

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
GRADUATION THESIS**

Автоматизация индексирования базы данных на основе истории запросов

Обучающийся / Student Кульбако Артемий Юрьевич

Факультет/институт/кластер/ Faculty/Institute/Cluster факультет программной инженерии и компьютерной техники

Группа/Group Р34112

Направление подготовки/ Subject area 09.03.04 Программная инженерия

Образовательная программа / Educational program Системное и прикладное программное обеспечение 2018

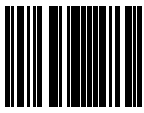
Язык реализации ОП / Language of the educational program Русский

Статус ОП / Status of educational program

Квалификация/ Degree level Бакалавр

Руководитель ВКР/ Thesis supervisor Гаврилов Антон Валерьевич, Университет ИТМО, факультет программной инженерии и компьютерной техники, преподаватель (квалификационная категория "преподаватель")

Обучающийся/Student

Документ подписан	
Кульбако Артемий Юрьевич	
20.05.2022	

(эл. подпись/ signature)

Кульбако
Артемий
Юрьевич

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Гаврилов Антон Валерьевич	
20.05.2022	

(эл. подпись/ signature)

Гаврилов Антон
Валерьевич

(Фамилия И.О./ name
and surname)

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /
OBJECTIVES FOR A GRADUATION THESIS**

Обучающийся / Student Кульбако Артемий Юрьевич
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет программной инженерии и компьютерной техники
Группа/Group P34112
Направление подготовки/ Subject area 09.03.04 Программная инженерия
Образовательная программа / Educational program Системное и прикладное программное обеспечение 2018
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Бакалавр
Тема ВКР/ Thesis topic Автоматизация индексирования базы данных на основе истории запросов
Руководитель ВКР/ Thesis supervisor Гаврилов Антон Валерьевич, Университет ИТМО, факультет программной инженерии и компьютерной техники, преподаватель (квалификационная категория "преподаватель")

Основные вопросы, подлежащие разработке / Key issues to be analyzed

Разработать систему автоматизированного индексирования базы данных на основе истории запросов.

Цели разрабатываемого в рамках ВКР ПО: минимизировать время, затрачиваемое на оптимизацию баз данных и максимально увеличить скорость выполнения запросов к ней, так как существует проблема, что старые базы данных могут содержать очень сложные и/или непонятные запросы из десятка таблиц и сотен полей, скорость выполнения которых бывает необходимо увеличить. В случае, если таких запросов много, возможность ручной оптимизации и рефакторинга займёт много времени и напрямую зависит от опыта работы сотрудника с данной даталогической моделью базы данных. Приложение должно работать в Windows, macOS, Linux, архитектурах x86-64, ARM.

Документами, на основании которых производится разработка, служат:

1. Документация языка программирования Kotlin (на нём ведётся разработка ПО) [Электронный ресурс] – URL: <https://kotlinlang.org/docs/home.html>
2. Документация базы данных PostgreSQL (выступает в качестве первой поддерживаемой базой данных) [Электронный ресурс] – URL: <https://www.postgresql.org/docs/>
3. Документация Redis (используется для хранения общего состояния между несколькими экземплярами основного ПО) [Электронный ресурс] – URL: <https://redis.io/docs/>

Форма представления материалов ВКР / Format(s) of thesis materials:

Отчет о выполнении работы в формате PDF, содержащий текст, рисунки, схемы и программный код

Дата выдачи задания / Assignment issued on: 01.02.2022

Срок представления готовой ВКР / Deadline for final edition of the thesis 30.05.2022

Характеристика темы ВКР / Description of thesis subject (topic)

Тема в области фундаментальных исследований / Subject of fundamental research: нет / not

Тема в области прикладных исследований / Subject of applied research: да / yes

СОГЛАСОВАНО / AGREED:

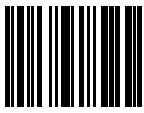
Руководитель ВКР/
Thesis supervisor

Документ подписан	
Гаврилов Антон Валерьевич	
13.05.2022	

(эл. подпись)

Гаврилов Антон
Валерьевич

Задание принял к
исполнению/ Objectives
assumed BY

Документ подписан	
Кульбако Артемий Юрьевич	
13.05.2022	

(эл. подпись)

Кульбако
Артемий
Юрьевич

Руководитель ОП/ Head
of educational program

Документ подписан	
Дергачев Андрей Михайлович	
15.05.2022	

(эл. подпись)

Дергачев
Андрей
Михайлович

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
SUMMARY OF A GRADUATION THESIS**

Обучающийся / Student Кульбако Артемий Юрьевич

Факультет/институт/кластер/ Faculty/Institute/Cluster факультет программной инженерии и компьютерной техники

Группа/Group P34112

Направление подготовки/ Subject area 09.03.04 Программная инженерия

Образовательная программа / Educational program Системное и прикладное программное обеспечение 2018

Язык реализации ОП / Language of the educational program Русский

Статус ОП / Status of educational program

Квалификация/ Degree level Бакалавр

Тема ВКР/ Thesis topic Автоматизация индексирования базы данных на основе истории запросов

Руководитель ВКР/ Thesis supervisor Гаврилов Антон Валерьевич, Университет ИТМО, факультет программной инженерии и компьютерной техники, преподаватель (квалификационная категория "преподаватель")

**ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
DESCRIPTION OF THE GRADUATION THESIS**

Цель исследования / Research goal

Целью работы является автоматизация процесса ускорения баз данных, что освободит администраторов и программистов от решения большей части задач по оптимизации, даст им дополнительное время на ускорении наиболее критичных и важных запросов вручную.

Задачи, решаемые в ВКР / Research tasks

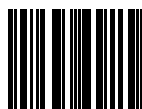
1. Обзор существующих решений. 2. Разработка требований, предъявляемых к создаваемому приложению. 3. Выбор средств реализации. 4. Проектирование и создание тестовой базы данных. 5. Проектирование и создание приложения. 6. Тестирование приложения. 7. Анализ полученных результатов.

Краткая характеристика полученных результатов / Short summary of results/findings

Приложение на Kotlin-React, работающее на ОС Windows, Linux, macOS, архитектурах x86-64, arm, в полной мере соответствующее функциональным и нефункциональным требованиям, способное: 1. Автоматически индексировать базу данных. 2. Формировать отчёты по результатам индексации. К преимуществам системы можно отнести: 1. Расширяемую архитектуру. 2. Поддержку нескольких форматов отчётов и способов их получения. 3. Понятный и прозрачный процесс тестирования БД на поиск наилучшего индекса.

Обучающийся/Student

Документ подписан
Кульбако Артемий Юрьевич
20.05.2022



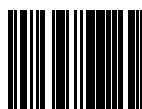
Кульбако
Артемий
Юрьевич

(эл. подпись/ signature)

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан
Гаврилов Антон Валерьевич
20.05.2022



Гаврилов Антон
Валерьевич

(эл. подпись/ signature)

(Фамилия И.О./ name
and surname)

ОГЛАВЛЕНИЕ

ОГЛАВЛЕНИЕ	6
СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ	7
ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ	8
ВВЕДЕНИЕ	10
1. ПОДГОТОВИТЕЛЬНЫЙ ЭТАП	12
1.1 ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ	12
1.2 ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ	13
1.3 ОБОСНОВАНИЕ ИСПОЛЬЗУЕМЫХ ТЕХНОЛОГИЙ	15
2. ЭТАП ПРОЕКТИРОВАНИЯ	19
2.1 ПРОЕКТИРОВАНИЕ СТРУКТУРЫ И ПРОЦЕССОВ	19
2.2 ПРОЕКТИРОВАНИЕ API	20
2.3 ПРОЕКТИРОВАНИЕ ТЕСТОВОЙ БАЗЫ ДАННЫХ	24
3. ЭТАП РАЗРАБОТКИ	28
3.1 СОЗДАНИЕ РАСШИРЯЕМОЙ АРХИТЕКТУРЫ	28
3.2 ДЕКЛАРАТИВНАЯ ОРГАНИЗАЦИЯ КОДА	30
3.3 ОБРАБОТКА ИСТОРИИ SQL-ЗАПРОСОВ	31
3.4 ГЕНЕРАЦИЯ ВЫРАЖЕНИЙ СОЗДАНИЯ ИНДЕКСОВ	35
3.5 ГРАФИЧЕСКИЙ ИНТЕРФЕЙС	36
4. АНАЛИЗ РЕЗУЛЬТАТОВ	41
4.1 СОЗДАНИЕ ТЕСТОВОЙ ЗАДАЧИ	41
ЗАКЛЮЧЕНИЕ	44
СПИСОК ИСТОЧНИКОВ	46
ПРИЛОЖЕНИЕ А	49

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

ИС – Информационная система.

СУБД – Система управления базами данных.

БД – База данных.

DSL – Domain specific language.

SQL – Structured Query Language.

BPMN – Business Process Model and Notation.

URI – Uniform resource identifier.

API – Application programming interface.

JDBC – Java Database Connectivity.

UML – Unified Modeling Language.

ОС – Операционная система.

GUI – Graphical user interface.

HTTP – Hypertext Transfer Protocol.

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

Информационная система – система, предназначенная для хранения, поиска и обработки информации, и соответствующие организационные ресурсы (человеческие, технические, финансовые и т. д.), которые обеспечивают и распространяют информацию (ISO/IEC 2382:2015)^[1].

База данных — совокупность данных, хранимых в соответствии со схемой данных, манипулирование которыми выполняют в соответствии с правилами средств моделирования данных (ISO/IEC 10032:2003)^[2].

Система управления базами данных – совокупность программ и лингвистических средств общего или специального назначения, управление создание и использование баз данных.

Фреймворк – набор функций и объектов, определяющий структуру программной системы, «каркас».

Domain specific language – предметно-ориентированный язык программирования, для использования в конкретной области применения.

Structured Query Language – язык структурированных запросов, для управления реляционной моделью данных в СУБД.

Business Process Model and Notation – система условных обозначений для моделирования бизнес-процессов. Также может применяться для описания взаимодействия компонентов в информационных системах^[3].

Uniform resource identifier – универсальный идентификатор ресурса, строка, позволяющая идентифицировать ресурс в сети интернет.

Application programming interface – спецификация взаимодействия с программой или программным компонентом.

Java Database Connectivity – интерфейс взаимодействия приложения на Java с различными СУБД.

Unified Modelling Language – графический язык объектного моделирования в области разработки программного обеспечения.

Синтаксический анализатор – программа для преобразования данных в структурированный формат, пригодный для обработки вычислительной техникой.

Операционная система – комплекс программ, предназначенный для управления компьютером и запуска пользовательских программ.

Graphical user interface – средство взаимодействия пользователя с информационной системой посредством графического представления.

Hypertext Transfer Protocol – прикладной протокол для передачи данных. Изначально использовался для передачи только гипертекста (текста с ссылками на другой текст), в настоящее время используется для передачи произвольных данных.

Репозиторий – место хранения и поддержки данных, чаще всего доступных к распространению по сети.

Индекс – объект базы данных, создаваемый с целью повышения производительности поиска данных.

Автоиндексирование – процесс построения индексов базы данных в автоматическом режиме.

ВВЕДЕНИЕ

Всю свою историю, человечество изобретало новые способы хранения информации: наскальные рисунки, записки на папирусе, печатные книги, компьютеры... Чем сильнее развивался человек, тем больше информации ему необходимо было сохранить, получить и обработать, что повышало требования и к эффективности хранения, и к скорости доступа к ней (информации). Массовый приход вычислительной техники в нашу жизнь в конце двадцатого века в очередной раз повысил эти требования. Тогда и были изобретены базы данных и системы управления базами данных.

Современные информационные системы невозможно представить без баз данных, они используются повсеместно: интернет-магазины, встраиваемые устройства, приложения и игры, банки, корпоративные системы.

Когда некий клиент обращается к информационной системе, система отправляет запрос к базе данных, а база данных, в свою очередь, должна выполнить этот запрос и вернуть результат – от этой операции сильно зависит общее время, затраченное клиентом на действие в информационной системе.

С развитием технологий, растёт и активное число пользователей различных информационных систем, растёт количество запросов, объём передаваемых данных, сложность данных, что делает задачу ускорения обработки запросов к базам данных **актуальной** на любом этапе развития технологий.

Целью работы является автоматизация процесса ускорения баз данных, что освободит администраторов и программистов от решения большей части задач по оптимизации, даст им дополнительное время на ускорении наиболее критичных и важных запросов вручную. Так как базы данных могут быть развёрнуты на всех актуальных на данный момент операционных системах (Windows, macOS, Linux) и архитектурах (x86-64, arm), то процесс автоматизации должен быть доступен на всех перечисленных конфигурациях.

Для реализации цели, передо мною были поставлены следующие задачи:

1. Обзор существующих решений.
2. Разработка требований, предъявляемых к создаваемому приложению.
3. Выбор средств реализации.
4. Проектирование и создание тестовой базы данных.
5. Проектирование и создание приложения.
6. Тестирование приложения.
7. Анализ полученных результатов.

Результатом работы является приложение, которое принимает на вход истории запросов к базе данных; производит действия, направленные на ускорение этих запросов; возвращает отчёт с результатами своей работы.

Работа поделена на четыре части: в первой содержится обзор существующих решений и обоснование используемых технологий. Во второй, описание этапа проектирования приложения и тестовой базы данных. В третьей части содержится описание этапа разработки, а в последней, четвертой, анализ результатов работы приложения.

1. ПОДГОТОВИТЕЛЬНЫЙ ЭТАП

1.1 Введение в предметную область

В широком смысле, базой данных можно назвать любое существующее средство хранения информации, будь то книга или молекула ДНК. В контексте компьютерных наук, первые базы данных появились в 50-х годах прошлого столетия, когда появилось программируемое оборудование для записи.

В 70-х, когда данных стало много, остро стала задача манипулирования этими данными, был разработан SQL (Structured Query Language, язык структурированных запросов) – язык программирования для управления, изменения, создания и удаления данных, основанный на реляционной модели данных. Реляционная модель данных – прикладная теория, которая определяет задачи обработки данных на трёх аспектах:

1. Структурный аспект – данные представляют собой набор отношений (часто их называют таблицами).
2. Аспект целостности – отношения отвечают условиям целостности, которые уникальны для каждой модели данных.
3. Аспект обработки – модель должна поддерживать операторы манипулирования отношениями, которые называются реляционной алгеброй.

Язык SQL, хотя и не был Тьюринг-полным языком (не позволял реализовать любую вычислимую функцию), но оказался чрезвычайно удобен для задач манипулирования данными. В течение десятилетия, появилось несколько СУБД, которые использовали SQL, к примеру: IBM System R, Ingres (будущая PostgreSQL), Oracle V2 (будущая Oracle Database). Популярность языка привела к его стандартизации комитетом ANSI в 1986 году, и к появлению класса SQL-совместимых СУБД, которые следовали стандарту ANSI-SQL, а также расширяли его, с целью получения конкурентных преимуществ. Это способствовало дальнейшему развитию языка: ANSI

стандартизировало лучшие практики из той или иной СУБД, которые не были защищены патентным правом. Также, свою роль в массовости SQL сыграл декларативный характер языка (пользователями необходимо было задавать спецификацию решения задачи, а не способ её получения) и развитие средств визуального построения запросов.

Параллельно с этим, развивался класс NoSQL СУБД, которые предлагали альтернативный язык для работы с реляционной моделью данных. NoSQL базы не были хуже, но не предлагали такой легкости и гибкости как SQL, поэтому закрепились в определённых сферах, где использование того или иного языка или СУБД давало значительные преимущества.

Так как класс SQL-совместимых баз данных самый крупный и популярный на данный момент^[4], а многие вещи стандартизированы, то задачу автоматизации стоит решать именно для них.

1.2 Обзор существующих решений

Перед разработкой стоит рассмотреть существующие решения. 7 из 10 самых популярных баз данных используют SQL как язык запросов. Рассмотрим их на наличие необходимой функциональности, данные будут представлены в таблице ([Таблица 1](#)).

Таблица 1 - Сравнение аналогов

СУБД	Функциональность
Oracle	Умеет автоматически строить индексы на основе истории запроса, но: 1. СУБД платная, цена даже за самую простую редакцию составляет несколько тысяч долларов ^[5] .

	2. Алгоритм и правила построения индексов представляют из себя «черный ящик» для пользователя.
MySQL	Автоматически строит индексы только для полей, представляющих из себя ключи, и только во время создания таблицы или внесения в неё нового ключа.
Microsoft SQL Server	Умеет автоматически строить индексы на основе истории запросов, но: <ol style="list-style-type: none"> 1. Данная функциональность доступна только обладателями платной подписки службы облачных вычислений Microsoft Azure. 2. Алгоритм и правила построения индексов представляют из себя «черный ящик» для пользователя.
PostgreSQL	Автоматически строит индексы только для полей, представляющих из себя ключи, и только во время создания таблицы или внесения в неё нового ключа.
IBM Db2	Автоматически строит индексы только для полей, представляющих из себя ключи, и только во время создания таблицы или внесения в неё нового ключа.
Microsoft Access	Автоматически строит индексы только для полей, представляющих из себя ключи, и только во время создания таблицы или внесения в неё нового ключа.
SQLite	Автоматически строит индексы только для полей, представляющих из себя ключи, и только во время создания таблицы или внесения в неё нового ключа.

Из таблицы можно сделать выводы о том, что существует потребность в создании приложения, которое позволит реализовать функциональность автоиндексирования для каждой из СУБД, с понятным и открытым принципом

работы, а также с более доступной, или вовсе бесплатной, моделью использования.

1.3 Обоснование используемых технологий

Ключевым требованием к разрабатываемому приложению является поддержка как можно большего числа конфигураций «операционная система – архитектура компьютера – СУБД», поэтому в качестве языка программирования следует выбрать тот, код которого будет ожидаемо и единообразно исполняться везде, где может запускаться. Таким языком является Java, благодаря Java Virtual Machine (JVM) – среде выполнения, доступной на трёх миллиардах устройств по заверениям разработчиков^[6].

Созданный в 1990 году Java – крайне распространённый язык, ежегодно занимающий лидирующие позиции^[7] в рейтингах популярности и любви от программистов за удачную реализацию принципов объектно-ориентированного программирования (ООП), наличия удобных средств разработки, пакетных менеджеров и многих других преимуществ. Тем не менее, за свою более чем тридцатилетнюю историю, как и любой другой язык, Java успел накопить проблем, в первую очередь связанных с слишком строгим следованием правилам ООП и громоздким синтаксисом. Со временем, стали появляться другие языки, которые исправляли накопившиеся проблемы, а их код мог выполняться в JVM, что обеспечивало полную совместимость с существующим кодом, написанным на Java.

Одним из таких языков является Kotlin, созданный в Санкт-Петербурге в 2011 году фирмой JetBrains, специализирующейся на разработке средств программирования, преимущественно на Java. Имея богатый опыт, контакт с сообществом, под вдохновением от книги «Java. Эффективное программирование», они создали язык, который поддерживал мультипарадигменное программирование и значительно улучшил оригинальный синтаксис. Многие из тех вещей, которые были введены в

Kotlin, в данный момент заимствуются в новых редакциях Java. Именно Kotlin будет использоваться при разработке приложения.

Другой причиной, почему была выбрана именно технология JVM, помимо языка, является существование протокола JDBC. Протокол обеспечивает общий интерфейс для взаимодействия с любой SQL СУБД, для который разработчики СУБД написали JDBC-драйвер, что они обычно и делают, ввиду широкого распространения Java/JVM в корпоративной разработке. Таким образом, программистам, решившим реализовать в своём приложении работу с новой СУБД, не нужно изучать библиотеку от разработчиков этой самой СУБД, нужно лишь один раз научиться работать с протоколом JDBC. Это также позволит мне удобно добавлять поддержку новых СУБД, так как интерфейс выполнения SQL выражений будет общий, сложная задача будет состоять в том, чтобы научить программу формировать выражения создания индексов, специфичные для каждой СУБД.

Приложению также нужен интерфейс взаимодействия с пользователем, графический или консольный. Стоит отметить, что, если реализовать взаимодействие с приложением через HTTP-протокол, появится возможность независимо от ядра приложения (той части, которая будет отвечать за разбор истории запросов и формирование индексов), обновлять графический интерфейс. Или создать консольный интерфейс ввода. Или создать другой клиент на любой вкус. Было решено так и поступить. Клиент я решил реализовать в виде веб-приложения, как это часто делают другие авторы ПО, используемого в качестве инфраструктурного, к примеру, Jenkins, WildFly или pgAdmin. Выбор фронтенд-фреймворка по большей части дело вкуса: каждый из трёх самых популярных фреймворков (Vue, React, Angular) имеет подробную документацию, активное сообщество, библиотеки компонентов. С первыми двумя у меня был опыт, в обоих случаях положительный, а потому я решил использовать React: мне он нравится своим декларативным подходом.

Выбор библиотеки компонентов зависит исключительно от внешнего вида компонентов, так как правила работы с ними определяет фреймворк, а

потому в каждой библиотеки они будут одинаковы. Я выбрал Ant Desing, стиль компонентов мне показался наиболее аккуратным, а документация очень удобной, ввиду наличия множества примеров использования компонентов, практически на каждый параметр компонента присутствует свой пример.

Так как ядро приложения будет является сервером, доступ к которому могут получить несколько клиентов в сети, или может быть развернуто несколько экземпляров приложения параллельно, нам нужно где-то хранить информацию, какая из баз данных сейчас индексируется приложением, чтобы не допустить параллельного построения одних и тех же индексов несколькими экземплярами приложения. Хранить достаточно лишь строку – url занятой базы данных. Для таких целей отлично подойдут хранилища типа «ключ-значение», самые известные из которых Redis и Memcached. Оба продукта представляют одинаковую функциональность, но Redis имеет чуть более удобную в использовании библиотеку для JVM, поэтому был выбран он.

Kotlin поддерживает те же серверные фреймворки, что и Java, в том числе Spring Boot, давно зарекомендовавший себя как лучший фреймворк для серверной разработки на JVM. Тем не менее, он создавался для Java, и его использование в паре с Kotlin приводит к некоторым ограничениям в отношении использования разных современных возможностей языка. Для Kotlin существует свой фреймворк – Ktor. Он написан с учётом всех плюсов языка: декларативен, основан на замыканиях, а не классах, по сути, представляет из себя DSL, что очень удобно.

Невозможно будет проверить работу программы без существования сложной базы данных. Модель данных будет описана в следующей части, здесь стоит определиться с СУБД, в который эта база будет создана. Для тестирования стоит выбрать ту, которая имеет несколько разных индексов, позволит создать сложную модель данных, а опыт работы в ней достаточен, чтобы в случае проблем быть уверенным, что проблемы именно в

разработанном приложении, а не в неправильно сконфигурированной СУБД.
Для меня такой является PostgreSQL.

Итого: клиентская часть на React, серверная на Kotlin, а СУБД – PostgreSQL.

2. ЭТАП ПРОЕКТИРОВАНИЯ

2.1 Проектирование структуры и процессов

Определившись с используемыми технологиями, нужно понять, когда и как эти технологии будут между собой взаимодействовать. При проектировании стоит учитывать, что процесс автоиндексирования базы данных может занимать длительное время, поэтому передача клиенту итогового отчёта будет асинхронным процессом. В таком случае, наилучшим вариантов будет дать возможность клиенту самому выбрать, куда получить отчёт, а не дожидаться HTTP-ответа от серверной части приложения. В связи с этим, я вижу этот процесс так:

1. Клиент создаёт задание на автоиндексирование, предоставляя историю запросов, данные для подключения к БД и данные для получения отчёта.
2. Сервер, получив запрос, выполняет необходимые проверки, среди которых, к примеру, проверка на доступность БД, над который будет проводиться работа и разбор SQL-выражений на корректность, не занята ли БД индексацией в данный момент.
3. Клиент получит ответное сообщение, в случае успеха: что его задание принято в работу, и отчёт он получит, как только работа будет завершена; в случае проблем на этапе проверки, сообщение об ошибке придёт сразу в теле HTTP-ответа; в случае проблем на этапе построения индексов, сообщение об ошибке придёт туда, куда он указал получать отчёты.

Наглядно этот процесс представлен на BPMN-схеме ([Рисунок 1](#)).

[Рисунок 2](#) показывает взаимодействие всех компонентов системы.

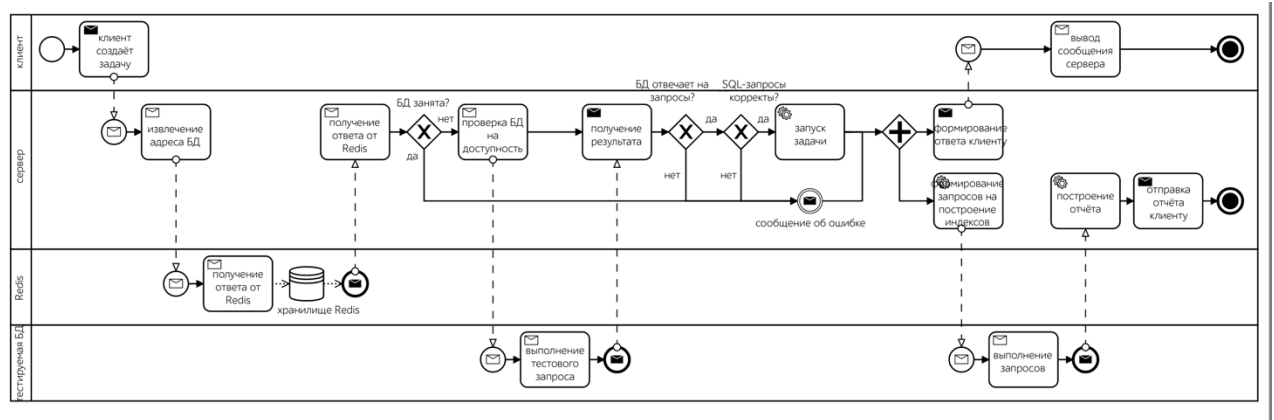


Рисунок 1 - BPMN-схема



Рисунок 2 - Компонентная схема

2.2 Проектирование API

Так как количество поддерживаемых приложением СУБД можно будет увеличить, путём подключения нового JDBC-драйвера, необходимо также отдавать клиентам информацию о том, какие СУБД поддерживаются на данный момент, чтобы они, в свою очередь, не пытались слать бессмысленные задачи на оптимизацию. Также, динамическое получение клиентами списка доступных СУБД позволит оставлять их код неизменным при добавлении нового драйвера к серверу. В качестве поддерживаемой изначально я выбрал

PostgreSQL, причины были описаны на этапе 1.3. Таким образом будет сформирован первый путь к серверу: `support/instances/`.

Отчёты, формируемые по результатам автоиндексирования, должны содержать информацию о том, какой индекс использовался при осуществлении запроса к БД и как изменилась скорость выполнения по сравнению с оригинальным запросом, выполненным без индекса. Изначально, в качестве формата отчёта я выбрал CSV: этот формат, ввиду небольшого количества служебных символов (запятые и символы переноса строк), легко читается человеком. Его просмотр доступен в средствах форматирования электронных таблиц, таких как Microsoft Excel. Тем не менее, помимо человека, отчёт может понадобиться другой клиентской программе, поэтому также необходимо предоставлять его в форме, которую также хорошо и быстро воспринимает вычислительная машина. Самым популярным таким форматом является JSON, чуть ли не повсеместно применяемый при передаче данных по протоколу HTTP. На выбор пользователю будут доступны оба формата. Если же, как и в случае с СУБД, отдавать список поддерживаемых форматов с сервера, то новые форматы можно будет добавлять, не внося изменений в клиент. Создадим путь `support/formats/`.

Было оговорено, что отчёты будут возвращаться пользователю асинхронно. Сущность, куда будет отправлен отчёт назовём «потребителем». Со списком потребителей можно организовать ту же схему, что и с список поддерживаемых СУБД и форматов – возвращать его клиенту, а клиент будет выбирать потребителя из этого списка. Нужно определиться с потребителями, которые будут доступны в приложении с самого начала. Предполагается, что разворачивать программы пользователь будет в рамках своей локальной или корпоративной сети: никто не захочет передавать логин и пароль от своей базы в третьи руки. Приоритетным потребителем в таком случае будет выступать файловая система, пользователю в таком случае надо будет указать директорию, куда сохранять отчёт. Тем не менее, возможны ситуации, когда нельзя или не надо будет сохранять отчёт на машине, где развёрнуто

приложение, в таком случае надо обеспечить альтернативный способ доставки. В локальных сетях часто используют смонтированный для общего доступа диск, доступ к которому возможен по разным протоколам. Одним из таких протоколов является SFTP: на машинах, доступ к которым есть по SSH, есть и доступ по SFTP. Альтернативой ему, часто используемой на устройствах с ОС Windows, является SMB. К тому же, может оказаться полезной отправка отчётов тем, у кого по тем или иным причинам нет доступа ни к диску, ни к машине с приложением. В качестве такого потребителя будет выступать электронная почта. Итого, по умолчанию доступны четыре потребителя: FS (файловая система), SFTP, SMB, EMAIL. Актуальный список поддерживаемых потребителей будет доступен на support/consumers/.

Теперь надо определиться с форматом запросов для создания задачи на автоиндексирование БД. В первую очередь, нужно передать информацию для подключения: протокол, адрес, порт, имя логической базы данных, логин и пароль. Всю эту информацию можно объединить в URI. Историю SQL-запросов можно передавать в виде массива строк. В виде строк можно передавать и требуемый формат отчёта, и наименование потребителя. Наиболее сложной задачей является каким-то образом стандартизировать информацию, необходимую для взаимодействия с потребителем. Серверу нужна разная информация для работы с потребителем: если потребитель является электронным почтовым ящиком, то необходим только адрес этого ящика. А если, к примеру, это SFTP хранилище, то нужны уже и адрес, и данные для авторизации. От потребителя к потребителю структура может становиться как проще, так и сложнее. Решено было компоновать эту информацию в одну строку и разделять информацию символом «;». Информация, необходимую каждому поставщику будет описана в документации. В большинстве случаев, это всё же обычный URI. Запрос на новую задачу по автоиндексированию будет располагаться по адресу: [/bench/](http://bench/).

Выдержка из спецификации API, оформленная в соответствии с стандартом OpenAPI 3^[8] ниже:

```

openapi: 3.0.3
info:
  title: SQL-OPTIMIZER
  version: 1.1.0
# ...
  BenchTask:
    type: object
    properties:
      connectionUrl:
        type: string
        required: true
      queries:
        type: array
        items:
          type: string
          required: true
      consumer:
        type: string
        required: false
        default: FS
      format:
        type: string
        required: false
        default: CSV
      consumerParams:
        type: string
        required: false
        default: ''
      saveBetter:
        type: boolean
        required: false
        default: false
paths:
# ...
  /bench:
    post:
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: 'components/schemas/BenchTask'
      responses:
        200:
          description: Info about test request
          content:
            application/json:
              schema:
                $ref: 'components/schemas/CommonMessage'
        400:
          description: This database not supported yet
# ...
  /support/formats:
    get:
      responses:
        200:
          description: Enum with supported formats
          content:
            application/json:
              schema:
                $ref: '#components/schemas/formats'
# ...

```

2.3 Проектирование тестовой базы данных

Чтобы удостовериться в работе приложения, необходимо проверить его на какой-либо базе данных. Эта база должна быть большой по количеству записей и обладать сложной моделью данных (наличие множества связей, ключей, промежуточных таблиц). Попробуем спроектировать такую БД. Определим предметную область как ЧВК – Частная военная компания. Новые отношения (таблицы) и их атрибуты можно придумывать бесконечно, а потому, будем усложнять модель, пока на это хватит фантазии. Далее, название таблиц будут указаны моноширинным шрифтом, а названия полей взяты в скобки и написаны курсивом.

Очевидно, что ЧВК имеет штат СОТРУДНИКОВ (*ИМЯ, ФАМИЛИЯ, ДАТА_РОЖДЕНИЯ, ОБРАЗОВАНИЕ, СЕМЕЙНЫЙ_СТАТУС, ДАТА_ЗАЧИСЛЕНИЯ_НА_СЛУЖБУ*).

Сотрудники могут занимать различные военные и невоенные ДОЛЖНОСТИ (*НАЗВАНИЕ, ЗАРПЛАТА, ВОИНСКОЕ_ЗВАНИЕ* если есть, номер комплекта ЭКИПИРОВКИ, тип *ВООРУЖЁННЫХ_СИЛ*).

Типы *ВООРУЖЁННЫХ_СИЛ* могут быть представлены перечислением: СВ (сухопутные войска), ВМФ (военно-морской флот), ВКС (воздушно-космические силы).

Так как работа штата сопряжена с риском, каждый сотрудник имеет МЕДКАРТУ (*РОСТ, ВЕС, ГРУППА_КРОВИ, список ТРАВМ_И_ЗАБОЛЕВАНИЙ, ЛОЛ*).

Сотрудники закреплены за БАЗАМИ (*МЕСТОПОЛОЖЕНИЕ, СТАТУС*).

Сотрудники могут отправляться на МИССИИ (*НАЗВАНИЕ, ДАТА_И_ВРЕМЯ_СТАРТА, ДАТА_И_ВРЕМЯ_ЗАВЕРШЕНИЯ, ЮРИДИЧЕСКИЙ_СТАТУС, МЕСТО_ОТПРАВЛЕНИЯ, МЕСТО_ПРИБЫТИЯ, ВРАГИ*), также должна быть доступна информация, какие сотрудники участвовали в той или иной миссии.

Миссии являются частью военной КАМПАНИИ (*НАЗВАНИЕ, ЗАКАЗЧИК, ПРИБЫЛЬ, ЗАТРАТЫ, СТАТУС*).

Ведётся учёт ТРАНСПОРТА (*НАЗВАНИЕ, ТИП, СОСТОЯНИЕ*) и проведённых ТЕХОСМОТРОВ (*НОМЕР_ТРАНСПОРТА, НОМЕР_ОБСЛУЖИВАЮЩЕГО, ДАТА_ОСМОТРА*).

Людам с военными должностями должен выдаваться комплект экипировки (КАМУФЛЯЖ, СРЕДСТВА_КОММУНИКАЦИИ, СРЕДСТВА_РАЗВЕДКИ, ИРП, ОРУЖИЕ). Каждый элемент не обязательно должен присутствовать в комплекте (кроме ИРП).

Индивидуальный рацион питания – ИРП (БЕЛКИ, ЖИРЫ, УГЛЕВОДЫ, КАЛОРИЙНОСТЬ, блюда ЗАВТРАКА, ОБЕДА, УЖИНА, ПИЩЕВЫХ_ДОБАВКИ).

В контексте оружия, мы не будем делать отличий между холодным, огнестрельным или любым другим на уровне таблиц, просто заведём отдельный атрибут для этого. Таким образом, все атрибуты, кроме названия и типа необязательны: ОРУЖИЕ (НАЗВАНИЕ, ТИП, КАЛИБР, СКОРОСТРЕЛЬНОСТЬ, ДЛИНА_СТВОЛА, ПРИЦЕЛЬНАЯ_ДАЛЬНОСТЬ). Полная схема даталогической модели – [Рисунок 3](#).

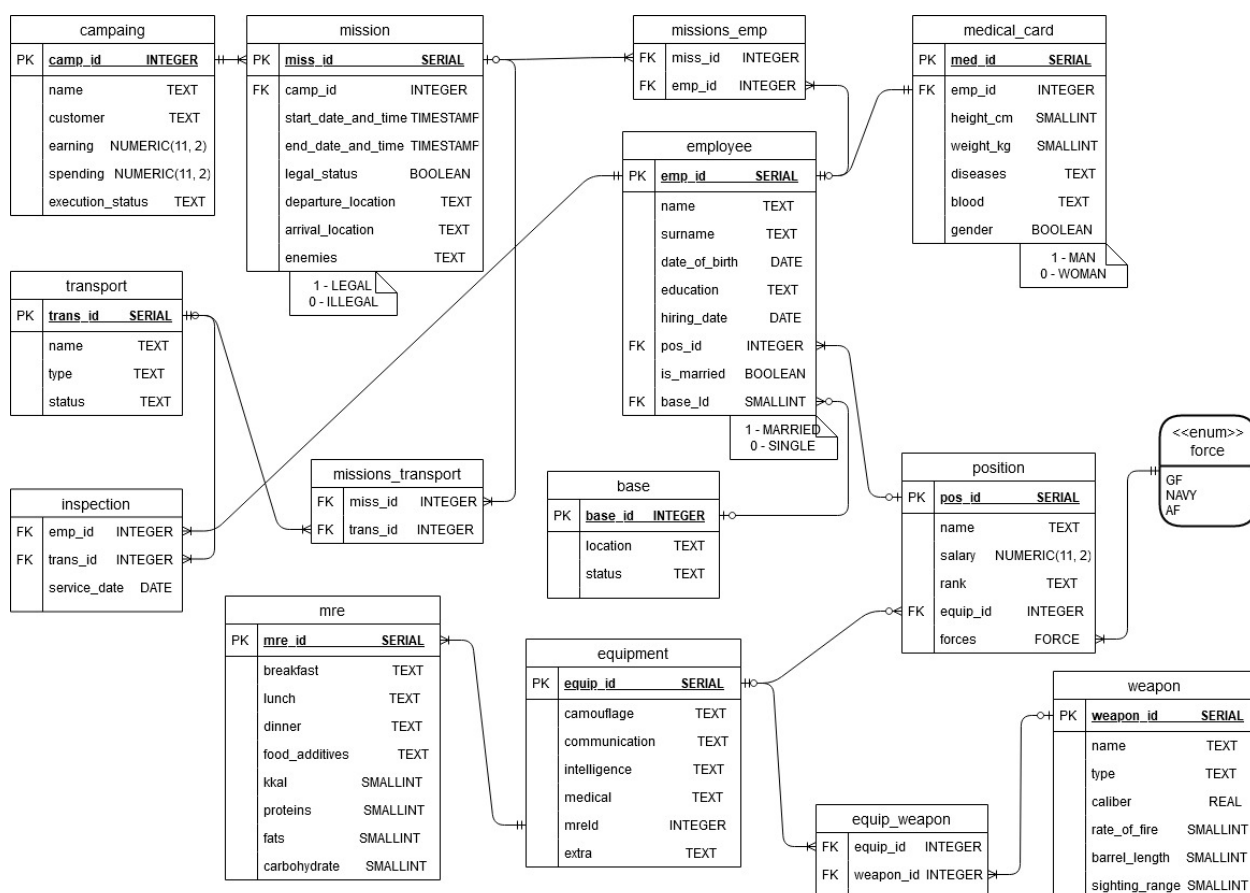


Рисунок 3 - Даталогическая модель тестируемой базы данных

Выдержка из SQL-скрипта, создающего модель:

```
CREATE TYPE force AS ENUM ('GF', 'NAVY', 'AF');

CREATE TABLE position
(
    pos_id    SERIAL PRIMARY KEY,
    name      TEXT          NOT NULL,
    salary    NUMERIC(11, 2) NOT NULL CHECK (salary >= 300),
    rank      TEXT,
    equip_id  INTEGER       REFERENCES equipment ON DELETE SET NULL,
    forces    FORCE
);
```

Базу данных нужно было заполнить. Для этого мною был написан небольшой скрипт на Kotlin Script – скриптовом расширении языка. Сложности процесса генерации возникают из-за того, что записи некоторых таблиц обязательно должны содержать ссылки на записи из других таблиц (к примеру, ЭКИПИРОВКА обязательно должна содержать ОРУЖИЕ). Таким образом, данные нужно генерировать в паре, либо перед генерацией ведомых данных, делать выборку ведущих данных. К тому же, для чистоты эксперимента, нужно генерировать осмысленные данные: атрибут СОТРУДНИК(ИМЯ) должен содержать именно имя, а не бессмысленный набор символов. Для решения этой проблемы я использовал библиотеку Java Faker^[9], которая умеет генерировать случайные данные различной тематики, будь то имя, фамилия или координаты. Также приходилось учитывать ограничения, накладываемые некоторыми атрибутами, для примера: вес и рост не могут принимать отрицательные значения. Пример использования этой библиотеки и попарной генерации данных:

```
@InternalAPI override fun generateAndInsert(n: Int) {
    val campIds = CampaignTable.selectAll().map { it[CampaignTable.camp_id] }
}

(1..n).forEach {
    val st = F.date().between(
        Date.from(LocalDate.of(2014, 3,
18).atStartOfDay(ZoneId.systemDefault()).toInstant()),
        Date()
    )
    val et = F.date().between(st, Date())
    MissionTable.insert {
        it[camp_id] = campIds.random()
    }
}
```

```

        it[start_date_and_time] = st.toLocalDateTime()
        it[end_date_and_time] = et.toLocalDateTime()
        it[legal_status] = Random.nextBoolean()
        it[departure_location] = "${F.address().latitude()}"
        it[F.address().longitude()] = "${F.address().longitude()}"
        it[arrival_location] = "${F.address().latitude()}"
        it[F.address().longitude()] = "${F.address().longitude()}"
        it[enemies] = arrayOf(F.nation().nationality(),
        F.name().fullName()).random()
    }
}
}

```

Было сгенерированы около пяти тысяч записей каждого типа. Запросы, которые будут использоваться для тестирования, нужно делать сложными, чтобы ещё больше воссоздать сценарий использования приложения. Также, запросы не будут абстрактной выборкой, а воссоздадут настоящий возможный сценарий использования. Пример такого запроса – получение наиболее подходящих кандидатов для отправки на боевую миссию: сотрудники, имеющие боевое звание, не женаты, давно не были на заданиях, наиболее опытные.

```

SELECT emp_id FROM employee
    JOIN position USING (pos_id)
    JOIN missions_emp USING (emp_id)
    JOIN mission USING (miss_id)
WHERE rank !~~ ''
ORDER BY is_married DESC, end_date_and_time DESC, hiring_date DESC
LIMIT 20;

```

3. ЭТАП РАЗРАБОТКИ

3.1 Создание расширяемой архитектуры

Главным требованием, предъявляемым к архитектуре серверной части приложения, является простота расширения: будь то добавление поддержки новой СУБД, или нового потребителя.

Несмотря на то, что для выполнения SQL-запросов к СУБД используется единственный интерфейс – интерфейс JDBC, он не обладает информацией о том, какие индексы существует в каждой конкретной СУБД, а также о синтаксисе выражения добавления индекса, который может отличаться от стандартного SQL. Необходимо было использовать такое средство языка программирования, которое будет удовлетворять следующим критериям:

1. Добавление новых экземпляров класса
2. Получение списка существующих экземпляров класса
3. Общий интерфейс

Есть два средства, которые подойдут: интерфейсы (interface) и перечисления (enum). Мною был выбран последний вариант: список поддерживаемых СУБД известен на этапе компиляции, динамически неизменен, а потому, необходимость в структурах, которые меняют свой размер отсутствует, перечисления работают быстрее и занимают меньше памяти.

Новый элемент перечисления принимает на вход названия индексов СУБД, выражение для создание индексов на основе стандарта SQL, переопределяет метод формирования запросов и возвращает множество этих запросов. Таким образом, можно выполнять запросы с помощью JDBC независимо от их (запросов) генерации, как бы сильно синтаксис конкретной СУБД не расходился с общим SQL-синтаксисом. UML-схема – [Рисунок 4](#).

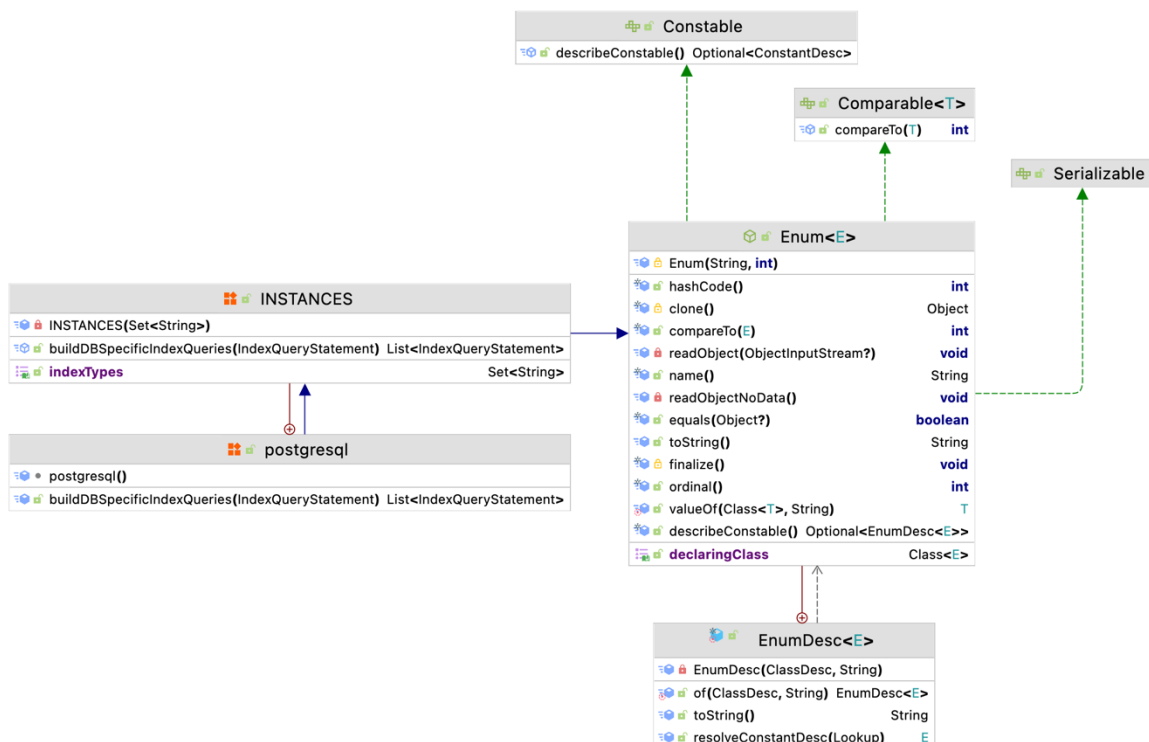


Рисунок 4 - Схема перечисления поддерживаемых СУБД

К потребителям предъявляются те же самые требования расширяемости, что и к средствам поддержки СУБД, поэтому я также выбрал перечисления в качестве хранилища, отличаться будет интерфейс. Мы определили, что информация, необходимая потребителю, будет передаваться в виде строки. Информация эта может сильно отличаться, поэтому каждый потребитель сам будет решать, что делать с строкой. Также потребитель должен получить контент, с которым он будет работать. Этим контентом будет отчёт – файл одного из поддерживаемых форматов. Для каждого оптимизируемого SQL-выражения будет создаваться свой отчёт, поэтому пускай потребитель принимает переменное число контента. [Рисунок 5](#) показывает схему потребителей.

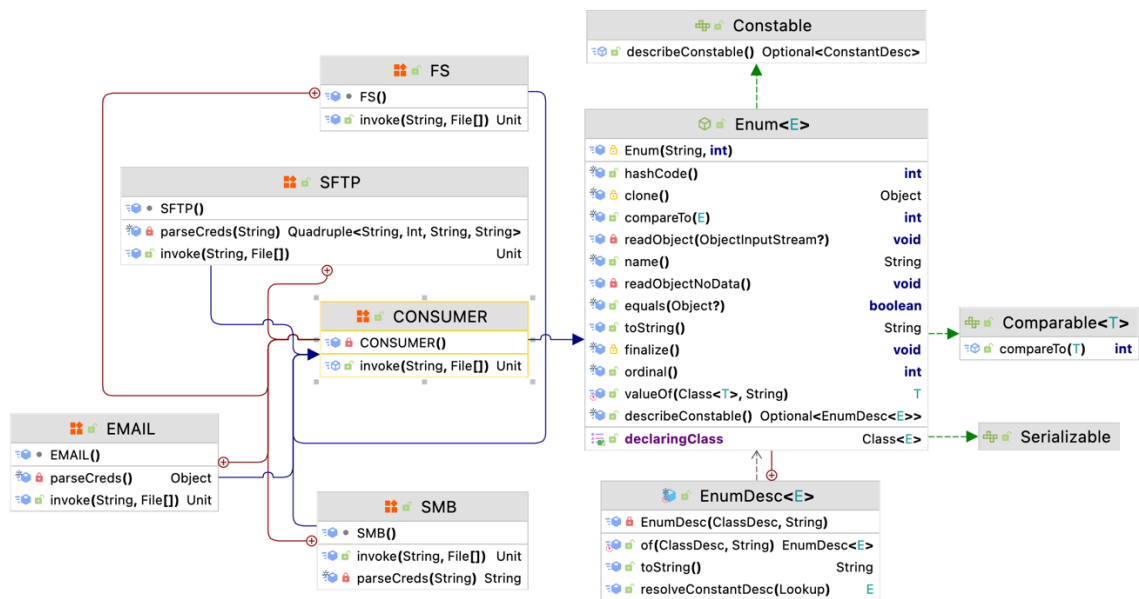


Рисунок 5 - Схема перечисления потребителей

3.2 Декларативная организация кода

Ещё одной особенностью, которую хотелось бы реализовать для повышения читаемости кода, и которая является одним из преимуществ языка Kotlin над Java, являются замыкания. Замыкания — это функции, которые имеют ссылки на переменные, объявленные вне тела этой функции. К тому же, они являются функциями первого класса, то есть такими функциями, которые можно передавать как параметр в другие функции. Благодаря этим особенностям, код можно писать декларативно, делая его визуально чище и понятнее.

Данную особенность я решил применить в местах, где поток кода зависит от результата какого-либо выражения. Вместо сложных ветвлений if-else, при положительном течении кода будет выполняться следующая вложенная функция, а при негативном — обработка исключения. Пример такого поведения — подключение к хранилищу занятых индексацией баз данных в Redis:

```

fun <T> executeLocking(dbUrl: String, lockingOps: () -> T): T =
    if (System.getenv("DEBUG").toBoolean()) lockingOps()
    else if (dbUrl !in this) {
        this + dbUrl
        try {
            val res = lockingOps()
            this - dbUrl
            res
        } catch (e: Exception) {
            this - dbUrl
            throw e
        }
    } else throw DatabaseBusyException()

```

Другим удачным примером использования замыкания, я считаю функцию для замера времени выполнения SQL-запроса, так как подобной функциональности по умолчанию в JDBC нет: она принимает некоторый SQL-запрос, выполняет его, и возвращает время этой операции. Замыкания были применены и в других местах программы:

```

fun measureQuery(queryFunc: () -> String): Long =
    getConnection().use {
        it.createStatement().use {
            measureNanoTime {
                it.execute(queryFunc())
            }
        }
    }

```

3.3 Обработка истории SQL-запросов

Важнейшей частью приложения является код обработки истории SQL-запросов. Это база, на который строится всё приложение. Я стал искать библиотеки для разбора SQL-запросов для Java. Одной из таких библиотек является General SQL Parser, которую рекомендуют на форуме для программистов Stack Overflow^[10]. К сожалению, чтобы использовать эту библиотеку, требуется приобрести лицензию, поэтому я продолжил поиски. Другим возможным вариантом является использование ANTLR — это фреймворк для генерации синтаксических анализаторов. Этот вариант уже лучше, чем писать синтаксический анализатор с нуля самостоятельно, но всё же требует изучения достаточно сложной технологии, избыточной для данной

задачи, а также идеального знания синтаксиса SQL. Далее я нашёл библиотеку JSqlParser^[11] – это бесплатная и открытая альтернатива General SQL Parser. Библиотека обладала следующей функциональностью:

4. Проверка синтаксиса запроса
5. Извлечение имён таблиц
6. Добавление псевдонимом (англ. aliases)
7. Построение новых запросов

К сожалению, JSqlParser не умеет извлекать имена колонок таблиц, а потому это нужно было реализовывать самому. Это не вызвало больших сложностей, нужно было разбить строку запроса по символу « » и проверить, есть ли вхождение очередной части строки среди столбцов таблицы, из которой осуществляется выборка данных SQL-запросом.

Для формирования множества всех возможных комбинаций индексов необходимо было создать булеан – множество всех возможных множеств. Несмотря на наличие обширного количества методов для работы с коллекциями в Kotlin, такая функция в нём отсутствует, поэтому пришлось создать её самостоятельно. Благодаря возможности расширять существующие классы в Kotlin и системе обобщений Java, написанная мною функция будет удобно доступна любой коллекции любого типа:

```
fun <T> Collection<T>.powerset(): Set<Set<T>> =  
    powerset(this, setOf(emptySet()))  
  
private tailrec fun <T> powerset(left: Collection<T>, acc: Set<Set<T>>):  
    Set<Set<T>> =  
    if (left.isEmpty()) acc  
    else powerset(  
        left.drop(1),  
        acc + acc.map { it + left.first() }  
    )
```

[Рисунок 6](#) показывает блок-схему алгоритма трансформации SQL-запроса из истории в отчёт, который будет отправлен пользователю.

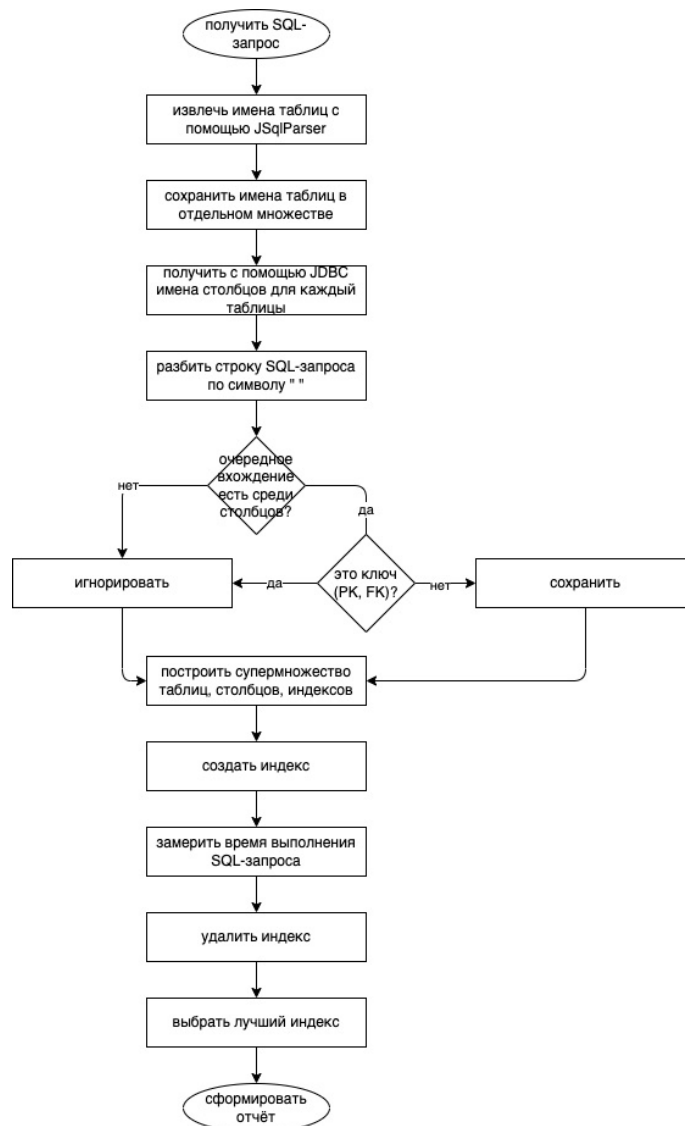


Рисунок 6 - Алгоритм формирования отчёта для SQL-запроса из истории

История SQL-запросов представляет из себя однотипные данные, над которыми производится цепочка операций, таких как преобразование, фильтрация, разбиение и другие. Такая ситуация называется конвейерным выполнением, а в контексте Java – потоками (англ. Stream). Kotlin также поддерживает потоки, и они являются подходящим инструментом для преобразования множества истории запросов в множество запросов создания индексов. Функция, выполняющая эту операцию приведена на рисунке:

```

private fun formIndexesQueries(): List<IndexQueryStatement> =
    with(CCJSqlParserUtil.parse(query) as Select) {
        TablesNamesFinder()
            .getTableList(this)
            .flatMap { t ->

```

```

        this.toString()
            .split(" ")
            .filter { it in support.getTableColumns(t) }
            .powerset()
            .filter { it.isNotEmpty() }
            // create index query statement for standart sql
            .map { IndexQueryStatement(t, it) }
            // create db-specific sql index query statement
            .flatMap {
support.creator.buildDBSpecificIndexQueries(it) }
    }
}

```

Так как операция тестирования БД может занимать значительное время, выполняться она должна параллельно, чтобы не блокировать сервер. Классическим средством для это в Java являются потоки (англ. Thread). В русском языке этот термин совпадает с термином Stream, а потому будет использовать менее известный перевод – нити. Нити позволяют коды выполняться параллельно, происходит это на уровне операционной системы, что имеет как плюсы, так и минусы. К первым, к примеру, можно отнести повышение скорости выполнения программы в ситуациях, когда параллельно выполняются несколько задач, особенно, если это задачи разные класса (обработка данных и операции ввода-вывода). Тем не менее, управление диспетчером ОС иногда избыточно, а потому, Kotlin поддерживает сопрограммы (coroutines) – средство для легковесной многопоточности. Использовать сопрограммы очень просто: нужно создать экземпляр класса Job, который позволяет управлять состояние сопрограммы, и передать функции launch объект Job и команды, которые необходимо выполнить параллельно. Ниже находится код сопрограммы автоиндексирования БД внутри обработчика сервера:

```

fun Route.actions() =
    route("/bench/") {
        post {
            val benchTask = call.receive<BenchTask>()
            val creds = benchTask.creds
            call.respond(DBsLock.executeLocking(creds.first) {
                DBsSupport(creds).let { sup ->
                    sup.checkDbAvailability()
                    launch(Job()) {
                        val results = benchTask.queries.map {
                            val tester = DBsTester(it, sup)

```

```

        val benchmarkingResult = tester.benchmarkQuery()
        val best = tester.findBest(benchmarkingResult)
        val origTime = sup.measureQuery { it }
        if (benchTask.saveBetter && best != null)
sup.execute { best.first.createIndexStatement }
            val report = Report(
                it,
                benchmarkingResult
                    .map { (k, v) ->
IndexResult(k.createIndexStatement, origTime, v) }
                    .sortedBy { i -> i.timeTaken },
                benchTask.format
            )
            val res = TestResult(
                best!!.first.createIndexStatement,
                origTime,
                best.second,
                best.second - origTime
            )
            report to res
    }

```

3.4 Генерация выражений создания индексов

Внимание стоит уделить генерации выражений создания индексов. Синтаксис этих выражений, как и набор индексов, может немного отличаться от ANSI SQL у каждой конкретной СУБД, а потому, стоит разбить этот процесс на две части: сначала генерировать общее выражение, без привязки к какой-либо СУБД, а после, на его основе, генерировать выражение, которое действительно будет выполняться. Выполнять процесс конвертации будет переопределяемая функция `buildDBSpecificIndexQueries(indexQuery: IndexQueryStatement): List<IndexQueryStatement>` перечисления `INSTANCES`. Таким образом, чтобы добавить поддержку новой СУБД, нужно лишь описать правило трансформации выражения создания индекса из ANSI SQL в SQL добавляемой СУБД.

В процессе тестирования, индексы нужно удалять, чтобы предыдущий индекс не оказывал влияния на время запроса при тестировании текущего индекса. Выражения удаления индекса могут отличаться от стандартного у разных СУБД, а потом с ними стоит поступить так же, как и с выражениями генерации.

Пример функции, которая превращает ANSI SQL-запросы в PSQL (язык запросов PostgreSQL), на рисунке ниже:

```
postgresql(setOf("HASH", "BTREE")) {
    override fun buildDBSpecificIndexQueries(indexQuery:
IndexQueryStatement): List<IndexQueryStatement> {
        val (beginning, end) = indexQuery.createIndexStatement.split(" (",
limit = 2, ignoreCase = true)
        if (beginning.endsWith(indexQuery.table))
            return indexTypes.mapNotNull {
                if (it == "HASH" && indexQuery.columns.count() > 1) null
                else indexQuery.copy().apply {
                    val newName = "${indexName}${it.toUpperCase()}"
                    createIndexStatement = "$beginning USING
$it($end".replace(indexName, newName)
                    indexName = newName
                    dropIndexStatement = "DROP INDEX IF EXISTS $indexName
RESTRICт;"
                }
            }
        else throw IndexCreationError(indexQuery.createIndexStatement)
    }
};
```

3.5 Графический интерфейс

Разбиение приложения на клиентскую и серверную часть создаёт простор для реализации и развития альтернативных клиентов с использованием разных технологий, либо для разных платформ в соответствии с нуждами пользователей. Тем не менее, наличие стандартного клиента позволит удобно запускать задачи по автоиндексированию без изучения HTTP-интерфейса или поиска альтернативных средств взаимодействия. Выбор технологий для реализации клиента был обоснован в главе 1.3, а потому сразу начну с описания архитектуры.

При разработке приложений с графическим пользовательским интерфейсом принято использовать шаблон MVC – Model-View-Controller (рус. Модель – Представление – Контроллер). Этот шаблон описывает структурное разбиение кода в соответствие с его назначением. Модель представляет данные и меняет своё состояние в соответствие с командами контроллера. Контроллер интерпретирует действия пользователя и

уведомляет модель о необходимости изменений. Представление отображает данные модели пользователю.

В качестве слоя представления будут выступать компоненты React – файла гипертекстовой разметки в формате `jsx`, из которых и будет сконструирован интерфейс.

За модель будут отвечать переменные, объявленные перед разметкой компонента с помощью функции `useState`. Она превращает произвольные данные в модель, а также сама выступает контроллером, отслеживая изменяя данных и модифицируя разметку соответствующим образом.

Контроллером станет и модуль `api.js`, который будет отвечает за взаимодействие клиента с сервером и инициацию изменений интерфейса согласно уже не модифицированным данным, а тому, какой код ответа вернул сервер на запрос: если код в диапазоне `[200; 300)`, то всё хорошо, иначе что-то пошло не так (это соответствует стандартному поведению протокола HTTP).

Нужно учитывать, что перед отправкой данных для создания задачи автоиндексирования БД, клиент должен получить с API список поддерживаемых СУБД и потребителей. Ошибка на одном из этих этапов сделает запуск задачи невозможной. В таком случае, нужно либо вывести пользователю информацию об ошибке и попросить перезагрузить страницу, либо попробовать отправить запрос ещё раз после некоторой паузы. Реализовать любой из вариантов можно полностью на клиентской стороне. Я выбрал первый, так как стандартный клиент будет разворачиваться вместе сервером, и возможность возникновения ошибки в таком случае крайне мала.

При создании графических интерфейсов основное внимание следует уделять удобству пользователя. Четыре из пяти полей отправляемого серверу объекта являются строками, пятый – массив строк. Тем не менее, обязанность вводить пользователем строки вручную повлечёт за собой раздражённость этого самого пользователя. Везде, где возможно, стоит максимально упростить процесс ввода данных. Так, для строки, содержащей URL базы данных, лучше сделать несколько полей ввода, отдельно для каждой части

этой строки (имя БД, пароль и т.д.). Список поддерживаемых СУБД будет запрошен с API, а потому его можно отобразить в виде набора кнопок или выпадающего списка, также стоит поступить и с потребителями, и с форматом отчётов.

Сложнее всего было реализовать ввод истории SQL-запросов. Ввод всех SQL-запросов одной строкой идея неудачная по нескольким причинам:

- Пользователям придётся вставлять в строку символ-разделитель запросов. Таким символом может служить «;», так как он завершает SQL-выражение, но за этим нужно следить.
- API требует на вход именно массив запросов. Придётся либо реализовывать альтернативный путь к серверу, который принимает строку и разделяет её на запросы, либо разделять на запросы на стороне клиента, либо изменять существующий API.
- Читать и редактировать историю запросов в таком виде неудобно.

Чтобы запрос соответствовал интерфейсу сервера пришлось организовать динамическое добавление дополнительных полей ввода в форму отправки данных. Также я добавил проверку количество полей и содержания: должно быть как-минимум одно непустое поле, иначе запрос не будет отправлен.

Пользователю может понадобится исправить или дополнить тот иной запрос истории. Делать это в обычном поле ввода строки неудобно, так как отсутствует подсветка синтаксиса, автодополнение, указание номеров строк. На многих сайтах, публикующих уроки по языкам программирования, можно встретить вставки кода. Как правило, с ним можно взаимодействовать: при нажатии на них, появляется возможность отредактировать код. Я нашёл библиотеку, с помощью которых реализуется такая функциональность (CodeMirror), а также компонент созданный специально для React^[12], который позволяет легко интегрировать его в форму React. CodeMirror позволяет добавить подсветку синтаксиса для любого языка благодаря системе расширений, но писать грамматику для SQL не пришлось: в репозитории

пакетов для JavaScript был найден подходящий. Реализация секции ввода SQL-запросов на картинке:

```
<Form.List
  name="queries"
  rules={[{
    validator: async (_, queries) => (!queries || queries.length < 1) ?
    Promise.reject(new Error('At least 1 queue')) : undefined,
  }]}
>
  {(queries, {add, remove}, {errors}) => (
    <>
      {queries.map((queue, index) => (
        <Form.Item
          {...(index === 0 ? formItemLayout : formItemLayoutWithoutLabel)}
          label={index === 0 ? 'Queries' : ''} required={false}
        key={queue.key}
        >
          <Form.Item
            {...queue}
            validateTrigger={['onChange', 'onBlur']}
            rules={[{
              required: true,
              whitespace: true,
              message: "Please input queue or delete this field.",
            }]}
            noStyle
          >
            <CodeMirror theme="dark" extensions={[sql()]} />
          </Form.Item>
          {queries.length > 1 ?
            <MinusCircleOutlined className="dynamic-delete-button"
            onClick={() => remove(queue.name)} />
            : null
          }
        </Form.Item>
      )]}
      <Form.Item>
        <Button type="dashed" onClick={() => add()} style={{width: '100%'}}
        icon={ <PlusOutlined /> }>
          Add queue
        </Button>
        <Form.ErrorList errors={errors} />
      </Form.Item>
    </>
  )}
</Form.List>
```

Те поля формы, для которых на стороне API не были указаны значения по умолчанию, являются обязательными для заполнения. После формы расположено место для вывода ответов сервера. Изначально, я выводил ответы с помощью React-компонента «Notification», но такие уведомления были слишком маленькими для удобного чтения, особенно если сервер вернул в

качестве ответа стек трассировки ошибки, поэтому от этой идеи с уведомлениями пришлось отказаться.

Далее был оформлен верхний колонтитул страницы с названием приложения. И нижний колонтитул с указанием авторства. Рисунок 7 показывает дизайн клиента приложения после успешного создания задачи на автоиндексирование.

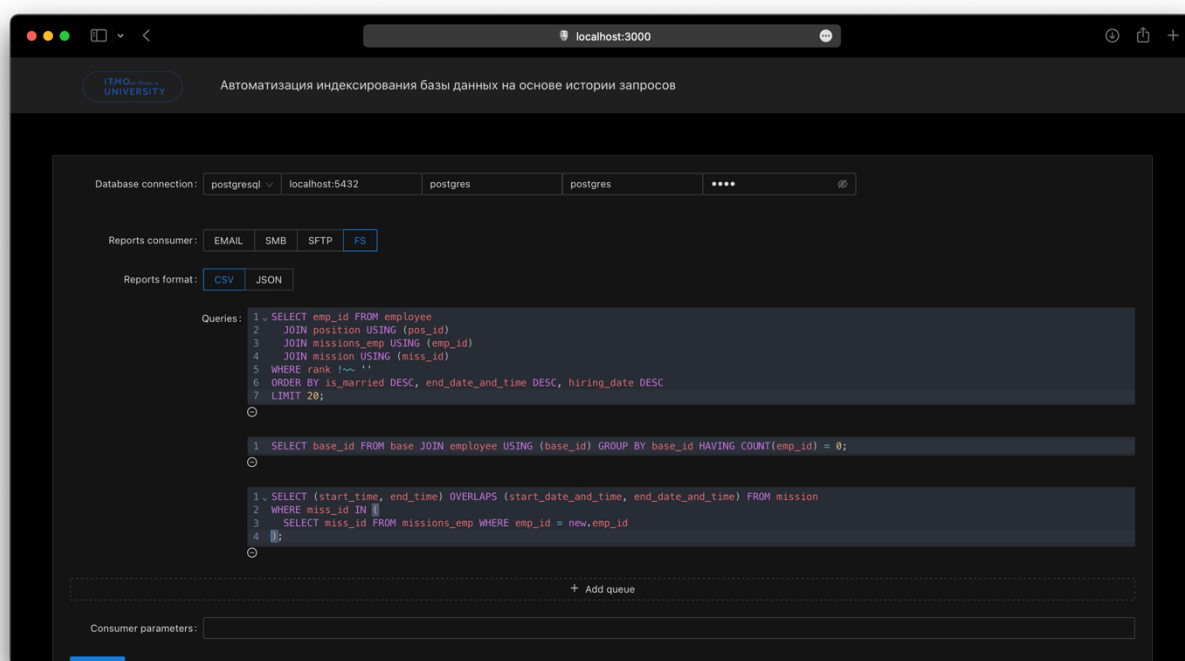


Рисунок 7 - Графический интерфейс приложения

4. АНАЛИЗ РЕЗУЛЬТАТОВ

4.1 Создание тестовой задачи

Для тестирования приложения мною была создана история запросов, установлены все прочие параметры, и поставлена задача на автоиндексирование. [Рисунок 8](#) показывает тело запроса тестируемой задачи, включая историю SQL-запросов в браузере Firefox.

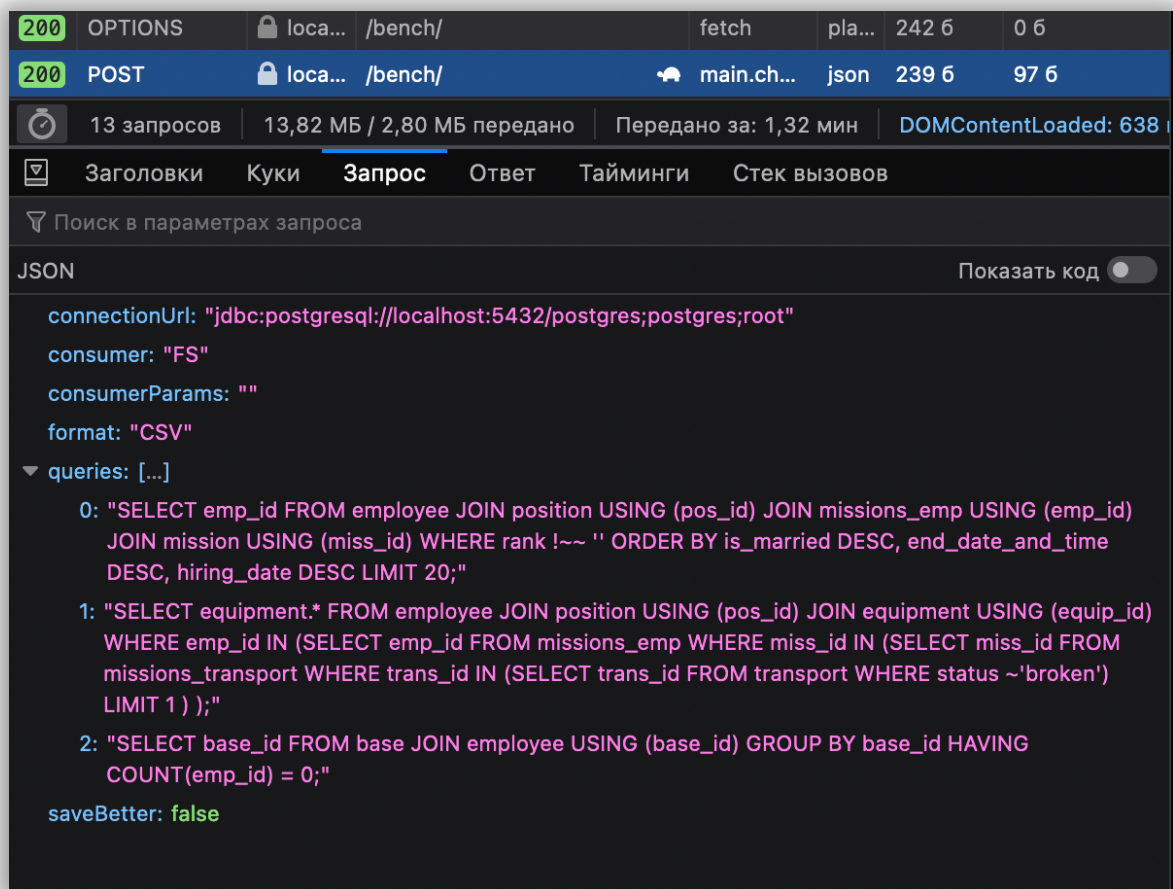


Рисунок 8 - Тестовая задача

Сервер перебрал все возможные комбинации индексов и построил по отчёту на каждый запрос. Проанализируем таблицу (Таблица 2) для первого из трёх выражений (*SELECT emp_id FROM employee JOIN position USING (pos_id) JOIN missions_emp USING (emp_id) JOIN mission USING (miss_id) WHERE rank !~~ '' ORDER BY is_married DESC, end_date_and_time DESC, hiring_date DESC LIMIT 20*),

так как на его основе было создано наибольшее количество вариантов оптимизации.

Таблица 2 - Результаты автоиндексирования для первого запроса

indexStatement	timeTaken	diff
<i>CREATE INDEX Is_marriedEmployeeHASH ON employee USING HASH(is_married);</i>	4956042	173166
<i>CREATE INDEX Is_marriedHiring_dateEmployeeBTREE ON employee USING BTREE(is_married, hiring_date);</i>	5005417	123791
<i>CREATE INDEX End_date_and_timeMissionBTREE ON mission USING BTREE(end_date_and_time);</i>	5050958	78250
<i>CREATE INDEX Emp_idIs_marriedHiring_dateEmployeeBTREE ON employee USING BTREE(emp_id, is_married, hiring_date);</i>	5057833	71375
<i>CREATE INDEX RankPositionHASH ON position USING HASH(rank);</i>	5083375	45833
<i>CREATE INDEX Emp_idMissions_empHASH ON missions_emp USING HASH(emp_id);</i>	5121833	7375
<i>CREATE INDEX End_date_and_timeMissionHASH ON mission USING HASH(end_date_and_time);</i>	5167042	-37834
<i>CREATE INDEX Emp_idMissions_empBTREE ON missions_emp USING BTREE(emp_id);</i>	5215417	-86209
<i>CREATE INDEX Is_marriedEmployeeBTREE ON employee USING BTREE(is_married);</i>	5238875	-109667
<i>CREATE INDEX Emp_idHiring_dateEmployeeBTREE ON employee USING BTREE(emp_id, hiring_date);</i>	5303000	-173792
<i>CREATE INDEX RankPositionBTREE ON position USING BTREE(rank);</i>	5311042	-181834
<i>CREATE INDEX Hiring_dateEmployeeBTREE ON employee USING BTREE(hiring_date);</i>	5338417	-209209
<i>CREATE INDEX Hiring_dateEmployeeHASH ON employee USING HASH(hiring_date);</i>	5543833	-414625
<i>CREATE INDEX Emp_idEmployeeBTREE ON employee USING BTREE(emp_id);</i>	5794584	-665376
<i>CREATE INDEX Emp_idIs_marriedEmployeeBTREE ON employee USING BTREE(emp_id, is_married);</i>	5803750	-674542
<i>CREATE INDEX Emp_idEmployeeHASH ON employee USING HASH(emp_id);</i>	8762875	-3633667

Алгоритм отработал корректно: он перебрал все возможные комбинации индексов, замерил время выполнения оригинального запроса с каждым из них. Можно видеть, что помимо ускорения, индексы также способны и навредить,

увеличив время запроса. Тем не менее, приложение свою цель выполняет – оно способно автоматически уменьшать время запроса к БД: для выбранного запроса время уменьшилось на 4% (1).

$$diff_p = \frac{100 * diff_t}{new_t + diff_t} = \frac{100 * 173166}{4956042 + 173166} \cong 4, \quad (1)$$

где $diff_p$ – разница в процентах; $diff_t$ – разница в наносекундах; new_t – время выполнения после создания индекса в наносекундах.

Если посмотреть на результаты двух других запросов, которые приведены в [ПРИЛОЖЕНИЕ А](#), можно сделать вывод о том, что чем сложнее исходный запрос, тем большее влияние на время выполнения оказывает индекс, что является нормальным поведением для СУБД.

ЗАКЛЮЧЕНИЕ

Целью данной работы являлась автоматизация процесса индексирования базы данных. И цель была достигнута: разработанная система в полной мере соответствует функциональным и нефункциональным требованиям, способна решать поставленные задачи. Была создана расширяемая архитектура, которая позволяет легко добавлять поддержку новых СУБД и потребителей данных.

По первоначальной задумке предполагалось, что оптимизация запросов будет производиться только «по требованию», то есть, когда пользователь решит провести оптимизацию, он запустит задачу автоиндексирования. В процессе написания работы я также понял, что благодаря клиент-серверной архитектуре можно дублировать каждый запрос к БД в разработанную систему, тем самым осуществляю оптимизацию «на лету».

Также, помимо PostgreSQL, была добавлена поддержка СУБД Microsoft SQLServer: я воссоздал модель данных в ней, добавил в приложение новые типы индексов и протестировал процесс автоиндексирования – всё работает, но, в работе это не рассматривается, так как в рамках выпускной квалификационной работы предусматривалась в качестве тестируемой и первой поддерживаемой именно PostgreSQL.

Я нахожу работу полезной, как с точки зрения решаемой приложением задачи, так и с точки зрения развития собственной компетенции в сфере программирования и баз данных.

Главной сложностью стал детальный и подробный процесс проектирование такой сложной системы.

Неожиданным и полезным моментом для меня стал факт того, что неподходящий индекс может даже замедлить базу данных: до этого я считал, что влияние индексов на время выполнения запроса может быть либо положительным, либо отсутствовать.

Полный исходный код приложения можно найти по ссылке (<https://github.com/testpassword/Database-auto-indexing-based-on-query-history>) или через [Рисунок 9](#).



Рисунок 9 - Исходный код приложения

СПИСОК ИСТОЧНИКОВ

1. Международный стандарт ISO/IEC 2382:2015 Information technology – Vocabulary [Электронный ресурс] – URL: <https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en>. – Режим доступа: свободный. – Дата обращения: 02.05.2022.
2. Международный стандарт ISO/IEC TR 10032:2003 Reference Model of Data Managment [Электронный ресурс] – URL: <https://www.iso.org/standard/38607.html>. – Режим доступа: свободный. – Дата обращения: 02.05.2022.
3. Статистика использования СУБД [Электронный ресурс] – URL: <https://db-engines.com/en/ranking>. – Режим доступа: свободный. – Дата обращения: 01.05.2022.
4. Цена на редакции СУБД Oracle Database [Электронный ресурс] – URL: <https://www.oracle.com/assets/technology-price-list-070617.pdf>. – Режим доступа: свободный. – Дата обращения: 01.05.2022.
5. История Java от Oracle [Электронный ресурс] – URL: <https://www.oracle.com/java/moved-by-java/timeline/>. – Режим доступа: свободный. – Дата обращения: 03.05.2022.
6. Статистика использования языков программирования [Электронный ресурс] – URL: <https://www.tiobe.com/tiobe-index/>. – Режим доступа: свободный. – Дата обращения: 03.05.2022.
7. Описание компонентов BPMN на русском языке [Электронный ресурс] – URL: <https://habr.com/ru/company/trinion/blog/331254/>. – Режим доступа: свободный. – Дата обращения: 05.05.2022.
8. Правила спецификации OpenAPI 3 [Электронный ресурс] – URL: <https://swagger.io/specification/>. – Режим доступа: свободный. – Дата обращения: 06.05.2022.

9. Документация библиотеки Java Faker [Электронный ресурс] – URL: <https://github.com/DiUS/java-faker>. – Режим доступа: свободный. – Дата обращения: 06.05.2022.
10. Список библиотек для разбора SQL-выражений на Java [Электронный ресурс] – URL: <https://stackoverflow.com/questions/660609/sql-parser-library-for-java>. – Режим доступа: свободный. – Дата обращения: 09.05.2022.
11. Документация библиотеки QSqlParser [Электронный ресурс] – URL: <https://github.com/JSQLParser/JSqlParser>. – Режим доступа: свободный. – Дата обращения: 09.05.2022.
12. Документация библиотеки React-CodeMirror [Электронный ресурс] – URL: <https://uiwjs.github.io/react-codemirror/>. – Режим доступа: свободный. – Дата обращения: 11.05.2022.
13. Документация языка программирования Kotlin [Электронный ресурс] – URL: <https://kotlinlang.org/docs/home.html>. – Режим доступа: свободный. – Дата обращения: 12.05.2022.
14. Документация библиотеки компонентов Ant Design [Электронный ресурс] – URL: <https://ant.design/docs/react/introduce>. – Режим доступа: свободный. – Дата обращения: 12.05.2022.
15. Документация фреймворка React [Электронный ресурс] – URL: <https://ru.reactjs.org/docs/getting-started.html>. – Режим доступа: свободный. – Дата обращения: 12.05.2022.
16. Документация СУБД PostgreSQL [Электронный ресурс] – URL: <https://www.postgresql.org/>. – Режим доступа: свободный. – Дата обращения: 12.05.2022.
17. Жемеров, Д. Kotlin в действии / Д. Жемеров, С. Исакова; перевод с английского А. Н. Киселев. — Москва: ДМК Пресс, 2018. — 402 с. — ISBN 978-5-97060-497-7.

18. Сомон., П. И. Волшебство Kotlin: руководство / П. И. Сомон. ; перевод с английского А. Н. Киселева.. — Москва: ДМК Пресс, 2020. — 536 с. — ISBN 978-5-97060-801-2.
19. Шёниг, Г. -. PostgreSQL 11. Мастерство разработки / Г. -. Шёниг; перевод с английского А. А. Слинкина. — Москва: ДМК Пресс, 2020. — 352 с. — ISBN 978-5-97060-671-1.
20. Фиайли, К. SQL / К. Фиайли. — Москва: ДМК Пресс, 2008. — 451 с. — ISBN 5-94074-233-5.
21. Программирование на языке Java. Конспект лекций: учебно-методическое пособие / А. В. Гаврилов, С. В. Клименков, Ю. А. Королёва [и др.]. — Санкт-Петербург: НИУ ИТМО, 2019. — 127 с.
22. Никольский, А. П. JavaScript на примерах: учебное пособие / А. П. Никольский. — Санкт-Петербург: Наука и Техника, 2017. — 272 с. — ISBN 978-5-94387-735-3.

ПРИЛОЖЕНИЕ А

Таблица 3 - Результаты автоиндексирования для второго запроса

indexStatement	timeTaken	diff
<i>CREATE INDEX Base_idEmployeeHASH ON employee USING HASH(base_id);</i>	3704416	13376
<i>CREATE INDEX Base_idEmployeeBTREE ON employee USING BTREE(base_id);</i>	3721459	-3667
<i>CREATE INDEX Base_idBaseBTREE ON base USING BTREE(base_id);</i>	5174208	-1456416
<i>CREATE INDEX Base_idBaseHASH ON base USING HASH(base_id);</i>	9110459	-5392667

Таблица 4 - Результаты автоиндексирования для третьего запроса

indexStatement	timeTaken	diff
CREATE INDEX Trans_idMissions_transportBTREE ON missions_transport USING BTREE(trans_id);	1301375	118500
CREATE INDEX Miss_idMissions_empHASH ON missions_emp USING HASH(miss_id);	1353000	66875
CREATE INDEX Trans_idTransportHASH ON transport USING HASH(trans_id);	1357084	62791
CREATE INDEX Trans_idStatusTransportBTREE ON transport USING BTREE(trans_id, status);	1383541	36334
CREATE INDEX Emp_idMiss_idMissions_empBTREE ON missions_emp USING BTREE(emp_id, miss_id);	1388750	31125
CREATE INDEX Trans_idMissions_transportHASH ON missions_transport USING HASH(trans_id);	1393875	26000
CREATE INDEX Miss_idTrans_idMissions_transportBTREE ON missions_transport USING BTREE(miss_id, trans_id);	1403000	16875
CREATE INDEX Miss_idMissions_transportBTREE ON missions_transport USING BTREE(miss_id);	1403083	16792
CREATE INDEX StatusTransportHASH ON transport USING HASH(status);	1426792	-6917
CREATE INDEX Miss_idMissions_empBTREE ON missions_emp USING BTREE(miss_id);	1452333	-32458
CREATE INDEX Miss_idMissions_transportHASH ON missions_transport USING HASH(miss_id);	1457042	-37167
CREATE INDEX Trans_idTransportBTREE ON transport USING BTREE(trans_id);	1481667	-61792

CREATE INDEX StatusTransportBTREE ON transport USING BTREE(status);	1497583	-77708
CREATE INDEX Emp_idMissions_empBTREE ON missions_emp USING BTREE(emp_id);	1550167	-130292
CREATE INDEX Emp_idEmployeeBTREE ON employee USING BTREE(emp_id);	1675709	-255834
CREATE INDEX Emp_idEmployeeHASH ON employee USING HASH(emp_id);	1682459	-262584
CREATE INDEX Emp_idMissions_empHASH ON missions_emp USING HASH(emp_id);	1683000	-263125