Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий

Работа допущена к защите
Руководитель ОП
_____ А.В. Щукин
«_____» _____ 2021 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
РАБОТА БАКАЛАВРА

ИССЛЕДОВАНИЕ И СРАВНЕНИЕ ИНСТРУМЕНТОВ ДЛЯ РЕАЛИЗАЦИИ И ПОДДЕРЖКИ ГРАФОВ ЗНАНИЙ

по направлению подготовки 09.03.03 Прикладная информатика
Направленность (профиль) 09.03.03_03 Прикладная информатика в области информационных ресурсов

Выполнил
студент гр. 3530903/70302 В.М. Ковшов

Руководитель
доцент,
к.т.н., О.Н. Тушканова

Консультант
по нормоконтролю В.А. Пархоменко

Санкт-Петербург
2021

САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Институт компьютерных наук и технологий

УТВЕРЖДАЮ
Руководитель ОП
_____ А.В. Щукин
«_____» _____ 2021г.

ЗАДАНИЕ

на выполнение выпускной квалификационной работы

студенту Ковшову Валерию Махайловичу гр. 3530903/70302

1. Тема работы: Исследование и сравнение инструментов для реализации и поддержки графов знаний.
2. Срок сдачи студентом законченной работы: 19.05.2021.
3. Исходные данные по работе: концепция построения графа знаний на основе реляционной модели данных [**designing-a-graph-data-structure-in-a-relational-data** и графовой модели данных [**build-knowledge-graph-nlp-ontologies**]; книга про графовые базы данных [**graph-databases-2nd**].
4. Содержание работы (перечень подлежащих разработке вопросов):
   4.1. Обзор технологических особенностей графов знаний.
   4.2. Разработка графов знаний на основе различных подходов хранения и организации данных.
   4.3. Разработка критериев оценки и подходов к тестированию качества графов знаний.
5. Перечень графического материала (с указанием обязательных чертежей):
   5.1. Архитектура графа знаний.
6. Консультанты по работе:
   6.1. Ассистент ВШИСиСТ, В.А. Пархоменко (нормоконтроль).
7. Дата выдачи задания: 26.01.2021.

Руководитель ВКР _____ О.Н. Тушканова

Задание принял к исполнению 26.01.2021

Студент _____ В.М. Ковшов

РЕФЕРАТ

На 33 с., 9 рисунков, 7 таблиц, 1 приложение

КЛЮЧЕВЫЕ СЛОВА: ГРАФ ЗНАНИЙ, POSTGRESQL, NEO4J, ARANGODB.

Тема выпускной квалификационной работы: «Исследование и сравнение инструментов для реализации и поддержки графов знаний».

В данной работе исследована предметная область, связанная с графами знаний. Определены основные отличия, влияющие на реализацию и для каждой реализации был подобран и описан стек технологий. Для проведения сравнения были разработаны задачи, метрики, а также интеграционная система для работы с различными реализациями. Проведено исследование, по результатам которого каждая реализация получила описание своих преимуществ и недостатков, а также области применения и оценку сложности реализации.

ABSTRACT

33 pages, 9 figures, 7 tables, 1 appendices

KEYWORDS: KNOWLEDGE GRAPH, POSTGRESQL, NEO4J, ARANGODB.

The subject of the graduate qualification work is «Research and comparison of tools for implementing and maintaining knowledge graphs».

In this work, the subject area related to knowledge graphs has been investigated. The main differences affecting the implementation of knowledge graphs and for each implementation have been identified. Tasks, metrics, and an integration system for working with different implementations were developed to make the comparison. A study was conducted, according to the results of which each implementation received a description of its advantages and disadvantages, as well as the areas of application and an estimate of the complexity of the implementation.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Количество информации, доступной сегодня в Интернете, поражает воображение, и оно постоянно расширяется. Например, существует более двух миллиардов веб-сайтов, связанных с всемирной паутиной, и поисковые системы (например, Google, Bing и т.д.) могут просматривать эти ссылки и предоставлять полезную информацию с большой точностью и скоростью. В большинстве этих успешных поисковых систем самым важным знаменателем является использование графов знаний. Не только поисковые системы, но и сайты социальных сетей (например, Facebook и т.д.), сайты электронной коммерции (например, Amazon и т.д.) также используют графы знаний для хранения и извлечения полезной информации.

Целью данной работы является исследование и сравнение инструментов для реализации и поддержки графов знаний. Для достижения данной цели необходимо решить следующие задачи:

- исследовать предметную область;
- определить основные отличия в реализациях графов знаний;
- разработать задачи и метрики для проведения сравнения инструментов для реализации и поддержки графов знаний;
- разработать систему для проведения сравнения инструментов для реализации и поддержки графов знаний;
- произвести исследование на основе разработанных метрик с помощью реализованной системы;
- обработать результаты исследования;
- подвести итоги сравнения инструментов для реализации и поддержки графов знаний.

ГЛАВА 1. ПРЕДСТАВЛЕНИЕ ЗНАНИЙ С ПОМОЩЬЮ ГРАФОВ ЗНАНИЙ

1.1. Представление знаний

Человеку свойственно понимать, рассуждать и интерпретировать знания. Знания, которые накапливаются со временем, позволяют людям выполнять задачи. Аналогичная концепция уже очень давно является целью компьютерных наук и искусственного интеллекта, а способ обработки таких знаний машинами заключается в представлении знаний. Представление знаний направлено на то, чтобы добавить следствие или рассуждения, лежащие в основе рассматриваемого объекта. Этим объектом может быть конкретная деятельность, медицинский диагноз и т.д. Важно отметить, что представление знаний - это не просто хранение данных в базе данных, но и способность учиться и совершенствовать эти знания, подобно тому, как ведет себя человек. В своей работе "What is a Knowledge Representation?"Дэвис, Шроуб и Шоловиц выводят идею о том, что знания для субъекта представлены самим субъектом и предполагаемыми отношениями, которые он имеет с другими субъектами, фактами, обстоятельствами и так далее.

Существует три основных способа представления знаний: логическое, семантическое и фреймовое. Семантические сети являются альтернативой логическому представлению, в том смысле, что они представляют знания в виде графических сетей. Графическая сеть состоит из узлов, представляющих объекты, и дуг, описывающих отношения между этими объектами. Пример на рис.1.1 визуализирует семантические сети.

Рис.1.1. Пример семантической сети

1.2. Графовое представление знаний

Семантические сети были разработаны как техника представления знаний, чтобы проиллюстрировать, как концепции связаны друг с другом. Семантические сети берут верх над логическими представлениями, потому что они более естественны и интуитивны, и обладают большей когнитивной адекватностью по сравнению со своим логическим аналогом. Графы знаний — это форма семантических сетей, обычно ограниченная определенным доменом, и управляемая как граф. Элринджер и Вёсс определяют графы знаний как знания, организованные таким образом, что машина может легко понять и извлечь из них информацию, относящуюся к конкретной области, а также учиться на основе поглощенной информации, поэтому она лучше справляется со связью вещей по мере поступления большего объема информации.

В отличие от обычной базы данных, которая наполняется и, в конечном счете, остается в состоянии покоя, граф знаний должен перепрофилироваться и давать новые знания и умозаключения. Поскольку информация представлена в графовом виде, онтологии легко "расширяться и пересматриваться по мере поступления новых данных". Онтология формально описывает типы, свойства и взаимосвязи между сущностями. Это набор аксиом (можно считать принципами), определяющих знания в определенной области. Таким образом граф знаний является динамическим в том смысле, что сам граф понимает, что связывает сущности, устраняя необходимость программировать каждый новый фрагмент информации вручную.

1.3. Основные функции графа знаний

Графы знаний могут иметь множество различных форм и могут быть представлены во многих вариациях, однако на рисунке 1.2 приведен общий архитектурный обзор того, как работает граф знаний на основе обработки естественного языка.

1.3.1. Источники данных

Для построения графа знаний можно использовать различные источники данных, в том числе структурированные данные в виде реляционных баз дан-

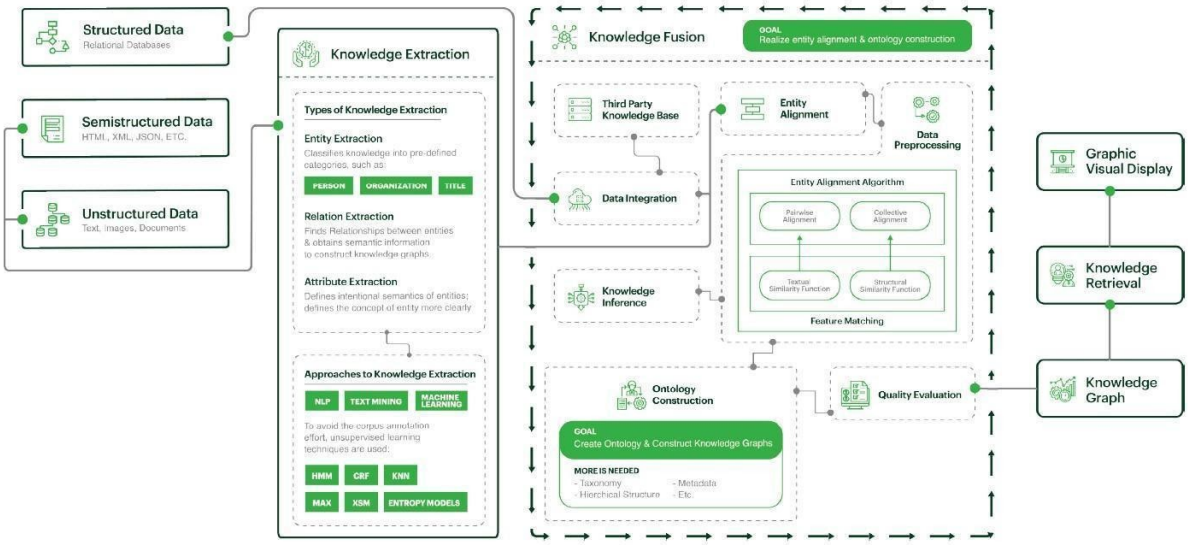

Рис.1.2. Архитектурный обзор того, как работает граф знаний на основе обработки естественного языка

ных; полуструктурированные данные в виде HTML, JSON, XML и так далее, а также неструктурированные данные, такие как свободный текст, изображения и документы. Во многих графах знаний используются данные из Википедии, а в специфических областях, таких как фильмы, используются базы знаний, такие как IMDB.

1.3.2. Извлечение данных

После приема данных начинается процесс извлечения знаний. В ходе этого процесса из вводимых полуструктурированных и неструктурированных данных извлекается информация, которая включает в себя сущности, связи и атрибуты. Это достигается с помощью методов обработки естественного языка, извлечения текста и машинного обучения (как под наблюдением, так и без наблюдения).

Основная идея извлечения сущностей (иначе известного как распознавание сущностей) проста: учитывая некоторый текст, можно ли определить, какие слова идентифицируют сущности определенных категорий? В общем, сущность может представлять человека, объект, место или вещь. Сущности могут относиться к различным классам, которые также могут иметь подклассы. Например, классом сущности может быть "человек". Классы сущностей "футболист "танцор "актер"могут относиться к классу сущностей "человек так как все они являются разновидностью человека. Предложение "Владимир Путин — один из президентов России"указывает на то, что сущность Владимир Путин относится к классу сущностей "президенты России".

После того, как все сущности извлечены, собирается информация об этих сущностях и их атрибутах. Сюда могут входить свойства сущностей, а также отношения между ними. Например, сущность "человек"может быть связана с "местом рождения "полом"и так далее. Иногда атрибут сущности может описывать отношения между двумя сущностями или классами сущностей. Например, сущность "гольф"может быть ассоциирована с атрибутом сущности "является подклассом"с классом сущности "спорт". Обычно, но не всегда, атрибут сущности является глаголом рассматриваемого предложения.

1.3.3. Объединение знаний

Идея слияния знаний состоит в том, чтобы объединить все базы знаний, поступающие из различных источников, чтобы получить всеобъемлющее представление. Ее конкретные цели заключаются в реализации слияния сущностей и онтологического конструирования. Совмещение объектов (или согласование объектов) должно быть связано с определением того, относятся ли "различные объекты к одним и тем же объектам в реальном мире". Стандартизация данных является важным этапом в процессе согласования структур, поскольку она позволяет получить общее представление о данных. Любое несоответствие данных разрешается на этом этапе. Граф знаний Linkedin является хорошим примером важности предварительной обработки и стандартизации данных. Поскольку их сущности являются органическими сущностями, генерируемыми пользователями, многие из них включают "недействительные или неполные атрибуты, бессмысленные имена или устаревшее содержимое". Может быть предпринято множество шагов, включая создание кандидатов на сущности, разделение сущностей на кластеры на основе контекстов, в которых сущности появляются, удаление дублирующих сущностей с помощью таких методов, как word2vec, и использование моделей машинного перевода для приведения всех сущностей к одному и тому же языку.

После того, как все данные интегрированы и согласованы, выполняется объединение записей, относящихся к одной и той же сущности, по парам и группам. Сравнение парного сходства выполняется с использованием различных функций сходства текста, таких как косинусное сходство, а также может интегрировать такие методы глубокого изучения, как word2vec, встраивание seq2seq и так далее. Групповое сопоставление осуществляется с помощью функций структурного сходства, таких как распознавание образов и так далее. Вся эта работа приводит к

созданию онтологии, которая дополняется добавлением таксономии, иерархических структур, метаданных и так далее. для повышения качества графа знаний. Для обеспечения общего качества графа знаний созданную онтологию сравнивают с отраслевыми схемами, такими как schema.org, и если она не соответствует требованиям, то происходит итерация и усовершенствование процесса. Важно подчеркнуть важность итеративного характера этапа слияния знаний, так как именно на этом этапе происходит основная часть моделирования.

1.3.4. Хранение, извлечение и визуальное представление знаний

После создания граф знаний может храниться в реляционной базе данных, базе данных NoSQL или графовой базе данных. Информация о всех данных представлена в RDF (Resource Description Framework) формате, который основывается на триплетах. RDF триплет состоит из субъекта, предиката и объекта, но в то же время к любому триплету может быть добавлена ещё одна ссылка для хранения метаданных.

В случае использования реляционной или NoSQL базы необходимы промежуточные шаги для трансформации данных в RDF граф, в то время как использование графовой базы данных позволяет осуществлять взаимодействие с данными без дополнительной обработки. Для получения данных из RDF графа чаще всего используется SPARQL — традиционный язык запросов для получения крупномасштабных графов знаний.

Визуализация большого количества графов знаний осуществляется через браузерные приложения, и остается одной из наиболее исследуемых тем в этой области.

1.4. Использование графа знаний

Рассмотрим основные направления использования графов знаний.

1.4.1. Доступ к данным по API

Прямой доступ к графу знаний с помощью простого поиска и запросов облегчает разработку приложений, в которых необходимо использовать данные из нескольких источников. Наличие единого представления о структуре данных предприятия позволяет быстро разрабатывать новые приложения. Одним из лучших

способов достижения этой цели является разработка "ориентированного на данные API в котором вызывающая сторона указывает данные, которые ей нужны, вместо того чтобы разрабатывать API для конкретных случаев использования.

Чаще всего в качестве API для графов знаний используется GraphQL. GraphQL — это ориентированный на данные API, в котором модель графовой информации может быть сглажена в JSON по запросу, так что не графовые приложения могут запрашивать графовые данные привычным и удобным способом.

В зависимости от случаев использования может также потребоваться создание API для специфических случаев использования. Чаще всего это касается получения данных для аналитики и машинного обучения.

1.4.2. Потоковые данные

Используя системы потоковой обработки данных наподобие Apache Kafka Streams, можно устранить узкие места на этапах обработки входных данных и поддержания графа в актуальном состоянии в режиме реального времени.

1.4.3. Аналитика

Учитывая, что граф знаний представляет собой полностью связное представление данных о предприятии, он, таким образом, является полномасштабным источником данных для системы BI/MI (Business Intelligence / Management Information). Работа по сбору и связыванию данных абстрагируется на графе, а архитектура BI/MI может быть упрощена для потребления информации из графа. Таким образом, граф знаний играет центральную роль в создании отчетов, выполнении аналитических запросов и предоставлении визуализаций. Он также может быть источником информации для современных поисковых решений на основе AI, которые используют тысячи параметров одновременно, чтобы показать "горячие точки"и отклонения, которые имеют значение.

1.4.4. Машинное обучение

Большая часть задач по сбору, связыванию и очистке данных выполняется самим графом, что позволяет упростить работу специалистов по работе с данными. Также граф знаний может использоваться для обновления моделей и тренировочных наборов данных для машинного обучения. Если учесть, что ключевой целью машинного обучения в бизнесе является прогнозирование, то граф знаний, который

охватывает все бизнес-подразделения, является ключевым источником входных данных для построения моделей.

Одной из общих задач машинного обучения, применяемой к графам является классификация узлов — обучение модели для определения того, к какому классу принадлежит узел. Модели классификации узлов используются для предсказания несуществующего свойства узла на основе других свойств узла. Несуществующее свойство узла представляет собой класс и называется целевым свойством. Указанные свойства узла используются в качестве входных параметров. Модель классификации узлов не опирается на информацию о взаимосвязях, однако алгоритм встраивания узлов может встраивать соседи узлов в качестве свойства, для передачи этой информации в модель Модели обучаются по частям входного графа и оцениваются по определенным метрикам. Разделение графа на обучающий и тестовый графы выполняется внутренним алгоритмом, а тестовый граф используется для оценки производительности модели.

Другой задачей машинного обучения, применяемой к графам является прогнозирование связей — построение модели для предсказания отсутствующих связей в наборе данных или связей, которые, вероятно, будут формироваться в будущем. Обычно работа с прогнозированием связи подразделяется на следующие этапы:

- создание графов для обучения и тестирования;
- обучение и оценивание кандидатов-моделей;
- применение модели для прогнозирования.

Еще одной задачей машинного обучения, применяемой к графам является поиск сообществ. При исследовании сложных сетей считается, что сеть имеет структуру сообщества, если узлы сети можно легко сгруппировать в наборы узлов таким образом, чтобы каждый набор узлов был плотно связан внутренне. В конкретном случае поиска не пересекающихся сообществ это означает, что сеть естественным образом разделяется на группы узлов с плотными внутренними связями и более узкие связи между группами. Но накладывающиеся друг на друга сообщества также допустимы. Более общее определение основано на принципе, что пары узлов с большей вероятностью будут связаны, если они оба являются членами одного и того же сообщества (сообществ), и с меньшей вероятностью будут связаны, если они не разделяют сообщества.

Поиск сообществ в произвольной сети может оказаться сложной вычислительной задачей. Количество сообществ, если таковые имеются, внутри сети, как

правило, неизвестно, и сообщества часто имеют неравные размеры и/или плотность. Однако, несмотря на эти трудности, было разработано и использовано несколько методов поиска сообществ с разной степенью успеха:

- метод наименьших разрезов;
- иерархическая кластеризация;
- алгоритм Гирван — Ньюмена;
- максимизация модульности;
- статический вывод.

1.5. Выводы

Графы знаний являются мощным инструментом, позволяющим полностью использовать графовую природу данных в самых различных направлениях — начиная от мониторинга взаимодействия сотрудников внутри небольшого предприятия, заканчивая улучшением результатов поиска крупнейшей поисковой системы Google.

На данный момент не существует единственного верного подхода для реализации графов знаний. Большинство различий заключается в выборе базы данных, в которой будет осуществляться хранение графовых данных. Так, для работы таких основных функций как: получение данных из внешних источников, извлечение и объединение данных - критическую роль играет поддержка хранения графовых структур на стороне базы данных и возможности встроенного языка запросов. А для таких направлений использования графа знаний как: доступ к данным по API и подготовка данных для аналитики и машинного обучения - критическую роль играет скорость сбора графовых данных на стороне базы и ориентированность встроенного языка на выполнение запросов разной степени сложности. Именно поэтому перед построением графа знаний необходимо произвести исследования различных типов баз данных по следующим направлениям:

- исследование способов хранения графовых данных — в рамках которого будет определено насколько база данных ориентирована на хранение графовых данных, какие дополнительные действия необходимо произвести в случае отсутствия поддержки по умолчанию и какой размер памяти будет занимать загруженный граф;
- исследование функциональности и производительности языка запросов при вставке данных — в рамках которого будет определено насколько язык

запросов ориентирован на вставку одиночных и множественных данных, а также время необходимое на совершение данных операций;

– исследование функциональности и производительности языка запросов при извлечении данных — в рамках которого будет определено насколько язык запросов ориентирован на извлечение данных по атрибутам и на основе графовых алгоритмов, а также время необходимое на совершение данных операций.

ГЛАВА 2. ВЫБОР ТЕХНОЛОГИЙ ДЛЯ РЕАЛИЗАЦИИ ГРАФА ЗНАНИЙ

2.1. Подходы к хранению графовых данных

Самой известной на сегодняшний день моделью данных, вероятно, является реляционная модель основанная Эдгаром Коддом в 1970 году. Данные организованы в отношения (таблицы), где каждое отношение представляет собой неупорядоченную коллекцию кортежей (строк). Целью реляционной модели было скрыть эту реализацию за чистым интерфейсом.

Поскольку данные хранятся в реляционных таблицах, между объектами в коде приложения и моделью базы данных требуется неудобный слой трансляции: объектно-реляционное отображение (ORM). Это несоответствие между моделями называется импедансным несовпадением.

В середине 2000-х годов из-за следующих потребностей появилась новая нереляционная модель:

– большая масштабируемость, так как реляционные базы данных не предназначены для горизонтального масштабирования;
– более гибкие схемы. Схемы являются полезным механизмом для документирования и принудительного применения этой структуры, однако подход "схема на чтение"(структура данных неявна и интерпретируется только при чтении данных) является выгодным, если элементы коллекции по каким-то причинам не имеют одинаковой структуры; произвольные ключи и значения могут быть добавлены в документ;
– лучшая локализация, по сравнению с многотабличной схемой; вся соответствующая информация находится в одном месте, и одного запроса достаточно для её извлечения.

На данный момент существует несколько типов NoSQL баз данных:

– ключ значение;
– семейство столбцов;
– документоориентированная база данных;
– графовая база данных.

Большинство баз данных организованы на основе одной модели данных, однако существуют системы предназначенные для поддержки множества моделей данных на основе единого интегрированного бэкэнда, такие базы данных называются мультимодельными.

Для проведение сравнение эффективности хранения и работы с графами знаний будут использованы базы данных различного типа, среди которых присутствует реляционная, графовая и мультимодальная база данных.

2.1.1. PostgreSQL

PostgreSQL — это объектно-реляционная система баз данных с открытым исходным кодом, использующая и расширяющая язык SQL в сочетании со многими функциями, которые позволяют безопасно хранить и масштабировать самые сложные данные. PostgreSQL заслужил прочную репутацию за свою проверенную архитектуру, надежность, целостность данных, надежный набор функций, расширяемость и преданность сообществу с открытым исходным кодом, стоящему за этим программным обеспечением, которое постоянно поставляет высокопроизводительные и инновационные решения.

PostgreSQL поставляется с большим количеством функций, которые помогают разработчикам создавать приложения, администраторам - защищать целостность данных и создавать отказоустойчивые среды, а также управлять данными, независимо от их размера и размера. Помимо того, что PostgreSQL является бесплатным приложением с открытым исходным кодом, он обладает широкими возможностями расширения. PostgreSQL старается соответствовать стандарту SQL в тех случаях, когда такое соответствие не противоречит традиционным возможностям или может привести к некачественным архитектурным решениям.

2.1.2. Neo4j

Neo4j — это NoSQL база данных с открытым исходным кодом, нативная база данных графов, которая обеспечивает ACID — совместимый бэкэнд транзакций.

Первоначальная разработка началась в 2003 году, но с 2007 года она стала общедоступной. Исходный код, написанный на Java и Scala, доступен для бесплатного скачивания на GitHub или в виде удобного для пользователя приложения.

Neo4j называется нативной базой данных графов, поскольку она эффективно реализует модель графов свойств вплоть до уровня хранения. В отличие от обработки графов или библиотек в памяти, Neo4j также предоставляет полные характеристики базы данных, включая соответствие транзакциям ACID, поддержку кластера и аварийное переключение во время выполнения, что делает его пригодным для использования графов для данных в производственных сценариях.

Некоторые из перечисленных ниже особенностей делают Neo4j очень популярным среди разработчиков, архитекторов и DBA:

- Cypher, язык декларативных запросов, похожий на SQL, но оптимизированный для работы с графами;
- постоянное время прохождения на больших графах как в глубину, так и в ширину, благодаря эффективному представлению узлов и связей;
- гибкая схема графов свойств, которая может адаптироваться с течением времени, что позволяет материализовать и добавлять новые отношения позже, чтобы сократить и ускорить данные в домене, когда бизнес нуждается в изменении;
- драйверы для популярных языков программирования, включая Java, JavaScript, .NET, Python и многие другие.

2.1.3. ArangoDB

ArangoDB — это нативная мультимодельная база данных с открытым исходным кодом, предназначенная для хранения данных в виде пар ключ-значение, графов и JSON документов, доступ к которым осуществляется с помощью одного декларативного языка запросов — AQL. С помощью ArangoDB можно создавать высокопроизводительные приложения и масштабировать их по горизонтали, используя все три модели данных в полном объеме. Кроме этого, ArangoDB делает акцент на высокой согласованности данных, упрощенное масштабирование и низкую совокупную стоимость владения.

2.2. Обработка и извлечение данных

2.2.1. Задачи для проведения сравнения извлечения данных

К основным типам извлечения данных в графах знаний относятся извлечение данных на основе атрибутов и классических графовых алгоритмов, поэтому для сравнения функциональности и производительности языка запросов будет произведено сравнение извлечение данных на основе:

- атрибутов сущности;
- атрибутов отношения;
- алгоритма поиска кратчайшего пути;
- алгоритма поиска ближайших соседей.

2.2.2. Встроенные инструменты извлечения данных

Neo4j предлагает встроенную в базу данных библиотеку “Graph Data Science Library” (GDSL), предоставляющую эффективно реализованные параллельные версии распространенных графовых алгоритмов, встраиваемых в виде процедур языка запросов Cypher. Библиотека содержит реализации следующих типов алгоритмов:

- поиск пути — эти алгоритмы помогают найти кратчайший путь или оценить доступность и качество маршрутов;
- центральность — эти алгоритмы определяют важность отдельных узлов в сети;
- обнаружение сообщества — эти алгоритмы оценивают, как группа группируется или разделяется, а также как она усиливается или разбивается на части;
- сходство — эти алгоритмы помогают вычислить сходство узлов;
- link prediction —эти алгоритмы определяют близость пар узлов;
- node embeddings — эти алгоритмы вычисляют векторное представление узлов в графе;
- классификация узлов — данный алгоритм использует машинное обучение для прогнозирования классификации узлов;

ArangoDB предоставляет возможности языка AQL, в котором присутствуют операторы для решения задач всевозможных обходов графов, поиска кратчай-

шего пути, а также операторы для поиска различного типа соседей (ближайшие, ближайшие k, все соседи).

2.2.3. Использование сторонних библиотек для извлечения данных

Так как среди баз данных представленных для сравнения присутствует база данных PostgreSQL, которая не предоставляет встроенную возможность работы с графами и поддержку графовых алгоритмов, существует необходимость создания дополнительного сервисного слоя, который будет этим заниматься. В качестве языка для сервиса был выбран язык Java из-за наличия большого количества библиотек для работы с графами и драйверов для подключениями к базам данных.

На сайте www.baeldung.com, который содержит статьи и руководства по экосистеме Java присутствует статья, посвященная работе с графами, в которой особое внимание уделается следующим библиотекам:

- JGraphT — одна из самых популярных библиотек на Java для графовой структуры данных. Она позволяет создавать всевозможные виды графов и предлагает множество алгоритмов работы с графовыми структурами данных;
- Google Guava — это универсальный набор библиотек которые предлагают ряд функций, включая структуру данных графа и его базовые алгоритмы.
- Apache Commons — это проект Apache, который включает в себя Commons Graph, предлагая инструментарий для создания и управления структурой данных графа. Также предоставляет общие графовые алгоритмы для работы со структурой данных.
- Sourceforge Java Universal Network/Graph (JUNG) — это Java-фреймворк, который предоставляет расширяемый язык для моделирования, анализа и визуализации любых данных, которые могут быть представлены в виде графа. JUNG поддерживает ряд алгоритмов, которые включают такие процедуры, как кластеризация, декомпозиция и оптимизация.

Большинство из представленных библиотек подходят для решения поставленной задачи, но в своей работе “JGraphT - A Java library for graph data structures and algorithms” авторы Димитриос Михаил и Йорис Кинабле подробно рассматривают реализацию структур данных в библиотеке JGraphT и проводят сравнение скорости работы графовых алгоритмов этой библиотеки с библиотекой Jung, а также библиотекми NetworkX, BGL, igraph реализованных на python и c++. По

результатам их исследования на задаче поиска кратчайшего пути, по результатам представленным на рис.2.10 видно, что JGraphT справляется с большими графами существенно лучше, чем Jung, поэтому именно библиотека JGraphT и будет использована для работы с графами на сервисном уровне.

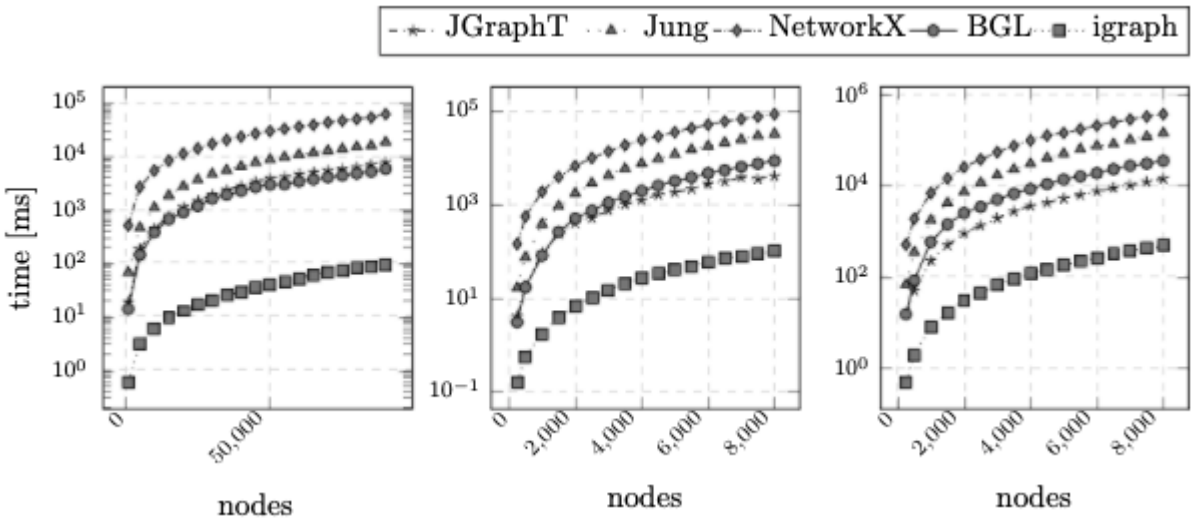

Рис.2.1. Сравнение скорости работы алгоритма поиска кратчайшего пути для разных библиотек

ГЛАВА 3. РЕАЛИЗАЦИЯ ПРОЕКТА ДЛЯ РАБОТЫ С БАЗАМИ ДАННЫХ

Для проведения исследований необходимо описать интерфейс для работы с базами данных независимо от их типа и создать реализации для каждой участвующей базы данных, а именно PostgreSQL, Neo4j и ArangoDB.

3.1. Описание интерфейса для работы с базами данных

Для описания абстрактных методов без конкретной реализации в языке Java существуют интерфейсы. Интерфейс для работы с базой данных будет называться “Database” и он будет содержать следующие методы, необходимые для исследования:

- init — метод для инициализации базы данных в случае необходимости подготовительных действий перед началом работы;
- clear — метод для очистки базы данных перед началом работы с новым графом;
- addNode — метод для добавления сущности (вершины графа);

- addEdge — метод для добавления отношения (ребро графа);
- addGraph — метод для одновременного добавления сущностей и отношений, если такая возможность присутствует в базе данных;
- getByNodeAttribute — метод для извлечения данных на основе атрибутов сущности (вершины графа);
- getByEdgeAttribute — метод для извлечения данных на основе атрибутов отношения (ребра графа);
- getShortestPath — метод для извлечения кратчайшего пути между двумя вершинами;
- getNearestNeighbors — метод для извлечения ближайших соседей к вершине на указанном уровне.

Дополнительно интерфейс “Database” расширяет интерфейс “AutoCloseable” для унаследования метода close в котором принято закрывать активные подключение к базам данных для правильной работы с ресурсами языка Java.

3.2. Реализация интерфейса для PostgreSQL

Для создания реляционной базы данных PostgreSQL на локальной машине используется Docker-образ “postgres” с переопределенным портом для подключения, выделенным одним процессором и настройками пользователя и базы данных.

```
postgres_db:
  container_name: postgres_db
  image: postgres
  ports:
    - 6666:5432
  environment:
    POSTGRES_DB: postgres
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    deploy:
      resources:
        limits:
          cpus: '1'
```

Рис.3.1. Настройки для запуска PostgreSQL в Docker-контейнере

Для подключения к созданной базе данных PostgreSQL в методе init класса “PostgresImpl” используются указанные в Docker-файле пользователь и база данных. При подключении в переменные класса сохраняются объекты подключение и состояния подключения к базе данных, для дальнейшего использования их в методах класса.

3.3. Реализация интерфейса для Neo4j

Для создания графовой базы данных Neo4j на локальной машине используется Docker-образ “neo4j” с предопределенными портами для подключения, выделенным одним процессором и настройками пользователя.

```
neo4j_db:
  container_name: neo4j_db
  image: neo4j
  ports:
    - 7474:7474
    - 7687:7687
  environment:
    NEO4J_AUTH: 'neo4j/QWErty123'
    deploy:
      resources:
        limits:
          cpus: '1'
```

Рис.3.2. Настройки для запуска Neo4j в Docker-контейнере

Для подключения к созданной базе данных Neo4j в методе init класса “Neo4jImpl” используется указанный в Docker-файле пользователь. При подключении в переменные класса сохраняются объекты драйвера и сессии подключения к базе данных, для дальнейшего использования их в методах класса.

3.4. Реализация интерфейса для ArangoDB

Для создания мультимодельной базы данных ArangoDB на локальной машине используется Docker-образ “arangodb” с переопределенным портом для подключения, выделенным одним процессором и отключенной аутентификацией.

```
arango_db:
  container_name: arangodb_db
  image: arangodb
  ports:
    - 8529:8529
  environment:
    ARANGO_NO_AUTH: 1
    deploy:
      resources:
        limits:
          cpus: '1'
```

Рис.3.3. Настройки для запуска ArangoDB в Docker-контейнере

Для подключения к созданной базе данных ArangoDB в методе init класса “ArangoDBImpl” в переменные класса сохраняются объекты базы данных и графа, для дальнейшего использования их в методах класса.

3.5. Создание интерфейса для логирования времени выполнения операций

Для измерения времени выполнения каждой операции нужна ещё одна реализация интерфейса “Database”. Эта реализация не будет привязана к конкретной базе данных, она будет реализовывать шаблон делегирования. Это значит, что класс “LoggableDatabaseImpl” будет в конструкторе получать и сохранять реализацию интерфейса “Database” и в каждом методе передавать ей управление замеряя время до и после выполнения операции.

3.6. Итоговая структура проекта для работы с базами данных

Для проведения сравнения качества работы различных типов баз данных на классических графовых задачах был описан интерфейс “Database” для работы с базами данных независимо от их типа и созданы реализации для каждой базы данных, а именно “PostgresImpl”, “Neo4jImpl” и “ArangoDBImpl”, а так же создана реализация “ LoggableDatabaseImpl” для замера времени выполнения каждого метода, описанного в интерфейсе “Database”.

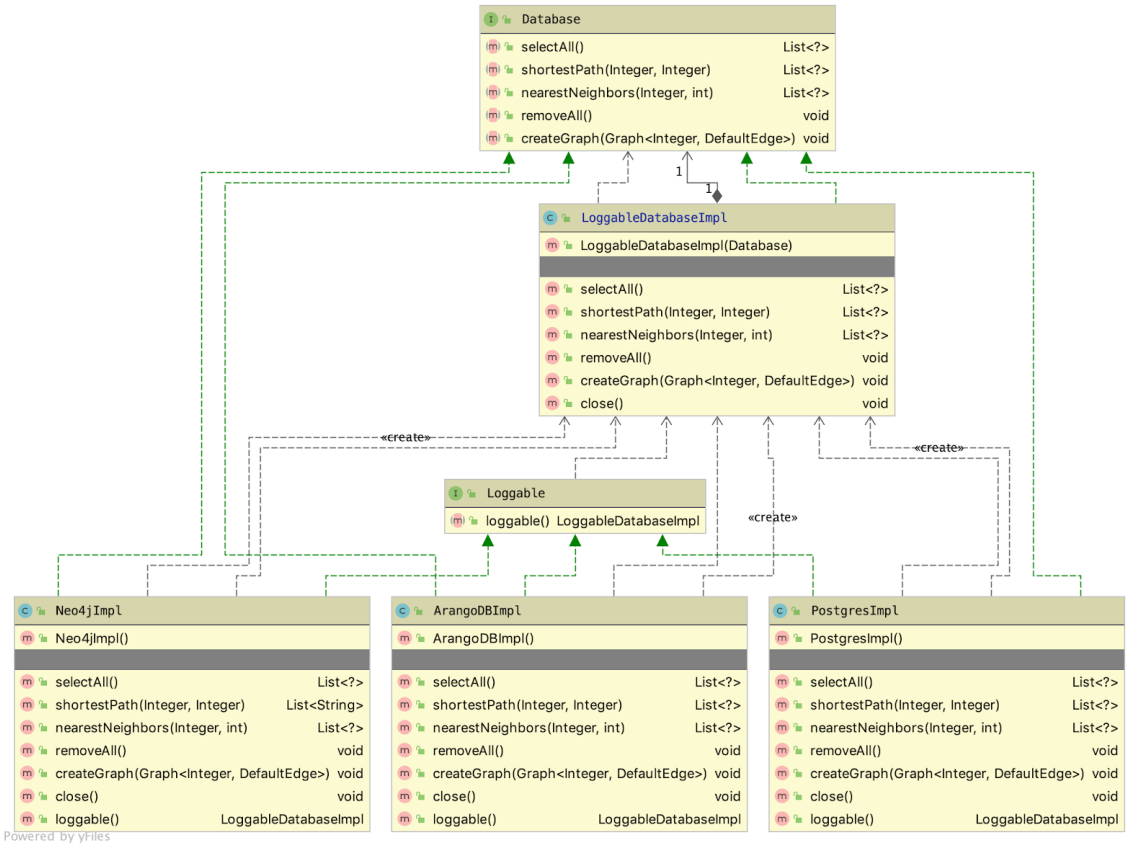

Рис.3.4. Структура проекта для работы с базами данных

ГЛАВА 4. ИССЛЕДОВАНИЕ РАБОТЫ БАЗ ДАННЫХ

4.1. Подготовка тестовых данных

Для тестирования будут использованы три различных графа, взятые из коллекции больших сетевых данных Стэнфорда — https://snap.stanford.edu/data/:

- CollegeMsg temporal network (далее COL) — ориентированный граф, содержащий 1899 вершин и 59835 ребер из которых 20296 являются уникальными. Этот набор данных состоит из личных сообщений, отправленных пользователями в социальной сети. Узлы представляют собой пользователей, а рёбра — факт отправки личного сообщения от одного пользователя другому. Особенностью графа является большое количество кратных рёбер, а именно 39539;
- Gnutella peer-to-peer network, 5 Aug 2002 (далее GNU) – ориентированный граф, содержащий 8846 вершин и 31839 ребер. Этот набор данных представляет собой снимок одноранговой файлообменной сети Gnutella за 9 августа 2002 года. Узлы представляют собой хосты в топологии сети Gnutella, а ребра - соединения между хостами Gnutella. Особенностью графа является низкий средний коэффициент кластеризации – 0.0072;
- Social circles: Facebook (далее FB) — ориентированный граф, содержащий 3483 вершины и 50865 ребер. Этот набор данных состоит из списка отправленных заявок на добавления в друзья на Facebook. Узлы представляют собой пользователей, а рёбра — факт отправки запроса на добавления в друзья. Особенностью графа является высокий средний коэффициент кластеризации – 0.6055.

4.2. Исследование способов хранения графовых данных

Для хранения графовых данных в PostgreSQL потребуются три таблицы:

- entity — описывает сущности, которые могут являться объектом или субъектом отношения. Хранит уникальный идентификатор сущности, её название и метаданные;
- predicate — описывает тип отношения. Хранит уникальный идентификатор отношения и его название;

– triple — содержит информацию об объекте, субъекте, типе отношения между ними и метаданные этого отношения.

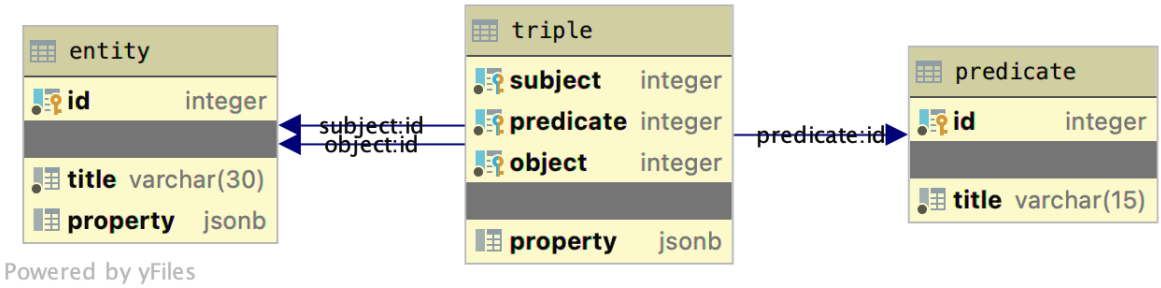

Рис.4.1. Диаграмма структуры базы данных PostgreSQL для хранения графа

Для хранения графовых данных в Neo4j не требуется никаких дополнительных настроек.

Для хранения графовых данных в ArangoDB необходимо создать коллекцию типа “Document” для хранения сущностей и коллекцию типа “Edges” для хранения отношений, после чего необходимо создать граф, с настройками, как указано на рис.4.2.

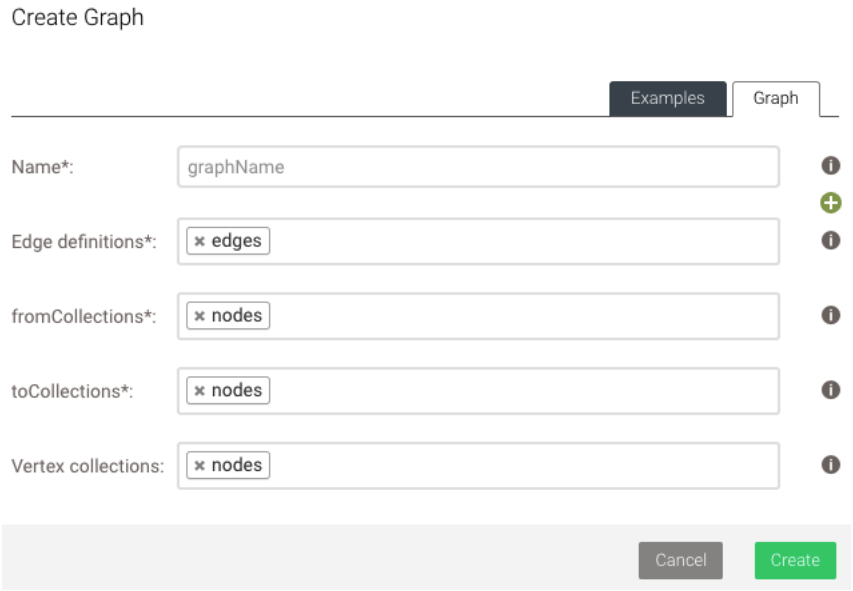

Рис.4.2. Создание графа в ArangoDB на основе двух коллекций

От структуры хранения графовых данных напрямую зависит количество памяти, которое понадобится базе данных для хранения загруженного графа. Результаты замеров использованной памяти для каждого из тестовых графов приведены в таблице 4.1.

Таблица 4.1

Количество памяти, необходимое базе данных для хранения графа

База данных	Граф	Использованная память
PostgreSQL	COL	10 Мб
PostgreSQL	GNU	11 Мб
PostgreSQL	FB	12 Мб
Neo4j	COL	239 Мб
Neo4j	GNU	396 Мб
Neo4j	FB	427 Мб
ArangoDB	COL	44 Мб
ArangoDB	GNU	66 Мб
ArangoDB	FB	53 Мб

Из таблицы 4.1 можно сделать вывод, что после занесении информации о графе в базу данных Neo4j, её размер превышает размер базы данных PostgreSQL примерно в 30 раз и размер базы данных ArangoDB примерно в 6 раз.

4.3. Исследование функциональности и производительности языка запросов при вставке данных

Результаты замеров времени поочередной вставки записей об отношениях в базы PostgreSQL и ArangoDB приведены в таблице 4.2.

Таблица 4.2

Время поочередной вставки записей об отношениях в базы данных

База данных	Граф	Среднее время	Стандартное отклонение	95-ый процентиль	98-ой процентиль	Суммарное время
PostgreSQL	COL	2.44 мс	0.69 мс	3.07 мс	3.53 мс	50 сек
PostgreSQL	GNU	2.65 мс	0.83 мс	3.56 мс	4.07 мс	85 сек
PostgreSQL	FB	2.72 мс	1.07 мс	3.77 мс	4.46 мс	139 сек
ArangoDB	COL	2.46 мс	0.98 мс	3.69 мс	4.73 мс	50 сек
ArangoDB	GNU	2.47 мс	0.91 мс	3.61 мс	4.69 мс	78 сек
ArangoDB	FB	2.40 мс	0.86 мс	3.73 мс	4.61 мс	122 сек

По таблице 4.2 видно, что операции поочередной вставки записей об отношениях в базы PostgreSQL и ArangoDB незначительно отличаются друг от друга по времени выполнения. База данных Neo4j в отличие от других не оптимизирована под единичную вставку, но оптимизирована под множественную вставку данных.

Результаты замеров суммарного времени вставки записей об отношениях в базу данных Neo4j приведены в таблице 4.3.

Таблица 4.3

Время множественной вставки записей об отношениях в базу данных Neo4j

База данных	Граф	Суммарное время
Neo4j	COL	30.46 сек
Neo4j	GNU	46.15 сек
Neo4j	FB	58.35 сек

Сравнивая результаты затраченного времени на вставку всех записей об отношениях из таблицы 4.2 и таблицы 4.3 можно сделать вывод, что база данных Neo4j примерно в 2 раза быстрее справляется с поставленной задачей за счёт оптимизации множественной вставки данных.

4.4. Исследование функциональности и производительности языка запросов при извлечении данных

Структура хранения графовой информации подразумевает возможность наличие у сущности неограниченного количества атрибутов. Например для сущности человек, могут существовать атрибуты: имя, дата рождения, пол, гендер и так далее. Результаты замеров времени при извлечении данных о сущностях на основе атрибутов представлены в таблице 4.4.

Таблица 4.4

Время извлечение данных на основе атрибутов сущности

База данных	Граф	Среднее время	Стандартное отклонение	95-ый процентиль	98-ой процентиль
PostgreSQL	COL	16,84 мс	8,91 мс	23,37 мс	29,67 мс
PostgreSQL	GNU	26,08 мс	10,44 мс	36,98 мс	42,78 мс
PostgreSQL	FB	36,21 мс	10,98 мс	50,82 мс	54,56 мс
Neo4j	COL	29.23 мс	15.12 мс	51.71 мс	72.10 мс
Neo4j	GNU	71.53 мс	29.36 мс	111.25 мс	135.28 мс
Neo4j	FB	37.12 мс	24.89 мс	57.86 мс	86.05 мс
ArangoDB	FB	13.54 мс	4.32 мс	22.14 мс	24.47 мс
ArangoDB	FB	56.41 мс	9.97 мс	67.73 мс	81.72 мс
ArangoDB	FB	23.41 мс	4.93 мс	31.38 мс	38.22 мс

Также структура графовой информации подразумевает возможность наличия у отношения неограниченного количества атрибутов. Например для отношения “сыграл в фильме” могут существовать атрибуты: роль, количество времени в кадре и так далее. Результаты замеров времени при извлечении данных об отношениях на основе атрибутов представлены в таблице 4.5.

Таблица 4.5

Время извлечение данных на основе атрибутов отношения

База данных	Граф	Среднее время	Стандартное отклонение	95-ый процентиль	98-ой процентиль
PostgreSQL	COL	70.78 мс	38.27 мс	109.79 мс	126.06 мс
PostgreSQL	GNU	107.57 мс	51.01 мс	156.08 мс	187.07 мс
PostgreSQL	FB	153.91 мс	47.74 мс	217.96 мс	251.98 мс
Neo4j	COL	148.87 мс	58.64 мс	222.54 мс	253.59 мс
Neo4j	GNU	207.4 мс	39.38 мс	292.25 мс	326.44 мс
Neo4j	FB	313.45 мс	72.98 мс	417.73 мс	490.48 мс
ArangoDB	FB	153.81 мс	9.32 мс	167.09 мс	186.26 мс
ArangoDB	FB	238.35 мс	12 мс	255.77 мс	276.04 мс
ArangoDB	FB	374.64 мс	17.5 мс	399.9 мс	425.61 мс

Из таблиц 4.4 и 4.5 можно сделать вывод, что извлечение данных на основе атрибутов быстрее всего происходит происходит из базы PostgreSQL, ArangoDB работает в среднем в 1.5 раз дольше, а Neo4j в отстаёт от PostgreSQL в среднем в 2 раза. Такие различия между реляционной и нереляционными базами данных обоснованы способом хранения информации. С помощью правильной декомпозиции графовой структуры данных на три таблицы удалось обеспечить высокую скорость работы с любыми типами атрибутов.

Помимо извлечения данных на основе атрибутов, не менее частой задачей является извлечение данных на основе графовых алгоритмов. Чаще всего применяются следующие алгоритмы: поиск кратчайшего пути, поиск ближайших соседей, ранжирование Пейджа, определение степени влиятельности. Результаты замеров времени при извлечении данных на основе алгоритма поиска кратчайшего пути представлены в таблице 4.6.

По таблице 4.6 видно, что быстрее всего с задачей поиска кратчайшего пути справляется база данных ArangoDB, примерно в 3.5 раз дольше этот же алгоритм работает в базе данных Neo4j и примерно в 30 раз дольше с аналогичной задачей справляется сервис, основанный на базе данных PostgreSQL. Такой результат

Таблица 4.6

Время извлечение данных на основе алгоритма поиска кратчайшего пути

База данных	Граф	Среднее время	Стандартное отклонение	95-ый процентиль	98-ой процентиль
PostgreSQL	COL	85.23 мс	21.15 мс	119.53 мс	124.49 мс
PostgreSQL	GNU	130.41 мс	25.77 мс	170.42 мс	223.52 мс
PostgreSQL	FB	215.04 мс	51.87 мс	303.44 мс	319.79 мс
Neo4j	COL	8.99 мс	4.95 мс	11.91 мс	14.38 мс
Neo4j	GNU	14.18 мс	6.21 мс	20.56 мс	26.99 мс
Neo4j	FB	12.01 мс	8.18 мс	17.14 мс	19.58 мс
ArangoDB	FB	2.69 мс	0.62 мс	3.53 мс	3.92 мс
ArangoDB	FB	2.99 мс	0.72 мс	4.16 мс	4.66 мс
ArangoDB	FB	3.72 мс	1.41 мс	6.18 мс	7.16 мс

вызван отсутствием поддержки графовых алгоритмов в PostgreSQL, вследствии чего задачей занимается сервисный слой, которому необходимо предварительно извлечь все данные для построения графа.

Результаты замеров времени при извлечении данных на основе алгоритма поиска ближайших соседей представлены в таблице 4.7.

Таблица 4.7

Время извлечение данных на основе алгоритма поиска ближайших соседей

База данных	Граф	Среднее время	Стандартное отклонение	95-ый процентиль	98-ой процентиль
PostgreSQL	COL	144.25 сек	63.46 сек	193.83 сек	217.32 сек
PostgreSQL	GNU	-	-	-	-
PostgreSQL	FB	-	-	-	-
Neo4j	COL	6.33 сек	5.98 сек	12.15 сек	16.49 сек
Neo4j	GNU	17.82 сек	7.11 сек	22.51 сек	27.98 сек
Neo4j	FB	15.79 сек	7.83 сек	23.85 сек	26.54 сек
ArangoDB	FB	0.88 сек	0.25 сек	1.29 сек	1.43 сек
ArangoDB	FB	1.51 сек	0.61 сек	2.41 сек	2.95 сек
ArangoDB	FB	1.35 сек	0.89 сек	2.44 сек	3.56 сек

Из таблицы 4.7 видно, что лучше всего с задачей поиска ближайших соседей справляется база данных ArangoDB, примерно в 10 раз больше времени требуется Neo4j для решения аналогичной задачи, а PostgreSQL даже при наличии сервисного слоя работает в сотни раз дольше на самом маленьком из тестовых графов, а на

двух больших графах и вовсе не смогла справиться с поставленной задачей ввиду нехватки память для вычислений, выделенной в рамках исследования.

4.5. Итоги исследований

После правильной адаптации для хранения графовых структур реляционная база данных PostgreSQL использует намного меньше дискового пространства и способна обеспечить конкурентоспособную вставку графовых данных на уровне нереляционных баз данных. Однако проблема отсутствия поддержки графовых алгоритмов обработки для извлечения данных не способна быть эффективно решена посредством добавления сервисного слоя. Это связано с тем, что для правильной работы сервисного слоя необходимо получить из базы всю имеющуюся информацию и обработать её повторно для построения графовой структуры на сервисе. Такое решение способно поддерживать небольшие графовые структуры, но оказывается совершенно неработоспособным при увеличении количества данных ввиду ограниченности оперативной памяти сервиса.

Графовая база данных Neo4j способна удовлетворить любые потребности в хранении и обработке графовых структур за счёт своей реализации и языка Cypher, но требует ощутимо больше дисковой памяти для хранения информации. Наличие поддержки и оптимизации для множественной вставки графовых данных выделяет эту базу данных на фоне конкурентов, но оптимизации встроенных механизмов отбора и извлечения данных, а также графовых алгоритмов зачастую недостаточно для уверенной победы над конкурентами в вопросах времени выполнения. При этом база данных Neo4j способна поддерживать графовые структуры данных очень большого объёма и предоставляет исчерпывающее количество инструментов для продуктивного взаимодействия с графовыми данными, среди которых можно выделить инструменты графо-нативного машинного обучения и мощное браузерное приложения для администрирования и управления данными.

Мультимодельная база данных ArangoDB при решении задач хранения графовых структур использует лучшее от документоориентированного и графового подхода к хранению данных. За счёт документоориентированного подхода, графовые данные хранимые в ArangoDB занимают незначительно больше места, чем в реляционной базе данных, а извлечение данных на основе атрибутов происходит быстро. За счёт графового подхода к обработке информации ArangoDB предоставляет мощные и быстрые инструменты языка AQL для извлечения данных на

основе графовых алгоритмов. Но в отличие от Neo4j эта база данных не имеет развитого инструментария для извлечения всех возможностей графовых данных, что делает её менее привлекательной для проектов, нуждающихся в оптимизации для большого количества данных и наличия продвинутых графовых алгоритмов и встроенного машинного обучения.

ЗАКЛЮЧЕНИЕ

В ходе работы была исследована предметная область, связанная с графами знаний. Определены основные отличия, влияющие на реализацию и для каждой реализации был подобран и описан стек технологий. Для проведения сравнения были разработаны задачи, метрики, а также интеграционная система для работы с различными реализациями. Проведено исследование, по результатам которого каждая реализация получила описание своих преимуществ и недостатков, а также области применения и оценку сложности реализации:

- PostgreSQL — реляционная база данных подходит для работы с небольшими объемами графовых данных. К преимуществам этого подхода относятся: низкая потребность дисковой памяти и скорость работы на вставку и простое извлечение данных. К недостаткам этого подхода относятся: отсутствие поддержки графовых алгоритмов, необходимость самостоятельной реализации архитектуры и отсутствие инструментов для извлечения выгоды из графовой природы данных;
- Neo4j — графовая база данных подходит для максимально продуктивной работы для работы с графовыми данными любой степени сложности. К преимуществам этого подхода относятся: наличие всех необходимых инструментов для работы и извлечения выгоды из графовой природы данных. К недостаткам этого подхода относятся: высокая потребность дисковой памяти для хранения данных и скорость выполнения операция на вставку и извлечение данных;
- ArangoDB — мультимодельная база данных подходит для работы с графовыми данными данными любой степени сложности. К преимуществам этого подхода относятся: низкая потребность дисковой памяти, скорость работы на вставку и извлечение данных любой сложности. К недостаткам этого подхода относятся: отсутствие инструментов для продуктивной работы с графовой природой данных и слабая ориентированность встроенного языка запросов под графовые задачи.

По полученным результатам можно заключить, что все поставленные задачи были успешно решены, цель достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

Приложение 1

Код проекта для работы с базами данных

```
// Database.java
package db;

import org.jgrapht.Graph;
import org.jgrapht.graph.DefaultEdge;

import java.util.List;

public interface Database extends AutoCloseable {

    void init() throws Exception;

    void clear() throws Exception;

    void addNode(String v) throws Exception;

    void addEdge(String v1, String v2) throws Exception;

    List<Long> addGraph(Graph<Integer, DefaultEdge> graph) throws
       Exception;

    List<?> getByNodeAttribute(String va) throws Exception;

    List<?> getByEdgeAttribute(String ea) throws Exception;

    List<?> getShortestPath(String v1, String v2) throws Exception;

    List<?> getNearestNeighbors(String v, int level) throws Exception;
}

// PostgresImpl.java
package db;

import org.jgrapht.Graph;
import org.jgrapht.GraphPath;
import org.jgrapht.Graphs;
import org.jgrapht.alg.shortestpath.DijkstraShortestPath;
import org.jgrapht.graph.DefaultEdge;
```

```
import org.jgrapht.graph.SimpleDirectedGraph;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Optional;
import java.util.Properties;
import java.util.Set;
import java.util.stream.Collectors;

public class PostgresImpl implements Database {

    private static final String CREATE_IF_NOT_EXIST = "" +
            "DROP TABLE IF EXISTS entity CASCADE;\n" +
            "CREATE TABLE entity\n" +
            "(\n" +
            "    id      SERIAL PRIMARY KEY,\n" +
            "    title   varchar(30) UNIQUE NOT NULL,\n" +
            "    property jsonb\n" +
            ");\n" +
            "\n" +
            "DROP TABLE IF EXISTS predicate CASCADE;\n" +
            "CREATE TABLE predicate\n" +
            "(\n" +
            "    id    SERIAL PRIMARY KEY,\n" +
            "    title varchar(15) UNIQUE NOT NULL\n" +
            ");\n" +
            "\n" +
            "DROP TABLE IF EXISTS triple;\n" +
            "CREATE TABLE triple\n" +
            "(\n" +
            "    subject  integer NOT NULL REFERENCES entity (id),\n" +
            "    predicate integer NOT NULL REFERENCES predicate (id),\n" +
            "    object   integer NOT NULL REFERENCES entity (id),\n" +
            "    property jsonb,\n" +
            "    PRIMARY KEY (subject, predicate, object)\n" +
            ");\n" +
            "\n" +
```

```
        "INSERT INTO predicate (title)\n" +
        "VALUES ('DEPENDENT')\n" +
        "ON CONFLICT DO NOTHING;\n;" +
        "\n" +
        "DROP FUNCTION IF EXISTS getEntity;\n" +
        "CREATE FUNCTION getEntity(varchar) RETURNS integer\n" +
        "AS 'SELECT id FROM entity WHERE title = $1;'\n" +
        "    LANGUAGE SQL\n" +
        "    IMMUTABLE\n" +
        "    RETURNS NULL ON NULL INPUT;\n";


private static final String ADD_NODE = "" +
        "INSERT INTO entity (title)\n" +
        "VALUES ('v%s')\n" +
        "ON CONFLICT DO NOTHING;";

private static final String ADD_EDGE = "" +
        "INSERT INTO triple (subject, predicate, object)\n" +
        "SELECT getEntity('v%s'), 1, getEntity('v%s');";

private static final String GET_ALL_EDGES = "" +
        "SELECT\n" +
        "      es.title AS subject,\n" +
        "      es.property AS subject_property,\n" +
        "      p.title AS predicate,\n" +
        "      t.property AS predicate_property,\n" +
        "      eo.title AS object,\n" +
        "      eo.property AS object_property\n" +
        "FROM triple t\n" +
        "        JOIN predicate p ON t.predicate = p.id\n" +
        "        JOIN entity es ON t.subject = es.id\n" +
        "        JOIN entity eo ON t.object = eo.id\n";

private static final String GET_BY_NODE_ATTRIBUTE = GET_ALL_EDGES + ""
    +
        "WHERE es.property::TEXT LIKE '%' || '%s' || '%' OR\n" +
        "     eo.property::TEXT LIKE '%' || '%s' || '%';";

private static final String GET_BY_EDGE_ATTRIBUTE = GET_ALL_EDGES + ""
    +
        "WHERE t.property::TEXT LIKE '%' || '%s' || '%';";
```

```
private static final String URL =
    "jdbc:postgresql://localhost:6666/postgres";
private static final Properties PROPERTIES = new Properties() {{
    setProperty("user", "postgres");
    setProperty("password", "postgres");
}};

private final Connection connection;
private final Statement statement;

public PostgresImpl() throws SQLException {
    this.connection = DriverManager.getConnection(URL, PROPERTIES);
    this.statement = connection.createStatement();
}

@Override
public void init() throws Exception {
    statement.execute(CREATE_IF_NOT_EXIST);
}

@Override
public void clear() throws Exception {
    statement.executeUpdate("TRUNCATE triple");
    statement.executeUpdate("TRUNCATE entity");
    //statement.executeUpdate("TRUNCATE predicate");
}

@Override
public void addNode(String v) throws Exception {
    statement.execute(String.format(ADD_NODE, v));
}

@Override
public void addEdge(String v1, String v2) throws Exception {
    statement.execute(String.format(ADD_EDGE, v1, v2));
}

@Override
public List<Long> addGraph(Graph<Integer, DefaultEdge> graph) throws
    Exception {
    for (var v : graph.vertexSet()) {
        addNode(v.toString());
    }
```

```
    var time = new ArrayList<Long>(graph.edgeSet().size());

    for (var e : graph.edgeSet()) {
        var v1 = graph.getEdgeSource(e);
        var v2 = graph.getEdgeTarget(e);
        var t = System.nanoTime();
        addEdge(v1.toString(), v2.toString());
        time.add(System.nanoTime() - t);
    }

    return time;
}

private List<List<String>> convertResult(ResultSet resultSet) throws
    SQLException {
    var result = new
        ArrayList<List<String>>(resultSet.getMetaData().getColumnCount());
    while (resultSet.next()) {
        result.add(
                List.of(
                        Optional.ofNullable(resultSet.getString(1)).orElse(""),
                        Optional.ofNullable(resultSet.getString(2)).orElse(""),
                        Optional.ofNullable(resultSet.getString(3)).orElse(""),
                        Optional.ofNullable(resultSet.getString(4)).orElse(""),
                        Optional.ofNullable(resultSet.getString(5)).orElse(""),
                        Optional.ofNullable(resultSet.getString(6)).orElse("")
                )
        );
    }
    return result;
}

public List<?> getAllEdges() throws Exception {
    var resultSet = statement.executeQuery(GET_ALL_EDGES);
    return convertResult(resultSet);
}

@Override
public List<?> getByNodeAttribute(String va) throws Exception {
    var resultSet =
        statement.executeQuery(GET_BY_NODE_ATTRIBUTE.replaceAll("%s",
        va));
```

```
    return convertResult(resultSet);
}

@Override
public List<?> getByEdgeAttribute(String ea) throws Exception {
    var resultSet =
       statement.executeQuery(GET_BY_EDGE_ATTRIBUTE.replace("%s", ea));
    return convertResult(resultSet);
}

private Graph<Integer, DefaultEdge> loadGraph(List<List<String>> data)
   {
    Graph<Integer, DefaultEdge> graph = new
       SimpleDirectedGraph<>(DefaultEdge.class);
    data.forEach(row -> {
        var v1 = Integer.parseInt(row.get(0).substring(1));
        var v2 = Integer.parseInt(row.get(4).substring(1));
        graph.addVertex(v1);
        graph.addVertex(v2);
        graph.addEdge(v1, v2);
    });
    return graph;
}

@Override
public List<?> getShortestPath(String v1, String v2) throws Exception {
    if (v1.startsWith("v")) {
        v1 = v1.substring(1);
    }
    if (v2.startsWith("v")) {
        v2 = v2.substring(1);
    }

    var data = convertResult(statement.executeQuery(GET_ALL_EDGES));
    var graph = loadGraph(data);

    return Optional.ofNullable(new DijkstraShortestPath<>(graph)
            .getPath(Integer.parseInt(v1), Integer.parseInt(v2)))
            .map(GraphPath::getEdgeList)
            .orElse(List.of());
}

@Override
```

```
    public List<?> getNearestNeighbors(String v, int level) throws
        Exception {
        var data = convertResult(statement.executeQuery(GET_ALL_EDGES));
        var graph = loadGraph(data);

        var result = new HashSet<>(Graphs.neighborListOf(graph,
            Integer.parseInt(v.substring(1))));
        for (int i = 1; i < level; i++) {
            Set.copyOf(result).forEach(v1 ->
                result.addAll(Graphs.neighborListOf(graph, v1)));
        }
        return
            result.stream().distinct().sorted().collect(Collectors.toList());
    }

    @Override
    public void close() throws Exception {
        statement.close();
        connection.close();
    }
}

// Neo4jImpl.java
package db;

import org.jgrapht.Graph;
import org.jgrapht.graph.DefaultEdge;
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Session;
import org.neo4j.driver.internal.InternalPath;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;
import java.util.stream.StreamSupport;

public class Neo4JImpl implements Database {

    public static final String URI = "bolt://localhost:7687";
    public static final String USER = "neo4j";
```

```
public static final String PASSWORD = "QWErty123";

private final Driver driver;
private final Session session;

public Neo4JImpl() {
    driver = GraphDatabase.driver(URI, AuthTokens.basic(USER,
        PASSWORD));
    session = driver.session();
}

@Override
public void init() {
    clear();
}

@Override
public void clear() {
    session.writeTransaction(tx -> tx.run("MATCH (n) DETACH DELETE
        n;"));
}

@Override
public void addNode(String v) {
    session.writeTransaction(tx -> tx.run(String.format("CREATE
        (v%s:NODE {id:%s})", v, v)));
}

@Override
public void addEdge(String v1, String v2) {
    session.writeTransaction(tx -> tx.run("MATCH\n" +
            "(v1:NODE), (v2:NODE)\n" +
            "WHERE v1.id = " + v1 + " AND v2.id = " + v2 + "\n" +
            "CREATE (v1)-[r:RELATED]->(v2)\n" +
            "RETURN type(r)"
    ));
}

@Override
public List<Long> addGraph(Graph<Integer, DefaultEdge> graph) {
    // Check data size
    if (graph.vertexSet().size() > 10_000) throw new
        RuntimeException("To many nodes");
```

```
        if (graph.edgeSet().size() > 100_000) throw new
            RuntimeException("To many edges");

        if (true) {
            return addGraphFixed(graph);
        }

        for (var v : graph.vertexSet()) {
            addNode(v.toString());
        }

        var time = new ArrayList<Long>(graph.edgeSet().size());

        for (var e : graph.edgeSet()) {
            var v1 = graph.getEdgeSource(e);
            var v2 = graph.getEdgeTarget(e);
            var t = System.nanoTime();
            addEdge(v1.toString(), v2.toString());
            time.add(System.nanoTime() - t);
        }

        return time;
    }

    public List<Long> addGraphFixed(Graph<Integer, DefaultEdge> graph) {
        // Render nodes
        String nodes = graph.vertexSet()
                .stream()
                .map(i -> String.format("(a%d:NODE{id:%d})", i, i))
                .collect(Collectors.joining(", "));

        // Render edges
        String edges = graph.edgeSet()
                .stream()
                .map(edge -> {
                    var v1 = graph.getEdgeSource(edge);
                    var v2 = graph.getEdgeTarget(edge);
                    return String.format("(a%d)-[:RELATED]->(a%d)", v1, v2);
                })
                .collect(Collectors.joining(", "));

        var t = System.nanoTime();
```

```
        session.writeTransaction(tx -> tx.run("CREATE " + nodes + " CREATE
            " + edges));
        return List.of(System.nanoTime() - t);
    }

    @Override
    public List<?> getByNodeAttribute(String va) {
        return session.writeTransaction(tx -> tx.run("MATCH (n) RETURN n;")
                .stream().collect(Collectors.toList()));
    }

    @Override
    public List<?> getByEdgeAttribute(String ea) {
        return session.writeTransaction(tx -> tx.run("MATCH (n) -[r]-> (m)
            RETURN r;")
                .stream().collect(Collectors.toList()));
    }

    private static final String SHORTEST_PATH = "" +
            "MATCH " +
            "   (v1:NODE {id: %s} )," +
            "   (v2:NODE {id: %s})," +
            "   path = shortestPath((v1)-[:RELATED*]-(v2)) " +
            "RETURN path";

    @Override
    public List<?> getShortestPath(String v1, String v2) {
        return session.writeTransaction(tx -> {
            InternalPath path = (InternalPath)
                tx.run(String.format(SHORTEST_PATH, v1, v2))
                    .single().asMap().get("path");
            return StreamSupport.stream(path.spliterator(), false)
                    .map(segment -> String.format("(%s : %s)",
                            segment.start().asMap().get("id"),
                            segment.end().asMap().get("id")))
                    .collect(Collectors.toList());
        });
    }

    private static final String NEAREST_NEIGHBORS = "" +
            "MATCH (m) -[*1..%d]-> (:NODE{id: %s}) -[*1..%d]-> (n) \n" +
            "RETURN m, n;";
```

```
    @Override
    public List<?> getNearestNeighbors(String v, int level) {
        return session.writeTransaction(tx ->
                tx.run(String.format(NEAREST_NEIGHBORS, level, v,
                    level)).stream()
                        .flatMap(record -> Stream.of(
                                record.get("m").asMap().get("id"),
                                record.get("n").asMap().get("id")))
                        .distinct()
                        .sorted()
                        .collect(Collectors.toList())
        );
    }

    @Override
    public void close() {
        session.close();
        driver.close();
    }
}

// ArangoDBImpl.java
package db;

import com.arangodb.ArangoDatabase;
import com.arangodb.ArangoGraph;
import com.arangodb.entity.DocumentField;
import com.arangodb.entity.DocumentField.Type;
import com.arangodb.entity.EdgeDefinition;
import com.arangodb.entity.VertexEntity;
import org.jgrapht.Graph;
import org.jgrapht.graph.DefaultEdge;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ArangoDBImpl implements Database {

    private static final String DB_NAME = "graph_db";
```

```
private static final String GRAPH_NAME = "graph";
private static final String EDGE_COLLECTION_NAME = "edges";
private static final String VERTEX_COLLECTION_NAME = "nodes";

private static final EdgeDefinition EDGE_DEFINITION = new
    EdgeDefinition()
        .collection(EDGE_COLLECTION_NAME)
        .from(VERTEX_COLLECTION_NAME)
        .to(VERTEX_COLLECTION_NAME);

private final com.arangodb.ArangoDB arangoDB;
private final ArangoDatabase db;
private ArangoGraph graph;

public ArangoDBImpl() {
    arangoDB = new com.arangodb.ArangoDB.Builder().build();

    if (!arangoDB.db(DB_NAME).exists()) {
        arangoDB.createDatabase(DB_NAME);
    }
    db = arangoDB.db(DB_NAME);

    init();
}

private <T> List<T> execute(String q, Class<T> tClass) throws
    Exception {
    List<T> result = new ArrayList<>();
    var arangoCursor = db.query(q, String.class);
    while (arangoCursor.hasNext()) {
        Object[] cells = arangoCursor.next().replaceAll("[\\[\\]\"]",
            "").split(",");
        var params = Collections.nCopies(cells.length,
            String.class).toArray(Class[]::new);
        result.add(tClass.getConstructor(params).newInstance(cells));
    }
    return result;
}

@Override
public void init() {
    if (!db.graph(GRAPH_NAME).exists()) {
        db.createGraph(GRAPH_NAME, List.of(EDGE_DEFINITION), null);
```

```
    }
    graph = db.graph(GRAPH_NAME);
    clear();
}

@Override
public void clear() {
    if (graph.exists()) {
        graph.drop();
        db.collection("edges").drop();
        db.collection("nodes").drop();
    }
    db.createGraph(GRAPH_NAME, List.of(EDGE_DEFINITION), null);
    graph = db.graph(GRAPH_NAME);
}

@Override
public void addNode(String v) {

}

@Override
public void addEdge(String v1, String v2) {

}

@Override
public List<Long> addGraph(Graph<Integer, DefaultEdge> graph) {
    // Check data size
    if (graph.vertexSet().size() > 10_000) throw new
        RuntimeException("To many nodes");
    if (graph.edgeSet().size() > 100_000) throw new
        RuntimeException("To many edges");

    // Insert nodes
    var vertexCollection =
        this.graph.vertexCollection(VERTEX_COLLECTION_NAME);
    var idToVertexEntity = new HashMap<Integer,
        VertexEntity>(graph.vertexSet().size());
    graph.vertexSet().forEach(vertex ->
            idToVertexEntity.put(vertex,
                vertexCollection.insertVertex(new Node(vertex))));
```

```
        // Insert edges
        var edgeCollection =
            this.graph.edgeCollection(EDGE_COLLECTION_NAME);
        var time = new ArrayList<Long>(graph.edgeSet().size());
        graph.edgeSet().forEach(edge -> {
            var v1 = graph.getEdgeSource(edge);
            var v2 = graph.getEdgeTarget(edge);
            var t = System.nanoTime();
            edgeCollection.insertEdge(new NodeEdge(
                    idToVertexEntity.get(v1).getId(),
                    idToVertexEntity.get(v2).getId()));
            time.add(System.nanoTime() - t);
        });

        return time;
    }

    @Override
    public List<?> getByNodeAttribute(String va) throws Exception {
        return execute("FOR node IN nodes RETURN [node._id, node._key]",
            Node.class);
    }

    @Override
    public List<?> getByEdgeAttribute(String ea) throws Exception {
        return execute("FOR edge IN edges RETURN [edge._id, edge._from,
            edge._to]", NodeEdge.class);
    }

    @Override
    public List<?> getShortestPath(String v1, String v2) throws Exception {
        String q = String.format("FOR v IN OUTBOUND SHORTEST_PATH
            'nodes/%s' TO 'nodes/%s' " +
                "GRAPH 'graph' RETURN [v._id, v._key]", v1, v2);
        return execute(q, Node.class);
    }

    @Override
    public List<?> getNearestNeighbors(String v, int level) throws
        Exception {
        String qIn = String.format("FOR v IN 1..%d INBOUND 'nodes/%s' " +
                "GRAPH 'graph' RETURN v._key", level, v);
        String qOut = String.format("FOR v IN 1..%d OUTBOUND 'nodes/%s' " +
```

```
            "GRAPH 'graph' RETURN v._key", level, v);
    return Stream.concat(
            execute(qIn, Integer.class).stream(),
            execute(qOut, Integer.class).stream()
    ).distinct().sorted().collect(Collectors.toList());
}

public List<?> shortestPath(Integer v1, Integer v2) throws Exception {
    String q = String.format("FOR v IN OUTBOUND SHORTEST_PATH
       'nodes/%d' TO 'nodes/%d' " +
            "GRAPH 'graph' RETURN [v._id, v._key]", v1, v2);
    return execute(q, Node.class);
}

public List<?> nearestNeighbors(Integer v, int level) throws Exception
    {
    String qIn = String.format("FOR v IN 1..%d INBOUND 'nodes/%d' " +
            "GRAPH 'graph' RETURN v._key", level, v);
    String qOut = String.format("FOR v IN 1..%d OUTBOUND 'nodes/%d' " +
            "GRAPH 'graph' RETURN v._key", level, v);
    return Stream.concat(
            execute(qIn, Integer.class).stream(),
            execute(qOut, Integer.class).stream()
    ).distinct().sorted().collect(Collectors.toList());
}

@Override
public void close() {
    arangoDB.shutdown();
}

private static class Node {
    @DocumentField(Type.ID)
    private String id;
    @DocumentField(Type.KEY)
    private final String value;

    @SuppressWarnings({"unused", "RedundantSuppression"}) //Used vie
       reflection
    public Node(String id, String value) {
        this.id = id;
        this.value = value;
    }
```

```
        public Node(Integer value) {
            this.value = String.valueOf(value);
        }

        @Override
        public String toString() {
            return value;
        }
    }

    private static class NodeEdge {
        @DocumentField(Type.ID)
        private String id;
        @DocumentField(Type.FROM)
        private final String from;
        @DocumentField(Type.TO)
        private final String to;

        @SuppressWarnings({"unused", "RedundantSuppression"}) //Used vie
           reflection
        public NodeEdge(String id, String from, String to) {
            this.id = id;
            this.from = from;
            this.to = to;
        }

        public NodeEdge(String from, String to) {
            this.from = from;
            this.to = to;
        }

        @Override
        public String toString() {
            return String.format("(%s : %s)", from, to);
        }
    }
}
```