**Министерство науки и высшего образования Российской Федерации**

**федеральное государственное автономное образовательное учреждение высшего образования**

**«Национальный исследовательский университет ИТМО»**

**(Университет ИТМО)**

**Факультет** Программной инженерии и компьютерной техники (ФПИ и КТ)

**Образовательная программа:** Системное и прикладное программное обеспечение

**Направление подготовки (специальность)**: 09.03.04, Программная инженерия

О Т Ч Е Т

*о производственной, преддипломной практике*

Тема задания: *Проектирование и разработка системы автоиндексации баз данных на основе истории запросов*

Обучающийся: *Кульбако Артемий Юрьевич, P34112*

Руководитель практики от университета: *Гаврилов Антон Валерьевич, преподаватель (факультет программной инженерии и компьютерной техники)*

Дата: 25.05.2022

# ОГЛАВЛЕНИЕ

[1. ВВЕДЕНИЕ 3](#_Toc104554337)

[2. ОПИСАНИЕ ВЫПОЛНЕНИЯ ИНДИВИДУАЛЬНОГО ЗАДАНИЯ 5](#_Toc104554338)

[2.1 Инструктаж обучающегося 5](#_Toc104554339)

[2.2 Подход к разработке программного продукта 5](#_Toc104554340)

[2.3 Реализация программного продукта 14](#_Toc104554341)

[2.4 Экспериментальные исследования 18](#_Toc104554342)

[2.5 Формирование отчёта по практике и нормконтроль 20](#_Toc104554343)

[3. ЗАКЛЮЧЕНИЕ 21](#_Toc104554344)

[4. СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 22](#_Toc104554345)

# 1. ВВЕДЕНИЕ

Преддипломная практика – заключительный этап образовательного процесса, направленный на проверку и закрепление компетенций обучающегося, полученных в процессе академического обучение и учебной практики путём работы над сформулированный вместе с руководителем практики заданием, включающим в себя создание продукта и защиту программного продукта.

Целью производственной практики является демонстрация учащимся того, что он способен выполнять работу в рамках своей специальности на всех этапах, от проектирования до представления. Проверка навыков осуществляется через выполнение индивидуального задания (*Таблица 1*).

Таблица 1 - Индивидуальное задание

|  |  |  |
| --- | --- | --- |
| **Порядковый № этапа** | **Наименование этапа** | **Задание этапа** |
| 1 | Инструктаж обучающегося | Инструктаж обучающегося по ознакомлению с требованиями охраны труда, техники безопасности, пожарной безопасности, а также правилами внутреннего трудового распорядка. |
| 2 | Подход к разработка программного продукта | Необходимо определить технологии, которые будут использованы для разработки приложения, обосновать выбор. Изучить аналоги, их преимущества и недостатки относительно разрабатываемого продукта. С учётом возможностей и ограничений выбранных технологий, спроектировать архитектуру системы, программные и графические интерфейсы. Спроектировать базу данных, которая будет использоваться для целей тестирования, они должна быть достаточно сложной. |
| 3 | Реализация программного продукта | Разработать программный продукт согласно документам, спецификация, планам предыдущего этапа. |
| 4 | Экспериментальные исследования | Провести тестирования созданного приложения, проанализировать результаты работы программы, исправить ошибки и проблемы, выявленные во время тестирования. |
| 5 | Формирование отчета по практике и нормоконтроль | Формирование отчетной документации по проведенной работе, проверка на заимствование ("Антиплагиат"). |

Представленное задание было оформлено в информационной системе университета, одобрено руководителем, и я принял его к исполнению.

# 2. ОПИСАНИЕ ВЫПОЛНЕНИЯ ИНДИВИДУАЛЬНОГО ЗАДАНИЯ

## 2.1 Инструктаж обучающегося

Преддипломная практика проходит в Университете ИТМО, а потому я ознакомился с документами, расположенными на официальной странице университета[[1]](https://edu.itmo.ru/ru/locallegalacts/), такими как: инструктаж обучающегося по ознакомлению с требованиями охраны труда, техники безопасности, пожарной безопасности, правила внутреннего трудового распорядка. После чего приступил к заданию.

## 2.2 Подход к разработке программного продукта

Ключевым требованием, предъявляемым к разрабатываемой программе, сформулированным в задании на выпускную квалификационную работу, является поддержка широкого спектра оборудования, а именно операционных систем Windows, Linux, macOS и процессорных архитектур arm, x86-64. Необходимо было выбрать язык программирования, который бы позволял запускать программу на всех перечисленных конфигурациях. Чтобы разработка проходила легче, стоило выбрать тот язык, который обладает развитым сообществом и подробной документацией. Я обратился к веб-сайту TIOBE, который занимается оценкой и аудитом программного обеспечения. Специалисты сайта поддерживают ежемесячный рейтинг популярности язык программирования[[2]](https://www.tiobe.com/tiobe-index/), опираясь на данные которого, я решил начать своё исследование.

Из представленного списка, наиболее подходящими кандидатами является Java. Тем не менее, за свою более чем тридцатилетнюю историю, язык успел накопить проблем, в первую очередь связанных с слишком строгим следованием правилам ООП и громоздким синтаксисом. Со временем, появились другие языки, которые исправляли эти проблемы, а их код мог выполняться в JVM (Java Virtual Machine *–* среда выполнения Java-кода).

Одним из таких языков является Kotlin, созданный в Санкт-Петербурге в 2011 году фирмой JetBrains. Имея богатый опыт разработки на Java и контакт с сообществом, они создали язык, который решал проблемы Java и был полностью совместим с ним.

Другой причиной, почему была выбрана именно технология JVM, является протокол JDBC: он обеспечивает 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. Это 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 каждый СУБД.

Приложению также нужен интерфейс взаимодействия с пользователем. Если реализовать взаимодействие с приложением через HTTP-протокол, появится возможность независимо от серверной части приложения, обновлять и улучшать графический интерфейс, использовать альтернативные клиенты. Выбор фронтенд-фреймворка по большей части дело вкуса: каждый из трёх самых популярный фрейворков (Vue, React, Angular) имеет подробную документацию, активное сообщество, библиотеки компонентов, потому я решил использовать React: мне он нравится своим декларативным подходом.

Так как ядро приложения будет является сервером, доступ к которому могут получить несколько клиентов в сети, 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. Хранить достаточно лишь 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.

Для Kolin существует свой серверный фреймворк – Ktor. Он написан с учётом всех плюсов и особенностей языка, поэтому я использовал его, а не Sprint Boot.

Невозможно будет проверить работу программы без существования сложной базы данных. Модель данных будет описана позже, здесь стоит определиться с тестовой СУБД. Стоит выбрать ту, которая имеет несколько разных индексов, 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.

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.

1. Клиент создаёт задание на автоиндексирование, предоставляя историю запросов, данные для подключения к БД и данные для получения отчёта.
2. Сервер выполняет необходимые проверки, к примеру, 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 сообщение об ошибке. Наглядно этот процесс представлен на BPMN-схеме (Рисунок 1).

Рисунок 2 показывает взаимодействие всех компонентов системы.

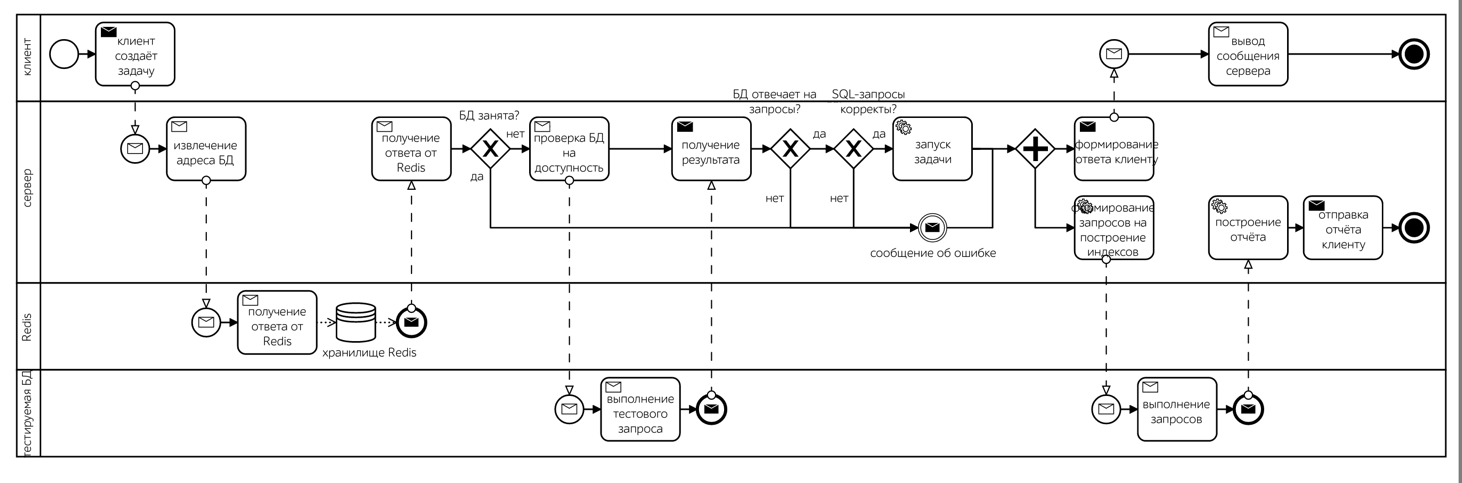


Рисунок 1 - BPMN-схема

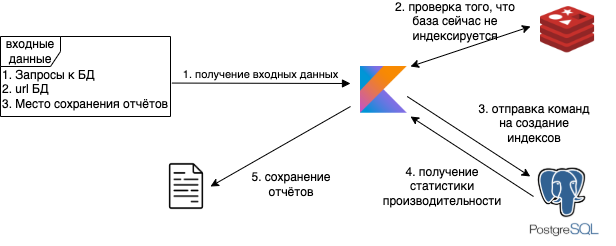


Рисунок 2 - Компонентная схема

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. 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. В качестве основного формата отчётов я выбрал CSV: этот формат, ввиду небольшого количества служебных символов легко читается человеком, а его просмотр доступен в самом популярном средстве форматирования электронных таблиц - Microsoft Excel. Тем не менее, помимо человека, отчёт может понадобится другой клиентской программе, поэтому также необходимо предоставлять его в форме, которую также хорошо и быстро воспринимает вычислительная машина. Самым популярным таким форматов является JSON. На выбор пользователю будут доступны оба формата. 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. 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. На unix-системах самым используемым является SFTP: на машинах, доступ к которым есть по SSH, есть и доступ к нему, а на Windows SMB. Может оказаться полезной отправка отчётов тем, у кого по тем или иным причинам нет доступа ни к диску, ни к машине с приложением: в таком случае подойдёт электронная почта. Актуальный список будет доступен на support/consumers/.

Спецификации API, оформленная в соответствии с стандартом OpenAPI 3[[3]](https://swagger.io/specification/) ниже:

|  |
| --- |
| openapi: 3.0.3 info:  title: SQL-OPTIMIZER  version: 1.1.0 components:  schemas:  FORMATS:  type: string  enum:  - CSV  - JSON  CONSUMERS:  type: string  enum:  - EMAIL  - SMB  - SFTP  - FS  INSTANCES:  type: string  enum:  - POSTGRESQL  - SQLSERVER  CommonMessage:  type: object  properties:  details:  type: string  required: true  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  404:  description: Provided connectionUrl doesn't not match pattern  504:  description: Can't ping database. Please check it's availability and creds or try again later.  500:  description: Error while creating database server specific index for query.  5XX:  description: Something goes wrong  /support/formats:  get:  responses:  200:  description: Enum with supported formats  content:  application/json:  schema:  $ref: '#components/schemas/formats'  /support/consumers:  get:  responses:  200:  description: Enum with supported methods for saving reports  content:  application/json:  schema:  $ref: '#components/schemas/consumers'  /support/instances:  get:  responses:  200:  description: Enum with DBMS which the program can build indexes  content:  application/json:  schema:  $ref: '#components/schemas/instances' |

Чтобы удостоверить в работе приложения, необходимо проверить его на какой-либо базе данных. Эта база должна быть большой по количеству записей и обладать сложной моделью данных. Попробуем спроектировать такую БД. Определим предметную область как Частная военная компания. Новые отношения (таблицы) и их атрибуты можно придумывать бесконечно, а потому, я усложнял модель, пока на это хватало фантазии. Описывать каждую сущность смысла нет, так как процесс исключительно творческий, а не технический, а потому сразу приведу полную схему и выдержку из скрипта создания этой модели.

Полная схема даталогической модели – Рисунок 3.

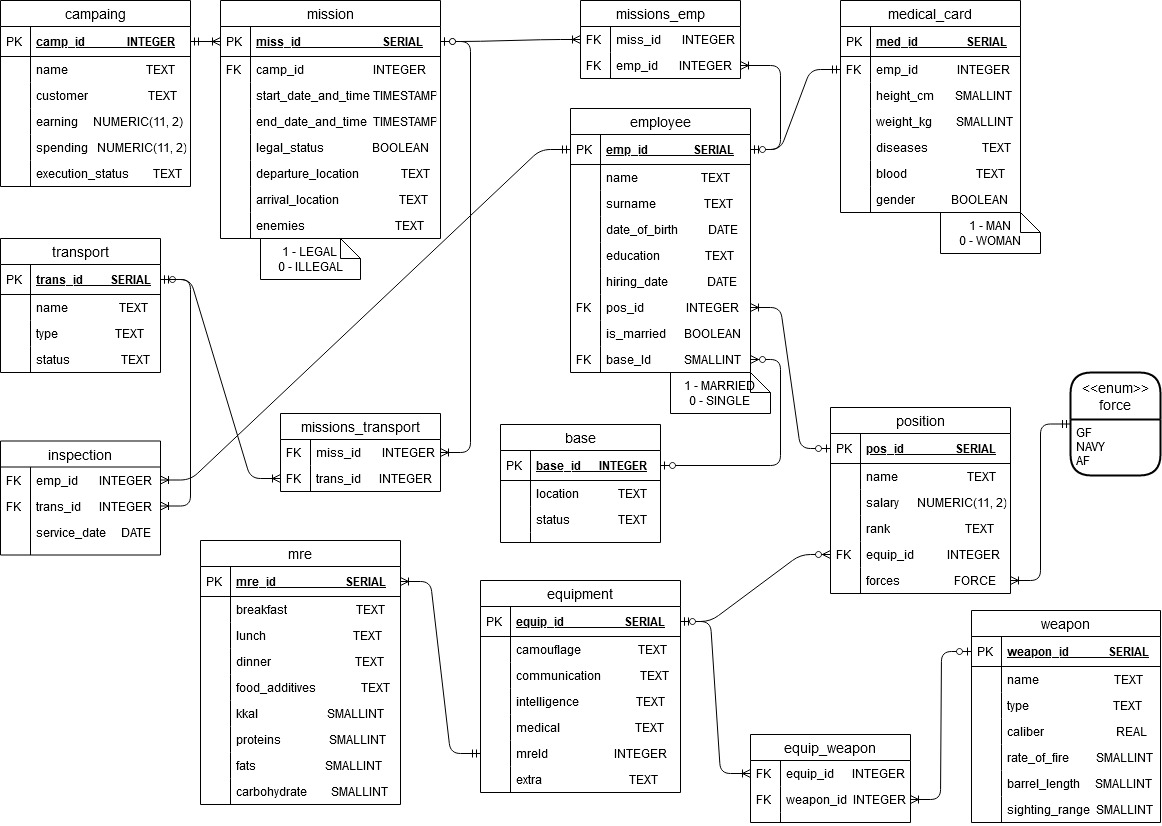


Рисунок 3 - Даталогическая модель тестируемой базы данных

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 генератор на Kotlin. Сложности процесса генерации возникают из-за того, что записи некоторых таблиц обязательно должны содержать ссылки на записи из других таблиц. Таким образом, данные нужно генерировать в паре, либо перед генерацией ведомых данных, делать выборку ведущих данных. Также приходилось учитывать ограничения, накладываемые некоторыми атрибутами, для примера: вес и рост не могут принимать отрицательные значения. Пример генерации одной из сущностей.

|  |
| --- |
| @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()**} ${**F.address().longitude()**}"** it[arrival\_location] = **"${**F.address().latitude()**} ${**F.address().longitude()**}"** it[enemies] = arrayOf(F.nation().nationality(), F.name().fullName()).random()  }  } } |

Запросы, которые будут использоваться для тестирования, нужно делать сложными, а запросы не должны быть абстрактной выборкой, а воссоздавать настоящий 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 опытны.

|  |
| --- |
| **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; |

## 2.3 Реализация программного продукта

Главным требованием, предъявляемым к архитектуре серверной части приложения, является простота расширения, что позволит просто и быстро добавлять поддержку новый СУБД.

Для выполнения SQL-запросов к СУБД используется интерфейс, предоставляемый 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. И использовал средство языка программирования, которое называется перечислением (enum). 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-cхема – Рисунок 4.

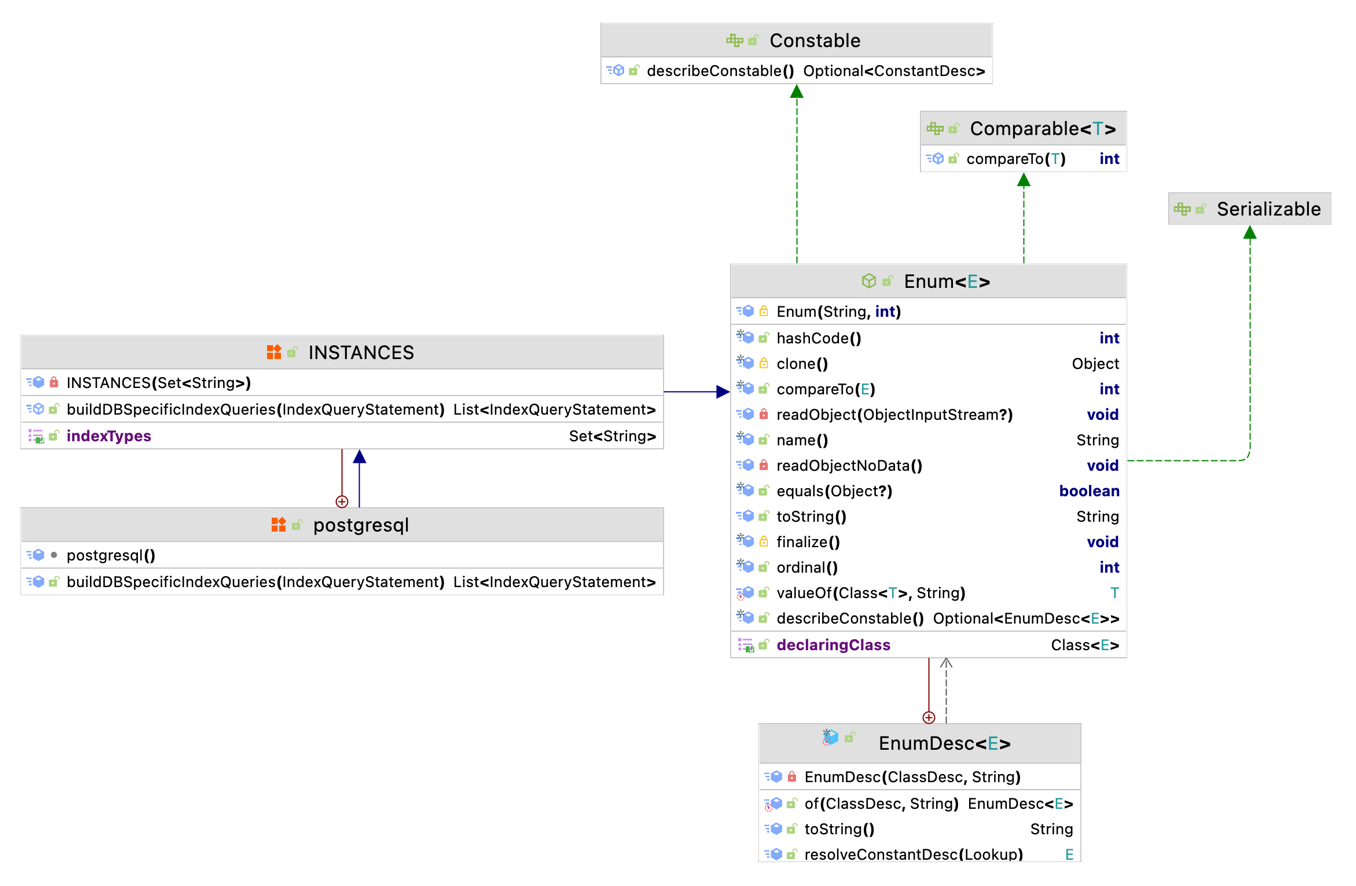


Рисунок 4 - Схема перечисления поддерживаемых СУБД

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, они являются функциями первого класса: их можно передавать как параметр в другие функции.

Пример использования замыкания – функция 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())  **}  }  }** |

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. К сожалению, это библиотека платная. Другим вариантом был ANTLR – 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[[4]](https://github.com/JSQLParser/JSqlParser). Библиотека обладала следующей функциональностью:

* Проверка синтаксиса запроса
* Извлечение имён таблиц
* Добавление псевдонимом (англ. aliases)
* Построение новых запросов

Для формирования множества всех возможных комбинаций индексов необходимо было создать булеан – множество всех возможных множеств. Несмотря на наличие обширного количество методов для работы с коллекциями в Kotlin, такая функция в нём отсутствует, поэтому пришлось создать её самостоятельно.

|  |
| --- |
| **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 5 показывает блок-схему алгоритма трансформации SQL-запроса из истории в отчёт, который будет отправлен пользователю.

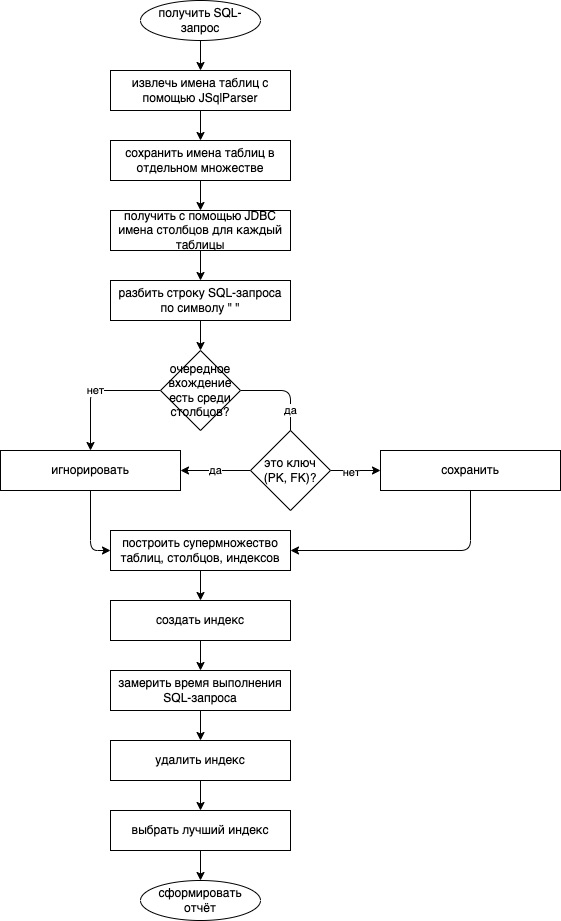


Рисунок 5 - Алгоритм формирования отчёта для SQL-запроса из истории

Так как операция тестирования БД может занять значительное время, выполняться она должна параллельно, чтобы не блокировать сервер. Классическим средством для это в Java являются нити (англ. Thread). Нити позволяют коды выполняться параллельно, происходит это на уровне операционной системы, что избыточно. Kotlin поддерживает сопрограммы (coroutines) – средство для легковесной многопоточности. 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  **}** |

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 выполнятся. Выполнять процесс конвертации будет переопределяемый 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.

Пример функции, которая превращает ANSI SQL-запросы в 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 RESTRICT;"  }  }  else throw** IndexCreationError(indexQuery.**createIndexStatement**)  } }; |

## 2.4 Экспериментальные исследования

Для тестирования приложения мною была создана история запросов, установлены все прочие параметры, и поставлена задача на автоиндексирование. Рисунок 6 показывает тело запроса тестируемой задачи, включая историю SQL-запросов в браузере Firefox.

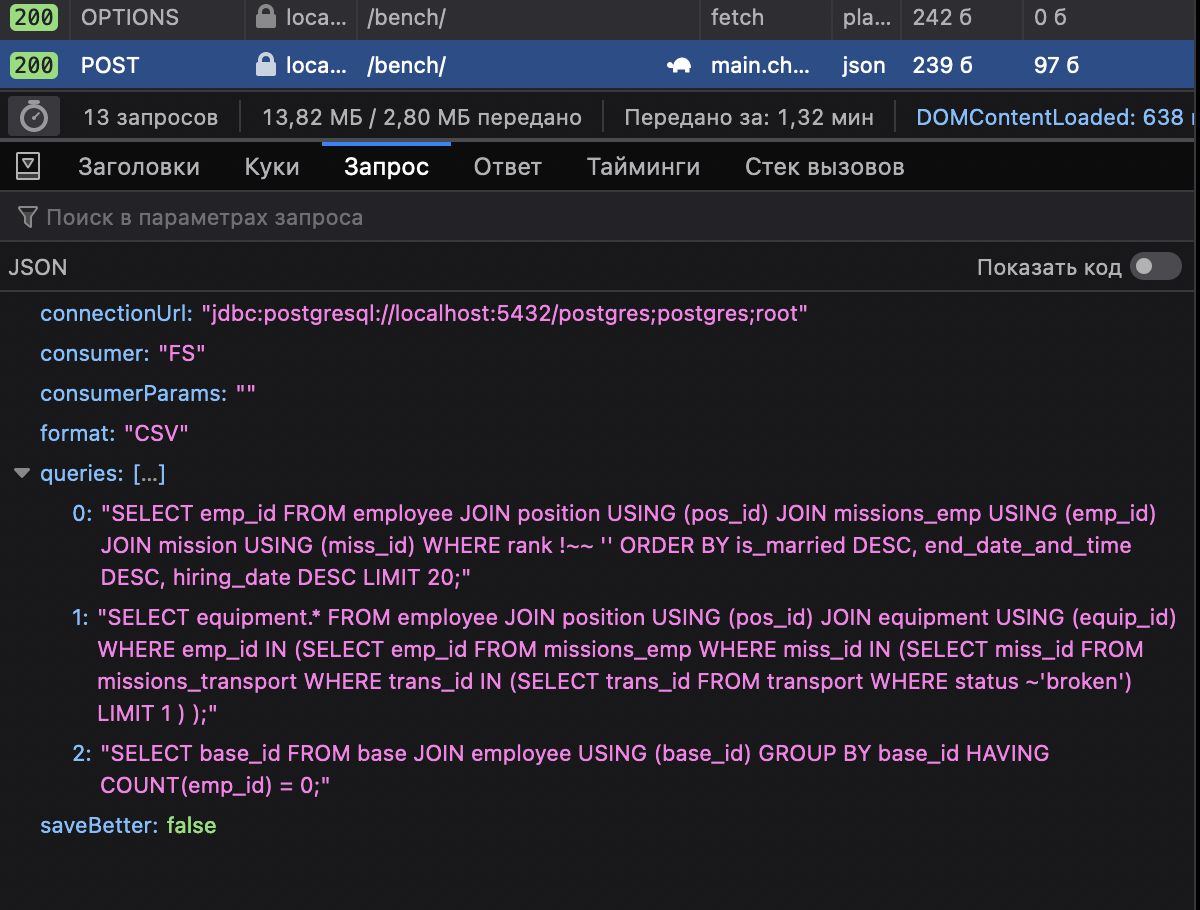


Рисунок 6 - Тестовая задача

Сервер перебрал все возможные комбинации индексов и построил по отчёту на каждый запрос. Данные отчётов были агрегированы в сводную таблицу, где можно увидеть, как уменьшилось время запросов к БД (Рисунок 7).

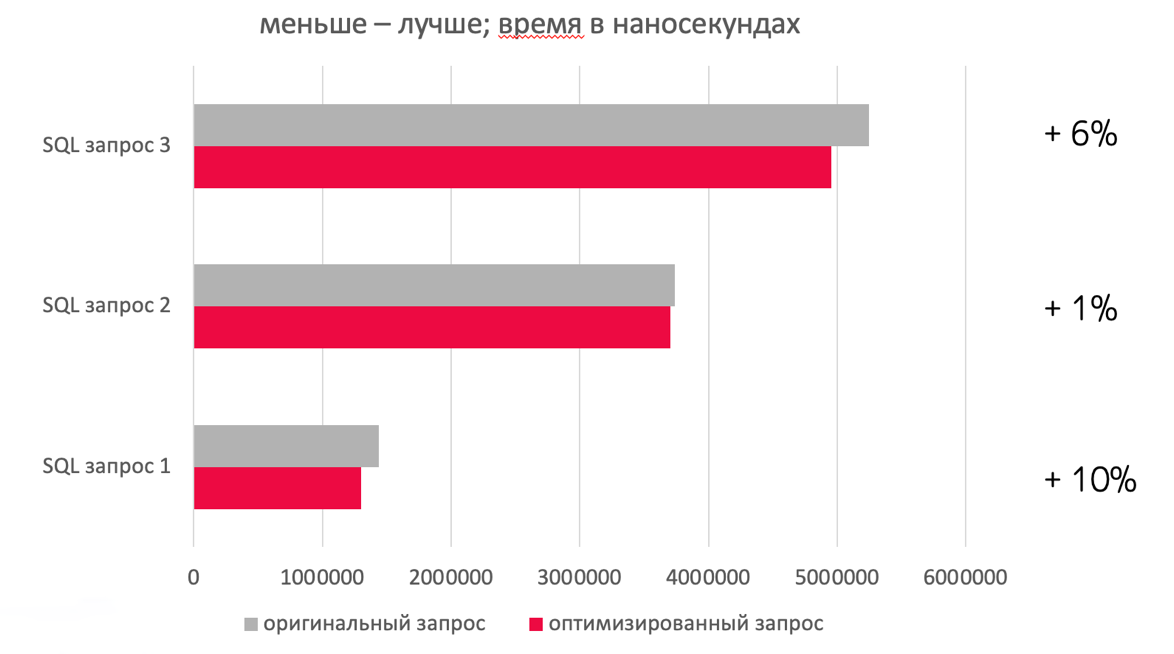


Рисунок 7 - Агрегированная таблица результатов

## 2.5 Формирование отчёта по практике и нормконтроль

После выполнения задания я приступил к оформлению отчёта по практике: это не вызвало никаких сложностей, процесс был отработан в рамках учебной и производственных практик.

Перед нормконтролем и антиплагиатом я несколько раз проверил работу, закрыл доступ к своему репозиторию с работой на Github, оформил список литературы, после чего прикрепил документ выпускной квалификационно й работы в ИСУ.

# 3. ЗАКЛЮЧЕНИЕ

Разработка продукта выпускной квалификационной работы проходила нормально: я сталкивался с трудностями, но типичными для разработки, которые решались поиском в сети, документации, книгах. Хорошо прошёл предзащиту, получил советы от научного руководителя и комиссии советы, что и как можно улучшить.

Работой приложения я доволён и имею планы на его развитие.

# 4. СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Нормативные акты Университета ИТМО [Электронный ресурс] – URL: <https://edu.itmo.ru/ru/locallegalacts/>. – Режим доступа: свободный. – Дата обращения: 24.05.2022.
2. Рейтинг языков программирования [Электронный ресурс] – URL: <https://www.tiobe.com/tiobe-index/>. – Режим доступа: свободный. – Дата обращения: 24.05.2022.
3. Спецификация OpenAPI [Электронный ресурс] – URL: <https://swagger.io/specification/>. – Режим доступа: свободный. – Дата обращения: 24.05.2022.
4. Спецификация JSqlParser [Электронный ресурс] – URL: <https://github.com/JSQLParser/JSqlParser>. – Режим доступа: свободный. – Дата обращения: 24.05.2022.
5. Документация фреймворка React [Электронный ресурс] – URL: <https://ru.reactjs.org/docs/getting-started.html>. – Режим доступа: свободный. – Дата обращения: 12.05.2022.
6. Документация СУБД PostgreSQL [Электронный ресурс] – URL: <https://www.postgresql.org/>. – Режим доступа: свободный. – Дата обращения: 12.05.2022.
7. 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.
8. 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.