Оглавление

Введение

За последнее десятилетие объем интернет-трафика колоссально вырос: коммерческие компании обрабатывают все больше данных и трафика, что, в свою, очередь, вынуждает их создавать новые инструменты, подходящие для эффективной работы в подобных масштабах.

Высоконагруженные данными приложения (data-intensive applications, DIA) открывают новые горизонты возможностей благодаря использованию современных технологических усовершенствований. Говорят, что приложение является высоконагруженным данными (data-intensive), если те представляют основную проблему, с которой оно сталкивается, — качество данных, степень их сложности или скорость изменений, — в отличие от высоконагруженного вычислениями (compute-intensive), где узким местом являются циклы CPU [1]. Далее высоконагруженное данными приложение будем называть просто высоконагруженным.

Инструменты и технологии, обеспечивающие хранение и обработку данных с помощью DIA, играют важную роль в разработке современных программных продуктов, а знание и умение применят те или иные технологии DIA в некоторой степени является показателем профессионализма разработчика.

Целью данного проекта является моделирование высоконагруженной системы с применением механизмов и алгоритмов масштабирования базы данных на примере RESTful [2] приложения, предоставляющего данные об перелетах, а также эмпирическая оценка эффективности разработанной системы. Для достижения поставленной цели в ходе работы требуется решить следующие задачи:

1. провести анализ существующих алгоритмов и методов построения высоконагруженной системы, выбрать из них подходящие для выполнения проекта;
2. с помощью выбранных методов разработать систему;
3. выбрать технологии (СУБД, язык программирования, фреймворк) для реализации поставленной задачи;
4. произвести нагрузочное тестирование системы.

1 Аналитическая часть

В данном разделе проанализирована поставленная задача, а также рассмотрены различные способы ее реализации.

1.1 Постановка задачи

В рамках курсового проекта необходимо реализовать высоконагруженное RESTful [2] приложение для поиска информации об авиаперелетах, и включить в него следующий функционал:

- предоставить возможность пользователем считывать данные с помощью GET-запросов на сервер;
- разрешить вносить изменения в данные только администратору сервиса.

Предполагается, что подавляющее большинство запросов - запросы на чтение.

1.2 Формализация данных

База данных должна хранить следующую информацию:

1. данные о самолетах, аэропортах, авиакомпаниях;
2. информацию о рейсах, связанную с данными об аэропортах, самолетах и авиакомпаниях.

В таблице 1.1 приведены выделенные категории данных и краткие сведения о них:

1.3 Типы пользователей

Для изменения данных в базе необходимо иметь соответствующие права администратора - необходима аутентификация суперпользователя.

.

Таблица 1.1: Категории данных

Категория	**Сведения о категории**
Самолет	ID самолета (номер регистрации); модель, месяц и год ввода в эксплуатацию; фотография судна
Аэропорт	ID аэропрта (в международной кодировке); расположение: город, провинция (штат), страна, координаты;
Авиакомпания	ID авиакомпании (в международной кодировке); название; немного контактной информации (номер телефона, адрес центрального офиса и т.д.)
Рейс	ID рейса; коды аэропортов вылета и прилета; ID самолета, выполнившего рейс; номер рейса (ID авиакомпании и числовой код); время в пути; преодолимое расстояние; статус рейса (задержан, отменен и т.д.).

Таким образом в системе различается два вида пользователей: неаутентифицированные («гости») и аутентифицорованные (администраторы). В таблице 1.2 приведен функционалл каждого типа пользователя.

Таблица 1.2: Функционал пользователя

Тип пользователя	**Функционал пользователя**
Неуаутентифицированный пользователь	Чтение данных
Аутентифицированный пользователь	Чтение, добавление, удаление, изменение данных

1.4 Обзор существующих СУБД

Система управления базами данных, сокр. СУБД — совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление созданием и использованием баз данных [3].

Основными функциями СУБД являются:

- управление данными во внешней памяти, в оперативной памяти с использованием дискового кэша;
- журнализация изменений, резервное копирование и восстановление базы данных после сбоев;

- поддержка языков БД.

1.4.1 Классификация СУБД по способу хранения

По способу хранения выделяют два вида баз данных:

1. СУБД с построчным хранением данных;
2. СУБД с колоночным хранением данных.

В СУБД с построчным хранением данных записи хранятся в памяти построчно. Для таких систем характерно большое количество коротких транзакций с операциями вставки, обновления и удаления данных. Зачастую их используют в транзакционных системах (OLTP-системы). Основными задачами транзакционных систем являются:

- быстрая обработка запросов на добавление, изменение, удаление записей;
- поддержание целостности данных.

Показателем эффективности системы является количество транзакций в секунду.

Записи в СУБД колоночного типа представляются в памяти по столбцам. Данный тип хранения данных нашел применяется в аналитических системах, для которых характерно относительно небольшое количество транзакций, а запросы на чтение зачастую вычислительно сложны и включают в себя агрегацию данных. Время отклика является показателем эффективности аналитических систем.

1.4.2 Классификация СУБД по модели данных

Модель данных — это абстрактное, самодостаточное, логическое определение объектов, операторов и прочих элементов, в совокупности составляющих абстрактную машину доступа к данным, с которой взаимодействует пользователь. Эти объекты позволяют моделировать структуру данных, а операторы — поведение данных [4].

Выделяют 2 основных типа моделей организации данных:

- реляционная (SQL);
- нереляционная (NoSQL).

Реляционная модель данных является совокупностью данных и состоит из набора двумерных таблиц. При табличной организации отсутствует иерархия элементов. Таблицы состоят из строк – записей и столбцов – полей. На пересечении строк и столбцов находятся конкретные значения. Для каждого поля определяется множество его значений. За счет возможности просмотра строк и столбцов в любом порядке достигается гибкость выбора подмножества элементов. Таблицы реляционной СУБД могут быть связаны между собой с помощью внешних ключей (англ. foreign key), таким образом образуя некоторые отношения в базе данных.

Реляционные базы данных обеспечивают набор свойств ACID: атомарность, непротиворечивость, изолированность, надежность. Так, атомарность требует, чтобы транзакция выполнялась полностью или не выполнялась вообще; непротиворечивость означает, что сразу по завершении транзакции данные должны соответствовать схеме базы данных; изолированность требует, чтобы параллельные транзакции выполнялись отдельно друг от друга, а надежность подразумевает способность восстанавливаться до последнего сохраненного состояния после непредвиденного сбоя в системе или перебоя в подаче питания.

Реляционная модель является удобной и наиболее широко используемой формой представления данных. Примером популярных реляционных СУБД являются PostgreSQL [5], Oracle [6], Microsoft SQL Server [7].

На рисунке 1.1 приведена схема реляционной модели данных.

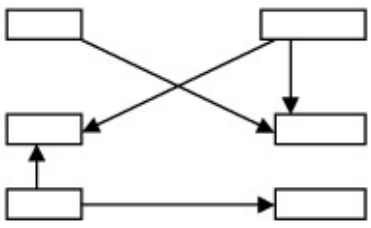

Рисунок 1.1: Схема реляционной модели данных

В отличие от реляционных СУБД, нереляционные базы данных не имеют одного общего формата. Данные в NoSQL СУБД могут хранится в виде пар ключ-значение (Redis [9], LevelDB [10]), документов (MongoDB, Tarantool), графа (Neo4j, Giraph).

Как правило, базы данных NoSQL предлагают гибкие схемы, что позволяет осуществлять разработку быстрее и обеспечивает возможность поэтапной реализации. Благодаря использованию гибких моделей данных БД NoSQL хорошо подходят для частично структурированных и неструктурированных данных.

NoSQL базы данных зачастую предлагают компромисс, смягчая жесткие требования свойств ACID ради более гибкой модели данных, которая допускает горизонтальное масштабирование.

1.5 Обзор существующих методов и технологий DIA

1.5.1 Индексирование

Чтобы выполнить SELECT-запрос с предикатом WHERE система должна последовательно ряд за рядом (строка за строкой) просканировать таблицу целиком, чтобы найти все, удовлетворяющие условию вхождения. Это крайне неэффективная стратегия поиска, если в таблице много строк, и гораздо меньше (например, всего пара) целевых значений, которые надо вернуть. Но если создать в системе индекс на атрибуте, по которому проверяется условие в SELECT-запросе, СУБД сможет использовать более эффективные стратегии поиска. Так, например, системе может потребоваться всего несколько шагов в глубину по дереву поиска.

Схожий подход используется в большинстве документальной литературе и энциклопедиях: основные термины и концепты, которые часто ищутся читателями, вынесены в алфавитный указатель в конце книги. Читатель может как прочитать всю книгу целиком и найти интересующую его информацию (последовательный поиск), так и открыть алфа-

витный указатель и быстро пролистать страницы до нужной (поиск по индексу).

Индексы могут также положительно сказываться на производительности UPDATE и DELETE запросов, могут использоваться в JOIN-поисках [8].

1.5.2 Кэширование

Зачастую приходится выполнять сложные SELECT-запросы: с JOIN-ом больших таблиц, оконными функциями и т.д. Если такой запрос СУБД выполняет относительно часто, тогда имеет смысл сохранять (на некоторое время) результат выполнения такого запроса, чтобы при следующем обращении в систему с таким же запросом, система вернула сохраненный результат, а не выполняла вычислительно сложный запрос еще раз. Это называется кэшированием результата запроса.

В качестве кэша часто используются key-value хранилища, такие как Redis [9] и LevelDB [10].

Существует множество стратегий (алгоритмов) кэширования. Самым популярным и простым алгоритмом кэширования является алгоритм LRU (от англ. Least Recently Used) [11]. Суть этого алгоритма заключается в том, что из кэша вытесняются значения, которые дольше всего не запрашивались. Реализуется простой очередью (FIFO). Новые и запрошенные объекты перемещаются в начало очереди, соответственно, в конце очереди оказываются самые непопулярные объекты.

Несмотря на свою простоту и легкость создания, кэш LRU имеет критический недостаток: при добавлении новых объектов в переполненную очередь-LRU из очереди может быть вытеснен «теплый» (часто запрашиваемый) объект. Чтобы решить эту проблему, был предложен алгоритм 2Q [12] (2 queue - англ. 2 очереди). Исходя из названия алгоритма, он использует вместо одной очереди две: в первой («горячей») очереди реализован обычный алгоритм LRU, а во второй («теплой») - простая очередь. Новые объекты помещаются в «теплую» очередь и постепенно сдвигаются к концу очереди, после чего вытесняются. Однако если пока объект находился «теплой» очереди, к нему произошло обращение, то

запрашиваемый объект перемещается в «горячую» очередь.

1.5.3 Репликация

Репликация - это создание копий БД на разных серверах, но с разными привилегиями (правами). Создав несколько копий БД, можно балансировать нагрузку между хостами.

Выделяют два типа реплик:

- Master-реплики - БД, в которой можно выполнять любые операции над данными.
- Slave-реплики - БД, из которой только можно считать информацию.

Репликация Master-Slave - это репликация, при которой единовременно существует один мастер и множество слейвов. Запросы на чтение равномерно распределяются между слейвами, в то время как запросы на модификацию данных поступают на мастер. Обновленная информация в мастере распространяется на слейвы, чтобы поддерживать их актуальность.Если выходит из строя мастер, нужно переключить все операции (и чтения, и записи) на слейв. Таким образом он станет новым мастером. А после восстановления старого мастера, настроить на нем реплику, и он станет новым слейвом.

Преобладание запросов на чтение, над запросами на изменение является ключевым критерием эффективности применения стратегии Master-Slave репликации [13]. Также некоторые из слейв-БД могут быть использованы в качестве резервной копии системы.

Master-Master репликация подразумевает наличие нескольких мастер-узлов. Это накладывает свои ограничения на систему: выход из строя одного из серверов практически всегда приводит к потере каких-то данных. Последующее восстановление также сильно затрудняется необходимостью ручного анализа данных, которые успели либо не успели скопироваться.

1.5.4 Шардирование

В жизненном цикле программного продукта может возникнуть ситуация, когда вертикальной стратегии масштабирования (путем наращивания вычислительной мощности хостов кластера: дисков, памяти и процессоров) на репликах может оказаться недостаточно. В этом случае можно разделить огромные таблицы данных на части и распределить их по репликам. В сущности, в этом и заключается смысл шардирования данных.

Выделяют три стратегии шардирования:

1. вертикальное шардирование - поколоночное деление таблиц;
2. горизонтальное шардирование - построчное деление таблиц ;
3. диагональное шардирование - как следует из названия, это комбинация вертикального и горизонтального подхода.

Шардирование обеспечивает несколько преимуществ, главное из которых — снижение издержек на обеспечение согласованного чтения, которое для ряда низкоуровневых операций требует монополизации ресурсов сервера баз данных, внося ограничения на количество одновременно обрабатываемых пользовательских запросов, вне зависимости от вычислительной мощности используемого оборудования). В случае шардинга логически независимые серверы баз данных не требуют взаимной монопольной блокировки для обеспечения согласованного чтения, тем самым снимая ограничения на количество одновременно обрабатываемых пользовательских запросов в кластере в целом [14].

Вывод

Изучив существующие виды СУБД, было принято решение использовать реляционную СУБД для хранения данных о перелетах, так как имеющиеся данные организованы в таблицы, а также имеют связь между

собой, которую удобно реализовать с помощью внешних ключей; нереляционную СУБД, в которой данные хранятся в виде пар ключ-значение, в качестве кэша запросов, поскольку такая структура хранения удобна для кэширования результатов исполнения запросов к БД: ключом является, например, SQL-выражение запроса, а значением - закэшированные данные.

Проведенный анализ существующих методов и технологий DIA позволяет сделать вывод, что для решения поставленной задачи наилучшим решением является применение каждого из подходов, за исключением шардирования, поскольку объем данных, используемый в проекте относительно невелик:

- создание индексов - для ускорения выполнения SELECT-запросов;
- кэширование - для сохранения частых запросов; в качестве кэша принято решение использовать алгоритм 2Q, поскольку в нем решена основная проблема алгоритма LRU - вытеснение «теплых» данных;
- репликация Master-Slave, поскольку по условию задания SELECT-запросы преобладают в количестве над запросами на изменение данных.

Определившись с методами и алгоритмами, возникает следующая проблема: адаптация всего этого под конкретную задачу. Подробно это, а также детали реализации в следующем разделе.

2 Конструкторская часть

В данном разделе представлены требования к программному обеспечению, а также приведены способы реализации методов и алгоритмов DIA, применимых к решению поставленной задачи.

2.1 Требования к программному обеспечению

Программа должна предоставлять доступ к следующим возможностям:

1. чтение, создание, изменение и удаление данных БД;
2. изменение пользователем параметров запроса (например, с помощью аргументов адресной строки).

К программе предъявляются следующие требования:

- программа должна запрещать создавать, изменять, удалять записи из базы данных неаутентифицированным пользователям.

2.2 Проектирование базы данных

На основании исследования существующих СУБД было решено использовать реляционную СУБД PostgreSQL, поскольку эта СУБД свободно распространяется и имеет обширную, подробную документацию. В PostgreSQL реализована поддержка языка plpgsql, который упрощает процесс проектирование БД. Кроме того, в PostgreSQL есть инструмент EXPLAIN, с помощью которого удобно вести разработку оптимизированных запросов (в частности, создания индексов).

База данных должна хранить данные, рассмотренные в таблице 1.1. В соответствии с этой таблицей можно выделить следующие сущности в базе данных:

- таблица самолетов **aircrafts**,
- таблица аэропортов **airports**,
- таблица авиакомпаний **airlines**,
- таблица информации о перелетах **delays**.

На рисунке 2.1 приведена ER-диаграмма спроектированной базы данных в нотации Чена [15].

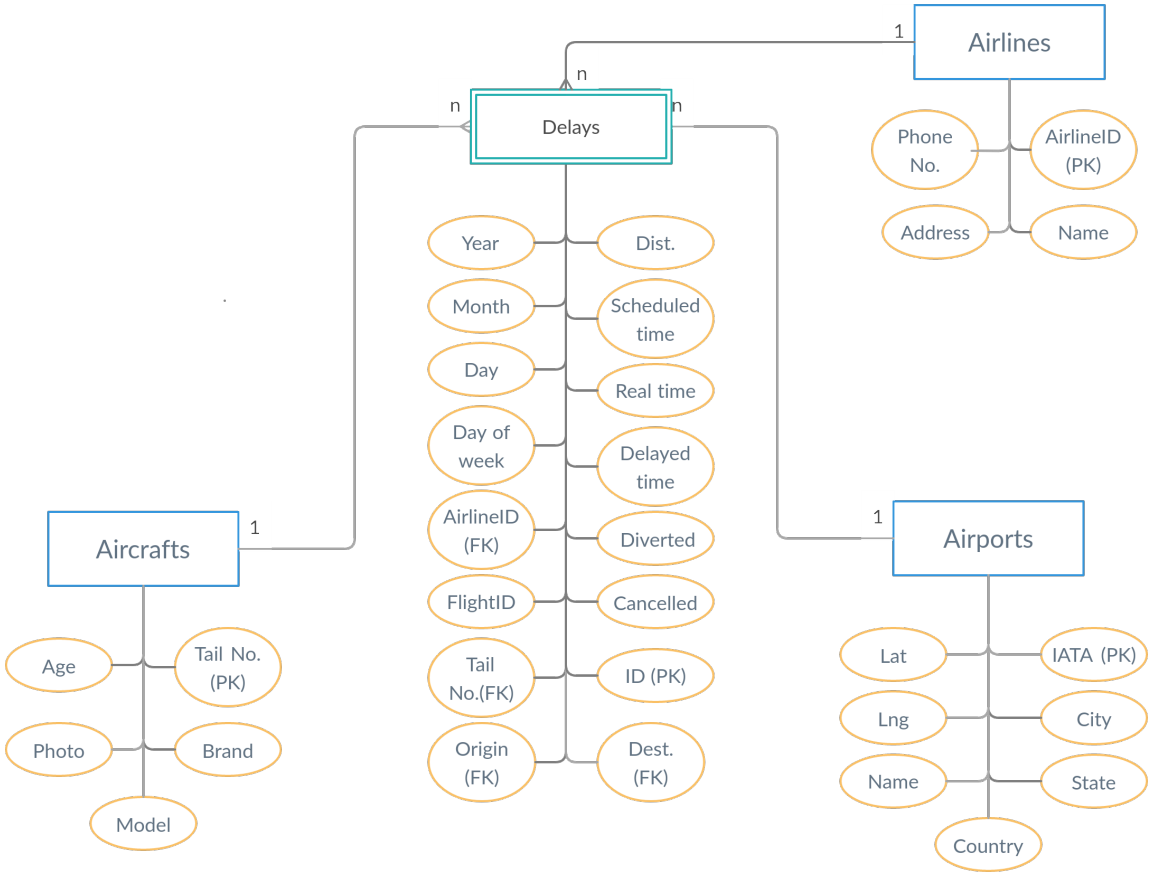

Рисунок 2.1: ER-диаграмма спроектированной БД

Далее приведены поля выделенных таблиц, их тип в СУБД PostgreSQL [5] а также их небольшие описания:

- таблица **aircrafts**:
  - tail_no (PK) - varchar, номер регистрации воздушного судна;
  - mfr - varchar, наименование производителя;
  - model - varchar, полное название модели;

  - bday - date, дата ввода в эксплуатацию;
  - photo - varchar, URL изображения судна;

- таблица **airports**:
  - iata (PK) - varchar, номер аэропорта в международной кодировке;
  - fullname - varchar, полное название аэропорта;
  - city - varchar, город;
  - _state - varchar, штат (область);
  - country - varchar, страна;
  - lat - numeric, широта;
  - lng - numeric, долгота;

- таблица **airlines**:
  - airline_id (PK) - varchar, код авиакомпании, состоящий из двух символов;
  - fullname - varchar, полное название авиакомпании;
  - addr - varchar, адрес главного офиса;
  - phone_no - varchar, контактный номер телефона;

- таблица **delays**:
  - delay_id (PK) - serial, уникальный идентификатор записи в таблице;
  - flight_date - date, дата совершения перелета;
  - day_of_week - integer, номер дня недели, в который был совершен рейс;
  - tail_no (FK) - varchar, номер регистрации воздушного судна;
  - airline_id (FK) - varchar, код авиакомпании, состоящий из двух символов;
  - flight_id - integer, численный код рейса;

– origin (FK) - varchar, номер аэропорта вылета в международной кодировке;
– dest (FK) - varchar, номер аэропорта прилета в международной кодировке;
– dist - numeric, длина маршрута в милях;
– scheduled_time - time, расчетное время в пути;
– real_time - time, фактическое время в пути;
– delayed - bit, флаг, был ли рейс задержан;
– diverted - bit, флаг, был ли рейс переведен в другой аэропорт;
– cancelled - bit, флаг, был ли рейс отменен.

2.3 Проектирование DIA-механизмов

2.3.1 Индексы

Решение о том, какие индексы создать, выносится на основание того, какие будут выполняться, в первую очередь SELECT, запросы. Самым простым, но тем не менее эффективным, решением является создание индексов на primary key (PK) каждой из таблицы. Это оправданное решение, поскольку с помощью инструмента EXPLAIN было установлено, что большинство запросов к БД в разработанной системе выполняют поиск записей именно по PK таблиц. Также было решено создать индекс на поле tail_no таблицы delays, поскольку это относительно частый ключ поиска записей в таблице информации о выполненных рейсах.

2.3.2 Master-Slave репликация

Для реализации Master-Slave реликациии необходимо иметь несколько машин. Для имитации нескольких хостов на одной машине были использованы linux-конйтенеры (LXC) [16]. Всего было создано 3 Slave-реплики с помощью linux-контейнеров.

Настройка реплик подробно рассмотрена в технологическом разделе.

2.3.3 Кэширование

Для реализации кэша запросов была выбрана in-memory key-value СУБД Redis по следующим причинам:

- все данные Redis хранятся в памяти, что обеспечивает низкую задержку и высокую пропускную способность доступа к данным;
- в Redis реализована поддержка разнообразных структур данных, которые, могут быть применены для реализации кэша, в частности.

Чтобы реализовать алгоритм 2Q [12], рассмотренный в аналитической части, в хранилище Redis, было принято решение использовать следующие встроенные структуры данных Redis:

- list (связанный список) - в качестве «теплой» и «горячей» очередей.
- hash table (хэш-таблица) - для хранения результатов выполнения SQL-запросов.

На рисунке 2.2 приведена схема алгоритма 2Q.

В очереди помещаются захэшированные строки (ключи), представляющие собой SQL-выражение. В соответствующие хэш-таблицы по заданному ключу хранится JSON-строка, содержащая результат выполнения запроса-ключа. Важно постараться исключить коллизии в ключах, поэтому было решено использовать алгоритм SHA-256 [18] для хэширования SQL-запросов, поскольку данный алгоритм является детерминированным (для одних и тех же запросов он вернет одинаковый хэш), а также возвращает хэши одного размера (256 бит). Отметим, что на сегодняшний день не было найдено коллизий хэшей SHA-256.

Очереди отличаются по размеру: «теплая» очередь должна быть длиннее «горячей». Точные размеры очередей подбираются, исходя из размеров сохраняемых данных, объема памяти устройства а также нагрузки на сервер. В рамках данного проекта было принято решение использовать «горячую» очередь длиной в 10 элементов, а «теплую» - в 20.

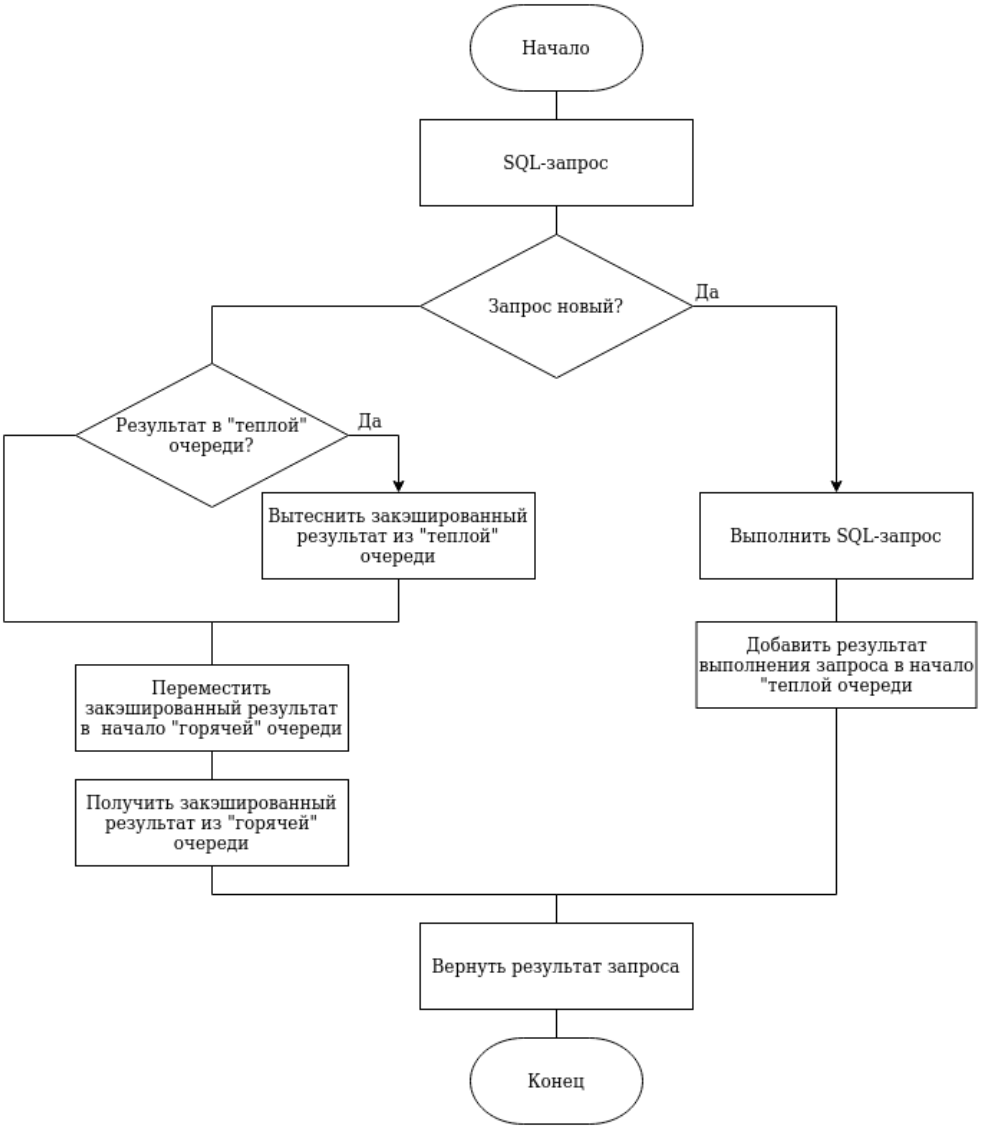

Рисунок 2.2: Схема алгоритма 2Q

За неимением больших вычислительных кластеров, было принято решение запустить сервер с Redis на том же устройстве, что и Master-реплику.

2.4 Проектирование приложения

2.4.1 Структура приложения

Каждый запрос на чтение, в первую очередь, проверяется, нет ли ответа в кэше. Если запрос не был закэширован, тогда запрос поступает на одну из реплик: благодаря хэшированию запроса алгоритмом SHA-256, а

затем взятию от полученного числа модуля, равного количеству реплик, нагрузка между репликами будет балансироваться равномерно. Запросы на изменение обрабатывает только Master-реплика. Изменения распространяются на Slave-реплики с задержкой в 10 минут. Это сделано для того, чтобы в случае возникновения чрезвычайной ситуации (например, случайного удаления всей БД), можно было успеть созранить данные на репликах, сделad одну из нод мастером.

Чтобы выполнить изменение данных, в теле HTTP-запроса необходимо передать токен аутентификации. Если переданный токен не зарегистрирован в системе, то пользователю запрещается вносить изменения.

На рисунке 2.3 приведена схема структуры разработанного приложения.

2.4.2 Ролевая модель

В случае, если по какой-то причине откажет Master, то необходимо повысить одну из Slave-реплик до Master, а старый Master, после починки, сделать Slave. Поскольку пул адресов хостов с БД остается неизменным, но роли реплик могут меняться с течением времени, удобно ввести ролевую модель на уровне БД.

Было реализовано две роли:

- postgres - суперпользователь, имеющий все привилегии в БД;
- guest - пользователь, для которого не требуется аутентификация токеном, для него разрешено только чтение данных.

.

Таким образом, подключение для чтения данных будет осуществляться под ролью guest к любой из реплик, в то время как для создания, удаления, изменения данных - пользователь postgres к Master-реплике.

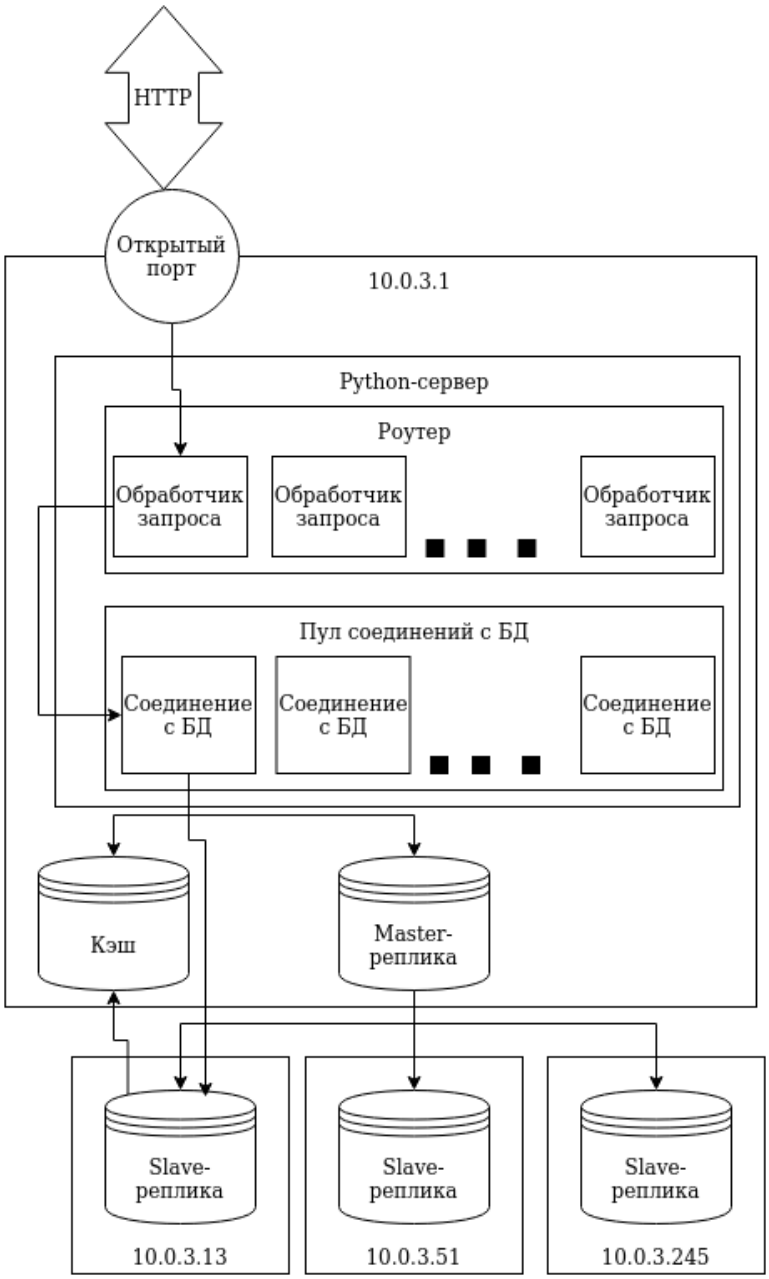

Рисунок 2.3: Структура разработанного приложения

Вывод

На основании выделенных сущностный, было принято решение создать 4 таблицы в базе данных: aircrafts, airports, airlines и delays. Выбранная СУБД - PostgreSQL.

Было решено использовать следующие методы и механизмы DIA:

- создать индексы было решено для PK таблиц БД, а так же для поля tail_no таблицы delays;

- lxc-контейнеры для создания Slave-реплик БД: всего было создано 3 Slave-реплики;

- для реализации алгоритма кэширования 2Q решено использовать встроенные структуры данных Redis: связанные списки и хэш-таблицы. Для генерации ключа-хэша используется алгоритм SHA-256.

В спроектированном приложении были выделены две роли на уровне БД: postgres (с правами на чтение и изменение данных) и guest (с правами только на чтение).

3 Технологическая часть

В данном разделе представлены средства разработки программного обеспечения, а также детали реализации.

3.1 Выбор технических средств

Для взаимодействия серверного программного обеспечения с СУБД былы выбрана технология object–relational mapping (ORM). Она обеспечивает работу с данными в терминах классов, а не таблиц данных, что ускоряет процесс разработки. Использование ORM позволяет в будущем легко подменять СУБД, не перписывая код, что является весомым аргументом в пользу использования этой технологии. Помимо всего выше перечисленного, ORMs используют параметризованные SQL-выражения, который защищают от прямых атак SQL-инъекциями.

В качестве языка разработки выбран язык программирования Python 3 [19]. Данный выбор обусловлен простотой использования, скоростью написания кода и большим количеством всевозможных библиотек, необходимых для разработки ПО. Редактор кода - VS Code [20]. Выбор среды обусловлен большим количеством доступных плагинов платформы, которые существенно облегчают и ускоряют процесс написание кода.

В качестве Web-framework был выбран Flask [21], поскольку он предлагает широкий функционал для написания веб-приложений на языке Python 3.

Для проведения нагрузочного тестирования была выбрана библиотека Locust [22]. Этот выбор обусловлен тем, что данная библиотека предоставляет удобный и исчерпывающий отчет проведенного тестирования, а так же визуализирует полученные результаты.

3.2 Детали реализации

3.2.1 Взаимодействие с разработанным приложением

В таблице 3.1 приведен список реализованных HTTP-запросов для взаимодействия с базой данных.

Таблица 3.1: Разработанные запросы к базе данных

HTTP-метод	**Действие**	**URL-путь**
GET	Получить информацию о самолете(-ах)	/api/aircrafts
POST	Добавить запись о самолете	/api/aircrafts
PUT	Изменить запись о самолете	/api/aircrafts
DELETE	Удалить запись о самолете	/api/aircrafts
GET	Получить информацию об аэропорте(-ах)	/api/airports
POST	Добавить запись об аэропорте	/api/airports
PUT	Изменить запись об аэропорте	/api/airports
DELETE	Удалить запись об аэропорте	/api/airports
GET	Получить информацию об авиакомпании(-ях)	/api/airlines
POST	Добавить запись об авиакомпании	/api/airlines
PUT	Изменить запись об авиакомпании	/api/airlines
DELETE	Удалить запись об авиакомпании	/api/airlines
GET	Получить информацию о перелетах(-ах)	/api/flights
POST	Добавить запись о перелете	/api/flights
PUT	Изменить запись о перелете	/api/flights
DELETE	Удалить запись о перелете	/api/flights
GET	Получить информацию об авиакомпании-операторе воздушного судна	/api/ac-operator
GET	Получить среднее время задержки рейсов заданной авиакомпании	/api/avg-time

Для GET-запросов предусмотрены следующие аргументы:

- /api/aircrafts - для выборки самолетов, удовлетворяющих следующим параметрам:
  - tail-no - номер регистрации воздушного судна;
  - mfr - компания-производитель;
  - model - модель самолета;

  - younger (older) - поступил в эксплуатацию после (до) указанной даты (в формате ГГГГ-ММ-ДД);
  - limit - число возвращенных записей (по умолчанию 10);

- /api/airports - для выборки аэропортов, удовлетворяющих следующим параметрам:
  - iata - трехсимвольный код аэропорта в международной кодификации;
  - city, state, country - город, штат (область) и страна, соответственно, в которой находится аэропорт;
  - limit - число возвращенных записей (по умолчанию 10);

- /api/airlines - для выборки авиакомпаний, удовлетворяющих следующим параметрам:
  - id - двухсимвольный код авиакомпании в международной кодификации;
  - limit - число возвращенных записей (по умолчанию 10);

- /api/flights - для выборки перелетов, удовлетворяющих следующим параметрам:
  - id - уникальный идентификатор записи;
  - flightid - номер рейса;
  - date - дата совершения перелета (в формате ГГГГ-ММ-ДД);
  - dow - день недели (воскресенье - 0, суббота - 6);
  - tail-no - номер регистрации воздушного судна, выполнявшего рейс;
  - from, to - код аэропорта вылета и прилета, соответственно;
  - limit - число возвращенных записей (по умолчанию 10);

- /api/ac-operator - для получения информации об операторе заданного самолета:
  - tail-no - номер регистрации воздушного судна;

- /api/avg-time - для получения информации о среднем времени задержок рейсов заданной авиакомпании:
  - id - двухсимвольный код авиакомпании в международной кодификации.

Запросы на сервер можно посылать как из браузера (URL), так и с помощью специальных утилит терминала (например, curl). Ответ от сервера приходит в виде JSON-объекта. Поле «ok» отражает, возникла ли ошибка при обработке запроса, а в поле «result» - непосредственно содержимое ответа.

3.2.2 Листинги скриптов и подпрограмм

Для реализации Master-Slave репликации на каждый из контейнеров была установлена ОС Ubuntu [17] и актуальная (такая же как и на Master-реплике) версия PostgreSQL [5]. Адреса хостов в локальной сети были получены с помощью утилиты ifconfig. В приложении на листингах 4.1 и 4.2 приведена настройка Master и Slave-реплик базы данных, путем редактирования системных файлов-конфигураторов СУБД (postgresql.conf и pg_hba.conf, в частности).

На листингах 4.3-4.5 в приложении приведены реализации механизмов масштабирования БД, рассмотренных в конструкторском разделе:

- на листинге 4.3 - балансировка запросов между репликами БД;
- на листинге 4.4 - SQL-скрипт создания индексов в БД;
- на листинге 4.5 - реализация алгоритма 2Q с помощью средств СУБД Redis.

Для GET-запроса /api/ac-operator была реализована функция на языке plpgsql. Реализации функции на языке plpgsql приведена в приложении на листинге 4.6.

В приложении на листинге 4.7 приведена реализации рассмотренной в конструкторском разделе ролевой модели на уровне БД. Роль postgres - созданная СУБД роль по умолчанию никак не была изменена.

Вывод

В данном разделе были подробно рассмотрены технические средства для написания программного обеспечения, а также детали реализации ПО (интерфейс взаимодействия, листинги скриптов и подпрограмм).

4 Исследовательская часть

В данном разделе представлены примеры работы программного обеспечения, а также описан эксперимент по проведению нагрузочного тестирования разработанной системы.

4.1 Пример работы программного обеспечения

На рисунках 4.1-4.4 приведены примеры использования разработанного приложения.

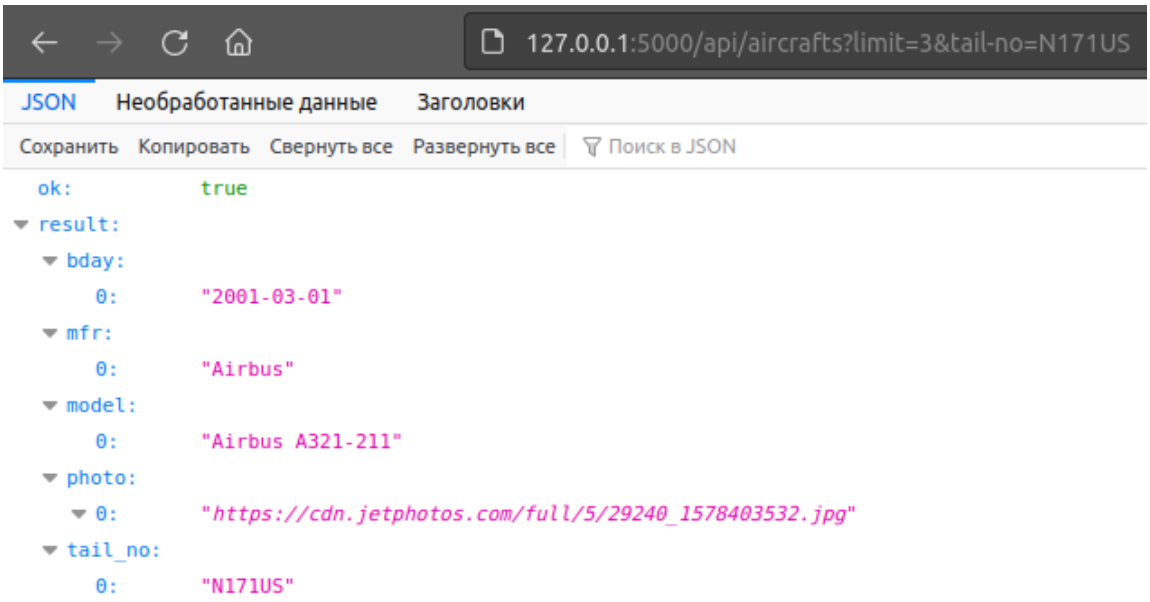

Рисунок 4.1: Пример запроса на чтение

4.2 Постановка эксперимента

Целью эксперимента является проведение нагрузочного тестирования разработанной системы. Критерием оценки выступили результаты аналогичного тестирования схожей системы, но в которой не реализован ни один из методов масштабирования БД.

Технические характеристики ЭВМ, на котором выполнялись исследования:

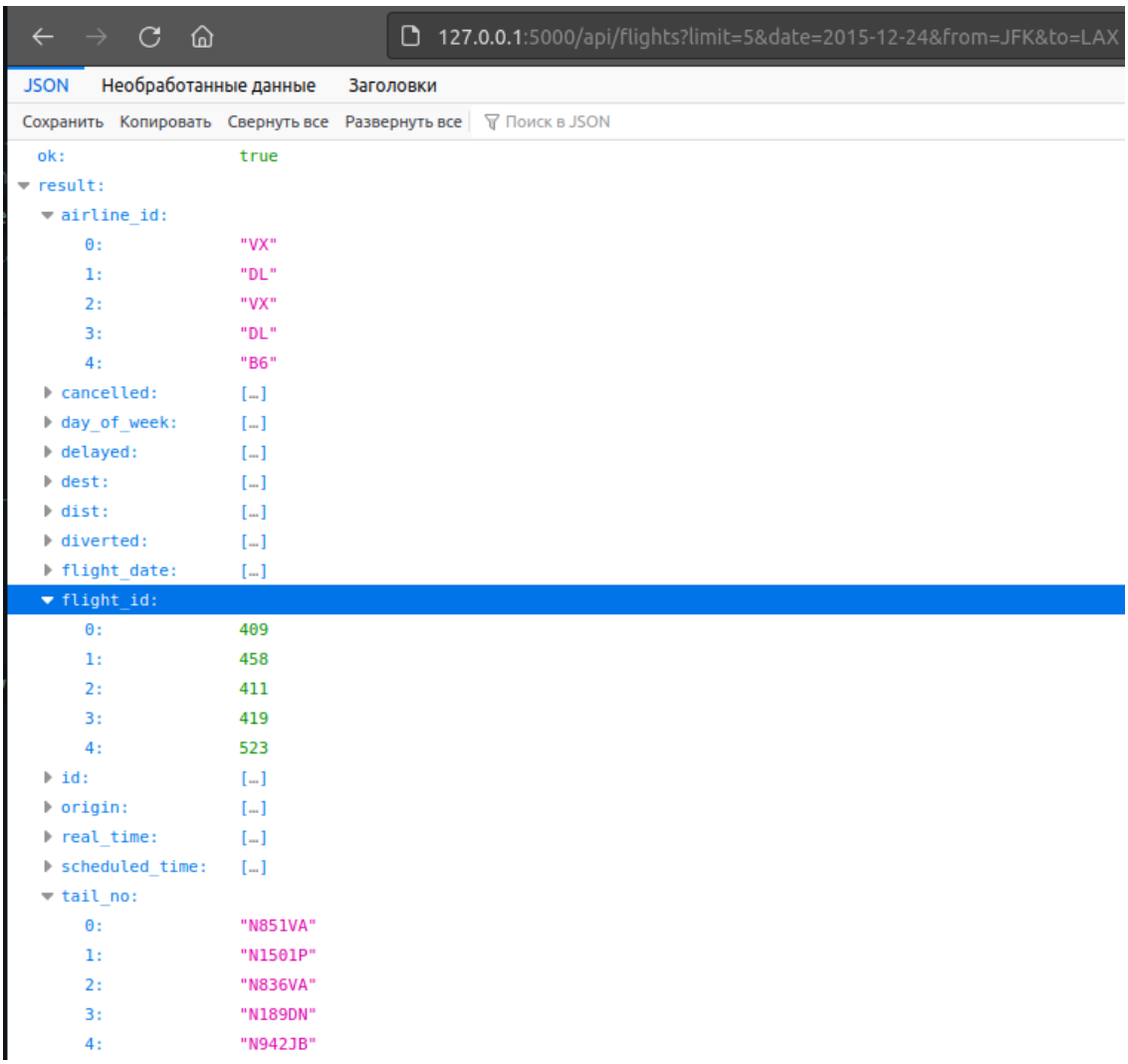

Рисунок 4.2: Пример запроса на чтение

```
❯ curl -X DELETE -H "Content-Type: application/json" \
    -d '{ "id": 1 }' \
    http://127.0.0.1:5000/api/flights
{"ok":false,"result":"401 - unauthorized"}
❯ curl -X DELETE -H "Content-Type: application/json" \
    -d '{"token": "wrong-token",  "id": 1}' \
    http://127.0.0.1:5000/api/flights
{"ok":false,"result":"403 - forbidden"}
```

Рисунок 4.3: Пример запросов на неаутентифицированное изменение данных

- процесор Intel Core i5-8250U 1.6ГГц (8 ядер);
- 8 Гб оперативной памяти;
- ОС Linux Mint 19.3.

Рисунок 4.4: Пример запроса на аутентифицированное изменение данных

В ходе проведения эксперимента ЭВМ была подключена к сети электропитания.

Эксперимент был поставлен по следующему сценарию:

- был создано два типа подключений: гость (guest), который имел большой набор разнообразных запросов на чтение, причем некоторые запросы отправлялись чаще других; и администратор (admin), который создает и изменяет записи в базе данных. 90% всех запросов составляют запросы гостя и, соответственно, 10% - администратора;
- каждую секунду количество подключений увеличивается на 10, пиковая нагрузка - 1000 пользователей.

В таблице 4.1 приведены агрегированные результаты системы с и без применения методов DIA.

Таблица 4.1: Агрегированные результаты тестирования

	Всего запросов отправлено	Число ошибок	Среднее время отклика (мс)	RPS	Failures/s
Базовая система	2728	1902	14055	23.3	16.3
Система DIA	5670	463	5713	49.9	4.1

В таблице 4.2 приведены результаты измерения времени отклика системы.

На рисунках 4.5 и 4.6 приведены графики результатов проведенного тестирования базовой системы, и системы с применением методов DIA, соответственно.

Относительно большое число ошибок связано со вычислительной мощностью машины, на которой проводилось тестирвоание, поскольку на одной ЭВМ были запущены одновременно Python-сервер, сервер Redis, три реплики БД, а так же тестирующее ПО.

.

Таблица 4.2: Время отклика системы (мс)

	50% процентиль	80% процентиль	90% процентиль	95% процентиль	100% процентиль
Базовая система	8300	16000	50000	60000	75000
Система DIA	3000	5800	10000	21000	71000

Применение методов DIA позволило увлечить RPS системы более чем в два раза, уменьшить в 2.5 раза среднее время отклика и в 4 раза число ошибок в секунду - что свидетельствует о том, что разработанная система эффективнее справляется с относительно большой нагрузкой.

Как следует из полученных графиков, число ошибок в секунду в базовой системе было примерно на уровне RPS, что означает, почти каждый новый запрос в систему возвращал ошибку. В то время как число ошибок DIA-сервиса держалось на одном уровне по мере увеличения нагрузки.

Вывод

В результате сравнения разработанной с применением методов и механизмов масштабирования БД системы с базовой, была доказана эффективность примененных решений для создания DIA.

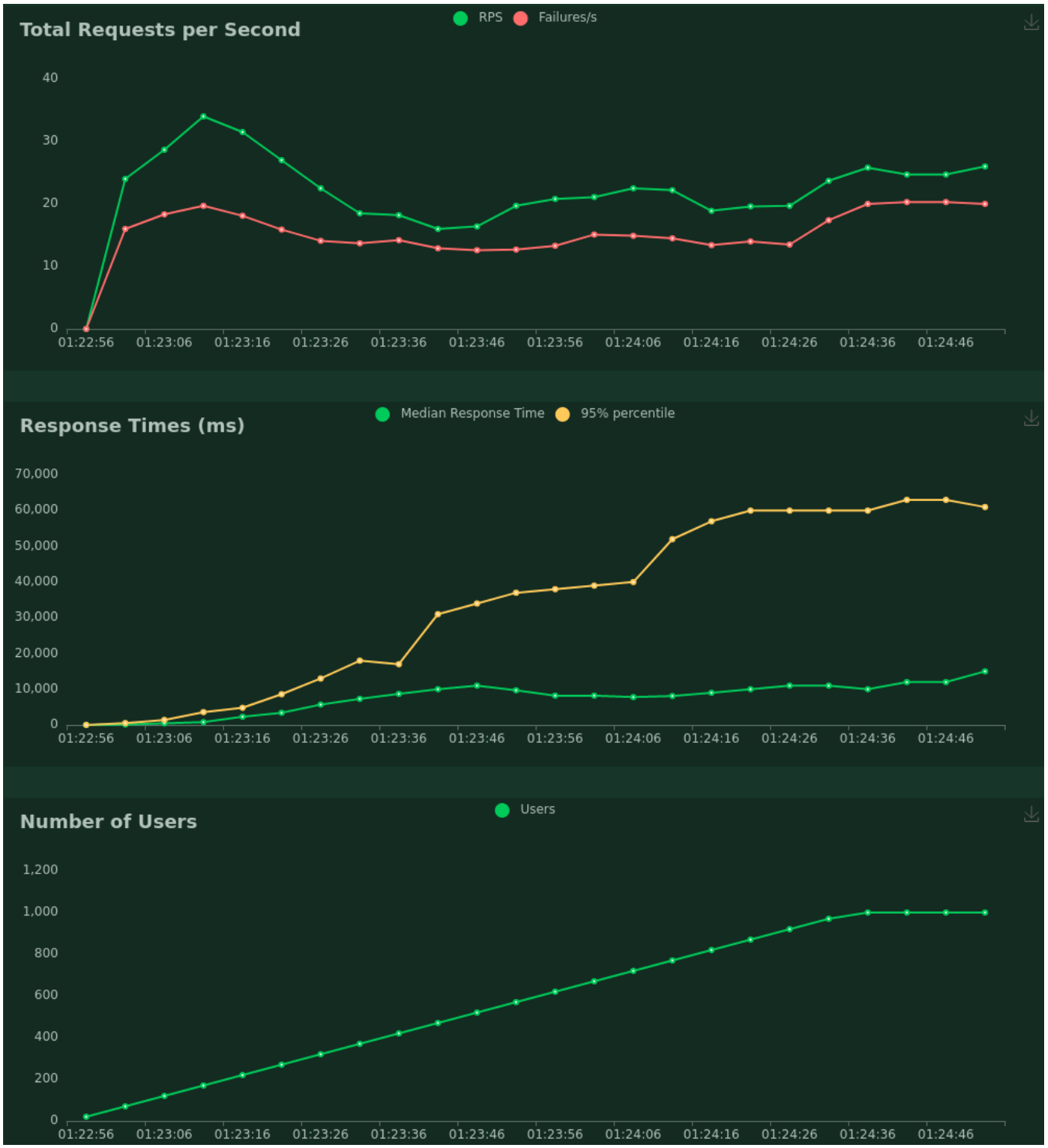

Рисунок 4.5: Графики, построенные по результатам тестирования базовой системы

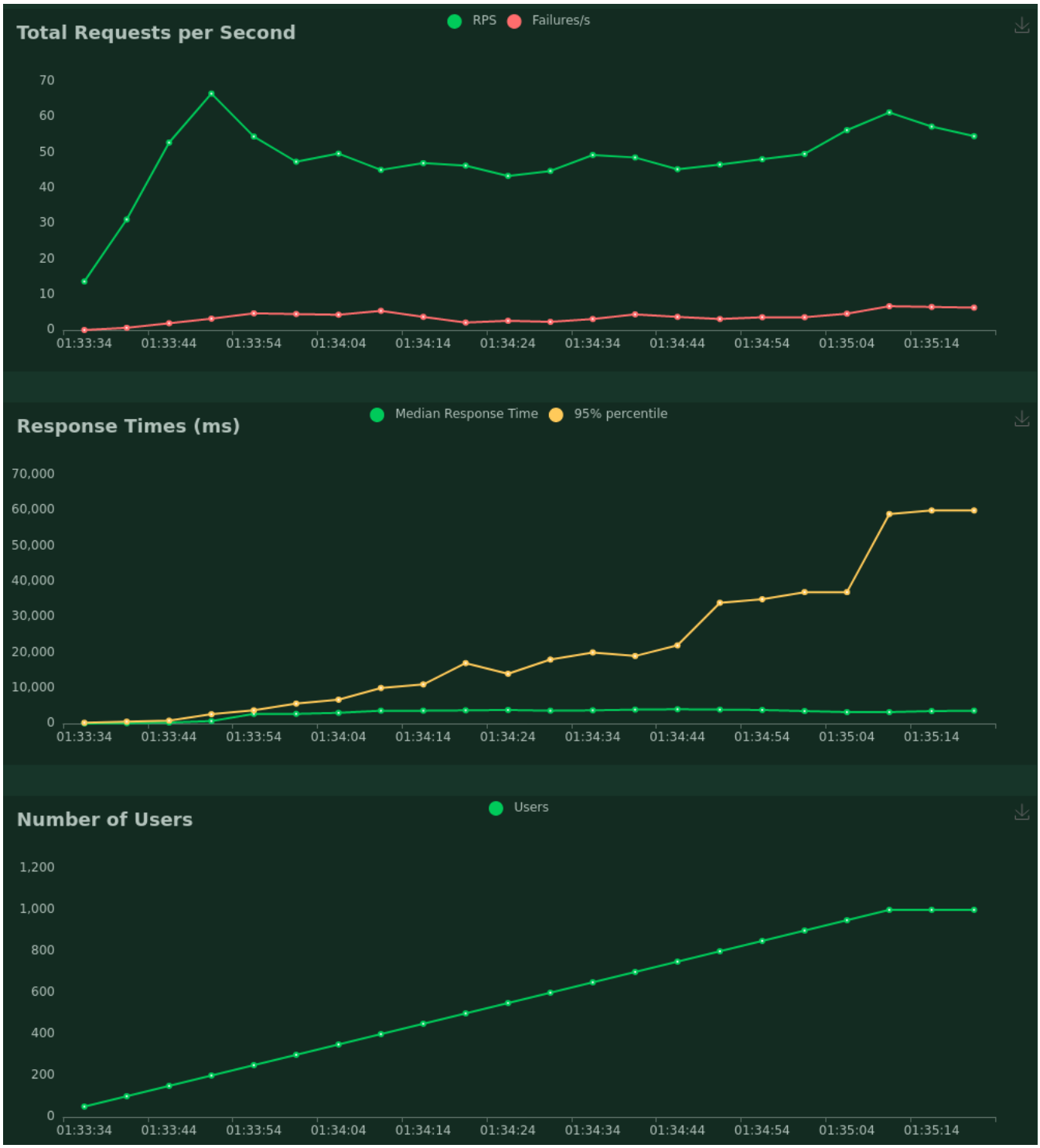

Рисунок 4.6: Графики, построенные по результатам тестирования DIA-системы

Заключение

В ходе выполнения курсового проекта было смоделирована высоконагруженная система с применением методов и механизмов масштабирования базы данных на примере сервиса, выдающего информацию о авиарейсах.

Во время выполнения поставленной задачи были получены знания в области проектирования и реализации высоко нагруженных данными систем: были изучены новые технологии и алгоритмы, особенности их применения и имплементации. Поиск материала по теме позволил повысить навыки поиска и анализа информации.

В результате проделанной работы был разработан программный продукт, в котором были успешно применены методы DIA, и который в дальнейшем возможно расширить и масштабировать для воплощения различных проектов (например, агрегатора авиабилетов).

В ходе выполнения экспериментально-исследовательской части было эмпирическим путем установлено, что разработанная система эффективнее справляется с нагрузкой, нежели базовая, недоработанное ПО.

Литература

[1] Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, Inc., 2017. P. 14–15.

[2] Automated Example Oriented REST API. Режим доступа: https://ase.cpsc.ucalgary.ca/wp-content/uploads/2018/05/Automated-Example-Oriented-REST-API.pdf. Дата обращения: 01.10.2021.

[3] ISO/IEC TR 10032:2003 Information technology — Reference Model of Data Management. Режим доступа: https://www.iso.org/standard/38607.html. Дата обращения: 01.10.2021.

[4] Дж Дейт К. Введение в системы баз данных. «Вильямс», 2006.

[5] PostgreSQL: The World's Most Advanced Open Source Relational Database. Режим доступа: https://www.postgresql.org/. Дата обращения: 01.10.2021.

[6] Oracle Database. Режим доступа: https://www.oracle.com/database/. Дата обращения: 01.10.2021.

[7] MS SQL Server 2019. Режим доступа: https://www.microsoft.com/en-us/sql-server/sql-server-2019. Дата обращения: 01.10.2021.

[8] PostgreSQL Indexes. Режим доступа: https://www.postgresql.org/docs/13/indexes.html. Дата обращения: 01.10.2021.

[9] Redis Documentation. Режим доступа: https://redis.io/documentation. Дата обращения: 01.10.2021.

[10] LevelDB Documentation. Режим доступа: https://github.com/google/leveldb/blob/master/doc/index.md. Дата обращения: 01.10.2021.

[11] An Evaluation of Buffer Management Strategies for Relational Database Systems. Режим доступа: http://www.vldb.org/conf/1985/P127.PDF. Дата обращения: 01.10.2021.

[12] 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. Режим доступа: http://www.vldb.org/conf/1994/P439.PDF. Дата обращения: 01.10.2021.

[13] IMPLEMENTING A MASTER/SLAVE ARCHITECTURE FOR A DATA SYNCHRONIZATION SERVICE. Режим доступа: https://www.theseus.fi/bitstream/handle/10024/143791/Nikolay_Baychenko.pdf. Дата обращения: 01.10.2021.

[14] Sharding by Hash Partitioning. A database scalability pattern to achieve evenly sharded database clusters. Режим доступа: https://www.researchgate.net/publication/276075813_Sharding_by_Hash_Partitioning_-_A_Database_Scalability_Pattern_to_Achieve_Evenly_Sharded_Database_Clusters. Дата обращения: 01.10.2021.

[15] The Entity-Relationship Model–Toward a Unified View of Data. Режим доступа: https://dspace.mit.edu/bitstream/handle/1721.1/47432/entityrelationshx00chen.pdf. Дата обращения: 01.10.2021.

[16] LXC - Documentation. Режим доступа: https://linuxcontainers.org/lxc/documentation/. Дата обращения: 01.10.2021.

[17] OS Ubuntu. Режим доступа: https://ubuntu.com/. Дата обращения: 01.10.2021.

[18] Verification of a Cryptographic Primitive: SHA-256. Режим доступа: https://www.cs.princeton.edu/~appel/papers/verif-sha.pdf. Дата обращения: 01.10.2021.

[19] Python 3 language official page. Режим доступа: https://www.python.org/. Дата обращения: 01.10.2021.

[20] Редактор кода VS Code. Режим доступа: https://code.visualstudio.com/. Дата обращения: 01.10.2021.

[21] Flask's documentation. Режим доступа: https://flask.palletsprojects.com/en/2.0.x/. Дата обращения: 01.10.2021.

[22] Locust - A modern load testing framework. Режим доступа: https://locust.io/. Дата обращения: 01.10.2021.

Приложение

```
sudo vim /etc/postgresql/13/main/pg_hba.conf

<...>
host replication postgres 10.0.3.0/24 md5
host all postgres 10.0.3.0/24 md5
<...>

########################################################################

sudo vim /etc/postgresql/13/main/postgresql.conf

<...>
listen_addresses = 'localhost, 10.0.3.1'

wal_level = hot_standby

wal_log_hints = on

max_wal_senders = 8
wal_keep_segments = 64

hot_standby = on
<...>

########################################################################

sudo service postgresql restart
```

Листинг 4.1: Листинг настройки Master-реплики БД

```
sudo service postgresql stop
sudo -u postgres

cd /var/lib/postgresql/13/
tar -cvzf main_backup-`date +%s`.tgz main
rm -rf main
mkdir main
chmod go-rwx main
pg_basebackup -P -R -X stream -c fast -h 10.0.3.1 -U postgres -D ./main

########################################################################

sudo vim /etc/postgresql/13/main/postgresql.conf

<...>
recovery_target_timeline = 'latest'
```

```
17
18 recovery_min_apply_delay = 10min
19 <...>
20
21 ################################################################
22
23 sudo service postgresql start
```

Листинг 4.2: Листинг настройки Slave-реплики БД

```
1  import os.path as path
2  from hashlib import sha256
3
4  import peewee as orm
5  from flask import request
6
7  from .loader import load_config
8
9
10 YAML_CONFIG_MASTER = path.abspath("flights_data/data/conn_master.yaml")
11 YAML_CONFIG_SLAVE = path.abspath("flights_data/data/conn_slave.yaml")
12 YAML_AUTHORIZED_USERS = path.abspath("flights_data/data/authorized.yaml")
13
14
15 def connect_db(request_path):
16     params = None
17     if request.method == "GET":
18         params = load_config(YAML_CONFIG_SLAVE)
19         hash =
               sha256(f"{request_path}{random()}".encode(encoding="utf-8")).digest()[0]
20         params["host"] = params["host"][hash % len(params["host"])]
21     else:
22         params = load_config(YAML_CONFIG_MASTER)
23
24     return orm.PostgresqlDatabase(**params)
```

Листинг 4.3: Листинг балансировки нагрузки между репликами

```
1  \c db_delays;
2
3  CREATE INDEX IF NOT EXISTS idx_ac_tailno
4  ON aircrafts(
5      tail_no
6  );
7
8
9  CREATE INDEX IF NOT EXISTS idx_al_id
10 ON airlines(
11     airline_id
```

```
);


CREATE INDEX IF NOT EXISTS idx_ap_iata
ON airports(
   iata
);


CREATE INDEX IF NOT EXISTS idx_fl_id
ON delays(
   delay_id
);
CREATE INDEX IF NOT EXISTS idx_fl_tailno
ON delays(
   tail_no
);
```

Листинг 4.4: Листинг создания индексов БД

```
import hashlib
import redis

from flask import request, make_response, jsonify
import json


r = redis.Redis(host='localhost', port=6379)


LRU_CACHE = "hot"
HLRU_CACHE = "h:hot"
LRU_SIZE = 10

WARM_CACHE = "warm"
HWARM_CACHE = "h:warm"
WARM_SIZE = 20


def cache(func):
   def _serialize(json_string):
      obj = json.loads(json_string)
      if obj is None:
         obj = None
      return obj

   def _in_cache(element, cache):
      return element in r.lrange(cache, 0, -1)

```

```
    def _clean_htable(cache, table):
        elements = None
        hkeys = None
        with r.pipeline(transaction=True) as pipe:
            pipe.multi()
            pipe.lrange(cache, 0, -1)
            pipe.hkeys(table)
            elements, hkeys = pipe.execute()
            to_delete = list(filter(lambda x: x not in elements, hkeys)) or
                ["stub"]

            pipe.multi()
            pipe.hdel(table, *to_delete)
            pipe.execute()

    def wrapper():
        if request.method != "GET":
            return func()

        hashed_query =
            hashlib.sha256(request.full_path.encode(encoding="utf-8")).digest()
        response = None

        with r.pipeline(transaction=True) as pipe:
            # If in the LRU cache
            if _in_cache(hashed_query, LRU_CACHE):
                pipe.multi()
                pipe.lrem(LRU_CACHE, 1, hashed_query)
                pipe.lpush(LRU_CACHE, hashed_query)
                pipe.ltrim(LRU_CACHE, 0, LRU_SIZE - 1)
                _clean_htable(LRU_CACHE, HLRU_CACHE)
                pipe.hget(HLRU_CACHE, hashed_query)
                result = pipe.execute()[3]
                response = make_response(jsonify({ "ok": True, 'result':
                    _serialize(result.decode()) }))
            # If in the WARM QUEUE
            elif _in_cache(hashed_query, WARM_CACHE):
                pipe.multi()
                pipe.lrem(WARM_CACHE, 1, hashed_query)
                pipe.lpush(LRU_CACHE, hashed_query)
                pipe.ltrim(LRU_CACHE, 0, LRU_SIZE - 1)
                _clean_htable(LRU_CACHE, HLRU_CACHE)
                pipe.hget(HWARM_CACHE, hashed_query)
                result = pipe.execute()[3]
                response = make_response(jsonify({ "ok": True, 'result':
                    _serialize(result.decode()) }))

                pipe.multi()
```

```
74              pipe.hdel(HWARM_CACHE, hashed_query)
75              pipe.hset(HLRU_CACHE, hashed_query, result)
76              pipe.execute()
77          # First ocasion
78          else:
79              pipe.multi()
80              pipe.lpush(WARM_CACHE, hashed_query)
81              pipe.ltrim(WARM_CACHE, 0, WARM_SIZE - 1)
82              _clean_htable(WARM_CACHE, HWARM_CACHE)
83              response = func()
84              temp = json.loads(response.response[0].decode())["result"]
85              pipe.hset(HWARM_CACHE, hashed_query, json.dumps(temp).encode())
86              pipe.execute()
87
88      return response
89
90  wrapper.__name__ = func.__name__
91  return wrapper
```

Листинг 4.5: Листинг алгоритма 2Q

```
1  \c db_delays;
2
3  CREATE OR REPLACE FUNCTION get_ac_operator(key_tailno VARCHAR(8))
4      RETURNS TABLE(
5          airline_id VARCHAR(2),
6          fullname VARCHAR,
7          tail_no VARCHAR(8),
8          mfr VARCHAR,
9          model VARCHAR,
10         photo VARCHAR
11     )
12     LANGUAGE plpgsql
13 AS
14 $$
15 BEGIN
16     RETURN QUERY
17         SELECT DISTINCT al.airline_id, al.fullname, ac.tail_no, ac.mfr,
               ac.model, ac.photo
18         FROM delays d JOIN aircrafts ac ON d.tail_no = ac.tail_no
19             JOIN airlines al on d.airline_id = al.airline_id
20         WHERE d.tail_no = key_tailno;
21 END;
22 $$;
```

Листинг 4.6: Листинг реализации функции get_ac_operator()

```
1  CREATE USER guest WITH PASSWORD 'guest';
2
```

```
3 GRANT CONNECT ON DATABASE db_delays TO guest;
4 GRANT USAGE ON SCHEMA public TO guest;
5
6 GRANT SELECT ON ALL TABLES IN SCHEMA public TO guest;
7 GRANT EXECUTE ON FUNCTION get_ac_operator TO guest;
```

Листинг 4.7: Листинг реализации ролевой модели на уровне БД