Москва 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
на тему
Поддержка использования в ClickHouse систем координации помимо ZooKeeper

Выполнил студент группы 165, 4 курса,

Левушкин Алексей Сергеевич

Руководитель ВКР:

Доцент, М базовая кафедра Яндекса

Миловидов Алексей Николаевич

Аннотация

ClickHouse – современная, все больше набирающая популярность по всему миру СУБД, которая как и любая современная система управления базами данных заточена на скорость и надежность. Один из аспектов надежности достигается за счет реплицируемых таблиц. Для их хранения используется сразу несколько реплик, в целях координации которых ClickHouse использует Apache ZooKeeper. Но не все пользователи ClickHouse хотят использовать его для репликации. Из-за чего актуальной задачей становится поддержания альтернативной системы координации, схожей по своему функционалу. Наиболее близкая из таких систем - Etcd, о которой и пойдет речь ниже.

Abstract

ClickHouse is a modern and increasingly gaining worldwide popularity DBMS that, like any today's database management system, was engineered to be reliable and speedy. One aspect of reliability is achieved through replicated tables, for storage of which several replicas are used. To coordinate those replicas, ClickHouse uses Apache ZooKeeper, a distributed key-value storage. It provides API that makes it possible to implement different primitives for reliable configuration storing. However, not every ClickHouse user wants to use it for replication. Maintaining a similar alternative coordination system is, therefore, a relevant problem. Etcd is the closest of these systems, and we will describe it below.

Ключевые слова

СУБД, репликации, координация реплик.

Оглавление

1. Введение

ClickHouse – современная, все больше набирающая популярность по всему миру СУБД, которая как и любая современная система управления базами данных заточена на скорость и надежность. Один из аспектов надежности достигается за счет реплицируемых таблиц. Для их хранения используется сразу несколько реплик, в целях координации которых ClickHouse использует Apache ZooKeeper. Но не все пользователи ClickHouse хотят использовать его для репликации. Из-за чего актуальной задачей становится поддержание альтернативной системы координации, схожей по своему функционалу. Наиболее близкая из таких систем – Etcd.

2. Актуальность

Несмотря на то, что поставленная перед нами задача реализует не новый функционал, а лишь дополнительную реализацию, она актуальна, так как в ней заинтересованы многие пользователи, подтверждением чему является популярное issue в репозитории. Целью работы является разработка реализации, дающей возможность использования Etcd для координации реплик ClickHouse, так как не все пользователи хотят использовать для этого ZooKeeper. Причина этому отсутствие у ZooKeeper «service discovery», проблематичная настройка (необходима JVM на каждом хосте и тд). Исходя из этого поддержание альтернативной системы координации сделает репликацию более прозрачной в глазах пользователей и более надежной на практике. Использовать Etcd вместо ZooKeeper весьма проблематично, так как она существенно отличается по интерфейсу и возможностям. Тем не менее для того, чтобы эта задача стала возможной, в ClickHouse обобщен интерфейс взаимодействия с ZooKeeper, благодаря чему на его место можно подставить другие реализации.

3. Цели и задачи

Как уже было сказано выше, целью работы является возможность использования альтернативной системы координации.

Были поставлены следующие задачи:

- исследовать алгоритм репликации таблиц ClickHouse

- исследовать систему ZooKeeper, которая используется для координации реплик

- найти систему наиболее близкую к функционалу ZooKeeper

- разработать схему хранения данных оптимизировав ее под выбранную систему

- реализовать код взаимодействия с выбранной системой

4. Обзор систем

4.1. ClickHouse

ClickHouse – open-source столбцовая система управления базами данных для онлайн обработки аналитических запросов под лицензией Apache-2.0, в основе работы которой лежит семейство движков MergeTree. Их основаная идея заключается в том, что бы при большом объеме вставляемых данных записывать много небольших отсортированных по первичному ключу блоков, а затем сливать их в большие отсортированные блоки в фоне. Тогда через некоторое время любая запись окажется в одном из больших отсортированных блоков, что позволяет быстро находить любую запись по первичному ключу. Такой алгоритм обработка данных в фоне позволяет экономить время при записи. Движок этого семейства под названием ReplicatedMergeTree обладает такими же свойствами, а также

поддерживает репликацию данных. В отличие от других СУБД, в ClickHouse реплицируются именно таблицы, а не сервера.

Подробнее про алгоритм репликации. Пусть есть таблица T семейства ReplicatedMergeTree, которая содержит в себе блоки b_1, …, b_n. В некоторый момент времени в ClickHouse сервер приходит клиент, и записывает новые данные. В таком случае создается новый блок, который записывается на ту реплику, на которую пришел клиент, после чего информация про записанный блок отображается в ZooKeeper, и уже после, остальные реплики заметив и считав информацию из ZooKeeper начинают скачивать этот блок себе по средствам обычного http-запроса. Для такой координации реплик используется ZooKeeper.

4.2. ZooKeeper

Подробнее о ZooKeeper. Его основаная задача – предоставить простое и высокопроизводительное ядро для построения более сложных координационных примитивов на клиенте. Если говорить более формально, то это распределенное key-value хранилище, с помощью API которого можно управлять простыми объектами данных, организованных иерархически, как в файловых системах. ZooKeeper поддерживает асинхронную работу с данными, что является важной особенностью, которая позволяет использовать его в высоконагруженных системах.

Данные в ZooKeeper представлены в виде znode, схожих с узлами в файловой системе, где полный путь до него это key, а содержимое это value. Так же узлы имеют детей, по аналогии с файловой системой. Кроме обычных узлов, ZooKeeper имеет эфимерные узлы, которые существуют пока жив создавший его клиент и последовательные узлы, в путь к которым добавляется последовательный номер (например /blocks/block-0000000001), для того, чтобы нумеровать узлы в общем для всех дочерних узлов порядке, возрастающем аналогично порядку создания узлов.

Важной особенностью является «watch» запросы, которые позволяют подписываться и следить за изменениями узлов.

ZooKeeper API предоставляет следующие вызовы для работы с узлами:

1. create(path, data, flags) – создает znode с переданными параметрами
2. delete(path, version) – удаляет znode если переданная версия коректна
3. exists(path, watch) – сообщает о наличии znode, и устанавливает флаг наблюдения если такой передан
4. getData(path, watch) – возвращает данные znode и устанавливает флаг наблюдения
5. setData(path, watch) – устанавливает новое значение znode если переданная версия корректна
6. getChildren(path, watch) – возращает список имен всех потомков znode
7. check(path, version) – проверяет соответствие версии вершины
8. multi(requests) – атомарно применяет все запросы, если все применились успешно, иначе не применяет ни один (поддерживаются только create, remove, set, check).
9. sync(path) – ждет завершения всех обновлений

* передаваемая версия корректна, если она совпадает с версией znode или равна -1

Каждый из этих вызывов имеет синхронную и асинхронную реализацию, но ClickHouse использует только асинхронную, по причине лучшей производительности.

Рассмотрим MultiRequest внимательно. Пусть есть multi_request состоящий из запросов r_1, $r_1 \ldots r_n$. И состояние хранилища X до применения multi_request-а. Тогда применение multi_request происходит следующим образом: по очереди обходим запросы r_i и формируем ответ q_i путем применения запроса r_i к состоянию X_{i-1}, где состояние X_i – состояние полученное путем применения запросов с 1 по i-тый. Если на шаге j применить операцию не удалось, состояние хранилища возвращается к состоянию X, ошибкой multi_request-а становится ошибка r_j операции, а ответами multi_request-а все ответы с 1 по j-ый. Если все операции применились успешно, то состояние хранилище переходит в состояние X_n, а ответами, все ответы с 1 по n-ый.

Смоделируем ситуацию в которой X = {/root}, r_1 = create(/root/tmp), r_2 = create(/root/tmp/x1), r_3 = set(/root/tmp, “tmp dir”), r_4 = remove(/home), в таком случае, multi_request не применится с ошибкой NONODE при применении r_4.

О ZooKeeper в ClickHouse. Как было сказанно выше, любая таблица ClickHouse состоит из блоков. Так как эти блоки имеют большой размер, ZooKeeper лишь оркестрирует их метаданными, а также метаданными реплик. Тоесть ZooKeeper является общим для всех реплик хранилищем метаданных, с помощью которых происходит оркестрирование репликацией таблиц. Примеры данных, которые хранятся в ZooKeeper:

- метаданные реплики - такая информация как активность реплики, наличие блоков таблицы, обрабатываемые сейчас операции и другие данные, записывается в потомков узла /replicas.

- хэш каждого блока для последующей проверки блока на целостность, записывается в потомков узла /blocks.

- информация о появлении новых блоков записывается в /log.

Для примера оркестрации реплик разберем такой важный аспект репликации как синхронизация блоков между репликами. Пусть на одну из реплик пришел новый блок таблицы b_i, тогда из него создается временный кусок в названии которого добавляется sequential number, полученный с помощью sequential узла (это помогает автоматически нумеровать все поступающие блоки). После чего этот блок добавляется в виде структуры MergeTreeData в состоянии PreCommited, это состояние говорит о том, что блок готов, но его еще нельзя использовать, так как информация об этом блоке еще не записана в ZooKeeper. Запись в ZooKeeper происходит в следующем порядке. Сначала записывается информацию о добавленном блоке в узел /log, из которого все остальные реплики узнают о новом блоке, так как подписаны на его изменения. Далее в /blocks записывается хэш

нового блока, для предотвращения дубликации. А также записывает информацию о блоке в /replicas/<r>/parts, чтобы обозначить наличие блока на реплике r. И только после всего этого блок помечается как Commited, и доступен для использования.

4.3. Etcd

Etcd – распределенное key-value хранилище для наиболее важных данных распределенных систем. Использует алгоритм консенсуса RAFT, что делает его более простым в реализации, чем ZooKeeper с ZAB. Он имеет две основные версии, которые значительно отличаются функциональностью. Вторая версия Etcd имеет только http API, из-за чего взаимодействие происходит с помощью синхронных http запросов, что не подходит для ClickHouse из соображений скорости. В третьей версии добавилась возможность взаимодействия через grpc, а значит и асинхронные запросы, так же появились транзакции, хранилище ключей стало плоским из-за чего теперь нет понятия директория, а только понятие ключ, а также ttl значений заменили на lease, что позволяет реализовать ephemeral узлы. Исходя из большей совместимости с ZooKeeper была выбрана третья версия, и взаимодействие через grpc. А значит дальнейшее упоминание Etcd подразумевает именно Etcd v3.

Etcd API. Мы будем использовать 3 grpc сервиса: KV, Watch, Lease.

Методы KV сервиса (осуществляется через Unary RPC):

- PutRequest(key, value) – запрос на создание пары или ее обновление

- RangeRequest(key, range_end) – запрос диапазона значений

- DeleteRangeRequest(key, range_end) – запрос на удаление диапазона значений

- Compare(compare_target, compare_result, key) – запрос на сравнение пары с ключем Key, где CompareTarget = {VERSION, CREATE, MOD, VALUE}, CompareResult = {EQUAL, NOT_EQUAL, LESS, GREATER}

- TxnRequest(compares, success_request_ops, failure_request_ops) – транзакция со сравнениями, которая вполнит success_request_ops если все сравнения успешны и failure_request_ops если хотя бы одно сравнение не успешно

- PutResponse(prev_kv) – ответ содержащий прошлую пару kv

- RangeResponse(kvs) – ответ содержащий набор пар kvs

- DeleteRangeResponse(deleted, prev_kvs) – ответ содержащий флаг, показывающий успешно ли завершилось удаление и набор содержащий предыдущие пары prev_kvs

- TxnResponse(succeeded, reponses) – ответ содержащий флаг, показывающий результат compares и successful или failure ответы в зависимости от результата сравнений.

Методы Watch сервиса (осуществляется через bidi streaming RPC):

- WatchRequest(WatchCreateRequest(key, range_end), WatchCancelRequest(watch_id)) – Запрос на создание или отмену Watch-а

-WatchResponse(watch_id, events(key, prev_kv))- ответ с event-ами содержащими затронутые kv

Методы Lease сервиса (осуществляется через bidi streaming RPC):

- LeaseGrantRequest(ttl) – запрос на создание lease с тайматуом ttl (в секундах)

- LeaseGrantResponse() – ответ, подтверждающий создание lease

- LeaseKeepAlive(lease_id) – обновление ttl у lease

Все описанные выше методы и сервисы можно посмотреть в файле rpc.proto.

Etcd поддерживает взаимодействие через grpc, что позволяет пользователю выбирать механизмы работы с сервисом (синхронный или асинхронный).

4.4. gRPC

gRPC — высокопроизводительный фреймворк для вызова удаленных процедур (RPC), разработанный компанией Google. Позволяет создавать клиентские библиотеки для работы с бэкендом на 10 языках. Производительность достигается за счет использования протокола HTTP/2 и Protocol Buffers. GRPC поддерживает 4 типа RPC: Unary RPC, Server streaming RPC, Client streaming RPC, Bidirectional streaming. Каждый из которых поддерживает синхронный и асинхронный режим. Асинхронный режим достигается благодаря использованию очереди запросов CompletionQueue. Алгоритм работы с CompletionQueue выглядит так: связываем запрос с ComplitionQueue уникальным тегом и вызываем метод ComplitionQueue::Next, блокирующий поток, до того момента пока не будет получен ответ какого либо из запросов, после чего ComplitionQueue возвращает тег, связанный с запросом.

4.5. Etcd vs ZooKeeper или почему пользователи ClickHouse не хотят использовать Apache ZooKeeper

ZooKeeper написан на языке java, а значит нуждается в JVM фиксированной версии на каждом узле кластера, а так же наследует множество проблем java (garbage collector и тд). Так же настройка ZooKeeper требует множество конфигурации, что увеличивает порог входа. Очередной проблемой является недоступность сервиса при создании снепшотов. Несмотря на все описанные выше минусы, Apache ZooKeeper используется в большом количестве систем, так как имеет длинную историю разработки, большое количество примеров, а также много вспомогательных инструментов и библиотек, написанных под различные языки. Если говорить про Etcd, то это активно разрабатывающаяся система, написанная на языке go, благодаря чему это всего лишь бинарный файл без конфигураций, лишенный описанных выше болячек.

5. Шаги для решения задачи

5.1. Добавление gRPC в ClickHouse

Для того что бы взаимодействие через grpc стало возможным необходимо добавить grpc в ClickHouse как third_party. Так как библиотека grpc использует некоторые библиотеки, которые уже присутствуют в contrib-ax ClickHouse-a, например protobuf, zlib, openssl, было принято решение положить отдельно сабмодуль grpc, и отдельно сабмодуль grpc-cmake, с единственным файлом CMakeLists.txt, который добавляет grpc в сборку, пере использовав уже используемые библиотеки. Так как в ClickHouse есть некоторые правила, по которым должны быть написаны CMake файлы для библиотек, CMakeLists.txt для grpc был переписан.

5.2. Анализ ключевых различий систем

Первое различие в схеме хранения узлов. ZooKeeper хранит узлы иерархически, что позволяет думать в рамках понятий «родитель», «ребенок», хранилище ключей Etcd плоское, а значит между ключами нет отношений связи.

Следующее различие в том, что znode в etcd – это не только пара key – value, а еще и набор атомарно изменяющихся метаданных (cxid, mxid, ctime, mtime, version, cversion, aversion, ephemeralOwner, dataLenght, numChildren, pzxid), активно использующихся в ClickHouse. Узел Etcd определяется парой key-value, версией, а также ревизией создания и модификации.

Еще одно важное отличие – 3 вида узлов в хранилище ZooKeeper: обычный, эфимерный и последовательный. В рамках Etcd не существует эфимерных и последовательных узлов.

И наиболее серьезное различие в свойстве транзакций. О котором мы поговорим ниже.

5.3. Изучение похожих решений

Zetcd – open-source проект, который как и Etcd написан на языке Go и является некоторой прокси, которая принимает вызовы отправленные в ZooKeeper, интерпретирует и отправляет их в Etcd, тем самым позволяет работать с Etcd по правилам ZooKeeper API. Данная библиотека не подходит для задач репликации в ClickHouse, так как не может использоваться как часть кода, а также имеет низкую производительность. Первая причина этому - идеология proxy, которая порождает 2 и более запроса на каждый запрос ClickHouse, а также схема не оптимизированная под ограниченные нужды ClickHouse. Zetcd использует следующую схему данных. На каждую единицу метаданных узла создается по узлу, из-за чего из одного узла в ZooKeeper создается 9 узлов в Etcd, что негативно сказывается на производительности из-за огромного количества узлов. Однако выделять отдельные узлы для некоторых метаданных znode – это правильная тактика из-за особенностей Etcd API.

5.4. Разработка схемы хранения данных

Важное свойство ZooKeeper – узлы, расположенные иерархически, а значит у каждого узла есть родитель и дети. При работе с плоским хранилищем Etcd , для получения родительского узла воспользуемся обрезанием строки-ключа текущего узла. Для получения дочерних узлы воспользуемся RangeRequest-ом с префиксом. Однако по свойствам ZooKeeper дети узла – это узлы уровень которых на один больше текущего, а значит RangeRequest вернет нам не детей узла, а потомков, из которых необходимо получить детей узла с помощью обрезания строки. Посчитаем сложность. Сложность получения ключа родителя O(key.size()), сложность получения детей узла O(n * key.size()), где n – количество значений в хранилище. Видно, что получение детей узла не оптимально, к тому же каждый лишний ключ увеличивает размер сообщения, а значит ведет к переполнению. Для того что бы сделать операцию получения детей оптимальнее, добавим номер уровня в начало

пути каждого узла, тоесть путь /clickhouse/task_queue/ddl превратится в /3/clickhouse/task_queue/ddl, тогда для получения всех его детей, необходимо запросить ключи с префиксом /4/clickhouse/task_queue/ddl/.

Другое важное свойство ZooKeeper – это sequential znode – это узлы, в путь к которым дописывается последовательное значение. Для того что бы это стало возможным в Etcd, создадим для каждого ключа дополнительный ключ со значением sequential number дочерних узлов. Тогда создание узла разобьется на две части: предварительное получение sequential number и создание ключа с добавленным в конец значением. Для того что бы различать тип вершины, будем дописывать ключевое слово в начало ключа. Тогда пара с данными превратится в /value/3/clickhouse/task_queue/ddl, а пара со счетчиком в /sequential/3/clickhouse/task_queue/ddl. Аналогично поступим со всеми используемыми метаданными.

Еще одно важное свойство ZooKeeper - ephimeral znode. Для их реализации достаточно воспользоваться сервисом Lease. lease_id так же помогает с ephemeralOwner полем статистики. Это поле используется для определения владельца блокировки.

MultiRequest или же транзакция. Она поддерживает 4 вида запросов: Create, Remove, Set, Check. Для реализации транзакций в Etcd воспользуемся TxnRequest, который состоит из множества Compare, а также из множества Range, Put и DeleteRange запросов. Как уже было сказанно выше, важным свойством MultiRequest-а является атомарность, тоесть в результате его выполнения, применятся либо все запросы, либо не один. А значит для того, чтобы TxnRequest работал аналогично, необходимо для каждого из запросов добавить Compares проверяющие состояние хранилище. Но так как результат сравнения будет общим для всех Compare запросов, необходимо все данные, формирующие ответ не только проверять с помощью Compare, но и запрашивать с помощью RangeRequest-ов, для

дальнейшего ручного формирования ответа, независимо от результата общего Compare. Вернемся к этому подробнее при обсуждении составляющих каждого из запросов. Так же по причине подавляющего большинства запросов, состоящих из более чем одно операции, будем любой запрос оборачивать в TxnRequest.

Как уже было сказано выше, для некоторых типов запросов необходимо делать несколько вызовов, например что бы получить sequential_number или проверить что у узла нет детей перед тем как его удалить (невозможно поддерживать актуальное количество детей в отдельной вершине, так как при удалении эфимерного узла из-за отвалившегося хоста кластера, количество детей и метаданные об этом, изменятся не атомарно). Тогда все запросы разделятся на составные (состоящие из нескольких запросов) и простые. Тогда выполнение каждого запроса разбивается на следующие шаги: проверить является ли запрос составным, если да, то приготовить и вызвать предварительный запрос, дождаться результата, если результат противоречит ожиданиям (например, у удаляемого узла есть дети), вернуть ответ с ошибкой, иначе готовим и вызываем основной запрос.

В случае если запрос неудачен, по причине изменившихся данных, полученных в результате предварительного запроса (например, создалась еще одна sequential вершина и обновился sequential number), ответ помечается как незавершенный и запрос добавляется в очередь снова.

Как уже было сказано выше, результат Compares общий, а значит при хотя бы двух сравнениях, мы не можем определить какой из них неуспешный, для того чтобы однозначно это установить, будем использовать Compare только для предотвращения ложного изменения состояния хранилища. То есть везде, где это возможно будем использовать rangeRequest, и затем сравнивать результат вручную, и добавлять Compare только в транзакции изменяющие состояние хранилища, так же для всех Compare запросов о которых будет сказано ниже по умолчанию будем

добавлять RangeRequest-ы для их ключей в failure_request_ops, и в случае failure результата транзакции обрабатывать результат вручную.

Введем обозначения вспомогательных путей для вершинны и ее метаданных. Пусть у нас есть etcd_key, тогда у него есть следующие вспомогательные пути:

- FullEtcdKey – путь, который будет записан в Etcd (вместе с префиксом и уровнем)

- CtimeKey – метаданные ctime

- EphemeralFlagKey – флаг, который показывает эфимерный ли этот узел

- SequentialCounterKey – флаг, содержащий sequential counter для дочерних узлов

- ChildrenPrefixKey – префикс дочерних узлов

- ChildrenFlagKey - узел-флаг детей, меняющий свое состояние, каждый раз, когда в ассоциированную с ним директорию добавляют новый узел

- ParentEtcdKey – родительский узел

- ParentEphemeralFlagKey – флаг, показывающий, является ли родительский узел эфимерным

- ParentSequentialCounterKey – последовательный счетчик ассоциированный с родительским узлом

- ParentChildrenFlagKey – родительский узел-флаг детей…

Обозначим существующие ошибки ответов:

- ZOK – успешно

- ZNONODE – самого узла или его родителя не существует

- ZBADVERSION – некорректная версия

- ZNOCHILDRENFOREPHEMERAL – нельзя создавать потомка эфимерного узла

- ZNODEEXISTS – узел уже существует

- ZNOTEMPTY – узел не пустой (есть потомки)

Рассмотрим ожидаемое поведение для каждого из запросов

- create - ZNODEEXISTS – если узел уже существует, ZNONODE – если родительского узла не существует, ZNOCHILDRENFOREPHEMERAL – если родительский узел эфимерны, иначе ZOK и создание узла

- remove - ZNONODE – если узла не существует, ZNOTEMPTY – если узел не пустой, ZBADVERSION – передана некорректная версия, иначе ZOK и удаление узла.

- exists - ZNONODE – если узла не существует, иначе ZOK и метаданные узла.

- get - ZNONODE – если узла не существует, иначе ZOK, data метаданные узла.

- set - ZNONODE – если узла не существует, ZBADVERSION – передана некорректная версия, иначе ZOK, обновление data и метаданные узла.

- list - ZNONODE – если узла не существует, иначе ZOK, vector детей и метаданные узла.

- check - ZNONODE – если узла не существует, ZBADVERSION – передана некорректная версия, иначе ZOK и удаление узла.

- multi – по описанным в разделе “О ZooKeeper” правилам.

Рассмотрим формирование каждого из запросов для Etcd:

- create – перед тем как создать вершину, необходимо проверить что родительский узел существует (rangeRequest(ParentEtcdKey)), он не является эфимерным, так как у эфимерного узла не может быть потомков

(rangeRequest(ParentEphemeralFlagKey)), создаваемого узла не существует (rangeRequest(FullEtcdKey)), а также, если создаваемый путь is_sequential, запросить sequential counter (rangeRequest(ParentSequentialCounterKey)). А значит, если создаваемый узел не корневой или последовательный, запрос является составным и описанные выше запросы – это предварительная транзакция. Из полученных данных мы можем выявить ошибки ZNOCHILDRENFOREPHEMERAL и ZNONODE, если родительский узел эфимерный или его не существует соответственно. В случае успешной проверки результатов предварительного запроса, формируется основной запрос, который состоит из создания всех ассоциированных с вершиной ключей, а так же проверки на существование родительского узла (compare(ParentEtcdKey, version, not_equal, -1)) так как за время получения предварительного результата, его могли удалить, обновления данных родительского узла-флга детей (preparePut(ParentChildrenFlagKey, FullEtcdKey)), а так же, если узел последовательный, сравнения и обновления значения счетчика (compare(ParentSequentialCounterKey, seq_num), preparePut(ParentSequentialCounterKey, seq_num + 1))

- remove – перед тем как создавать вершину, необходимо проверить что она существует (rangeRequest(FullEtcdKey)) и у нее нет детей (rangeRequestWithPrefix(ChildrenPrefixKey)), а так же запросить версию узла-флага детей (rangeRequest(ChildrenFlagKey)). Из полученнх данных выявляем ошибки ZNONODE, ZNOTEMPTY и запоминаем версию ChildrenFlagKey. В случае успешной проверки, добавляем проверку версии узла-флага детей (compare(ChildrenFlagKey, version, equal, children_flag_version)), проверяем данную версию, на корректность (compare(FullEtcdKey, version, equal, version)) и удаляем все связанные с узлом ключи.

- exists – так как цель exists запроса – это получить ответ, существует ли вершина, а так же получить метаданные вершины, запросим все это за один запрос, который состоит из запроса самой вершины, что проверяет ее существование (rangeRequest(FullEtcdKey)), ctime rangeRequest(CtimeKey), а так же из запроса детей, что бы заполнить метаданные numChildren (rangeRequestWithPrefix(ChildrenPrefixKey)).

- get – тот же exists, только в ответ записывается еще и data.

- set – для того что бы установить новое значение, необходимо сравнить версию узла с данной (compare(FullEtcdKey, version, equal, version)), что так же ответит на вопрос, существует ли узел. Установить новое значение (putRequest(FullEtcdKey, data)), и так как SetResponse содержит метаданные узла необходимо их запросить rangeRequestWithPrefix(ChildrenPrefixKey), rangeRequest(CtimeKey).

- list – тот же exists запрос, только необходимо дополнительно сформировать vector детей.

- check – запрос состоит из единственной проверки версии узла (compare(FullEtcdKey, version, equal, version))

- multi – так как multi request – это всего лишь набор запросов, то для того, чтобы сформировать предварительный запрос, просуммируем предварительные запросы каждого. Если найдется хотя бы один запрос, multiRequest составной и требует предварительного вызова, в таком случае вызываем предварительный запрос, обрабатываем результат путем его передачи в обработку для каждого из подзапросов. После чего суммируем основные запросы каждого из подзапросов и вызываем. Снова обрабатываем путем отправки результата в обработку для каждого подзапроса.

Рассмотрим дополнительные детали Multi запроса.

- Как было описано выше, цель MultiRequest-а это найти ранний из запросов который не может примениться, а значит мы не можем полагаться на ошибку при обработке предварительного запроса, так как она может оказаться не первой по порядку. А значит если мы нашли ошибку в каком-либо запросе после обработки результатов предварительного вызова, мы должны, намеренно добавить невозможный Compare, сформировать основной запрос из подзапросов, которые не отсеялись после обработки предварительного результата, и уже после обработки основного запроса найти первую не применённую операцию. Исходя из описанного выше, необходимо модифицировать все запросы так, чтобы корректный результат даже в случае неуспешного compare. Для этого как уже было сказано выше, добавим rangeRequest для каждого ключа из compare.

- TxnRequest облагается правилом, что ключи не могут дублироваться, а значит при создании нескольких последовательных узлов в одной директории Putrequest-ы с обновленными seq counter-ами будут дублироваться. Модифицируем для этого create request, введем переменную seq_delta, и в одном create запросе будем создавать seq_delta последовательных ключей.

Используемые в ClickHouse особенности ZooKeeper. В целях дедубликации вставляемых блоков, создается запрос, который представляет из себя MultiRequest: create blockA, remove blockA, create blockA. Делается это для того, чтобы в случае если blockA уже существует в хранилище, транзакция отменилась с ошибкой ZNODEEXISTS. Так как TxnRequest не может содержать дублирующиеся ключи, это невозможно. Поэтому все такие проверки заменены на RangeRequest в предварительном запросе, а соответствующие им create-remove request-ы помечены как ложные.

6. Реализация

Так как код ClickHouse написан на языке C++, а также все составляющие принято разрабатывать как часть кода, а не отдельные подсистемы, был реализован класс EtcdKeeper, на основе интерфейса IKeeper.

Класс содержит в себе 5 потоков, первый из них, создает запросы и добавляет в общую очередь, второй поток достает их из очереди, присваивает каждому запросу уникальный xid, добавляет запрос в map, формирует и отправляет TxnRequest-ы связывая их с ComletionQueue для KV сервиса (KV_cq), при наличии watch-а в запросе, так же формируется и отправляется WatchRequest связываясь с ComletionQueue для Watch сервиса (Watch_cq), а так же отправляет KeepAlive запросы. KV_cq блокирует третий поток вызовом Next до получения очередного ответа, если получен ответ на предварительный запрос, проверяется его статус, в случае если код ошибки отличен от ZOK. Основной ответ не формируется, а callback вызывается с текущим. Если код ошибки ZOK, запрос снова добавляется в очередь запросов. Если получен основной запрос и он не помечен как finished, он очищается и снова добавляется в очередь. Иначе вызывается callback. Четвертый поток блокируется Watch_cq. И последний поток блокируется Lease_cq. Так как в этом коде происходит работа из разных потоков, все обращения к потоконебезопасным структурам обернуты в mutex-ы.

7. Результат

Здесь однозначно должны быть замеры скорости ZooKeeper против EtcdKeeper, но описанная выше схема не проходит 10 тестов из 68, из-за чего схема признана не совершенной, и находится на доработке.

8. Заключение

Описанный выше подход и полученный результат дает надежду что ZooKeeper возможно заменить на Etcd.

9. Источники

1. PR[online] // GitHub 2020 URL: https://github.com/ClickHouse/ClickHouse/pull/8435
2. PR[online] // GitHub 2020 URL: https://github.com/ClickHouse/ClickHouse/pull/10376
3. ClickHouse [online] // https://clickhouse.tech/docs/en/
4. ZooKeeper [online] // https://zookeeper.apache.org/
5. Etcd [online] // https://etcd.io/docs/v3.4.0/
6. Zookeeper [online] // https://static.usenix.org/event/atc10/tech/full_papers/Hunt.pdf
7. gRPC [online] // https://grpc.github.io/