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
Программный проект на тему
Реализация в ClickHouse протокола PostgreSQL

Выполнил студент группы 161, 4 курса,
Элбакян Мовсес Андраникович

Руководитель ВКР:
приглашенный преподаватель, Миловидов Алексей Николаевич

Оглавление

1. Аннотация

Базы данных являются одними из важнейших частей современных программ. На данный момент в мире существует много баз данных, предназначенных для различных целей, однако одна из самых распространенных и высокоуровневых классификаций – это SQL/NoSQL базы данных. SQL базы данных (каковыми являются ClickHouse и PostgreSQL, рассматриваемые в данной работе) имеют довольно похожий синтаксис запросов. Теоретически, можно было бы предположить, что клиент одной базы данных мог бы обращаться к базе другого типа. Однако, в действительности, они несовместимы. Поэтому большинство баз данных предоставляют спецификации так называемых wire protocol, которые могут быть реализованы на сервере одной базы данных, чтобы принимать соединения от клиентов другой, и наоборот. Данная работа рассматривает имплементацию PostgreSQL wire protocol для серверов в аналитической базе данных Yandex ClickHouse.

1. Abstract

Databases are an essential part of most modern programs. There are many types of databases for different purposes, but one of the most common high-level classifications is SQL/NoSQL. SQL databases (ClickHouse and PostgreSQL are some of those) have similar syntax for making queries. Theoretically, a client of the one database implementation can be used to make queries to a server of another. In practice, they are not compatible. That is why most databases provide specifications of wire protocols, which can be implemented on the server-side to support queries from another database client and vice-versa. This work provides an overview of PostgreSQL database server-side wire protocol implementation in Yandex ClickHouse analytical database.

2. Обзор литературы

Специфика данной работы заключается в том, что стоит четкая цель и уже в самом начале работы описано «методологическое пособие по выполнению», а

именно PostgreSQL wire protocol[1] однозначно диктует поведение программы в тех или иных ситуациях. Единственное, чего не специфицирует протокол так это то, каким образом это должно быть имплементировано с точки зрения дизайна архитектуры. Таким образом, исследование может пойти двумя путями: искать реализации протокола в других базах данных и утилитах, или же посмотреть аналогичные решения уже внутри ClickHouse.

Первый подход довольно очевидный, к тому же есть явное преимущество в том, что можно посмотреть какие-то части и перенять опыт. Однако, все это преимущество нивелируется тем, что стек технологий, язык программирования отличаются, и то, что хорошо в одном месте, в ClickHouse это уже может не работать. К тому же, в действительности не так уж и много мест, где можно посмотреть на имплементацию серверной части протокола. По своей сути – только в самой PostgreSQL. Но по причинам, названным выше, исследование сильно бы затянулось и было бы малоэффективным.

Подход с исследованием аналогичных решений в ClickHouse по очевидным причинам более эффективен: возможность переиспользовать код, более подходящий стек технологий и общая совместимость с архитектурой базы данных. Однако, единственным препятствием могло бы стать, что из аналогов окажется только сам нативный протокол. Нативный протокол в этом смысле плох тем, что он слишком сильно заточен под конкретную базу данных и некоторые решения могут уходить глубоко в архитектуру. Но в прошлом году был реализован MySQL wire protocol[2], который также, как и PostgreSQL wire protocol, является инородным по отношению ClickHouse. При более подробном изучении, оказалось, что на абстрактном уровне они имели немало сходных черт, поэтому при имплементации протокола PostgreSQL были приложены усилия к тому, чтобы сделать некоторые абстракции, которые могли бы быть при добавлении какого-либо еще протокола обобщены. Переиспользовать код, к сожалению, особо не получилось, так как различия тем не менее были

[1] PostgreSQL: Documentation: Chapter 52. Frontend/Backend Protocol [Электронный ресурс] /. — Электрон. текстовые дан. — 2020: Режим доступа: https://www.postgresql.org/docs/current/protocol.html, свободный
[2] [WIP] Implementation of MySQL wire protocol [Электронный ресурс] /. — Электрон. журн. — 2019. — Режим доступа: https://github.com/ClickHouse/ClickHouse/pull/4715, свободный

существенные.

3. Список ключевых слов

ClickHouse, wire protocol, PostgreSQL.

4. Введение

Современные базы данных в классическом случае представляют из себя в действительности две сущности: клиент и сервер. Вся логика хранения, организация доступов и механизмы манипуляции данных находятся на стороне сервера. Клиент же используется для совершения запросов непосредственно к базе данных. PostgreSQL и ClickHouse, рассматриваемые в данной работе устроены таким же образом.

Стоит отметить, что если серверы существуют (как кодовая база) в единственном виде (по модулю выпускаемых версий), то клиентов может быть большое количество под разные платформы и языки программирования. Чтобы обеспечить простую и прозрачную разработку новых клиентов создается специальный протокол общения между клиентом и сервером – *wire protocol*, которому обе стороны строго следуют при создании соединения и дальнейшем обмене данными.

Может показаться, что wire protocol нужен только для создания новых клиентов для базы данных. Однако, на самом деле можно использовать его ровно наоборот – сделать другую базу данных совместимой с клиентами данной. Настоящая работа как раз рассматривает такой пример: сделать сервер ClickHouse совместимым с клиентами PostgreSQL.

Обосновать внедрение такой технологии в ClickHouse можно следующими факторами: во-первых, для разработчиков приложений это уменьшает стек технологий, что позволяет легче оперировать сущностями внутри проекта. Например, если уже есть основная база данных в виде PostgreSQL, то подключение ClickHouse для аналитических целей не потребует внедрения еще одной библиотеки для работы с ним. Во-вторых, это влияет на популяризацию

ClickHouse. PostgreSQL является одной из самых популярных SQL баз данных[3] и занимает 4 место, ClickHouse же занимает только 71 место. Поддержка wire protocol'а позволит многим разработчикам рассмотреть возможность использования ClickHouse в своих приложениях. В-третьих, результаты данной работы могут служить примером для исследований унификации API между разными базами данных, что является актуальной задачей с момента появления различных баз данных.

Таким образом, перед работой были поставлены следующие цели и задачи:

Цель: *имплементация wire protocol'а PostgreSQL в аналитической базе данных ClickHouse.*

Задачи:

1. Сделать базу данных ClickHouse максимально совместимой с как можно большим количеством PostgreSQL клиентов.
2. Добиться как можно более оптимизированного кода и избежать просадки скорости в области подготовки запроса и отправки результатов.
3. Реализация интеграционного тестирования для избегания будущих багов при изменении кодовой базы.

В ходе работы был имплементирован код, позволяющий взаимодействовать. базой данных ClickHouse с помощью PostgreSQL клиентов, были имплементированы различные методы аутентификации и поддержка TLS соединений для защищенной работы с базой данных. Также были сделаны интеграционные тесты с использованием популярных клиентов PostgreSQL. Помимо этого, было выявлено, что архитектура ClickHouse не позволяет имплементировать все возможности, описанные в спецификации PostgreSQL wire protocol. Более того, некоторые части протокола завязаны на синтаксис PostgreSQL, который является не совместимым с тем, что реализован в ClickHouse.

Далее в работе будет дано более детальное описание протокола вместе с

[3] DB-Engines Ranking [Электронный ресурс] /. — Электрон. текстовые дан. — 2020: Режим доступа: https://db-engines.com/en/ranking, свободный

деталями имплементации. Также будет приведен стек технологий, использованных в ходе написания решения. Помимо этого, будет произведен анализ совместимости ClickHouse с PostgreSQL wire protocol, возникших проблем при имплементации и методов их решения.

5. Выбор методов решения

5.1. Совместимость ClickHouse с протоколом PostgreSQL

Для имплементации совместимого с PostgreSQL сервера будет использована спецификация wire protocol, описанная самими создателями PostgreSQL. Более детальное описание спецификации совместно с реализацией будет дано в следующей главе, здесь же рассмотрим совместимость протокола с возможностями ClickHouse.

Рассмотрение совместимости стоит начать с методов шифрования трафика. PostgreSQL поддерживает обмен сообщениями по TCP без шифрования, с SSL(TLS) шифрованием и GSSAPI шифрованием. ClickHouse поддерживает только первые два, поэтому они и будут использованы. Добавление GSSAPI возможно в будущем, если в ClickHouse будет добавлен такой метод соединения, сейчас же это не является ни необходимой, ни приоритетной задачей.

Далее изучим методы аутентификации. PostgreSQL поддерживает аутентификацию без пароля, с обычным паролем, аутентификацию по схеме SCRAM-SHA-256, SCRAM-SHA-256-PLUS, KerberosV5 (является устаревшей), аутентификация с MD5 хешированием с солью и аутентификация через GSSAPI. ClickHouse же в свою очередь поддерживает первые четыре метода, а также еще один специфичный (DOUBLE-SHA1), используемый для MySQL wire protocol. Также было решено добавить поддержку аутентификация через MD5, так как она нередко встречается при использовании PostgreSQL. Более детальное описание методов аутентификации будет приведено в главе с описанием решения.

PostgreSQL поддерживает несколько видов запросов (по классификации внутри wire protocol’a):

1. Simple query. Обычные транзакционные запросы к данным.

2. Extended query. Разбивает simple query в несколько этапов, позволяя также создавать подготавливаемые запросы с подстановкой параметров. Преимущество такого подхода заключается в том, что таким образом можно избежать SQL инъекций внутри параметров запроса. Также параметризованные запросы полезны в случае создания некоторого REST API приложения, внутри которых по HTTP можно обращаться в end point, передавая внутри запроса некоторые параметры, которые уже дальше вместе с запросом будут переданы в базу данных для обработки.
3. Function call. Позволяет вызвать некоторую функцию из каталога pg_proc, является устаревшим по протоколу. В качестве альтернативы протокол предлагает делать это через extended query.
4. Copy query. Копирование данных от одной стороны к другой. Это чисто бинарный протокол, который отправляет требуемые данные в нужном формате в виде бинарных кусков.

ClickHouse из этого списка явно не поддерживает function call, так как он заточен под специальный Postgres каталог, поэтому данная возможность не будет имплементирована.

Copy query также решено было не имплементировать, так как в нем есть завязка на специфичный для PostgreSQL синтаксис. Более точно, было принято решение, что можно имплементировать специальный парсер таких запросов внутри ClickHouse, если это будет нужно пользователям. Однако востребованность данной возможности не так очевидна по модулю того, что это предполагается использовать вместе с ClickHouse, более того это интуитивно неочевидно использовать синтаксис специфичный для PostgreSQL при запросах в ClickHouse.

Extended query также не была реализована, однако уже в виду архитектурной проблемы. Данная проблема вскрылась непосредственно в ходе имплементации данного вида запросов. Короткое объяснение заключается в том, что на одном из этапов сервер должен вернуть информацию о выходных параметрах, однако в виду особенности устройства ClickHouse это было

невозможно сделать. Более подробная информация будет дана в разделе с описанием extended query.

Теперь рассмотрим инструменты, которые используются при разработке протокола.

5.2. Используемый стек технологий

Реализация самого протокола происходит на языке C++ 17, который является основным для разработки ClickHouse (в действительности ClickHouse поддерживает и спецификацию 20 версии, однако в виду некоторых ограничений с проприетарным использованием внутри компании-разработчика данной базы данных, использование данной версии пока не является возможным).

В качестве системы сборки используется широко используемая в мире программа CMake. Взаимодействие с кодовой базой происходит через систему контроля версий Git, а непосредственно новый код вливается через pull requests в системе GitHub. Работа с сетью, шифрованием и различными алгоритмами хеширования выполняется через библиотеку с открытым исходным кодом Poco.

Также в ходе разработки зачастую приходится посмотреть какие именно сообщения отправляются по сети между клиентом и сервером, для этого используется утилита tcpflow, которая позволяет слушать определенный порт и читать пакеты, отправляемые через него, причем как в формате hexdump, так и в ASCII. Однако в случае с базой данных PostgreSQL полезно использовать утилиту socat, которая умеет передавать данные из сокета и в него. Более подробно это нужно в виду того, что для локальных соединений PostgreSQL использует не tcp, а сокеты (хотя в целом это конфигурируется). Для того, чтобы перехватывать данные в таком соединении можно использовать следующие команды:

```
mv .s.PGSQL.5432 .s.PGSQL.5433.original
sudo socat TCP-LISTEN:8089,reuseaddr,fork UNIX-CONNECT:.s.PGSQL.5433.original
sudo socat UNIX-LISTEN:.s.PGSQL.5432,fork TCP-CONNECT:127.0.0.1:8089
sudo tcpflow -p -i lo0 -cD port 8089
```

Данный код делает следующее: переименовывает исходный сокет, созданный PostgreSQL в новый, далее с помощью socat перенаправляет все TCP пакеты с

порта 8089 в переименованный сокет, затем опять же с помошью socat открывает новый сокет, данные из которого перенаправляет по адресу 127.0.0.1:8089. Затем мы с помощью tcpflow подключаемся к порту 8089 и перехватываем все пакеты.

Конечно, нельзя было бы вести разработку без различных PostgreSQL клиентов. На начальных стадиях использовалась утилита psql, которая дистрибуцируется самими разработчиками PostgreSQL вместе с базой данных. Также будут использованы клиенты для Python3 и Java, которые нужны для написания интеграционных тестов и проверки работоспособности протокола.

6. Обзор решения задачи

В этом разделе будет рассмотрено устройство протокола вместе с деталями реализации в ClickHouse. Здесь также будут приведены некоторые части протокола, которые не попали в конечную реализацию с детальным объяснением проблем, воспрепятствовавших этому.

6.1. Прием соединений и виды сообщениий

В первую очередь нужно сделать так, чтобы ClickHouse мог принимать соединения по TCP. Здесь уже есть внутри проекта устоявшаяся практика в виде наследования от Poco::Net::TCPServerConnectionFactory для создания Poco::Net::TCPServerConnection, который уже в свою очередь умеет принимать соединения. TCPServerConnection представляет из себя абстрактный класс, в котором требуется реализовать метод `run`, внутри которого будет происходить вся логика обработки запросов. TCPServerConnectionFactory передает внутрь TCPServerConnection сокет, который используется для обмена сообщениями. На основе переданного сокета строятся буферы для чтения и записи. Также стоит тут отметить, что соединения могут быть в защищенном режиме. Для этого будет дополнительно создан Poco::Net::SecureStreamSocket поверх основного сокета, этот момент еще будет затронут далее.

Для передачи данных протокол оперирует понятием сообщений, которые бывают трех типов: frontend (F), backend (B), frontend & backend (FB). Каждый каждое сообщение по своей сущности это просто набор байт, причем любое

сообщение содержит в себе его размер. Сообщение содержит в себе поля, строго специфицированные протоколом. Поля бывают нескольких типов:

- Int**n**(***i***) (целое число с битностью n и если указано i, то число в сообщении строго равно i, используется сетевой порядок байт)
- Int**n**[***k***] (массив размером k целых чисел с битностью n, используется сетевой порядок байт)
- String(***s***) (null-terminated строка, если в спецификации сообщения указано s, то строка всегда равна s)
- Byte**n**(**c**) (n байт, если в спецификации сообщения указано c, то набор байт в сообщении всегда равен c).

Также протокол дает сообщениям типы (представлен в виде одного байта), они не уникальны для каждого сообщения, а скорее показывают к какому идейному типу они относятся. Благодаря им можно понять какие действия сейчас должен предпринять сервер. Однако здесь есть исключение, по историческим причинам самое первое сообщение, которое отправляет клиент не имеет типа, однако протокол указывает способ как правильно классифицировать первое сообщение. Стоит еще отметить, что для удобства буферизации и валидации в каждом сообщении после байта с типом дается размер сообщения в байтах. В действительности чтобы распарсить это сообщение не требуется знание о размере сообщения, сам формат каждого сообщения указывает способ парсинга.

Для работы с сообщениями в коде были созданы следующие абстрактные классы:

- IMessage, у которого есть метод getMessageType, возвращающий enum с типом сообщения (этот тип точно определяет сообщение, предоставляя его уникальный идентификатор, и является внутренней особенностью имплементации, не стоит путать с типом, которым оперирует сам протокол).
- ISerializable с методами serialize и size. serialize принимает WriteBuffer, в который будет сериализован объект. Метод size возвращает размер отправляемого сообщения (более точно, это значение будет записано в

специальное поле, которое должно быть у любого сообщения по протоколу).

- FrontMessage. Наследуется от IMessage и добавлет метод deserialiaze, который принимает ReadBuffer, из которого будет читаться поток байт и интерпретироваться в объект согласно протоколу. Важно уточнить, что deserialize не должен считывать тип сообщения.
- BackendMessage. Наследуется от IMessage и ISerializable.
- FirstMessage. Наследуется от FrontMessage, удаляет конструктор по умолчанию и добавляет обязательное поле payload_size. Это сделано в виду специфики чтения самого первого сообщения, так как классифицировать тип можно только прочитав первые два поля, причем размер сообщения здесь идет первым.

Для более удобной работы с чтением и записью сообщений был создан класс MessageTransport с шаблонными методами, принимающий в себя WriteBuffer и ReadBuffer или же просто WriteBuffer (конструктор только с WriteBuffer сделан для специального кейса, когда ReadBuffer неизвестен, но требуется отправлять сообщения, ниже будет приведен пример использования), которые в свою очередь создаются в PostgreSQLHandler (наследник TCPServerConnection). Данный класс имеет следующие методы:

- std::unique_ptr<TMessage> receiveWithPayloadSize(), который предназначен для получения сообщений-наследников FirstMessage
- std::unique_ptr<TMessage> receive(), через который происходят чтения обычных сообщений
- FrontMessageType receiveMessageType(), который позволяет прочитать тип сообщения (по протоколу), в имплементации это представлено в виде enum FrontMessageType
- void send(TMessage & message, bool flush=false), send(TMessage && message, bool flush=false), send(char message, bool flush=false) – три метода для отправки сообщений, первые два отправляют полноценные классы-сообщения, последний используется для отправки одного байта. Параметр flush позволяет сбросить буфер и отправить сообщения клиенту.

- void flush(), который позволяет сбросить леажщиие в выходном буфере данные
- void dropMessage(), который позволяет пропустить текущее сообщение (внутри просто происходит увеличение переменной смещения позиции в буфере, размер данных можно узнать по передаваемому в каждом сообщении первым полем длину сообщения)

Стоит отметить, что методы receiveWithPayloadSize и receive внутри вызывают метод deserialize у TMessage, а первые два метода send вызывают serialize у переданного TMessage.

6.2. Установление контакта между клиентом и сервером

6.2.1. Обработка первого сообщения и SSL

Далее опишем установление соединения между клиентом и сервером. Соединение инициирует клиент. Первым сообщением может оказаться StartUpMessage, SSLRequest или CancelRequest (еще может быть GSSEncRequest, но он не был поддержан, поэтому не будет упоминаться далее). CancelRequest будет рассмотрен в отдельном разделе ниже.

SSLRequest сообщает о запросе шифрования соединения по SSL. Данный метод соединения позволяет установить защищенное соединение в местах, где может быть проведена атака MITM (Man In The Middle). Если сервер не поддерживает соединение (например, сервер был собран без поддержки SSL), то он может отправить отказ клиенту – байт 'N', иначе отправляется байт 'S'. Если клиента не устроил ответ, то он может тут же закрыть соединение. В противном случае далее между ними происходит SSL handshake. В плане кода после отправки согласия на защищенное соединение достаточно вызвать Poco::Net::SecureStreamSocket::attach поверх текущего сокета, а также пересоздать ReadBuffer, WriteBuffer и MessageTransport. Далее по протоколу происходит отправка StartUpMessage клиентом. В любой момент общения сервер в случае каких-либо ошибок может отправить сообщение ErrorResponse, обозначающее, что на сервере произошла ошибка, внутри сообщения содержатся данные о ней.

StartUpMessage содержит в себе версию протокола, а также последовательность параметров в виде пар имя-значение. Среди них обязательно содержится имя пользователя, также опционально имя базы данных, к которой производится подключение. Остальные параметры передаются для дополнительного использования сервером. Набор параметров в данном случае ограниченный Следующим этапом происходит аутентификация пользователя.

6.2.2. Аутентификация

Аутентификация по протоколу происходит следующим образом: на основе переданного имени пользователя данных сервер выбирает метод аутентификации и отправляет соответствующий запрос (если таковая вообще нужна). Если клиент не поддерживает данный метод, то он должен закрыть соединение. Иначе происходит процесс обмена сообщениями для аутентификации (зависит от конкретного метода аутентификации).

С точки зрения имплементации для удобства был сделан класс AuthenticationManager, который создается в конструкторе PostgreSQLHandler. Ему из factory передается список классов-наследников абстрактного класса AuthenticationMethod. Далее он внутри себя создает словарь из Authentication::Type в AuthenticationMethod (соответствующий тип может быть получен непосредственно из AuthenticationMethod). После получения имени пользователя вызывается метод authenticate у AuthenticationManager, который внутри себя получает из внутренней базы пользователей ClickHouse какой метод аутентификации должен быть применен для данного пользователя, и если он поддержан (есть в списке при инициализации), то вызывается метод authenticate у требуемого AuthenticationMethod.

Как было описано выше, в рамках реализации протокола будет имплементировано четыре метода аутентификации: без пароля, обычный незашифрованный пароль, MD5 с солью и SCRAM-256. Опишем каждый немного подробнее.

Самый простой и наиболее незащищенный метод аутентификации – это аутентификация без пароля. Если пользователю дан доступ без пароля, то сервер

сразу же отправляет AuthenticationOk, после чего протокол переходит к следующей стадии. Стоит отметить, что если злоумышленник знает логин такого привилегированного пользователя и адрес базы данных, то он получает доступ ко всем таблицам, к которым имел данный пользователь.

Чуть более защищенный метод – аутентификация с паролем. Сервер посылает сообщение AuthenticationCleartextPassword, на которое клиент должен ответить сообщением PasswordMessage с паролем для данного пользователя. Если пароль оказался корректным, то сервер отправляет AuthenticationOk и обе стороны переходят к следующему этапу протокола. Иначе сервер отвечает сообщением ErrorResponse. Несмотря на то, что при таком методе уже нельзя злоумышленнику напрямую получить доступ к данным, однако если он имеет возможность прослушивать сеть между пользователем и базой данных, то он легко может перехватить сообщение с паролем.

Более защищенным методом является MD5 с солью. Это представитель так называемого семейства алгоритмов аутентификации “Challenge-response authentication”. Суть таких алгоритмов заключается в том, что для совершения аутентификации обе стороны должны произвести обмен сообщениями, специфицируемыми конкретным алгоритмом.[4] В простейшем случае обычный пароль тоже можно отнести к такому семейству, однако более интересны примеры, где между сторонами происходят некоторые вычисления, которые не используют явно пароль. Идея алгоритма MD5 с солью заключается в том, что после получения имени пользователя, сервер отправялет сообщение AuthenticationMD5Password, в котором содержится сгенерированная случайная соль размером четыре байта. Далее клиент, используя эту соль, вычисляет ответ по формуле ('md5', md5(concat(md5(concat(password, username)), random-salt))). Ответ отправляется серверу, который также вычисляет по этой формуле ответ и сравнивает их. Если они совпали, то отправялется AuthenticationOk, иначе ErrorResponse. Отдельно стоит отметить, что соль генерируется каждый раз разная, чтобы злоумышленник не смог переиспользовать ответ. Также соль

[4] What is Challenge-Response Authentication [Электронный ресурс] /. — Электрон. журн. — Режим доступа: https://www.techopedia.com/definition/26138/challenge-response-authentication, свободный

усложняет перебор и взлом пароля в оффлайне (если вдруг злоумышленнику понадобился исходный пароль).

Самым защищенным алгоритмом является SCRAM-SHA256, который описан в RFC 7677[5]. Далее приведем идею алгоритма. В процессе аутентификации клиент и сервер обмениваются четырьмя сообщениями:

1. client-first. Клиент отправляет в нем специальный заголовок, имя пользователя и случайно сгенерированную последовательность символов (cnonce).
2. server-first. Сервер конкатенирует к отправленному cnonce собственную последовательность snonce, а также добавляет соль, использованную сервером для вычисления хеша пароля пользователя, а также количество итераций для алгоритма хеширования.
3. client-final. Клиент отправляет сообщение, в котором содержится параметр c-bind-input (используется для вариации аутентификации SCRAM-SHA256-PLUS), а также конкатенацию cnonce, snonce и cproof.
4. server-final. Сервер отправляет последнее сообщение с подтверждением sproof. Обе стороны должны проверить совпадение данных.

Для вычисления cproof и sproof определим хеш пароля с солью spassword = PBKDF2(HMAC, p, s, i, d), где PBKDF2 – это функция получения ключа, разработанная RSA Laboratories, используемая для получения стойких ключей на основе хеша[6], HMAC – алгоритм генерации MAC (message authentication code) на основе криптографического хеша (в данном случае SHA256) и криптографического ключа[7], p – пароль, s – соль, i – количество итераций, d – длина ключа. Таким образом формулы для вычисления

[5] RFC 7677 – SCRAM-SHA-256 and SCRAM-SHA-256-PLUS [Электронный ресурс] / T. Hansen. — Электрон. журн. — 2015. — Режим доступа: https://tools.ietf.org/html/rfc7677, свободный
[6] Стандарт PBKDF2 [Электронный ресурс] /. — Электрон. журн. — Режим доступа: https://defcon.ru/cryptography/485/, свободный
[7] RFC 2401 – HMAC: Keyed-Hashing for Message Authentication [Электронный ресурс] / Krawczyk, et. al.. — Электрон. журн. — 1997. — Режим доступа: https://tools.ietf.org/html/rfc2104, свободный

подтверждений:

```
ckey = HMAC(spassword, 'Client Key')
skey = HMAC(spassword, 'Server Key')
cproof = ckey XOR HMAC(SHA256(ckey), Auth)
sproof = HMAC(skey, Auth)
```

6.2.3. Завершение установки соединения

После того как прошла успешная аутентификация, клиент начинает ждать сигнала от сервера, а тот в свою очередь делает подготовительный сетап к приему запросов. Более точно, сервер пытается применить параметры, которые отправил клиент и отправляет их статус (клиент в свою очередь не должен как-либо отвечать на это). Также отправляется сообщение BackendKeyData, в котором содержится идентификатор процесса сервера и специальный текстовый секрет. Данные параметры нужны для отмены запросов. В данной имплементации генерируется случайное число для секрета, а в качестве идентификатора процесса используется выданный идентификатор от PostgreSQLHandlerFactory, который гарантированно уникальный для каждого процесса (используется атомарная инкрементация переменной внутри данного класса). Если никаких ошибок не произошло, то отправляется ReadyForQuery, иначе сервер отправляет ErrorResponse и закрывает соединение.

6.3. Обработка запросов

После получение ReadyForQuery клиент начинает отправлять запросы. В конкретной имплементации ReadyForQuery отправляет перед приемом запросов. Сам прием запросов имплементирован как switch case в бесконечном цикле внутри метода run у PostgreSQLHandler. Далее будут рассмотрены два основных вида запросов: Simple query, Extended query и Copy (последние два кратко).

6.3.1. Simple query

Simple query – это способ создания запроса, в котором полностью передается вся информация о нем (происходит передача запроса в текстовом виде).

Для отправки запроса клиент создает сообщение типа Query, в котором находится запрос в текстовом виде. Стоит уточнить, что запросов внутри строки может оказаться несколько, разделенных символом ';'. Также по протоколу во время обработки все запросы внутри сообщения находятся внутри неявного транзакционного блока, однако так как в ClickHouse нет поддержки транзакций, то этот факт не учитывался при имплементации.

Обработка simple query находится внутри функции processQuery, которая вызывается из нужной ветки switch case, описанной выше. Внутри запрос в первую очередь проверяется на «пустоту», то есть что в нем нет никаких символов кроме как пробельных. Делается это с помощью класса RegularExpression из библиотеки Poco. Само регулярное выражение крайне простое: \A\s*\z. Несмотря на то, что пустая строчка подходит под этот запрос, перед этим сделана проверка на пустую строчку для большей оптимальности. Если сообщение оказалось пустым, то клиенту отправляется сообщение EmptyQueryResponse и обработчик заканчивает свою работу и ждет новых сообщений от клиента.

Помимо проверки на пустоту также есть клиенто-специфичные проверки: библиотека psycopg2 для Python при открытии курсора шлет команду BEGIN, а при закрытии COMMIT. Таким образом, явно контролируются транзакции, однако такие команды не совместимы с синтаксисом ClickHouse, если они были переданы, то сразу же будет послана команда CommandComplete, обозначающая успешное завершение запроса, а затем будет произведен выход из обработчика. Похожее условие есть для Java клиента, он в качестве первого запроса отправляет “SET extra_float_digits 3”, обработка такого запроса происходит аналогично с psycopg2.

Если же запрос не прошел предыдущие проверки, то далее текст запроса парсится и делится по токену ';' на отдельные самостоятельные запросы и по очереди каждый запускается с помощью функции executeQuery, после которой отправляется сообщение CommandComplete, которое содержит тип запроса (INSERT, DELETE, UPDATE, SELECT, MOVE, FETCH, COPY) и количество строк, затронутых запросом (в случае когда возвращается результат – это

количество результирующих строк, иначе количество записей). В виду архитектурных особенностей, количество записей невозможно посчитать, поэтому было решено возвращать всегда 0. К тому же данная информация не является особенно полезной, так как количество возвращается в самом конце. Тип запроса же парсится с помощью прохода по запросу с выравниванием регистра.

Также стоит уделить внимание отправке результатов запроса. Результаты обработки запросов попадают в наследника класса IOutputFormat (в данном случае PostgreSQLOutputFormat). Конкретный формат выбирается из контекста (или же выбирается дефолтный, если он не был указан явно), формат в данном случае выставляется в PostgreSQLHandler. Наследникам IOutputFormat требуется имплементировать следующие методы:

1. doWritePrefix(). Данный метод вызывается до обработки результатов. В случае с протоколом здесь формируется и отправляется специальный заголовок с информацией о колонках, а также их типе. Более подробно создается массив из объектов типа FieldDescription, в котором есть название колонки, тип данных и код формата (текстовый или бинарный, однако у simple query формат всегда текстовый). Более подробно о типах ниже.
2. consume(Chunk). Данный метод используется непосредственно для обработки результата. В нем по очереди проходятся столбцы и формируется строчка. Это все кладется в массив с классом ISerializable (это сделано для того, чтобы можно было писать разные типы данных). Для каждой ячейки, если она не Null зовется сериализатор, который преобразует ее в текстовый формат. Если же ячейка Null, то пишется специальный класс, в котором размер поля отрицательный (это индикатор того, что поле пустое). Далее этот массив кладется в класс DataRow, который, собственно, и отправляется уже клиенту.
3. finalize(). Этот метод зовется в конце, после обработки всех данных. В нашем случае этот метод пустой. В первых вариантах здесь

писалось сообщение CompleteCommand, однако для поддержки мультизапросов это было вынесено в PostgreSQLHandler.

4. flush(). Данный метод сбрасывает все данный из буфера в выходной поток данных. В сущности просто зовется метод flush() у MessageTransport.

Опишем более подробно типизацию. Тип колонки транслируется из типов ClickHouse в PostgreSQL (полная таблица типов указана в каталоге pg_type). У PostgreSQL набор типов сильно богаче, однако не все типы из ClickHouse присутствуют там, а некоторые просто несовместимы. Поэтому для всех типов кроме небольшого набора тип выставляется varchar. Ниже представлена таблица, описывающий отображение из типов ClickHouse в типы PostgreSQL.

Таблица 6.1. Отображение из типов ClickHouse в типы PostgreSQL

Тип в ClickHouse	**Тип в PostgreSQL**
Int8	char
UInt8	int2
Int16	int2
UInt16	int4
Int32	int4
UInt32	int4
Int64	int8
Float32	float4
Float64	float8
FixedString	varchar
String	varchar
Date	date
Decimal32	numeric
Decimal64	numeric

Таблица 6.1. Отображение из типов ClickHouse в типы PostgreSQL

Decimal128	numeric
UUID	uuid
Остальные типы	varchar

6.3.2. Extended query

В этом разделе мы кратко опишем альтернативный метод приема и обработки запросов. Стоит напомнить, что данный метод не был включен в конечную имплементацию ввиду архитектурных особенностей ClickHouse, более детальные причины будут даны ниже.

Extended query разбивает Simple query на несколько этапов: Parse, Describe, Bind, Execute, Sync, Flush, Close. Опишем каждый по отдельности:

- Parse. Это первое сообщение, отправляемое клиентом в данном виде запросов. В нем содержится название запроса, сам запрос (возможно с параметрами), опционально дается информация о самих параметрах (например, тип), а также название курсора. Сервер, получив это сообщение, его обрабатывает и отвечает сообщением ParseComplete. Стоит отметить, что если название курсора или самого запроса пустые, то кэш для данного запроса держится до следующего Parse, иначе до конца сессии (если явно не сделать удаление).
- Describe. В этом сообщении клиент передает 1 байт 'S' или 'P', обозначающий statement или portal соответственно, а также название подготовленного запроса или портала. Если было передано имя портала, то возвращается RowDescription с описанием выходных полей. Если передано имя запроса, то сначала отдается сообщение типа ParameterDescription, в котором описываются типы для входных параметров, а затем отправляется RowDescription как в предыдущем примере.
- Bind. В данном запросе клиент передает уже сами параметры и указывает имя запроса и курсора, к которому их нужно применить и еще формат данных (бинарный/текстовый). На это сообщение после обработки сервер

отвечает сообщением BindComplete.

- Execute. Здесь клиент непосредственно инициирует выполнение запроса и получение результата. В параметрах сообщения присутствует название курсора, а также лимит на количество возвращаемых строк. Сервер отвечает уже готовыми результатами запроса.
- Sync. Закрывает текущий блок транзакции. Сервер посылает в ответ ReadyForQuery.
- Flush. Сервер в ответ на это сообщение должен сбросить все, что скопилось в его буфере.
- Close. По структуре сообщение совпадает с Describe. Сервер должен закрыть портал/запрос, переданный в сообщении. Сервер отвечает CloseComplete на данное сообщение.

Как уже было отмечено выше, у такого подхода есть плюсы в виде того, что можно избежать SQL инъекций с помощью валидации параметров, которые подставляются. Также в PostgreSQL сделаны специальные оптимизации, которые позволяют переиспользовать данные при повторных запросах с другими параметрами.

Теперь опишем, почему такая схема не может быть имплементирована на данный момент в базе данных ClickHouse. Дело в том, что когда клиент отправляет сообщение Describe, нужно вернуть данные о выходных параметрах. До подстановки параметров это сделать невозможно, более того, даже если попробовать подставить значения по умолчанию, это может работать не всегда. Также такой подход не может быть использован, потому что будут тратится значительные ресурсы в определенных случаях на то, чтобы это рассчитать.

6.3.3. Copy

Copy запросы применяются для копирования данных с клиента на сервер и наоборот (мы не рассматриваем здесь copy запросы только на сервере). В случае с PostgreSQL это запросы вида COPY FROM STDIN и COPY TO STDOUT. Происходят они схематично следующим образом: клиент отправляет запрос, сервер отвечает, что готов принимать/передавать данные и далее происходит трансфер сообщений с данными. Данные отправляются в бинарном формате.

Более детально с тонкостями процесса можно ознакомиться в протоколе, но так как эта функциональность не была имплементирована, то мы здесь не приводим это.

Решение не имплементировать было принято ввиду того, что для запросов клиент использует специфичный для PostgreSQL синтаксис, и чтобы это поддержать требовалось написать отдельный парсер для этого. Более того, для потенциальных пользователей данной функциональности это является неочевидным решением. В конечном итоге, данную протокол копирования будет имплементирован только по запросу пользователей.

7. Заключение

В результате работы было получено рабочее решение, позволяющее PostgreSQL клиентам взаимодействовать с аналитической базой данных ClickHouse. Решение было разработано строго согласно протоколу, поэтому большинство клиентов должны быть совместимы с ClickHouse. Также были имплементированы различные методы аутентификации и защищенные соединения, позволяющие клиентам безопасно взаимодействовать с базой данных.

Также были имплементированы интеграционные тесты для регулярной проверки корректности работы протокола, которые при неправильном изменении кода могли бы сигнализировать о неполадках. Стоит отметить, что они были крайне полезны даже в разработке и помогли выявить несколько ошибок.

Однако ввиду некоторых объективных причин, описанных детально в работе, не все возможности протокола были имплементированы. В качестве развития данной работы возможна в будущем их имплементация, хотя это и зависит во многом от общего вектора развития аналитической базы данных ClickHouse. Также в качестве улучшений может быть рассмотрена интеграция с большим количеством различных клиентов, добавление тестов для них, а также интеграция некоторых языковых возможностей из PostgreSQL, которые мешали разработке определенных возможностей протокола.

8. Библиографический список

1. PostgreSQL: Documentation: Chapter 52. Frontend/Backend Protocol [Электронный ресурс] /. — Электрон. текстовые дан. — 2020: Режим доступа: https://www.postgresql.org/docs/current/protocol.html, свободный
2. [WIP] Implementation of MySQL wire protocol [Электронный ресурс] /. — Электрон. журн. — 2019. — Режим доступа: https://github.com/ClickHouse/ClickHouse/pull/4715, свободный
3. DB-Engines Ranking [Электронный ресурс] /. — Электрон. текстовые дан. — 2020: Режим доступа: https://db-engines.com/en/ranking, свободный
4. What is Challenge-Response Authentication [Электронный ресурс] /. — Электрон. журн. — Режим доступа: https://www.techopedia.com/definition/26138/challenge-response-authentication, свободный
5. RFC 7677 – SCRAM-SHA-256 and SCRAM-SHA-256-PLUS [Электронный ресурс] / T. Hansen. — Электрон. журн. — 2015. — Режим доступа: https://tools.ietf.org/html/rfc7677, свободный
6. Стандарт PBKDF2 [Электронный ресурс] /. — Электрон. журн. — Режим доступа: https://defcon.ru/cryptography/485/, свободный
7. RFC 2401 – HMAC: Keyed-Hashing for Message Authentication [Электронный ресурс] / Krawczyk, et. al.. — Электрон. журн. — 1997. — Режим доступа: https://tools.ietf.org/html/rfc2104, свободный

9. Приложения

Имплементация была оформлена в виде pull reqest на GitHub. Ссылка на него – https://github.com/ClickHouse/ClickHouse/pull/10242.