Движки таблиц Clickhouse

Мы рассмотрели типы данных, которые используются в Clickhouse. Если посмотреть на команду CREATE TABLE, то следующий важный момент, с которым нужно определиться при создании таблиц — это движки таблиц (table engines). Собственно говоря, именно движки таблиц и связанная с ней организация хранения данных – это уникальная особенность Clickhouse, то, что отличает Clickhouse от традиционных баз данных, таких, как Oracle, SQL Server и Postgres.

Все типы движков таблиц в Clickhouse разделяются на 4 большие категории:

* MergeTree;
* Log;
* интеграционные (для связи с внешними источниками данных);
* специальные.

В следующих разделах эти семейства и входящие в них движки будут рассмотрены подробно.

Семейство MergeTree

*Описание*

Семейство MergeTree — основное семейство движков таблиц в Clickhouse. Именно оно используется (и подразумевается) по умолчанию. И именно это семейство обеспечивает преимущества Clickhouse в скорости работы с большими объемами данных.

Главная идея этого семейства таблиц: при добавлении данных в таблицу данные очень быстро добавляются крупными кусками, а потом они в фоновом режиме по определенному достаточно хитрому алгоритму объединяются (merge) в единое дерево (tree) таблицы.

Главные свойства всех движков семейства MergeTree:

* колоночное хранение данных, когда данные столбцов таблицы хранятся в отдельных файлах. Такая организация хранения дает большие преимущества при сканировании больших объемов данных;
* данные хранятся, будучи упорядоченными по первичному ключу (который в Clickhouse не обязан быть уникальным, в отличие от традиционных баз данных). За счет этого в Clickhouse индекс для первичного ключа создается разряженным (sparse), то есть записи в индексе создаются не для каждой записи (определяется параметром index\_granularity, по умолчанию для каждой 8192 записи). Индекс получается маленьким и поиск по нему происходит очень эффективно;
* для таблиц можно настроить секционирование, то есть разбить таблицы на куски прозрачно для пользователя;
* поддерживают сэмплирование данных (выборку определенного процента случайно выбранных данных).

Основным типом движка для этого семейства считается MergeTree (именно он выбирается в Clickhouse по умолчанию, если не указан тип движка для таблицы). Для отказоустойчивости, которая обычно необходима в производственной среде, используются движки из семейства Replicated, которые позволяют реплицировать данные по узлам кластера. И есть еще множество модификаций основного MergeTree, которые добавляют дополнительную функциональность для специальных ситуаций.

Варианты движков MergeTree рассматриваются в следующих разделах.

*MergeTree*

Как уже говорилось, именно движок MergeTree является основным для Clickhouse и именно он выбирается по умолчанию, если при создании таблицы не указать его явно.

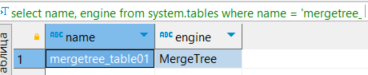
Например:

create table mergetree\_table01 (col1 UInt32, сol2 String) order by col1

select name, engine from system.tables

where name = 'mergetree\_table01'

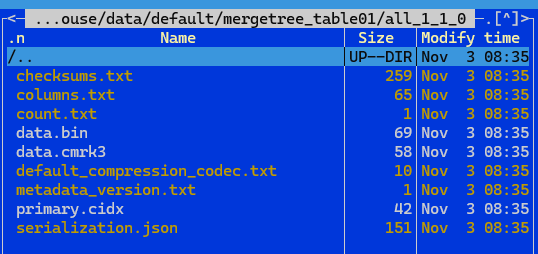
Мы увидим, что для таблицы был выбран именно движок MergeTree.



Добавим в эту таблицу какую-нибудь строку:

insert into mergetree\_table01 values (01, 'Иванов')

Что у нас произошло на диске после создания таблицы и добавления в нее данных?

Добавилась папка для таблицы в каталог /var/lib/clickhouse/data/default. В ней можно увидеть и сами данные, и список столбцов, и контрольные суммы, и много еще чего:

Кратко опишем то, что появилось:

* папка all\_1\_1\_0 — это кусок данных (data part). Он у нас пока единственный. Название расшифровывается как all — название партиции (таблица не секционирована, поэтому она вся лежит в all), 1\_1 — срез блоков в парте, 0 — уровень в дереве слияний (0 — первые парты (протопарты), а после, например, слияния двух партов для получившегося Clickhouse назначит значение 1);
* checksums.txt — контрольные суммы для парта;
* columns.txt — список столбцов с типами данных;
* count.txt — количество записей в парте;
* data.bin — значения столбцов, отсортированных по первичному ключу (или столбцу сортировки). Данные сжаты и хранятся в блоках. В Clickhouse есть специальная утилита, чтобы смотреть сжатые данные. Применение ее может выглядеть примерно так:

cat all\_1\_1\_0/date.bin | clickhouse-compressor -d | od -i

* data.cmrk3 — файл засечек (marks), в котором записано, где в файле bin начинаются значения каждого столбца;
* default\_compression\_codec.txt — указатель на то, какой алгоритм используется для сжатия данных (в нашем случае — LZ4);
* metadata\_version.txt — версия метаданных (в нашем случае метаданные не менялись, поэтому версия 0);
* primary.cidx — данные первичного ключа. Поскольку индексы в Clickhouse по умолчанию разреженный, там лежит информация вида: значение для строки 0, значение для строки 8192 (следующего блока, определяется параметром index\_granularity), значение для строки 8192\*2 и т.п. Для каждого запроса читается целое количество гранул. Для небольших таблиц значение index\_granularity можно уменьшить, чтобы читались блоки меньшего размера, а для больших — увеличить, чтобы индекс был меньше (он обычно помещается в оперативную память и поэтому слишком большое количество записей в индексе приведет к перерасходу оперативной памяти);
* serialization.json — тоже данные о столбцах, но в формате JSON.

Еще добавилась папка для таблицы в /var/lib/clickhouse/metadata/default. В ней появился файл mergetree\_table01.sql с текстом команды CREATE TABLE на создание этой таблицы.

Какие еще параметры можно указать в команде CREATE TABLE при создании таблицы с ядром MergeTree?

Можно указать параметр on cluster. Например, предположим, что у нас создан кластер Cluster01 (подробно про это рассказывалось в разделе про кластеры). Создание таблицы с параметром on cluster может выглядеть так:

create table mergetree\_table\_\_on\_cluster

on CLUSTER 'cluster01'

(col1 UInt32, c2 String)

engine = MergeTree()

order by col1

Таблица будет создана на всех серверах, входящих в кластер Cluster01. Но это будут просто отдельные две таблицы, никак не связанные между собой. Чтобы настроить репликацию или распределение данных, нужно указывать параметры Replicated или Distributed. А в нашем примере команда on cluster — это просто указание, что команду нужно выполнить на всех серверах, входящих в кластер.

*Сортировка и первичный ключ*

Следующее, что можно указать в команде CREATE TABLE при создании таблице на основе движка MergeTree — это первичный ключ и порядок сортировки. Правильный выбор первичных ключей и порядка сортировки имеет критическое значение для производительности при работе с таблицей.

Вначале немного о синтаксисе. Первичный ключ при создании таблицы можно указать так:

CREATE TABLE pk\_table

(

id UInt32,

name String,

created\_date Date

)

ENGINE = MergeTree

PRIMARY KEY (id,name)

А сортировку — так:

CREATE TABLE sort\_table

(

id UInt32,

name String,

created\_date Date

)

ENGINE = MergeTree

ORDER BY (id, name)

На самом деле в обоих случаях мы сделали одно и то же: создали таблицу с первичным ключом на основе столбцов id и name.

Загрузим в обе таблички по миллиону строк:

INSERT INTO pk\_table

SELECT \* FROM generateRandom('id Int32, name String, created\_date Date', 10, 10, 1) LIMIT 1000000

INSERT INTO sort\_table

SELECT \* FROM generateRandom('id Int32, name String, created\_date Date', 10, 10, 1) LIMIT 1000000

Если заглянуть в /var/lib/clickhouse/data/default в папки для каждой таблицы, мы увидим, что файлы данных и первичного ключа получились практически идентичными.

А можно указать и первичный ключ, и порядок сортировки с разным набором столбцов:

CREATE TABLE pk\_sort\_table

(

id UInt32,

name String,

created\_date Date

)

ENGINE = MergeTree

PRIMARY KEY (id, name)

ORDER BY (id, name, created\_date)

В этом случае разница будет. Для поддержки фильтров в запросов будут использоваться только два столбца, которые есть и в PRIMARY KEY, и в ORDER BY, а третий столбец (created\_date) просто будет отсортирован для повышения эффективности для сортировок в запросах. Ускорения в запросах при указании этого столбца в фильтре WHERE не будет.

Первичный ключ в Clickhouse — совсем не то же, что первичный ключ в транзакционных базах типа Oracle/SQL Server/Postgres. Название вроде бы одно, а отличия очень большие:

* первичный ключ в Clickhouse не обязан содержать уникальные значения. Пустые значения (если вы создадите первичный ключ на столбец Nullable) тоже вполне в нем допускаются. В традиционных базах данных первичный ключ — это средство контроля целостности данных, а в Clickhouse — это средство организации хранения данных и вытекающей из нее скорости работы с данными;
* в традиционных базах данных композитный первичный ключ (то есть первичный ключ, который состоит из нескольких столбцов) — это обычно сомнительное решение. В Clickhouse композитные первичные ключи использовать можно и нужно для тех столбцов, по которым производится фильтрация в запросах. При этом принципиален порядок столбцов в первичном ключе (мы в этом убедимся ниже);
* для первичных ключей в Clickhouse создается индекс, как и в традиционных базах данных. Но индекс необычный: он разреженный (sparse), и по умолчанию одна запись в индексе создается для блока записей в таблице размером 8192. Такой блок называется гранулой и его размер можно задать при помощи параметра index\_granularity. Соответственно, даже при извлечении одной записи из таблицы Clickhouse придется считывать целый блок в 8 тыс. записей (и еще и разжимать его). Поэтому для точечных запросов (которые должны получить одну-десяток записей) Clickhouse будет использовать намного больше ресурсов, чем традиционная база данных. Зато, например, агрегирование по большим блокам данных в Clickhouse работает очень эффективно;
* первичные ключи в Clickhouse хранятся в оперативной памяти. Вообще говоря, на оперативную память в Clickhouse много потребителей кроме первичных ключей (например, словари или джойны). Поэтому если вы сделали большой первичный ключ для таблицы с большим размером столбцов, то это может привести к перерасходу оперативной памяти. Ну и в принципе заметим, что на оперативной памяти для сервера Clickhouse экономить не нужно, а к ее настройкам на уровне сервера следует относиться очень внимательно.

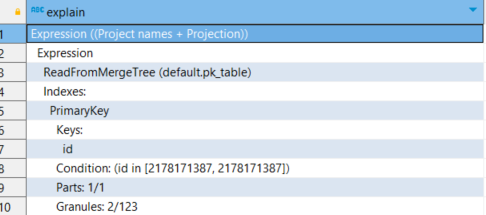
Отдельно скажем про порядок столбцов в первичном ключе и его важность. У нас в предыдущих примерах была сгенерирована таблица pk\_table с миллионом записей. В ней ворох случайных значений, я для своего запроса взял первые попавшиеся.

Выполняем запрос с фильтром по первому столбцу и смотрим его метрики:

explain indexes=1

select \* from pk\_table where id = 2178171387

Метрики получились такие:



Как мы видим, все прилично, для получения данных нам пришлось обратиться всего к двум гранулам (потому что в наборе случайно сгенерированных значений наше значение 2178171387 встретилось два раза), примерно 65 Кб данных пришлось считать с диска (это значение можно получить, если выполнить запрос в clickhouse-client).

Запускаем запрос с фильтром по второму столбцу, который указан в первичном ключе

select \* from pk\_table where name = 'u\_('

Гранул пришлось считывать уже 123, а clickhouse-client показывает, что с диска считано 14 Мб данных. Разница есть.

Если запрос будет с двумя условиями, то вначале фильтр будет по первому столбцу в первичном ключе, а потом — во второму. Поэтому принципиально важно, чтобы столбцы с наибольшим количеством уникальных значений (high cardinality) были первыми в первичном ключе. И конечно, надо учитывать, какие столбцы чаще всего используются в фильтрах запросов. Их тоже нужно указывать первыми в первичном ключе.

Отметим, что при использовании движка MergeTree можно обойтись и без первичных ключей/сортировок. Для этого достаточно при создании таблицы указать параметр ORDER BY tuple(). Данные все равно будут упорядочены, но не по какому-то столбцу/набору столбцов, а по порядку вставки. Чтобы извлекать данные в SELECT в том же порядке, нужно будет на время выполнения запроса поправить параметр конфигурации max\_threads, отключив параллелизм (по умолчанию значение в этом параметре равно количеству доступных процессорных ядер).

*Секционирование в таблицах MergeTree*

Следующий параметр, который можно указать при создании таблиц семейства MergeTree — это параметр PARTITION BY, при помощи которого можно определить партиционирование (секционирование).

Вначале — про терминологию. Редакторы в издательствах требуют писать по-русски, а не калькой с английского, и называть все правильно: секционирование и секции, вместо партиционирования (partitioning) и партиций (partitions). Но в большинстве рабочей документации и в обсуждениях используются именно слова "партиционирование" и "партиции", и мы тоже будем их использовать, чтобы было понятней.

И еще один терминологический момент. В Clickhouse есть **парты (parts, куски)** и **партиции (partitions, секции)**. Это — совершенно разные объекты, хотя и называются похоже.

**Парты** — это куски таблицы, управление которыми обычно производится автоматически. Физически парт — это каталог с файлами, которые представляют часть данных одной таблицы (или все данные, если таблица состоит из одного парта). По определенному алгоритму Clickhouse может сливать парты вместе.

**Партиции**— это тоже куски таблицы, но

1) вы сами выбираете, как делить таблицу на партиции (например, по месяцам или по диапазону числовых данных);

2) партиции не сливаются вместе (каждая партиция может состоять из своих партов)

3) партиции используются для оптимизации запросов и упрощения добавления/удаления данных из таблиц Clickhouse.

Партиции в Clikhouse есть смысл использовать в следующих ситуациях:

1) когда у вас много запросов, которые четко попадают в границы одной партиции. Например, партиции созданы по годам, и вам в рамках запроса нужны данные только за какой-то конкретный год (например, за 2025). В этом случае Clickhouse при выполнении запроса обратится только к партиции за 2025 год, а к партициям за 2023 и 2024 год обращений не будет. Такая технология называется partition pruning (исключение партиций). Выигрыш в производительности получается достаточно большой.

А вот если в фильтре запроса вы не попали в ключ секционирования (не было фильтра по году), то Clickhouse придется обращаться к нескольким партициям сразу и объединять полученные результаты. Это приведет, наоборот, к снижению производительности.

2) для удобства добавления или удаления больших блоков данных. Вы можете присоединить командой ATTACH PARTITION FROM данные за следующий год к вашей таблице, и это будет очень быстро. А если какие-то архивные данные нужно удалить, можно воспользоваться командами DETACH PARTITION/DROP PARTITION.

Как правило, нет смысла настраивать секционирование на небольших объемах данных. Обычно минимальный размер таблицы, с которого можно начинать думать о партициях - в пределах 10 Гбайт.

Партиции определяются при помощи выражения PARTITION BY в команде CREATE TABLE.

Секционировать можно:

1) по диапазону дат (на практике — самый частый способ). Например:

--по годам

PARTITION BY toYear(timestamp)

--по месяцам

PARTITION BY toYYYYMM(timestamp)

--по неделям

PARTITION BY toMonday(timestamp)

--по дням

PARTITION BY toDate(timestamp)

--по часам

PARTITION BY toStartOfHour(timestamp)

2) по диапазонам целочисленных значений:

--в данном случае для каждого миллиона id будет создана своя партиция

PARTITION BY int\_col(id, 1000000)

3) по автоматически рассчитанным диапазонам данных (в Postgres это называется секционированием по хэшам):

--будут созданы партиции по остатку от деления id на 100 (то есть будет создано ровно 100 партиций)

--PARTITION BY id % 100

Покажем работу с партициями на примерах.

Вначале создадим обычную, несекционированную таблицу — как эталон для сравнений:

CREATE TABLE unpart\_table

(t Date, label UInt8, value UInt32)

ENGINE = MergeTree ORDER BY (label, t)

Заполним ее данными — случайными числами, распределенными по датам за последние 10 дней:

INSERT INTO unpart\_table

SELECT today() - rand32() % 10, rand32() % 10000, rand32()

FROM numbers(100000000);

Дальше создаем уже секционированную таблицу с таким же набором столбцов:

CREATE TABLE part\_table

(t Date, label UInt8, value UInt32)

ENGINE = MergeTree PARTITION BY toDate(t) ORDER BY label

В этом примере я явно указал секционирование по дням (PARTITION BY toDate(t)), но если бы я написал PARTITION BY (t), то результат был бы тем же самым, потому что секционирование по дням производится по умолчанию.

Копируем наши сто миллионов записей из несекционированной таблицы в секционированную:

INSERT INTO part\_table SELECT \* FROM unpart\_table

Дальше можно посмотреть, что происходит на физическом уровне с нашими данными:

SELECT partition, formatReadableSize(bytes),

FROM system.parts

WHERE table = 'part\_table'

Мы увидим, что для каждой секции Clickhouse создал большое количество parts (кусков), которое постепенно будет уменьшаться по мере того, как Clickhouse эти куски будет сливать вместе. У меня с нескольких тысяч количество партов дошло до 66 — это количество Clickhouse посчитал отптимальным.

А вот если мы посмотрим на партиции, то это число у нас будет точно равно 10 — по количеству дней в столбце t:

SELECT partition, formatReadableSize(bytes),

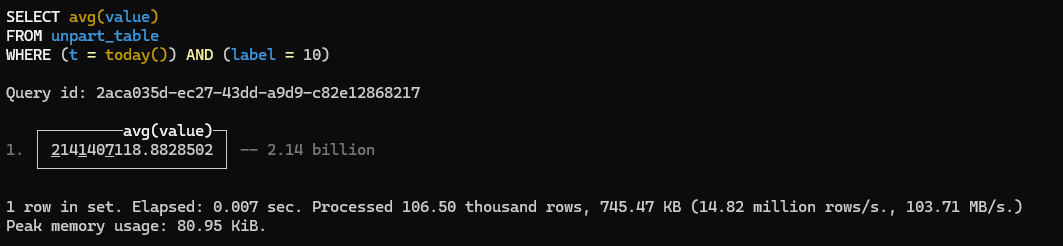
FROM system.parts

WHERE table = 'part\_table'

Секционированную таблицу мы создали, следующая задача — определить, как секционирование повлияло на скорость выполнения запросов. Запросы для целей сравнения производительности лучше выполнять в консольном clickhouse-client, потому что консольный клиент выводит из заголовков http дополнительную важную информацию о затраченных ресурсах (чего DBeaver, например, делать не умеет).

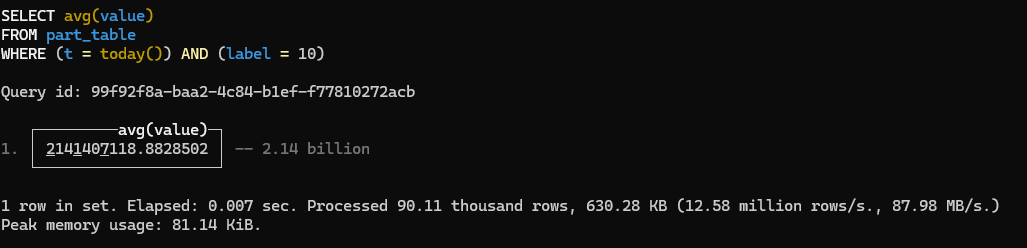
Пусть запрос наш будет таким:

SELECT avg(value) FROM unpart\_table WHERE t = today() AND label = 10

Для несекционированной таблицы результат такой:

А вот для секционированной таблицы результат будет немного отличаться:

SELECT avg(value) FROM part\_table WHERE t = today() AND label = 10



Мы видим, что для секционированной таблицы потребовалось перебрать меньшее количество строк, потому что мы в фильтре запроса использовали ключ секционирования и за счет этого смогли исключить ненужные партиции. Перед запросом можно написать EXPLAIN ESTIMATE, чтобы убедиться, что количество просматриваемых партов для секционированной таблицы меньше.

Но существенного выигрыша в производительности не получилось.

А если у нас не будет фильтра по партициям, то мы еще и проиграем в производительности:

SELECT t, avg(value) FROM unpart\_table WHERE label = 10 GROUP BY t

SELECT t, avg(value) FROM part\_table WHERE label = 10 GROUP BY t

На моей системе время выполнения запроса на секционированной таблице выросло в два раза (зато оперативной памяти потребовалось в два раза меньше, потому что агрегирование вначале производится по каждой секции, а потом создается итоговый результат).

Основной выигрыш при применении партиций достигается при добавлении/удалении данных (когда добавляется или удаляется партиция целиком). Например, если мы удалим данные за день в несекционированной таблице:

ALTER TABLE unpart\_table DELETE WHERE t = today()

то Clickhouse пометит данные как удаленные, а потом в фоновом режиме будет физически удалять записи.

А при использовании команды вида

ALTER TABLE part\_table DROP PARTITION '2024-12-22'

и сама операция пройдет быстрее, и не будет фоновой загрузки Clickhouse (посмотреть точные названия партиций можно через таблицу system.parts).

Еще бывает очень удобно вначале подготовить данные в промежуточной таблице и потом добавить их как секцию к основной таблице. Эта операция тоже происходит очень быстро. А потом промежуточную таблицу можно удалить — все данные останутся в основной таблице. Только проверьте, чтобы у основной и промежуточной таблицы были одинаковы:

1. список столбцов и типы данных
2. ключ секционирования
3. первичный ключ и порядок сортировки

Код для добавления партиции из промежуточной таблицы может выглядеть, например, так:

--создаем промежуточную таблицу

CREATE TABLE staging\_table

(t Date, label UInt8, value UInt32 )

ENGINE = MergeTree ORDER BY (label)

--заполняем ее данными за определенное число

INSERT INTO staging\_table

SELECT '2024-12-23', rand32() % 10000, rand32()

FROM numbers(10000000);

--добавляем промежуточную таблицу как секцию в основную таблицу

ALTER TABLE part\_table ATTACH PARTITION '2024-12-23' FROM staging\_table

--удаляем за ненадобностью промежуточную таблицу

--(все данные останутся в основной)

drop table staging\_table

*Сэмплинг и параметр SAMPLE BY*

Еще одно выражение, которое можно использовать в команде CREATE TABLE при создании таблицы с движком MergeTree — это параметр SAMPLE BY. Если вы создадите таблицу с параметром SAMPLE BY, это позволит использовать в запросах выражение SAMPLE, то есть использовать сэмплинг: использование для расчетов случайно выбранное подмножество записей.

Например, вам нужно посчитать среднее значение по очень большой таблице. Если брать для расчета среднего значения все записи, то расчет будет производиться долго. Но если данные в таблице распределены более-менее равномерно, то можно посчитать среднее значение на основе, например, каждой тысячной записи. Результат принципиально от этого не изменится, а расчет будет производиться намного быстрее.

Несколько принципиальных моментов при использовании сэмплинга, которые нужно учитывать:

1. Ключ сэмплирования должен быть строго целочисленным. Другие типы данных не подходят;
2. Очень рекомендуется для ключа сэмплирования использовать хэши, а не значения из столбца напрямую. Применение хэшей гарантирует, что значения будут выбираться случайно. Традиционно для сэмплирования используется семейство функций CityHash(), которое очень быстро генерирует целочисленные хэш-значения для строк или чисел;
3. Ключ сэмплирования обязательно должен входить в первичный ключ или выражение ORDER BY. Иначе таблицу просто создать не удастся.
4. Самый большой выигрыш в производительности при сэмплировании получается, если первым столбцом в первичном ключе будет набор значений с большим количеством повторяющихся значений (low cardinality). Наглядный пример, который позволит это осознать — ниже.

Покажем пример использования параметра SAMPLE BY и как он влияет на производительность запросов.

Вначале создадим обычную таблицу с использованием SAMPLE BY с обычным id, который генерируется случайно (повторяющихся значений будет мало):

CREATE TABLE sample\_table\_high\_cardinality

(

id UInt32,

name String,

created\_date Date

)

ENGINE = MergeTree

PRIMARY KEY (id, cityHash64(id) ) SAMPLE BY cityHash64(id)

А после этого создадим такую же таблицу, но со специфическим первичным ключом, в котором понизим уникальность (если что, то в первой таблице ключ создастся при заполнении примерно по 98 миллионам значений, а во второй — по пяти):

CREATE TABLE sample\_table\_low\_cardinality

(

id UInt32,

name String,

created\_date Date

)

ENGINE = MergeTree

PRIMARY KEY (round(id / 1000000000), cityHash64(id) ) SAMPLE BY cityHash64(id)

Заполняем таблицы похожими данными:

INSERT INTO sample\_table\_high\_cardinality

SELECT \* FROM generateRandom('id Int32, name String, created\_date Date', 10, 10, 1) LIMIT 100000000

INSERT INTO sample\_table\_low\_cardinality

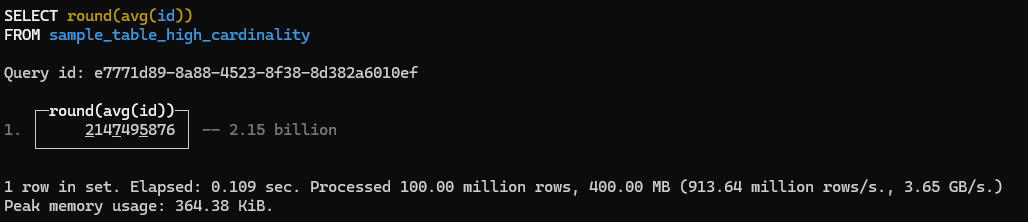
SELECT \* FROM generateRandom('id Int32, name String, created\_date Date', 10, 10, 1) LIMIT 100000000

А дальше смотрим на производительность в clickhouse-client.

Вначале — точный расчет среднего по столбцу id, без сэмплинга:

select round(avg(id)) from sample\_table\_high\_cardinality

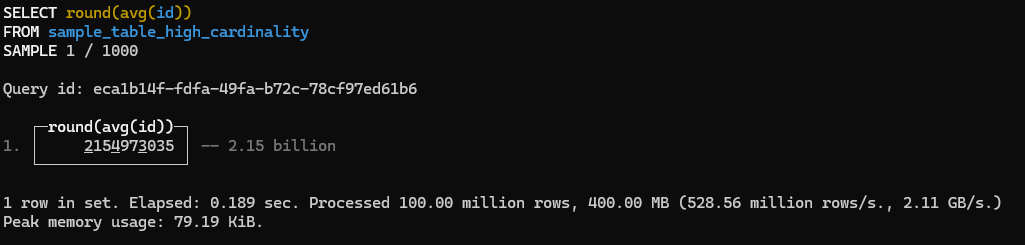
Результат получился таким:



Как мы видим, была произведена обработка 100 миллионов строк, 400 Мбайт данных.

Считаем среднее по каждой тысячной записи по таблице с почти уникальным первичным ключом:

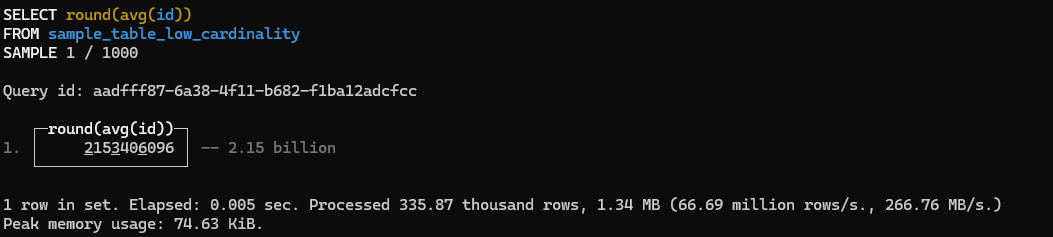
select round(avg(id)) from sample\_table\_high\_cardinality sample 1/1000



Получается, что перебираем те же 400 Мбайт данных (а по времени выполнения все даже чуть хуже, чем без использования сэмплинга).

А теперь посчитаем по таблице со специфическим первичным ключом, в котором специально понижена кардинальность:

select round(avg(id)) from sample\_table\_low\_cardinality sample 1/1000



Как мы видим, в таком варианте получилось перебрать в 300 раз меньший объем данных, да и скорость выполнения запроса выросла раз в 20. А рассчитанное значение среднего получилось очень похожим.

*Время жизни данных и параметр TTL*

Следующий параметр, который можно использовать при создании таблиц семейства MergeTree — это параметр TTL (Time To Live, время жизни). Как понятно из названия, TTL — это время жизни данных. Этот параметр можно настроить:

1) **на уровне столбцов.** Для настроенных таким образом столбцов значения, которые старше указанного времени, будут заменяться на значения по умолчанию для данного типа данных (или сжиматься по указанному алгоритму)

2) **на уровне таблицы.** В этом случае строки старше указанного времени будут удаляться или переноситься на другой диск или заменяться на одну строку со средним значением. Удаление старых строк происходит обычно не мгновенно, и эта операция требует дополнительного внимания (а, может быть, и дополнительных настроек, о которых будет рассказано дальше).

С настройкой TTL связаны и другие интересные моменты. Например, при восстановлении из резервной копии с параметрами по умолчанию есть риск получить пустые таблицы. То есть данные будут восстановлены, но, поскольку время жизни данных уже прошло, то они сразу будут удалены. Поэтому восстанавливать таблицы с настроенным TTL нужно по особому алгоритму.

Но вначале покажем работу с TTL на простых примерах.

Первый пример — **настройка TTL на уровне столбцов**. Не сказать, чтобы этот функционал использовался в практической работе очень часто, но раз он есть, нужно его показать.

Вначале создадим таблицу, в которой столбец с целочисленными значениями будет приводиться к значениям по умолчанию через 30 секунд (на основе информации о дате/времени в dt\_col):

CREATE TABLE column\_ttl

(

dt\_col DateTime,

int\_col Int TTL dt\_col + toIntervalSecond(30),

string\_col String

)

ENGINE = MergeTree

ORDER BY (dt\_col, string\_col)

Затем занесем в эту таблицу какую-нибудь строку с текущей датой/временем и подождем 30 секунд:

insert into column\_ttl values (now(), 10000, 'test')

Сразу после вставки значение в столбце int\_col будет равно 10000, а через 30 секунд — 0.

Для столбцов с TTL можно также указывать, что по истечении времени данные будут сжиматься с использованием указанного кодека. Для этой цели используется синтаксис с ключевым словом RECOMPRESS, например в нашей таблице можно было бы определить столбец int\_col как:

int\_col Int TTL dt\_col + INTERVAL 1 MONTH RECOMPRESS CODEC(ZSTD(17))

Второй пример — **настройка TTL для автоматического удаления строк из таблицы**. Сразу заметим, что удаление — это операция, которая по умолчанию производится со строками, у которых истек срок жизни. Но можно настроить и другие варианты. Полный список операций, который можно произвести по истечению времени жизни на уровне таблицы, выглядит так:

1. DELETE (по умолчанию) — удалить
2. TO DISK 'имя\_диска' или TO\_VOLUME 'имя\_логического\_тома'  — перенести
3. GROUP BY — заменить строки, у которых прошло время жизни, на одну. По каким-то столбцам (например, по дате) производится группировка, а в другие прописывается агрегированное значение (например, среднее).

Пример с удалением строк может выглядеть так:

--создаем таблицу с TTL

create table table\_ttl

(

dt\_col DateTime,

int\_col Int,

string\_col String

)

ENGINE = MergeTree

ORDER BY (dt\_col, string\_col)

TTL dt\_col + INTERVAL 30 SECOND

--заносим в нее данные

insert into table\_ttl

values

(now(), 10000, 'test'),

(now(), 20000, 'test'),

(now(), 30000, 'test')

--ждем 30 секунд и проверяем

select \* from table\_ttl

Все записи через 30 секунд должны исчезнуть.

Есть вариант, когда удаляются не все записи, а только те, которые соответствуют какому-то условию. Например, чтобы в нашей таблице удалялись по истечении TTL только те записи, для которых в string\_col значится 'test', соответствующая строка в определении таблицы может выглядеть так:

TTL dt\_col + INTERVAL 30 SECOND

DELETE WHERE string\_col = 'test'

Пример с переносом на диск может выглядеть так:

TTL dt\_col + INTERVAL 30 SECOND TO DISK 'hdd\_disk'

SETTINGS storage\_policy = 'storage\_policy01'

Как мы видим, при настройке автоматического переноса на другой диск должны использоваться объекты DISK и STORAGE POLICY. Эти объекты необходимо определить в файле конфигурации config.xml. Подробнее про них будет рассказано в разделе про политики хранения данных.

И последний вариант определения времени жизни на уровне таблицы — это замена по истечении TTL многих строк одной с почитанным агрегированным значением (обычно средним). Поскольку при расчете среднего значения используется выражение GROUP BY, то обычно этот вариант настройки TTL опознается именно как 'GROUP BY'.

Выглядеть это может так:

1) создаем таблицу, в которой по истечении 30 секунд значение столбца int\_col меняется на среднее по этому столбцу:

CREATE TABLE column\_ttl\_group\_by

(

dt\_col DateTime,

int\_col Int,

string\_col String

)

ENGINE = MergeTree

ORDER BY (toYYYYMM(dt\_col), string\_col)

TTL dt\_col + INTERVAL 30 SECOND group by toYYYYMM(dt\_col), string\_col set int\_col = avg(int\_col);

2) заполняем эту таблицу данными

insert into column\_ttl\_group\_by values

(now(), 10000, 'test'),

(now(), 20000, 'test'),

(now(), 30000, 'test')

3) ждем 30 секунд… и видим, что почему-то ничего не происходит. Запрос вида

select \* from column\_ttl\_group\_by

покажет те же данные, что мы заносили, а не замененные на среднее значение. Это происходит потому, что для экономии ресурсов Clickhouse заменяет данные с истекшим сроком на агрегированное значение (в нашем случае — среднее) по расписанию согласно значению параметра merge\_with\_ttl\_timeout. А у этого параметра значение по умолчанию — 14400 секунд, то есть 4 часа.

4) чтобы не ждать 4 часа, можно подтолкнуть merge, то есть замену на среднее значение вручную:

OPTIMIZE TABLE column\_ttl\_group\_by FINAL DEDUPLICATE;

И тогда запрос

select \* from column\_ttl\_group\_by

покажет, что все наши строки заменены на одну со средним значением 20000.

*Дополнительные настройки (раздел SETTINGS)*

Последний параметр, который можно указать при создании таблиц семейства MergeTree — это параметр SETTIINGS. Как понятно из названия, в этом разделе определяются разнообразные настройки для конкретной таблицы MergeTree. Настроек этих много (в версии Clickhouse 24.12 их нашлось 216 штук), и просмотреть полный список можно при помощи команды

select \* from system.merge\_tree\_settings

Настройки на уровне всего сервера определяются в файле конфигурации сервера config.xml, а вот если их нужно переопределить для конкретной таблицы — для этот и существует раздел SETTINGS для команды CREATE TABLE.

Значительная часть параметров относится к поведению таблицы при загрузке данных/мутациях/слиянии данных. Эти параметры будут рассмотрены в соответствующих разделах. Пока мы просто приведем пример использования этого параметра.

Количество кусков (parts), из которых может состоять таблица в Clickhouse, ограничено, и по умолчанию составляет 100 000. Предположим, что для какой-то таблицы нам нужно большее количество кусков. Решение может быть таким:

CREATE TABLE settings\_example

(

dt\_col DateTime,

int\_col Int,

string\_col String

)

ENGINE = MergeTree

ORDER BY (dt\_col, string\_col)

SETTINGS max\_parts\_in\_total = 500000

Те настройки, которые были определены для таблицы, можно просмотреть при помощи запроса вида

select name, engine\_full from system.tables

where name = 'settings\_example'

Репликация таблиц семейства MergeTree и ядро ReplicatedMergeTree

*Термины и основные принципы*

Очень важная возможность Clickhouse — это поддержка репликации.

В Clickhouse предусмотрено очень большое количество возможностей, связанных с распределенными данными, и начинающим специалистам по Clickhouse легко в них запутаться. Шарды, кластеры, реплики, базы с ядром Replicated, таблицы с ядром Replicated, таблицы с ядром Distributed, настройки на уровне таблицы и на уровне сервера, Zookeper/Clickhouse Keeper…И как это все свести в единую картину тому, кто видит все это в первый раз?

Поэтому в этом разделе мы постараемся максимально простыми словами разобраться в ситуации с репликацией, чтобы за деревьями можно было рассмотреть лес.

Вначале дадим определение репликации.

**Репликация в Clickhouse — это синхронизация информации между разными серверами.** Например, можно сделать реплики таблицы table1 и в этом случае на разных серверах будут копии этой таблицы. Или сделать реплики шарда таблицы table1, и на разных серверах будут копии не всей таблицы, а поднабора данных (например, за 2024 год). А за 2023 данные будут только на одном сервере.

В Clickhouse есть еще концепция шардинга. **Шардинг — это распределение поднаборов данных таблицы между разными серверами**. Например, данные за 2023 год можно поместить на один сервер, за 2024 год — на второй, за 2025 — на третий. При этом пользователи будут видеть общую таблицу с данными за все годы, и выполнять к этой таблице запросы точно так же, как будто все данные находятся в едином месте.

**Шардинг и репликация в Clickhouse дополняют друг друга**. Например, можно таблицу шардировать, разложив разные годы на разные серверы, а потом для всех или некоторых шардов настроить репликацию.

**Для репликации в Clickhouse используются таблицы с движком Replicated** (например, ReplicatedMergeTree, ReplicatedSummingMergeTree и т.п.), **а для шардинга используются таблицы с движком Distributed**.

Репликация в Clickhouse используется для решения трех задач:

1. **Отказоустойчивости.** В случае недоступности одной реплики пользователей можно оперативно перенаправить на другую. Заметим, что репликация дополняет резервное копирование, но никоим образом не заменяет его. Если, например, произошло случайное удаление важных данных, то оно реплицируется по всем репликам, и без резервных копий здесь не спастись.
2. **Удобства обслуживания.** Если вам нужно установить обновления операционной системы или перейти на новую версию Clickhouse, очень удобно вначале всех пользователей перевести на одну реплику, сделать все, что нужно, на другой, а потом сделать то же самое с остальными репликами.
3. **Распределение нагрузки.**  Вы вполне можете либо просто распределить пользователей и их запросы между репликами, либо приоритизировать доступ, предоставив доступ на отдельную реплику на мощном сервере привилегированным пользователям.

Основные принципы репликации Clickhouse можно сформулировать так:

* **при репликации Clickhouse информацию можно менять на любой реплике**: она применится ко всем остальным репликам. В некоторых статьях такую репликацию называют multimaster, но это не совсем верно: multimaster подразумевает наличие модуля разрешения конфликтов (на одной реплике фамилию сотруднице поменяли на "Иванова", на другой реплике ей же — на "Петрова", какой вариант получится в итоге?) В Clickhouse модуля разрешения конфликтов нет, но добавлять/изменять/удалять информацию действительно можно на любой реплике. Изменения будут применяться последовательно, и в итоге останется последний вариант.
* **за координацию при синхронизации данных в репликах отвечает внешний программный модуль Zookeeper или Clickhouse Keeper.** Keeper хранит в себе информацию о метаданных, а также о том, какие блоки данных (парты) были применены к какой реплике (или какие предстоит применить). Clickhouse вполне может работать в режиме одиночного сервера, но репликации без Keeper не получится.
* **по умолчанию репликация происходит в асинхронном режиме**, то есть изменения вначале происходят на одной реплике, а потом передаются на остальные. Поэтому данные между репликами вполне могут в течение какого-то времени отличаться. Если в вашей системе это принципиально недопустимо, то можно использовать параметр insert\_quorum. Для insert\_quorum указывается количество реплик, для которых должно произойти изменение, прежде чем вставка завершится успешно. Если указать в insert\_quorum все реплики, то можно считать, что вставка производится синхронно на все реплики. Но такой подход может привести к существенному замедлению вставки данных.
* **реплицируются команды:**
  + **INSERT/UPDATE/DELETE/TRUNCATE**
  + **OPTIMIZE**
  + **ALTER TABLE (кроме FREEZE, MOVE TO DISK, FETCH)**
* **не реплицируются команды:**
  + **CREATE TABLE**
  + **DROP TABLE**
  + **RENAME TABLE**
  + **ATTACH/DETACH TABLE**

*Простой пример создания реплицируемых таблиц*

Как уже говорилось, для настройки репликации в Clickhouse используются таблицы с движком ReplicatedMergeTree. Этот движок позволяет использовать все возможности MergeTree плюс к нему добавляются возможности репликации.

Предположим, что у нас работает набор из трех серверов Clickhouse с настроенным Clickhouse Keeper (пример настройки был приведен в предыдущем разделе).

Самый простой вариант создания реплицируемой таблицы может выглядеть так:

--на первом сервере выполняем команду

CREATE TABLE test\_replication

(

id Int64,

string\_col String

) ENGINE = ReplicatedMergeTree('/clickhouse/tables/replicated/test\_replication', 'replica\_1')

ORDER BY id

--на втором сервере выполняем команду

CREATE TABLE test\_replication

(

id Int64,

string\_col String

) ENGINE = ReplicatedMergeTree('/clickhouse/tables/replicated/test\_replication', 'replica\_2')

ORDER BY id

--на первом сервере добавляем данные

insert into test\_replication

values

(1, 'Данные добавлены на первом сервере')

--на втором сервере добавляем данные

insert into test\_replication

values

(2, 'Данные добавлены на втором сервере')

--на любом сервере выполняем команду

select \* from test\_replication

--и видим обе строки данных в каждой реплике

Добавленные данные можно изменить командой UPDATE или удалить командой DELETE (или почистить всю таблицу командой TRUNCATE) — и изменения в данных тут же отобразятся на всех репликах.

Прокомментируем главную для репликации часть нашей команды CREATE TABLE:

ENGINE = ReplicatedMergeTree('/clickhouse/tables/replicated/test\_replication', 'replica\_1')

ReplicatedMergeTree — это, конечно, название движка, предназначенного для работы с реплицируемыми таблицами.

'/clickhouse/tables/replicated/test\_replication' — это "путь для реплики" (replica path) в базе данных Keeper. Фактически это уникальный идентификатор реплицируемой таблицы в Keeper, который должен быть одинаков для всех реплик. Жестких требований к этому пути нет, кроме того, что он должен начинаться со слэша. Например, путь вида '/test' тоже вполне себе подойдет. Но очень рекомендуется стандартизация на вашем предприятии, чтобы путь для всех реплицируемых таблиц был единообразным.

'replica\_1' — это название реплики. Вот оно должно быть разным между репликами таблицы. Опять-таки жестких требований к названиям реплик нет, но рекомендуется называть реплики так, чтобы все было максимально понятным.

*Применение ядра базы данных Replicated*

В предыдущем примере мы прописывали путь реплики (ее идентификатор) вручную. Если таблиц у вас много, и вы хотите реплицировать все или большинство таблиц в базе данных, можно упростить себе работу, использовав базу данных с движком Replicated.

Выглядеть это может так:

1) создаем на первом сервере базу данных с движком Replicated:

CREATE database db\_replicated ENGINE=Replicated('/db\_replicated','shard01','replica01')

При этом /db\_replicated — это путь, который будет автоматически дописываться к пути реплики (replica path) всех создаваемых таблиц. shard01 — имя шарда (в данном случае может быть любым, по шардам группируются создаваемые таблицы). replica01 — название реплики, которое автоматически будет использоваться для всех создаваемых в этой базе данных таблиц с движком ReplicatedMergeTree.

2) Создаем на втором сервере такую же базу данных, но с другим названием реплики:

CREATE database db\_replicated ENGINE=Replicated('/db\_replicated','shard01','replica02')

Можно проверить, что база данных создана на обоих серверах, при помощи запроса вида

select \* from system.databases

3) далее создаем в этой базе данных таблицу с движком ReplicatedMergeTree:

CREATE TABLE db\_replicated.t1

(

key Int64,

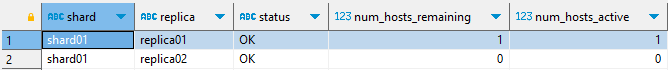
someCol String

)

ENGINE = ReplicatedMergeTree

ORDER BY key;

Результат выполнения этой команды будет примерно таким:



Мы видим, что таблица создана одновременно на двух серверах, и нам не потребовалось указывать ни путь реплики, ни ее название. Теперь точно так же, как и в предыдущем примере мы можем в эту таблицу строки на любом сервере и точно так же они будут видны на любом сервере, где есть реплики этой таблицы.

*Применение кластеров и выражение ON CLUSTER при создании таблиц ReplicatedMergeTree*

В самом простом нашем примере при создании реплицируемой таблицы команды CREATE TABLE пришлось выполнять на нескольких серверах. В следующем примере, при использовании базы данных с ядром Replicated, было достаточно выполнить одну команду для создании реплицируемой таблицы на всех серверах, на которых находилась такая база данных. Способ избежать дублирования команд DDL за использования баз данных с ядром Replicated является современным и наиболее рекомендованным.

Но есть и еще один способ добиться того же самого: использовать определение кластера в файле конфигурации Clickhouse, а затем выполнять команду DDL с модификатором ON CLUSTER. Этот способ является традиционным и менее рекомендованным, чем применение баз данных с ядром Replicated, но он есть, он вполне используется и поэтому мы его рассмотрим.

Применение кластера при создании базы данных нами уже рассматривалось в соответствующем разделе. Теперь применим кластер для создания реплицируемых таблиц без необходимости дублирования DDL (то есть выполнения команд на всех серверах, на которых нужна реплика).

Первое, что необходимо сделать — прописать на каждом сервере кластер в файлы конфигурации Clickhouse (по умолчанию — /etc/clickhouse/config.xml). Кластер записывается в раздел remote\_servers, после определения кластера default, примерно так

--оставляем код для кластера default как он есть

</default>

<cluster01>

<shard>

<replica>

<host>ch01.domain.com</host>

<port>9000</port>

<user>default</user>

<password>password</password>

</replica> </shard>

<shard>

<replica>

<host>ch02.domain.com</host>

<port>9000</port>

<user>default</user>

<password>password</password>

</replica> </shard>

</cluster01>

</remote\_servers>

Но этого мало. В команде на создание реплицируемой таблицы должны быть указаны шард, в которой она создается, и название реплики. Этого в определении кластера нет, но эти параметры можно (и нужно) записать в определение кластера в config.xml. Делается это через макрос (то есть специальные подстановочные значения). Нужно найти в config.xml раздел

<!--

<macros>

<shard>01</shard>

<replica>example01-01-1</replica>

</macros>

-->

снять с него комментарии и прописать значения шарда и реплики, чтобы они автоматически подставлялись в команды DDL на серверах. Шард должен быть одинаков на всех серверах, на которых будут создаваться наши таблицы, а вот реплики — разные. Например, на первом сервере можно написать

<macros>

<shard>01</shard>

<replica>replica01</replica>

</macros>

на втором:

<macros>

<shard>01</shard>

<replica>replica02</replica>

</macros>

Перезапускать сервер Clickhouse не обязательно: при сохранении файла конфигурации он автоматически воспримет изменения.

После этого на первом сервере можно запускать команду вида

CREATE TABLE table\_cluster\_demo on cluster cluster01

(

key Int64,

someCol String

)

ENGINE = ReplicatedMergeTree

ORDER BY key;

Реплицируемая таблица автоматически появится на всех серверах, которые были указаны в определении кластера.

*Переделка обычной таблицы в реплицируемую*

Иногда возникает ситуация, когда вам необходимо переделать обычную таблицу приличного размера в реплицируемую. Можно, конечно, просто создать реплицируемую таблицу и скопировать в нее данные, но если таблица очень большая, то это не самый лучший вариант.

Clickhouse позволяет сконвертировать обычную таблицу в реплицируемую. Для этой цели есть два способа.

Первый способ — традиционный, который работает во всех версиях Clickhouse. Он связан с отсоединением партиций от нереплицируемой таблицы и подключением их в реплицируемой. На большой таблице эта операция может быть достаточно трудоемкой. Поэтому пошаговые действия мы здесь указывать не будем, ограничимся только ссылкой на инструкцию:

<https://kb.altinity.com/altinity-kb-setup-and-maintenance/altinity-kb-converting-mergetree-to-replicated/>

Второй способ появился в версии Clickhouse 24.12 и он намного проще (и в настоящее время является рекомендованным). Мы просто отключаем обычную таблицу (DETACH TABLE) и подключаем ее как реплицируемую (ATTACH TABLE AS REPLICATED). Пример кода может выглядеть так:

1) создаем обычную таблицу на движке MergeTree и складываем в нее какие-то данные:

CREATE TABLE table\_to\_convert

(

key Int64,

someCol String

)

ENGINE = MergeTree

ORDER BY key;

insert into table\_to\_convert

values

(1, 'Данные 1')

2) отключаем таблицу:

DETACH TABLE table\_to\_convert;

3) подключаем таблицу как REPLICATED

ATTACH TABLE table\_to\_convert AS REPLICATED;

4) переводим таблицу из режима "только чтение" в режим "чтение и запись", генерируя информацию о реплике в Keeper:

SYSTEM RESTORE REPLICA table\_to\_convert;

Команда SYSTEM RESTORE REPLICA используется обычно в ситуации, когда реплика физически на диске есть, а вот информации в Keeper для нее нет, и по этой причине реплика доступна только на чтение. SYSTEM RESTORE REPLICA исправляет ситуацию, генерируя нужные для репликации данные в Keeper и переводит таблицу в режим "чтение и запись". В нашей ситуации она тоже подойдет.

На первом сервере реплицируемая таблица есть, но к ней нужны реплики на других серверах. Каких их создать? Некоторая проблема заключается в том, что путь реплики (уникальный идентификатор таблицы) у нас генерируется автоматически, а при создании реплики его нужно указать. Добыть путь реплики, а заодно и название реплики на первом сервере можно так:

select zookeeper\_path, replica\_name from system.replicas r where table = 'table\_to\_convert'  
Мы увидим результат примерно такого вида:



Как мы видим, в пути используется UUID таблицы и имя шарда по умолчанию (01). Копируем этот путь и используем его при создании дополнительной реплики на втором сервере:

CREATE TABLE table\_to\_convert

(

key Int64,

someCol String

)

ENGINE = ReplicatedMergeTree('/clickhouse/tables/e498b734-9058-4f36-a046-ae95dc5f6f1c/01','replica02')

ORDER BY key;

Все, еще одна реплика нашей таблицы появилась на втором сервере.

Обратная операция по переделке реплицируемой таблицы в реплицируемую производится точно так же, через DETACH для реплицируемой таблицы и ATTACH AS NOT REPLICATED. А после этого нужно удалить данные репликации для реплик этой таблицы из Keeper:

SYSTEM DROP REPLICA 'replica011' FROM ZKPATH ''/clickhouse/tables/e498b734-9058-4f36-a046-ae95dc5f6f1c/01';

Проекции (projections) в таблицах семейства MergeTree

Еще одна возможность, которая предусмотрена для таблиц MergeTree — это проекции (projections). Проекции — это альтернативный (теневой) вариант имеющейся таблицы на движке MergeTree. Например, такая теневая таблица может быть создана:

* с другим порядком сортировки
* с другим набором столбцов
* с преагрегированными с помощью выражения GROUP BY и агрегатных функций данными.

При помощи проекции мы создаем теневую копию таблицы (или выбранных данных), жертвуя местом на диске, но выигрывая при выполнении запросов.

Покажем, как обычно, работу с проекциями на простом примере.

Создаем таблицу MergeTree с двумя столбцами — числовым и текстовым, при этом для ключа сортировки будет использоваться числовой столбец:

CREATE TABLE projections\_demo (

key\_col UInt64,

string\_col String

)

ENGINE = MergeTree()

ORDER BY key\_col;

Генерируем для этой таблицы 10 миллионов строк:

INSERT INTO projections\_demo

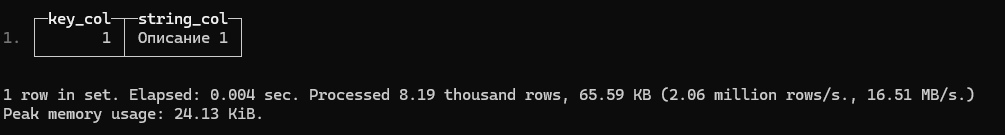
select number as key\_col, concat('Описание ', toString(number)) as string\_col

from numbers(10000000);

А дальше выполняем два запроса (как и все, что связано с оценкой производительности, лучше их выполнять в консольном clickhouse-client).

Первый запрос — с фильтром по ключу:

select \* from projections\_demo where key\_col = 1

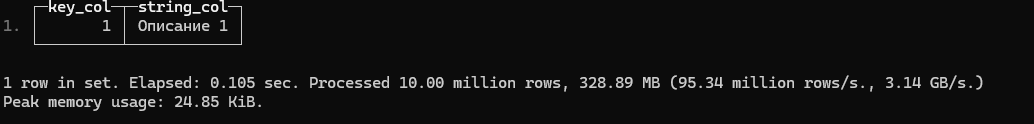
Результат получился таким:

Все вполне эффективно. Проанализирована одна гранула (8192 строки), запрос выполнился за 4 миллисекунды.

А второй запрос — с фильтром по неключевому столбцу:

select \* from projections\_demo where string\_col = 'Описание 1'

Результат:



С этим запросом все хуже. Ключа с индексом для столбца string\_col у нас нет, и поэтому пришлось перебирать все записи в таблице. Запрос работает в 25 раз дольше.

Для ускорения поиска можно, конечно, создать индекс. Но в Clickhouse есть и другое решение, более гибкое и функциональное (но и требующего большего расхода дискового пространства): создание проекции, то есть теневой таблицы с нужным нам ключом сортировки.

Выглядит это так:

--создаем проекцию для таблицы:

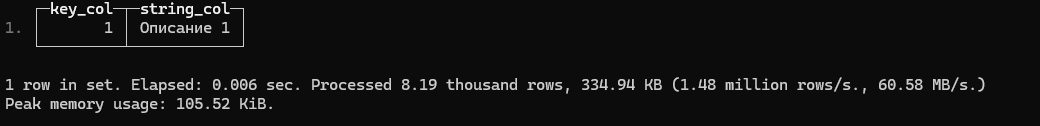
ALTER TABLE projections\_demo ADD projection string\_col\_projection (SELECT key\_col, string\_col ORDER BY string\_col);

--заполняем ее данными:

ALTER TABLE projections\_demo MATERIALIZE projection string\_col\_projection;

--и еще раз запускаем тот же запрос с поиском по string\_col:

select \* from projections\_demo where string\_col = 'Описание 1'

Результат:

Как мы видим, мы пришли к той же скорости, что и при выполнении запроса с фильтром по ключу сортировки. Полного перебора записей уже нет, а если мы посмотрим на план запроса, дописав к запросу EXPLAIN, то увидим, что при выполнении этого запроса использовалась наша проекция string\_col\_projection:



Чем мы заплатили за скорость?

Смотрим размер исходной таблицы projection\_demo:

SELECT table, active,

formatReadableSize(sum(data\_compressed\_bytes)) as size

FROM system.parts

where table = 'projections\_demo' and active = 1

GROUP BY table, active

Размер получился равным 87,68 MiB.

Смотрим размер проекции:

SELECT

table,

name,

formatReadableSize(sum(data\_compressed\_bytes) AS size)

FROM system.projection\_parts

WHERE table = 'projections\_demo' and active = 1

GROUP BY table, name

Размер проекции получился 86,07 MiB. Место, занимаемое таблицей, за счет проекции практически удвоилось. Но это у нас такой пример: мы в проекцию поместили все столбцы нашей таблицы. Если бы на практике нам потребовались не все столбцы, или, например, мы бы сделали проекцию на основе сгруппированных и агрегированных значений, то проекция могла бы получиться намного меньше, чем исходная таблица. Так что использовать проекцию или нет — зависит от конкретной ситуации.

Что еще нужно сказать про проекции?

1. Как мы видели в нашем примере, создание проекции — это просто помещение ее метаданных в системные таблицы. Чтобы реально поместить в проекцию данные, нужно выполнить команду ALTER TABLE MATERIALIZE PROJECTION. Чтобы удалить данные из проекции, нужно использовать команду ALTER TABLE CLEAR PROJECTION, а чтобы полностью удалить проекцию, используется команда ALTER TABLE DROP PROJECTION. Все эти команды реализованы как мутации, поэтому, если вам очень нужно подтолкнуть выполнение этих команд, можно использовать команду вида

optimize table projections\_demo final

А посмотреть, закончились ли мутации, можно при помощи команды вида

select \* from system.mutations where table = 'projections\_demo'

Если есть мутации со значением 0 в столбце is\_done, это — незавершенные мутации.

1. Если вы создали проекцию с избранным набором столбцов из исходной таблицы, запрос должен обращаться только к тем столбцам, которые есть в проекции: иначе проекция использоваться запросом не будет.
2. При выполнении запроса Clickhouse будет использовать проекцию только в том случае, если это выгоднее (по количеству чтений с диска), чем обращение к таблице напрямую. То же самое относится к ситуации, когда для вашей таблицы создано несколько проекций (в этом случае можно вручную повлиять на выбор при помощи параметров preferred\_optimize\_projection\_name и force\_optimize\_projection\_name).
3. Если проекция используется для столбцов с группировкой и агрегацией (или другими выражениями), в запросе должны использоваться точно такие же выражения, как в определении проекции.
4. Команды DELETE (они же в терминологии Clickhouse — lightweight DELETE) в течение долгого времени использоваться с таблицами, в которых есть проекции, не могли. Сейчас по умолчанию то же самое, но в версии 24.7 появился параметр lightweight\_mutation\_projection\_mode, который позволяет изменить это поведение.
5. Начиная с версии 24.8 появилась возможность использовать (в разных режимах) проекции с таблицами на движке ReplicatedReplacingMergeTree. Поведение таких проекций в реплицируемой таблице регулируется при помощи параметра конфигурации deduplicate\_merge\_projection\_mode.

Таблицы с движком ReplacingMergeTree и автоматическое удаление дубликатов

Еще одна разновидность движков семейства MergeTree — это ReplacingMergeTree, то есть "заменяющее дерево слияния". Под заменой в данном случае подразумевается замена дубликатов строк на единую версию, то есть скорее удаление.

Чаще всего ReplacingMergeTree используется в двух ситуациях:

1) когда вам нужно удалить дубликаты, возникающие в процессе загрузки данных в Clickhouse. В реальной жизни может встретиться, например, повторная загрузка части данных из-за ошибок и повторов при операции ETL. Движок ReplacingMergeTree такие ошибки исправляет практически в автоматическом режиме;

2) обновления в режиме, близком к реальному времени, когда вам нужно иметь только последний вариант какой-то записи (последняя цена сделки, последний остаток по товару и т.п.)

При использовании движка ReplacingMergeTree необходимо обязательно учитывать несколько важных моментов:

* удаление дубликатов производится в ходе операций Merge, которые производятся по внутреннему расписанию Clickhouse в асинхронном режиме. Поэтому сложно гарантировать, что в какой-то момент времени у вас действительно будет только одна версия записи. Конечно, можно подтолкнуть слияние при помощи команды OPTIMIZE, но эта команда является достаточно ресурсоемкой и стоит хорошо подумать, а стоит ли ее использовать;
* удаление дубликатов производится в рамках партиции. Поэтому если у вас таблица с набором партиций, то дублирующиеся значения между партициями удаляться не будут;
* если в обычном MergeTree первичный ключ выбирается из соображений производительности, то в ReplacingMergeTree первичный ключ ближе к своему традиционному значению и планируется исходя из соображений уникальности. Но тут возникает некоторый конфликт между производительностью и функциональностью. Поэтому есть смысл обдумать вариант, когда первичный ключ состоит из множества столбцов, и первыми в первичном ключе выбраны менее уникальные столбцы, а последними — с наибольшим количеством уникальных значений;
* команды INSERT при добавлении данных в таблицы ReplacingMergeTree обрабатываются практически в реальном режиме времени. При этом происходит либо добавление записи (если дубля для данной нет), либо автоматическое исключение дублирующихся записей, либо замена на новую версию, в зависимости от параметров таблицы. Иногда для таких команд INSERT используется термин 'UPSERT', хотя, конечно, это не совсем UPSERT в понимании, например, Postgres.

Первый пример применения ReplacingMergeTree — самый простой. Дедуплицироваться будут значения, совпадающие по первичному ключу.

--создаем таблицу

CREATE TABLE simple\_replacing

(

key Int64,

someCol String

)

ENGINE = ReplacingMergeTree

ORDER BY key;

--заполняем ее данными с одинаковыми значениями в первичном ключе:

insert into simple\_replacing

values

(01,'Москва'),

(02,'Санкт-Петербург'),

(01,'Москва'),

(01,'Адыгея')

--смотрим, что получилось

select \* from simple\_replacing

У меня остались строки только для Санкт-Петербурга и Адыгеи, все остальное было практически мгновенно удалено. Но при желании можно подтолкнуть процесс слияния (в ходе которого и выполняется дедупликация):

OPTIMIZE TABLE simple\_replacing FINAL

На большой таблице будьте осторожней с командой OPTIMIZE: она потребляет большое количество ресурсов, в особенности оперативной памяти.

Второй вариант ReplacingMergeTree — это вариант с версиями. Останется только тот вариант строки из набора дубликатов по первичному ключу, у которого последняя версия. При этом версию можно определять как целочисленным столбцом, так и датой-временем.

Пример может выглядеть так:

--столбец версии указывается в определении движка

CREATE TABLE simple\_replacing\_version

(

key Int64,

someCol String,

dateCol DateTime

)

ENGINE = ReplacingMergeTree(dateCol)

ORDER BY key;

--добавляем данные

insert into simple\_replacing\_version

values

(01,'Москва', '2024-01-01'),

(02,'Санкт-Петербург', '2024-01-01'),

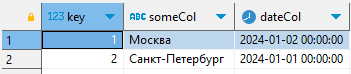
(01,'Москва', '2024-01-02'),

(01,'Адыгея', '2024-01-01')

--проверяем

select \* from simple\_replacing\_version

Результат:



Движок SummingMergeTree

Движок SummingMergeTree позволяет автоматически суммировать данные, совпадающие по ключу. Отчасти он напоминает по смыслу результат выполнения запроса с GROUP BY и SUM().

Основное назначение таблиц SummingMergeTree — это ускорение запросов с группировкой и суммированием, если данных у вас много. В документации предлагается создавать такие таблицы на основе исходных, в которых лежат несуммированные данные, и обращаться к разным таблицам в зависимости от запросов.

Есть два варианта создания таблиц с SummingMergeTree: с указанием столбцов, которые будут суммироваться, и без такого указания (в этом случае будут суммироваться все столбцы с числовыми типами данных).

Самый простой вариант применения движка SummingMergeTree может выглядеть так:

--создаем таблицу

CREATE TABLE summing\_merge\_table

(

product String,

sold\_at DateTime,

qty UInt32

)

ENGINE = SummingMergeTree(qty)

ORDER BY (sold\_at, product);

--заносим в нее данные

insert into summing\_merge\_table values

('Товар 1', '2024-12-01', 800),

('Товар 1', '2024-12-01', 1000),

('Товар 2', '2024-12-01', 210),

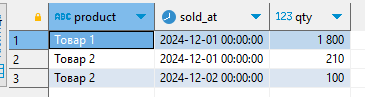
('Товар 2', '2024-12-02', 100);

--проверяем, что получилось

SELECT \* FROM summing\_merge\_table

ORDER BY product, sold\_at

Результат должен быть таким:



Как мы видим, две строчки для первого товара превратились в одну суммированную, потому что значения в них совпали по ключу сортировки (столбцы product и sold\_at).

Выполнение операции по суммированию данных и замене строк точно так же производится во время слияний (мержей), в асинхронном режиме. Поэтому точно гарантировать, что все необходимые строки будут просуммированы на определенный момент времени, сложно.

Если все-таки нужно гарантировать, что все просуммировано, есть два варианта:

1) выполнять к таблице SummingMergeTree запросы с GROUP BY для дополнительной подстраховки. Например, в нашем случае

SELECT

product,

sold\_at,

sum(qty) AS qty

FROM summing\_merge\_table

GROUP BY

product, sold\_at

ORDER BY

product, sold\_at

2) принудительно форсировать мержи при помощи команды вида

OPTIMIZE TABLE summing\_merge\_table deduplicate

(но на большой таблице эта операция может быть очень ресурсоемкой, в особенности по расходу оперативной памяти!)

Еще один принципиальный вопрос — а что произойдет со столбцами, которые не указаны в ключе сортировки и в наборе столбцов для суммирования? В запросе с GROUP BY такие столбцы использовать в выдаче запрещено, а в таблицах с SummingMergeTree вроде как их никто не запрещает использовать.

Ответ на вопрос звучит просто: значения таких столбцов будут приведены к единому значению (первому, которое было в суммирующихся строках). Для начинающего специалиста по Clickhouse такое поведение может оказаться неожиданным, поэтому проиллюстрируем его на примере.

Модифицируем нашу таблицу, добавив в нее дополнительный столбец Country:

CREATE TABLE summing\_merge\_table\_country

(

product String,

sold\_at DateTime,

qty UInt32,

country String

)

ENGINE = SummingMergeTree(qty)

PARTITION BY toYYYYMM(sold\_at)

ORDER BY (sold\_at, product);

Добавим в нее значения (внимание на первые две строки):

insert into summing\_merge\_table\_country

values

('Товар 1', '2024-12-01', 800, 'Россия'),

('Товар 1', '2024-12-01', 1000, 'Беларусь'),

('Товар 2', '2024-12-01', 210, 'Китай'),

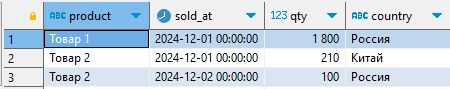
('Товар 2', '2024-12-02', 100, 'Россия');

Проверяем:

select \* from summing\_merge\_table\_country

ORDER BY product, sold\_at

Результат получится таким:

  
Как мы видим, первые две строки просуммировались, а в значения в столбце country приведены к единому значению "Россия".

Движок AggregatingMergeTree

Движок AggregatingMergeTree можно считать более универсальным, но и более сложным в применении, чем движок SummingMergeTree. SummingMergeTree суммирует значения для строк, у которых совпадает ключ сортировки, а AggregatingMergeTree позволяет применять к значениям любые агрегатные функции, которые есть в Clickhouse (а их в Clickhouse десятки, в том числе множество уникальных, которые более нигде не встречаются). Но при этом в определении агрегируемых столбцов придется использовать специфические типы данных SimpleAggregateFunction или AggregateFunction.

Приведем пример с использованием движка AggregatingMergeTree и типа данных SimpleAggregateFunction.

Создаем таблицу:

CREATE TABLE aggregating\_merge\_table

(

product String,

sold\_at DateTime,

sum\_col SimpleAggregateFunction(sum, Nullable(Int64)),

min\_col SimpleAggregateFunction(min, Nullable(Int64)),

max\_col SimpleAggregateFunction(max, Nullable(Int64))

)

ENGINE = AggregatingMergeTree()

ORDER BY (sold\_at, product);

Как мы видим, для столбцов, в которых будут агрегированные данные, используется специальный тип данных SimpleAggregateFunction с указанием на ту самую простую агрегатную функцию плюс ожидаемый тип данных.

Добавляем в таблицу данные:

insert into aggregating\_merge\_table values

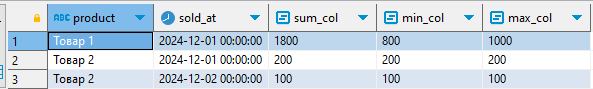
('Товар 1', '2024-12-01', 800, 800, 800),

('Товар 1', '2024-12-01', 1000, 1000, 1000),

('Товар 2', '2024-12-01', 200, 200, 200),

('Товар 2', '2024-12-02', 100, 100, 100);

И смотрим на полученный результат:



Как мы и добивались, данные по Товару 1 за 01 декабря получились агрегированными, с суммой, минимумом и максимумом.

Заносить информацию в столбцы с типом данных AggregationFunction сложнее. В них надо передавать не значения, а состояние. Обычно это делается через INSERT..SELECT и агрегатные функции с комбинатором '-state' (например, uniqState()). Чаще всего для ситуаций, когда важны движок AggregatingMergeTree и типы данных AggregationFunction, используются материализованные представления, и примеры работы с ними будут приведены в соответствующих разделах.

Движок CollapsingMergeTree

Еще один вариант движка MergeTree — CollapsingMergeTree. В чем его смысл?

Предположим, что для каких-то сущностей в таблице нам нужно хранить только последнее значение, которое постоянно меняется (например, количество трафика, потребленного пользователем). В транзакционной базе, такой как Postgres, для изменения данных для данного пользователя достаточно бы было обычного UPDATE. Но Clickhouse — система не транзакционная, изменения данных в ней дорогие.

Поэтому был придуман движок CollapsingMergeTree. Смысл его работы выглядит так:

1) заносим в таблицу на движке CollapsingMergeTree строку для пользователя, например

|  |  |  |
| --- | --- | --- |
| Пользователь | Израсходовано трафика | sign(специальное поле) |
| Иванов | 100 Мб | 1 |

2) приходят новые данные о потреблении трафика Ивановым (150 Мб). Вместо того, чтобы просто изменить существующую строку, мы создаем две новые:

|  |  |  |
| --- | --- | --- |
| Пользователь | Израсходовано трафика | sign(специальное поле) |
| Иванов | 100 Мб | 1 |
| Иванов | 100 Мб | -1 |
| Иванов | 150 Мб | 1 |

3) через какое-то время в асинхронном режиме во время слияния (мержа) Clickhouse удалит две строки с полярным значением sign и у нас останется одна новая строка для Иванова:

|  |  |  |
| --- | --- | --- |
| Пользователь | Израсходовано трафика | sign(специальное поле) |
| Иванов | 150 Мб | 1 |

Движок CollapsingMergeTree, во первых, решает проблему с обновлением данных, которое проводится в фоновом режиме, не мешая вставкам.

Во-вторых, при использовании этого движка учитывается, что изменение данных в таблице идет из-за асинхронного режима с задержкой, и у нас есть возможность очень просто модифицировать синтаксис агрегатных функций, чтобы они, несмотря на то, что мержи еще не прошли, все равно показывали правильное значение. Например, чтобы посчитать количество записей в таблице, вместо привычного Count() можно использовать Sum(sign).

Для специального столбца sign обязательно должен использоваться тип данных Int8. С точки зрения терминологии строки, в которых значение sign положительное, принято называть строками состояния (state row), а те, в которых значение sign отрицательное — строками отмены (cancel row).

Как обычно, покажем работу с CollapsingMergeTree на примере.

Создаем таблицу:

CREATE TABLE collapsing\_merge\_table

(

id UInt64,

used\_mb UInt64,

sign Int8

)

ENGINE = CollapsingMergeTree(sign)

ORDER BY id

Заносим в нее данные:

INSERT INTO collapsing\_merge\_table

values

(1, 100, 1),

(1, 100, -1),

(1, 150, 1)

Смотрим, что получилось:

SELECT \* FROM collapsing\_merge\_table

Должна остаться только последняя строка, со значением в used\_mb, равным 150.

При использовании CollapsingMergeTree есть одна большая проблема: записи в таблицу должны добавляться в строго определенном порядке (вначале с положительным sign, а потом — отрицательным), иначе правильного коллапсирования не произойдет. Например, почистим таблицу из нашего примера:

truncate TABLE collapsing\_merge\_table

И добавим в нее те же данные, но по другому:

INSERT INTO collapsing\_merge\_table

values

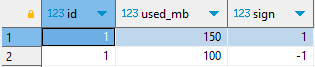
(1, 100, -1)

INSERT INTO collapsing\_merge\_table

values

(1, 100, 1),

(1, 150, 1)

Результат будет таким:

Как мы видим, строка со значением 100 схлопнулась как-то странно: ее версия с sign 1 удалилась, а с -1 — нет. И применение команды вида

optimize table collapsing\_merge\_table final

тоже не поможет.

Если в вашей ситуации добавление записей производится в многопоточном режиме, и порядок занесения записей может быть разным, есть смысл подумать про использование альтернативы — движка VersionedCollapsingMergeTree (о нем — в следующем разделе).

Отметим, что если в таблицу CollapsingMergeTree добавлять строки одной командой INSERT с большим количеством значений, то получается, что вставка идет одним партом. Сливать нечего, и Clickhouse может не "схлопнуть" записи (даже если они идут в правильном порядке). В этой ситуации нужно подумать о замене одной команды INSERT с большим количеством значений на множество мелких INSERT.

Движок VersionedCollapsingMergeTree

Как уже говорилось, у движка CollapsingMergeTree есть недостаток: он требует определенного порядка вставки записей, чтобы всегда вначале вставлялся вариант строки с положительным значением sign, а затем — с отрицательным. Если у вас идет многопоточная вставка данных (а, возможно, и из нескольких источников), то отследить порядок вставки может быть сложно. В этой ситуации лучше использовать движок VersionedCollapsingMergeTree.

В таблице на этом движке не один специальный столбец sign, а два специальных столбца: sign и version (оба должны быть строго с типом данных UInt8). При мержах схлопываются строки с одинаковыми значениями ключа сортировки, версии и полярными значениями sign. При этом порядок вставки строк не важен.

Покажем работу с этим движком на примере.

Создаем таблицу:

CREATE TABLE versioned\_collapsing\_merge\_table

(

id UInt64,

used\_mb UInt64,

sign Int8,

version UInt8

)

ENGINE = VersionedCollapsingMergeTree(sign, version)

ORDER BY id

Заносим в нее данные в неправильном порядке:

INSERT INTO versioned\_collapsing\_merge\_table

values

(1, 100, -1, 1)

INSERT INTO versioned\_collapsing\_merge\_table

values

(1, 100, 1, 1),

(1, 150, 1, 1)

Как мы помним, в такой же ситуации движок CollapsingMergeTree оставлял строку с отрицательным значением sign, и ждать слияния или подталкивать его командой OPTIMIZE было бесполезно. А вот движок VersionedCollapsingMergeTree проблему решит.

Инициируем слияние (merge) командой OPTIMIZE:

optimize table versioned\_collapsing\_merge\_table final

И видим, что у нас осталась только одна запись со значением 150, что и требовалось.

Семейство таблиц Log

Движки таблиц семейства MergeTree — это очень функциональные движки, которые много что умеют, как мы видели в предыдущих разделах. Но иногда вся их функциональность не нужна. В простых ситуациях важнее сэкономить ресурсы.

В этой ситуации могут пригодиться таблицы с движком Log. Принципиально такие таблицы вообще не предназначены для долговременного хранения данных. Это скорее "staging" (промежуточные) таблицы в терминологии ETL. Они предназначены для того, чтобы быстро единым пакетом залить в них относительно небольшой блок данных и потом эти данные поместить в таблицы MergeTree уже для долговременного хранения (возможно, при перемещении эти данные нужно будет проверить/преобразовать/почистить). Идеальный вариант для использования движка Log — когда нужно быстро загрузить в таблицу Clickhouse до миллиона строк и потом целиком их считать с максимальной скоростью и минимальным расходом ресурсов.

Название "Log" — это обманное название! Автор встречался с ситуациями, когда начинающие специалисты по Clickhouse попадались на название и решали для себя что-то вроде "если нам нужно хранить логи, то вот же специализированный тип движков". Нет, хранить логи нужно в таблицах MergeTree, а таблицы Log — для промежуточных данных.

Что не умеют все варианты семейства Log по сравнению с MergeTree?

Вот список:

1. В эти таблицы можно только добавлять данные (UPDATE, DELETE, мутации не поддерживаются). Притом данные добавляются строго в конец файлов таблицы, а на время операции INSERT вся таблица блокируется для всех пользователей и на чтение и на запись. В Clickhouse нет системы блокировок на уровне записей, поэтому приходится поступать именно так и блокировать всю таблицу.
2. Индексов для таких таблиц не предусмотрено, поэтому далеко не все запросы будут эффективно работать для таких таблиц;
3. Атомарные операции записи не поддерживаются, поэтому если при записи в таблицу произойдет сбой, вся таблица будет испорчена.

Конечно же, не поддерживаются ключ сортировки/первичный ключ, слияния, репликация, секционирование, сэмплинг, TTL и прочие продвинутые возможности MergeTree.

Метаданных для таблиц семейства Log тоже хранится минимум, поэтому, например, чтобы понять физический размер таблицы, придется складывать размеры файлов на диске, предварительно выяснив путь к каталогу при помощи запроса вида

SELECT data\_paths FROM system.tables WHERE name = 'имя\_нашей\_таблицы'

В семейство движков Log входят три движка:

1. TinyLog — самый простой и непритязательный вариант, использующий наименьшее количество ресурсов. Все данные читаются в один поток, для каждого столбца создается отдельный файл. В документации рекомендуется использовать этот движок для хранения до одного миллиона строк.
2. Log — разрешает многопоточное чтение (для этой цели используются специальные файлы \_\_marks.mrk с информацией о смещениях в файлах столбцов), каждый столбец хранится в отдельном файле;
3. StripeLog — то же, что и Log, но все столбцы хранятся в общем файле. Считываться информация будет медленнее по сравнению с Log, но мы сэкономим на дескрипторах открытых файлов в файловой системе. Такой движок может пригодиться, если, например, мы загружаем в Clickhouse сто тысяч относительно небольших таблиц для дальнейшей отправки этих данных в MergeTree. Если для каждой таблицы создавать еще набор файлов по числу столбцов, то это может стать проблемой. Здесь нам и пригодится StripeLog.

Простой пример для работы с любым из движков Log может выглядеть так:

--создаем таблицу

CREATE TABLE log\_table

(

key Int64,

someCol String

)

ENGINE = Log

--добавляем в нее данные

INSERT INTO log\_table

VALUES

(77, 'Москва'),

(78, 'Санкт-Петербург')

--вычищаем таблицу

truncate TABLE log\_table

Движок Distributed для диапазонов таблиц и шардинг в Clickhouse

Еще один важный и часто используемый движок для таблиц на сервере Clickhouse — это движок Distributed. Он используется для организации шардинга, когда части данных в таблице распределены по разным серверам.

Сразу скажем про отличия шардинга в Clickhouse от шардинга в транзакционных базах данных, например, Postgres.

В Postgres основной выигрыш от шардинга — это pruning, то есть возможность выполнить запрос с фильтром по одному шарду или небольшому поднабору шардов, что выгоднее, чем производить поиск по большой таблице. Транзакции по шардам при добавлении/изменении записей также идут быстрее, чем транзакции по большой таблице. Чаще всего шардинг в Postgres используется по диапазону значений (BY RANGE).

Clickhouse — аналитическая, а не транзакционная система, и это накладывает свой отпечаток и на шардинг. Основной выигрыш в Clickhouse от шардинга — это распределение нагрузки по нескольким серверам при осуществлении параллельных сканирований. Сам шардинг чаще всего настраивается по хэшам, а не по диапазонам значений.

Для совсем начинающих специалистов по Clickhouse на всякий случай напомним, чем шардинг отличается от репликации.

Репликация — это дублирование данных между разными серверами. Например, у нас есть таблица на миллион записей. Если мы ее реплицируем с использованием двух серверов, то на первом сервере у нас будет миллион записей, и на втором сервере будет миллион записей. Репликация проектируется исходя из соображений отказоустойчивости и удобства обслуживания серверов.

Шардинг — это распределение данных между разными серверами. Если мы шардируем таблицу на миллион записей на два сервера, то это миллион распределится между серверами. Условно говоря, на одном сервере 500 тыс. записей и на другом сервере 500 тыс. записей. Шардинг проектируется исходя из необходимости горизонтального масштабирования, когда мощностей одного сервера уже не хватает для работы с большой таблицей и нужно использовать мощности второго сервера.

Шардинг и репликация взаимно дополняют друг друга и часто используются вместе. Обычно на верхнем уровне используется шардинг для распределения данных по серверам, а внизу данные шардов реплицируются, чтобы обеспечить их отказоустойчивость.

С технической точки зрения репликации обязательно требует координатора в лице Keeper (Clickhouse Keeper или Zookeeper), а шардингу Keeper не нужен: диспетчером при выполнении распределенных запросов выступает сам сервер Clickhouse, на котором находится таблица с ядром Distributed.

А еще шардинг — это распределенные команды DDL (например, CREATE TABLE ON CLUSTER), и при настройке шардинга очень рекомендуется настроить кластер Postgres при помощи файла конфигурации config.xml. О том, как это делается, рассказано в разделе про кластеры Clickhouse.

Покажем работу с распределенными таблицами (то есть с таблицами на движке Distributed) на простом примере с комментариями.

Вначале договоримся, что в файле конфигурации Clickhouse у нас настроен кластер с непритязательным именем cluster01 согласно инструкции в разделе про кластеры. И в этом кластере определено два сервера Clickhouse: ch1.domain.com и ch2.domain.com.

Затем командой CREATE TABLE создаем обычные таблицы на движке MergeTree в кластере, то есть на двух серверах сразу. Таблица на ядре Distributed, которую мы создадим на следующем шаге — это на самом деле вовсе не таблица, а виртуальный интерфейс для проброса команд SELECT и прочих (например, на загрузку данных) в реальные таблицы, которые нужно предварительно создать. Поэтому мы их создаем:

CREATE TABLE shard ON CLUSTER 'cluster01' (

id Int32,

someCol String,

someVal Int32

)

ENGINE = MergeTree

ORDER BY (id)

Поскольку таблица создана с командой ON CLUSTER, она создана на всех узлах кластера (в нашем случае — на серверах ch1.domain.com и ch2.domain.com). Это легко проверить через системную таблицу system.tables.

Затем создаем распределенную таблицу:

CREATE TABLE distributed\_table ON CLUSTER 'cluster01' (

id Int32,

someCol String,

someVal Int32

)

ENGINE = Distributed('cluster01', 'default', 'shard', intHash64(id))

Распределенную таблицу я тоже создал на кластере при помощи команды ON CLUSTER, поэтому она появилась на двух серверах, но, в принципе, делать это было совсем необязательно. Можно было создать распределенную таблицу на одном сервере.

Прокомментируем самую важную часть нашей команды CREATE TABLE — параметры движка Distributed:

* cluster01 — имя кластера, в который были созданы реальные таблицы (шарды);
* default — имя базы для распределенной таблицы;
* shard — имя реальных таблиц (шардов), которые мы создали перед созданием распределенных;
* intHash64(id) — название хеширующий функции для распределения данных по шардам. В принципе, можно было бы включить параметр конфигурации insert\_distributed\_one\_random\_shard = true, и можно было бы не указывать функцию хеширования: данные бы автоматически распределялись по шардам случайным образом.

Еще можно указать политику работы с временными файлами при загрузке данных, но загрузка данных/решардинг распределенных таблиц — это тема, которая вместе с многочисленными настройками и вариантами будет рассмотрена в отдельном разделе.

Пока же загрузим миллион записей в нашу распределенную таблицу самым простым образом:

INSERT INTO distributed\_table

SELECT \*

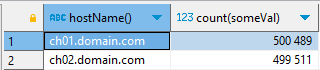
FROM generateRandom('id Int32, someCol String, someVal Int32', NULL, 10)

LIMIT 1000000

И выполним запрос вот такого вида:

SELECT hostName(), count(someVal) FROM distributed\_table GROUP BY hostName()

Запрос покажет нам при помощи специальной функции hostName(), сколько записей было помещено на каждый шард. У меня получился вот такой результат:



Мы создали самую простую распределенную таблицу и в самом простом варианте загрузили в нее данные. В реальной работе вам потребуется оптимизировать загрузку данных в распределенные таблицы, оптимизировать запросы к таким таблицам, производить периодический решардинг данных (когда в ваш кластер добавляется новый узел и нужно часть данных шардированной таблицы перекинуть на него). Но эти моменты будут рассмотрены в отдельных разделах про загрузку данных и оптимизацию запросов.

Другие движки таблиц Clickhouse

Наиболее часто используемые движки таблиц Clickhouse мы рассмотрели в предыдущих разделах. Но в Clickhouse множество и других движков, которые используются для интеграционных либо специальных целей.

Интеграционные движки (для Postgres, Kafka, хранилищ S3 и т.п.) будут рассмотрены в отдельных разделах про интеграцию.

А краткий перечень специальных движков представлен ниже:

* **Dictionary (словарный)** — этот движок можно считать надстройкой над словарями Clickhouse. Он позволяет работать со словарем Clickhouse как с таблицей. Словари Clickhouse — это достаточно важный объект, который рассматривается в отдельном разделе.
* **Memory** — этот движок предназначен для хранения данных в оперативной памяти. Скорость — максимальная, индексов нет, при перезапуске сервера все данные из таблицы удаляются. Таблицы Memory предназначены для использования в качестве временных таблиц небольшого размера с максимальной скоростью доступа.
* **Buffer** — этот движок нужен для того, чтобы накапливать данные в оперативной памяти и с определенной частотой сбрасывать их в обычные таблицы MergeTree. Рекомендуется вместо этого движка использовать альтернативу в виде асинхронных вставок.
* **Merge**— это виртуальный интерфейс, который позволяет для запросов объединять информацию из множества мелких таблиц (обычно TinyLog) и распараллеливать такие запросы.
* **Executable** — это ссылка на скрипт (чаще всего на Python), который должен лежать в папке /var/lib/clickhouse/user-scripts и выводить набор строк в stdin. Этот скрипт будет запускаться каждый раз при выполнении команды SELECT к таблице с движком Executable.
* **File**— этот движок позволяет обращаться как к таблице Clickhouse к файлу поддерживаемого формата на диске. На каждую таблицу в каталоге /var/lib/clickhouse/data/имя\_базы\_данных создается отдельная папка. Поддерживаемых форматов файлов — около 100 (и CSV, и JSON, и XML, и Avro, и Parquet, и SQLInsert — практически что угодно). Этот движок можно использовать для экспорта данных из Clickhouse во внешние файлы разных форматов или, наоборот, чтобы менять данные в Clickhouse, заменяя внешние файлы.
* **SET**— этот движок похож на Memory тем, что все данные хранятся в оперативной памяти. Отличается тем, что к таблице на движке SET нельзя обращаться с запросами SELECT: такая таблица предназначена только для использования в операторе IN. Кроме того, при перезапуске сервера данные сохраняются на диске.
* **JOIN**— этот движок очень похож на SET (все данные также хранятся в оперативной памяти), но такие таблицы предназначены для использования в запросах с объединениями (джойнами).
* **URL**— движок предназначен для получения данных по HTTP с Web-сервера. Данные должны приходить в одном из поддерживаемых форматов (список форматов такой же, как для движка File).
* **View**— это движок, который используется в Clickhouse для создания представлений командой CREATE VIEW.
* **External Data**— очень специфический движок, который позволяет получить данные (из файла, по HTTP и т.п.) разово при использовании клиента командной строки Clickhouse и сразу использовать эти данные в запросе. Сразу после завершения запроса данные исчезнут.
* **GeneratedRandom**— движок, который больше похож на функцию, генерирующий случайные данные для указанной схемы таблицы. Используется, конечно, чтобы сгенерировать тестовые данные и поместить их в таблицу.
* **Null**— движок-заглушка, который отправляет все данные, отправляемые в данную таблицу, в никуда. Используется обычно только для того, чтобы на основе таблиц с Null создавать материализованные представления. Данные будут только в материализованном представлении (а в таблице их не будет).
* **KeeperMap**— если сервер Clickhouse у вас развернут с использованием Clickhouse Keeper или Zookeeper, то при помощи этого движка вы можете использовать Keeper для хранения набора данных вида ключ/значение.
* **FileLog** — специфический движок, который позволяет обращаться к логам операционной системы или программ и однократно считывать из них данные (с отслеживанием добавляемых строк). Используется обычно только для наполнения данными материализованных представлений.

# Изменение данных в Clickhouse

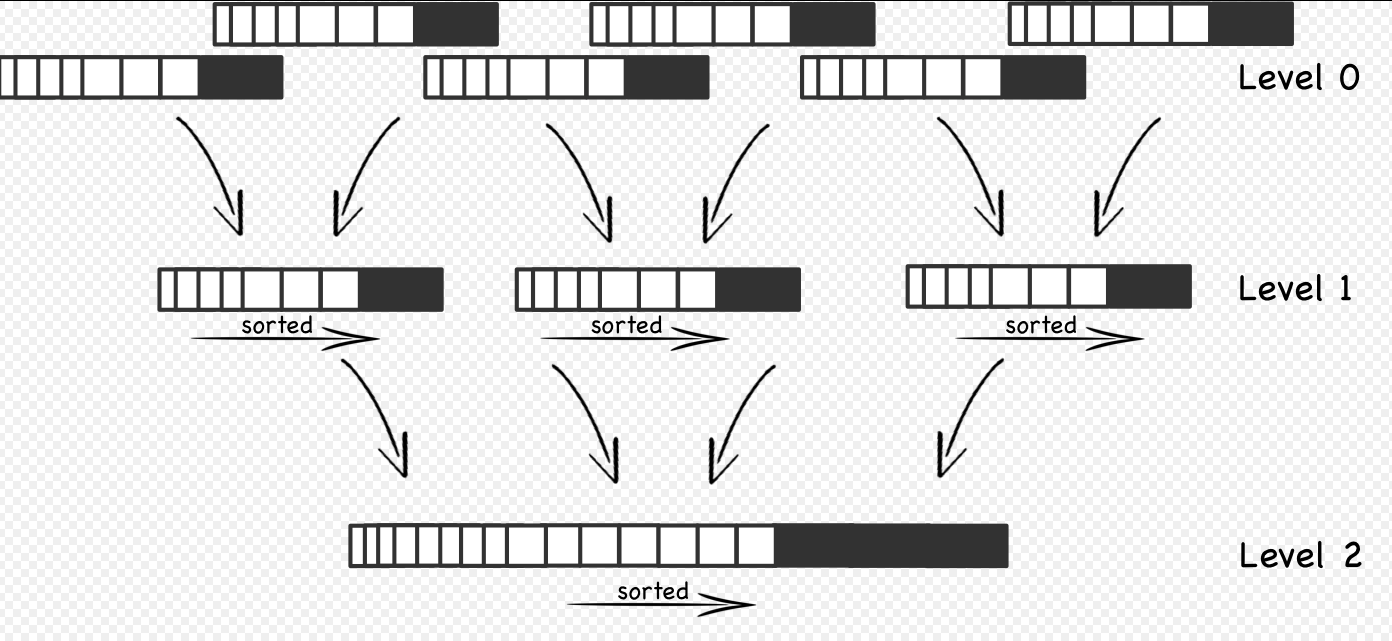
## Особенности изменения данных в Clickhouse

В предыдущих разделах мы рассмотрели основные объекты Clickhouse, главными из которых являются таблицы. Но в таблицы нужно еще вставить данные, а иногда — их изменить или удалить. Загрузка данных из внешних источников будет рассмотрена в следующих разделах.

Clickhouse — аналитическая, а не транзакционная система. Она ориентирована на загрузку больших объемов данных, но и обычные команды INSERT работают в ней вполне эффективно. С удалением данных в Clickhouse все сложнее (хотя есть несколько способов, которые могут повысить эффективность), а с изменением существующих записей совсем большие проблемы. Чтобы понимать, как эффективно работать с Clickhouse, нужно разобраться с особенностями его ядра и с тем, как работает добавление/удаление/изменение данных. Эти моменты рассмотрены в следующих разделах.

## Вставка данных в Clickhouse и механика команды INSERT

Основное ядро Clickhouse, на котором работают таблицы с движками MergeTree, использует концепцию LSM (Log-Structured Merge). Если вы еще не сталкивались с этой концепцией, то для лучшего понимания приведу рисунок:



Основное принцип такой: данные вставляются кусками (партами) на нижнем уровне (на рисунке — уровень 0). Эти куски в фоновом режиме по хитрому алгоритму сливаются (merge) в сортированные куски большего размера (следующие уровни), и в конце концов в идеале у нас остается один сортированный кусок для всей таблицы (в Clickhouse окончательного слияния может и не произойти, если ядро посчитает, что это не нужно).

Каждая операция вставки (например, команда INSERT) создает отдельный парт. Если вы одной командой вставили одну запись — парт будет из одной записи, если вы одной командой вставили 100 записей — парт будет из 1000записей. В документации указано, что максимально эффективной будет вставка от 10,000 до 100,000 записей за один раз. Максимальный размер вставляемого блока данных в строках в Clickhouse определяется параметром max\_insert\_block\_size и по умолчанию равен 1 048 449 строк. Если вы попытаетесь добавить за один раз большее число строк, ничего страшного: Clickhouse сам разобьет вставляемые данные на несколько блоков.

Можно организовать объединение вставляемых строк в блоки (батчи) на стороне клиента. А если клиент посылает вместо батчей на тысячи строк тысячи атомарных команд INSERT и повлиять на него невозможно, то можно использовать объединение вставок в батчи на стороне сервера. Для этой цели используется параметр **async\_insert**.

По умолчанию значение этого параметра равно 0, и это означает, что для любого INSERT будет немедленно формироваться файл на диске. А если при создании таблицы на движке MergeTree установить для async\_insert значение 1, то вставляемые записи будут накапливаться в буфере в оперативной памяти и уже блоком записываться в парт на диск. Сброс данных в парт производится либо по накоплению определенного объема данных (определяется параметром async\_insert\_max\_data\_size, по умолчанию равен 10 Мбайт) или по таймеру (параметр async\_insert\_busy\_timeout\_min\_ms, по умолчанию 50 миллисекунд).

Еще один параметр, над которым имеет смысл подумать при асинхронных вставках — это параметр wait\_for\_async\_insert. Он определяет, когда вернется результат команды INSERT:

* 1 (по умолчанию) — когда данные будут записаны в парт на диск;
* 0 — когда данные будут помещены в буфер в оперативной памяти. Это значение использовать не рекомендуется, потому что если по причине какого-то сбоя данные не дойдут до парта на диске, клиент об этом может не узнать.

Асинхронную вставку можно включить на уровне пользователя (командой ALTER USER), на уровне сеанса и на уровне конкретной команды INSERT:

INSERT INTO table01 SETTINGS async\_insert=1, wait\_for\_async\_insert=1 VALUES (...)

Покажем механику вставки данных в Clickhouse на простом примере.

Создаем простую таблицу INSERT\_DEMO:

create table insert\_demo

(c1 UInt64, c2 String)

ENGINE = MergeTree()

order by c1

Заполняем ее данными на 10 миллионов строк:

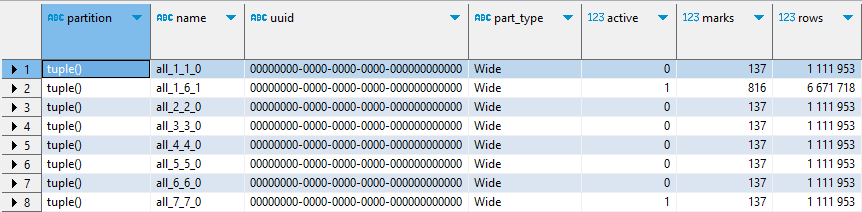
insert into insert\_demo

select number as c1, toString(number) as c2 from numbers (1,10000000)

Запускаем запрос, который покажет, что у нас с партами для этой таблицы:

select \* from system.parts where table = 'insert\_demo'

Результат может быть примерно таким (он может отличаться в зависимости от того, насколько быстро после вставки вы запустили этот запрос):



Мы видим, что Clickhouse начал процесс слияния (merge), по пока не довел до конца. Часть партов уже слиты в другие, большего размера (те, у которых в столбце active стоит значение 0), часть (последняя строка) — еще не слиты. Подтолкнуть слияние вручную можно при помощи команды

optimize table insert\_demo

или

optimize table insert\_demo final

Final означает, что слияние нужно подтолкнуть даже в той ситуации, когда в таблице остался один парт (обычно для дедупликации).

Обратим внимание, что неактивные парты еще какое-то время находятся на диске, хотя обращений к ним уже не будет. Это сделано специально, исходя их механики слияний. При выполнении слияния и создании нового куска fsync (сохранение данных на диск) для нового куска сразу не вызывается, и какое-то время он находится только в оперативной памяти. Если в это время сервер Clickhouse перезапустится, то часть данных может быть потеряна. Неактивные парты лежат на диске в течение какого-то времени как раз для того, чтобы в случае сбоя можно было восстановиться.

Время хранения старых партов на диске определяется при помощи параметра old\_parts\_lifetime и по умолчанию равно 480 секунд (8 минут). Этот параметр можно переопределить на уровне всего сервера или на уровне отдельной таблицы. Например, в нашем случае можно для таблицы поставить одну секунду:

create table insert\_demo

(c1 UInt64, c2 String)

ENGINE = MergeTree()

order by c1

SETTINGS old\_parts\_lifetime = 1

В этом случае неактивные парты будут удаляться практически мгновенно (но в производственной среде это не рекомендуется по соображениям отказоустойчивости).

Отметим также, что при слияниях данные на какое-то время дублируются, и лучше это учитывать при работе с большими таблицами. Рекомендуется всегда иметь на диске с базами Clickhouse запас свободного места (в идеале — 50 процентов для удобства восстановления).