Moscow 2020

Федеральное государственное автономное образовательное
учреждение высшего образования
«Национальный исследовательский университет
«Высшая школа экономики»

Факультет компьютерных наук
Основная образовательная программа
Прикладная математика и информатика

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

(Программный проект)

на тему

Wait-free каталог баз данных в ClickHouse

Выполнил студент группы БПМИ165, 4 курса,
Токмаков Александр Викторович

Руководитель ВКР:
Приглашенный преподаватель Базовой кафедры Яндекс,
Миловидов Алексей Николаевич

Оглавление

Аннотация

ClickHouse - это колоночная реляционная система управления базами данных, используемая многими компаниями для аналитики в реальном времени. Для синхронизации оперирующих метаданными таблиц DDL-запросов, таких как **`CREATE TABLE`**, **`RENAME TABLE`** и **`DROP TABLE`**, в ClickHouse по историческим причинам используется сложный механизм блокировок на уровне таблиц, который имеет ряд недостатков. Целью данной работы является реализация альтернативного подхода к синхронизации некоторых DDL-запросов и хранению метаданных в ClickHouse, который не требует использования табличных блокировок и исключает длительное ожидание при выполнении указанных выше запросов. Для этого в рамках работы был реализован движок баз данных **`Atomic`**, который в дальнейшем может заменить существовавший ранее движок **`Ordinary`**, в настоящее время использующийся как движок по умолчанию.

Abstract

ClickHouse is a column-oriented relational database management system, which is used for real-time analytics by many companies. For synchronization of some DDL-queries, such as **`CREATE TABLE`**, **`DROP TABLE`** and **`RENAME TABLE`**, ClickHouse uses complicated table-level locking mechanism for historical reasons and it has some disadvantages. The main purpose of this work is to implement an alternative approach to DDL queries synchronization and metadata storing. Proposed approach does not require usage of table-level locks and excludes long waiting on execution of mentioned queries. To achieve it new database engine **`Atomic`** was implemented in ClickHouse. This engine may replace pre-existing default database engine **`Ordinary`** in the future.

Ключевые слова

ClickHouse, СУБД, метаданные, DDL-запросы, синхронизация

1. Введение

Для синхронизации таких запросов как **SELECT** и **INSERT**, производящих манипуляции с данными в таблицах, и DDL-запросов, таких как **CREATE**, **DROP** и **RENAME**, манипулирующих каталогом баз данных и модифицирующих метаданные таблиц, в ClickHouse используется механизм рекурсивных честных блокировок таблиц на чтение и запись (**RWLock**) [1]. Запросы, производящие изменения в каталоге баз данных, на время своего выполнения блокируют затрагиваемые таблицы на запись, что исключает параллельное выполнение других запросов к этим таблицам. Запросы, работающие только с данными таблиц, блокируют используемые таблицы на чтение, что исключает параллельное изменение метаданных, переименование или удаление этих таблиц, при этом не мешая параллельной работе других манипулирующих данными запросов.

Со временем этот механизм синхронизации стал достаточно сложным, и в нём проявился ряд недостатков. В частности, чтобы избежать ситуации, когда DDL-запрос не может захватить блокировку на запись из-за непрерывного потока пересекающихся по времени **SELECT** и **INSERT**, удерживающих блокировку таблицы на чтение, блокировки таблиц были сделаны честными. Однако, в реализации честного **RWLock** была обнаружена проблема, которая в редких случаях могла приводить к дедлокам и зависаниям запросов. Для решения этой проблемы в реализацию **RWLock** в ClickHouse был добавлен эвристический алгоритм предотвращения дедлоков, который прерывает выполнение запроса в случае опасности возникновения взаимной блокировки [2]. Основным недостатком этого решения является большое количество ложноположительных срабатываний алгоритма, что часто приводит к прерыванию тех запросов, в которых не возникло бы дедлока.

Кроме того, в существующей реализации с честными блокировками могут возникать ситуации, когда запрос **RENAME** или **DROP** ожидает завершения выполняющихся запросов **SELECT**, в то время как новые запросы **SELECT** фактически не выполняются и ожидают завершения начавшегося ранее DDL-запроса, при этом расходуя квоту на количество одновременно выполняющихся запросов. При наличии долгих запросов **SELECT** это приводит к превышению квоты и прекращению

выполнения новых запросов до тех пор, пока долгие **SELECT** запросы, начавшие выполняться раньше DDL-запроса, не завершатся.

В данной работе был реализован альтернативный подход к синхронизации DDL-запросов, который позволяет избежать описанных выше проблем. Он заключается в использовании UUID в качестве персистентных идентификаторов таблиц, подсчете ссылок на таблицы и фоновом удалении неиспользуемых таблиц.

2. Обзор литературы

Различные СУБД в зависимости от своих особенностей используют различные подходы к организации каталога баз данных и работе с метаданными. Классические транзакционные СУБД, как правило, хранят метаданные в специальных системных таблицах аналогично тому, как они хранят данные в обычных таблицах. Например, в PostgreSQL метаинформация о созданных пользователями базах данных и таблицах хранятся в системных таблицах **pg_databases** и **pg_tables**. Запросы **CREATE**, **RENAME** или **DROP TABLE** фактически вставляют, модифицируют или удаляют соответствующую строку в **pg_tables** [3], что позволяет использовать для этих запросов тот же механизм управления транзакциями, который используется для DML-запросов [4]. Однако, в ClickHouse отсутствует поддержка транзакций, и подобный подход не может быть реализован. Кроме того, основной движок таблиц **MergeTree**, хоть и поддерживает multiversion concurrency control в некотором виде и допускает возможность реализации механизма транзакций в будущем, плохо подходит для точечных запросов и частых модификаций данных, поэтому не может быть использован для хранения метаданных. Системные таблицы **system.databases** и **system.tables** являются в ClickHouse виртуальными и не хранят метаданные самостоятельно.

Более близкая к ClickHouse система Apache Druid, тоже являющаяся колоночной и предназначенной для выполнения аналитических запросов, для хранения метаданных требует использования внешних транзакционных СУБД, таких как PostgreSQL и MySQL, и полностью полагается на них при работе с метаданными. Встроенное в Druid хранилище метаданных, согласно официальной документации, не подходит для использования в production [5]. ClickHouse может использовать ZooKeeper для хранения метаданных и координации работы кластера, но маленькие инсталляции могут эксплуатироваться и без внешних зависимостей. Другая колоночная аналитическая СУБД Vertica, которую можно считать ближайшим ближайшим аналогом ClickHouse, является коммерческой и её исходный код закрыт, что не позволяет получить достаточно информации о деталях реализации.

3. Описание архитектуры

3.1. Общее описание

За управление таблицами в ClickHouse отвечают движки баз данных, реализующие интерфейс **`IDatabase`**. Они загружают таблицы при запуске сервера, позволяют получить **`StoragePtr`** (разделяемый указатель на объект таблицы) по имени, создать, удалить или переименовать таблицу в базе, переместить таблицу между базами данных, если они имеют одинаковый движок, и он поддерживает перемещение. Объектами самих баз данных владеет единственный объект класса **`SharedContext`**, в котором содержится некоторое глобальное разделяемое состояние сервера. Они могут быть получены по имени базы данных через любой объект класса **`Context`**, который может быть глобальным контекстом, контекстом сессии, контекстом запроса или некоторым локальным контекстом. Кроме обычных таблиц, в ClickHouse также есть временные таблицы, которые не принадлежат к какой-либо базе данных и существуют только в рамках некоторого контекста. Это может быть контекст сессии, для таблиц созданных запросом **`CREATE TEMPORARY TABLE`**, или контекст запроса для временных данных (например, для результата выполнения подзапроса). Таким образом, чтобы получить таблицу по её квалифицированному имени, **`Context`** получает объект базы данных по имени базы и обращается к нему чтобы получить **`StoragePtr`** по имени таблицы. Если имя базы данных не указано, то возвращается временная таблица из текущего контекста (если в нём такая существует) или таблица из текущей базы.

Метаданные в ClickHouse хранятся в текстовых файлах в директории **`/clickhouse_root/metadata/`**. В ее поддиректориях базы данных с движком **Ordinary** хранят текстовые файлы, содержащие **`ATTACH`** запросы для соответствующих таблиц. Эти запросы во многом аналогичны **`CREATE`**, но они не создают новые таблицы, а присоединяют уже созданные. При запуске сервера база данных читает файлы метаданных и выполняет эти запросы, тем самым загружая таблицы. Запрос **`ATTACH`** для таблицы **`table_name`** в базе данных **`db_name`** хранится в файле **`/clickhouse_root/metadata/db_name/table_name.sql`**. Аналогично, данные таблицы хранятся в директории **`/clickhouse_root/data/db_name/table_name/`**. Таким образом, имя таблицы встречается сразу в трех местах: в имени файла метаданных, в записанном в файл метаданных **`ATTACH`** запросе и в имени директории с

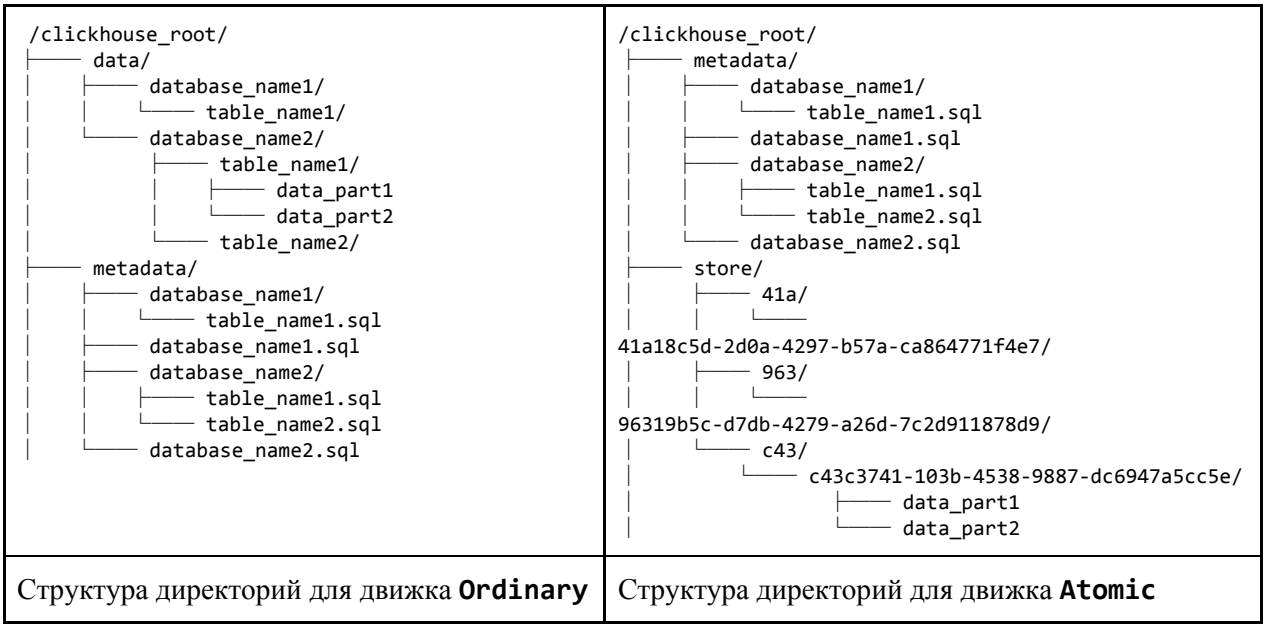
данными, что делает затруднительным атомарное переименование таблицы. Кроме того, это делает невозможным выполнение запроса **RENAME TABLE** при наличии параллельных запросов, использующих расположенные в соответствующей директории данные таблицы. Аналогично, при выполнения запроса **DROP TABLE** необходимо дождаться завершения всех использующих таблицу запросов, прежде чем удалить директорию с данными.

Для защиты от параллельного выполнения запросов **CREATE (ATTACH), DROP (DETACH)** или **RENAME TABLE** с одним именем таблицы, каждый из этих запросов блокирует имя таблицы с помощью объекта **DDLGuard**, который является обычным мьютексом и ассоциирован с именем. Эта блокировка также необходима для корректной работы запросов **CREATE TABLE IF NOT EXISTS** и **DROP TABLE IF EXISTS**, которые должны дождаться завершения других DDL-запросов с тем же именем таблицы и либо создать/удалить таблицу, либо ничего не сделать и успешно завершиться. На работу остальных видов запросов **DDLGuard** не влияет.

Движки таблиц реализуют интерфейс **IStorage**. Некоторые из них требуют дополнительных действий при создании и удалении, таких как запуск/остановка фоновых потоков, добавление/удаление триггеров на вставку данных в другую таблицу (для движка **MeterializedView**) или манипуляций с метаданными в ZooKeeper (для движка **ReplicatedMergeTree**). Для этого существуют методы **IStorage::startup()**, который должен быть вызван сразу после создания или присоединения (**ATTACH**) таблицы, и **IStorage::shutdown()**, который должен быть вызван перед удалением или отсоединением (**DETACH**) таблицы. За полное удаление данных таблицы отвечает метод **IStorage::drop()**, а за перемещение данных при при переименовании таблицы - **IStorage::rename(...)**. Помимо этого они также могут выполнять дополнительные действия, такие как удаление информации о реплике из ZooKeeper или сброс кэша засечек при перемещении данных. Последние два метода должны вызываться при эксклюзивной блокировке табличных **RWLock**, чтобы гарантировать отсутствие параллельно работающих с этой таблицей запросов.

3.2. Изменения

Основное отличие движка баз данных **Atomic** заключается в использовании UUID как постоянных идентификаторов таблиц и отображении имен таблиц на эти идентификаторы. При создании, удалении или переименовании таблицы изменяется только это отображение. Данные таблиц хранятся в директории **`/clickhouse_root/store/xxx/xxxyyyyy-yyyy-yyyy-yyyy-yyyyyyyyyyyy/`**, где **`xxxyyyyy-yyyy-yyyy-yyyy-yyyyyyyyyyyy`** - UUID таблицы, имя директории не зависит от имен таблицы и базы данных и является постоянным (Схема 3.1). Фактические удаление данных происходит в фоновом потоке после завершения всех использующих таблицу запросов, что определяется с помощью счётчика ссылок.



Структура директорий для движка **`Ordinary`**	Структура директорий для движка **`Atomic`**

Схема 3.1. Изменения структуры директорий.

Эти отличия потребовали некоторых изменений в работе с объектами таблиц. Использования отдельных имен базы и таблицы были частично заменены на использование структуры **`StorageID`**, которая помимо этих имен содержит UUID. Для таблиц в базах с движком **`Atomic`** и временных таблиц UUID ненулевой и может использоваться для идентификации таблицы вместо её имени. Переименование таких таблиц происходит без перемещения данных на диске и сбрасывания кэша засечек, метод **`IStorage::renameInMemory(...)`** просто обновляет имя таблицы в памяти, не требуя эксклюзивной блокировки табличных **`RWLock`**.

Из класса **SharedContext** был выделен синглтон **DatabaseCatalog**, который полностью отвечает за работу с каталогом баз данных, фоновое удаление неиспользуемых таблиц и получение **StoragePtr** по **StorageID**. При наличии ненулевого UUID это происходит без обращения к объекту базы данных по имени базы, чтобы иметь возможность перемещать таблицы между базами без длительного удержания блокировок. Временные таблицы всех видов теперь создаются в специальной системной базе данных **_temporary_and_external_tables**, которая также находится в каталоге баз данных, а различные объекты **Context** содержат лишь отображение имен временных таблиц на их идентификаторы и производят разрешение имён относительно текущего контекста в методе **Context::resolveStorageID(...)**. Это упрощает отделение этапа разрешения имен таблиц от остальных этапов анализа запроса и позволяет в дальнейшем добиться атомарной замены имен всех входящих в запрос таблиц на идентификаторы.

4. Реализация

4.1. Файлы метаданных, запросы **CREATE** и **ATTACH**

В запросе **CREATE** или **ATTACH** может быть указан UUID таблицы, который имеет смысл для баз данных с движком **Atomic**. По нему база идентифицирует таблицу и определяет, где расположена директория данных таблицы. В файл метаданных движок **Atomic**, в отличие от **Ordinary**, записывает UUID, но не записывает имя таблицы, т.е. вместо

```
ATTACH TABLE table_name ...
```

в **table_name.sql** записывается

```
ATTACH TABLE _ UUID '8e73547a-52fe-4a97-a911-e94e3653d946' ...
```

Нижнее подчёркивание в этом запросе является заменителем имени таблицы, чтобы запрос оставался синтаксически корректным. При загрузке сервера оно игнорируется, а имя таблицы определяется по имени файла метаданных.

При выполнении запроса **CREATE** пользователем, для новой таблицы генерируется случайный UUID. Идентификатор также может быть указан явно, но это не рекомендуется т.к. может приводить к конфликтам и невозможности создать таблицу, если такой UUID уже используется. Для сокращенного синтаксиса запроса **ATTACH**, который содержит только имя созданной ранее таблицы, описание таблицы вместе с её идентификатором будут прочитаны из файла метаданных. Для запросов **ATTACH** с полным описанием таблицы идентификатор должен быть явно указан в запросе, чтобы можно было определить, где находятся данные.

При создании или присоединении таблицы движок **Atomic** обновляет глобальный маппинг UUID таблиц в **DatabaseCatalog**, после чего объект таблицы может быть получен по её идентификатору.

4.2. Атомарное переименование, запрос **RENAME**

Поскольку имя файла метаданных является единственным местом, где имя таблицы содержится на диске, то для переименования таблицы достаточно переименовать файл с метаданными и обновить простые структуры данных в памяти под короткой блокировкой обычного мьютекса, что не мешает параллельному выполнению запросов, работающих с данными таблицы. В случае аварийного

завершения процесса сервера во время выполнения запроса **RENAME**, метаданные не придут в неконсистентное состояние, т.к. переименование файла метаданных атомарно. Перемещение таблицы в другую базу данных с движком **Atomic** отличается лишь тем, что для этого нужно обновить структуру данных, хранящую в памяти множество таблиц базы, сразу в обеих базах данных. Для этого блокируются мьютексы обеих баз (всегда в определённом порядке).

Была реализована возможность перемещать таблицы между движками **Ordinary** и **Atomic**. Такое перемещение не атомарно, требует эксклюзивной блокировки табличных **RWLock** и работает как **DETACH** таблицы в одной базе данных и **CREATE** в другой.

Множественное переименование таблиц запросом

```
RENAME TABLE t1 TO t2, t3 TO t4, ...
```

работает как несколько отдельных переименований, т.е. в целом такой запрос выполняется неатомарно и некоторые **SELECT** могут наблюдать частично переименованные таблицы. Однако, наиболее частым случаем использования множественных переименований является замена одной таблицы на другую, например

```
RENAME TABLE table TO table_old, table_new TO table;
```

Специально для этого случая был реализован запрос

```
EXCHANGE TABLES table AND table_new;
```

который делает это атомарно. При обмене таблиц движок **Atomic** меняет местами их файлы метаданных и обновляет свои структуры данных в памяти аналогично переименованию таблицы.

Для атомарного обмена файлов используется доступный в ядре Linux с версии 3.15 системный вызов **renameat2** с флагом **RENAME_EXCHANGE** [6]. На других системах этот запрос не работает. Если же **renameat2** доступен, то движок Atomic использует его и для обычных переименований файлов метаданных с флагом **RENAME_NOREPLACE**, который запрещает перемещать файл, если путь назначения уже существует и занят другим файлом. Это защищает от случайного перезаписывания метаданных таблицы, которая была отсоединена запросом **DETACH**.

4.3. Запросы **DROP** и **DETACH**, фоновое удаление

При удалении таблицы движок Atomic помечает таблицу как удалённую, перемещая файл метаданных **/clickhouse_root/metadata/db_name/table_name.sql** в **/clickhouse_root/metadata_dropped/db_name.table_name.<uuid>.sql**, обновляет свои структуры данных в памяти под короткой блокировкой обычного мьютекса и уведомляет **DatabaseCatalog** о удалении таблицы. После этого таблица перестаёт быть видна новым запросам, при этом старые запросы могут продолжать использовать эту таблицу и эксклюзивная блокировка табличных **RWLock** не требуется, т.к. данные не удаляются. Это не мешает созданию новой таблицы с таким же именем, т.к. новая таблица будут иметь другой UUID и хранить данные в другой директории.

За фоновое удаление таблиц отвечает **DatabaseCatalog**. Он в отдельном потоке периодически просматривает список помеченных удалёнными таблиц и проверяет счётчик ссылок на таблицу в **StoragePtr**, который является **std::shared_ptr**. Поскольку в коде ClickHouse не используются **std::weak_ptr** для таблиц, а **shared_from_this()** может быть использован только в фоновых потоках таблицы, которые останавливаются вызовом **IStorage::shutdown()** ещё до удаления таблицы из базы, равенство счётчика ссылок единице означает, что **DatabaseCatalog** является единственным владельцем объекта таблицы, и все использовавшие таблицу запросы завершились. При выполнении этого условия фоновый поток **DatabaseCatalog** вызывает метод **IStorage::drop()**, чтобы оканчательно удалить данные таблицы с диска и информацию о реплике из ZooKeeper (для реплицируемых таблиц). В случае ошибки (например, из-за недоступности ZooKeeper) таблица помещается в конец очереди на удаление, и попытка повторяется позже. После успешного удаления таблицы файл метаданных удаляется из **metadata_dropped/**. Для дополнительной защиты от случайного запроса **DROP**, запущенного пользователем по ошибке, удаление таблицы может быть отложено на время, конфигурируемое настройкой **database_atomic_delay_before_drop_table_sec** (по умолчанию 8 минут). В течение этого времени таблица может быть восстановлена путём остановки сервера, ручного удаления файла из **metadata_dropped/** и выполнения запроса **ATTACH**.

Поскольку файлы метаданных для помеченных удалёнными таблиц находятся в отдельной директории **metadata_dropped/**, запрос **DROP DATABASE** может выполняться без ожидания фактического удаления всех таблиц в базе, а в случае завершения (в т.ч. аварийного) процесса сервера информация о помеченных удалёнными таблицах не теряется. При перезапуске сервера **DatabaseCatalog** просканирует эту директорию и добавит эти таблицы в очередь на удаление (для отсчёта времени задержки удаления используется время изменения файла). Таким образом, все помеченные удалёнными таблицы рано или поздно будут полностью удалены, не оставляя мусора на диске и в ZooKeeper.

При отсоединении таблицы запросом **DETACH** таблица удаляется только из структур данных в памяти движка базы данных, использующие таблицу запросы продолжают выполняться. Для защиты от присоединения этой таблицы запросом **ATTACH**, движок **Atomic** запоминает UUID отсоединенных таблиц и запрещает их присоединение до тех пор, пока использующие таблицу запросы не завершатся. В противном случае, могли бы существовать два инстанса таблицы, разделяющие данные на диске, но выполняющие запросы независимо друг от друга.

4.4. Использование **DDLGuard**

Поскольку точкой коммита для описанных выше запросов **CREATE**, **DROP** и **RENAME TABLE** является атомарное переименование файла метаданных, а параллельное выполнение этих запросов до момента переименования файла не должно нарушить каких-либо инвариантов, они могли бы выполняться без дополнительных блокировок для DDL-запросов. Запрос **CREATE** создаёт временный файл **table_name.sql.tmp** с флагом **O_EXCL** и записывает метаданные в него, а затем переименовывает этот файл в **table_name.sql**. То же самое делает запрос **ALTER** при обновлении метаданных таблицы. При перезапуске сервера **.sql.tmp** файлы не завершившихся запросов удаляются. Если выполнять все такие переименования с помощью renameat2 с флагом **RENAME_NOREPLACE** для запроса **CREATE** и обменивать файлы **.sql.tmp** и **.sql** для **ALTER** с флагом **RENAME_EXCHANGE** (с последующим удалением **.sql.tmp** файла), переименование файла метаданных будет также являться проверкой существования

или несуществования некоторой таблицы. Такая проверка не позволит, например, перезаписать метаданные только что созданной или переименованной параллельным запросом таблицы, потому что таблица с таким именем уже существует. Параллельно выполняющиеся **CREATE** с одинаковым именем таблицы не являются проблемой, т.к. они используют разные UUID, а до момента переименования файла метаданных создаваемые таблицы не видны другим запросам.

Однако, запросы **CREATE**, **DROP** и **RENAME** на время своего выполнения блокируют имена затрагиваемых таблиц с помощью **DDLGuard** по следующим причинам:

1. Запросы с **IF EXISTS** и **IF NOT EXISTS** таким образом дожидаются завершения других DDL-запросов с тем же именем таблицы и либо начинают выполняться, либо ничего не делают, если таблица уже создана/удалена другим запросом. Впрочем, после небольших изменений они могли бы выполняться параллельно, достаточно только не считать ошибкой для этих запросов неудачную попытку создания/удаления таблицы, которая уже существует/удалена.
2. Запросы **ATTACH** (в том числе с **IF NOT EXISTS**) для одной таблицы не могут выполняться параллельно, т.к. они будут использовать общий UUID таблицы, а в процессе загрузки таблица может выполнять не только readonly-операции. Может быть исправлено переносом таких операций в **IStorage::startup()**.
3. **DDLGuard** используется запросом **SYSTEM RESTART REPLICA**, который реализован через **DETACH** и **ATTACH** реплицируемой таблицы.
4. При множественном **RENAME TABLE** блокировка **GGLGuard** гарантирует, что не будет параллельных DDL-запросов с теми же именами таблиц, которые участвуют в переименовании.

В отличие от **RWLock**, использование **DDLGuard,** как правило, не создаёт каких-либо проблем, т.к. блокирует только DDL-запросы с таким же именем базы данных и таблицы. Такие запросы редко приходят на сервер одновременно, а если приходят, в большинстве случаев действительно являются конфликтующими (кроме случая **IF [NOT] EXISTS**), поэтому польза от отказа от использования этой блокировки не очевидна.

5. Заключение

В рамках этой работы в ClickHouse был реализован движок баз данных **`Atomic`**: https://github.com/ClickHouse/ClickHouse/pull/7512. Он не требует использования табличных блокировок для выполнения запросов **`CREATE`**, **`DROP`** и **`RENAME`** и лишён основных недостатков движка **`Ordinary`**, которые были описаны во введении.

6. Список источников

[1] RWLock.cpp // GitHub URL: https://github.com/ClickHouse/ClickHouse/blob/ce073e07523888d67e2e0cdb7f0877ddc5f377a3/dbms/src/Common/RWLock.cpp (дата обращения: 2020-05-20).

[2] Fixed very rare deadlocks // GitHub URL: https://github.com/ClickHouse/ClickHouse/pull/6764 (дата обращения: 2020-05-20).

[3] System Catalogs // PostgreSQL Documentation URL: https://www.postgresql.org/docs/9.1/catalogs.html (дата обращения: 2020-05-20).

[4] Transactional DDL // PostgreSQL wiki URL: https://wiki.postgresql.org/wiki/Transactional_DDL_in_PostgreSQL:_A_Competitive_Analysis (дата обращения: 2020-05-20).

[5] Metadata Storage // Apache Druid URL: https://druid.apache.org/docs/latest/dependencies/metadata-storage.html (дата обращения: 2020-05-20).

[6] rename(2) // Linux manual page URL: http://man7.org/linux/man-pages/man2/rename.2.html (дата обращения: 2020-05-20).