Содержание

Аннотация

Существуют два вида баз данных (БД): SQL [13] и NoSQL [9]. ClickHouse [26] – реляционная БД (SQL), которая хранит данные по столбцам. MongoDB [21] – NoSQL БД, которая хранит и принимает данные в формате типа json [30]. У этих баз данных разная структура и поэтому принципиально разный синтаксис запросов. Для того, чтобы ClickHouse мог принимать запросы для MongoDB и отдавать их в соотвествующем виде, нужно реализовать отдельный веб-обработчик (ручку), который будет посредником в их общении.

Ключевые слова: базы данных, разработка C++, промышленная разработка.

Введение

Описание предметной области

Почти каждое приложение в современном мире использует базы данных для хранения информации. Однако база данных – это не просто приложение, где хранится какая-то информация, ведь часто над данными нужно проводить различные манипуляции: поиск, добавление, сравнение, счет и т.д.

В связи с этим появилось много разных приложений, которые используют свои методы работы с данными, где информация хранится по-разному: где-то данные хранятся в строку (MySQL [19]), где-то в столбец (ClickHouse [26]), а где-то и вовсе в виде произвольного документа вида json (MongoDB [21]).

Иногда пользователю нужно написать запрос на одном языке, но пользоваться он хочет удобствами другой базы данных, которая не поддерживает этот язык. В этом случае он пользуется специальными переводчиками запросов (Translation Layer), которые получают запросы клиента базы данных A, преобразуют его в формат базы данных B, и отправляют его на сервер B. После исполнения, B отсылает запрос обратно, который преобразуется в вывод в формате A и отправляется клиенту A.

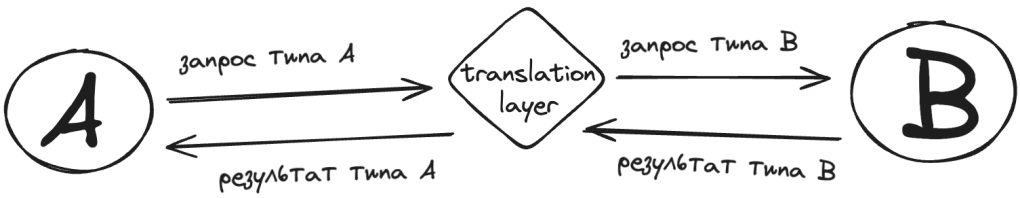

Рис. 0.1: Принцип работы Translation Layer

До начала разработки этого проекта ClickHouse не поддерживал MongoDB протокол. Проект был написан с нуля, без какого-либо существовавшего прототипа.

Постановка задачи

В данной работе передо мной была поставлена задача реализации веб-обработчика (ручки) транслятора запросов. В данном случае ручка должна принимать запросы типа MongoDB по специальному сетевому протоколу [7], затем перевести эти запросы на язык базы данных ClickHouse и исполнить, выводя ответ согласно MongoDB формату, то есть типа bson [28] - бинарный вариант формата json.

Структура работы

Всю работу можно разделить на 4 части. Сначала нужно реализовать прототип ручки (мок). Обработчик будет принимать лишь привественные сообщения, читать и отвечать на них.

Затем реализовать получение запросов, не меняюших таблицу (immutable - немутабельных), они будут обрабатываться, результат работы будет выведен согласно протоколу MongoDB. Содержимое запросов тоже будет обрабатываться с некоторыми ограничениями: будет приниматься лишь плоский тип json, то есть документ не должен иметь вложенные поля.

Далее планируется уметь обрабатывать произвольную структуру json, но запросы все еще должны быть немутабельными.

В конечном итоге будет добавлена возможность обрабатывать и мутабельные запросы (те, что меняют таблицу, то есть добавление, удаление, создание и т.д.), но с ограничениями: содержимым запроса должен быть плоский json.

Основные результаты

В проекте было реализовано первые 2 задачи, то есть немутабельные плоские запросы. Однако пункты неравноценны по трудности: в общей сложности была сделана бОльшая часть работы. На данный момент, клиент общается с сервером, принимаются команды find, aggregate.

Обзор литературы

В начале работы я выбрал 3 примера, на которые можно было ориентироваться:

1) Oxide [10] - это проект на языке Rust [23], который транслирует запросы MongoDB для PostgreSQL [22] (строчная SQL БД). Oxide принимает запросы от MongoDB клиента, парсит их, отправляет PostgreSQL серверу, получает обратно результат работы, парсит его и отправляет MongoDB клиенту. Проект поддерживает также mutable запросы с nested json. Проект разрабатывался парой людей для личного опыта.

2) FerretDB [4] - проект на языке GO [20], идейный вдохновитель проекта Oxide, который также транслирует MongoDB запросы для PostgreSQL. Делает то же самое, что и Oxide, но написан на языке GO. Однако сам проект намного больше Oxide: в нем имеется поддержка другой базы данных SQLite [24], проект имеет планы на будущее и над ним работает целая команда разработчиков [3].

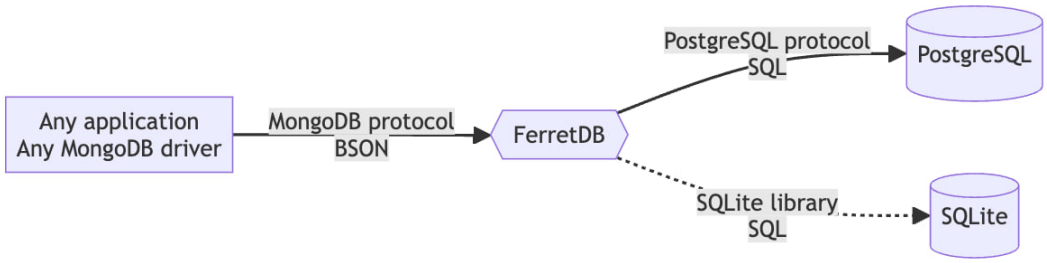

Рис. 0.2: Принцип работы FerretDB

3) Ручка в ClickHouse [29], которая транслирует запросы PostgreSQL. Аналогично моему проекту, эта ручка парсит сообщение, исполняет запрос и отправляет результат на PostgreSQL клиент.

Все эти проекты схожи с моим по своей специфике: они все трансляторы запросов, связанные либо с MongoDB, либо с ClickHouse. Однако ни один не похож на этот проект в точности. В ClickHouse не реализован протокол общения с MongoDB, кроме этого проекта.

FerretDB на платформе Github [**GitHub**] имеет около 8.5 тысяч звезд [4], показывая значимость этого проекта для сообщества. Такая популярность дает понять, что аналогичный инструмент для ClickHouse может стать общественно значимым: ClickHouse в некоторых аспектах лучше PostgreSQL, поэтому пользователи, которые хотят писать свои запросы через MongoDB, смогут перейти с FerretDB на ClickHouse через транслятор, работая с более удобной для них базой данных.

Также стоит обратить внимание на наличие PostgreSQL (и MySQL) ручки в ClickHouse, которая также транслируют запросы. Благодаря ним пользователю легче перейти из дру-

гой базы данных в ClickHouse, так как скрипты переписывать не надо: разработчики уже реализовали слой трансляции запросов.

Начало работы: Парсинг

Формат BSON

MongoDB отличается от стандартных SQL-языков тем, что он принимает запросы и хранит свои данные в виде json-документов. Для компактного представления этого формата в двоичном виде был придуман bson [28] (binary json - бинарный json).

Bson используется для передачи данных в клиент-сервер коммуникации. Поэтому для работы с клиентом мне требовалось реализовать bson decoder и encoder, то есть функции, которые могли из бинарного вида восстановить содержимое bson и обратно.

Сначала я заглянул в открытую библиотеку Poco [11], на которой частично базировался код ClickHouse. В библиотеке уже был написан функционал для работы со структурой типа bson, а именно: сам класс bson (или Document), классы для каждого поддерживаемого типа данных (Int32, String, bool, double и т.д.), а также классы BSONWriter и BSONReader, которые имели в себе функции для чтения и записи элементов. Как известно, C++ - строго типизированный язык, но класс bson может хранить произвольные данные, в связи с этим стоит объяснить, как bson был реализован.

Один из способов хранить произвольные данные в языке C++ - это использовать механизм наследования [17]. Конкретно здесь вводился базовый класс Element, у которого было только имя (строка). От него наследовался шаблонный класс ConcreteElement<T>, который уже хранил значение с соответственным типом. Далее в структуре Document, которая отражала bson документ, хранился вектор std::vector<Element::Ptr>, где Element::Ptr - это shared_ptr (std или Poco) [12] от класса Element. Благодаря наследованию в ячейке вектора мог храниться произвольный указатель на ConcreteElement. Операции добавления, удаления, поиска по имени делаются тривиально.

Как уже было сказано ранее, эти классы были реализованы в библиотеке Poco, однако пришлось переписать код. Переиспользовать не получилось по причине того, что в Poco и ClickHouse были несовместимые классы для чтения/записи. В связи с этим пришлось переписать полностью bson структуру из библиотеки Poco под классы ReadBuffer, WriteBuffer - базовые классы для чтения/записи в ClickHouse.

Также стоит отметить, что из нововведений мной был добавлен другой алгоритм вы-

вода: в Poco вывод сначала пишется в строку, затем считается и выводится размер (согласно сетевому протоколу MongoDB в начале сообщения пишется его размер) и содержимое строки уже копируется в вывод. Я реализовал функцию getLength(), которая выводила количество байт, которые займет объект при выводе. В связи с этим в выводе пропали лишние копирования.

Формат сообщений MongoDB

После написания вспомогательных классов для bson стоило приступить к классам для MongoDB сообщений.

MongoDB в общей сложности состоит из 10 разных типов сообщений, однако все кроме одного давно устарели и сейчас используется лишь OpMsg. Это сообщение состоит из заголовка, флагов и документа/документов типа bson.

Несмотря на то, что остальные типы устарели, помимо OpMsg мне еще пришлось реализовать OpQuery и OpReply - запрос от клиента и ответ сервера соответственно. Пришлось это сделать из-за обратной совместимости, а именно из-за поддержки старых серверов клиент вначале присылает приветственное сообщение с помощью типа OpQuery, и если в ответном сообщении версия сервера будет подходящей, клиент начнет отправлять OpMsg сообщения.

Чтение и запись сообщений реализовать достаточно легко: главное знать их структуру. С OpMsg не возникало проблем, ведь структура этого типа описана на официальном сайте, но для OpQuery и OpReply пришлось обратиться к сторонним сайтам [6].

Простой эхо ответчик

После реализации bson формата и сообщений нужно было начать работать с сетью. Мне не удалось получить какой-либо документации по созданию своего сервера в ClickHouse, однако у меня для примера был вышеупомянутый PostgreSQL сервер трансляции запросов. Благодаря нему я настроил базовый ввод и вывод потока байт.

В реализации работы сервера не было трудных концепций. ClickHouse рассчитан на то, что ему не будет приходить много запросов: его должны использовать небольшое количество клиентов, но с большими нечастыми запросами. Из-за такого подхода сервера в ClickHouse синхронные [27] и однопоточные [18] (для каждого соединения создается новый поток): с помощью обертки вызова epoll в linux [25] сервер ждет изменений на сокете (готовность к чтению или ошибке), затем делает все нужные операции также синхронно и отправляет результат.

Перед тем как исполнять сложные запросы, стоило для начала проверить работоспособность уже написанного кода. Для этого я написал эхо-ответчик. Это простейший обработчик сообщений, который выводит ровно то, что ему приходит (по аналогии с командой echo [15] в bash). Работоспособность можно было увидеть по логам: сообщения корректно читались и писались.

Кроме логов, также применялось ручное тестирование. Для этого я использовал утилиту nc [16] - утилита в linux для простейших сетевых взаимодействий (отправка и получение информации), чтобы читать, что выводит сервер на мои сообщения. Затем, чтобы распарсить ответы, я написал свой парсер на языке Rust.

Итоги главы

В начале работы я столкнулся с многими трудностями: как общается MongoDB, как устроен bson, переписать библиотеку Poco или написать свою реализацию, где искать описания сообщений и так далее. Спустя огромные часы работы я пришел к следующему результату: был реализован Document для работы с типом bson, были написаны разные типы сообщений и их парсеры, а также был написан простой эхо-сервер, который получал сообщения и отправлял их обратно.

Работоспособность этого этапа была проверена с помощью логирования и ручных тестов при помощи утилиты nc.

Установка коммуникации с клиентом

После того, как был написан простой эхо-обработчик, шел следующий этап: подключение. Перед началом работы с сервером, клиент обменивается несколькими служебными запросами, чтобы узнать составляющую сервера. К примеру, определить его версию, и после этого общаться с ним по устаревшему или текущему протоколу.

Главной задачей было определить, на какие запросы нужно отвечать, чтобы клиент дал работать с севрером. К сожалению, интересующая информация не была найдена в открытых источниках, поэтому я решил обратиться к вышеупомянутым проектам, которые также обрабатывают приветственные сообщения MongoDB, то есть Oxide и FerretDB. Там я нашел содержимое сообщений, однако не нашел, какие из них требуются в приветствии.

Позже я решил написать прокси на языке Rust. Его реализация была простой: он слушал сообщения с клиента, выводил их содержимое (в распарсенном виде, который был

реализован до этого) в файл, затем отправлял запрос дальше на сервер. Потом прокси делал то же самое для сервера.

При помощи такого инструмента мне удалось найти приветственные сообщения: hello, IsMaster, getLog, buildInfo и т.д. Благодаря тому же прокси я посмотрел, какую информацию на них нужно выводить, и захардкодил эту информацию в код. Захардкодить значит вписать данные, которые должны браться из каких-то источников (к примеру из конфигов), в код, как есть, то есть без реальной добычи их откуда-либо.

Таким образом, мне открылся доступ к выполнению команд в MongoDB клиенте.

Команды

Парсер команд

Перед тем, как приступать к реализации конкретных команд, решено было сначала написать их парсер. Я написал структуру Command, которая в себе хранила необходимую информацию для команд, такую как: данные запроса, название ДБ, в которую идет запрос и таблица, к которой обращаются. В парсере parseCommand я доставал все эти поля из сообщения OpMsg, затем возвращал структуру Command.

Для удобного добавления/удаления команд и в целом их отображения я также ввел enum [2] CommandTypes, где были описаны все поддерживаемые на текущий момент команды.

Запуск команд

После того, как я из сообщения получал распарсенную команду, я решил посмотреть, как запускать команды в ClickHouse, и из этого думать, как мне переводить запросы на язык ClickHouse.

Для того, чтобы найти, как запускать команды, я посмотрел реализацию вышеупомянутой ручки PostgreSQL в ClickHouse. Она проводила аутентификацию пользователя, заполняла контекст запроса (название таблицы, базы данных и т.д. - служебная информация) и запускала функцию executeQuery. Функция принимала класс, реализующий поток ввода/-вывода (ReadBuffer/WriteBuffer) и контекст запроса. В потоке ввода нужно было написать запрос точно так же, как если бы я написал его в клиенте ClickHouse, к примеру SELECT * FROM table_name. В потоке вывода функция писала столбцы результата.

Затем я реализовал функцию, которая переводит результат запроса в bson формат. Какой формат сообщения требуется клиенту MongoDB, я узнал с помощью своего прокси на Rust, запуская нужные мне запросы с клиента, подключенного к обычному MongoDB серверу.

На данном этапе оставалась последняя проблема: для вывода в формат bson нужно было знать типы выводимых данных, из-за чего сначала я выводил весь результат в формате строки. Убедившись, что все работает корректно, я начал искать решение проблемы.

Ответ находился на официальном сайте ClickHouse в разделе запроса SELECT, секции FORMAT [5]. ClickHouse поддерживает различные форматы вывода: CSV, JSON, XML и т.д. Меня интересовал формат TabSeparatedWithNamesAndTypes. Он в первых двух строках выводил название и тип данных. Я написал функцию, которая переводила строку с названием типа в ConcreteElement с тем же типом и значением.

Таким образом я наладил выдачу результата выполнения команды, осталось команды исполнять.

Find

Find - самая базовая немутабельная операция в MongoDB. Она позволяет достать какие-то данные из коллекции (название таблицы в MongoDB) по какому-то фильтру, задать сортировку этих данных, задать столбцы для вывода, поставить лимит строк в выводе и так далее. По сути эквивалентна SQL запросу SELECT с командами WHERE, ORDER BY, LIMIT, OFFSET и т.д.

Для удобного использования данных из сообщения я написал структуру FindCommand. Она хранила нужные мне поля запроса в отдельных переменных. Вместе с этим я написал простую функцию, переводящую Command в FindCommand.

Фильтрация

Сначала я начал реализовывать фильтрацию строк. Для этого в MongoDB запросе есть отдельное поле filter. В MongoDB поддерживается много разных режимов фильтрации: строки можно фильтровать по точному совпадению имени или регулярному выражению, числа можно сравнивать операторами $>$, $<$, $==$, можно фильтровать документы по наличию или отсутствию определенного поля. Я решил сделать базовые: операторы сравнения чисел и строки.

В ClickHouse фильтрация происходит стандартным для SQL способом через оператор WHERE. Условия перечисляются через запятую после оператора. В упрощенном виде условия пишутся как <имя колонки> <оператор> <значение>, к примеру это условие: WHERE age >= 18 выдаст все строки, где колонка "age"имеет значение не меньше 18. В общем случае на месте имени колонки может стоять какое-то выражение, к примеру WHERE monthly_income * 12 > 350000 выведет все строки, где годовой доход больше 350 тысяч.

В проекте был реализовал упрощенный вид WHERE условия. Я написал функцию, которая принимала документ типа bson, которое было в поле filter запроса. Затем смотрел, для какой колонки очередное условие, смотрел его тип и писал после WHERE через запятую. Результирующую строку выводил.

Таким образом получилось реализовать условия вида SELECT * FROM table_name WHERE <условие>.

Проекция

Проекция (или project) - это часть запроса find, которая отвечает за то, какие столбцы нужно выводить. Без каких-либо условий выводятся все столбцы запроса, в том числе и столбец _id - служебный столбец в MongoDB, который создается автоматически для каждого документа.

Формат проекции простой: <имя столбца>: <true/false> - включать или не включать столбец в вывод. Если вводятся только значения false, то сервер должен включить в вывод все столбцы, кроме тех, что перечислены в документе. Если имеется хотя бы один столбец с значение true, то выводятся только те, которые были перечислены в документе с значением true. Дополнительно всегда выводится служебный столбец _id, если он не был явно исключен.

В ClickHouse SQL запросе есть механизм, который делает то же самое, что и проекция. Для этого достаточно в SELECT запросе написать не "*"(все столбцы) а явно поименно указать те, которые нам нужны. К примеру, SELECT name, age FROM table выведет только name и age столбцы из всей таблицы.

Здесь я столкнулся с трудностью в том, что имена столбцов даются после выполнения запроса, а они нужны были до. Поэтому я реализовал проекцию несколько другим способом: в ClickHouse query я передавал SELECT *, то есть просил его вывести все столбцы, и уже впоследствии оставлял только те, которые были нужны.

Данное решение проблемы крайне неэффективно, так как таблицы обычно имеют

много столбцов, из которых требуется вывести небольшое количество, сильно ускоряя запрос. Однако, на данный момент это была самая простая реализация. В будущем будет нетрудно внести изменения, чтобы запрос работал более эффективно. Для этого, к примеру, можно перед основным запросом сделать вспомогательный, который вернет структуру таблицы.

Сортировка

Поле sort в find запросе задает сортировку результата, а именно по каким столбцам, в какую сторону и в какой очередности сортировать столбцы результата.

Для сортировки в ClickHouse запросе существует специальный оператор ORDER BY, который после себя принимает набор столбцов с указанием порядка: ASC (ascending - возрастающий) и DESC (descending - убывающий). К примеру, запрос SELECT * FROM table ORDER BY money DESC выведет столбцы в порядке убывания поля money.

В MongoDB запросе поле sort похоже на project. Сортировка столбца задается так: <имя столбца> : <1/-1>, где 1 соотвествует ASC, а -1 - DESC. Соответственно, функция обработки поля sort похожа на функцию обработки поля project: через запятую по очереди обрабатывается имя и значение 1 или -1, затем все вставляется в запрос с оператором "ORDER BY".

Лимит

limit поле в запросе find представляется в виде целочисленного значения. limit обозначает количество документов, которые нужно вывести в результирующем запросе.

В ClickHouse limit задается как LIMIT <число> в конце запроса. К примеру, запрос SELECT * FROM table ORDER BY age DESC LIMIT 10 выведет 10 самых взрослых людей из таблицы.

В MongoDB поле limit также представляется просто как целочисленное значение. Трансляция оператора limit была тривиальной: нужно было просто вставить в конец запроса LIMIT <число>.

Итоги главы

Был реализован запрос find, поддерживаюший также дополнительные операторы filter, project, sort, limit, которые фильтруют, задают столбцы для вывода, сортируют и обрезают

запрос соответственно.

Запрос, эквивалентный SELECT age, name WHERE age <= 30, age >= 18 ORDER BY name ASC LIMIT 100 в MongoDB корректно отработает и выведет 100 столбцов age, name для людей, не младше 18, но и не старше 30, в алфавитном порядке имен.

Aggregate

Команда aggregate [1] и ее аналоги в SQL языках очень популярны, так как они позволяют группировать данные по каким-то признакам и получать общую информацию об этой группе. К агрегирующим операция относится сумма, среднее, минимум, максимум и т.д. Эти функции очень полезны для аналитиков. К примеру, если нужно рассчитать средний чек за период времени или найти количество всех товаров, прибывших вчера.

Запрос типа aggregate в MongoDB разделен на этапы (stage), на каждом из которых выполняется свой тип запроса. Из основых типов можно выделить: group, sort, limit, project, match, skip, count. Их все я и реализовал.

Group

Поле group основное для aggregate запроса, так как оно говорит, по какому столбцу будет идти группировка. Для этого в group есть элемент _id, который либо равен строке, обозначающей название столбца, либо принимает null значение, обозначая, что запрос идет для всей таблицы. В ClickHouse группировка выражается в виде оператора GROUP BY <name1>, <name2>, ..., который группирует объекты по значениям в полях. К примеру, SELECT city, COUNT(*) as population FROM table GROUP BY city выведет количество строк, соотвествующих этому городу.

Кроме _id в group есть и другие поля вида <название нового столбца>: <выражение>, где в выражении описано, какую операцию применить к какому столбцу. К примеру, "c": {"avg": "age"} будет соответствовать запросу SELECT AVG(age) as c FROM table - где в столбец "c"запишется средний возраст по всей таблице.

Реализация group не составляла особых трудностей. Единственное, что нужно было реализовать, это парсинг типа операции и ее имя, после чего в соответственном формате строка вставлялась после SELECT.

Остальные операторы

Операторы sort, limit, project, match - аналогичны операторам sort, limit, project, filter из функции find. Их реализация почти не отличалась от аналогичных операторов.

Оператор SKIP представим аналогично оператору LIMIT: тоже принимал одно целочисленное значение. SKIP действует, как OFFSET в ClickHouse: OFFSET <n> пропускает первые n строк в запросе.

Оператор COUNT реализован на базовом уровне как аналог функции COUNT в ClickHouse. Этот оператор считает количество строк в запросе, как SUM(1). Текущая реализация поддерживает лишь COUNT над одним столбцом.

Итоги главы

В проекте была реализована функция aggregate - важный инструмент при работе с базами данных. На данный момент поддерживаются операторы sum, avg и частично count, вместе с другими операторами, такими как sort, limit, offset и т.д.

Тестирование

В ClickHouse существуют свои инструменты для тестирования. Я использовал stateless [14] тестирование, то есть то, в котором нет никаких данных до запроса - таблицы заполняются в самом запросе. Тест описывается в bash скрипте, затем запускается специальная программа, которая по очереди запускает эти скрипты и сравнивает вывод с нужным. Эталонный вывод запроса нужно написать самостоятельно до запуска тестов.

Я написал несколько тестов на команды find и aggregate. Для команды find тесты написаны для каждого реализованного оператора, а для aggregate, по совету руководителя, тесты были взяты из специального репозитория [8]. Перенесена была лишь часть тестов, поддерживаемая на данный момент.

Заключение

В проекте были переписаны классы для работы с bson и MongoDB сообщениями, была налажена коммуникация между сервером и клиентом, реализованы команды find, aggregate

для immutable запросов к таблице. Также написаны функции для парсинга результатов запроса, приведение к MongoDB выводу.

Над этим проектом была проделана огромная работа с моей стороны, однако он еще далек от идеала. В будущем можно будет добавить поддержку мутабельных операций, добавить больше функций, расширить функционал уже имеющихся (добавить больше операторов в aggregate), настроить админские действия (просмотры существующих баз данных и таблиц, получение логов, авторизация). На данный момент ручка трансляции - это упрощенная версия аналогичных проектов, но уже сейчас в ней можно базово взаимодействовать с ClickHouse, выполняя нужные запросы.

Положительными моментами можно отметить то, что большинство кода прологгировано, в большом количестве мест есть выброс прописанных исключений, которые помогут в случае возникновения ошибки.

Из отрицательных сторон можно отметить сырость проекта как полноценной ручки. Многое еще нужно доделать и, возможно, частично переписать старое. Во многих моментах остались TODO и FIXME, которые при внимании приведут к более готовому продукту.

Список литературы

[1] *Aggregate в MongoDB*. URL: https://www.mongodb.com/docs/manual/reference/command/aggregate/.

[2] *Enum в c++*. URL: https://en.cppreference.com/w/cpp/language/enum.

[3] *FerretDB компания*. URL: https://github.com/FerretDB.

[4] *FerretDB репозиторий*. URL: https://github.com/FerretDB/FerretDB.

[5] *FORMAT в ClickHouse*. URL: https://clickhouse.com/docs/ru/sql-reference/statements/select/format.

[6] *MongoDB deprecated OP messages*. URL: https://russianblogs.com/article/2790759058/.

[7] *MongoDB Wire Protocol*. URL: https://www.mongodb.com/docs/manual/reference/mongodb-wire-protocol/.

[8] *MongoDB тесты в ClickHouse*. URL: https://github.com/ClickHouse/ClickBench/blob/main/mongodb/queries.js.

[9] *NoSQL википедия*. URL: https://ru.wikipedia.org/wiki/NoSQL.

[10] *Oxide репозиторий*. URL: https://github.com/fcoury/oxide.

[11] *POCO C++ libraries*. URL: https://pocoproject.org/.

[12] *Poco SharedPtr*. URL: https://docs.pocoproject.org/current/Poco.SharedPtr.html.

[13] *SQL википедия*. URL: https://ru.wikipedia.org/wiki/SQL.

[14] *Stateless тесты в ClickHouse*. URL: https://github.com/ClickHouse/ClickHouse/tree/master/tests/queries/0_stateless.

[15] *Команда echo в Linux*. URL: https://losst.pro/komanda-echo-v-linux.

[16] *Команда nc*. URL: https://losst.pro/komanda-nc-v-linux.

[17] *Наследование c++*. URL: https://en.cppreference.com/book/intro/inheritance.

[18] *Однопоточное выполнение*. URL: https://ru.wikipedia.org/wiki/%D0%9E%D0%B4%D0%BD%D0%BE%D0%BF%D0%BE%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D0%B5_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5.

[19] *Официальный майт MySQL*. URL: https://www.mysql.com/.

[20] *Официальный сайт Go*. URL: https://go.dev/.

[21] *Официальный сайт MongoDB*. URL: https://www.mongodb.com/.

[22] *Официальный сайт PostgreSQL*. URL: https://www.postgresql.org/.

[23] *Официальный сайт Rust*. URL: https://www.rust-lang.org/.

[24] *Официальный сайт SQLite*. URL: https://www.sqlite.org/.

[25] *Понятие epoll*. URL: https://ru.wikipedia.org/wiki/Epoll.

[26] *Репозиторий ClickHouse*. URL: https://github.com/ClickHouse/ClickHouse.

[27] *Синхронное выполнение*. URL: https://habr.com/ru/articles/453192/.

[28] *Спецификация BSON*. URL: https://bsonspec.org/spec.html.

[29] *Транслятор PostgreSQL для ClickHouse*. URL: https://github.com/ClickHouse/ClickHouse/blob/master/src/Server/PostgreSQLHandler.cpp.

[30] *Формат JSON*. URL: https://www.json.org/json-en.html.