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 25](#__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 Расширение для PostgreSQL 30](#__RefHeading___Toc1729085_1399105856)

[4.1.1 Процесс загрузки расширения в сервер СУБД 30](#__RefHeading___Toc2424_3594881201)

[4.1.2 Низкоуровневая составляющая расширения 31](#__RefHeading___Toc2426_3594881201)

[4.1.2 Скрипт с процедурами на pl/pgSQL 34](#__RefHeading___Toc2428_3594881201)

[4.2 Парсер запросов к графовой модели данных 35](#__RefHeading___Toc1729093_1399105856)

[4.2.1 Лексический анализатор запроса 35](#__RefHeading___Toc2197_793195812)

[4.2.2 Синтаксический анализатор запроса и построение AST-дерева 37](#__RefHeading___Toc2199_793195812)

[4.3 Отладка библиотеки кодогенерации 40](#__RefHeading___Toc1729095_1399105856)

[5 Результат работы программы 40](#__RefHeading___Toc1729097_1399105856)

[5.1 Результат кодогенерации 40](#__RefHeading___Toc1729099_1399105856)

[5.2 Вычислительный эксперимент 45](#__RefHeading___Toc1729101_1399105856)

[ЗАКЛЮЧЕНИЕ 48](#__RefHeading___Toc1729103_1399105856)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 49](#__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

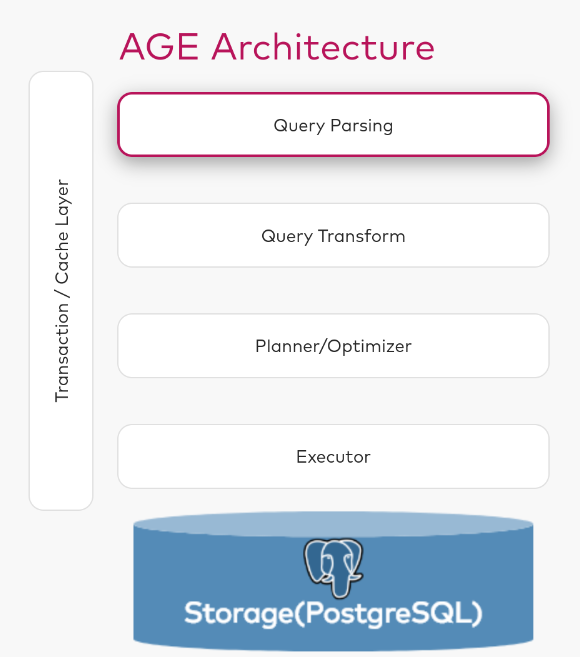
2.1 Анализ существующих решений

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.

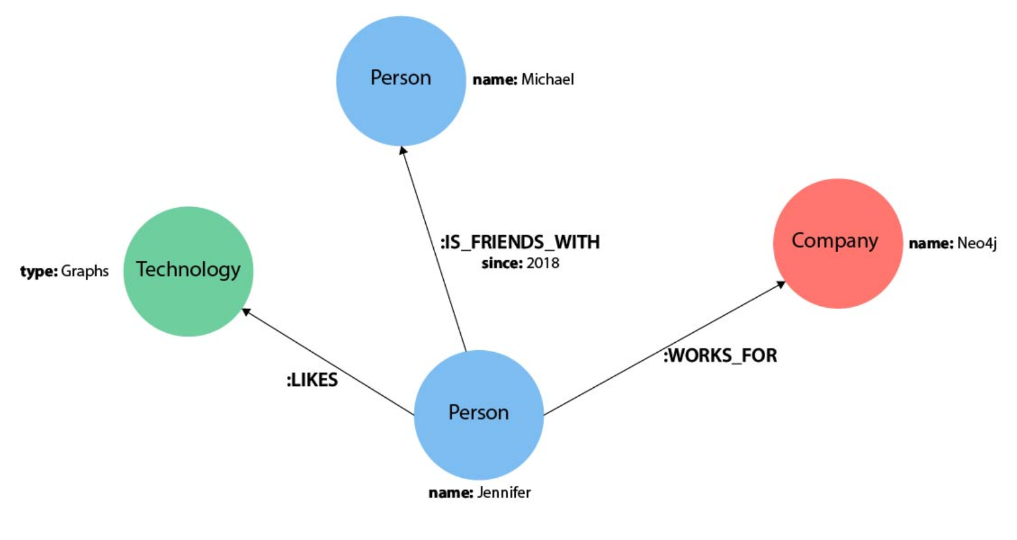


Рисунок 2 – Пример графа свойств

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

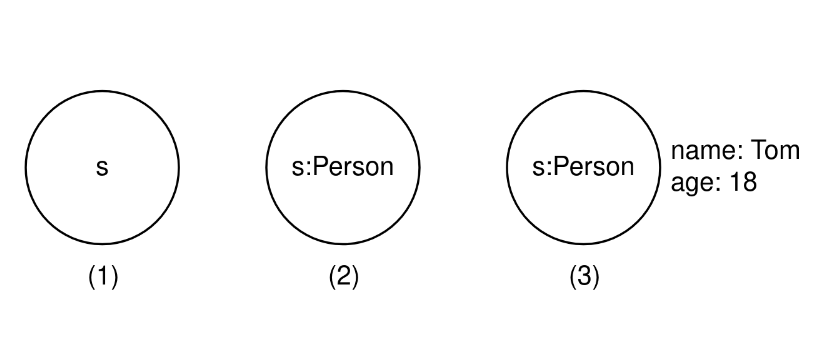


Рисунок 3 – Визуализация узлов

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)

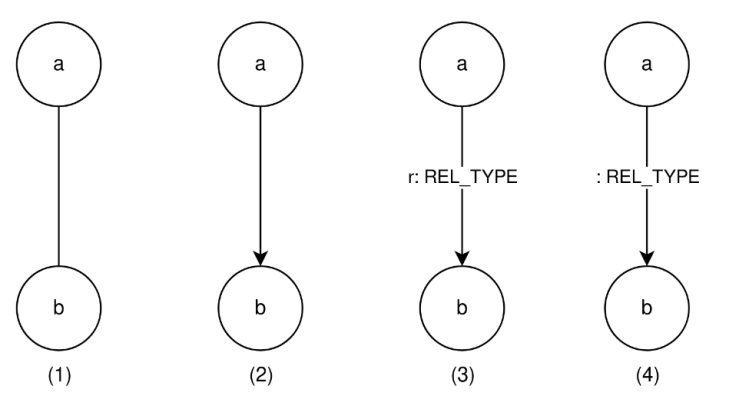


Рисунок 4 - Визуализация отношений в графе

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

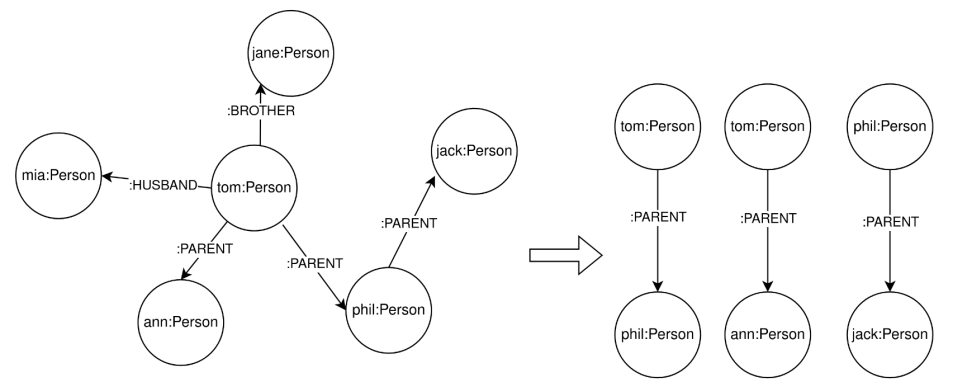


Рисунок 5 - Визуализация выполнения запроса выборки

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)

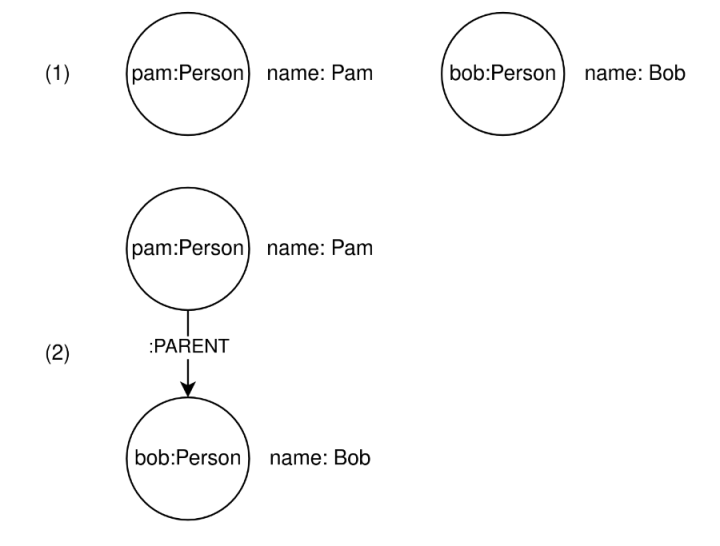


Рисунок 6 - Визуализация процесса создания сущностей в базе данных

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

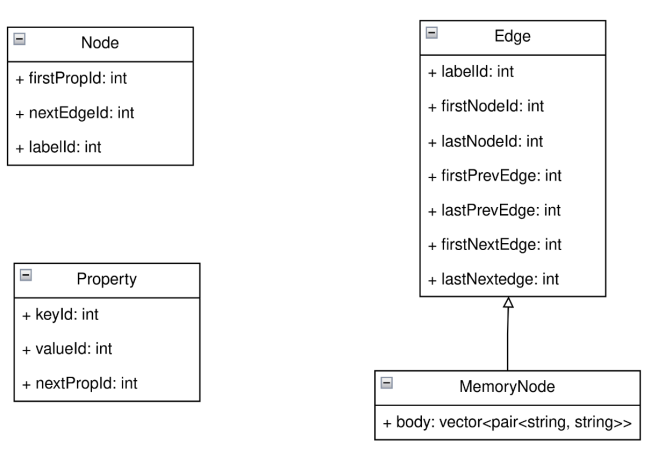
2.4 Структура графового хранилища

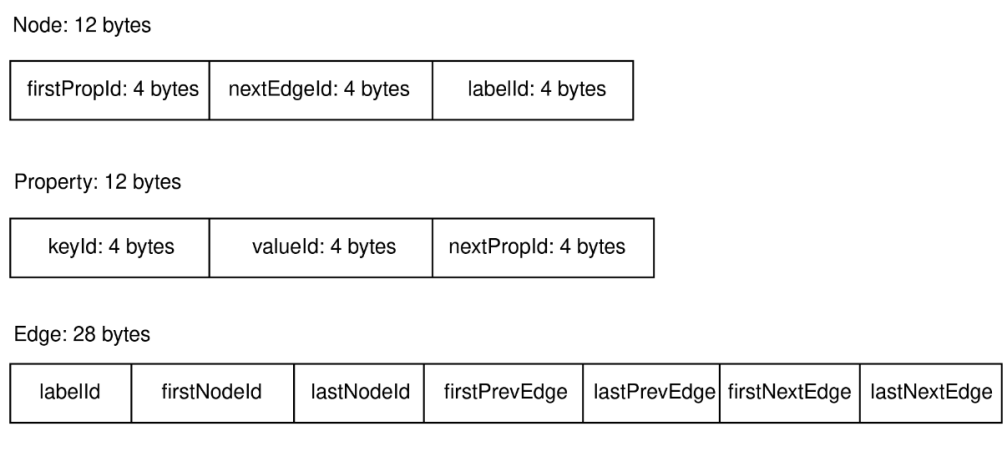
Графовое хранилище состоит из нескольких бинарных файлов, каждый из которых отвечает за отдельный компонент графа свойств:\

* 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

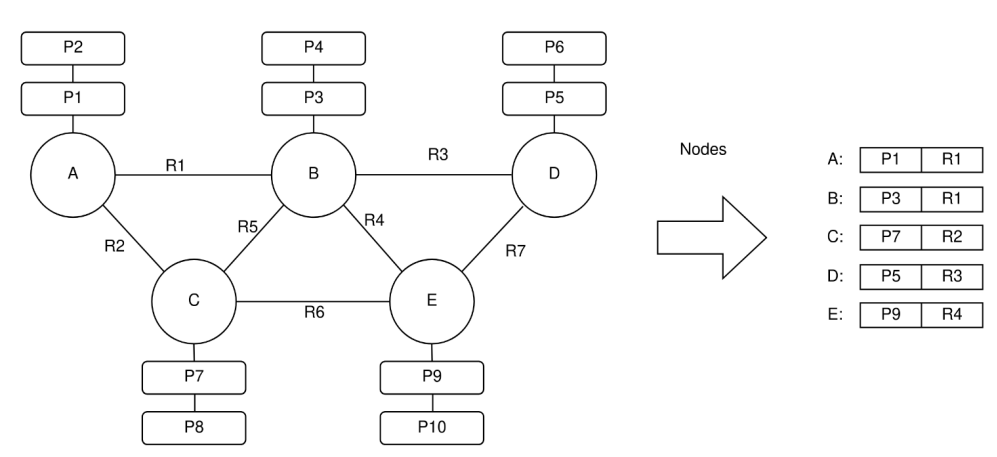
Рисунок 8 — Представление сущностей в файлах

2.6 Алгоритм построения графа

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

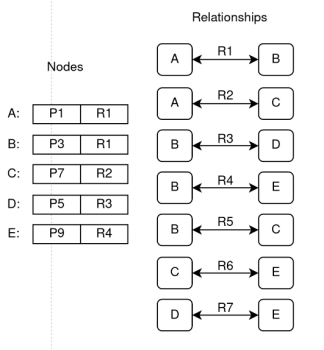
2.6.1 Построение узлов и свойств узлов

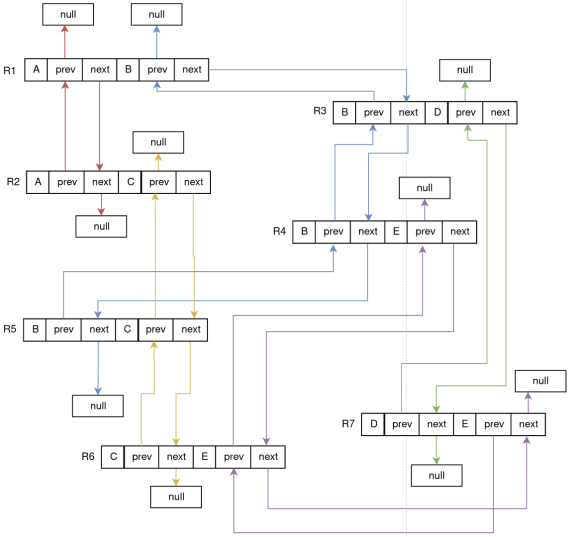
Пусть задан граф как на рисунке 9, и его необходимо перевести в вид сущностей, представленных выше. Сначала для каждого узла определяется первое ребро по счету, в котором узел был задействован: его номер по счету в общем списке ребер заносится в поле nextEdgeId. После этого для каждой пары «ключ-значение» 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 CLion 2023.3.

CLion – кроссплатформенная интегрированная среде разработки программного обеспечения для проектов на C, C++ и других языков программирования от компании JetBrains. Поддерживает компиляторы GCC, Clang, Microsoft Visual C++.

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

В ходе проектирования система была разбита на следующие компоненты:

* расширение для СУБД PostgreSQL;
* лексический и синтаксический парсер запросов на языке Cypher;
* менеджер графовых сущностей, каждый элемент которого отвечает за управление определенными объектами графа или его составляющими в соответствующем файле на диске;
* интерфейс командной строки (CLI), предоставляющий выполнение запросов на Cypher напрямую к серверу графового хранилища.

Диаграмма пакетов системы представлена на рисунке 12.

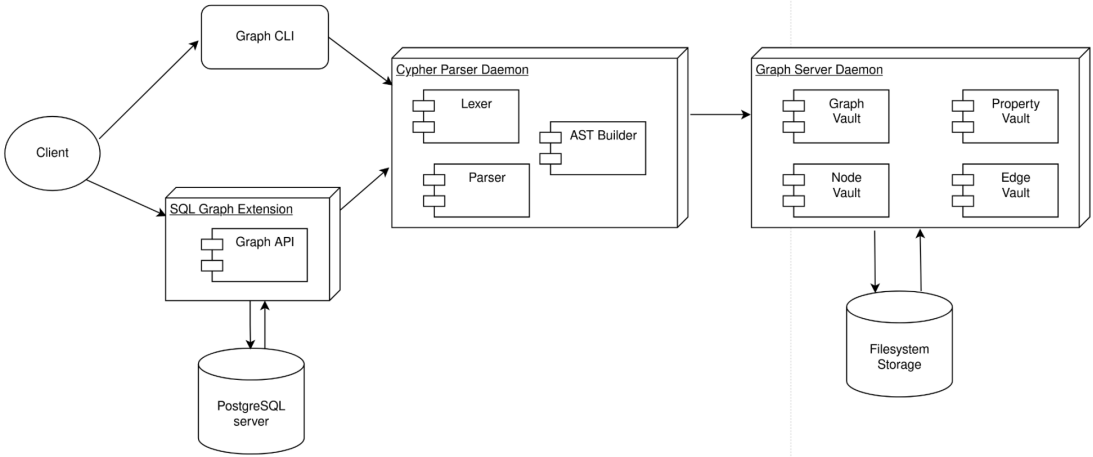


Рисунок 12 – Диаграмма пакетов графового хранилища данных

4.1 Расширение для PostgreSQL

4.1.1 Процесс загрузки расширения в сервер СУБД

Расширение для системы управления базами данных PostgreSQL представляет собой комбинацию SQL-скрипта с хранимыми процедурами, написанными на специальном языке серверного программирования pl/pgSQL, и файла с кодом на языке программирования C[12]. Низкоуровневый код при компиляции связывается с заголовочными файлами сервера PostgreSQL, предоставляемыми библиотекой libpq-devel, и образует динамическую (разделяемую) библиотеку формата .so, которая содержит интерфейс для обращения к ней из SQL-процедур во время обработки запроса базой данных. Также создается специальный файл с названием расширения и форматом .control, в котором указывают необходимые метаданные для корректной загрузки расширения в сервер: описание; текущая версия; местоположение относительно базового каталога с расширениями; флаг, с помощью которого можно задать загрузку расширения каждый раз при запуске. Пример такого файла приведет в листинге 9.

Листинг 9 — Пример файла graph\_db.control

comment = 'Extension to graph-based database'

default\_version = '1.0'

module\_pathname = '$libdir/graph\_db'

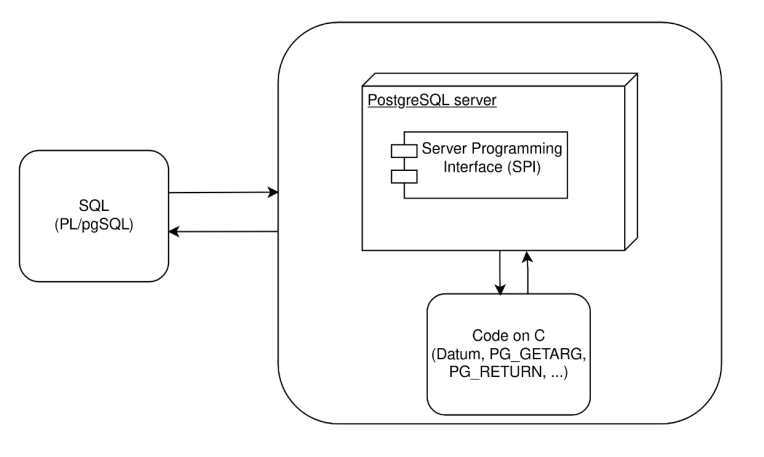
relocatable = false

Для сборки используется поставляемая в дистрибутиве PostgreSQL система сборки PGXS, которая облегчает сборку простых расширений. Для ее использования в CMake необходимо включить модуль add\_postgresql\_extension, который принимает название расширения, исходные коды и файл .control).

После того, как расширение скомпилировано и перемещено в каталог с другими расширениями, его необходимо загрузить. Для этого в клиенте PostgreSQL (в основном, psql), необходимо выполнить команду CREATE EXTENSION <название расширения>. Информацию об расширении можно получить с помощью команды \dx <название раширения>. Для выгрузки расширения из сервера используется команда DROP EXTENSION.

4.1.2 Низкоуровневая составляющая расширения

Расширение содержит в себе файлы с исходным кодом на языке C, в котором можно осуществлять обращение напрямую к серверу с данными СУБД. Такую возможность предоставляет специальный C-интерфейс — SPI (Server Programming Interface), содержащий функции для обращения к таблицам и выборки данных из их. Диаграмма пакетов, отображающая это взаимодействие, приведена на рисунке 13.

Рисунок 13 – Диаграмма пакетов расширения для PostgreSQL

В начале файла обязательно должны включаться заголовочные файлы <postgres.h> и <fmgr.h>, которые предоставляют доступ к специальным макросам. Макрос PG\_MODULE\_MAGIC указывает на то, что текущий файл будет скомпилирован как расширение и связан с SQL-процедурами. С помощью макроса PG\_FUNCTION\_INFO\_V1() задается название функции, которая может вызываться из кода pl/pgSQL. Пример описания функций приведен в листинге 10.

Листинг 10 — Объявление внешних функций

#include <postgres.h>

#include <fmgr.h>

PG\_MODULE\_MAGIC;

PG\_FUNCTION\_INFO\_V1(cypher);

PG\_FUNCTION\_INFO\_V1(key\_value);

PG\_FUNCTION\_INFO\_V1(graph\_nodes);

PG\_FUNCTION\_INFO\_V1(filter\_graph\_nodes);

PG\_FUNCTION\_INFO\_V1(graph\_edges);

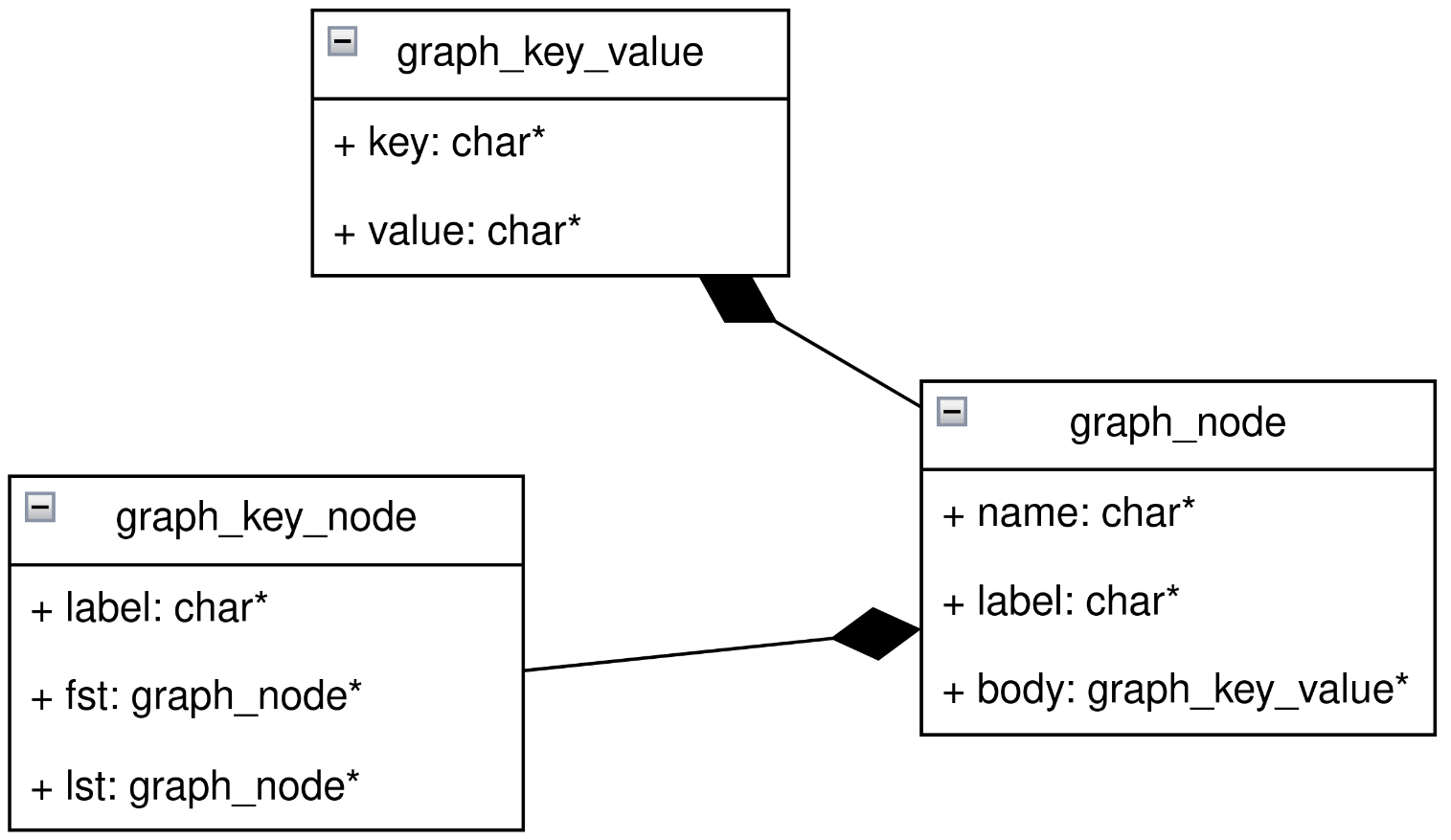
PG\_FUNCTION\_INFO\_V1(filter\_graph\_edges);

PG\_FUNCTION\_INFO\_V1(create\_graph);

Чтобы функция являлась внешней для pl/pgSQL, она должна возвращать специальную структуру Datum, которая представляет собой аналог целочисленного указателя uintptr\_t, то есть возврат осуществляется через указатель для доступа без копирования, так как это может повлечь за собой большие расходы процессорного времени и памяти на копирование больших объектов (например, выборки из таблицы). В качестве единственного аргумента указывается макрос PG\_FUNCTION\_ARGS, который во время препроцессинга развертывается во все типы аналогичной процедуры на pl/pgSQL. Получение данных из аргументов и возврат значения осуществляется через макросы PG\_GETARG\_\* и PG\_RETURN\_\*, где вместо звездочки подставляется необходимый тип данных[13]. В качестве типа данных могут использоваться как стандартные скалярные типы и указатели на собственные структуры, так и типы PostgreSQL.

В расширении для графового хранилища все функции осуществляют преобразование типов данных, основная работа с данными делегируется на менеджер графовых сущностей через внешний C-интерфейс (функции, помеченные как extern).

В самом начале определены вспомогательные структуры graph\_key\_value, graph\_node, graph\_edge, которые представляют типы данных, возвращаемые в качестве ответа от сервера в SQL-процедурах и в которые производится конвертация данных от сервера графового хранилища. Диаграмма классов приведена на рисунке 14.

Рисунок 14 — Диаграмма классов графовых сущностей для SQL

Для структуры узла графа (graph\_node) тело свойств body будет преобразовано в строку формата JSON. Для этого были определены функции kv\_to\_string и body\_to\_string, осуществляющие процесс конвертации в строки на куче, которые должны быть освобождены после использования. Работа с памятью происходит с помощью менеджеров памяти palloc/pfree, предоставляемых PostgreSQL.

Основная функция расширения — функция обработки целого пользовательского запроса на языке Cypher (cypher). Она получает 2 строки: название графа, над которым будет производиться запрос, и саму строку с запросом. Аргументы выделяются с помощью макроса PG\_GETARG\_CSTRING. После этого происходит вызов функции менеджера графовых сущностей execute\_cypher. Возвращаемый тип представляет собой структуру cypher\_return, в которой содержится поле перечисления cypher\_return\_type (листинг 11), позволяющее определить, какие данные пришли от сервера, чтобы выполнить преобразование поля с указателем на структуру данных ответа cypher\_request. Полученный тип аналогичен одному из указанных выше типов возврата. После этого осуществляется возврат указателя на преобразованные данные через макрос PG\_RETURN\_POINTER.

Листинг 11 — Перечисление cypher\_return\_type

enum cypher\_return\_type {

cypher\_nothing,

cypher\_info\_string,

cypher\_nodes,

cypher\_edges

}

Следующая основная функция — create\_graph, с помощью которой можно создать граф с заранее определенными узлами и ребрами. В качестве аргументов принимает название нового графа и список строк, в которых заданы узлы и ребра в формате языка Cypher. Делегирует выполнение функции cypher\_create\_graph и возвращает строку с информацией, был ли создан граф или нет. Для отдельного добавления узлов или ребер предусмотрены функции append\_nodes и append\_edges, каждая из которых принимает название графа и список строк с соотвествующими сущностями в формате Cypher.

Далее идут функции graph\_nodes и graph\_edges, принимающие в качестве аргумента название графа и возвращающие списки всех его узлов и ребер в порядке их добавления в граф (происходит делегация соответствующим функциям cypher\_graph\_nodes и cypher\_graph\_edges).

4.1.2 Скрипт с процедурами на pl/pgSQL

Хранимые процедуры, написанные на языке серверного программирования pl/pgSQL, являются основным компонентом расширения (так как низкоуровневой части может и не быть вовсе). В системе графового хранилища они являются связующими между клиентской стороной и кодом на C, выполняя роль оберток.

Создание процедуры начинается с ключевых слов CREATE FUNCTION, за которыми следует объявление сигнатуры функции с ее аргументами(указываются только типы данных, которые функция принимает). Затем идет ключевое слово RETURN вместе с типом данных, который функция будет возвращать. После этого начинается тело функции либо описание делегирования работы низкоуровневому коду на C. Пример приведен на листинге 12.

Листинг 12 — Пример SQL-процедуры расширения

CREATE FUNCTION graph\_nodes()

RETURNS graph\_nodes

AS 'MODULE\_PATHNAME', 'graph\_nodes'

LANGUAGE C IMMUTABLE STRICT;

В данном случае объявление MODULE\_PATHNAME заменяется на путь, указанный в файле с расширением .control в аналогичной переменной. Ключевое слово IMMUTABLE означает, что функция не может быть изменена или подменена в процессе выполнения запроса, а STRICT означает проверку совпадения типов аргументов функций. Скрипт содержит объявления всех функций, представленных в низкоуровневом коде на C в качестве внешних.

Также были объявлены собственные типы PostgreSQL graph\_node и graph\_edge, аналогичные структурам графа. Для создания типа в самом начале идет объявление с помощью конструкции CREATE TYPE <название типа>, а в конце кода идет его полное определение:

* INTERNALLENGTH — длина объекта типа в байтах (в случае объектов переменной длины устанавливается в -1);
* INPUT — функция, которая переводит строку в определяемый тип;
* OUTPUT — функция, конвертирующая определяемый тип в строку;
* STORAGE — тип хранилища для объекта (установлено в extended для переменного типа).

4.2 Парсер запросов к графовой модели данных

Парсер языка запросов к графовому хранилищу, являющегося подмножеством языка Cypher, состоит из двух частей — лексического и синтаксического анализаторов входного текста.

4.2.1 Лексический анализатор запроса

Анализатор был сгенерирован с помощью генератора лексических анализаторов flex. Основной файл содержит в себе все необходимые правила для поиска лексем во входящей строке, которые заданы как явно, так и с помощью синтаксиса регулярных выражений:

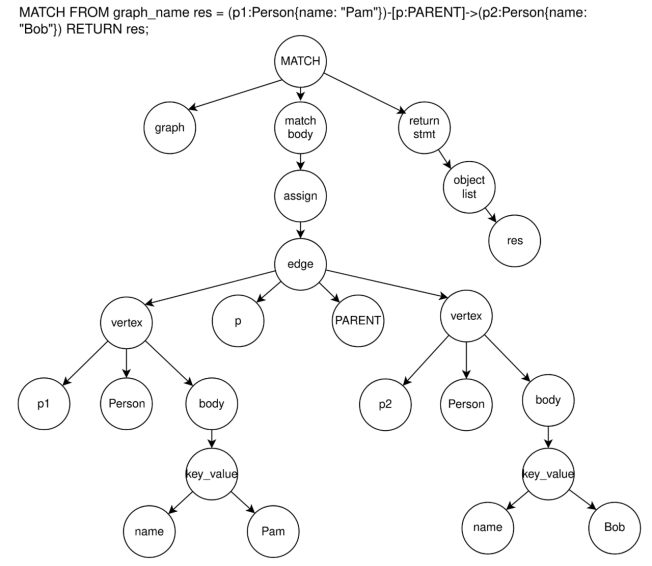
* ключевые слова (CREATE, MATCH, RETURN и тд);
* скобки (фигурные, круглые, прямоугольные);
* знаки препинания, выступающие в качестве разделителей (точка, запятая, точка с запятой, двоеточие);
* регулярное выражение для поиска целых чисел;
* регулярное выражение для поиска слов (имена объектов, строковые значения);
* регулярное выражение для поиска слов, в которые могут входить цифры (метки).

В качестве действия для каждой лексемы определен возврат соответствующего значения перечисления токенов, которое лежит в пространстве имен yy::parser и объявлено в заголовочном файле astgrammar.tab.hh. Данный файл генерируется во время компиляции синтаксического анализатора с помощью программы bison.

Библиотека для разработки лексических анализаторов libfl-devel предоставляет интерфейс к генератору через заголовочный файл FlexLexer.h. В нем объявлен класс генератора FlexLexer, который является генератором по умолчанию. Для того, чтобы обработать особые случаи разбора запроса и создать собственную обертку над лексером для взаимодействия с анализатором синтаксиса, был создан класс AstDriver, входящий в пространство имен класса FlexLexer yy, который хранит в себе указатель на FlexLexer, определяет аналогичные методы обработки очередного входного токена yylex и запуска разбора parse. Метод yylex принимает указатель на структуру, которая содержит тип токена, возвращаемый из правила, и значение, которое было обработано. В нем осуществляется поиск литералов (как целочисленных, так и строковых) и явное преобразование их к соответствующим типам C++: int или std::string. Метод parse создает объект синтаксического парсера из заголовочного файла, генерируемого также bison, и запускает процесс обработки токенов

4.2.2 Синтаксический анализатор. Описание узлов AST-дерева

Синтаксический анализатор был сгенерирован с помощью программы bison из специальных правил, описывающих нисходящую грамматику запроса. Результатом его работы является построенное в памяти абстрактное синтаксическое дерево (AST) всего запроса. Пример построения подобного дерева представлен на рисунке 15.

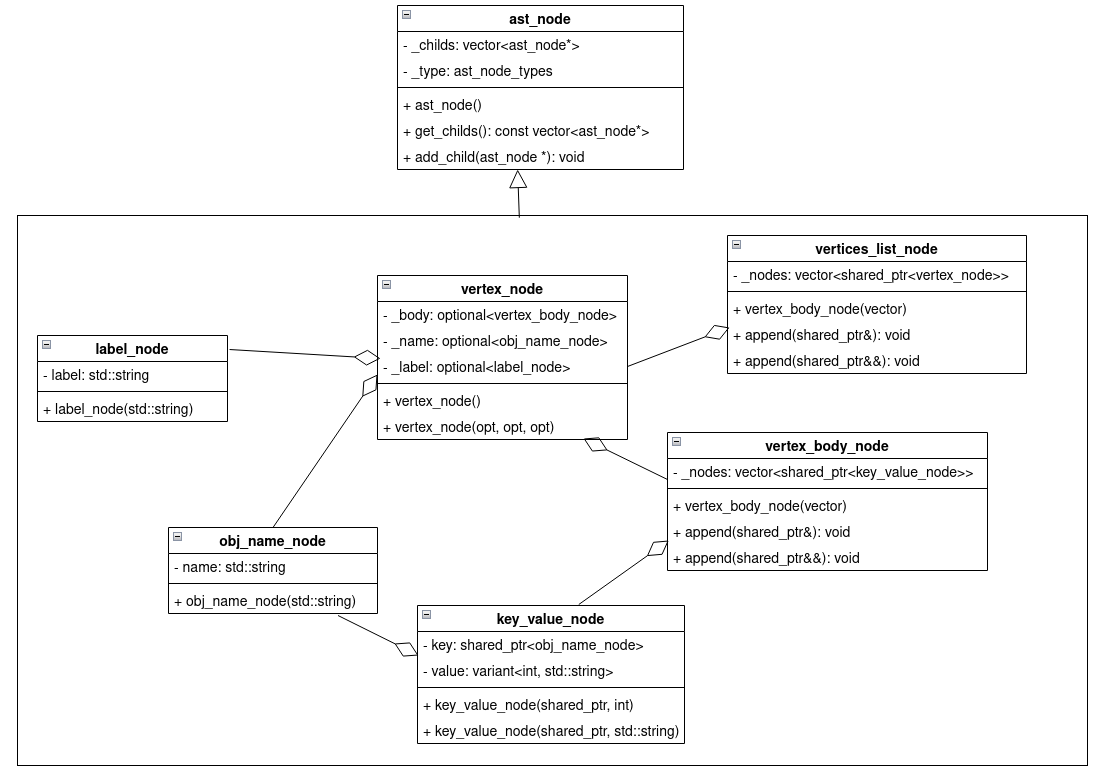
Рисунок 15 — AST-дерево запроса к графовой модели

Для описания и построения AST-дерева были созданы классы, представляющие отдельные синтаксические группы токенов в виде отдельных узлов. Все узлы лежат в пространстве имен cypher::tree и наследуются от базового класса узла дерева ast\_node. Данный класс в качестве protected-членов содержит в себе поле \_childs — std::vector из указателей на другие узлы, которые являются его дочерними узлами, и тип узла \_type — значение перечисления ast\_node\_types. Константный метод get\_childs является геттером для \_childs и возвращает ссылку для предотвращения копирования. Метод add\_child добавляет указатель на узел в список дочерних. Деструктор указан как виртуальный для возможности освобождения выделенной памяти конкретного класса узла через указатель на ast\_node.

Первая схожая синтаксическая группа узлов с это сами узлы графа. Она представлена следующими классами:

* obj\_name\_node — название объекта (узла, ребра, свойства, графа, идентификатора);
* key\_value\_node — пара «ключ-значение»;
* label\_node — метка узла или ребра;
* vertex\_body\_node — список свойств или пар «ключ-значение»;
* vertex\_node — вершина (узел) графа;
* vertices\_list\_node — список вершин.

Диаграмма классов отображает иерархию группы на рисунке 16.

Рисунок 16 — Синтаксическая группа узлов графа

Класс obj\_name\_node содержит в себе поле \_name строкового типа std::string, которое хранит в себе название объекта графа, а также переопределяет виртуальный метод print для вывода информации об узле в стандартный поток вывода std::cout. Класс label\_node аналогичен по функциональности.

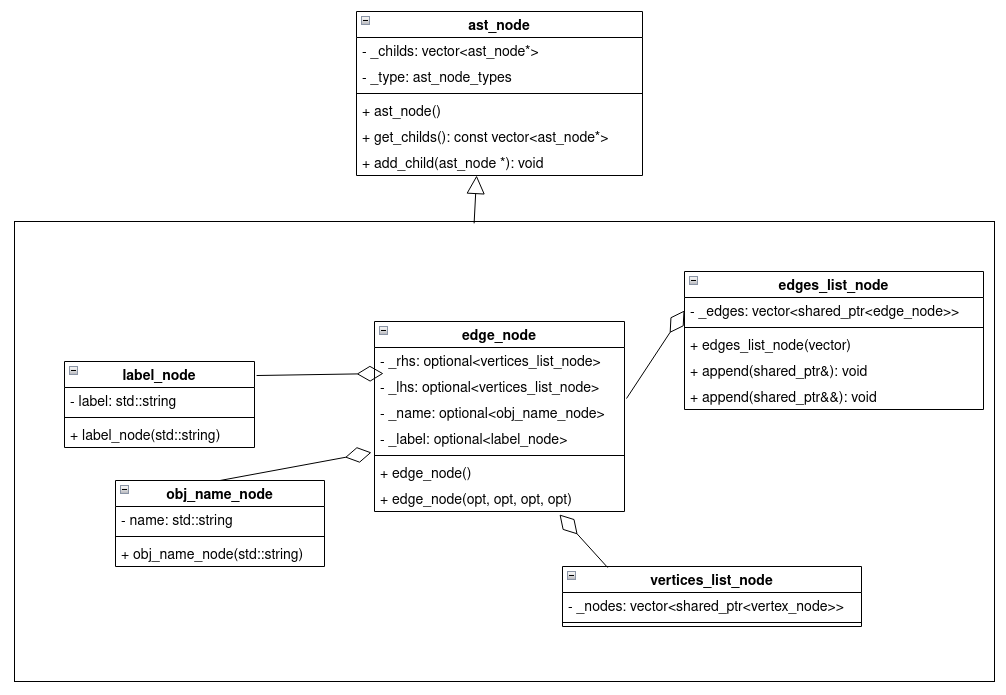
Класс key\_value\_node хранит в себе поле \_key, которое является std::shared\_ptr на obj\_name\_node. std::shared\_ptr<T> — это один из типов «умных указателей» в языке C++, которые автоматически аллоцируют память на куче под переданных при создании их экземпляров и освобождают ее при уничтожении объекта. Отличие shared\_ptr от других заключается в специальном счетчике ссылок на экземпляры этого класса, через которые можно обращаться к одному и тому же участку памяти. Таким способом shared\_ptr реализует концепцию разделяемого владения ресурсом. Второе поле \_value представляет собой класс std::variant, который может хранить в себе объекты типов int (целочисленные значения свойств) или std::string(строковые значения свойств). Std::variant<T, V, …> c это типобезопасный вариант объединения, доступный со стандарта C++17, которое хранит объект одного из перечисленных при специализации шаблона класса variant типов. Значение можно получить с помощью функции стандартной библиотеки std::get<T>, которая предназначена для обработки подобных контейнеров времени компиляции. key\_value\_node также переопределяет метод print.

Класс vertex\_body\_node хранит поле списка пар «ключ-значение» (список представлен классом std::vector, пары — умными указателями shared\_ptr на объекты key\_value\_node). Конструктор принимает ссылку на аналогичный список. Метод append предназначен для добавления новой пары (или свойства узла графа), причем перегружается как для lvalue-ссылки (один амперсанд), так и для rvalue-ссылки (два амперсанда) для возможности использования легковесной операции обмена двумя значениями по ссылкам std::swap.

Класс vertex\_node содержит поля тела узла \_body, названия узла \_name и метки узла \_label. Каждое поле представляет собой объект std::optional на соответствующий умный указатель на vertex\_body\_node, obj\_name\_node или label\_node. std::optional — это контейнер-обертка над типом, которая может содержать в себе объект этого типа, а может и не содержать. Если значения нет, то optional содержит тип std::nullopt. Так как части узла не являются обязательными при написании запроса, то optional подходит здесь наилучшим образом.

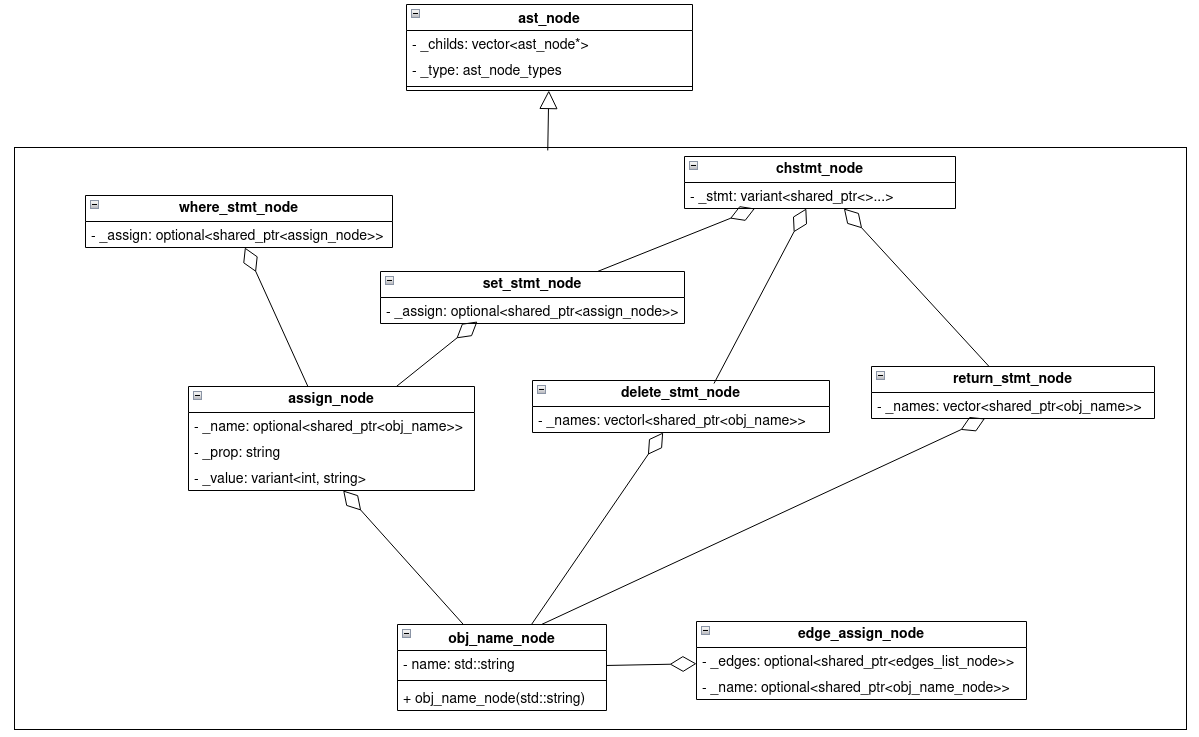
Класс vertices\_list\_node аналогичен по функциональности классу vertex\_body\_node, вместо указателей на key\_value\_node используются указатели на vertex\_node.

Следующая синтаксическая группа связывает ребро и список ребер. Диаграмма классов приведена на рисунке 17.

Рисунок 17 — Синтаксическая группа ребер графа

Класс ребра edge\_node хранит в себе следующие поля: \_label — метка ребра; \_name — идентификатор ребра при выполнении запроса; \_rhs и \_lhs — ссылки на узлы. Узлы представлены объектами класса vertices\_list\_node, так как он является обобщением как для нескольких узлов, перечисленных через запятую, так и для одного единственного узла. Класс edges\_list\_node представляет собой список ребер, его функциональность аналогична классу vertices\_list\_node.

Дальше идет группа выражений, которые включает в себя результаты выполнения запроса. Диаграмма классов для них приведена на рисунке 18.

Рисунок 18 — Синтаксическая группа выражений в запросе

Здесь можно выделить особую подгруппу узлов AST-дерева — результирующие выражения, которые всегда указываются в конце. Она представлена классом возврата значений, классом установки определенного поля и классом удаления объектов.

Класс возврата значений return\_stmt\_node (ключевое слово RETURN) хранит в себе поле \_names, представляющее собой список указателей на объекты идентификаторов obj\_name\_node, так как можно вернуть несколько значений узлов или ребер (аналогично можно просто выйти из запроса без возврата значений, указав чистый RETURN).

Класс обновления значения для поля узла set\_stmt\_node (ключевое слово SET) содержит в себе поле \_assign, которое является возможным указателем на экземпляр класса описания присваивания assign\_node (в классе предусмотрена возможность указания для основного идентификатора дополнительно его поля).

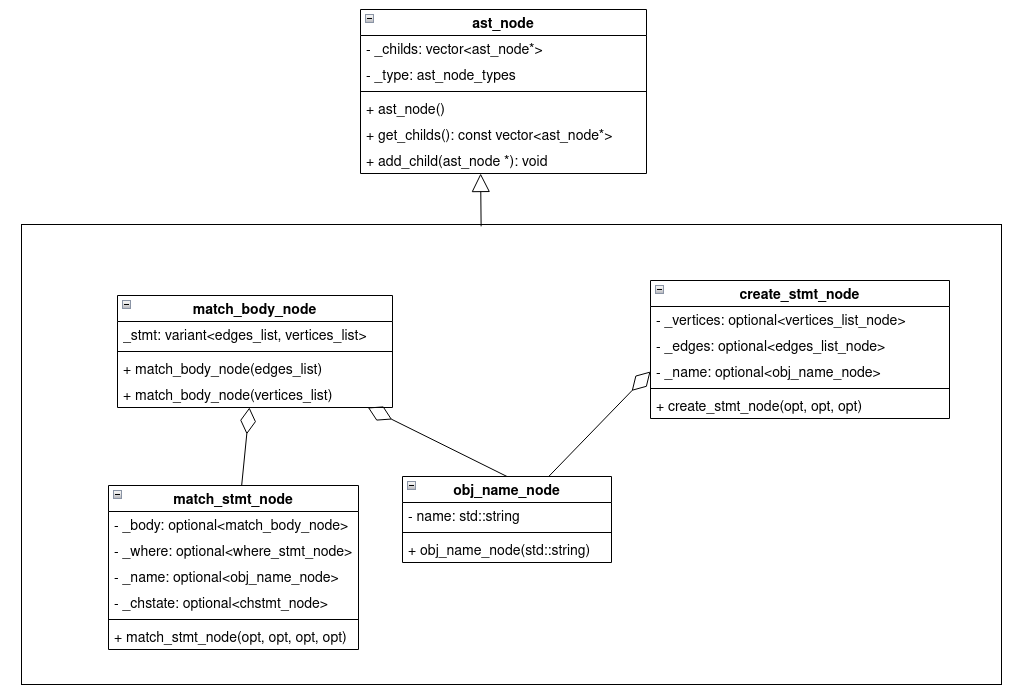
Класс удаления объектов графа по идентификаторам delete\_stmt\_node (ключевое слово DELETE) аналогичен по своей структуре классу return\_stmt.

Класс присваивания значения идентификатору или его полю (если это узел) assign\_node хранит в себе указатель на идентификатор \_ident, вариабельное значение целого или строкового типа \_value и необязательное строковое значение \_prop, которое представляет собой свойство из тела узла графа. Конструктор узла перегружается по полю \_value (первый вариант — целое число, второй — строка).

Класс присваивания ребра edge\_assign\_node используется в основном в запросах выборки, когда необходимо вернуть целое ребро или их список. Он содержит поле \_edge, которое является указателем на экземпляр класса edges\_list\_node, но хранит в себе одно ребро; поле \_name — указатель на идентификатор, которому происходит присваивание.

Класс условия where\_stmt\_node используется для фильтрации выбираемых объектов, в основном узлов по их свойствам, и аналогичен пофункциональности оператору WHERE в SQL. Он содержит в себе указатель на экземпляр присваивания, который в данном случае используется как проверка на равенство.

Последняя синтаксическая группа — типы графовых запросов. Их диаграмма классов представлена на рисунке 19.

Рисунок 19 — Синтаксическая группа типов запроса

Данные классы представляют собой описания запросов MATCH и CREATE в Cypher. Класс match\_body\_node представляет собой список узлов или ребер, которые будут выбраны из графового хранилища и над которыми будут производиться необходимые преобразования. В классе match\_stmt\_node хранятся все основные части запроса: \_name — название хранилища, откуда будет производиться выборка данных; \_body — данные выборки (экземпляр match\_body\_node); \_where - необязательное условие фильтрации узлов и ребер вместе с ними; \_chstate — результирующее выражение запроса;

Класс create\_stmt\_node описывает как создание нового графа, так и добавление отдельных его компонентов уже в существующее хранилище. Графовые структуры для добавления представлены полями \_vertices (узлы) и \_edges (ребра), каждое из которых является необязательным, так как они могут быть добавлены по отдельности. Также имеется поле \_name для хранения имени нового или существующего графа.

4.2.3 Генерация синтаксического анализатора с помощью bison

Для генерации синтаксического анализатора в специальном файле были описаны все необходимые правила нисходящей грамматики. Разбор начинается от общего текста запроса к его наименьшим синтаксическим узлам. Для поиска узлов были заранее обозначены все действующие токены (в данном контексте это реальные части запроса: строка, число, идентификатор или метка) и нетерминальные символы (сочетания токенов, которые представляют абстрактную часть запроса) с помощью директив %token и %nterm. Для токенов и нетерминалов задаются типы, с помощью которых они будут представлены в исходном коде для дальнейшей обработки. Здесь токены представлены стандартными типами значений запроса в виде целого числа и строки, а для каждого нетерминала поставлено в соответствие его описание в виде класса узла AST-дерева. В грамматических правилах обработки очередного нетерминального символа содержится код на C++, в котором происходит создание текущего узла дерева, при этом рекурсивно создаются все использующиеся нетерминалы в строке. Для избежания копирования тяжелых объектов узлов и были введены интеллектуальные указатели в виде объектов std::shared\_ptr<T>, которые дополнительно при создании дочерних узлов заносятся в список \_childs. Пример подобного грамматического правила приведен в листинге 13.

Листинг 13 — Правило разбора запроса CREATE

create\_stmt: CREATE GRAPH obj\_name LBRACE vertices\_list RBRACE COMMA LBRACE edges\_list RBRACE {   
 $$ = std::make\_shared<create\_stmt\_node>(std::move(std::make\_optional<std::shared\_ptr<object\_name\_node>>($3)),  
 \_optional<std::shared\_ptr<vertices\_list\_node>>($5)),  
 \_optional<std::shared\_ptr<edges\_list\_node>>($9)));   
 }  
 | CREATE GRAPH obj\_name LBRACE vertices\_list RBRACE {  
 $$ = std::make\_shared<create\_stmt\_node>(std::move(std::make\_optional<std::shared\_ptr<object\_name\_node>>($3)),  
 \_optional<std::shared\_ptr<vertices\_list\_node>>($5)),  
 std::nullopt);  
 }  
;

Для перемещения экземпляров узлов в соответствующие поля при создании нового узла используется функция std::move, которая в данном случае присваивает указатель новому объекту shared\_ptr при конструировании. Таким образом предотвращаются излишние расходы на копирование.

Итоговое дерево хранится в классе лексера AstDriver. Для возможности использования токенов, найденных лексическим анализатором, в секции кода правил были объявлены функции parse и yylex в пространстве имен yy (определены же они в AsrDriver). С помощью подобного способа происходит связывание лексического и синтаксического анализаторов кода в единый парсер запросов при компиляции.

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.

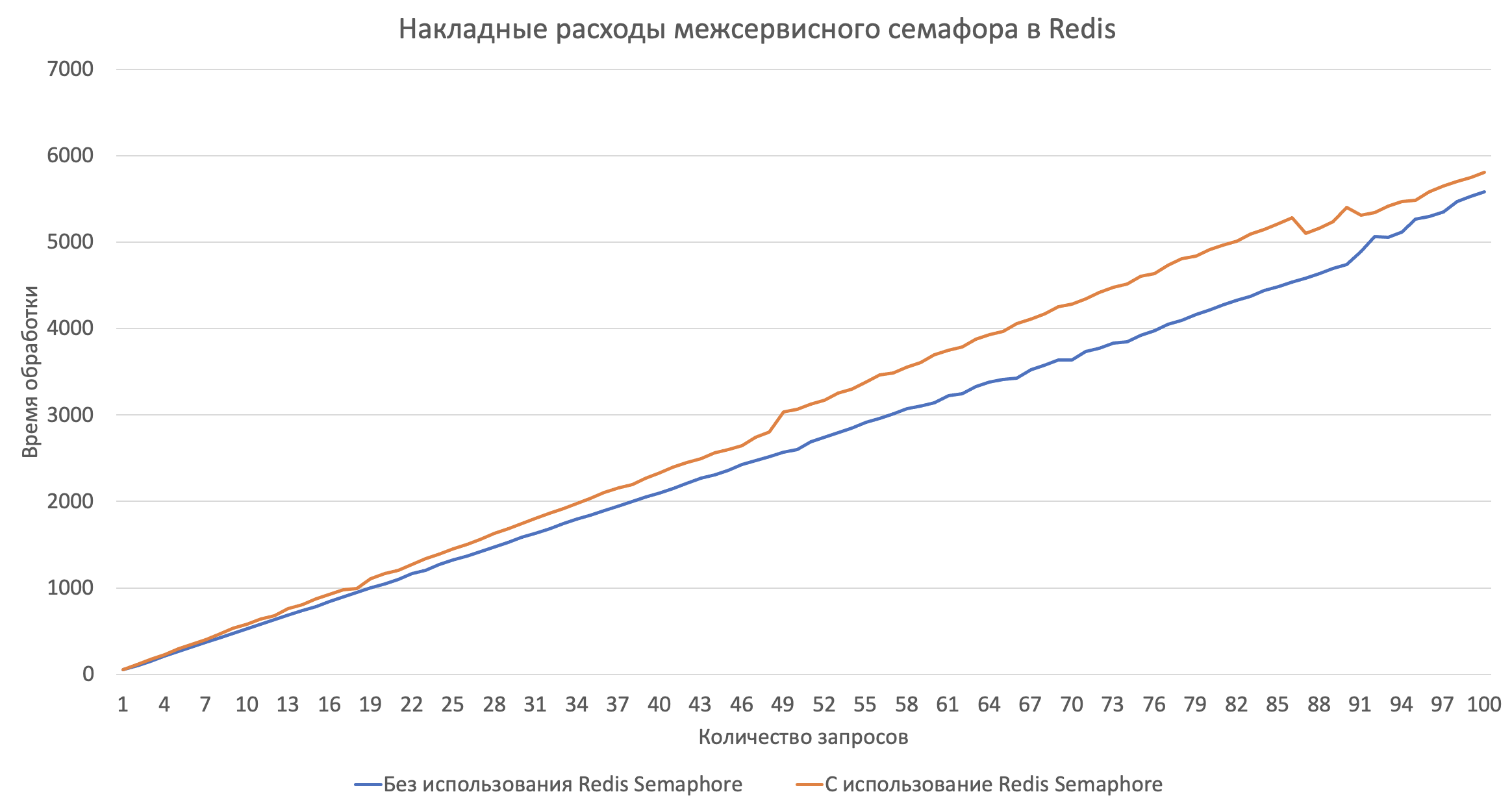


Рисунок 16 - График сравнения времени работы с использованием межсервисного семафора в Redis и без использования межсервисного семафора в Redis

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.

## 