МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

(ФГБОУ ВО «ВГУ»)

Факультет компьютерных наук

Кафедра программирования и информационных технологий

*Разработка расширения для PostgreSQL в виде хранилища графовых моделей*

Бакалаврская работа

09.03.02 Информационные системы и технологии

Профиль «Программная инженерия в информационных системах»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Зав. кафедрой | \_\_\_\_\_\_\_\_\_\_\_ | Махортов С. Д. | д.ф.-м.н., профессор | \_\_.\_\_.20\_\_\_ |
| Обучающийся | \_\_\_\_\_\_\_\_\_\_\_ | Мамонов Д. В. |  |  |
| Руководитель | \_\_\_\_\_\_\_\_\_\_\_ | Самойлов Н. К. | ст. преподаватель |  |

МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

(ФГБОУ ВО «ВГУ»)

Факультет компьютерных наук

Кафедра программирования и информационных технологий

|  |  |
| --- | --- |
|  | УТВЕРЖДАЮ  заведующий кафедрой  программирования и информационных технологий  \_\_\_\_\_\_\_\_\_\_\_ С. Д. Махортов  \_\_\_.\_\_\_.2024 |

**ЗАДАНИЕ**

**НА ВЫПОЛНЕНИЕ ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ**

**ОБУЧАЮЩЕГОСЯ** Мамонова Дмитрия Владимировича

1. Тема работы «Разработка расширения для PostgreSQL в виде хранилища графовых моделей», утверждена решением ученого совета факультета компьютерных наук от \_\_ .\_\_.2024

2. Направление подготовки: 09.03.02 Информационные системы и технологии

3. Срок сдачи законченной работы: \_\_.\_\_.2022

4. Календарный план (строится в соответствии со структурой ВКР):

|  |  |  |  |
| --- | --- | --- | --- |
| № | Структура ВКР | Сроки выполнения | Примечание |
| 1 | Введение | 01.11.2023 – 06.11.2023 |  |
| 2 | 1. Постановка задачи | 07.11.2023 – 12.11.2023 |  |
| 3 | 1.1 Цели | 13.11.2023 – 18.11.2023 |  |
| 4 | 1.2 Задачи работы | 19.11.2023 – 26.11.2023 |  |
| 5 | 2. Анализ предметной области | 27.11.2023 – 02.12.2023 |  |
| 6 | 2.1 Анализ существующих решений | 03.12.2023 – 08.12.2023 |  |
| 7 | 2.1.1 Quartz | 09.12.2023 – 15.12.2023 |  |
| 8 | 2.1.2 Guava | 16.12.2023 – 22.12.2023 |  |
| 9 | 2.1.3 Lombok | 23.12.2023 – 28.12.2023 |  |
| 10 | 2.2 Метапрограммирование с Java | 29.12.2023 – 04.01.2024 |  |
| 11 | 2.3 Annotation processing | 05.01.2024 – 10.01.2024 |  |
| 12 | 2.4 Многопоточное программирование на Java | 11.01.2024 – 17.01.2024 |  |
| 13 | 2.5 Средства синхронизации Java | 18.01.2024 – 25.01.2024 |  |
| 14 | 2.6 Представление многопоточных программ | 26.01.2024 – 01.02.2024 |  |
| 15 | 2.7 Запуск задач по расписанию | 02.02.2024 – 10.02.2024 |  |
| 16 | 2.8 Синхронизация на данных (Lock Striping) | 11.02.2024 – 15.02.2024 |  |
| 17 | 2.9 Межсервисные синхронизаторы | 16.02.2024 – 22.02.2024 |  |
| 18 | 3. Выбор средств реализации системы | 23.02.2024 – 27.02.2024 |  |
| 19 | 3.1 JavaPoet | 28.02.2024 – 05.03.2024 |  |
| 20 | 3.2 Redis | 06.03.2024 – 10.03.2024 |  |
| 21 | 4. Реализация | 11.03.2024 – 17.03.2024 |  |
| 22 | 4.1 Библиотека с функциональностью шаблонов многопоточного кода | 18.03.2024 – 23.03.2024 |  |
| 23 | 4.1.1 Scheduling | 24.03.2024 – 28.03.2024 |  |
| 24 | 4.1.2 Lock Striping | 29.03.2024 – 03.04.2024 |  |
| 25 | 4.1.3 Межсервисный синхронизатор | 04.04.2024 – 08.04.2024 |  |
| 26 | 4.2 Библиотека кодогенерации шаблонов многопоточного кода | 09.04.2024 – 13.04.2024 |  |
| 27 | 4.3 Отладка библиотеки кодогенерации | 14.04.2024 – 19.04.2024 |  |
| 28 | 5. Результат работы программы | 20.04.2024 – 23.04.2024 |  |
| 29 | 5.1 Результат кодогенерации | 24.04.2024 – 26.04.2024 |  |
| 30 | 5.2 Вычислительный эксперимент | 27.04.2024 – 30.04.2024 |  |
| 31 | Заключение | 01.05.2024 – 05.05.2024 |  |
| 32 | Список использованных источников | 06.05.2024 – 09.05.2024 |  |

|  |  |  |
| --- | --- | --- |
| Обучающийся | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Мамонов Д. В. |
| Руководитель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Самойлов Н. К. |

РЕФЕРАТ

Бакалаврская работа \_\_ с., \_\_ рис., \_\_ табл., \_\_ использованных источников, \_\_ приложений.

РАСШИРЕНИЕ, ГРАФ СВОЙСТВ, СУЩНОСТЬ, УЗЕЛ, ОТНОШЕНИЕ, ШАБЛОНЫ, SPI.

Объект исследования – взаимодействие с сервером СУБД PostgreSQL посредством расширений на языке низкого уровня, алгоритмы оптимального хранения графов и взаимодействия с ними.

Цель исследования – спроектировать и разработать программную систему, которая будет являться графовым хранилищем для объектов в виде документов и с которыми можно будет взаимодействовать через расширение для PostgreSQL, изучив принципы построения расширений на C и языке pl/pgSQL.

Результаты работы: разработана программная система, представляющее собой графовое хранилище с интерфейсом для СУБД PostgreSQL. Спроектирована структура кода для серверного хранилища графов и алгоримов взаимодействия с ним, парсера специального языка запросов, расширения с интерфейсом к хранилищу. Проведено тестирование программного продукта.

Область применения результатов: разработанная система может применяться для ускорения обработки данных, которые можно представить в виде графов, при этом нет необходимости менять СУБД на проектах как коммерческих, так и образовательных.

СОДЕРЖАНИЕ

[РЕФЕРАТ 5](#__RefHeading___Toc1729037_1399105856)

[СОДЕРЖАНИЕ 6](#__RefHeading___Toc1729039_1399105856)

[ГЛОССАРИЙ 8](#__RefHeading___Toc1729041_1399105856)

[ВВЕДЕНИЕ 10](#__RefHeading___Toc1729043_1399105856)

[1 Постановка задачи 11](#__RefHeading___Toc1729045_1399105856)

[1.1 Цели 11](#__RefHeading___Toc1729047_1399105856)

[1.2 Задачи работы 11](#__RefHeading___Toc1729049_1399105856)

[2 Анализ предметной области 12](#__RefHeading___Toc1729051_1399105856)

[2.1 Анализ существующих решений 12](#__RefHeading___Toc1729053_1399105856)

[2.1.1 Neo4j 12](#__RefHeading___Toc1729055_1399105856)

[2.1.2 Apache AGE 12](#__RefHeading___Toc1729057_1399105856)

[2.2 Модель графа свойств 13](#__RefHeading___Toc1729059_1399105856)

[2.3 Язык графовых запросов Cypher 15](#__RefHeading___Toc1729061_1399105856)

[2.4 Структура графового хранилища 19](#__RefHeading___Toc4879717_1399105856)

[2.6 Алгоритм построения графа 21](#__RefHeading___Toc4879719_1399105856)

[2.6.1 Построение узлов и свойств узлов 21](#__RefHeading___Toc4879721_1399105856)

[2.6.2 Построение ребер графа 21](#__RefHeading___Toc4879723_1399105856)

[2.7 Средства построения расширений СУБД PostgreSQL 23](#__RefHeading___Toc1729065_1399105856)

[3 Выбор средств реализации системы 24](#__RefHeading___Toc1729077_1399105856)

[3.1 libpq 25](#__RefHeading___Toc1729079_1399105856)

[3.2 flex 25](#__RefHeading___Toc1729081_1399105856)

[3.3 bison 26](#__RefHeading___Toc3195014_1399105856)

[3.4 CMake 27](#__RefHeading___Toc3195016_1399105856)

[4 Реализация 29](#__RefHeading___Toc1729083_1399105856)

[4.1 Библиотека с функциональностью шаблонов многопоточного кода 30](#__RefHeading___Toc1729085_1399105856)

[4.1.1 Scheduling 30](#__RefHeading___Toc1729087_1399105856)

[4.1.2 Lock Striping 35](#__RefHeading___Toc1729089_1399105856)

[4.1.3 Межсервисный синхронизатор 38](#__RefHeading___Toc1729091_1399105856)

[4.2 Библиотека кодогенерации шаблонов многопоточного кода 43](#__RefHeading___Toc1729093_1399105856)

[4.3 Отладка библиотеки кодогенерации 45](#__RefHeading___Toc1729095_1399105856)

[5 Результат работы программы 46](#__RefHeading___Toc1729097_1399105856)

[5.1 Результат кодогенерации 46](#__RefHeading___Toc1729099_1399105856)

[5.2 Вычислительный эксперимент 50](#__RefHeading___Toc1729101_1399105856)

[ЗАКЛЮЧЕНИЕ 53](#__RefHeading___Toc1729103_1399105856)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 54](#__RefHeading___Toc1729105_1399105856)

# ГЛОССАРИЙ

PostgreSQL – это система управления базами данных (СУБД) с открытым исходным кодом, реализующая реляционную модель хранения данных и открытая для расширения с пользовательской стороны.

Boot time – время начальной загрузки приложения.

Прокси метод – метод в объекте, подменяющем оригинальный, который оборачивает логику оригинального метода дополнительной логикой.

VarHandle – низкоуровневый механизм для поиска, адаптации и вызова методов.

Метапрограммирование - вид [программирования](https://ru.wikipedia.org/wiki/Программирование), связанный с созданием [программ](https://ru.wikipedia.org/wiki/Компьютерная_программа), которые порождают другие программы как результат своей работы(в частности, на стадии [компиляции](https://ru.wikipedia.org/wiki/Компиляция_(программирование)) их [исходного кода](https://ru.wikipedia.org/wiki/Исходный_код)) либо программ, которые меняют себя во время выполнения.

Фреймфорк - программная платформа, определяющая структуру программной системы.

Байт-код – промежуточное представление Java приложение, набор инструкций, исполняемых виртуальной машиной Java.

Пул потоков - коллекция рабочих потоков, которые эффективно выполняют асинхронные обратные вызовы от имени приложения.

Инстанс – экземпляр.

Шедулинг – запуск задач по расписанию.

Консистентность – согласованность данных друг с другом, целостность данных, а также внутренняя непротиворечивость.

Шард – фрагмент данных.

Утилитный класс – класс, содержащий методы общего назначения.

TTL – time to live, время жизни объектов/данных.

Cron выражение - строка, состоящая из 6 или 7 полей, отделенных между собой пробелами, предназначенная для возможности задавать периодичность и время срабатывания задач планировщика.

Javac – компилятор Java.

JDWP – протокол, реализованный поверх TCP, для удалённой отладки Java приложений.

ВВЕДЕНИЕ

На сегодняшний день наиболее распространенными системами управления базами данных (СУБД) являются программные продукты, основанные на реляционной модели отображения данных, такие PostgreSQL, MySQL, Oracle Database. Несмотря на то, что они отлично справляются со структурированными данными и простыми отношениями, при работе с сильно взаимосвязанными данными или быстро меняющимися схемами возникают трудности и падение производительности. Поэтому во многих отраслях становятся популярными графовые базы данных, которые относятся к классу NoSQL системам управления базами данных. Такие системы позволяют эффективно представлять и запрашивать сложные взаимосвязи между объектами данных. Узлы, ребра и свойства являются основными компонентами графовой базы данных, обеспечивая интуитивно понятную и наглядную модель данных. Но в большинстве как коммерческих проектов, так и продуктов с открытым исходным кодом, кодовая база уже заранее основана на какой-то конкретной системе управления базами данных. Чаще всего как раз таки выбирается реляционная СУБД в виду своей распространенности. Но наиболее распространенной системой управления базами данных является PostgreSQL, которая может интегрировать в себя пользовательские серверные расширения. Данная тема была выбрана для работы, так как разработка подобного расширения, через интерфейс которого осуществлялось бы взаимодействие с графовым хранилищем, используя некоторые заранее предоставляемые SQL-процедуры и язык графовых запросов, способна решить проблему внедрения нового класса СУБД в уже эксплуатируемую систему.

1 Постановка задачи

1.1 Цели

Цели данной работы:

* реализация расширения для СУБД PostgreSQL[1], представляющего собой хранилище для объектов в виде графов.
* исследование технологий взаимодействия с сервером PostgreSQL и клиентской частью с помощью инструментов, предоставляемых библиотекой для разработки пользовательских расширений .

1.2 Задачи работы

Задача состоит в том, чтобы:

* разработать программную систему для хранения графовых моделей с раширением для PostgreSQL со следующими возможностями:
  1. создание графов с возможностью редактирования узлов и ребер;
  2. получение выборки узлов и ребер с помощью декларативного языка;
  3. предоставление SQL-процедур для работы с хранилищем из PostgreSQL.
* протестировать работоспособность продукта;

2 Анализ предметной области

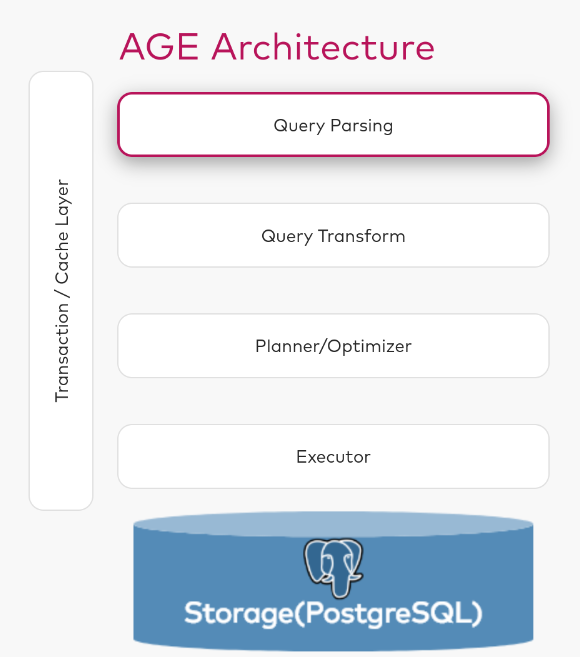
2.1 Анализ существующих решений

2.1.1 Neo4j

Самой распространенной и совершенной графовой СУБД является NoSQL база данных Neo4j[2]. Neo4j получил более широкое распространение на рынке, чем любое другое решение. Фактически, основатель Neo4j Эмиль Эфрайм на самом деле придумал термин "графическая база данных". Данная СУБД является одной из немногих по-настоящему нативных графовых баз данных, которая обеспечивает index-free реализацию для значительного увеличения производительности. Neo4j задает планку соответствия требованиям High Avaliability - от массовых приложений реального времени до приложений для машинного обучения с аналитической ориентацией на графики и графические данные. Благодаря гибкости графовой модели и возможности горизонтального масштабирования, СУБД может успешно применяться в различных сценариях и масштабироваться при необходимости. Также в Neo4j применяется язык запросов Cypher, который стал основой для разработки стандарта языка запросов к графовым СУБД — GQL.

2.1.2 Apache AGE

Apache AGE (Analyze Graph Everywhere)[3] - это расширение для PostgreSQL, которое добавляет поддержку графовых баз данных и запросов к графам в PostgreSQL. Оно позволяет использовать реляционную СУБД как графовую базу данных, что обеспечивает гибкость и удобство работы с графовыми данными. Основным недостатком данного продукта является отсутствие index-free реализации взаимодействия между узлами графов и ребрами, так как в основе расширения лежит отображение графов на реляционную модель данных, что показано на рисунке 1.

Рисунок 1 – Структура расширения Apache AGE

2.2 Модель графа свойств

В качестве модели хранения и обработки данных в графовых системах управления базами данных применяется обычно модель графа свойств[4].

В теории графов он может быть определен как ориентированный, с помеченными узлами и ребрами мультиграф, где ребра могут быть уникально идентифицированы. В графе свойств понятие узла отражает вершину, а понятие отношения – ребра. Такое представление приведено на рисунке 2.

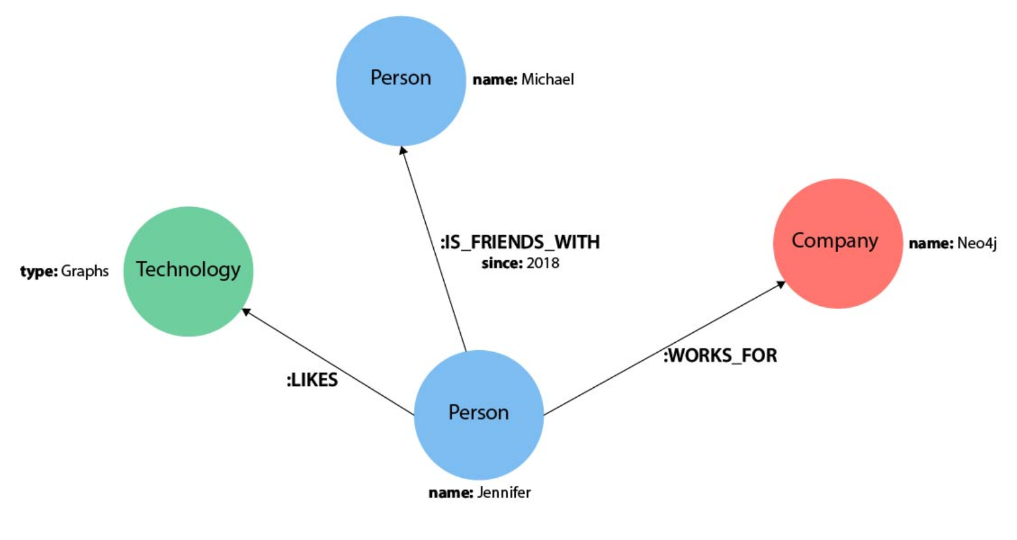


Рисунок 2 – Пример графа свойств

Графовая модель включает в себя следующие понятия: сущность; путь в графе; токен; свойство:

1. Сущность – уникальный объект, который можно сравнить на идентичность с другими аналогичными объектами. Она содержит в себе специальный контейнер свойств – словарь "ключ - значение". Каждое свойство (т.е. "ключ") является уникальным в пределах описания одной сущности. Сущность представлена узлом и отношением между узлами.
2. Узел – базовая сущность графа с уникальными атрибутами, которая может существовать независимо от других сущностей. Он может хранить в себе набор свойств, который не является обязательным. Узел может иметь входящие и исходящие отношения (либо вообще не иметь отношений, связывающих его с остальными узлами).
3. Отношение – сущность, обозначающая направленное соединение между двумя узлами (исходный и целевой узлы). Оно может быть исходящим и входящим. Исходящее отношение рассматривается со стороны исходного узла, входящее – со стороны целевого узла. Отношение может иметь только один тип (или метку).
4. Путь в графе отображает обход в графе свойств и состоит из набора узлов, соединяемых отношениями между собой. Он всегда начинается с определенного узла и заканчивается узлом. Самым кратчайшим путем считается один узел, являющийся и начальным и конечным. Длина пути – характеристика, обозначающая количество отношений в нем.
5. Токен – непустая строка символов в кодировке Unicode. Токенами представлены метки, типы отношений и ключи свойств. Метка присваивается только узлам, тип отношений - отношениям. Ключи свойств - названия уникальных идентификаторов полей в наборе свойств.
6. Свойство – пара «ключ-значение», которая может быть инициализирована значением определенного типа или списком таких значений.

2.3 Язык графовых запросов Cypher

Cypher[5] – специальный декларативный язык запросов, который был разработан инженером Андреасом Тейлором во время его работы в компании Neo4j.Inc в 2011 году. На этапе создания предполагался для взаимодействия только с графовой базой данных Neo4j, но в 2015 году появился проект openCypher, который позволил интегрировать его и с другими СУБД .

После создания проекта openCypher сообществом были предприняты попытки стандартизации языка в качестве базового языка обработки графовых данных, совместимого с языком запросов к реляционным СУБД SQL. Для этого были проведены 5 очных совещаний разработчиков openCypher. Первое совещание состоялось в феврале 2017 года в штаб-квартире SAP в Вальдорфе (Германия) и совпало с заседанием Совета по тестированию связанных данных. Последнее OCIM состоялось в Берлине одновременно с семинаром W3C по веб-стандартам управления графическими данными в марте 2019 года. По итогу работы в сентябре 2019 года был создан проект по разработке GQL (Graph Query Language) , который был одобрен голосованием национальных органов по стандартизации, которые являются членами Объединенного технического комитета ISO/IEC. На текущий момент стандарт находится в разработке и еще не является общедоступным.

Cypher использует ASCII-представление для описания запросов, т.е. информация должна быть наглядно визуализирована для пользователя. Это делает язык очень наглядным и удобным для чтения, поскольку он как визуально, так и структурно представляет данные, указанные в запросе. Запросы Cypher объединяются с шаблонами узлов и связей с любой указанной фильтрацией по меткам и свойствам для создания, чтения, обновления, удаления данных, найденных в указанном шаблоне.

Самое простое описание узла это его имя, заключенное в круглых скобках. Дополнительно при описании узла могут указываться его метка (например, при выборе определенных узлов) и тело свойств (например, при создании нового отношения), что представлено на листинге 1 и рисунке 3.

Листинг 1 – Описание узлов

(s) // 1. Простейшее описание

(s:Person) // 2. Описание с меткой

(s:Person{name: 'Tom', age: 18}) // 3. Описание со свойствами

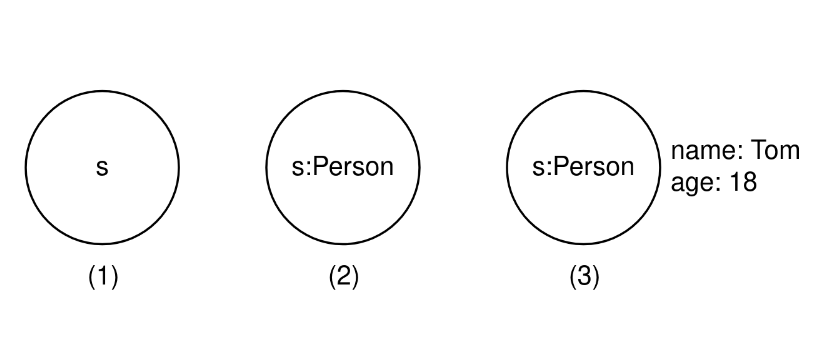


Рисунок 3 – Визуализация узлов

В случае если набор свойств отображен в выборке, то он накладывает дополнительное ограничение на ее результат. При выполнении запроса CREATE набор будет установлен по умолчанию в новых узлах и отношениях.

Для описания простейшего отношения между двумя узлами используется конструкция вида(a)-[]-(b). В этом случае такие отношения будут образовывать неориентированный граф. Для того чтобы указать направление отношения, используются знаки ->, представляющие собой «стрелочку». Как и узлы, отношения могут иметь имена и метки, обозначающие их тип (или класс, к которому они относятся). Описать отношение, которое имеет определенный тип, можно, указав в квадратных скобках идентификатор отношения и его тип через знак двоеточия. В отличие от узлов, при задании нового отношения или при построении шаблона имя отношения можно опустить и использовать только название метки, как показано в листинге 2 и на рисунке 4 .

Листинг 2 – Описание отношения между узлами

// 1. Простейшее неориентированное отношение

(a)-[]-(b)

// 2. Простейшее ориентированное отношение

(a)-[]->(b)

// 3. Отношение с указанием его типа

(a)-[r:REL\_TYPE]->(b)

// 4. Отношение с указанием его типа и с опущением идентификатора

(a)-[:REL\_TYPE]->(b)

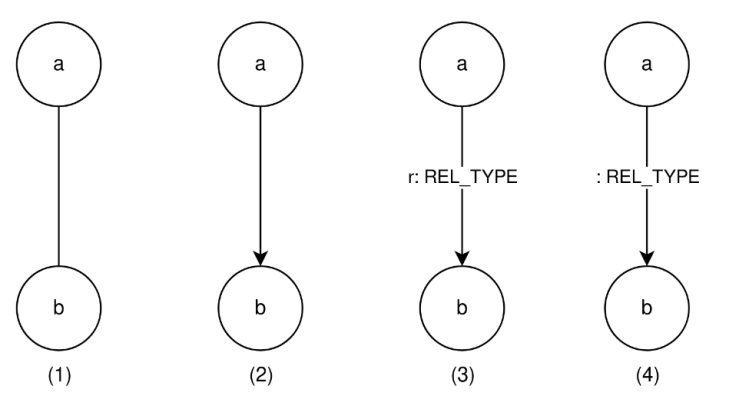


Рисунок 4 - Визуализация отношений в графе

Подобно другим языкам запросов, Cypher содержит множество ключевых слов для задания шаблонов, фильтрации шаблонов и возврата результатов. Наиболее распространенными для построения запросов являются MATCH и RETURN (представлены в листинге 3 и на рисунке 5).

Ключевое слово MATCH обозначает поиск существующих узлов, отношений, меток, свойств или шаблонов в СУБД. Оно является аналогичным слову SELECT в SQL. В запросах, построенных с MATCH, можно производить поиск всех меток и типов отношений, узлов и отношений на основе шаблонов и т.д.

Ключевое слово RETURN в Cypher указывает, какие значения или результаты можно вернуть из запроса. В качестве объектов, которые можно возвращать, могут использоваться узлы, отношения, свойства узлов и отношений, шаблоны. RETURN не требуется при выполнении запросов записи, но необходим для выборки. Названия переменных узла и отношения становятся необходимыми при использовании RETURN. Чтобы вернуть узлы, отношения, свойства или шаблоны, в запросе MATCH должны быть указаны переменные для данных, которые надо вернуть.

Листинг 3 - Примеры запросов на шаблонах с MATCH и RETURN

MATCH (p1:Person)-[:PARENT]->(p2:Person)

RETURN p1, p2

MATCH ans = (:Person {name: "Pam"})-[:PARENT]->(:Person {name: "Bob"})

RETURN ans

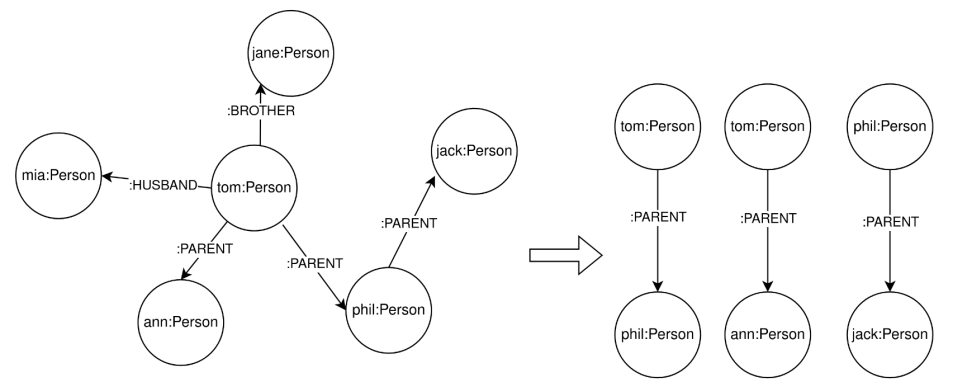


Рисунок 5 - Визуализация выполнения запроса выборки

Редактирование базы данных (создание новых узлов и отношений, их изменение и удаление) обеспечивается при помощи ключевых слов CREATE, SET и DELETE.

Запрос CREATE является отдельно выполняемым запросом (как и INSERT в SQL), но при создании отношений может являться частью MATCH. Он позволяет создавать как отдельные узлы и отношения из уже имеющихся узлов, так и целый граф, в котором будут содержаться новые данные. Примеры запросов представлены в листинге 4, результаты — на рисунке 6.

Листинг 4 - Запросы на создание сущностей через CREATE

CREATE (n:Person)

CREATE (n:Person {name: 'Andres', title: 'Developer'})

MATCH (a:Person), (b:Person)

WHERE a.name = 'Node A' AND b.name = 'Node B'

CREATE (a)-[r:RELTYPE]->(b)

RETURN r

CREATE pam: Person{name: "Pam"},

bob: Person{name: "Bob"},

(pam: Person)-[:PARENT]->(bob: Person)

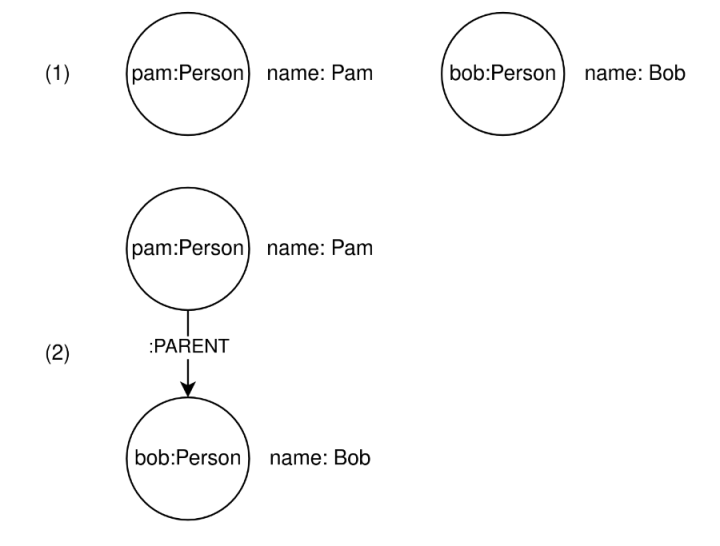


Рисунок 6 - Визуализация процесса создания сущностей в базе данных

Запросы SET и DELETE выполняются только в теле запроса MATCH и могут проводиться как над всеми выбранными данными, так и над результатом их фильтрации, как показано в листинге 5. Узлы разрешается удалять только после того, как они не будут включены ни в одно отношение.

Листинг 5 - Запросы на изменение данных в графе

MATCH (n {name: 'Andres'})

SET n.surname = 'Taylor'

RETURN n

MATCH (n:Useless)

DELETE n

MATCH (p1:Person)-[r]->(p2:Person)

DELETE r

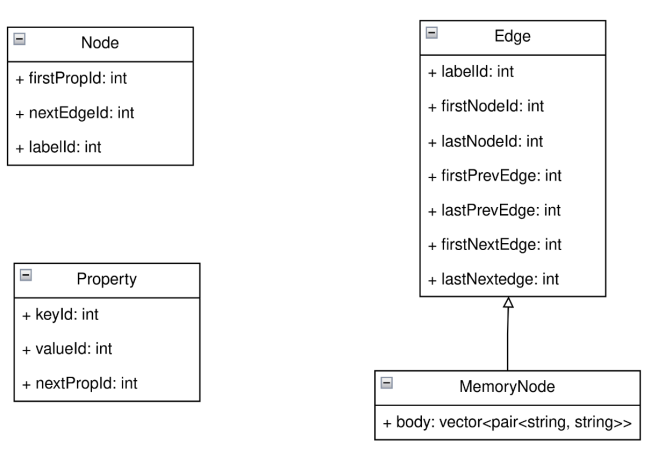
2.4 Структура графового хранилища

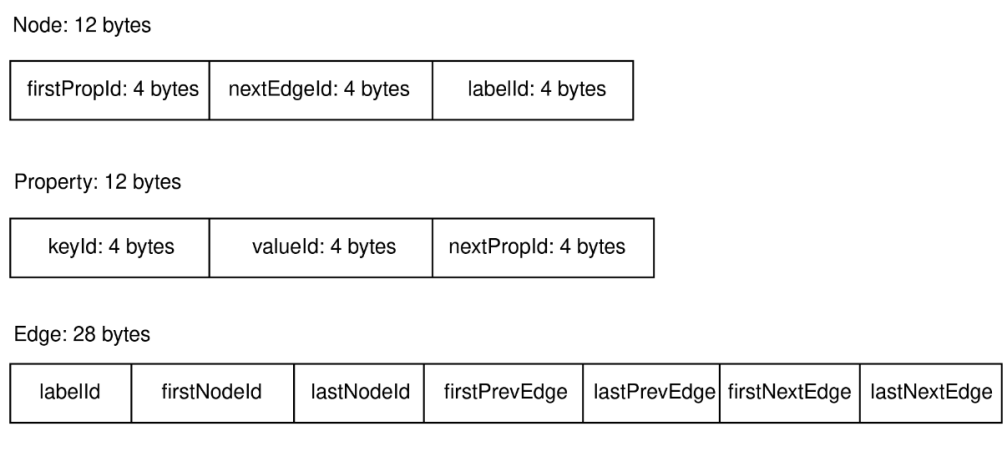
Графовое хранилище состоит из нескольких бинарных файлов, каждый из которых отвечает за отдельный компонент графа свойств:\

* узлы графа;
* ребра графа;
* свойства узлов графа;

Отдельно можно выделить текстовый файл, в котором хранятся все строковые значения для свойств, названий объектов и меток узлов и ребер.

Каждая сущность в файле представлена непрерывным списком байтов и состоит только из беззнаковых целочисленных полей, что отражает диаграмма классов на рисунке 7. Это необходимо для поддерживания выравнивания по байтам и оптимизации во время чтения структуры сущности из файла. Порядок расположения значений для каждой сущности показан на рисунке 8. Все сущности записываются в файле по порядку их добавления в хранилище, и зная их номер, под которым они были записаны, и размер структуры, которая отражает сущность, можно вычислить смещение для сдвига начала потока для чтения в файле и прочитать оттуда ту сущность, которая необходима.

Рисунок 7 — Диаграмма класса сущностей в графовом хранилище

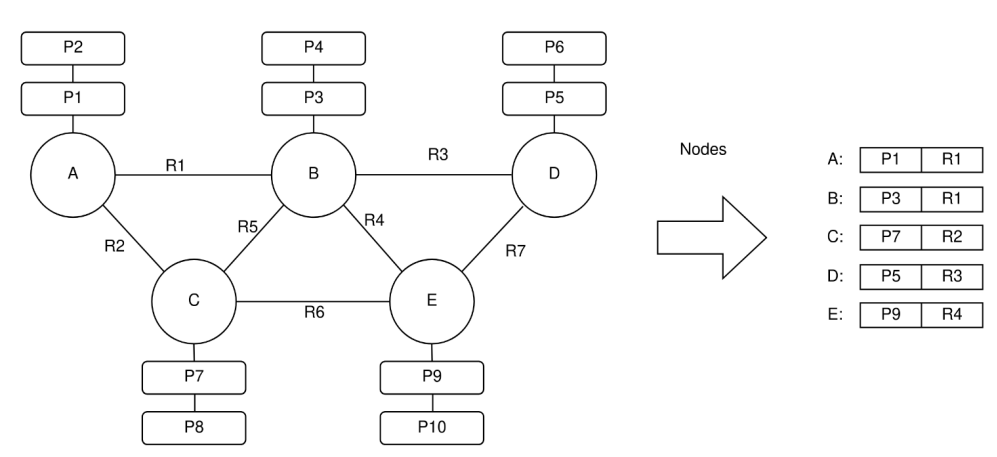
Рисунок 8 — Представление сущностей в файлах

2.6 Алгоритм построения графа

Алгоритм основан на использовании двусвязных списков для ребер и односвязных списков для свойств узлов.

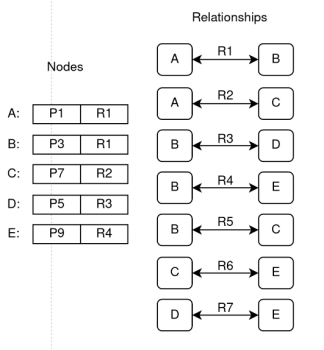
2.6.1 Построение узлов и свойств узлов

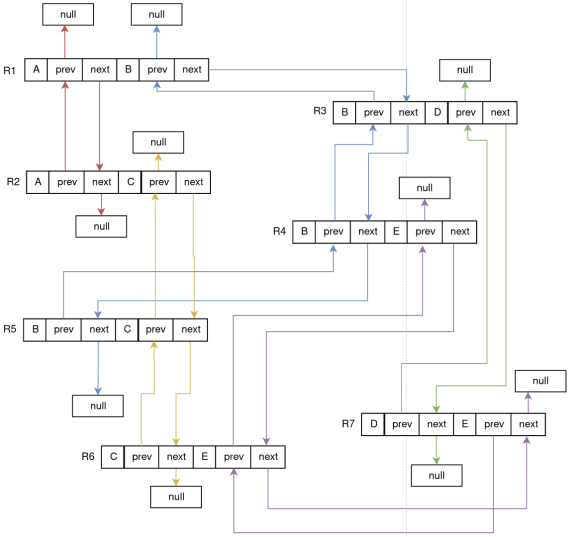
Пусть задан граф как на рисунке 9, и его необходимо перевести в вид сущностей, представленных выше. Сначала для каждого узла определяется первое ребро по счету, в котором узел был задействован: его номер по счету в общем списке ребер заносится в поле nextEdgeId. После этого для каждой пары «ключ-значение» из тела свойств в текстовый файл заносятся названия ключа и значения, которое он хранит; номера по счету в файле заносятся в структуру в поля keyId и valueId, а в поле nextPropId заносится номер следующего узла. Номер первого в списке свойства заносится в структуру узла в поле firstPropId, таким образом для восстановления всего списка свойств необходимо по номеру первого свойства в узле прочитать его структуру и дальше идти как по односвязному списку.

Рисунок 9 — Определение узлов в заданном графе

2.6.2 Построение ребер графа

После этого для каждого узла в ребре ищутся первые попавшиеся ребра из заданного списка ребер, как на рисунке 10, в которых узел задействован. Если такие найдены, то между ними устанавливаются двунаправленные связи: в поле firstNodeId и lastNodeId заносятся первый и второй узлы соответственно; в поле firstNextEdge заносится номер ребра, которое идет следующим после этого для первого узла; в следующем ребре поле firstPrevEdge хранит в себе номер предыдущего ребра и так далее для каждого узла по всем ребрам. В итоге получается множественный двусвязный список, схема которого представлена на рисунке 11. Преимущество такого варианта хранения заключается в быстром поиске соседей для определенного узла и реализации свободного индекса для перемещения между ребрами. Для индекса не выделяется никакая дополнительная память и к ней не происходит обращений, так как его как такового не существует; так как сущности графа представляют собой структуры небольшого размера (не больше трех десятков байт), то операция чтения структуры из файла как самая дорогостоящая будет выполняться с наименьшем временем по сравнению с обращением к файлам реляционной СУБД.

Рисунок 10 — Заданный исходный список ребер и построенные узлы

Рисунок 11 — Результат построения графа

2.7 Средства построения расширений СУБД PostgreSQL

PostgreSQL предлагает расширяемую архитектуру и реализует поверх нее свои внутренние типы данных, операторы, функции, индексы и многое другое[6]. Эта архитектура открыта для каждого, кто может реализовать и добавить свою собственную функциональность в систему управления базами данных. Архитектура СУБД предоставляет возможностями определять новые типы данных со специальными операторами и функциями или без них, в зависимости от варианта использования.

Со временем сообщество разработало набор расширений, полезных для их собственных нужд и для большого количества приложений - иногда даже для требований и определений, данных организациями по стандартизации. Вот некоторые популярные примеры:

1. типы данных, операторы и функции для обработки пространственных данных, таких как точки, полилинии, перекрытия и тд — расширение OSGeo;
2. функциональность для полнотекстовогопоиска;
3. доступ к данным за пределамитекущей базы данных (другой экземпляр PostgreSQL, другая система баз данных SQL, NoSQL или BigData, LDAP, плоские файлы, такие как CSV, JSON, XML).

Жизненный цикл расширения начинается с реализации его функций группой лиц или компанией. После публикации расширение может использоваться и расширяться другими лицами или компаниями сообщества. Иногда такие расширения остаются независимыми от системы PostgreSQL, например, PostGIS, в других случаях они поставляются со стандартной загрузкой и явно указаны как дополнительный поставляемый модуль в документации с подсказками по их установке. И в редких случаях расширения включаются в основную систему, так что они становятся родной частью PostgreSQL.

Для того, чтобы воспользоваться расширением, необходимо загрузить и установить все его файлы (если они не поставляются со стандартной загрузкой) и выполнить команду CREATE EXTENSION <extension\_name>;в SQL-клиенте, подобном консольному psql. Чтобы контролировать, какие расширения уже установлены, применяется команда \dx внутри psql.

3 Выбор средств реализации системы

В качестве средств реализации разрабатываемой системы были выбран язык программирования C++17 вместе со стандартной библиотекой шаблонов STL, а также следующие инструменты:

* libpq — библиотека на низкоуровневом языке C, которая предоставляет интерфейс SPI для обращения к серверу PostgreSQL;
* flex – генератор лексического анализатора входящего текста по токенам;
* bison — генератор синтаксического анализатора входящего текста по специальным правилам из лексем;
* CMake — система сборки проектов с открытым исходным кодом.

3.1 libpq

libpq[7] – библиотека, которая предоставляет интерфейс SPI (Server Programming Interface) к серверу СУБД PostgreSQL. libpq предоставляет набор функций для работы с PostgreSQL через протокол клиент-сервер. Библиотека обеспечивает безопасное подключение к базе данных и защиту от SQL-инъекций, предоставляя возможность использовать параметризованные запросы вместе с шифрованием данных запроса по протоколам TLS/SSL. libpq поддерживает выполнение асинхронных запросов, поэтому обращение к серверу СУБД происходит в параллельном режиме работы. Продукт имеет множество оберток на различных языках программирования, в том числе и на C++.

3.2 flex

flex[8] — генератор лексических анализаторов текста, представляющий собой переработку генератора lex из поставки операционной системы UNIX под лицензией GNU GPL. Задача лексического анализатора состоит в том, чтобы по заданным шаблонам в специальном конфигурационном файле сгенерировать код программы на языке C или C++, который будет разбивать входящий текст на отдельные его части — лексемы, с помощью которых будет производиться синтаксический анализ текста. В качестве языка написания шаблонов поиска лексем используется язык регулярных выражений из стандарта POSIX. Пример задания обработки лексем приведен в листинге 6.

Листинг 6 — Регулярные выражения для поиска лексем

[0-9]+"."[0-9]\* {

yylval->doubleVal = atof(yytext);

return token::DOUBLE;

}

[A-Za-z][A-Za-z0-9\_,.-]\* {

yylval->stringVal = new std::string(yytext, yyleng);

return token::STRING;

}

/\* пропускаем пробелы и табуляции \*/

[ \t\r]+ {

yylloc->step();

}

Для получения анализатора регулярные выражения необходимо сохранить в файле с расширением .l. После этого он подается в качестве аргумента командной строки программе flex, на выходе который получается файл с кодом на языке программирования C. Для его использования или компиляции необходимо подключать библиотеку libflex.

3.3 bison

bison[10] - генератор синтаксических анализаторов текста, представляющий собой переработку генератора yacc из поставки операционной системы UNIX под лицензией GNU GPL. Синтаксические анализаторы генерируются на основе контекстно-свободных грамматик, заданных в текстовом формате. Эти грамматики определяют синтаксис анализируемого языка, подробно описывая, как последовательности лексем (единиц лексикографии) могут быть объединены для формирования допустимых утверждений или выражений. bison поддерживает стратегии синтаксического анализа LL(1) и LALR(1), что делает его универсальным для широкого спектра грамматик. Анализаторы LL(1) работают сверху вниз, в то время как анализаторы LALR(1) работают снизу вверх, каждый из которых имеет свои преимущества и применим к различным типам грамматик. bison обычно интегрируется вместе с flex, которые образуют полную систему парсинга текста, как лексического, так и синтаксического.

Чтобы использовать bison, создается грамматический файл с правилами обработки лексем, которые определяет синтаксис и семантику языка. Обычно этот файл имеет расширение ".y" и содержит объявления для токенов, грамматических правил и действий. Пример приведен в листинге 7.

Листинг 7 — Пример правил для bison

constant : INTEGER

{

$$ = new CNConstant($1);

}

| DOUBLE

{

$$ = new CNConstant($1);

}

variable : STRING

{

if (!driver.calc.existsVariable(\*$1)) {

error(yyloc, std::string("Unknown variable \"") + \*$1 + "\"");

delete $1;

YYERROR;

}

else {

$$ = new CNConstant( driver.calc.getVariable(\*$1) );

delete $1;

}

}

Как только файл грамматики написан, bison обрабатывает его для создания исходного файла на C (или другом языке, в зтом числе и на C++), содержащего синтаксический анализатор. Затем этот исходный файл можно скомпилировать и связать с другими частями программного проекта с помощью библиотеки libyacc.

3.4 CMake

CMake[11] — система сборки проектов с открытым исходным кодом.н позволяет абстрагироваться от сложностей различных сред сборки и цепочек инструментов, позволяя разработчикам писать независимые от платформы сценарии сборки, которые могут использоваться в различных операционных системах. CMake использует независимый от платформы формат файла конфигурации (CMakeLists.txt), который определяет параметры проекта, зависимости и инструкции по сборке. Данный проект предоставляет возможности автоматического поиска зависимостей в виде системных библиотек и пользовательских, добавленных в виде модулей с расширением .cmake; интеграцию с пользовательскими расширениями (например CPack); интеграцию с различными IDE. Файл CMakeLists.txt содержит описание зависимостей, необходимых переменных окружения для комбинации и правил сборки компонентов проекта. Его пример приведен на листинге 8.

Листинг 8 — Пример простого файла CMakeLists.txt

cmake\_minimum\_required(VERSION 3.1)

project (hello\_cpp11)

add\_executable(hello\_cpp11 main.cpp)

target\_compile\_features(hello\_cpp11 PUBLIC cxx\_auto\_type)

После этого в каталоге, в котором будет производиться сборка, выполняется команда cmake с путем к каталогу, в котором лежит файл CMakeLists.txt. В зависимости от платформы, на котором будет производиться сборка, будут сгенерированы соответствующие файлы, после чего останется выполнить сборку для них.

4 Реализация

Разработка производится в Intellij Idea Ultimate Edition 2021.3.

Intellij Idea – интегрированная среде разработки программного обеспечения для Java, JavaScript, Python и других языков программирования от компании JetBrains.

Ultimate Edition – версия, предназначена для фулстек-разработки и создания корпоративных приложений. Поддерживает широкий набор фреймворков и технологий для бэкенда и фронтенда и включает инструменты для профилирования и работы с базами данных, HTTP-клиент и много других функций.

Выбор IDE обусловлен обширным набором инструментов для рефакторинга (перепроектирования), оптимизации кода, проведения тестирования, а также наличием большого количества дополнений и расширений, облегчающих процесс разработки программного обеспечения.

В ходе проектирования системы, она была разбита на две библиотеки для разделения ответственности, возможности использовать шаблоны многопоточного программирования без кодогенератора и уменьшения объёма генерируемого кода. Библиотеку содержащую основную функциональность для использования шаблонов многопоточного кода и библиотеку, отвечающую за кодогенерацию по этим шаблонам. Схема использования данных библиотек показана на рисунке 4.

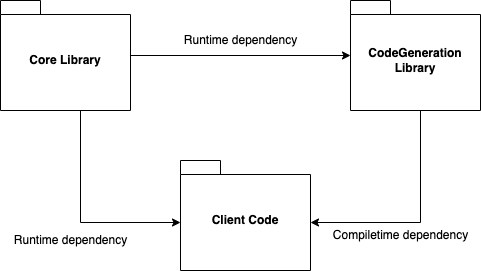


Рисунок 4 – Диаграмма пакетов программной библиотеки для генерации кода

4.1 Библиотека с функциональностью шаблонов многопоточного кода

4.1.1 Scheduling

Выполнение задач по расписанию – широко распространённый подход в программировании, существуют библиотеки, предоставляющие реализацию данного шаблона и позволяющие с минимальными усилиями написать код для решения такой задачи. Для реализации данного шаблона необходима сущность, которая будет содержать задачи в себе и запускать их, когда придёт время. Отдельно стоит рассмотреть ситуацию, когда предыдущий экземпляр задачи ещё не завершил работу, а время для запуска нового экземпляра уже наступило. Система, реализованная в рамках данной работы, предлагает следующие решения для таких ситуаций:

* запустить задачу немедленно, не обращая внимания на предыдущий экземпляр задачи;
* пропустить запуск нового экземпляра;
* отложить запуск нового экземпляра в очередь и выполнить при первой возможности.

Схематично данные стратегии представлены на рисунке 5.

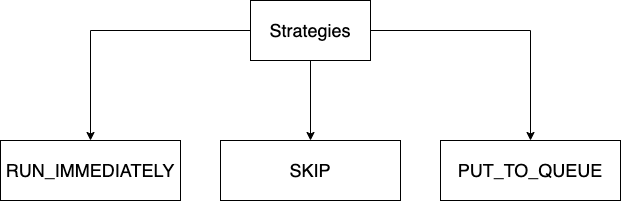


Рисунок 5 – Стратегии запуска задач по расписанию

На рисунке 6 представлена диаграмма классов для шаблона запуска задач по расписанию.

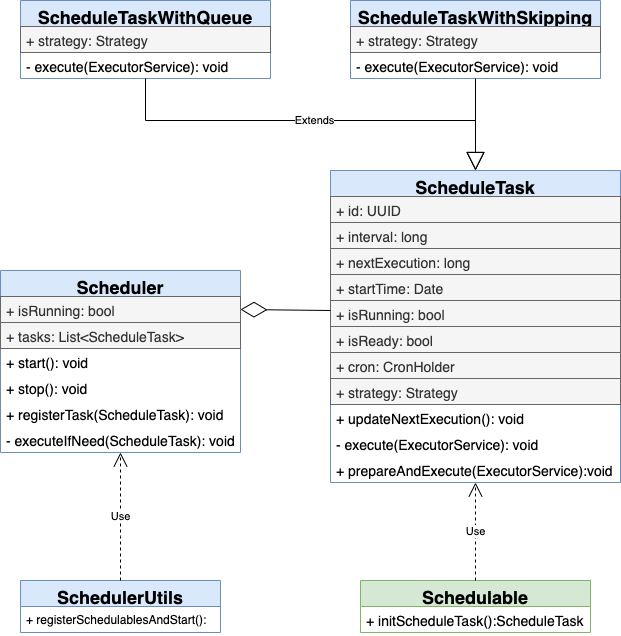


Рисунок 6 – Диаграмма классов для запуска задач по расписанию

Основными классами для запуска задач по расписанию являются Scheduler, сам планировщик задач и ScheduleTask, задача для запуска. Планировщик содержит в себе список задач для запуска и ExecutorService для асинхронного запуска задач из списка. Для старта работы планировщика необходимо вызвать метод start(), после чего ExecutorService внутри планировщика запустит цикл в отдельном потоке, в котором будет проходить по списку задач и пытаться их запустить с помощью метода executeIfNeed(ScheduleTask). Для остановки планировщика необходимо вызвать метод stop(), который остановит ExecutorService и запуск задач по расписанию соответственно. Метод executeIfNeed делает асинхронный вызов метода prepareAndExecute(ExecutorService) у задачи из списка. Логика по проверке необходимости запуска задачи и самого запуска инкапсулирована в методе prepareAndExecute и зависит от класса задачи по расписанию. Класс ScheduleTask запускает исполнение новой задачи вне зависимости от того заршилось предыдущее исполнение этой задачи или нет. Класс ScheduleTaskWithSkipping пропускает запуск нового исполнения, если старое ещё не завершено. Класс ScheduleTaskWithQueue откладывает исполнение задачи в очередь для последующего исполнения, если старое исполнение не закончилось. За проверку необходимости запуска нового исполнения задачи отвечает класс CronHolder, экземпляр которого содержится в объекте ScheduleTask и его наследниках. У CronHolder можно получить время следующего исполнения и обновить это время, то есть установить следующее время исполнения согласно cron выражению.

Класс SchedulerUtils является утилитным классом и необходим для запуска всех зарегистрированных задач с помощью аннотации @Scheduled при старте приложения. Интерфейс Schedulable является маркировочным интерфейсом с одним методом initScheduleTask, который возвращает экземпляр задачи для запуска по расписанию. При вызове метода registerSchedulablesAndStart класса SchedulerUtils сканируется все пакеты приложения и в них находятся все имплементации интерфейса Schedulable с помощью рефлексии, после чего у данных имплементаций вызывается метод initScheduleTask и полученные задачи добавляются в планировщик по умолчанию, который является статическим полем в классе Scheduler. После добавления задач в планировщик по умолчанию, он запускается и задачи начинают исполняться согласно расписанию. Аннотация @Scheduled содержит следующие параметры для настройки запуска задачи по расписанию:

* SchedulingIntersectionStrategy strategy определяет поведение нового исполнения задачи при незаконченном прошлом исполнении;
* long interval определяет интервал между запусками задач в миллисекундах при отсутствии cron выражения в параметре start;
* String start является строкой cron выражением для расписания запуска задачи или значением по умолчанию “DEFAULT”. При значении по умолчанию задача начинает исполняться сразу после запуска приложения с интервалом, заданным в параметре interval;
* String id является идентификатором задачи в формате UUID и необходим для логирования и отслеживания ошибок в конкретных задачах;
* int queueSize используется при указании стратегии PUT\_TO\_QUEUE и задаёт максимальный размер очереди для задач откладываемых в неё.

На рисунке 7 представлена диаграмма последовательности для работы планировщика.

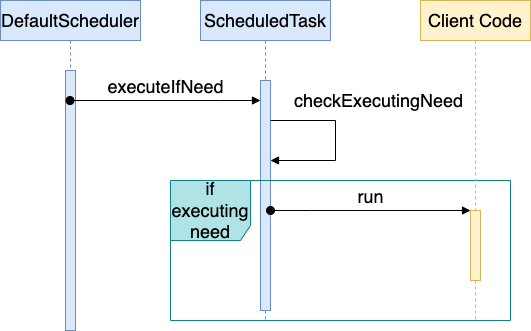


Рисунок 7 – Диаграмма последовательности для запуска задач по расписанию

На данной диаграмме можно увидеть, что запуск задачи планировщиком происходит асинхронно с помощью метода executeIfNeed без ожидания возвращаемого значения. Также и сама задача запускает клиентский код без ожидания возвращаемого значения.

Важной особенностью является ограничение на методы, которые можно запускать по расписанию с помощью аннотации @Scheduled. Данные методы должны иметь модификатор доступа public и модификатор static, так как для вызова нестатичного метода необходим экземпляр класса, а класс может иметь какое-то состояние и средствами программной библиотеки и кодогенерации его невозможно создать. Так как разрабатываемая библиотека для запуска задач по расписанию не интегрирована ни с каким фреймворком для инверсии зависимостей и управлением жизненным циклом приложения, нельзя получить состояние экземпляра класса с методом для задачи по расписанию в рантайме. По этой же причине у метода должны отсутствовать входные параметры.

4.1.2 Lock Striping

Подход Lock Striping или «расщепляющиеся блокировки» помогает увеличить пропускную способность при синхронизации доступа к разделяемым ресурсам. Суть подхода в том, чтобы блокироваться не на каждом вызове и дальнейшем обращении к ресурсам, требующим синхронизации, а присвоить каждому ресурсу уникальный идентификатор и производить блокировку по данному идентификатору, что сохранит целостность и консистентность данных и уменьшит общее время, проведённое в блокировке. На рисунке 8 представлена диаграмма классов Lock Striping.

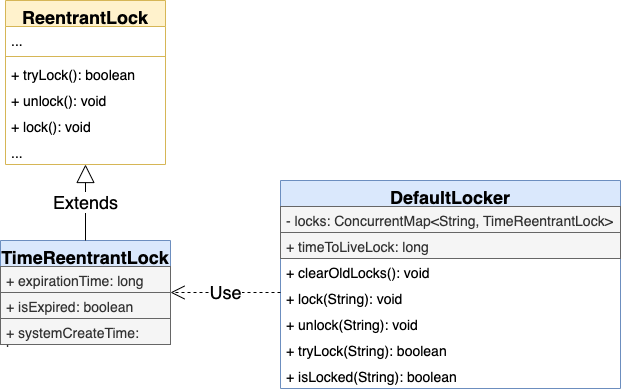


Рисунок 8 – Диаграмма классов Lock Striping

Основные классы DefaultLocker и TimeReentrantLock. TimeReentrantLock унаследован от класса ReentrantLock[10] из стандартного пакета Java механизмов синхронизации. Его отличие в том, что он также хранит в себе время создания блокировки и время её жизни, чтобы можно было удалять объекты блокировок из памяти по истечение какого-то времени. Класс TimeReentrantLock имеет метод isExpired, который проверяет просрочилась ли блокировка и нужно ли её удалять.

Класс DefaultLocker хранит в себе список блокировок в потокобезопасной структуре ConcurrentHashMap[11], которая предоставляет интерфейс доступа к объектам через ключ значение. Также в блокировщике содержится поле timeToLiveLock, которое обозначает время жизни добавляемых блокировок в миллисекундах. DefaultLocker похож на интерфейс класса ReentrantLock и имеет методы блокировки по ключу, попытки блокировки, разблокировки, проверки заблокирован ли ключ и очистки истекших блокировок. Метод isLocked проверяет блокировку по ключу, если ключ заблокирован возвращается true, если ключ не заблокирован или ключ отсутствует в таблице возвращается fakse. Метод tryLock пытается произвести блокировку по ключу, если это получается, происходит блокировка и возвращается значение true, в противном случае возвращается false и поток не переходит в режим ожидания. Метод clearOldLocks удаляет из таблицы блокировки, время жизни которых больше, чем значение поля timeToLiveLock класса DefaultLocker. На рисунке 9 представлена диаграмма последовательности Lock Striping.

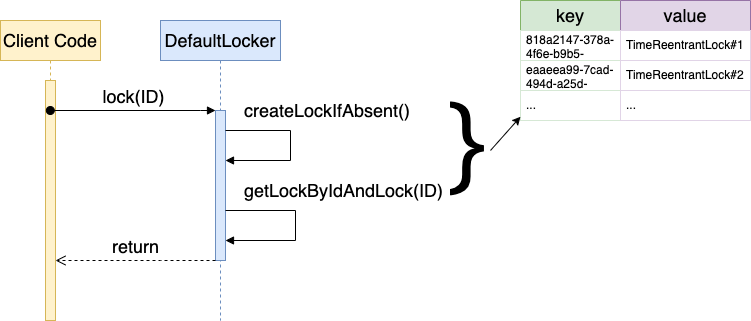


Рисунок 9 – Диаграмма последовательности блокировки Lock Striping

Клиентский код вызывает метод lock у экземпляра класса DefaultLocker, который реализует шаблон проектирования «одиночка» и имеет статическое поле INSTANCE для доступа к методам. В качестве параметра в метод lock передаётся идентификатор, по которому нужно произвести блокировку. DefaultLocker проверяет наличие блокировки с таким идентификатором и создаёт блокировку, при её отсутствии, после чего на полученном объекте ReentrantLock производится блокировка. На рисунке 10 показана диаграмма последовательности для разблокировки Lock Striping.

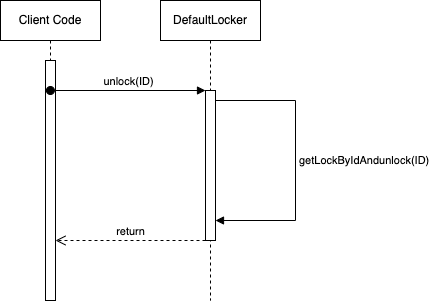


Рисунок 10 – Диаграмма последовательности разблокировки Lock Striping

Клиентский код вызывает метод unlock у DefaultLocker и передаёт в качестве параметра идентификатор блокировки, после чего DefaultLocker производит разблокировку ReentrantLock, если находит его по идентификатору, если же по переданному идентификатору отсутствует объект ReentrantLock, вызов игнорируется.

4.1.3 Межсервисный синхронизатор

Синхронизация между несколькими сервисами важна, так как общие ресурсы, с которыми они взаимодействуют могут не иметь внутри механизмов синхронизации. В качестве синхронизатора для реализации был выбран семафор. Семафор является синхронизацией со счётчиком, позволяющим получить одновременно доступ к общему ресурсу заранее заданному числу потребителей данного ресурса. В качестве внешнего хранилища для реализации была выбрана база данных, хранящаяся в памяти Redis, так как она хорошо подходит для хранения короткоживущих данных. На рисунке 11 представлена диаграмма классов для межсервисного семафора, хранящего состояние в Redis.

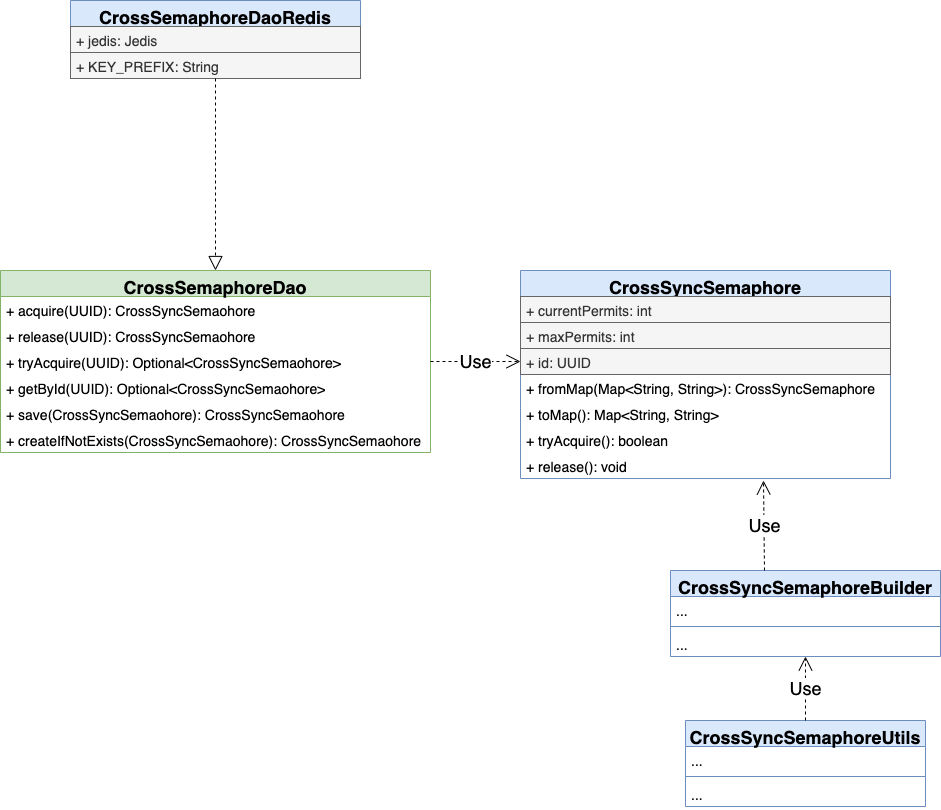


Рисунок 11 – Диаграмма классов межсервисного семафора в Redis

Основным интерфейсом для взаимодействия с семафором является CrossSemaphoreDao, он предоставляет API для взаимодействия с внешним источником, хранящим состояния семафоров. В разработанной программной библиотеке имплементацией данного интерфейса является класс CrossSemaphoreDaoRedis, который содержит в себе логику взаимодействия с внешней базой данных Redis. Так как CrossSemaphoreDao – интерфейс, могут быть и другие его реализации, хранящие состояния семафоров, например в MongoDB или реляционных базах данных. CrossSemaphoreDao предоставляет следующие методы для взаимодействия с семафорами:

* acquire(UUID) принимающий на вход идентификатор семафора, захватывающий семафор и уменьшающий счётчик на единицу. Если счётчик равен нулю, то поток пытающийся захватить семафор блокируется. Возвращает захваченный семафор;
* release(UUID) принимающий на вход идентификатор семафора, освобождающий семафор и увеличивающий счётчик на единицу. Возвращает освобожденный семафор;
* tryAcquire(UUID) принимающий на вход идентификатор семафора, и пытающийся захватить семафор. В отличие от метода acquire(UUID), если счётчик семафора равен нулю, поток пытавшийся захватить семафор не блокируется, а продолжает выполнение. Метод возвращает Optional<CrossSyncSemaphore>, при удачной попытке захвата семафора вернётся сущность семафора, если счётчик был равен нулю, то вернётсся пустой Optional;
* getById(UUID) возвращает Optional<CrossSyncSemaphore>, который пустой при отсутствии семафора с таким идентификатором во внешнем источнике или содержит семафор полученный по идентификатору из внешнего источника;
* save(CrossSyncSemaphore) сохраняет сущность семафора во внешний источник и возвращает её же;
* createIfNotExists(CrossSyncSemaphore) сохраняет сущность семафора во внешний источник при её отсутствии там, если сущность с таким идентификатором уже существует, она же и возвращается.

Класс CrossSyncSemaphore предстваляет собой сущность семафора, которой пользователь оперирует, взаимодействуя с внешним источником. Он содержит в себе идентификатор в формате UUID, максимальное значение счётчика и текущее значение счётчика. Также в CrossSyncSemaphore присутствуют методы трансформации в Map<String, String> и получения объекта CrossSyncSemaphore из Map<String, String>. Класс CorssSyncSemaphoreUtils содержит вспомогательные методы для создания семафоров с помощью фабричных методов и создания CrossSemaphoreDaoRedis с настройкой Redis клиента. В качестве Redis клиента был выбран Jedis[12], так как он имеет удобный и понятный API и прост в использовании. Класс CrossSyncSemaphoreBuilder используется для более удобной и гибкой настройки Redis клиента. На рисунке 12 представлен формат хранения состояния семафоров в Redis.

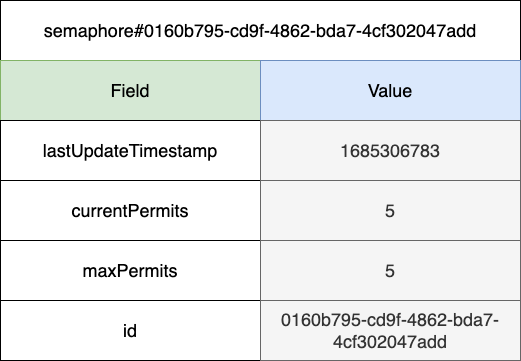


Рисунок 12 – Формат хранения семафора в Redis

На данном рисунке можно заметить, что для хранения всех семафоров используется специальный префикс, который по умолчанию «semaphore#», но его можно изменить. Также присутствует поле lastUpdateTimestamp, в котором хранится время последнего изменения состояния семафора в формате timestamp. При настройке клиента для взаимодействия с Redis также можно указать параметр TTL, который определяет время жизни записей в базе в секундах и по умолчанию отключен. На рисунке 13 представлена диаграмма последовательности для межсервисного семафора в Redis.

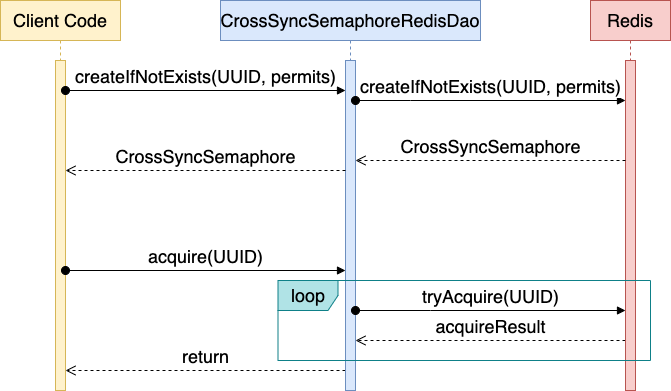


Рисунок 13 – Диаграмма последовательности межсервисного семафора в Redis

На данной диаграмме видно, что перед каждой попыткой захватить семафор сначала проверяется его наличие в базе и при отсутствии семафора, он создаётся, чтобы избежать ошибок. Сам захват семафора работает по принципу оптимистичной блокировки[13], смотрится время последнего изменения записи и если оно отличается от времени, полученного предыдущим запросом, то захват не происходит и CrossSyncSemaphoreRedisDao заново пытается захватить семафор и уменьшить счётчик. Если при получении семафора счётчик уже равен нулю, то семафор будет доставаться из базы пока счётчик не окажется больше 0. Освобождение семафора происходит по такому же принципу с использованием подхода оптимистичной блокировки.

4.2 Библиотека кодогенерации шаблонов многопоточного кода

Кодогенерация происходит с помощью процессоров аннотаций и библиотеки JavaPoet. На рисунке 14 представлена диаграмма классов процессоров аннотаций.

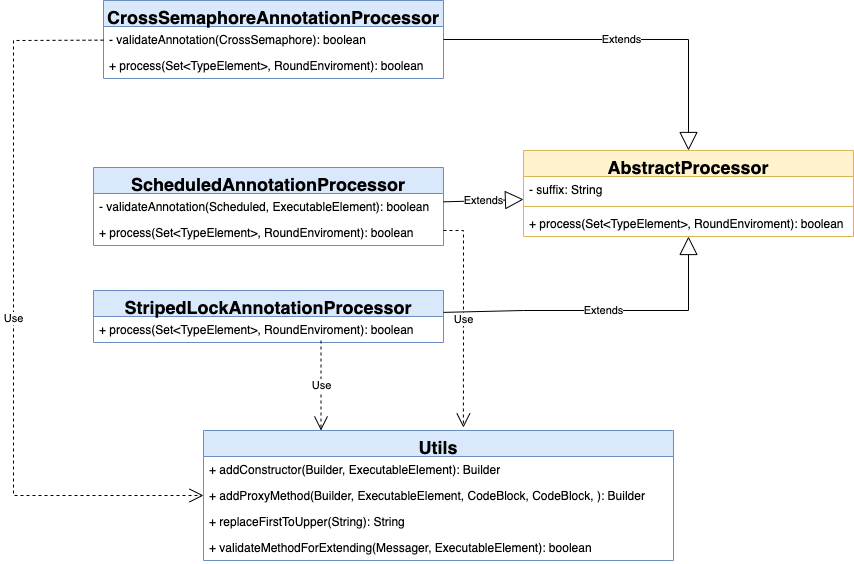


Рисунок 14 – Диаграмма классов процессоров аннотаций

Все процессоры аннотаций наследуются от абстрактного класса AbstractProcessor. Каждый класс наследник отвечает за обработку отдельной аннотации. CrossSemaphoreAnnotation processor обрабатывает элементы помеченные аннотацией @CrossSemaphore. Метод process получает на вход множество всех элементов, помеченных данной аннотацией, сначала нужно провалидировать все эти элементы на корректность параметров аннотации. Данная проверка происходит в методе validateAnnotation (CrossSemaphore), проверяется идентификатор будущего семафора на соответствие с форматом UUID и параметр permits. Параметр permits – целое число, которое должно быть больше 0. Если хотя бы одна аннотация не проходит проверку, в лог компиляции выводится сообщение об ошибке с именем файла, описанием и местом, где допущена ошибка с помощью класса Messager. После проверки, все аннотации-элементы группируются по элементу – их родительскому классу, это нужно для того, чтобы сгенерировать одного наследника на каждый класс, котором присутствует больше одного метода, помеченного аннотаций @CrossSemaphore. После группировки для каждой группы создаётся описание будущего типа с помощью класса TypeSpec библиотеки JavaPoet.

ScheduledAnnotationProcessor обрабатывает элементы помеченные аннотацией @Scheduled. Метод process получает на вход множество всех элементов, помеченных данной аннотацией, и для каждого элемента совершает проверку на корректность параметров с помощью метода validateAnnotation и генерирует новый класс с кодом регистрации задачи по расписанию. Сгенерированный класс является имплементацией интерфейса Schedulable с одним методом, возвращающим сконструированную задачу для запуска по расписанию.

StripedLockAnnotationProcessor обрабатывает элементы аннотированные @StripedLock. Метод process получает на вход множество всех элементов, помеченных данной аннотацией, после чего группирует их по элементу – их родительскому классу. После группировки для каждой группы генерируется класс наследник с переопределёнными методами, содержащими логику использования Lock Striping. Аннотация @StripedLock содержит один параметр lockIdentifier, в котором указывается блок кода, который будет получать идентификатор для блокируемого ресурса. Данный параметр не требует валидации, так как он будет проверен при компиляции сгенерированного кода.

Класс Utils содержит вспомогательные методы для генерации кода. Метод addConstructor принимает на вход объект Builder, который отвечает за построение конкретного класса и объект ExecutableElement, который представляет собой конструктор родительского класса, который необходимо переопределить, добавляет в новый класс переопределённый конструктор и возвращает обновлённый объект Builder с информацией о переопределённом конструкторе. Метод addProxyMethod принимает на вход объект Builder, который отвечает за построение конкретного класса, и объект ExecutableElement, представляющий собой метод в родительском классе, который нужно запроскировать и два объекта CodeBlock, которые представляют собой куски исполняемого кода, которым нужно обернуть бизнес-логику родительского метода. Метод addProxyMethod добавляет в объект Builder, описывающий класс наследник, переопределённый метод элемента ExecutableElement с добавленной логикой, после чего возвращает модифицированный объект Builder. Метод replaceFirstToUpper принимает на вход строку, меняет первый её символ на заглавный и возвращает обновлённую строку. Метод validateMethodForExtending принимает на вход объект Messager и объект ExecutableElement представляющий проверяемый метод, валидирует метод на возможность быть переопределённым (проверяет, чтобы метод не имел модификаторов static и final и был объявлен с модификатором доступа public или protected) и возвращает true в случае успеха или выводит сообщение об ошибке в лог компиляции и возвращает false.

4.3 Отладка библиотеки кодогенерации

Так как разработанная библиотека кодогенерации работает во время компиляции нет возможности воспользоваться стандартными способами отладки Java приложений, потому что на данном этапе Java приложение ещё не запущено и отлаживать нечего. Библиотека кодогенерации работает вместе компилятором javac и как следствие нужно подключиться к javac в режиме отладки. Для этого существует специальные параметры запуска компилятора. С помощью ключа Xdebug включить режим отладки компилятора и с помощью ключа Xrunjdwp:transport=dt\_socket,server=y,suspend=y,address=8000 включить возможность удалённой отладки для javac по протоколу jdwp на порте 8000.

5 Результат работы программы

5.1 Результат кодогенерации

Результатом работы программной библиотеки являются сгенерированные исходные файлы Java, на основании аннотаций в пользовательском коде. На листинге 7 представлен исходный код, на основе которого сгенерирован пример для запуска задачи по расписанию.

Листинг 7 – Исходный код для запуска задач по расписанию

public class ExampleScheduling {  
  
 @Scheduled  
 public static void scheduledMethod1() {  
 System.out.println("1");  
 }  
   
 @Scheduled(strategy = SchedulingIntersectionStrategy.PUT\_TO\_QUEUE, id = “4f2b6fe2-742b-4f75-b736-ce8d271955f4” start = "0 \* \* \* \*", queueSize = 100)  
 public static void scheduledMethod2() {  
 System.out.println("2");  
 }  
}

На данном классе два метода помечено аннотацией @Scheduled, одна аннотация без параметров, вторая с параметрами. На листинге 8 представлен результат кодогенерации для запуска по расписанию метода scheduledMethod1.

Листинг 8 – Результат кодогенерации для запуска по расписанию метода scheduledMethod1

public class ExampleSchedulingScheduledMethod1ScheduledInit implements Schedulable {

public ScheduleTask initScheduleTask() {  
 return ScheduleTask.of(RUN\_IMMEDIATELY, 1000, "DEFAULT", () -> ExampleScheduling.scheduledMethod1(), "11111111-1111-1111-1111-111111111111", 100);  
 }  
}

На данном листинге сгенерирован класс реализация интерфейса Schedulable с имплементацией метода initScheduleTask. Название сгенерированного класса является конкатенацией имени класса, котором был метод, помеченный аннотацией @Scheduled, для которого сгенерирован код, создающий задачу по расписанию, имени самого метода для запуска по расписанию и суффикса ScheduledInit. Метод initScheduleTask возвращает сконструированную задачу для запуска по расписанию, так как аннотация, помечающая метод scheduledMethod1 была без параметров, задача сконструирована с параметрами по умолчанию:

* стратегия запуска RUN\_IMMEDIATELY;
* интервал между запусками задач 1000 миллисекунд;
* параметр DEFAULT указывает, что запуск производится вместе со стартом планировщика;
* лямбда-выражение с запуском метода scheduledMethod1;
* идентификатор задачи в формате UUID.

На листинге 9 представлен результат кодогенерации для метода scheduledMethod2.

Листинг 9 – Результат кодогенерации для запуска по расписанию метода scheduledMethod2

public class ExampleSchedulingScheduledMethod2ScheduledInit implements Schedulable {

public ScheduleTask initScheduleTask() {  
 return ScheduleTask.of(PUT\_TO\_QUEUE, 1000, "0 \* \* \* \*", () -> ExampleScheduling.scheduledMethod2(), "4f2b6fe2-742b-4f75-b736-ce8d271955f4”, 100);  
 }  
}

На данном листинге параметры для конструирования задачи для запуска по расписанию были взяты из аннотации. Так как третий параметр задан и является cron выражением, то задача будет запускаться согласно расписанию данного cron выражения и второй параметр указывающий интервал между запусками будет игнорироваться. Параметр queueSize задаёт размер очереди для откладывания задач, не успевших начать исполнение вовремя.

На листинге 10 представлен исходный код, на основе которого сгенерирован пример для Lock Striping.

Листинг 10 – Исходный код для Lock Striping

public class TestSL {  
  
 public TestSL() {  
 }  
  
 public TestSL(int i, String aaa) {  
 //initialization  
 }  
  
 @StripedLock(lockIdentifier = "arg.toUpperCase()")  
 public void test1(String arg) {  
 //business logic  
 }  
  
 @StripedLock(lockIdentifier = "param")  
 public String test2(String param) {  
 //business logic  
 return "s";  
 }  
}

На данном листинге представлен класс с двумя методами, помеченными аннотацией @StripedLock. На листинге 11 представлен результат кодогенерации для вышеупомянутого класса.

Листинг 11 – Результат кодогенерации для Lock Striping

public class TestSLStripedLockProxy extends TestSL {  
 public TestSLStripedLockProxy() {  
 super();  
 }  
  
 public TestSLStripedLockProxy(int i, String aaa) {  
 super(i, aaa);  
 }  
  
 @Override  
 public void test1(String arg) {  
 DefaultLocker.getInstance().lock(arg.toUpperCase());  
 super.test1(arg);  
 DefaultLocker.getInstance().unlock(arg.toUpperCase());  
 }  
  
 @Override  
 public String test2(String param) {  
 DefaultLocker.getInstance().lock(param);  
 var returnValueGenerated = super.test2(param);  
 DefaultLocker.getInstance().unlock(param);  
 return returnValueGenerated;  
 }  
}

На данном листинге представлен результат кодогенерации для класса с методами, помеченными аннотацией @StripedLock. Сгенерированный класс наследуется от класса, в котором были методы, помеченные аннотацией @StripedLock и имеет имя, составленное из имени родительского класса и суффикса StripedLockProxy. В классе потомке унаследованы конструкторы со всеми аргументами конструкторов в классе родителе и переопределены методы, помеченные аннотацией @StripedLock в родительском классе. В тело переопределённых методов добавлена блокировка по идентификатору до вызова тела метода-родителя и освобождение блокировки после вызова метода родителя. Также учитывается возвращаемый тип метода, если он void, то между блокировкой и разблокировкой происходит просто вызов родительского метода, если метод возвращает какой-либо тип, то в переопределённом методе результат записывается в локальную переменную returnValueGenerated и возвращается после освобождения блокировки. Параметр lockIdentifier из аннотации используется как кусок исполняемого кода для получения идентификатора блокировки, в методе test1 это выражение arg.toUpperCase(). Также учитывается, что в одном классе может быть несколько методов аннотированных @StripedLock.

На листинге 12 представлен исходный код для генерации межсервисного семафора в Redis.

Листинг 12 – Исходный код для межсервисного семафора в Redis

public class CsTest {  
  
 public CsTest() {}  
 public CsTest(int i, String str) {}  
  
 @CrossSemaphore(semophoreId = "4f2b6fe2-742b-4f75-b736-ce8d271955f4", permits = 5)  
 public void test() {  
 System.out.println();  
 }  
}

Метод test помечен аннотацией @CrossSemaphore с двумя параметрами:

* параметр semaphoreId представляет идентификатор в формате UUID для хранения семафора в Redis;
* параметр permits представляет собой начальное значение счётчика семафора.

Результат кодогенерации для межсервисного семафора представлен на листинге 13.

Листинг 13 – Результат кодогенерации для межсервисного семафора в Redis

public class CsTestCrossSemaphoreProxy extends CsTest {  
 public CsTestCrossSemaphoreProxy() {  
 super();  
 }  
  
 public CsTestCrossSemaphoreProxy(int i, String str) {  
 super(i, str);  
 }  
  
 @Override  
 public void test() {  
 CrossSyncSemaphoreUtils.getDefaultDaoInstance().createIfNotExists(CrossSyncSemaphore.newInstance(UUID.fromString("4f2b6fe2-742b-4f75-b736-ce8d271955f4"), 5, 5));CrossSyncSemaphoreUtils.getDefaultDaoInstance().acquire(UUID.fromString("4f2b6fe2-742b-4f75-b736-ce8d271955f4"));  
 super.test();  
 CrossSyncSemaphoreUtils.getDefaultDaoInstance().release(UUID.fromString("4f2b6fe2-742b-4f75-b736-ce8d271955f4"));  
 }  
}

Класс CsTestCrossSemaphoreProxy унаследован от класса CsTest и имеет имя составленное из имени родительского класса и суффикса CrossSemaphoreProxy. Конструкторы наследуются со всеми параметрами, методы, помеченные аннотацией @CrossSemaphore переопределяются с добавлением логики захвата семафора до вызова родительского метода и освобождением семафора после родительского метода. Также как и для Lock Striping кодгенерация учитывает типы возвращаемых значений переопределяемых методов и возможность наличия нескольких методов, аннотированных @CrossSemaphore в одном классе.

5.2 Вычислительный эксперимент

Было произведено тестирование генерируемого кода на производительность средствами стандартной библиотеки Java[14]. Был проведён вычислительный эксперимент для выяснения накладных расходов использования подхода Lock Striping и межсервисного семафора[15] в Redis. На рисунке 10 представлен график зависимости времени работы метода от количества входящих запросов для случая без использования Lock Striping и с использованием Lock Striping по одному идентификатору.



Рисунок 15 – График сравнения времени работы с использованием Striping Lock и без использования Striping Lock

По данному графику видно, что время работы увеличивается линейно для обоих случаев и Lock Striping практически не добавляет времени работы и накладные расходы на его использования составляют около 2,45% или же 1мс на вызов метода, проксированного Lock Striping.

На рисунке 11 представлен график зависимости времени работы метода от количества входящих запросов для случая без использования межсервисного семафора в Redis и с использованием межсервисного семафора в Redis.

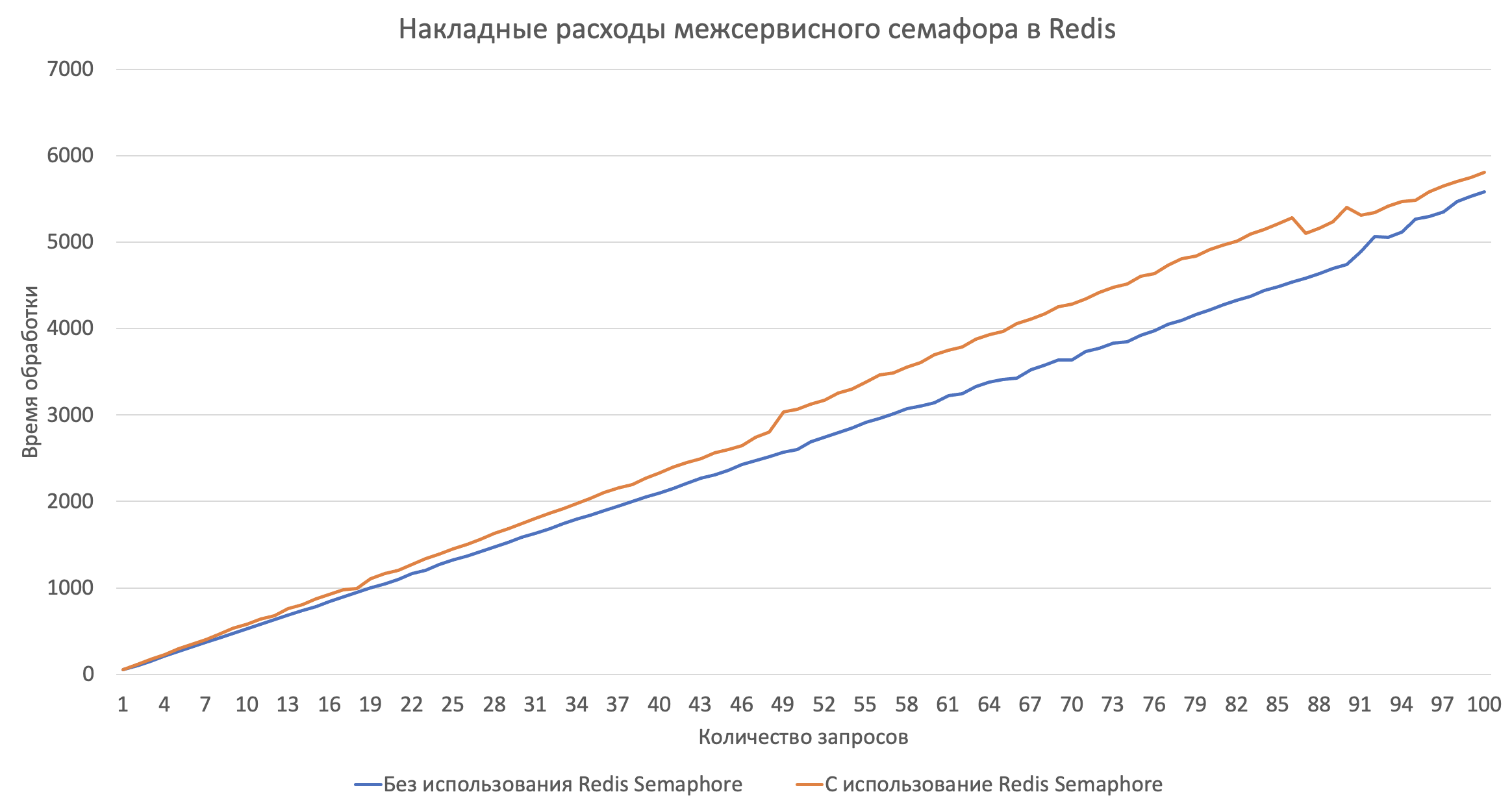


Рисунок 16 - График сравнения времени работы с использованием межсервисного семафора в Redis и без использования межсервисного семафора в Redis

По данному графику можно заметить, что время работы увеличивает линейно при увеличении числа запросов при использовании семафора и без его использования. Накладные расходы на использование межсервисного семафора в Redis составляют примерно 12% или же 6мс на вызов метода, проксированного межсервисным семафором. Это связано с тем, что семафор хранит своё состояние вне приложения и появляются дополнительные затраты времени на передачу данных по сети.

ЗАКЛЮЧЕНИЕ

Результатом выполнения работы является разработанная программная библиотека, позволяющая автоматически генерировать код для многопоточной работы для Java приложений.

В ходе реализации были выполнены следующие задачи:

* проанализирована предметная область;
* в соответствии с особенностями предметной области была спроектирована и реализована программная библиотека, реализующая кодогенерацию для следующих подходов:
  1. запуск задач по расписанию;
  2. Lock Striping;
  3. межсервисный синхронизатор.
* произведено тестирование программы и замер производительности генерируемого кода.

Результатом работы является готовая и протестированная система, которая может использоваться для оптимизации и ускорения работы многопоточных приложений на Java. Выполненная работа может быть использована в различных областях применения, где требуется работа с многопоточностью, где важны быстродействие и производительность.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Java Annotation Processing and Creating a Builder . – Текст : электронный // Baeldung : [сайт]. – 2023. – URL : <https://www.baeldung.com/java-annotation-processing-builder> (дата обращения 10.11.2022)
2. Introduction to Lock Striping. – Текст : электронный // Baeldung : [сайт]. – 2023. – URL : <https://www.baeldung.com/java-lock-stripping> (дата обращения 25.11.2022)
3. Introduction to Quartz. – Текст : электронный // Baeldung : [сайт]. – URL : <https://www.baeldung.com/quartz> (дата обращения 5.12.2022)
4. Annotation Processing 101. – Текст : электронный // HannesDorfmann : [сайт]. – 2023. – URL : <https://hannesdorfmann.com/annotation-processing/annotationprocessing101> (дата обращения 20.12.2022)
5. Многопоточность в Java. Работа с потоками. – Текст : электронный // Хабр [сайт]. – 2023. – URL : <https://habr.com/ru/sandbox/167189/> (дата обращения 10.01.2023)
6. Introduction to Thread Pools in Java. – Текст : электронный // Baeldung : [сайт]. – 2023. – URL : <https://www.baeldung.com/thread-pool-java-and-guava> (дата обращения 25.01.2023)
7. Справочник по синхронизаторам java.uti.concurrent.\* . – Текст : электронный // Хабр : [сайт]. – 2023. – URL : <https://habr.com/ru/articles/277669> (дата обращения 20.02.2023)
8. Redis Docs. – Текст : электронный // Redis : [сайт]. – 2023. – URL : <https://redis.io/docs/about/> (дата обращения 26.02.2023)
9. Introduction to JavaPoet. – Текст : электронный // Baeldung : [сайт]. – 2023. – URL : <https://www.baeldung.com/java-poet> (дата обращения 06.03.2023)
10. Брайн Гоец. Java Concurrency in Practice 1st Edition / Брайн Гоец – Санкт-Петербург, 2020. – 371 с. – Текст : непосредственный.
11. ConcurrentMap (Java Platform SE) . – Текст : электронный // Oracle : [сайт]. – 2023. – URL : <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.htmll> (дата обращения 16.03.2023)
12. Intro to Jedis – the Java Redis. – Текст : электронный // Baeldung : [сайт]. – 2023. – URL : <https://www.baeldung.com/jedis-java-redis-client-library> (дата обращения 22.03.2023)
13. Кратко о блокировках в БД. – Текст : электронный // Github : [сайт]. – 2023. – URL : <https://uthark.github.io/2009/04/22/blog-post_22> (дата обращения 02.04.2023)
14. Джошуа Блох. Java Эффективное программирование. 3-е изд. / Джошуа Блох – Санкт-Петербург : Диалектика-Вильямс, 2019. – 464 с. – Текст : непосредственный.
15. Скотт Оакс. Java Threads (Java Series) 3rd Edition / Скотт Оакс – Санкт-Петербург, 2004. – 350 c. – Текст : непосредственный.

## 