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

Воронеж 2024

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: __.__.2024
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 Neo4j – графовая СУБД	09.12.2023 – 15.12.2023	
8	2.1.2 Apache AGE – расширение для	16.12.2023 –	

	PostgreSQL	22.12.2023	
9	2.2 Модель графа свойств	23.12.2023 – 28.12.2023	
10	2.3 Язык графовых запросов Cypher	29.12.2023 – 04.01.2024	
11	2.4 Средства построения расширений СУБД PostgreSQL	05.01.2024 – 10.01.2024	
12	3. Выбор средств реализации системы	11.01.2024 – 17.01.2024	
13	3.1 libpq – клиентский интерфейс к PostgreSQL	18.01.2024 – 25.01.2024	
14	3.2 flex – генератор лексических анализаторов	26.01.2024 – 01.02.2024	
15	3.3 bison – генератор синтаксических анализаторов	02.02.2024 – 10.02.2024	
16	3.4 CMake – система сборки проектов	11.02.2024 – 15.02.2024	
17	3.5 Clion – среда разработки (IDE)	16.02.2024 – 22.02.2024	
18	4. Реализация	23.02.2024 – 27.02.2024	
19	4.1 Расширение для PostgreSQL	28.02.2024 – 05.03.2024	
20	4.1.1 Процесс загрузки расширения в сервер СУБД	06.03.2024 – 10.03.2024	
21	4.1.2 Низкоуровневая составляющая расширения	11.03.2024 – 17.03.2024	
22	4.1.3 Скрипт с процедурами на pl/pgSQL	18.03.2024 – 23.03.2024	
23	4.2 Парсер запросов к графовой модели данных	24.03.2024 – 28.03.2024	
24	4.2.1 Лексический анализатор запроса	29.03.2024 – 03.04.2024	
25	4.2.2 Синтаксический анализатор. Описание узлов AST-дерева	04.04.2024 – 08.04.2024	
26	4.2.3 Генерация синтаксического анализатора с помощью bison	09.04.2024 – 13.04.2024	
27	4.3 Структура файлов графового хранилища	14.04.2024 – 19.04.2024	
28	4.4 Алгоритм построения графа из файлов	20.04.2024 – 23.04.2024	
29	4.4.1 Построение узлов и свойств узлов	24.04.2024 – 26.04.2024	
30	4.4.2 Построение ребер графа	27.04.2024 – 30.04.2024	
31	4.5 Реализация менеджера графовых сущностей	01.05.2024 – 05.05.2024	
32	4.5.1 Хранилище свойств узлов графа	06.05.2024 – 09.05.2024	

33	4.5.2 Хранилище узлов и ребер графа	10.05.2024 – 12.05.2024	
34	4.5.3 Класс графового менеджера	13.05.2024 – 15.05.2024	
35	4.6 Интерфейс командной строки Cypher CLI	16.05.2024 – 19.05.2024	
36	5 Результат работы программы	20.05.2024 – 22.05.2024	
37	Заключение	23.05.2024 – 26.05.2024	
38	Список использованных источников	27.05.2024 – 30.05.2024	

Обучающийся _____ Мамонов Д. В.

Руководитель _____ Самойлов Н. К.

РЕФЕРАТ

Бакалаврская работа 55 с., 25 рис., 0 табл., 15 использованных источников, 0 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

ГЛОССАРИЙ

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

NoSQL — класс систем управления базами данных, не относящихся к реляционным СУБД.

SQL-процедура — это предварительно скомпилированный и сохранённый набор SQL-инструкций, который может быть выполнен на сервере базы данных по запросу.

Граф свойств — это специализированный вид графа, который используется для представления объектов и их взаимосвязей, где каждый узел (или вершина) является объектом, а рёбра между узлами указывают на отношения или связи между объектами.

Свободный индекс — указатель или ссылка на следующую доступную позицию в индексной структуре, где может быть добавлен новый элемент или узел графа.

Система сборки — это инструмент или набор инструментов, который автоматизирует процесс сборки, компиляции и упаковки программного обеспечения.

Лексический анализатор — программное обеспечение, которое разбивает входной текст на определенный поток лексем (токенов).

Синтаксический анализатор — программное обеспечение, которое анализирует входной текстовый поток или файл согласно определённым синтаксическим правилам и грамматике языка программирования или естественного языка.

GNU GPL — GNU General Public License, одна из самых распространенных свободных лицензий на программное обеспечение.

TLS/SSL — протоколы защищенной передачи данных с помощью асимметричного шифрования на открытых и закрытых ключах.

Динамическая библиотека — исполняемый файл, содержащий общий код и ресурсы, которые могут быть использованы несколькими процессами одновременно.

AST-дерево — структура данных, представляющая программу или выражение в виде дерева узлов, каждый из которых является определенной комбинацией лексем.

Умный указатель — это класс, который является оберткой для обычного указателя и автоматически управляет жизненным циклом объекта, на который указывает.

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 – графовая СУБД

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 – расширение для PostgreSQL

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.

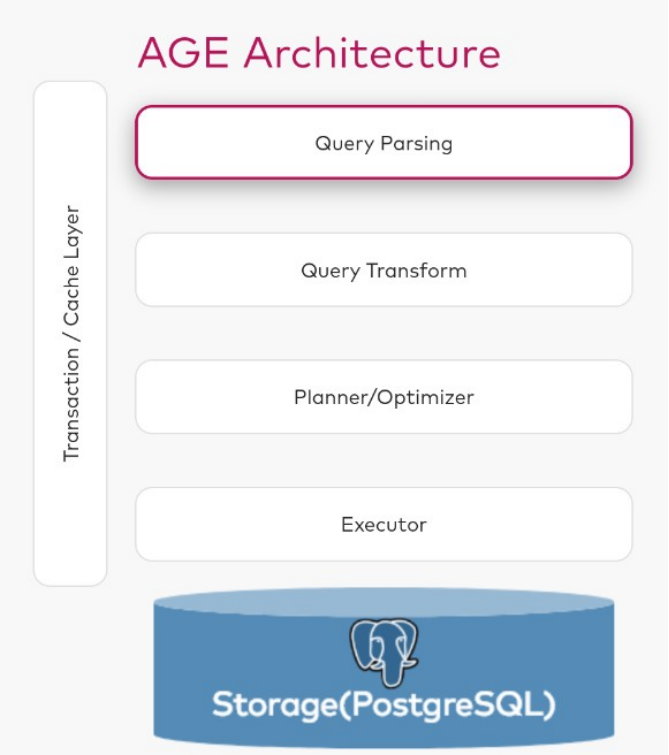

Рисунок 1 – Структура расширения Apache AGE

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.

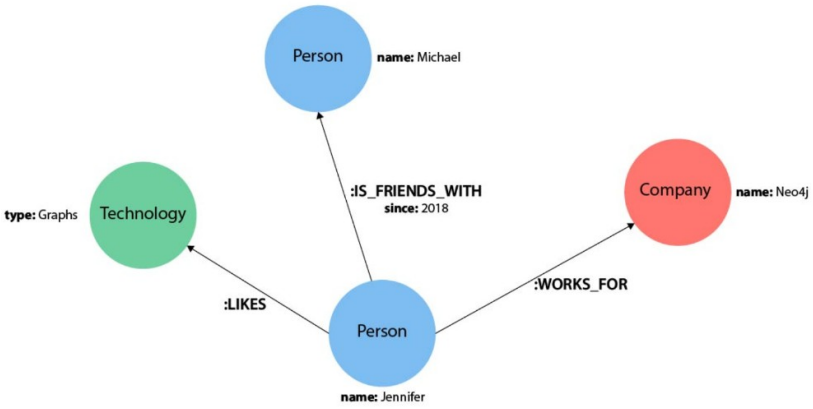

Рисунок 2 – Пример графа свойств

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
```

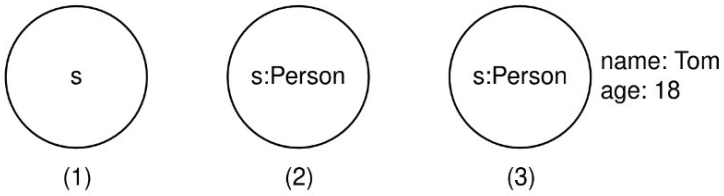

Рисунок 3 – Визуализация узлов

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)
```

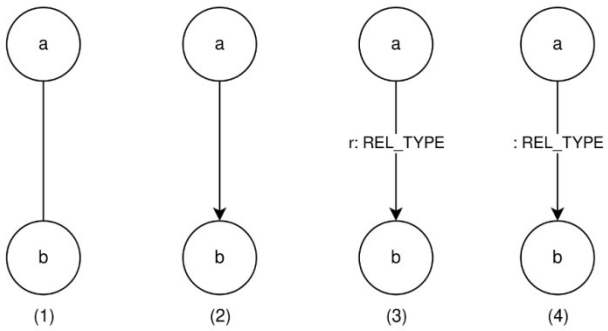

Рисунок 4 - Визуализация отношений в графе

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[6]. 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
```

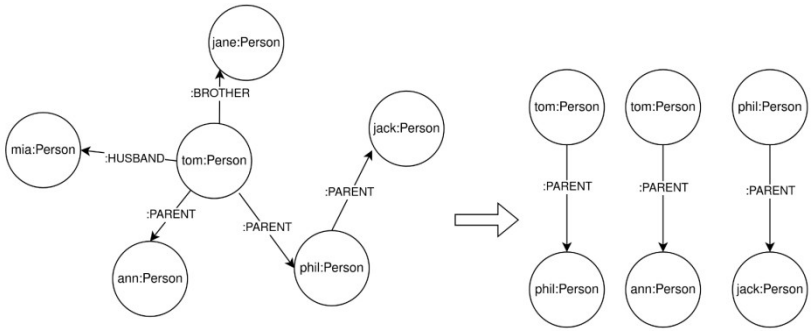

Рисунок 5 - Визуализация выполнения запроса выборки

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)
```

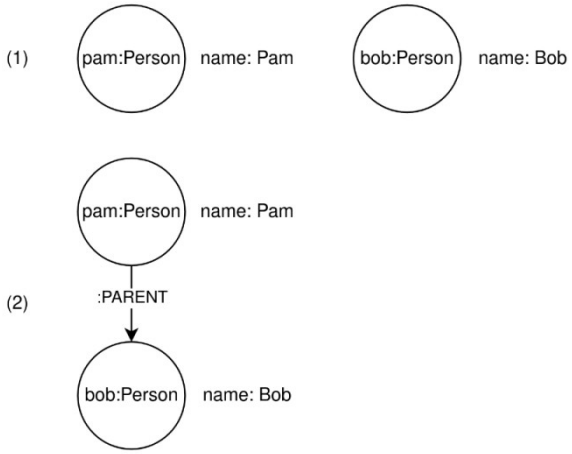

Рисунок 6 - Визуализация процесса создания сущностей в базе данных

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 Средства построения расширений СУБД PostgreSQL

PostgreSQL предлагает расширяемую архитектуру и реализует поверх нее свои внутренние типы данных, операторы, функции, индексы и многое другое[7]. 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 полнотекстового поиска.

3. Доступ к данным за пределами текущей 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.

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.
- Clion — интерактивная среда разработки

3.1 libpq – клиентский интерфейс к PostgreSQL

libpq[8] – 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 – генератор лексических анализаторов

flex[9] — 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 – генератор синтаксических анализаторов

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 этом 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 – система сборки проектов

CMake[11] — система сборки проектов с открытым исходным кодом. Она 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.

3.5 Clion – среда разработки (IDE)

Разработка производилась 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.

4 Реализация

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 7.

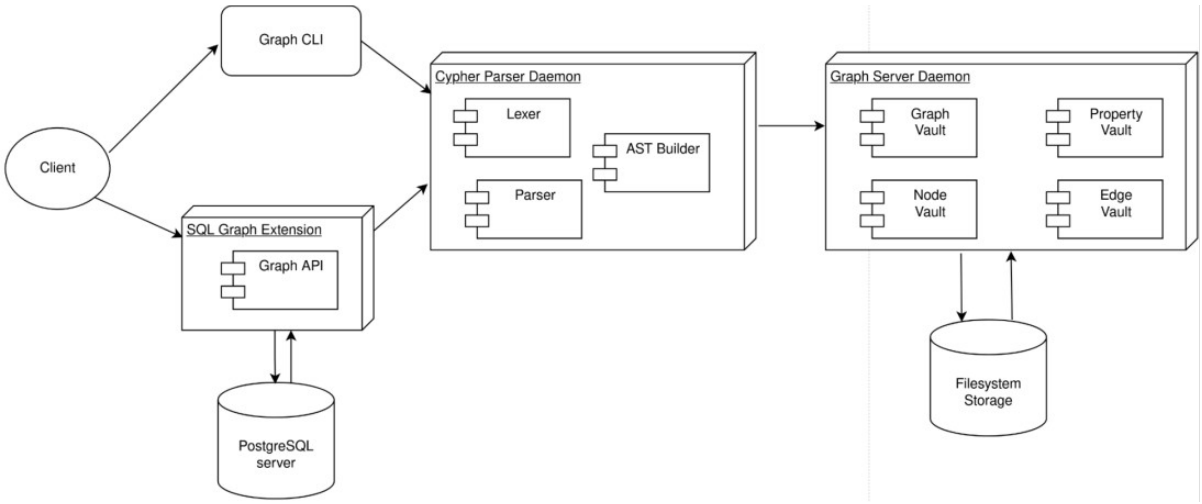

Рисунок 7 – Диаграмма пакетов графового хранилища данных

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 8.

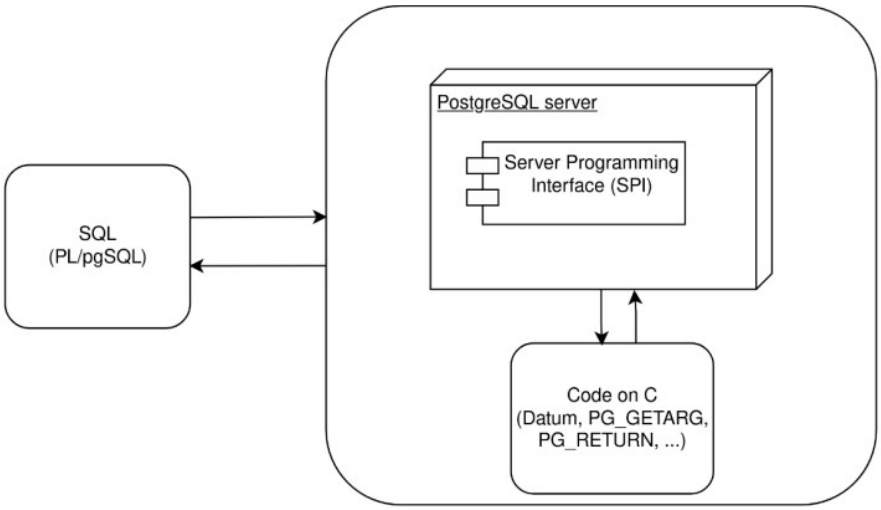

Рисунок 8 – Диаграмма пакетов расширения для PostgreSQL

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 9.

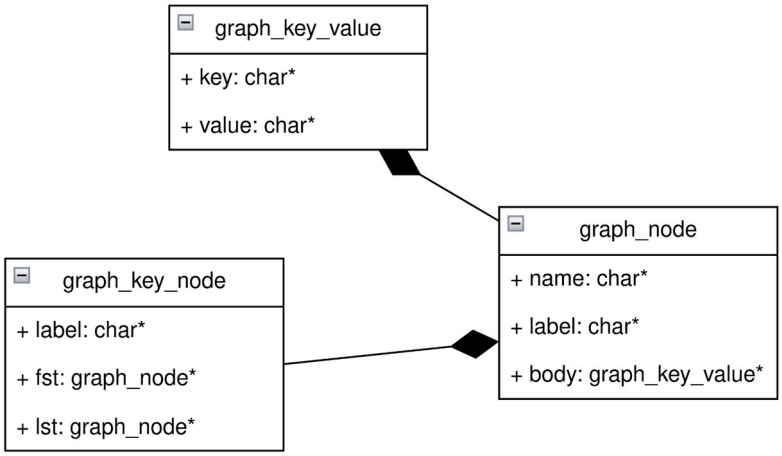

Рисунок 9 — Диаграмма классов графовых сущностей для SQL

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).

4.1.3 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[14].

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 10.

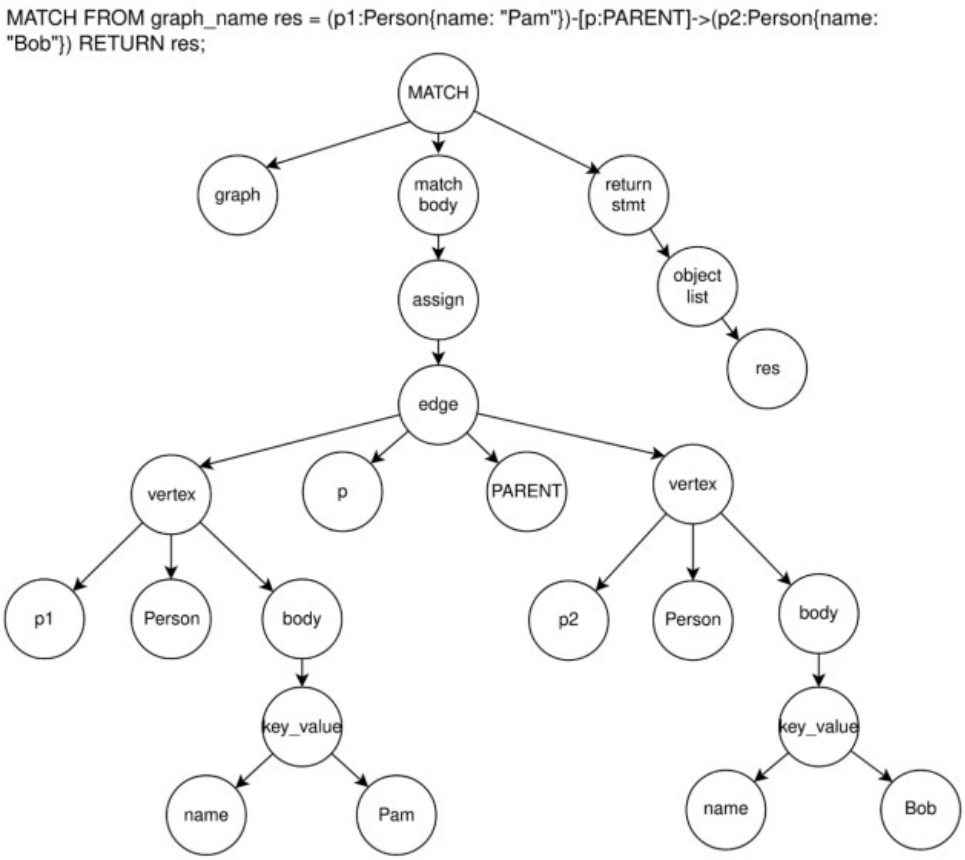

Рисунок 10 — AST-дерево запроса к графовой модели

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 11.

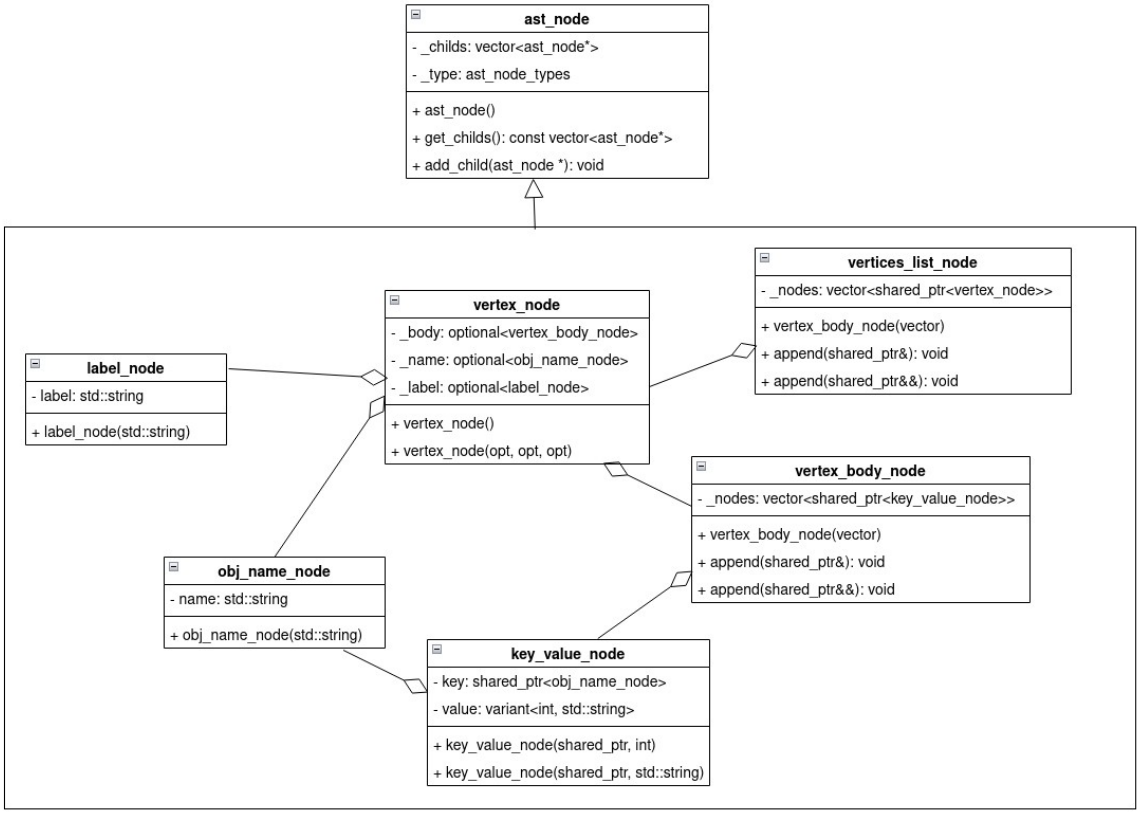

Рисунок 11 — Синтаксическая группа узлов графа

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, …> с 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 12.

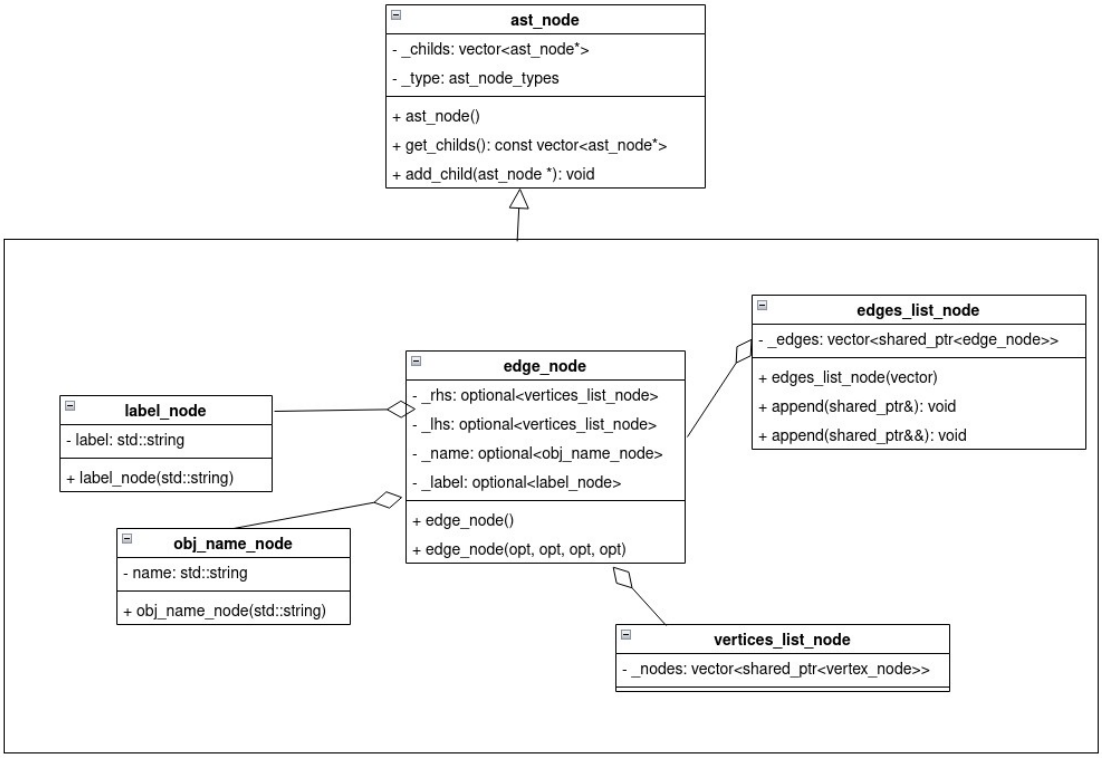

Рисунок 12 — Синтаксическая группа ребер графа

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 13.

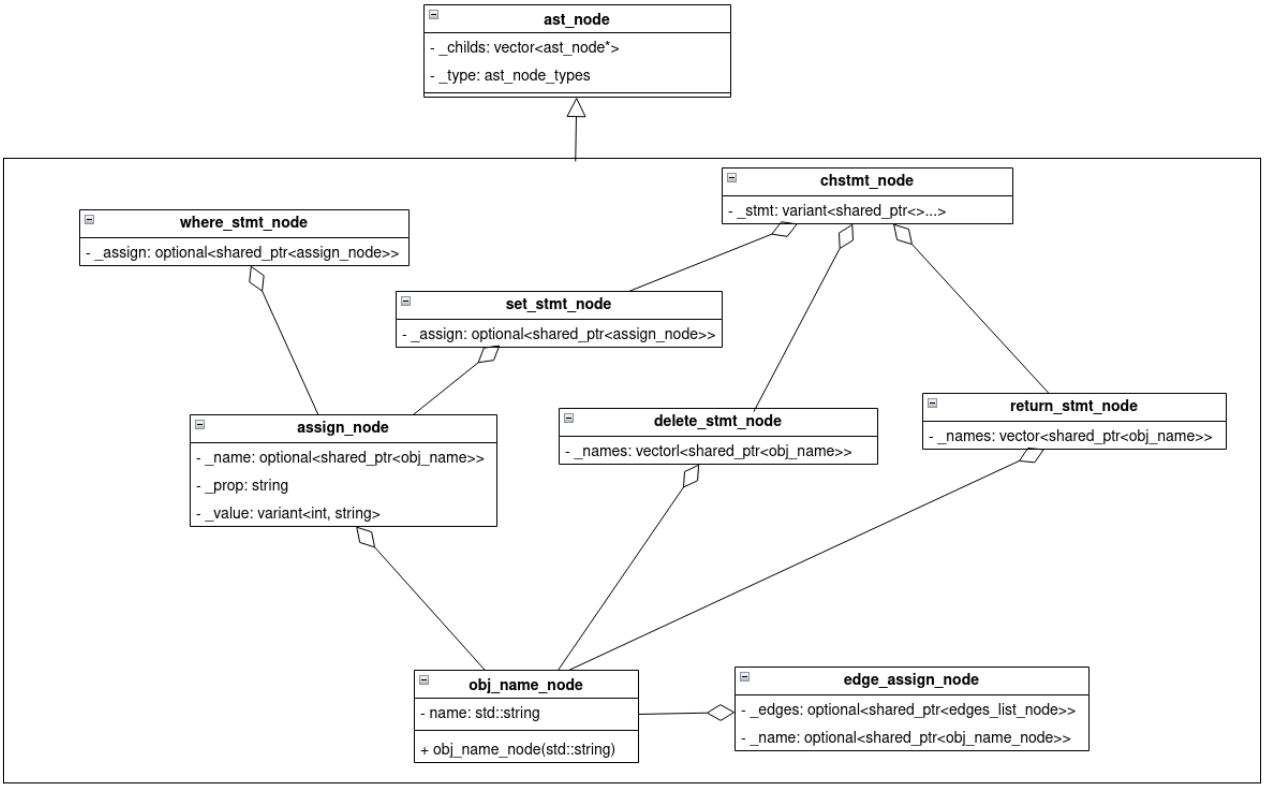

Рисунок 13 — Синтаксическая группа выражений в запросе

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 14.

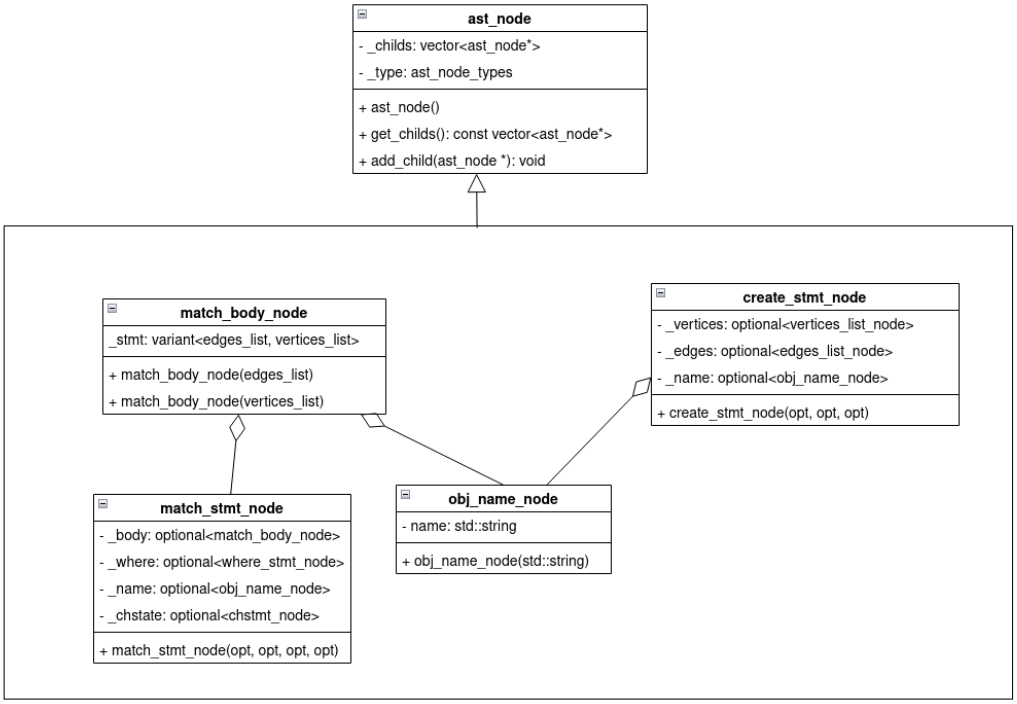

Рисунок 144 — Синтаксическая группа типов запроса

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[15]. 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 AstDriver). С помощью подобного способа происходит связывание лексического и синтаксического анализаторов кода в единый парсер запросов при компиляции.

4.3 Структура файлов 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 15. 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 16. 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.

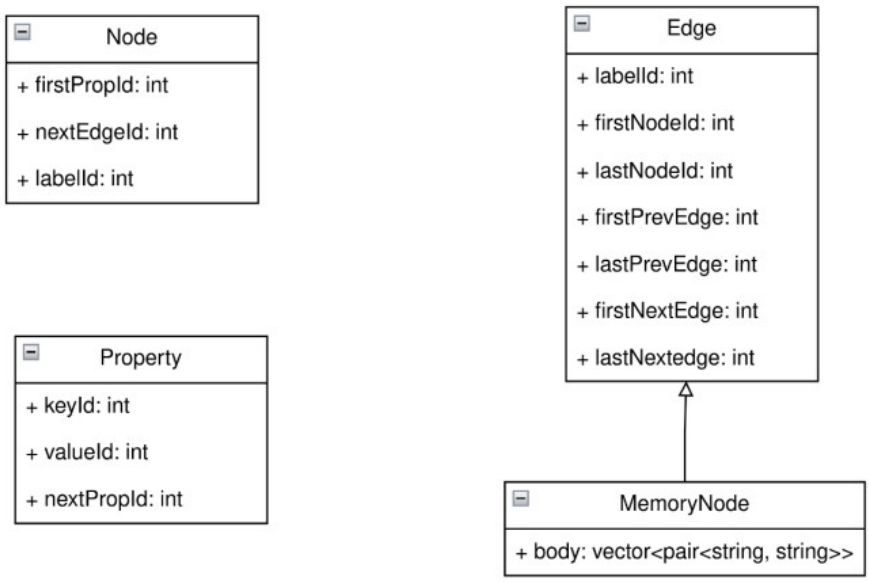

Рисунок 15 — Диаграмма класса сущностей в графовом хранилище

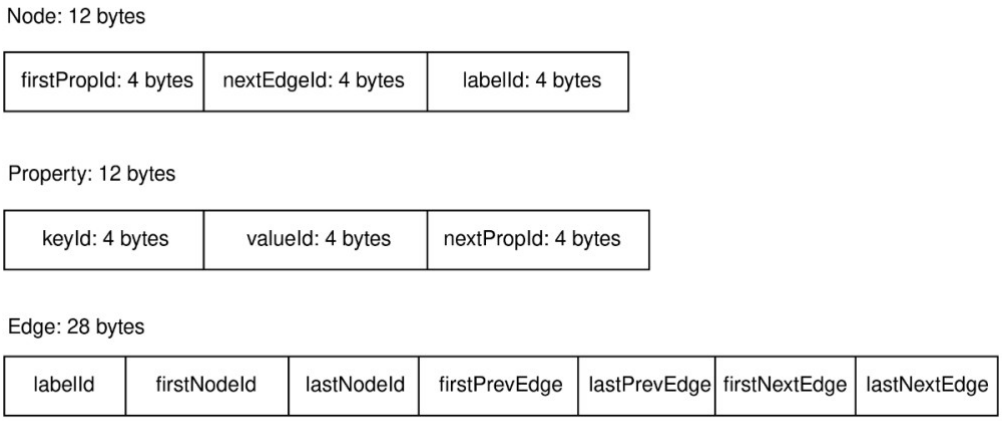

Рисунок 16 — Представление сущностей в файлах

4.4 Алгоритм построения графа из файлов

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

4.4.1 Построение узлов и свойств узлов

Пусть задан граф как на рисунке 17, 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.

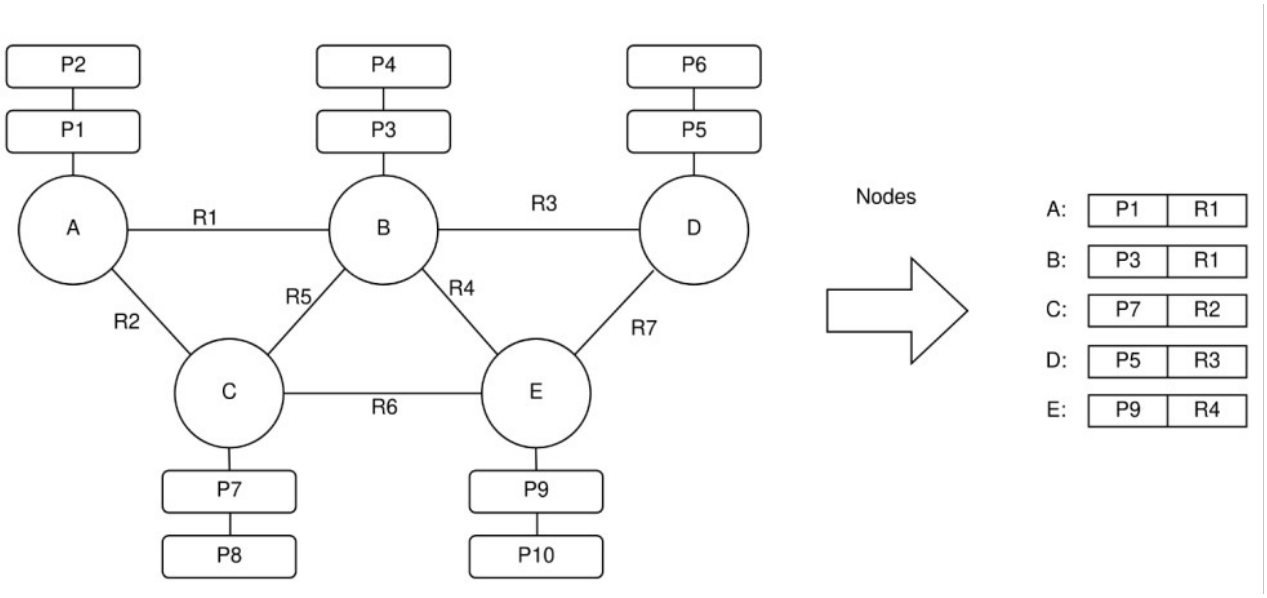

Рисунок 17 — Определение узлов в заданном графе

4.4.2 Построение ребер графа

После этого для каждого узла в ребре ищутся первые попавшиеся ребра из заданного списка ребер, как на рисунке 18, 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 19. 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.

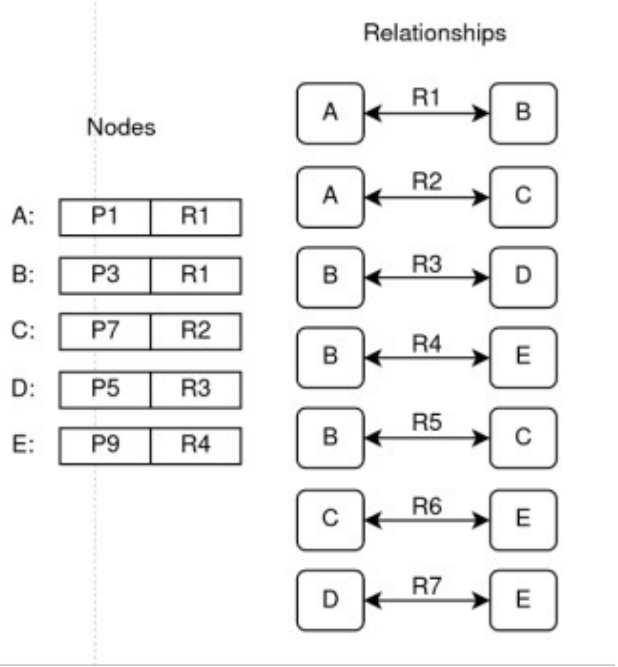

Рисунок 18 — Заданный исходный список ребер и построенные узлы

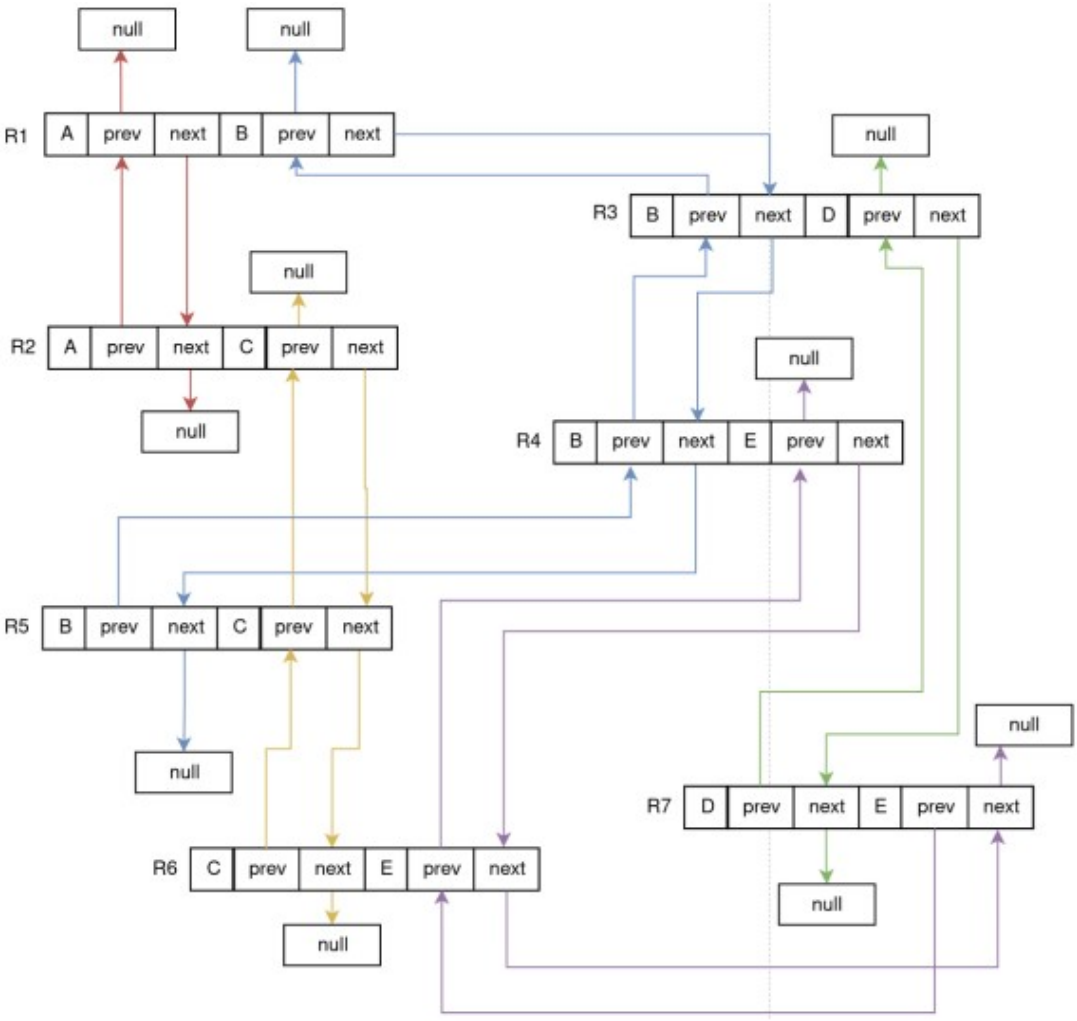

Рисунок 19 — Результат построения графа

4.5 Реализация менеджера графовых сущностей

В систему входит отдельный модуль менеджера графовых сущностей, отвечающего за управление данными графов в бинарных и текстовых файлах операционной системы, а также за выполнение алгоритмов обхода графа и поиска в нем узлов и ребер по различным условиям.

4.5.1 Хранилище свойств узлов графа

Так как, согласно модели графа свойств, все свойства узла можно представить в виде словаря или списка пар «ключ-значение», то все составляющие свойств (и ключи, и значения) можно хранить в строковом виде в текстовом файле (целочисленные значения при необходимости конвертируются из строковых). При этом должен быть создан второй бинарный файл, в котором

хранятся структуры с идентификаторами нужных ключей и значений, в данном случае, номерами строк в текстовом файле, а также ссылкой на позицию следующего абстрактного свойства в теле узла. Такую концепцию реализуют классы StringsVault и PropVault. Их диаграмма классов приведена на рисунке 20.

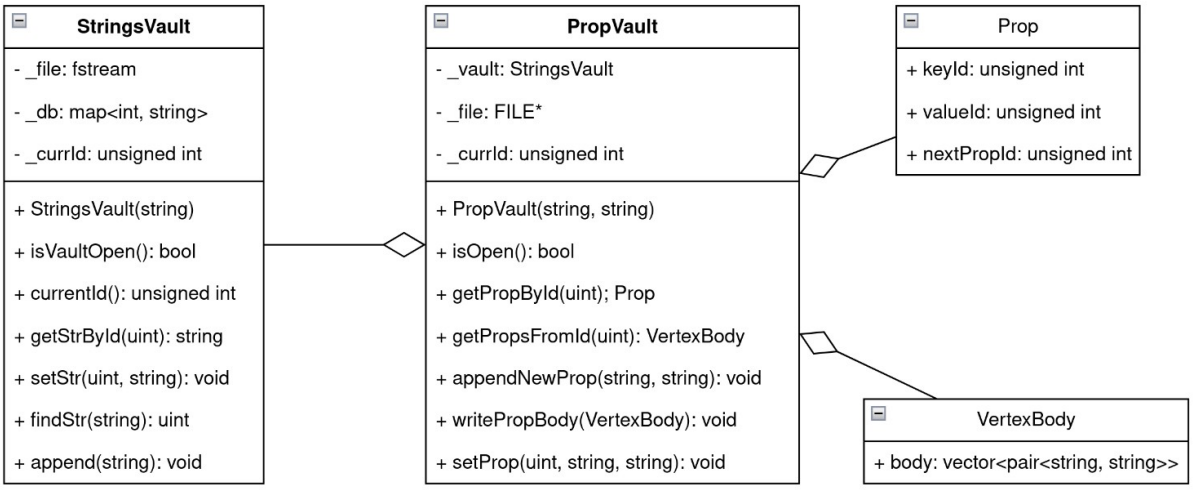

Рисунок 20 — Диаграмма классов хранилища свойств графа

Класс StringsVault представляет собой строковое хранилище и управление над текстовым файлом. Он содержит следующие поля: _file — является потоком над файлом std::fstream с вожможностью чтения/записи; _db — словарь std::map с целочисленным типом ключа и строковым типом значения, представляющий собой кэш недавно прочитанных данных из файла по типу LRU-кеша; _currId — текущий идентификатор (номер строки в файле) последнего записанного значения. Для хранилища доступны операции проверки доступа к хранилищу, получение и обновление строки по ее идентификатору в файлу, поиск строки в файле и добавление нового значения в конец файла.

Класс PropVault оперирует экземплярами структуры Prop, представляющей собой абстрактное свойство, описанное выше. Его отличие от StringsVault заключается в том, что для файла используется указатель на структуру FILE из стандартной библиотеки языка программирования C, так как над таким типом проще производить операции расчета позиции в файле и вычисления смещений для точного чтения всей структуры Prop с помощью

функций fseek и ftell. Класс позволяет также строить сразу все тело узла с помощью экземпляра типа VertexBody, который является псевдонимом для вектора пар строковых значений. Для построения используется метод getPropsFromId, который принимает идентификатор самого первого свойства в теле. Метод writeVertexBody записывает полностью все свойства из тела в конец файла, так что для узла можно гарантировать непрерывность его объекта в памяти.

4.5.2 Хранилище узлов и ребер графа

За управление сущностями узла графа и ребра отвечают классы NodeEdgeVault, Node и Edge. Диаграмма классов для них приведена на рисунке 21.

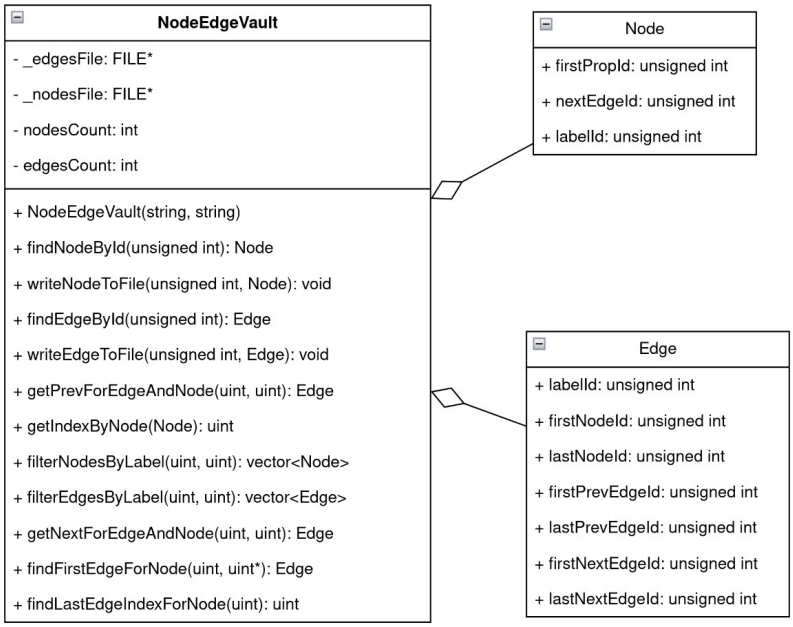

Рисунок 21 — Диаграмма классов хранилища ребер и узлов

Классы сущностей Node и Edge аналогичны тем, которые были представлены на диаграмме на рисунке 7. Для каждого из них создается собственный бинарный файл (поля _edgesFile и _nodesFile в классе NodeEdgeVault). В классе хранилища доступны операции: поиска сущностей по их идентификатору в файлах (findNodeById, findEdfeById); записи в файл в определенное место (writeNodeToFile, writeEdgeToFile); получения

предыдущего и следующего ребер для определенного ребра и узла, которое принадлежит ребру (getPrevForEdgeAndNode, getNextForEdgeAndNode); фильтрации узлов и ребер в моменте выборки по идентификатору метки (filterNodesByLabel, filterEdgesByLabel); поиск первого и последнего ребер для узла в цепочке обхода (findFirstEdgeForNode, findLastEdgeIndexForNode).

4.5.3 Класс графового менеджера

За централизованное взаимодействие с хранилищами сущностей графа отвечает класс Graph. Его диаграмма с описанием представлена на рисунке 22.

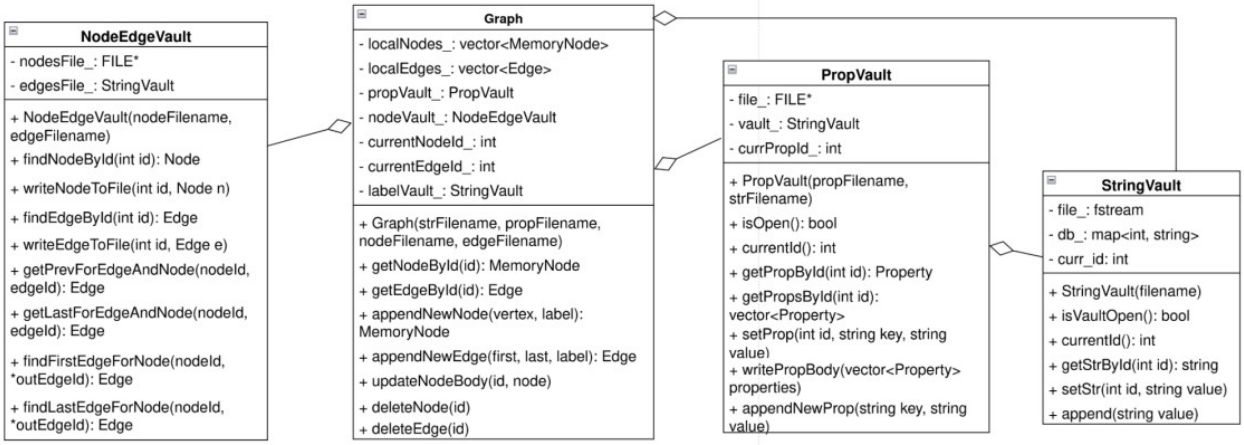

Рисунок 22 — Класс графового менеджера Graph

В данном классе вместо упрощенной файловой структуры Node используется унаследованная структура MemoryNode, которая расширяется полем body класса VertexBody, то есть хранит в себе полностью данные об узле графа. Остальные структуры Prop и Edge аналогичны описанным выше.

Класс Graph реализует в себе методы выборки всех узлов и ребер, в том числе с фильтрацией (getNodesByLabel, getNodesContainsBody); выборки всех ребер, которые соединены с определенным узлом (getEdgesByNode); получения узлов и ребер по их идентификаторам (getNodeById, getEdgeById); добавление новых сущностей в граф (appendNewNode, appendNewEdge); обновление тела узла через операцию SET (updateNodeBody); удаления узлов и ребер (deleteNode, deleteEdge).

4.6 Интерфейс командной строки Cypher CLI

Для графового хранилища был разработано отдельное консольное, который представляет интерфейс командой строки с возможностью написания запросов на языке Cypher напрямую к хранилищу данных. Само приложение состоит из основной функции считывания запроса с потока ввода getQueryFromStdin, которая для отправки строки на сервер использует те же заголовочные файлы, что и функции расширения PostgreSQL. При этом структуры данных и ответы сервера преобразуются в строки через функции nodeToString, edgeToString и выводятся на стандартный вывод программы. Для закрытия необходимо ввести exit или нажать сочетание клавиш Ctrl+D.

5 Результат работы программы

Весь код был скомпилирован и протестирован на дистрибутиве ОС Linux Fedora 40. Для тестирования разбора запросов был собран отдельный исполняемый файл, который принимает на стандартный поток ввода строку запроса и печатает построенное AST-дерево в стандартный поток вывода (в терминал). Результат разбора MATCH-запроса с условием приведен на рисунке 23.

```
MATCH FROM graph (p1:Person)-[p:PARENT]->(p2:Person) WHERE p2.name = "Bob" RETURN p1, p2;

MATCH query:
  -- object value: graph
  -- match query body:
    -- edge assign statement:
      -- edges list:
        -- edge:
          -- object value: p
          -- label value: PARENT
          -- vertices list:
            -- vertex:
              -- object value: p1
              -- label value: Person
          -- vertices list:
            -- vertex:
              -- object value: p2
              -- label value: Person
  -- where statement:
  -- assign:
    -- object value: p2
   = "Bob"
   -- change statement:
    -- return statement:
      -- object list:
        -- object value: p2
        -- object value: p1
```

Рисунок 23 — AST-дерево запроса

По выводу в терминал можно увидеть все отдельные токены и промежуточные узлы дерева, которые были построены из нетерминальных символов.

Для запуска расширения использовался дистрибутив PostgreSQL 15.4 и консольный клиент psql. На рисунке 24 приведено создание простого графа свойств и выборка узлов из него.

```
psql (15.4, сервер 14.10)
Введите "help", чтобы получить справку.

postgres=# SELECT * FROM cypher("new_graph", "
CREATE (:Person {name: "Zeus"})-[:FATHER_OF]->(:Person {name: "Persey"})
") AS (f);
 a
---
(0 rows)

postgres=# SELECT * FROM cypher("new_graph", "
MATCH (p1:Person)-[:FATHER_OF]->(p1:Person) RETURN p1, p2
") AS (p);

id  | object | name
----+--------+---------
 1  |   p1   | Zeus
 2  |   p2   | Persey
(2 rows)


postgres=#
```

Рисунок 24 — Выполнение запросов в оболочке PostgreSQL

Также был запущен интерфейс командной строки Cypher CLI и были выполнены аналогичные запросы создания графа и выборки информации из него, что отражает рисунок 25.

```
Graph Cypher CLI (version 1.0.0)
> CREATE GRAPH persons (pam: Person{name: "Pam"}, bob: Person{name: "Bob", age: 19}), ((pam: Person)-[p:PARENT]->(bob: Person));

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

> MATCH FROM persons (pam: Person) RETURN pam;

MATCH query
pam:Person {
    name: Pam
}

> MATCH FROM persons (:Person)-[p:PARENT]->(:Person) RETURN p;

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

> EXIT
```

Рисунок 25 — Выполнение запросов в оболочке Cypher CLI

ЗАКЛЮЧЕНИЕ

Результатом выполнения работы является разработанная программная для хранения графовых моделей с раширением для PostgreSQL и интерфейсом командной строки (CLI).

В ходе реализации в соответствии с особенностями предметной области была спроектирована и реализована программная система со следующими возможностями:

— Создание графов с возможностью редактирования узлов и ребер. Был разработан менеджер графовых сущностей, который предоставляет функции для добавления в файловое хранилище новых графов, редактирования их узлов и ребер, удаления отдельных компонентов графа, а также выборки различных сочетаний узлов и ребер при помощи фильтров.

— Получение выборки узлов и ребер с помощью декларативного языка Cypher.. Был реализован синтаксический парсер языка Cypher для его разбора и построения соответствующего AST-дерева и его последующего обхода с целью сбора информации о текущем запросе.

— Предоставление SQL-процедур для работы с хранилищем из PostgreSQL. Создано расширение для сервера PostgreSQL, которое состоит из SQL-функций и типов, представляющих графовые сущности в файлах, а также из низкоуровневого кода на языке C для обращения через ядро СУБД к парсеру запросов и менеджеру графовых сущностей.

Результатом работы является готовая и протестированная система, которая может использоваться для оптимизации хранения большого количества тесно связанных данных и ускорения их обработки.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. What is PostgreSQL. – Текст : электронный // PostgreSQL: The World's Most Advanced Open Source Relational Database : [сайт]. – 2024. – URL : https://www.postgresql.org/about (дата обращения 10.11.2023)

2. Neo4j Graph Database. – Текст : электронный // Neo4j : [сайт]. – 2024. – URL : https://neo4j.com/product/neo4j-graph-database (дата обращения 25.11.2023)

3. Introduction of AGE. – Текст : электронный // Apache AGE : [сайт]. – URL : https://age.apache.org/overview (дата обращения 5.12.2023)

4. Property Graph Model. – Текст : электронный // openCypher : [сайт]. – 2024. – URL : https://github.com/opencypher/openCypher/blob/master/docs/property-graph model.adoc (дата обращения 20.12.2023)

5. Cypher: An Evolving Query Language for Property Graphs / N. Francis [et al.] // SIGMOD’18 Proceedings of the 2018 International Conference on Management of Data. – Houston, United States, June, 2018. – P. 2-3

6. Anthapu, R. Graph Data Processing with Cypher / R. Anthapu. – Packt Publishing, 2022. – 332 p. – ISBN 978-1804611-07-4. – Текст : непосредственный.

7. PostgreSQL Server Programming - Second Edition / U. Dar, H. Krosing, J. Mlodgenski, K. Roybal;. – 2nd ed. – Packt Publishing, 2015. – 322 p. – ISBN 978-1-783980581. – Текст : непосредственный.

8. libpq — C Library . – Текст : электронный // PostgreSQL: The World's Most Advanced Open Source Relational Database : [сайт]. – 2024. – URL : https://www.postgresql.org/docs/current/libpq.html (дата обращения 20.02.2024)

9. Levine, J. flex & bison / J. Levine. – 1st ed. – O'Reilly Media, 2009. – 289 p. – ISBN 978-0596155971. – Текст : непосредственный.

10. Принципы Bison. – Текст : электронный // OpenNET : [сайт]. – 2024. – URL : https://www.opennet.ru/docs/RUS/bison_yacc/bison_4.html (дата обращения 06.03.2024)

11. Martin, K. Mastering CMake / K. Martin. – 3rd ed. – Kitware, Incorporated, 2015. – 706 p. – ISBN 978-1930934313. – Текст : непосредственный.

12. Учимся писать расширения на языке C для PostgreSQL. – Текст : электронный // Записки программиста : [сайт]. – 2024. – URL : https://eax.me/postgresql-extensions (дата обращения 16.03.2024)

13. C-Language Functions. – Текст : электронный // PostgreSQL: The World's Most Advanced Open Source Relational Database : [сайт]. – 2024. – URL : https://www.postgresql.org/docs/16/xfunc-c.html (дата обращения 22.03.2024)

14. Рогов Е. В. PostgreSQL 16 изнутри. / Е. В. Рогов. – 1-е изд. – Москва : ДМК Пресс, 2024. – 664 с. – ISBN 978-5-93700-305-8. – Текст : непосредственный.

15. Компиляторы: принципы, технологии и инструментарий / А. В. Ахо, М. С. Лам, Р. Сети, Д. Д. Ульман. – 2-е изд. – Москва ; Санкт-Петербург : Диалектика, 2020. – 1184 с. – ISBN 978-5-907114-28-9. – Текст : непосредственный.