race condition при репликации, которая может приводить к потере данных. Проблеме подвержены версии 1.1.54310 и 1.1.54318. Если вы их используете и у вас есть Replicated таблицы, то обновление обязательно. Понять, что эта проблема существует, можно по сообщениям в логе Warning вида Part ... from own log doesn't exist. Даже если таких сообщений нет, проблема всё-равно актуальна.

Релиз ClickHouse 1.1.54318, 2017-11-30

Релиз содержит изменения к предыдущему релизу 1.1.54310 с исправлением следующих багов: * Исправлено некорректное удаление строк при слияниях в движке SummingMergeTree * Исправлена утечка памяти в нереплицированных MergeTree-движках * Исправлена деградация производительности при частых вставках в MergeTree-движках * Исправлена проблема, приводящая к остановке выполнения очереди репликации * Исправлено ротирование и архивация логов сервера

Релиз ClickHouse 1.1.54310, 2017-11-01

Новые возможности:

- Произвольный ключ партиционирования для таблиц семейства MergeTree.

- Движок таблиц Kafka.
- Возможность загружать модели CatBoost и применять их к данным, хранящимся в ClickHouse.
- Поддержка часовых поясов с нецелым смещением от UTC.
- Поддержка операций с временными интервалами.
- Диапазон значений типов Date и DateTime расширен до 2105 года.
- Запрос CREATE MATERIALIZED VIEW x TO y (позволяет указать существующую таблицу для хранения данных материализованного представления).
- Запрос ATTACH TABLE без аргументов.
- Логика обработки Nested-столбцов в SummingMergeTree, заканчивающихся на -Map, вынесена в агрегатную функцию sumMap. Такие столбцы теперь можно задавать явно.
- Максимальный размер IP trie-словаря увеличен до 128M записей.
- Функция getSizeOfEnumType.
- Агрегатная функция sumWithOverflow.
- Поддержка входного формата Cap'n Proto.
- Возможность задавать уровень сжатия при использовании алгоритма zstd.

Обратно несовместимые изменения:

- Запрещено создание временных таблиц с движком, отличным от Memory.
- Запрещено явное создание таблиц с движком View и MaterializedView.
- При создании таблицы теперь проверяется, что ключ сэмплирования входит в первичный ключ.

Исправления ошибок:

- Исправлено зависание при синхронной вставке в Distributed таблицу.
- Исправлена неатомарность при добавлении/удалении кусков в реплицированных таблицах.
- Данные, вставляемые в материализованное представление, теперь не подвергаются излишней дедупликации.
- Запрос в Distributed таблицу, для которого локальная реплика отстаёт, а удалённые недоступны, теперь не падает.
- Для создания временных таблиц теперь не требуется прав доступа к БД default.
- Исправлено падение при указании типа Array без аргументов.
- Исправлено зависание при недостатке места на диске в разделе с логами.
- Исправлено переполнение в функции toRelativeWeekNum для первой недели Unix-эпохи.

Улучшения сборки:

- Несколько сторонних библиотек (в частности, Poco) обновлены и переведены на git submodules.

Релиз ClickHouse 1.1.54304, 2017-10-19

Новые возможности:

- Добавлена поддержка TLS в нативном протоколе (включается заданием tcp_ssl_port в config.xml)

Исправления ошибок:

- ALTER для реплицированных таблиц теперь пытается начать выполнение как можно быстрее
- Исправлены падения при чтении данных с настройкой preferred_block_size_bytes=0
- Исправлено падение clickhouse-client при нажатии Page Down
- Корректная интепретация некоторых сложных запросов с GLOBAL IN и UNION ALL
- Операция FREEZE PARTITION теперь работает всегда атомарно
- Исправлено зависание пустых POST-запросов (теперь возвращается код 411)
- Исправлены ошибки при интепретации выражений типа CAST(1 AS Nullable(UInt8))
- Исправлена ошибка при чтении колонок типа Array(Nullable(String)) из MergeTree таблиц
- Исправлено падение при парсинге запросов типа SELECT dummy AS dummy, dummy AS b
- Корректное обновление пользователей при невалидном users.xml
- Корректная обработка случаев, когда executable-словарь возвращает ненулевой код ответа

Релиз ClickHouse 1.1.54292, 2017-09-20

Новые возможности:

- Добавлена функция pointInPolygon для работы с координатами на плоскости.
- Добавлена агрегатная функция sumMap, обеспечивающая суммирование массивов аналогично SummingMergeTree.
- Добавлена функция trunc. Увеличена производительность функций округления round, floor, ceil, roundToExp2. Исправлена логика работы функций округления. Изменена логика работы функции roundToExp2 для дробных и отрицательных чисел.
- Ослаблена зависимость исполняемого файла ClickHouse от версии libc. Один и тот же исполняемый файл ClickHouse может запускаться и работать на широком множестве Linux систем. Замечание: зависимость всё ещё присутствует при использовании скомпилированных запросов (настройка compile = 1, по-умолчанию не используется).
- Уменьшено время динамической компиляции запросов.

Исправления ошибок:

- Исправлена ошибка, которая могла приводить к сообщениям part ... intersects previous part и нарушению консистентности реплик.
- Исправлена ошибка, приводящая к блокировке при завершении работы сервера, если в это время ZooKeeper недоступен.
- Удалено избыточное логгирование при восстановлении реплик.
- Исправлена ошибка в реализации UNION ALL.
- Исправлена ошибка в функции concat, возникающая в случае, если первый столбец блока имеет тип Array.
- Исправлено отображение прогресса в таблице system.merges.

Релиз ClickHouse 1.1.54289, 2017-09-13

Новые возможности:

- Запросы SYSTEM для административных действий с сервером: SYSTEM RELOAD DICTIONARY, SYSTEM RELOAD DICTIONARIES, SYSTEM DROP DNS CACHE, SYSTEM SHUTDOWN, SYSTEM KILL.
- Добавлены функции для работы с массивами: concat, arraySlice, arrayPushBack, arrayPushFront, arrayPopBack, arrayPopFront.
- Добавлены параметры root и identity для конфигурации ZooKeeper. Это позволяет изолировать разных пользователей одного ZooKeeper кластера.
- Добавлены агрегатные функции groupBitAnd, groupBitOr, groupBitXor (для совместимости доступны также под именами BIT_AND, BIT_OR, BIT_XOR).
- Возможность загрузки внешних словарей из MySQL с указанием сокета на файловой системе.
- Возможность загрузки внешних словарей из MySQL через SSL соединение (параметры ssl_cert, ssl_key, ssl_ca).
- Добавлена настройка max_network_bandwidth_for_user для ограничения общего потребления сети для всех запросов одного пользователя.
- Поддержка DROP TABLE для временных таблиц.
- Поддержка чтения значений типа DateTime в формате unix timestamp из форматов CSV и JSONEachRow.
- Включено по-умолчанию отключение отстающих реплик при распределённых запросах (по-умолчанию порог равен 5 минутам).
- Используются FIFO блокировки при ALTER: выполнение ALTER не будет неограниченно блокироваться при непрерывно выполняющихся запросах.
- Возможность задать umask в конфигурационном файле.
- Увеличена производительность запросов с DISTINCT.

Исправления ошибок:

- Более оптимальная процедура удаления старых нод в ZooKeeper. Ранее в случае очень частых вставок, старые ноды могли не успевать удаляться, что приводило, в том числе, к очень долгому завершению сервера.
- Исправлена рандомизация при выборе хостов для соединения с ZooKeeper.

- Исправлено отключение отстающей реплики при распределённых запросах, если реплика является localhost.
- Исправлена ошибка, в связи с которой кусок данных таблицы типа ReplicatedMergeTree мог становиться битым после выполнения ALTER MODIFY элемента Nested структуры.
- Исправлена ошибка приводящая к возможному зависанию SELECT запросов.
- Доработки распределённых DDL запросов.
- Исправлен запрос CREATE TABLE ... AS <materialized view>.
- Исправлен дедлок при запросе ALTER ... CLEAR COLUMN IN PARTITION для Buffer таблиц.
- Исправлено использование неправильного значения по-умолчанию для Enum-ов (0 вместо минимального) при использовании форматов JSONEachRow и TSKV.
- Исправлено появление zombie процессов при работе со словарём с источником executable.
- Исправлен segfault при запросе HEAD.

Улучшения процесса разработки и сборки ClickHouse:

- Возможность сборки с помощью pbuilder.
- Возможность сборки с использованием libc++ вместо libstdc++ под Linux.
- Добавлены инструкции для использования статических анализаторов кода Coverity, clang-tidy, cppcheck.

На что обратить внимание при обновлении:

- Увеличено значение по-умолчанию для настройки MergeTree max_bytes_to_merge_at_max_space_in_pool (максимальный суммарный размер кусков в байтах для мержа) со 100 GiB до 150 GiB. Это может привести к запуску больших мержей после обновления сервера, что может вызвать повышенную нагрузку на дисковую подсистему. Если же на серверах, где это происходит, количество свободного места менее чем в два раза больше суммарного объёма выполняющихся мержей, то в связи с этим перестанут выполняться какие-либо другие мержи, включая мержи мелких кусков. Это приведёт к тому, что INSERT-ы будут отклоняться с сообщением "Merges are processing significantly slower than inserts". Для наблюдения, используйте запрос SELECT * FROM system.merges. Вы также можете смотреть на метрику DiskSpaceReservedForMerge в таблице system.metrics или в Graphite. Для исправления этой ситуации можно ничего не делать, так как она нормализуется сама после завершения больших мержей. Если же вас это не устраивает, вы можете вернуть настройку max_bytes_to_merge_at_max_space_in_pool в старое значение, прописав в config.xml в секции <merge_tree> <max_bytes_to_merge_at_max_space_in_pool>107374182400</max_bytes_to_merge_at_max_space_in_pool> и перезапустить сервер.

Релиз ClickHouse 1.1.54284, 2017-08-29

- Релиз содержит изменения к предыдущему релизу 1.1.54282, которые исправляют утечку записей о кусках в ZooKeeper

Релиз ClickHouse 1.1.54282, 2017-08-23

Релиз содержит исправления к предыдущему релизу 1.1.54276: * Исправлена ошибка DB::Exception: Assertion violation: !_path.empty() при вставке в Distributed таблицу. * Исправлен парсинг при вставке в формате RowBinary, если входные данные начинаются с ';'. * Исправлена ошибка при рантайм-компиляции некоторых агрегатных функций (например, groupArray()).

Релиз ClickHouse 1.1.54276, 2017-08-16

Новые возможности:

- Добавлена опциональная секция WITH запроса SELECT. Пример запроса: WITH 1+1 AS a SELECT a, a*a
- Добавлена возможность синхронной вставки в Distributed таблицу: выдается Ok только после того как все данные записались на все шарды. Активируется настройкой insert_distributed_sync=1
- Добавлен тип данных UUID для работы с 16-байтовыми идентификаторами
- Добавлены алиасы типов CHAR, FLOAT и т.д. для совместимости с Tableau
- Добавлены функции toYYYYMM, toYYYYMMDD, toYYYYMMDDhhmmss для перевода времени в числа
- Добавлена возможность использовать IP адреса (совместно с hostname) для идентификации сервера при работе с кластерными DDL запросами

- Добавлена поддержка неконстантных аргументов и отрицательных смещений в функции substring(str, pos, len)
- Добавлен параметр max_size для агрегатной функции groupArray(max_size)(column), и оптимизирована её производительность

Основные изменения:

- Улучшение безопасности: все файлы сервера создаются с правами 0640 (можно поменять, через параметр в конфиге).
- Улучшены сообщения об ошибках в случае синтаксически неверных запросов
- Значительно уменьшен расход оперативной памяти и улучшена производительность слияний больших MergeTree-кусков данных
- Значительно увеличена производительность слияний данных для движка ReplacingMergeTree
- Улучшена производительность асинхронных вставок из Distributed таблицы за счет объединения нескольких исходных вставок. Функциональность включается настройкой distributed_directory_monitor_batch_inserts=1.

Обратно несовместимые изменения:

- Изменился бинарный формат агрегатных состояний функции groupArray(array_column) для массивов

Полный список изменений:

- Добавлена настройка output_format_json_quote_denormals, включающая вывод nan и inf значений в формате JSON
- Более оптимальное выделение потоков при чтении из Distributed таблиц
- Разрешено задавать настройки в режиме readonly, если их значение не изменяется
- Добавлена возможность считывать нецелые гранулы движка MergeTree для выполнения ограничений на размер блока, задаваемый настройкой preferred_block_size_bytes - для уменьшения потребления оперативной памяти и увеличения кэш-локальности при обработке запросов из таблиц со столбцами большого размера
- Эффективное использование индекса, содержащего выражения типа toStartOfHour(x), для условий вида toStartOfHour(x) op constexpr
- Добавлены новые настройки для MergeTree движков (секция merge_tree в config.xml):
- replicated_deduplication_window_seconds позволяет задать интервал дедупликации вставок в Replicated-таблицы в секундах
- cleanup_delay_period - периодичность запуска очистки неактуальных данных
- replicated_can_become_leader - запретить реплике становиться лидером (и назначать мержи)
- Ускорена очистка неактуальных данных из ZooKeeper
- Множественные улучшения и исправления работы кластерных DDL запросов. В частности, добавлена настройка distributed_ddl_task_timeout, ограничивающая время ожидания ответов серверов кластера.
- Улучшено отображение стэктрейсов в логах сервера
- Добавлен метод сжатия none
- Возможность использования нескольких секций dictionaries_config в config.xml
- Возможность подключения к MySQL через сокет на файловой системе
- В таблицу system.parts добавлен столбец с информацией о размере marks в байтах

Исправления багов:

- Исправлена некорректная работа Distributed таблиц, использующих Merge таблицы, при SELECT с условием на поле _table
- Исправлен редкий race condition в ReplicatedMergeTree при проверке кусков данных
- Исправлено возможное зависание процедуры leader election при старте сервера
- Исправлено игнорирование настройки max_replica_delay_for_distributed_queries при использовании локальной реплики в качестве источника данных
- Исправлено некорректное поведение ALTER TABLE CLEAR COLUMN IN PARTITION при попытке очистить несуществующую колонку
- Исправлено исключение в функции multiIf при использовании пустых массивов или строк
- Исправлено чрезмерное выделение памяти при десериализации формата Native

- Исправлено некорректное автообновление Trie словарей
- Исправлено исключение при выполнении запросов с GROUP BY из Merge-таблицы при использовании SAMPLE
- Исправлено падение GROUP BY при использовании настройки distributed_aggregation_memory_efficient=1
- Добавлена возможность указывать database.table в правой части IN и JOIN
- Исправлено использование слишком большого количества потоков при параллельной агрегации
- Исправлена работа функции if с аргументами FixedString
- Исправлена некорректная работа SELECT из Distributed-таблицы для шардов с весом 0
- Исправлено падение запроса CREATE VIEW IF EXISTS
- Исправлено некорректное поведение при input_format_skip_unknown_fields=1 в случае отрицательных чисел
- Исправлен бесконечный цикл в функции dictGetHierarchy() в случае некоторых некорректных данных словаря
- Исправлены ошибки типа Syntax error: unexpected (...) при выполнении распределенных запросов с подзапросами в секции IN или JOIN, в случае использования совместно с Merge таблицами
- Исправлена неправильная интерпретация SELECT запроса из таблиц типа Dictionary
- Исправлена ошибка "Cannot mremap" при использовании множеств в секциях IN, JOIN, содержащих более 2 млрд. элементов
- Исправлен failover для словарей с источником MySQL

Улучшения процесса разработки и сборки ClickHouse:

- Добавлена возмозможность сборки в Arcadia
- Добавлена возможность сборки с помощью gcc 7
- Ускорена параллельная сборка с помощью ccache+distcc

Релиз ClickHouse 1.1.54245, 2017-07-04

Новые возможности:

- Распределённые DDL (например, CREATE TABLE ON CLUSTER)
- Реплицируемый запрос ALTER TABLE CLEAR COLUMN IN PARTITION
- Движок таблиц Dictionary (доступ к данным словаря в виде таблицы)
- Движок баз данных Dictionary (в такой базе автоматически доступны Dictionary-таблицы для всех подключённых внешних словарей)
- Возможность проверки необходимости обновления словаря путём отправки запроса в источник
- Qualified имена столбцов
- Квотирование идентификаторов двойными кавычками
- Сессии в HTTP интерфейсе
- Запрос OPTIMIZE для Replicated таблицы теперь можно выполнять не только на лидере

Обратно несовместимые изменения:

- Убрана команда SET GLOBAL

Мелкие изменения:

- Теперь после получения сигнала в лог печатается полный стектрейс
- Ослаблена проверка на количество повреждённых/лишних кусков при старте (было слишком много ложных срабатываний)

Исправления багов:

- Исправлено залипание плохого соединения при вставке в Distributed таблицу
- GLOBAL IN теперь работает при запросе из таблицы Merge, смотрящей в Distributed
- Теперь правильно определяется количество ядер на виртуалках Google Compute Engine
- Исправления в работе executable источника кэшируемых внешних словарей
- Исправлены сравнения строк, содержащих нулевые символы
- Исправлено сравнение полей первичного ключа типа Float32 с константами

- Раньше неправильная оценка размера поля могла приводить к слишком большим аллокациям
- Исправлено падение при запросе Nullable столбца, добавленного в таблицу ALTER-ом
- Исправлено падение при сортировке по Nullable столбцу, если количество строк меньше LIMIT
- Исправлен ORDER BY подзапроса, состоящего только из константных значений
- Раньше Replicated таблица могла остаться в невалидном состоянии после неудавшегося DROP TABLE
- Алиасы для скалярных подзапросов с пустым результатом теперь не теряются
- Теперь запрос, в котором использовалась компиляция, не завершается ошибкой, если .so файл повреждается

Исправлено в релизе 18.12.13 от 10 сентября 2018

CVE-2018-14672

Функция для загрузки CatBoost моделей некорректно обрабатывала пути к файлам, что позволяло читать произвольные локальные файлы на сервере Clickhouse через сообщения об ошибках.

Обнаружено благодаря: Андрею Красичкову из Службы Информационной Безопасности Яндекса

Исправлено в релизе 18.10.3 от 13 августа 2018

CVE-2018-14671

unixODBC позволял указать путь для подключения произвольного shared object в качестве драйвера базы данных, что приводило к возможности выполнить произвольный код на сервере ClickHouse.

Обнаружено благодаря: Андрею Красичкову и Евгению Сидорову из Службы Информационной Безопасности Яндекса

Исправлено в релизе 1.1.54388 от 28 июня 2018

CVE-2018-14668

Табличная функция "remote" допускала произвольные символы в полях "user", "password" и "default_database", что позволяло производить атаки класса Cross Protocol Request Forgery.

Обнаружено благодаря: Андрею Красичкову из Службы Информационной Безопасности Яндекса

Исправлено в релизе 1.1.54390 от 6 июля 2018

CVE-2018-14669

В ClickHouse MySQL клиенте была включена функциональность "LOAD DATA LOCAL INFILE", что позволяло получать доступ на чтение к произвольным файлам на сервере, где запущен ClickHouse.

Обнаружено благодаря: Андрею Красичкову и Евгению Сидорову из Службы Информационной Безопасности Яндекса

Исправлено в релизе 1.1.54131 от 10 января 2017

CVE-2018-14670

Некоррректная конфигурация в deb пакете могла привести к неавторизованному доступу к базе данных.

Обнаружено благодаря: the UK's National Cyber Security Centre (NCSC)

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