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
на тему
РЕПЛИЦИРОВАННЫЕ БАЗЫ ДАННЫХ

Выполнил студент группы БПМИ166, 4 курса,
Батурин Валерий Юрьевич

Научный руководитель:
доцент,
базовая кафедра Яндекс,
Миловидов Алексей Николаевич

Москва 2020

Аннотация

Репликация данных на сегодняшний день является распространенной потребностью среди пользователей различных СУБД, включая Clickhouse. В данной работе описан процесс создания нового движка баз данных в Clickhouse, который позволит реплицировать метаданные таблиц автоматически. Это улучшение позволит удобно работать с реплицированными таблицами. Здесь будет описано три подхода к реализации данного движка, приведены сравнения между ними и обоснование лучшего из них, а также приведины технические подробности реализации нового движка Clickhouse.

Ключевые слова: Clickhouse, СУБД, репликация данных, zookeeper.

Abstract

Nowadays, data replication is a common user's need when an application is a database management system. Clickhouse has a special family of tables which covers this need. They're called replicated tables. As we know, a set of tables always belongs to a database and in Clickhouse databases can have different engines. In this paper, the process of implementing a new Clickhouse database engine is described. The new engine called DatabaseReplicated is aimed to make users' work with replicated tables easier. Technical details and 3 approaches to metadata log replication are covered here.

Keywords: Clickhouse, DBMS, data replication, zookeeper.

Содержание

1 Введение

1.1 Описание задачи

Задача состоит в реализации движка баз данных в Clickhouse[1], который бы осуществлял репликацию метаданных, в частности такие данные, как лог DDL операций(`CREATE`, `DROP`, `RENAME`, `ALTER`) и множество имеющихся таблиц. Помимо этого нужно обеспечить восстановление реплик таких баз данных, а также при создании новых реплик осуществить загрузку на них последних метаданных.

1.2 Текущая ситуация с репликацией

На данный момент репликация в Clickhouse осуществима лишь на уровне отдельных таблиц. Данное решение является очень гибким. К примеру, осуществима следующая ситуация: на каком-либо сервере одна из таблиц может быть не реплицирована, другая иметь двухкратную репликацию, а третья - реплицирована по всем серверам. Однако, если представить, что все таблицы в базе данных реплицированы одинаковым образом, то управление таким кластером не является оптимальным. К примеру, при необходимости восстановления сервера после падения, необходимо создавать реплику каждой таблицы в индивидуальном порядке. При реализации реплицированного движка баз данных, управление репликами станет заметно проще.

1.3 Обзор хранения данных в Clickhouse

Очевидно, что пользователи Clickhouse заинтересованы в надёжном хранении данных ввиду того, что они представляют собой определённую ценность. Для оптимизации хранения и обработки гомоморфных данных используются типы данных. Так, имея необходимость работы с датами, наиболее выгодной стратегией будет хранить эти данные как тип Дата, который имеется в Clickhouse под названием Date. Более того, оперируя данными конкретных типов, можно использовать специальные функции, предназначенные для обработки конкретных типов данных. Например, можно посчитать разницу в

днях между двумя датами, используя соответствующую функцию. При хранении данных о дате в виде текстового типа такой возможности не было бы. Далее данные, которые связаны между собой, принято хранить внутри одной строки. К примеру, если мы имеем дело с людьми, то нам зачастую нужно хранить имя и фамилию некоего человека. Этот человек будет иметь специальную строку, под его информацию. Если же мы имеем дело с некоторым множеством людей, где информация о человеке записывается в строку и каждая строка имеет одинаковые типы, то эти данные помещаются в таблицу. Поднявшись еще на один уровень выше, объединение разных таблиц называется базой данных. Внутри Clickhouse можно управлять более чем одной базой данных. Более того, базы данных в Clickhouse отличаются не только по составу, схеме и названиям таблиц внутри них, но и еще по движку. Данная работа направлена на создание нового движка баз данных, где ключевой особенностью будет репликация.

2 Исследование компонент Clickhouse

Для того, чтобы реализовать новый движок баз данных в Clickhouse[2], необходимо иметь представление о релевантных компонентах кодовой базы. В этой секции будет приведет обзор основных и наиболее важных компонент, взаимодействие с которыми было неизбежным на пути решения поставленной задачи.

2.1 Движки баз данных

Так как сам реплицируемый движок баз данных является в первую очередь движком баз данных, опишем что представляет собой движок в терминах Clickhouse. Кодовая база предоставляет интерфейс IDatabase, который ответственен за

- инициализацию множества известных таблиц и словарей,
- проверку существования таблицы и получения таблицы как объекта
- получения списка со всеми таблицами,
- создание и удаление таблиц,
- переименование таблиц и их перемещение между базами данных одного и того же типа движка.

Таким образом видно, что движок базы данных является своего рода менеджером таблиц и более высокоуровневой абстракцией, нежели просто множеством таблиц.

На текущий момент Clickhouse предоставляет выбор из X движков для использования, каждый из которых обладает уникальными свойствами и особенностями. Однако в рамках текущей работы наибольший интерес представляют такие движки как DatabaseOnDisk и DatabaseAtomic. Стоит добавить, что движки баз данных образуют иерархию путём использования такой особенности языка C++ как подтиповый полиморфизм. Так, DatabaseAtomic является подтипом DatabaseOnDisk, а предложенный в этой работе DatabaseReplicated является подтипом DatabaseAtomic, что в свою очередь делает его в том числе и подтипом DatabaseOnDisk. Вершиной дерева иерархии движков является движок IDatabase, который был описан выше.

2.1.1 DatabaseOnDisk

Говоря о базах данных мы подразумеваем коллекцию всех данных, которые описывают что находится внутри базы, при этом не имея в виду сами данные. То есть база данных отвечает за структуру хранимых данных. Существует различие в определении данных, которые непосредственно хранятся в таблице и таких данных о таблицах, как описание структуры таблицы, а именно типов и наименований полей в строках; помимо этого в Clickhouse таблицы, как и базы данных, могут иметь различные движки, о которых будет дана информация ниже. Такие данные о таблице, характеризующие ее структуру, но не относящиеся непосредственно к экземплярам хранимых данных называются **метаданные таблиц**. Свойство движка баз данных "на диске"заключается в хранении метаданных таблиц на диске. Хранение на диске, в отличие от хранения в оперативной памяти, обладает свойством персистентности. Таким образом можно заключить, что используя экземпляры движков баз данных DatabaseOnDisk в Clickhouse, мы имеем дело с персистентным хранением метаданных. Это позволяет базам данных восстанавливать схему таблиц даже после падения сервера или плановой перезагрузки.

2.1.2 DatabaseAtomic

Стандартным движком баз данных, на текущий момент, является DatabaseOrdinary. Под стандартным здесь понимается, что если не указать определённый движок при создании базы данных, то будет выбран именно стандартный. Однако в данной работе DatabaseReplicated является подтипом DatabaseAtomic, который в свою очередь является подтипом DatabaseOrdinary. Для того, чтобы понять, чем отличается DatabaseAtomic от DatabaseOrdinary, необходимо дать описание процессу хранения данных, где под данными подразумевается непосредственно строки таблиц. На уровне кода даётся такая абстракция как **Storage**, которая отвечает, помимо прочего, непосредственно за хранение данных таблиц. Для обычных, невиртуальных таблиц, хранение данных производится на диске. Путь, по которому лежат данные отличается для таблиц, управляемыми движком DatabaseAtomic. Данные этих таблиц располагается по пути $/clickhouse\_path/store/prefix/UUID/$, где UUID – это универсальный уникальный идентификатор таблицы, а prefix – это первые три символа

UUID. Такое ограничение на путь дает возможность выполнять CREATE и RENAME запросы, без лока в Storage на чтение и запись.

2.2 Движки таблиц

В рамках текущей работы наибольший интерес представляют реплицированные таблицы. Сам алгоритм репликации основан на структуре данных Merge Tree. Если нереплицированные таблицы работают со своими данными как есть, то реплицированные таблицы с учетом наличия более чем одной реплики (в противном случае репликация с одной репликой не является репликацией как такой) реплицируют данные запросов INSERT и ALTER.
Однако запросы CREATE, DROP, ATTACH, DETACH и RENAME выполняются только на одной реплике. Можно заметить, что, разные реплики одной и той же таблицы могут иметь разные названия, что вызывает своего рода неконсистентность в пользовательском представлении. Хорошая новость состоит в том, что репликацию таких запросов можно делать на уровне движка баз данных, на что и направленна эта работа.

2.3 Интерпретация запросов

Запуская сервер Clickhouse, пользователи могут отправлять запросы на него используя клиент, или взаимодействовать напрямую через один из поддерживаемых протоколов. В не зависимости от того, по какому именно протоколу пришел запрос на сервер, и опуская такие вещи, как авторизация запросов, давайте рассмотрим, как именно Clickhouse реагирует на поступивший запрос. Условно обработку запроса можно разбить на две стадии, а именно Parsing и Interpreting.

2.3.1 Context

Стоит учитывать, что обработка запросов внутри Clickhouse не происходит в изоляции, то есть обработка одного и того же запроса может быть интерпретирован или исполнен по разному, в зависимости от того, в каком состоянии сейчас находится сервер. Примером различного исполнения и интерпретации

может служить запрос `DESC TABLE clicktable;`. В зависимости от состояния, а именно в зависимости от выбранной базы данных в момент интерпретации запроса может оказаться, что имеется в виду таблица разных баз данных. Например, если перед запросом `DESC TABLE clicktable;` был запрос `USE mydatabase`, то интерпретация будет такой:
`DESC TABLE mydatabase.clicktable;`
И для одной и той же интерпретации может быть разный результат исполнения. Это вполне очевидно, так как описание схемы таблицы, зависит от того, какая сейчас схема у этой таблицы. За состояние Clickhouse сервера отвечает такой объект как Context. Context необходим для интерпретации запросов.

2.3.2 Interpreter

Взаимодействие с Clickhouse сервером начинается с функции `executeQuery`, которая является входной точкой в интерпретирование запросов. В зависимости от типа запроса, для его обработки вызывается один из 35 интерпретаторов. К примеру, для запроса типа `create` интерпретатор понимает создание какого именно типа объекта ожидается от запроса (базы данных, таблицы или словаря), и после некоторых преобразований запроса и проверок валидности выполняется сам запрос. Так, например, для создания таблицы внутри базы данных с движком Atomic необходимо на этом этапе сгенерировать UUID таблицы, более того, если таблица создается впервые и в запросе указан её UUID, то это считается некорректным запросом и интерпретатор отказывается его выполнять, сообщая пользователю об ошибке. Создание таблицы повторно тоже не является корректным запросом. Но давайте вернемся к первому примеру. Было сказано, что UUID таблицы добавляется в запрос. Известно, что запрос приходит в строковом представлении, но модифицирование строковых данных обладает свойством неэффективности в случае добавления в середину подстроки. Более того, на каждую проверку приходилось бы итерироваться по строке и анализировать её. Такая стратегия исполнения запросов не могла бы отличиться своей эффективностью, скорее наоборот, давала бы значительные задержки на каждый запрос. Поэтому перед исполнением запроса, запрос преобразуется в эффективный тип, для удобной обработки компьютером.

2.3.3 Parsing

Преобразование представления запроса происходит на этапе парсинга запроса. Из строкового типа по завершению этапа парсинга получается тип дерева абстрактного синтаксиса (`AST`). Точкой входа в этап парсинга является функция `parseQuery`. Эта функция вызывается внутри `executeQuery`. Для построения дерева из строкового представления необходимо для начала преобразовать строку в набор токенов. Для большей эффективности, используется итератор. Получение токенов из `TokenIterator` происходит при помощи лексера, в котором описаны все лексемы языка запросов.

3 Пользовательский интерфейс

Для создания нового движка баз данных в Clickhouse достаточно создать класс, который будет наследовать интерфейс IDatabase, и зарегистрировать его в DatabaseFactory.
По итогу у нас выходит следующий вариант пользовательского интерфейса для создания базы данных с реплицированным движком:
`CREATE DATABASE name ENGINE = Replicated(zk_path, replica_name)`,
с параметром имени для самой базы, а также двумя специфичными для реплицируемой базы параметрами - путь в зукипере[3] и уникальное имя реплики. Для того, чтобы создать реплику, необходимо указать при создании базы данных тот же путь в зукипере, и другое имя реплики. На одном сервере нельзя создать более одной реплики ввиду конфликта UUID таблиц, которые возникают при использовании DatabaseAtomic как родительского класса.

4 Алгоритм репликации

Помимо интерфейса создания базы данных с новым движком, ещё существует реализация самой базы данных, которая управляет таблицами. В нашем случае база данных реплицирует метаданные таблиц в зукипере и реализует управление репликами при помощи запросов лишь к одной реплике.

4.1 Общий источник метаданных

Изначальная попытка реализации реплицированного движка была на основе уже существующего движка, который работал с метаданными. Как известно из описания DatabaseOnDisk, данный движок обеспечивает хранение метаданных на диске. Соответственно реализация движка DatabaseReplicated заключалась в том, чтобы все метаданные точь-в-точь хранить помимо того, что на диске, но ещё и в зукипере. Но в таком случае появляется вопрос, как быть с тем, что на разных репликах могут быть разные метаданные в один момент времени?
Предположим, что какой-то из реплик уже пришел запрос на изменение схемы таблицы. Тогда, другим репликам нужно подгрузить эти метаданные. К

тому же на незначительные изменения схемы таблицы приходилось бы полностью переопределять схему на репликах. Также возникало много проблем с самими данными, так как при изменении схемы таблицы, данные которой уже лежат на репликах, не хотелось бы перезаписывать или перезаливать их.

Помимо этого, что делать, если реплика, которая изменила метаданные и адаптировала данные к новому типу становится неактивной по тем или иным причинам?

Использование общего источника метаданных создает много проблем, хотя и является очень простым и на первый взгляд изящным решением. Но от данной идеи было принято решение отказать в пользу иного подхода.

4.2 Хранение реплицированного лога

Следующий подход использует упорядоченный набор запросов как источник текущего состояния базы данных на всех репликах. При создании запроса на реплике, перед тем как она сама его исполнит, будет вызван метод базы данных `propose`, который принимает дерево абстрактного синтаксиса запроса и записывает его в строковом представлении как узел зукипера с названием `log.N`, где N – это порядковый номер запроса. Каждая из реплик имеет свой локальный номер последнего исполненного запроса. Для решения конфликта конкурирующих записей при параллельном исполнении метода `propose` на разных репликах используется блокировка[4] на основе фантомного узла в зукипере.

Помимо метода `propose`, который вызывается на этапе интерпретации, в случае если движком выбранной базы данных является Replicated, отличительной особенностью движка является то, что при старте запускается в фоне тред в специально выделенном тредпуле под реплицированные базы данных, который отвечает за обновление состояния и исполнение новых запросов. В случае, если на реплику приходит запрос, который нужно поместить в лог, то в начале реплика обновляется до последнего состояния, а затем уже происходит запись запроса в лог и исполнение этого запроса на всех репликах.

4.3 Гибридный подход

Подход с хранением реплицированного лога является оптимальным решением в в плане консистентности метаданных: при использовании реплицированного лога реплики, которые отстают всегда могут постепенно накатить обновления и продолжить работу. Однако было принято решение использовать снепшоты состояния и регулярную чистку лога в с определенной частотой.

Подход снепшотирования является хорошей практикой сам по себе, он позволяет ограничивать максимальное количество записей в логе, а также ускоряет процесс обновления до последнего состояния для новых реплик.

Для упрощения реализации, создание снепшота происходит с реплики, на которой вызван `propose` и происходит лишь в том случае, если все зарегистрированные реплики находятся в последнем состоянии. Такая реализация элиминирует возможность нахождения реплик в неконсистентном состоянии. И при этом реплики имеют возможность обновиться до свежего состояния путем исполнения лога.

5 Технические подробности

Для исполнения лога с запросами создается задача в специальном Background Pool, которая выполняет `executeQuery` с особым контекстом, в котором указано, что запрос пришел из лога реплицированной базы. Это нужно для того, чтобы избежать рекурсивного добавления записи в лог.

Далее интерпретатор запросов в случае исполнения запроса пришедшего из лога подменяет базу данных, по отношению к которой выполняется запрос. Так как базы данных, которые являются репликами, локально могут иметь различные имена, то спецификация базы данных внутри лога не имеет значение, однако во время интерпретации запроса необходимо подменить базы данных на базу данных соответствующую реплике. Это сделано с помощью `setCurrentDatabase` контекста и модификации дерева абстрактного синтаксиса.

Ещё одной интересной подробностью реализации будет являться уникальная обработка запросов связанных с реплицированными таблицами. Первая осо-

бенность заключается в создании таких таблиц. Так как у нас уже есть путь в зукипере и уникальное название реплики из базы данных, то для создания реплицируемых таблиц не обязательно указывать путь в зукипере для таблицы и название реплики таблицы, как это требуется в случае использования всех остальных движков. Путь в зукипере для таблицы берётся как конкатенация пути зукипера базы данных, `/tables/` и UUID таблицы. Так как реплицированный движок основан на DatabaseAtomic, то у таблиц есть UUID, к тому же этот UUID вычисляется на реплике, к которой приходит запрос на создание таблицы и она добавляет его в дерево абстрактного синтаксиса запроса, таким образом унифицируя этот параметр для всех реплик, что добавляет консистентности в пути хранения самих данных. Для названия реплики таблицы используется название реплики базы данных, что не создает конфликтов и является простым для использования решением.
Вторая особенность заключается в полном игнорировании `ALTER` запросов для таких таблиц. Это связано с тем, что если `ALTER` запрос успешно завершен для ReplicatedMergeTree, то в логе таблицы будет запись `ALTER_METADATA`. Затем, если возвращается старая реплика, которая была недоступна, то можно запустить таблицу со старыми метаданными. Дальше таблица либо обновит свое состояние применив лог (после чего вызовет `IDatabase::alterTable`, чтобы обновить локальные метаданные в базе), либо же полностью склонирует состояние одной из живых реплик. В любом случае такая таблица сама придёт к нужным метаданным без участия базы данных. Получается, при сохранении лога базы можно просто игнорировать `ALTER` запросы реплицируемых таблиц.

6 Список литературы

[1] Clickhouse documentation. — https://clickhouse.tech/docs/en/. — 2020.

[2] Clickhouse source code. — https://github.com/ClickHouse/ClickHouse. — 2020.

[3] Zookeeper: Wait-free coordination for internet-scale systems. / Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, Benjamin Reed // USENIX annual technical conference. — Vol. 8. — 2010.

[4] A guide to creating higher-level constructs with zookeeper. — https://zookeeper.apache.org/doc/r3.1.2/recipes.html#sc_recipes_Locks.