# Аннотация

В рамках дипломной работы поставлены задачи хранения и анализа биллинговой информации. Исследованы существующие СУБД, позволяющие решать эти задачи, рассмотрены их преимущества и недостатки. Спроектирована и реализована графовая база данных, позволяющая хранить и обрабатывать биллинговые данные заявленных объемов. Для обеспечения доступа к данным спроектирован и реализован специализированный язык графовых запросов.

В организационно-экономической части представлено технико-экономическое обоснование разработки графовой базы данных, проанализирована структура затрат на проект и построен календарный план-график проекта.

Пояснительная записка к работе содержит текст на 96 листах формата А4, 35 таблиц, 8 рисунков, список литературы из 34 библиографических ссылок. К данной работе разработана презентация в формате \*.pdf из 14 слайдов.

**Содержание**

[Аннотация 1](#_Toc421742285)

[Введение 5](#_Toc421742286)

[1. Обзор существующих решений 7](#_Toc421742287)

[1.1. Графовые базы данных 7](#_Toc421742288)

[1.1.1. СУБД Neo4J GraphDB 9](#_Toc421742289)

[1.1.2. СУБД Sparksee GraphDB 11](#_Toc421742290)

[1.1.3. СУБД Orient GraphDB 11](#_Toc421742291)

[1.1.4. СУБД Titan GraphDB 12](#_Toc421742292)

[1.1.5. СУБД Sones GraphDB 14](#_Toc421742293)

[1.2. Языки запросов к графовым данным 14](#_Toc421742294)

[1.2.1 Язык запросов Cypher 15](#_Toc421742295)

[1.2.2 Язык запросов Gremlin 16](#_Toc421742296)

[1.2.3 Язык запросов GQL 17](#_Toc421742297)

[1.3. Постановка задачи 18](#_Toc421742298)

[2. Разработка графовой базы данных и языка запросов 19](#_Toc421742299)

[2.1. Задачи накопления и анализа 19](#_Toc421742300)

[2.1.1. Накопление биллинговых данных 19](#_Toc421742301)

[2.1.2. Анализ биллинговых данных 20](#_Toc421742302)

[2.2. Структура хранилища 22](#_Toc421742303)

[2.2.1. Списки смежности 22](#_Toc421742304)

[2.2.2. Отображение идентификаторов 23](#_Toc421742305)

[2.2.3. Перечисление ребер 24](#_Toc421742306)

[2.2.4. Индексирование сообществ 24](#_Toc421742307)

[2.2.5. Вспомогательные файлы 25](#_Toc421742308)

[2.3. Алгоритмы аналитических операций 26](#_Toc421742309)

[2.3.1. Алгоритм поиска соседних вершин 26](#_Toc421742310)

[2.3.2. Поиск путей между вершинами 27](#_Toc421742311)

[2.3.3. Поиск окружающей сети 27](#_Toc421742312)

[2.3.4. Объединение попарных пересечений 28](#_Toc421742313)

[2.3.5. Поиск сообществ 28](#_Toc421742314)

[2.4. Язык запросов к графовому хранилищу 29](#_Toc421742315)

[2.4.1. Функции языка запросов 29](#_Toc421742316)

[2.4.2. Описание языка запросов 29](#_Toc421742317)

[2.5. Архитектура графовой СУБД 33](#_Toc421742318)

[2.5.1. Модуль графового хранилища 33](#_Toc421742319)

[2.5.2. Модуль графовых операций 33](#_Toc421742320)

[2.5.3. Модуль подключения аналитических операций 34](#_Toc421742321)

[2.5.4. Модуль службы сетевого доступа 34](#_Toc421742322)

[2.5.5. Модуль интерпретатора языка запросов 34](#_Toc421742323)

[3. Реализация графовой базы данных и языка запросов 36](#_Toc421742324)

[3.1. Реализация графового хранилища 36](#_Toc421742325)

[3.1.1. Диаграмма классов графового хранилища 36](#_Toc421742326)

[3.1.2. Класс GraphDB 38](#_Toc421742327)

[3.1.3. Класс Analyzer 43](#_Toc421742328)

[3.2. Реализация работы с графовыми операциями 49](#_Toc421742329)

[3.2.1. Диаграмма классов описания аналитических операций 49](#_Toc421742330)

[3.2.2. Описание графовых операций 50](#_Toc421742331)

[3.2.3. Диаграмма классов подключения аналитических операций 53](#_Toc421742332)

[3.2.4. Создание аналитических операций 55](#_Toc421742333)

[3.3. Реализация службы сетевого доступа 57](#_Toc421742334)

[3.3.1. Диаграмма классов службы сетевого доступа 57](#_Toc421742335)

[3.3.2. Очередь выполнения операций 58](#_Toc421742336)

[3.3.3. Служба сетевого доступа 59](#_Toc421742337)

[3.4. Реализация интерпретатора языка запросов 62](#_Toc421742338)

[3.4.1. Диаграмма классов интерпретатора языка запросов 62](#_Toc421742339)

[3.4.2. Программная реализация лексического анализа 63](#_Toc421742340)

[3.4.3. Программная реализация синтаксического анализатора 65](#_Toc421742341)

[3.4.4. Программная реализация семантического анализа 69](#_Toc421742342)

[4. Тестирование 73](#_Toc421742343)

[5. Технико-экономическое обоснование 77](#_Toc421742344)

[5.1. Экономическая проблема 77](#_Toc421742345)

[5.2. Расчет затрат 78](#_Toc421742346)

[5.2.1. Трудоемкость разработки 78](#_Toc421742347)

[5.2.2. Исполнители 79](#_Toc421742348)

[5.2.3. Заработная плата 81](#_Toc421742349)

[5.2.4. Стоимость продукта 82](#_Toc421742350)

[5.3. Вывод 83](#_Toc421742351)

[Заключение 85](#_Toc421742352)

[Список литературы 86](#_Toc421742353)

[Приложение А. Руководство пользователя 89](#_Toc421742354)

[A.1. Системные требования 89](#_Toc421742355)

[A.2. Составные части 89](#_Toc421742356)

[A.3. Установка и запуск 90](#_Toc421742357)

[A.4. Выполнение запросов к графовой базе данных 91](#_Toc421742358)

[A.4.1. CREATE DATABASE 91](#_Toc421742359)

[A.4.2. DROP DATABASE 91](#_Toc421742360)

[A.4.3. TRUNCATE 92](#_Toc421742361)

[A.4.4. INSERT NODE 92](#_Toc421742362)

[A.4.5. INSERT EDGE 92](#_Toc421742363)

[A.4.6. DELETE NODE 93](#_Toc421742364)

[A.4.7. DELETE EDGE 94](#_Toc421742365)

[A.4.8. IMPORT FILE 94](#_Toc421742366)

[A.4.9. SELECT 94](#_Toc421742367)

# Введение

С 80-х годов реляционные СУБД занимали доминирующее положение среди средств хранения данных. Несмотря на то, что реляционные хранилища обеспечивают наилучшее сочетание простоты, устойчивости, гибкости, производительности, масштабируемости и совместимости, их показатели по каждому из этих пунктов не обязательно выше, чем у аналогичных систем, ориентированных на конкретную функциональность. Однако универсальность реляционных СУБД перевешивала какие-либо другие недостатки.

Сегодня ситуация несколько иная. В последние годы значительно возросло количество так называемых NoSQL хранилищ [1] (Not only SQL, не только SQL), реализующих модели данных, имеющие существенные отличия от традиционной реляционной модели. Основная цель таких решений — расширить возможности баз данных в тех областях, где реляционная модель и SQL недостаточно гибки, и не вытеснять их там, где они справляются со своими задачами. Создателями таких баз данных указывается множество преимуществ использования NoSQL решений, в том числе высокая производительность при использовании специфических моделей данных и легкость работы с ними.

Одним из наиболее популярных и актуальных подвидов нереляционных хранилищ являются графовые базы данных. Как ясно из названия, основная модель данных в них - это классический математический граф. В последнее время наблюдается бурный рост интереса к графовым базам данных в связи с тем, что такая система представления данных оказалась естественной и востребованной в современном мире различных социальных связей (Интернет, социальные сети и т.д.). К достоинствам графовых моделей баз данных по сравнению с традиционной реляционной моделью исследователи относят не только возможность естественной реализации графовых операций (поиска путей, выделения сообществ и т.п.) в такой базе данных, но и гибкую схему данных, позволяющую унифицировать хранение разнородных объектов.

С каждым годом наблюдается существенный рост активности абонентов сотовой связи и, соответственно, кратное увеличение объемов биллинговой информации. Таким образом, все более долгой и трудоемкой становится ее обработка с помощью реляционных хранилищ. Вариантом оптимизации такой обработки предлагается выполнение аналитических задач при помощи графовых баз данных.

Однако, как и в случае с реляционными базами данных, разработчики таких продуктов стараются в той или иной степени поддерживать универсальность своего хранилища данных. Это неизбежно приводит к падению производительности на тех операциях, которые сами по себе могли бы выполняться очень эффективно, но в силу универсальности приложения задействуют множество дополнительных модулей и производят большое количество посторонних операций.

Таким образом, в дипломной работе ставится задача разработки собственного, достаточно узкоспециализированного графового хранилища, способного накапливать биллинговую информацию на жестком диске и реализующего все необходимые алгоритмы работы с ней. В таком хранилище будут использоваться только те структуры данных, которые позволят эффективно выполнять запрошенные алгоритмы, следствием чего будет являться очень производительное решение поставленных аналитических задач с ограничениями объемов данных, во много раз превосходящие ограничения других графовых баз данных.

Помимо этого, для удобства работы с базой данных будет реализовано несколько интерфейсов доступа, которые предложат пользователю возможность выполнять различными способами импорт данных, все операции анализа данных и производить мониторинг состояния графового хранилища.

# Обзор существующих решений

Проведем краткий обзор уже существующих графовых баз данных [2]. Рассмотрим конкретные особенности каждой из них: способы хранения и анализа графов, область применения, наличие интерфейса доступа к данным, условия распространения и технической поддержки. Кроме того, отдельно рассмотрим существующие языки запросов к графовым хранилищам, отдельно выделив предоставляемую ими функциональность по выборке данных.

**Отметим некоторые особенности биллинговых данных:**

* большой объем. За один день среди операторов «большой тройки» может происходить больше миллиарда соединений;
* естественная графовая структура, особенно в отличие от данных, сгенерированных случайным образом;
* постоянное обновление и, соответственно, частое возникновение задач анализа.

## Графовые базы данных

В рамках данной работы будет рассмотрена возможность применения графовых баз данных в качестве программного обеспечения для программно-аппаратных комплексов обработки биллинговой информации. Биллинг оператора сотовой связи [3] представляет собой информацию обо всех соединениях за определенный период времени внутри сотовой сети. Эти данные естественным образом представляются в виде графа, если считать абонента вершиной, а соединение между абонентами – ребром.

Определим задачу обработки биллинговой информации в виде двух подзадач: накопления и анализа. Решение задачи накопления должно позволять после предварительной подготовки, выполнять загрузку исходного графа, заданного в текстовом виде, в специальное графовое хранилище. Такое хранилище должно предоставлять возможность в любой момент времени осуществить доступ к конкретной вершине или ребру графа. При необходимости, этот же продукт должен пополнять созданное хранилище новыми графовыми данными. Решение задачи анализа должно предоставлять возможность выполнения на созданном хранилище следующих графовых алгоритмов: обход графа в глубину, обход графа в ширину до любого уровня, поиск всех путей между заданными вершинами определенной длины. Результатом работы любого из алгоритмов должен быть список ребер соответствующего подграфа.

Зададим ограничения на объем данных в поставленных задачах и тип оборудования, на которых они будут выполняться. Будем говорить, что графовое хранилище полностью удовлетворяет условиям задачи накопления биллинговой информации, если хранилище позволяет накопить до 16 миллиардов графовых объектов при скорости загрузки данных выше 10000 ребер в секунду. Этих значений достаточно для обработки биллинговых данных самых загруженных мобильных операторов за период времени, равный году. Хранилище полностью удовлетворяет условиям задачи анализа, если любой из перечисленных алгоритмов на загруженном графе выполняется не дольше 60 минут. Допустимое оборудование для работы графового хранилища: процессор уровня не ниже Intel Core 2 Duo 2х2.0 GHz, оперативная память объемом не менее 4 GB и жесткий диск объемом не менее 120 GB.

В первую очередь, необходимо выяснить, существуют ли графовые базы данных, позволяющие накапливать данные в описанных объемах, а не только проводить разовые операции с неизменяемыми графами [4]. Помимо этого, требуется определить, возможно ли выполнять сложные аналитически операции над большими графами в рамках одного локального сервера с заданной конфигурацией. На сегодняшний день существует множество различных реализаций баз данных с графовой моделью хранения [2, 3]. В текущем обзоре мы остановимся на пяти из них, которые представляют наибольший интерес с точки зрения функциональности, производительности и области применения: Neo4J, Sparksee, OrientDB, Titan и Sones. Общие характеристики рассматриваемых графовых БД приведены в таблице 1.

**Таблица 1 ‒ Сравнительные характеристики графовых БД**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Характерис-тика | СУБД | | | | |
| Neo4J | Sparksee  (DEX) | OrientDB | Titan | Sones |
| Разработчик | Neo Technology | Sparsity Technology | Orient Technology | Aurelius | Sones GMBH |
| Год выпуска | 2009 | 2008 | 2010 | 2012 | 2009 |
| Последняя версия | 2.1 | 5.1 | 2.0 | 0.5 | 2.1 |
| Лицензии | Free, Enterprise | Free, Commercial, Educational | Free, Enterprise | Free, Commercial | Free |
| Язык реализации | Java | С++ | Java | Java | С# |
| Поддержка API | Java,  .NET, JavaScript, Python, Ruby,  PHP,  REST и др. | C++,  Java,  .NET | Java,  REST | Java,  REST | .NET,  REST |
| Модель данных | Граф | Граф | Документ, Граф | Граф,  Таблица | Граф |

### ****СУБД Neo4J GraphDB****

Графовая БД Neo4J [5] имеет полноценную возможность устойчивого хранения данных на жестком диске, и, следовательно, объем хранящейся информации ограничен лишь объемами жесткого диска [6]. Для достижения максимальной производительности в Neo4J существует два типа кэширования: файловый кэш (file buffer cache) и объектный кэш (object cache). Первый кэширует данные с жесткого диска с целью увеличения скорости операций ввода/вывода. Второй кэш хранит в себе различные объекты графа: вершины, ребра и свойства в специальном оптимизированном формате для увеличения производительности обходов графа.

Neo4J обладает режимом импорта большого объема данных BatchInserter. В этом режиме происходит отключение транзакций в базе данных, следствием чего является резкое увеличение скорости импорта. В Neo4J существует два режима построения распределенной системы: режим отказоустойчивости, в котором данные полностью копируются между серверами, и режим горизонтального масштабирования, в котором данные распределяются между несколькими серверами. Эти режимы могут быть скомбинированы. В Neo4J используется собственный язык запросов к графовым данным Cypher, который поддерживает все возможности хранилища и постоянно развивается.

В Neo4J реализованы алгоритмы обхода графов, в том числе все необходимые алгоритмы для анализа биллинговых данных. Накопление и обработка графа в Neo4J были протестированы на оборудовании большей мощности, чем минимально необходимое [7, 8, 9]. В качестве тестовых данных использовались графы объемом от 10 до 100 миллионов объектов, потому что больший объем невозможно было загрузить с заявленной скоростью 10000 ребер в секунду [7, 8, 9]. Загрузка осуществлялась в режиме BatchInserter, но после того, как у Neo4J заполнялся кэш, скорость импорта заметно снижалась. Проанализировав результаты, можно сделать вывод, что основной областью применения Neo4J являются системы быстрого доступа к графовым данным, объемы которых не превышают 100 млн ребер. При работе с графами такого размера, Neo4J показывает очень хорошую производительность на запросах поиска всех ближайших соседей вершины - до 10 секунд на запрос. На поставленных задачах анализа биллинговой информации Neo4J работает значительно хуже: до 45 минут на запрос [7, 8]. Таким образом, Neo4J хорошо подходит для работы в качестве подсистемы хранения для различных онлайн-проектов, где не требуется выполнять обходы графа, но для анализа биллинговой информации данная СУБД не подходит потому, что не позволяет загрузить достаточное количество данных с требуемой скоростью.

### ****СУБД Sparksee GraphDB****

Sparksee GraphDB [10], как и Neo4J, имеет полноценную поддержку устойчивого хранения данных [11]. Но, в отличие от Neo4J, ядро Sparksee написано на C++, что очень хорошо сказывается на производительности [7, 8]. Графовая БД Sparksee имеет только объектный кэш, который обеспечивает хранение в оперативной памяти всех часто используемых объектов хранилища. В СУБД присутствует режим горизонтального масштабирования для увеличения количества хранимых данных. Однако, в Sparksee отсутствует полноценный язык запросов, потому доступ к данным предоставляется только с помощью API на нескольких языках программирования, в том числе C# и Java.

База данных Sparksee имеет широкие возможности по обходу графов. Для этих целей СУБД предоставляет множество встроенных алгоритмов, таких как поиск в глубину, поиск в ширину, алгоритм Дейкстры, поиск путей и поиск компонент сильной связности. Таким образом, в СУБД присутствует необходимая функциональность по анализу биллингового графа. Благодаря специализированному ядру обработки графов, написанному на C++, Sparksee – единственная графовая база данных, позволяющая в рамках одного сервера осуществлять хранение достаточно больших графов объемом до 1 млрд объектов. Накопление и анализ графа в Sparksee тестировались на оборудовании большей мощности, чем минимально необходимое [7, 8, 9]. СУБД демонстрирует действительно высокую производительность при поисках путей и обходах графа: поставленные задачи анализа выполняются не дольше 15 минут на запрос [7, 8, 9]. Таким образом, Sparksee подходит для биллингового анализа на графах, но с ограниченным объемом данных: до 1 млрд объектов [9].

### СУБД Orient GraphDB

OrientDB, аналогично Neo4J и Sparksee, имеет полноценную устойчивую систему хранения данных на жестком диске [12]. Что касается модели данных, то OrientDB объединяет в себе возможности документо-ориентированной и графовой БД. Также поддерживается интерфейс объектно-ориентированной БД, который работает поверх документо-ориентированного слоя. Как и в Neo4J, поддерживается режим разового импорта больших массивов данных – BulkImport. Благодаря совмещению различных моделей данных, появляется возможность удобно хранить графы с большим количеством дополнительных свойств и сопутствующих данных, но это же приводит к проблемам с производительностью импорта. СУБД обладает режимом горизонтального масштабирования, который обеспечивает увеличение объема хранимых данных. Для доступа к данным применяется графовый язык запросов – Gremlin.

Так же, как и в Neo4J, в OrientDB реализованы все основные аналитические алгоритмы для работы с биллинговыми данными, такие как обходы в глубину, ширину, поиск путей. Накопление и обработка графа в OrientDB были протестированы на оборудовании большей мощности, чем минимально необходимое [3]. Импорт осуществлялся в режиме BulkImport. Уже при объеме графа, превышающем 2 млн объектов, наблюдается резкое падение скорости импорта [3], а при меньшем количестве данных производительность является схожей с Neo4J [3]. Запрос получения всех ближайших соседей выполняется менее 2-х секунд, а более сложные запросы биллингового анализа выполняются до 3-5 минут. Таким образом, оценив производительность загрузки и анализа данных в OrientDB, можно сделать вывод о ее полной непригодности для решения поставленных задач из-за невозможности работать с графами требуемого объема.

### СУБД Titan GraphDB

В отличие от других графовых баз данных, Titan GraphDB для устойчивого хранения данных использует сторонние технологии: Apache Cassandra, Apache HBase и Oracle BerkeleyDB [13]. Во время создания конкретного экземпляра базы данных имеется возможность выбора системы хранения данных, а графовые интерфейсы над этими данными предоставляются Titan GraphDB. Кроме того, так же на выбор предоставляются несколько систем полнотекстового поиска: ElasticSearch, Solr, Lucene. Возможно практически неограниченное горизонтальное масштабирование графового хранилища на несколько серверов с целью увеличения объема хранимых данных и ускорения выполнения графовых алгоритмов. Для доступа к графовым данным предоставляется язык запросов Gremlin.

Аналогично другим графовым БД, Titan предоставляет все необходимые алгоритмы обработки графов, необходимые для биллингового анализа. Тестирование СУБД Titan производилось на оборудовании большей мощности, чем минимально необходимое [3]. Благодаря возможности выбора подходящей системы хранения, в Titan GraphDB наблюдается очень хорошая производительность импорта данных, особенно по сравнению с другими графовыми базами данных: до 50000 ребер в секунду. При использовании распределенности, потенциально можно загрузить до 2-х миллиардов объектов [14]. На таком объеме данных возможен только анализ производительности запросов извлечения и создания графовых объектов (до 5 секунд на запрос). Но при этом, время выполнения биллинговых аналитических запросов на одном сервере с меньшим объемом графа, до нескольких миллионов объектов, отличается в худшую сторону: до 45 минут на запрос [3]. Таким образом, можно сделать вывод, что Titan GraphDB подходит для таких проектов, где требуется хранить графовые данные большого объема и есть возможность использовать кластеризацию, т.е. распределение на много серверов. Для решения задач анализа биллинговой информации графовая база данных Titan не подходит, поскольку не позволяет загрузить достаточное количество графовых данных в рамках одного локального сервера и выполнить аналитические задачи быстрее 60 минут.

### СУБД Sones GraphDB

Поскольку компания-разработчик прекратила свое существование, то единственная версия базы данных, которую возможно протестировать – это бесплатная версия Сommunity, в которой доступна только загрузка графа в оперативную память. Таким образом, тестирование будет ограничено небольшими объемами данных, которые целиком поместятся в оперативную память. Соответственно, работать с такими данными возможно либо только в случае их маленького объема, либо в случае постоянной очистки текущего хранилища от старых данных [15]. В связи с этим, БД Sones интересна в первую очередь полноценной реализацией языка запросов к графовым данным – GQL. Он является SQL-подобным и предоставляет удобный доступ ко всем возможностям хранилища.

Sones имеет ограниченную поддержку графовых алгоритмов. При работе с данным продуктом все их нужно реализовывать разработчику с помощью предоставляемого инструментария доступа к графовым данным. Поскольку хранение графа доступно только в памяти, можно вести речь исключительно об использовании этой базы данных как хранилища небольших графов для удобного доступа к ним, что позволяет язык запросов GQL.

## Языки запросов к графовым данным

Помимо программного интерфейса, для графового хранилища необходим удобное средство доступа к данным, а именно язык запросов. Графовое хранилище биллинговой информации берет на себя основную роль в обработке данных, в связи с чем на язык запросов не должен обеспечивать реализацию логики анализа. Проведем обзор существующих языков запросов к графовым базам данных с целью выявления подходов, которые позволят сделать язык запросов к графовому хранилищу наиболее простым.

На текущий момент имеется три языка запросов, которые используются в графовых базах данных и развитие которых продолжается:

* Cypher – самый распространенный язык запросов к графовым данным, активно развивается, поддерживается базой данных Neo4J;
* Gremlin – следующий по распространенности язык запросов, поддерживается базой данных Titan.
* GQL – наименее распространенный из полноценных языков запросов к графовым БД, реализован в графовой базе Sones, сейчас поддерживается сообществом.

Остановимся более подробно на каждом из этих языков для того, чтобы изучить, как в каждом из них реализован способ извлечения и фильтрации информации в базе.

### 1.2.1 Язык запросов Cypher

Язык запросов Cypher [16] обладает следующими основными возможностями:

* использование шаблонов фильтрации для получения информации из графа;
* добавление к вершинам меток с целью связывания каждой вершины с определенной информацией;
* создание, обновление и удаление вершин, ребер, меток и свойств;
* управление индексами и ограничениями.

Грамматика запроса для извлечения данных из хранилища выглядит следующим образом [16] (где MATCH – это и есть шаблон фильтрации):

[MATCH WHERE]

[WITH [ORDER BY] [SKIP] [LIMIT]]

RETURN [ORDER BY] [SKIP] [LIMIT]

В целом, язык запросов Cypher, несмотря на свою распространенность и поддержку в Neo4J, достаточно плохо применим для анализа биллинговой информации. Главная причина тому – необходимость применения большого количества шаблонов фильтрации, в то время как основным требованием для языка запросов к графовому хранилищу является его максимальная простота.

### 1.2.2 Язык запросов Gremlin

Язык запросов Gremlin [17] обладает следующими основными возможностями:

* использование последовательного вызова функций, применяющихся одна к другой для получения информации из графа;
* добавление к вершинам меток с целью связывания каждой вершины с определенной информацией;
* реализация интерфейса Tinkerpop [18], позволяющего стандартизировать языки запросов к графовым данным;
* создание, обновление и удаление вершин, ребер, меток и свойств;
* управление индексами и ограничениями.

В языке Gremlin к какому-либо подграфу применяется последовательность функций, благодаря чему можно выполнять обходы и проводить фильтрацию. В языке существуют следующие основные функции:

* V(key,value) – фильтрация вершин по заданному ключу и значению;
* E(key,value) – фильтрация ребер по заданному ключу и значению;
* id()– функция получения идентификатора элемента;
* label() – функция получения метки элемента.

Важным преимуществом языка Gremlin перед другими языками запросов является возможность выполнять последовательно применяемые друг к другу графовые функции. Такой подход позволяет удобно выполнять анализ биллинговой информации, представленной в виде графа. Кроме того, при таком подходе легко добавлять новые методы анализа, для этого достаточно лишь реализовать аналитическую функцию в графовом хранилище и добавить эту функцию к языку запросов.

### 1.2.3 Язык запросов GQL

Язык запросов GQL [19] обладает следующими основными возможностями:

* SQL-подобный синтаксис, который серьезно упрощает изучение языка;
* вершины и ребра разных типов, которые создаются отдельно и могут иметь достаточно сложную иерархию наследования;
* доступ к стандартным DDL и DML операциям.

Синтаксис операции выборки данных имеет следующий вид:

FROM Identifier alias [,Identifier Alias]

SELECT \* | selection\_list

[WHERE where\_condition]

[GROUP BY group\_expr]

[ORDER BY order\_expr]

[OFFSET offset\_expr]

[LIMIT limit\_expr]

[DEPTH depth\_expr],

где selection\_list – это аналогичный SQL-запросу список условий, по которым выполняется выборка объектов из графа.

Отдельно стоит отметить такую особенность языка запросов GQL, как отсутствие внутреннего идентификатора у объекта. Это значит, что идентификатор требуется задавать при создании объекта, что приводит к возможности создания ребер между несуществующими вершинами.

Главное преимущество GQL – это SQL-подобный синтаксис языка. Если необходимо выполнить выборку данных, то данный язык является лучшим вариантом, поскольку SQL является привычным для большинства пользователей и за долгие годы зарекомендовал свое удобство и многофункциональность. Таким образом, GQL имеет широкий потенциал для дальнейшего расширения функциональности, которая при этом будет оставаться привычной для большинства пользователей.

## Постановка задачи

Оценка существующих графовых хранилищ позволяет определить, что они вполне пригодны для использования в определенных целях, но не подходят для решения поставленных задач анализа биллинговой информации в связи с отсутствием возможности обрабатывать графы объемом более 1 млрд ребер. Таким образом, можно сделать вывод о необходимости реализации собственного графового хранилища, предоставляющего только необходимую функциональность для анализа биллинговой информации, но удовлетворяющего заданным значениям производительности.

Что касается языков запросов, то использование рассмотренных языков в чистом виде невозможно в силу их универсальности, которая позволяет решить многие задачи, но сопряжена с ростом трудоемкости. Потому требуется реализация языка запросов, основанного на таких особенностях рассмотренных языков, как SQL-подобный синтаксис и последовательные вызовы аналитических операций. Реализация такой функциональности в языке запросов позволит производить наиболее удобный доступ к биллинговым данным.

# Разработка графовой базы данных и языка запросов

В разделе 1.1 была определена задача обработки биллинговой информации как совокупность подзадач накопления и анализа, кроме того, были заданы ограничения выполнения таких задач. Теперь более подробно рассмотрим отдельно подзадачи накопления и анализа, язык запросов к графовому хранилищу и общую архитектуру проекта.

## Задачи накопления и анализа

### Накопление биллинговых данных

Биллинг оператора сотовой связи естественным образом может быть представлен в виде ориентированного графа *G := (V, E)*, где абоненты – вершины V, а соединения между абонентами – ребра E. Кроме того, у каждого соединения между абонентами есть один важный параметр, который постоянно требуется учитывать при анализе – время совершения соединения. Таким образом, биллингом L оператора сотовой связи будем называть множество троек следующего вида, каждая из которых обозначает соединение между двумя абонентами:

*{source\_node*; *destination\_node; rel\_time}*,

где

* *source\_node* – строковое значение, идентифицирующее начальную вершину соединения;
* *destination\_node* – строковое значение, идентифицирующее конечную вершину соединения;
* *rel\_time* – строковое значение, определяющее время появления соединения.

Кроме того, будем говорить, что биллинг *L* является биллингом *Lt* за период времени *t*, если все значения *rel\_time* попадают в заданный интервал времени *t*. Например, *Lday* – это биллинг с данными о соединениях за один день, а *Lmonth* – это биллинг с данными о соединениях за один месяц.

Исходные данные могут содержаться в каком-либо внешнем хранилище, в реляционной базе данных или текстовых файлах. Определим задачу накопления и хранения биллинговых данных, исходя из того, что набор таких данных определяет биллинг за конечный период времени.

Пусть задан некоторый период времени *t*, по истечении которого в графовое хранилище должна происходить загрузка биллинговых данных *Lt* за этот период времени. Например, если периодом времени выбран один день, то каждый день в хранилище должны импортироваться биллинговые данные за последний день. Структура хранения графа должна позволять выполнять выборку только того подграфа, который был загружен за определенный период времени.

### Анализ биллинговых данных

Одной из самых актуальных задач, возникающих при анализе биллинговой информации, является поиск соседних вершин какой-либо заданной вершины. Помимо непосредственных соседей вершины, может возникнуть потребность найти всех соседей, находящихся на определенном удалении от заданной вершины. Для выполнения такой задачи должна быть реализована функция, в которую передается глубина поиска *n*, а возвратиться должны все соседние вершины, до которых минимальное расстояние не более *n*. Обозначим кратчайшее расстояние от вершины *s* до какой-либо вершины *v* функцией *σs(v)*. Тогда результатом работы реализованной функции будет множество:

Помимо поиска соседей, крайне востребованной операцией в анализе биллинговой информации, да и графов в целом, является поиск всех путей между двумя вершинами, длина которых не превышает n. Для выполнения этой операции, должна быть реализована функция, которой передаются две исходные вершины и длина *n*, в пределах которой необходимо найти все пути между заданными вершинами. Обозначив путь между двумя вершинами как , а его длину как , тогда результатом работы такой функции будет множество:

Введем еще одну, уже весьма специализированную для анализа биллинговой информации, операцию поиска окружающей сети. Она является логическим расширением операции поиска соседей и ее суть заключается в том, что отсечение вершин производится не только по удаленности от исходной, а еще и по количеству инцидентных ребер у каждой вершины. В наиболее распространенном случае рассматриваются такие вершины, у которых инцидентных ребер больше одного. Таким образом, обозначив количество инцидентных ребер вершины *v* функцией , результатом выполнения функции поиска окружающей сети будет множество:

Еще одна востребованная операция обработки биллинговой информации – пересечение найденных соседей двух вершин. Эта операция должна быть реализована как функция, в которую передается список вершин, для каждой вершины находятся соседи этой вершины не глубже определенного уровня, после чего для всех пар вершин производится пересечение таких множеств. Если пересечение производилось для найденных соседей вершин *s1* и *s2*, то результатом будет множество:

Кроме того, отдельного упоминания требует еще одна важная задача анализа биллинговой информации - поиск сообществ. Сообществом будем называть такой подграф *Vcom* графа *V*, в котором кратность каждого ребра *mult(e)* (где *e* - ребро) лежит в диапазоне значений [*multmin*; *multmax*] и количество вершин *count(Vcom)* в нем лежит в диапазоне значений [*cntmin*; *cntmax*]. Параметры *multmin*, *multmax*,*cntmin*, и *cntmax*определяются в зависимости от задачи. Несмотря на то, что фактически это задача анализа, а не хранения графа, ее эффективное решение сводится к хранению специализированной информации, позволяющей эффективно находить сообщества. Таким образом, сообществом является следующее множество:

## Структура хранилища

Экземпляром хранилища биллинговой информации будем называть директорию на диске, в которой содержатся все файлы с данными и прочие вспомогательные файлы. В дальнейшем, выполнение операции импорта или анализа подразумевается на каком-либо экземпляре хранилища. При хранении данных во внешней памяти часто приходится использовать пары вида ключ-значение. Это необходимо для хранения полного графа в виде списков смежности и различной сопутствующей информации. Для устойчивого хранения таких пар на жестком диске воспользуемся структурой данных B+ - дерево, т.к. оно позволяет наиболее эффективно выполнять операции добавления, поиска и удаления данных во внешней памяти [24].

Структура хранилища состоит из следующих элементов:

* файлы списков смежности входящих и исходящих ребер;
* файлы отображения строковых идентификаторов вершин на внутренние идентификаторы хранилища;
* файлы перечисления соединений за определенный период времени;
* файлы индексирования сообществ за определенный период времени;
* вспомогательные файлы статистики и подсчета периодов времени.

### Списки смежности

Хранение списков смежности осуществляется в двух файлах: в файле, котором хранится список смежности для входящих ребер и в аналогичном файле для исходящих ребер. Для хранения списков смежности используется высокопроизводительное B+ - дерево, потому что количество объектов в них может увеличиваться до миллиардов. Таким образом, необходима именно такая реализация B+ - дерева, которая обеспечивает наилучшую производительность на больших объемах данных. Ключом в дереве выступает вершина, а значением – уникальный список соседей данной вершины. При сериализации такого списка во внешнюю память он будет представлять простую последовательность идентификаторов.

При обработке биллинговой информации крайне редко приходится обращаться к одному конкретному соседу вершины, обычно требуется выполнить обход всех соседей, а потому такой подход оправдан. Кроме того, следует выделить возможность использования еще одной оптимизации хранения, являющейся следствием того, что осуществляется хранение именно биллинговых данных. При анализе биллинга сотового оператора часто бывает так, что некоторые вершины имеют значительно больше соединений, чем среднестатистические. Обычно, это различные короткие номера. При обходе графа или поиске сообществ они не учитываются, но при этом занимают слишком много места в хранилище. Поэтому, для каждой вершины необходимо ввести дополнительный параметр, который обозначает, какое максимальное количество соседей следует сохранять.

### Отображение идентификаторов

Хранить идентификаторы вершин биллинговых данных в исходных строках в виде телефонных номеров – задача излишне трудоемкая и по производительности, и по размеру занимаемого места на диске. Намного выгоднее хранить идентификаторы в виде 64-разрядных целых чисел. Этой разрядности достаточно для хранения заявленного количества объектов, и сравнение таких чисел производится достаточно быстро.

Потому возникает необходимость хранить отображение исходных строковых идентификаторов во внутренние целочисленные идентификаторы и обратно, чтобы при выборке данных получать граф с исходными идентификаторами. Отображения хранятся в двух файлах, каждый из которых представляет собой B+ - дерево. В первом файле ключом является строковый идентификатор, а значением – целочисленный. Во втором файле ключом является целочисленный идентификатор, а значением – строковый идентификатор. В биллинговых данных многие исходные идентификаторы являются телефонными номерами, которые сами по себе легко преобразуются в 64-разрядное целое число. Потому, в случае возможности преобразования строки в целое число, новый внутренний идентификатор создаваться не будет.

### Перечисление ребер

Для получения выборки биллинговых данных из хранилища, ограниченных временным диапазоном, необходимо для каждого ребра результирующей выборки проверить, принадлежит ли оно заданному диапазону. Для этого при импорте биллинга за определенный период времени создается специальный файл, содержащий в себе перечисление ребер из этого биллинга. Файл находится в отдельной директории, по имени которой можно определить период связей в биллинге. В качестве оптимизации можно исключить хранение всех кратных ребер, которые могут присутствовать в биллинговых данных в значительном количестве.

Таким образом, пусть результатом какой-либо операции выборки данных является подграф *Gres*, и нужно выбрать из него подграф *Gres\_d1\_d2*, такой, что в нем все соединения были совершены в диапазон времени [*d1*; *d2*]. Для этого требуется проверить присутствие ребер подграфа *Gres*хотя бы в одном файле с перечислениями ребер, принадлежащих диапазону времени [*d1*; *d2*].

### Индексирование сообществ

Наиболее трудоемкой задачей при выполнении поиска сообществ является поиск всех таких ребер, кратность которых *multij* принадлежит заданному диапазону значений. При загрузке биллинга за определенный период времени, для ускорения выполнения поиска сообществ можно предвычислять все ребра *e*, кратность которых равна заданному значению *mult(e)*.

С этой целью при импорте биллинга за определенный период времени создается специальный файл предвычисления кратности ребер, находящийся в той же папке, что и файл с перечислением ребер. Файл предвычисления кратности ребер представляет собой особую модификацию B+ - дерева, которая, в отличие от обычной, обладает возможностью хранить в одном файле несколько несвязанных друг с другом деревьев, что является очень удобным способом сохранить несколько множеств ребер. Таким образом, в этом файле создается несколько деревьев, каждое из которых является множеством ребер с одинаковой кратностью *mult(e)*.

В дальнейшем, при необходимости выполнить операцию поиска сообществ с параметрами кратности ребер [*multmin*; *multmax*] и параметрами количества вершин в сообществе [*cntmin*; *cntmax*], достаточно будет выполнить поиск по всем файлам кратностей ребер за необходимый период времени, извлекая из них все множества ребер с такими значениями кратностей *mult(e),* что *mult(e) >= multmin* и *mult(e)* *<= multmax*. После чего в полученном наборе ребер найти все компоненты связности, в которых множество вершин *V* удовлетворяет условию *count(V) >= vmin*и *count(V) <= vmax*. Благодаря тому, что поиск компонент связности осуществляется только по ограниченному набору ребер, уже удовлетворяющему условию кратности, этот поиск будет происходить значительно быстрее, чем по всему графу.

### Вспомогательные файлы

Дополнительно, для взаимодействия с хранилищем, используются два вспомогательных файла. Первый хранит в себе статистическую информацию в текстовом виде для быстрого доступа к ней, не взаимодействуя с B+ - деревьями. К этой информации относятся количество всех вершин, ребер, количество вершин в каждом списке смежности и количество загруженных биллинговых данных.

Второй файл содержит в себе информацию о временных интервалах, которым принадлежат загруженные биллинги, и пути к соответствующим папкам с файлами перечисления ребер и предподсчета их кратностей. Сам файл является B+ - деревом, в котором ключом является временной промежуток, которому принадлежит загруженный биллинг, а значением – путь к соответствующей папке. Таким образом, при выполнении задач анализа биллинговой информации с отсечением по времени с помощью данного файла можно сразу же найти необходимые папки, содержащие файлы перечисления ребер и предподсчета кратностей ребер.

## Алгоритмы аналитических операций

В разделе 2.1.2 были в общем виде рассмотрены наиболее часто встречающиеся операции анализа биллинговых данных. Остановимся более подробно на алгоритме выполнения каждой такой операции.

### Алгоритм поиска соседних вершин

Основной задачей является поиск соседних вершин для каждой переданной вершины с заданной глубиной поиска. Таким образом, в функцию передается параметр глубины *n*, а возвратиться должны все соседние вершины, до которых минимальное расстояние не больше *n*. Такая задача будет реализована с помощью алгоритма поиска в ширину.

На вход алгоритма подаётся заданный невзвешенный граф, и номер стартовой вершины *s*. Граф может быть как ориентированным, так и неориентированным, для алгоритма это не важно. Алгоритм выполняется с помощью очереди вершин, в котором изначально находится стартовая вершина *s.* Таким образом, алгоритм представляет собой следующую последовательность шагов:

1. поместить стартовую вершину *s* в изначально пустую очередь;
2. извлечь из начала очереди вершину и поместить ее в результирующее множество;

а) если глубина текущей вершины больше *n*, то завершить алгоритм;

б) в противном случае, в конец очереди добавляются все соседи данной вершины, которые не находятся в результирующем множестве или в очереди;

1. повторить п. 2.

### Поиск путей между вершинами

Поиск путей осуществляется таким образом, чтобы между заданными вершинами найти все пути длины не более *n*. Пусть начальная вершина имеет обозначение *source\_v*, а конечная – *dest\_v*. Тогда данная операция выполняется по следующему алгоритму:

1. осуществляется поиск соседей вершины *source\_v* до глубины
2. осуществляется поиск соседей вершины *dest\_v* до глубины
3. полученные множества вершин пересекаются;
4. для каждой вершины *mid\_v* в результирующем множестве строятся два отрезка пути от вершины *source\_v* до *mid\_v* и от вершины *mid\_v* до *dest\_v*;
5. полученные пары отрезков объединяются в конечные пути, которые и будут результатом работы операции.

### Поиск окружающей сети

Поиск окружающей сети заключается в том, чтобы найти всех таких соседей вершины не далее заданного уровня глубины, у которых количество ближайших соседей больше фиксированного значения *k* (в наиболее распространенном случае больше одного).

Таким образом, решение этой задачи заключается в следующих четырех этапах:

1. найти для каждой вершины соседей до указанного уровня с помощью поиска в ширину, описанного ранее;
2. для каждой найденной вершины найти те, у которых количество ближайших соседей равно *k* и удалить их;
3. объединить результаты поиска для каждой исходной вершины.

### Объединение попарных пересечений

Поиск пересечений для каждой пары переданных вершин представляет собой некоторую функцию, в которую передается список вершин, для каждой вершины находятся соседи этой вершины не далее определенного уровня, после чего для всех пар вершин производится пересечение таких множеств и возвращается их объединение.

Таким образом, решение задачи будет заключаться в следующем цикле по всем парам вершин (вначале результирующее множество пусто):

1. найти для каждой вершины соседей до указанного уровня с помощью поиска в ширину, описанного ранее;
2. найти пересечение найденных множеств соседей двух вершин;
3. объединить найденное пересечение с результирующим множеством.

### Поиск сообществ

Задача поиска сообществ подразумевает поиск такого подграфа, в котором все вершины этого подграфа связаны друг с другом значительно сильнее, чем с вершинами вне подграфа. Алгоритм выполнения операции поиска сообществ подразумевает, что уже произведен предподсчет кратностей всех ребер графа, как описано в разделе 2.1.2. В этом случае, решение поставленной задачи будет производиться с помощью следующего алгоритма:

1. выбрать подмножество ребер, которое удовлетворяет условиям минимального значения кратности и максимального значения кратности ребер;
2. в результирующем множестве выполнить поиск всех компонент связности;
3. из полученных компонент связности выбрать те, в которых количество вершин удовлетворяет заданным ограничениям на количество вершин в сообществе.

## Язык запросов к графовому хранилищу

Вначале рассмотрим функции, выполнение которых должен поддерживать язык запросов к графовому хранилищу, после чего опишем его грамматику.

### Функции языка запросов

Основными функциями, которые должен предоставлять язык запросов к графовой базе данных, являются:

* создание экземпляра графового хранилища с определенным идентификатором;
* удаление базы данных по идентификатору;
* очистка базы данных от всех хранящихся в ней объектов;
* добавление вершины со строковым идентификатором в граф;
* добавление ребра, соединяющего две вершины, в граф, с возможностью указания временной метки ребра;
* удаление вершины из графа по идентификатору;
* удаление ребра из графа по идентификаторам вершин, которые оно соединяет;
* импорт данных в хранилище из текстового файла, содержащего биллинг;
* выборка данных из хранилища с помощью последовательно примененных друг к другу аналитических операций.

На вход любой функции подаётся список вершин и набор параметров. Список вершин может быть задан явно или быть результатом работы другой функции. Аналогично, функция возвращает результат работы, к которому потом может быть применена другая аналитическая функция.

### Описание языка запросов

Рассмотрим теперь основные лексические, синтаксические и грамматические особенности разрабатываемого языка запросов к графовой базе данных.

В языке запросов мы определены пять типов лексем: идентификаторы, ключевые слова, константы, знаки операций и другие разделители. Компилятор языка игнорирует пробелы, знаки табуляции и перевода строки, расположенные между лексемами. При этом требуется, чтобы хотя бы один из этих символов разделял смежные идентификаторы, ключевые слова и константы.

При разбиении входного потока на лексемы должно учитываться правило самой длинной лексемы, а именно: если входной поток был разбит на лексемы вплоть до некоторой позиции, то начиная с этой позиции, из входного потока выбирается самая длинная последовательность символов, которая может составлять лексему.

Идентификатор представляет собой последовательность букв и цифр, начинающуюся с буквы. Константы разделяются на целочисленные и строковые. Целочисленные константы в языке запросов представляют собой неотрицательные целые числа в диапазоне от 0 до 232 - 1. Они записываются в виде последовательности цифр, перед которыми в фигурных скобках может указываться основание системы счисления. Цифрами считаются арабские цифры от 0 до 9 и латинские буквы: A означает 10, B означает 11 и т.д. Основание системы счисления записывается в виде десятичного числа в диапазоне от 2 до 36. Если основание не указано, то константа считается десятичной. Строковая константа представляет собой последовательность символов, заключенных в кавычки. Комментарием является любой фрагмент языка запросов, который заключен между парами символов /\* и \*/.

В таблице 2 перечислены идентификаторы, зарезервированные для использования в качестве управляющих слов (все эти слова не зависят от регистра):

Таблица 2 **‒** Управляющие слова

|  |  |  |  |
| --- | --- | --- | --- |
| CREATE | DROP | DATABASE | IN |
| INSERT | NODE | INTO | VALUE |
| TIME | DELETE | FROM | SELECT |
| IMPORT | FILE | AND | OR |
| TRUNCATE | EDGE | DATERANGE | NEIGHBORS |
| INTERSECTION | SURROUND\_NETWORK | ALLPATHS | COMMUNITIES |

Запросы могут разделяться точкой с запятой, и каждый из них отвечает за одну из описанных выше функций. Типичный сценарий запросов к графовому хранилищу представляет собой последовательность запросов определенного вида, применяемых к графовой базе данных.

Рассмотрим синтаксис каждого запроса:

* *CREATE DATABASE database\_id [IN directory\_path]* – создание экземпляра графовой базы данных с идентификатором *database\_id* в директории *directory\_path*, где *directory\_path* представляет собой строковую константу (если путь в запросе не указан, директорией по умолчанию считается корневой каталог программы, выполняющей запрос);
* *DROP DATABASE database\_id* – удаление экземпляра графовой базы данных с идентификатором *database\_id*;
* *TRUNCATE database\_id* – удаление всех объектов из экземпляра графовой базы данных с идентификатором *database\_id*;
* *INSERT NODE INTO database\_id VALUE node\_value* – вставка вершины *node\_value* в базу данных с идентификатором *database\_id*, где *node\_value* – строковая константа;
* *INSERT EDGE INTO database\_id VALUE source\_node, dest\_node [TIME date\_time]* – вставка ребра, соединяющего вершины *source\_node* и *dest\_node* в базу данных с идентификатором *database\_id*, где *source\_node* и *dest\_node* – строковые константы (возможно указание дополнительного параметра *date\_time* – временной метки ребра);
* *DELETE NODE FROM database\_id VALUE node\_value* – удаление вершины *node\_value* из базы данных с идентификатором *database\_id*;
* *DELETE EDGE FROM database\_id VALUE source\_node, dest\_node* – удаление ребра, соединяющего вершины *source\_node* и *dest\_node* из базы данных с идентификатором *database\_id*;
* *IMPORT FILE file\_path INTO database\_id* – импорт файла с данными *file\_path* в базу данных с идентификатором *database\_id*, где *file\_path* представляет собой строковую константу.

Более подробно остановимся на операторе выборки данных из хранилища – SELECT. Этот оператор имеет следующую грамматику:

*<select\_operator> ::= FROM <database\_id>*

*[DATE\_RANGE <start\_date> <end\_date>]*

*SELECT { \* | <selection\_cond> }*

*<selection\_cond> ::= <nodes\_list> . <functions>*

*<nodes\_list> ::= (node\_id [ ,…n ])*

*<functions> = <analysis\_function>[.<functions>]*

*<analysis\_function> = { NEIGHBORS | INTERSECTION | SURROUND\_NETWORK | ALLPATHS | COMMUNITIES }*

Таким образом, оператор SELECT позволяет выбрать все данные из хранилища или же применить к определенному подграфу последовательность аналитических операций. Кроме того, можно обозначить временной диапазон анализа с помощью ключевого слова DATERANGE.

Для вызова конкретных аналитических функций используются следующие ключевые слова:

* NEIGHBORS – поиск соседних вершин;
* INTERSECTION – поиск объединения попарных пересечений;
* SURROUND\_NETWORK – поиск окружающей сети;
* ALLPATHS – поиск всех путей между двумя вершинам;
* COMMUNITIES –поиск всех сообществ.

## Архитектура графовой СУБД

Перейдем к рассмотрению архитектуры графового хранилища и языка запросов к нему. Графовая база данных, предназначенная для анализа биллинговой информации, состоит из пяти основных модулей:

* + модуль графового хранилища;
  + модуль графовых операций;
  + модуль подключения аналитических операций;
  + модуль службы сетевого доступа;
  + модуль интерпретатора языка запросов.

Рассмотрим более подробно предназначение каждого из них.

### Модуль графового хранилища

Модуль графового хранилища обеспечивает хранение биллинговых данных в виде графа на жестком диске с помощью B+ - деревьев и предоставление доступа к этим данным. Каждый экземпляр графового хранилища хранится в отдельной директории с набором файлов, каждый из которых поддерживает выполнение одной из функций, описанных в разделе 2.2. Предоставление доступа к определенному экземпляру хранилища выполняется с помощью специального объекта, спроектированного с помощью шаблона «синглетон» [27]. Этот объект содержит информацию обо всех экземплярах графовых хранилищ. Такой подход позволяет хранить в оперативной памяти только один объект, ссылающийся на экземпляр графового хранилища, и при каждом запросе к базе данных не приходится заново загружать объекты с диска.

### Модуль графовых операций

Модуль графовых операций обеспечивает выполнение графовых операций, которые делятся на два основных типа: операции импорта и операции анализа. За эти типы операций отвечают абстрактные классы, и любая операция над графовым хранилищем должна наследоваться от одного из этих классов, в зависимости от того, будет эта операция импорта или анализа.

Структура данных, которая хранит в себе аналитические операции, представляет собой очередь специального вида. Она предназначена для хранения последовательности выполнения операций работы с графовой базой данных. Для поддержки корректной последовательности выполнения операций используется специальный компаратор, сортирующий аналитические операции, в зависимости от их приоритета и времени добавления.

### Модуль подключения аналитических операций

Разработка системы анализа графов изначально подразумевает, что должно поддерживаться большое количество средств проведения анализа графа. Таким образом, число методик анализа будет постоянно увеличиваться и должен быть реализован достаточно простой механизм добавления новых типов аналитических операций. Модуль подключения аналитических операций выполняет эту задачу с помощью применения шаблона проектирования «фабричный метод» [27].

### Модуль службы сетевого доступа

Одна из основных задач, которую должна решить реализуемая система – предоставление сетевого доступа к графовому хранилищу. Модуль службы сетевого доступа для решения, позволяет предоставлять программный интерфейс взаимодействия с графовым хранилищем в форме сетевой службы. Клиент, присоединившийся к службе, работает именно с этим интерфейсом. В нем перечислены все методы, необходимые для доступа к данным графового хранилища и обработки аналитических операций.

### Модуль интерпретатора языка запросов

Модуль интерпретатора языка запросов решает задачу выполнения трех основных этапов компиляции:

* лексического анализа ‒ на этом этапе последовательность символов исходного файла преобразуется в последовательность лексем;
* синтаксического анализа ‒ последовательность лексем преобразуется в дерево разбора;
* семантического анализа ‒ дерево разбора обрабатывается с целью установления его семантики. Для каждого узла дерева разбора определяется тип графовой операции, которую необходимо выполнить.

Лексический анализатор в качестве результата своей работы возвращает список лексем, которые необходимо проверить на корректность синтаксиса. Синтаксический анализ будет реализован с использованием метода рекурсивного спуска [29]. Грамматика для реализации этого метода описана в разделе 2.4.2.

Конечной целью обработки какого-либо графового запроса является вызов необходимого метода или последовательности методов, предоставляемых интерфейсом доступа к графовой базе данных. Соответственно, перед вызовом методов, нужно проверить входные данные на корректность, что происходит на этапе выполнения семантического анализа.

# Реализация графовой базы данных и языка запросов

Рассмотрим реализацию модулей проекта, изобразив диаграмму классов каждого из них, дав подробное описание каждого класса и особенности их программной реализации.

## Реализация графового хранилища

### Диаграмма классов графового хранилища

За все операции с конкретным экземпляром графового хранилища отвечает класс GraphDB, реализующий интерфейс IGraphDB. Класс GraphDB является наиболее важным из всех классов модуля работы с графовым хранилищем, поскольку он обеспечивает доступ ко всем файлам базы данных на жестком диске. Реализация интерфейса нужна для того, чтобы можно было создавать графовые хранилища разного типа, в которых отличается структура хранения и типы данных.

Хранение данных во внешних файлах осуществляется с помощью B+ - деревьев в реализациях BPlusTree 14 [25] и STSdb 4.0.7 [26]. Реализация BPlusTree используется по причине своего очень высокого быстродействия [25], что необходимо для эффективного доступа к спискам смежности. А в STSdb присутствует функция хранения в одном файле нескольких несвязанных друг с другом деревьев [26], что позволяет в удобной форме хранить предподсчитанные кратности ребер, используемые для поиска сообществ. Соответственно, для работы с деревом типа BPlusTree в классе GraphDB используются объекты BPlusTree<Tkey, TValue>, а для работы с STSdb – объекты типа IStorageEngine. При загрузке данных в хранилище или доступе к ним во время анализа используются вспомогательные классы Edge и NeighborsInfo, которые описывают одно ребро и список соседей вершины.

Все аналитические операции вынесены в отдельный класс Analyzer. Это позволяет проводить операции анализа над экземплярами графовых хранилищ разного типа. Начальный ввод графа из файлов и вывод в файл результатов аналитических операций осуществляется с помощью класса GraphsIO.

Непосредственный доступ к экземплярам класса IGraphDB осуществляется с помощью класса DatabaseManager. Класс DatabaseManager хранит строго по одному объекту IGraphDB, соответствующего графовому хранилищу на жестком диске.

Диаграмма классов модуля графового хранилища изображена на рисунке 1.



**Рисунок 1 ‒ Диаграмма классов модуля графового хранилища**

Классы, которые реализуют модуль хранения графа представлены в таблице 3.

Таблица 3 **‒** Классы графового хранилища

|  |  |
| --- | --- |
| Класс | Описание |
| IGraphDB | Итерфейс абстрактного графового хранилища, предоставляющий все необходимые методы для извлечения данных из графа |
| GraphDB | Реализация конкретного графового хранилища биллиннговой информации |
| NeighborsInfo | Вспомогательный класс, описывающий всех соседей одной вершины |
| Edge | Вспомогательный класс, описывающий одно ребро графа |
| Analyzer | Класс, реализующий аналитические операции |
| GraphsIO | Класс, реализующий начальный ввод графа из файла и вывод результатов аналитических операций в файл |
| BPlusTree | Класс, реализующий B+-дерево BPlusTree |
| IStorageEngine | Класс, реализующий B+-дерево STSdb |

### Класс GraphDB

Методы интерфейса IGraphDB позволяют применять аналитические операции к экземплярам хранилища разных типов. В работе аналитические операции применяются к классу GraphDB, который реализует хранилище биллинговой информации. Более подробно рассмотрим все методы интерфейса IGraphDB и члены класса GraphDB, которые представлены в таблицах 4, 5 и 6.

Таблица 4 **‒** Методы интерфейса IGraphDB

|  |  |
| --- | --- |
| Метод | Описание |
| GetNumberNodes(); | Возвращает количество вершин в графе |
| GetNumberEdges(); | Возвращает количество ребер в графе |
| GetDatabasePath(); | Возвращает путь к экземпляру хранилища на жестком диске |
| GetAliasByID(long node); | Возвращает исходный идентификатор вершины по внутреннему идентификатору |
| GetIDByAlias(string alias); | Возвращает внутренний идентификатор вершины по исходному идентификатору |
| GetNeighborsByDirection(long node, EdgeDirection direction); | Возвращает список соседей вершины |

Главные методы интерфейса IGraphDB – GetNeighborsByDirection, GetAliasByID и GetIDByAlias, позволяющие получать данные и возвращать результат по исходным идентификаторам, в то время как внутренняя обработка происходит по собственным идентификаторам вершин.

Таблица 5 **‒** Поля класса GraphDB

|  |  |
| --- | --- |
| Поле | Описание |
| BPlusTree<long, NeighborsInfo> btree\_vertices\_incoming | Дерево, хранящее в себе всех соседей каждой вершины, доступных по входящим ребрам |
| BPlusTree<long, NeighborsInfo> btree\_vertices\_outgoing | Дерево, хранящее в себе всех соседей каждой вершины, доступных по исходящим ребрам |
| BPlusTree<KeyValuePair<DateTime, DateTime>,string> btree\_cg\_blocks | Пути к биллинговым данным, загруженным за определенный период |
| IStorageEngine engine | Дерево, хранящее в себе отображения исходные идентификаторы вершин во внутренние |
| string database\_path | Путь к экземпляру хранилища на жестком диске |
| long all\_nodes | Количество вершин |
| long all\_edges | Количество ребер |

Для того чтобы создать B+-дерево в реализации BPlusTree, которыми являются поля btree\_vertices\_incoming, btree\_vertices\_outgoing и btree\_cg\_blocks, необходимо сначала создать объект с его настройками:

|  |
| --- |
| BPlusTree<long, HashSet<long>>.OptionsV2 options =  new BPlusTree<long, HashSet<long>>.OptionsV2(      PrimitiveSerializer.Int64,  new NormalHashSetSerializer<long>(PrimitiveSerializer.Int64))  {      CreateFile = CreatePolicy.IfNeeded,      StoragePerformance = StoragePerformance.Fastest,      FileName = database\_path,      CacheKeepAliveMinimumHistory = 4096,      CacheKeepAliveMaximumHistory = 8192,      FileBlockSize = file\_block\_size  }.CalcBTreeOrder(8, 16); |

Такой объект настроек дерева содержит в себе описания структуры дерева (какие объекты являются ключами, а какие значениями), и его поведения. Для типов данных необходимо указать сериализаторы, которые описывают способы сохранения объектов на диске и чтения их с диска. После чего можно создавать сами объекты деревьев:

|  |
| --- |
| btree\_vertices\_incoming =  new BPlusTree<long, NeighborsInfo>(options\_vertices\_incoming);  btree\_vertices\_outgoing =  new BPlusTree<long, NeighborsInfo>(options\_vertices\_outgoing);  btree\_cg\_blocks =  new BPlusTree<KeyValuePair<DateTime, DateTime>,  string>(options\_cg\_blocks); |

Дерево STSdb можно создать, открыв необходимую таблицу в объекте IStorageEngine. В нашем случае, это позволяет создать отдельно два отображения идентификаторов:

|  |
| --- |
| long\_to\_string = this.engine.OpenXTable<long, string>(@"ll2str");  string\_to\_long = this.engine.OpenXTable<string, long>(@"str2ll"); |

Методы класса GraphDB, помимо реализации интерфейса IGraphDB, выполняют загрузку данных в хранилище разными методами. Эти методы рассмотрены в таблице 6.

Таблица 6 **‒** Методы класса GraphDB

|  |  |
| --- | --- |
| Метод | Описание |
| void Import(  StorageForImport storage) | Обычная загрузка данных в хранилище |
| void BulkImport(  string directory) | Быстрая загрузка большого количества данных |
| long CreateIdByAlias(  string alias) | Вспомогательный метод, который создает новое отображение исходного идентификатора вершины на внутреннее, если оно отсутствует. |
| void TruncateDB() | Очистка всех данных хранилища |
| void InsertNode(  string node) | Вставка вершины |
| void DeleteNode(  string node) | Удаление вершины |
| void DeleteEdge(  Edge edge) | Удаление ребра |

Метод Import(StorageForImport storage) принимает на вход объект с подготовленными данными, после чего они импортируются в хранилище. За подготовку данных отвечает класс StorageForImport, который считывает граф из файла с помощью метода LoadFile(string file). Целью этого метода является создание трех объектов:

|  |
| --- |
| SortedDictionary<long, HashSet<long>> input\_dict\_in;  SortedDictionary<long, HashSet<long>> input\_dict\_out;  Dictionary<KeyValuePair<DateTime, DateTime>,  SortedDictionary<Edge, int>> new\_nums; |

Объект input\_dict\_in содержит в себе всех соседей вершины по входящим ребрам, объект input\_dict\_out – всех соседей по исходящим ребрам, а new\_nums – кратность каждого загруженного ребра в каждом промежутке загруженного биллинга. При загрузке файла осуществляется построчное чтение строк, каждая из которых представляет собой одну связь биллинга (или одно ребро графа). Для каждой связи выполняется метод InsertPair(), который принимает на вход исходную вершину связи, конечную вершину и время совершения связи.

Преобразование идентификаторов из исходных строковых во внутренние целочисленные осуществляется с помощью отображений long\_to\_string и string\_to\_long. Если строка распознается как 8-байтовое целое число, то это число и будет идентификатором, если не распознается, то идентификатором будет значение, на единицу большее размера словаря long\_to\_string со знаком минус:

|  |
| --- |
| Int64 a;  if (!Int64.TryParse(a\_input, out a) || a\_input[0] == '0')  {  if (string\_to\_long.Exists(a\_input))  a = string\_to\_long[a\_input];  else  a = -long\_to\_string.Count() - 1;  long\_to\_string[a] = a\_input;  string\_to\_long[a\_input] = a;  }  Int64 b;  if (!Int64.TryParse(b\_input, out b) || b\_input[0] == '0')  {  if (string\_to\_long.Exists(b\_input))  b = string\_to\_long[b\_input];  else  b = -long\_to\_string.Count() - 1;  long\_to\_string[b] = b\_input;  string\_to\_long[b\_input] = b;  } |

После чего происходит формирование объектов input\_dict\_in, input\_dict\_out и new\_nums:

|  |
| --- |
| if (!input\_dict\_in.ContainsKey(b))  input\_dict\_in.Add(b, new HashSet<long>());  input\_dict\_in[b].Add(a);  if (!input\_dict\_out.ContainsKey(a))  input\_dict\_out.Add(a, new HashSet<long>());  input\_dict\_out[a].Add(b);  if (use\_closed\_groups)  {  var edge = new Edge(a, b);  KeyValuePair<DateTime, DateTime> date\_range =  CreateKeyByDate(time);  if (!new\_nums.ContainsKey(date\_range))  new\_nums.Add(date\_range,  new SortedDictionary<Edge, int>(Edge.GetComparer));  var new\_num\_of\_edges = new\_nums[date\_range];  if (!new\_num\_of\_edges.ContainsKey(edge))  new\_num\_of\_edges.Add(edge, 1);  else  new\_num\_of\_edges[edge]++;  } |

Метод BulkImport(string directory) принимает на вход директорию с файлами, содержащими биллинговые данные. Загрузка данных происиходит похожим образом, как и в методе Import(), но только предварительно сортируя входные файлы с биллинговыми данными по первой вершине, второй вершине и времени связи. Таким образом, данные для загрузки в хранилище формируются последовательно по мере чтения исходного файла, что позволяет загружать очень большие объемы исходных данных. Методы Truncate, InsertNode(), DeleteNode() и DeleteEdge() являются вспомогательными и нужны для работы языка запросов к хранилищу.

### Класс Analyzer

Класса Analyzer отвечает за выполнение аналитических операций в графовом хранилище. Методы класса Analyzer представлены в таблице 7.

Таблица 7 **‒** Методы класса Analyzer

|  |  |
| --- | --- |
| Метод | Описание |
| NeighborsWithEdges(  List<long> nodes,  EdgeDirection direction = EdgeDirection.All,  int depth = 0,  int max\_edges = 0) | Аналитическая операция поиска соседей вершины. |
| SocialCircles(  List<long> nodes,  int depth = 0,  int max\_edges = 0) | Аналитическая операция поиска объединения попарных пересечений. |
| SurroundNetwork(  List<long> nodes,  int depth = 0,  int max\_edges = 0) | Аналитическая операция поиска окружающей сети. |
| AllPaths(  long a\_node,  long b\_node,  int depth,  int max\_edges = 0) | Аналитическая операция поиска путей между вершинами. |
| ClosedGroups(  ClosedGroupsParameters parameters) | Аналитическая операция поиска сообществ. |

Для реализации алгоритма поиска соседних вершин до определенного уровня будет использоваться модифицированный поиск в ширину. В этом алгоритме используется метод, который получает всех ближайших соседей для заданной вершины, в зависимости от направления ребер:

|  |
| --- |
| private HashSet<long> GetNeighborsByDirection  (long node, EdgeDirection direction)  {  if (direction == EdgeDirection.Incoming)  {  if (btree\_vertices\_incoming.ContainsKey(node))  return btree\_vertices\_incoming[node];  else  return new HashSet<long>();  }  if (direction == EdgeDirection.Outgoing)  {  if (btree\_vertices\_outgoing.ContainsKey(node))  return btree\_vertices\_outgoing[node];  else  return new HashSet<long>();  }  HashSet<long> incoming  = new HashSet<long>(), outgoing = new HashSet<long>();  if (btree\_vertices\_incoming.ContainsKey(node))  incoming = btree\_vertices\_incoming[node];  if (btree\_vertices\_outgoing.ContainsKey(node))  outgoing = btree\_vertices\_outgoing[node];  incoming.UnionWith(outgoing);  return incoming;  } |

В этом методе возвращаются все соседние вершины из дерева входящих или исходящих ребер. В случае если нужно получить вообще всех соседей, то берутся множества вершин из обоих деревьев и для этих множеств выполняется операция объединения.

Как было сказано в разделе 2.3.4, прямой алгоритм решения задачи попарных пересечений производит пересечение всех пар заданного списка вершин. Это не очень эффективно, если каждый раз требуется находить множество соседей вершины до достаточно большого уровня. Поэтому, для увеличения производительности, считать таких соседей мы будем только первый раз и запоминать в специальном контейнере neighbors. При всех последующих проходах поиск соседей производиться не будет, а просто будет браться множество из контейнера neighbors:

|  |
| --- |
| for (int i = 0; i < nodes.Count; ++i)  {  neighbors.Add(NeighborsWithPath(nodes[i],  EdgeDirection.All,  depth, max\_edges) );  } |

Таким образом, алгоритм поиска объединения попарных пересечений, описанный в разделе 2.3.4, в коде выглядит следующим образом:

|  |
| --- |
| for (int first\_node = 0;  first\_node < neighbors.Count - 1;  ++first\_node)  {  for (int second\_node = first\_node + 1;  second\_node < neighbors.Count;  ++second\_node)  {  var nodes1 = neighbors[first\_node];  var nodes2 = neighbors[second\_node];  HashSet<long> intersection = new HashSet<long>();  foreach (var neigh in nodes1)  {  if (nodes2.ContainsKey(neigh.Key))  intersection.Add(neigh.Key);  }  ans\_nodes.UnionWith(intersection);  }  } |

Но пока что был получен лишь список вершин, описывающих решение поставленной задачи. Для полноценного решения необходимо еще найти все ребра, которые соединяют найденные вершины. Такая функциональность реализована в следующем вспомогательном методе:

|  |
| --- |
| public HashSet<Edge> GetEdgesByNodes(HashSet<long> nodes)  {  HashSet<Edge> ans = new HashSet<Edge>(Edge.GetEqualityComparer);  foreach (long source in nodes)  {  var neighbors =  GetNeighborsByDirection(source, EdgeDirection.Outgoing);  foreach (long dest in neighbors)  if (nodes.Contains(dest))  ans.Add(new Edge(source, dest));  }  return ans;  } |

Для выполнения поиска окружающей сети осуществляется поиск в графе таких последовательностей вершин, у которых количество соседей равно одному:

|  |
| --- |
| foreach (var pair in neighbors)  {  long current\_node = pair.Key;  while (neighbors[current\_node].Count == 1 && current\_node != node)  {  var enumerator = neighbors[current\_node].GetEnumerator();  if (!enumerator.MoveNext())  {  throw new IndexOutOfRangeException();  }  long next\_node = enumerator.Current;  bad\_nodes.Add(current\_node);  neighbors[next\_node].Remove(current\_node);  current\_node = next\_node;  }  } |

После чего, генерируется результирующая окружающая сеть без отсеянных вершин:

|  |
| --- |
| foreach (var pair in neighbors)  {  if (bad\_nodes.Contains(pair.Key))  continue;  ans\_nodes.Add(pair.Key);  foreach (long dest\_node in pair.Value)  {  ans\_edges.Add(new Edge(Math.Min(pair.Key, dest\_node),  Math.Max(pair.Key, dest\_node)));  }  } |

Поиск путей между двумя вершинами осуществляется с помощью вспомогательного метода NeighborsWithPath(), который ищет всех соседей вершины до определенного уровня, при этом сохраняя путь до каждой вершины от исходной и метода RecoverEdge(), который восстанавливает путь от исходной вершины. Таким образом, для поиска всех путей между вершинами, сначала вычисляется глубина поиска соседей для начальной и конечной вершин:

|  |
| --- |
| int a\_depth = depth / 2;  int b\_depth = depth - a\_depth; |

После чего находятся все соседи исходной и конечной вершин с путями:

|  |
| --- |
| Dictionary<long, KeyValuePair<long, int>> nodes1, nodes2;  if (a\_depth > 0)  nodes1 =  NeighborsWithPath(a\_node, EdgeDirection.All, a\_depth, max\_edges);  else  {  nodes1 = new Dictionary<long, KeyValuePair<long, int>>();  nodes1.Add(a\_node, new KeyValuePair<long, int>(-1, 0));  }  if (b\_depth > 0)  nodes2 =  NeighborsWithPath(b\_node, EdgeDirection.All, b\_depth, max\_edges);  else  {  nodes2 = new Dictionary<long, KeyValuePair<long, int>>();  nodes2.Add(b\_node, new KeyValuePair<long, int>(-1, 0));  } |

Затем производится пересечение найденных соседей:

|  |
| --- |
| HashSet<long> intersection = new HashSet<long>();  foreach (var pair in nodes1)  {  long node = pair.Key;  if (nodes2.ContainsKey(node))  intersection.Add(node);  } |

И находятся все пути от исходной до конечной вершины, не более заданной длины:

|  |
| --- |
| foreach (long node in intersection)  {  long depth1 = nodes1[node].Value;  long depth2 = nodes2[node].Value;  if (depth1 + depth2 <= depth)  {  List<long> path = new List<long>();  path.AddRange(RecoverPath(nodes1, node, true));  path.AddRange(RecoverPath(nodes2, node));  paths.Add(path);  }  } |

Как уже отмечалось в пункте 2.3.5, основными параметрами при поиске сообществ считаются кратность ребер, входящих в сообщество и количество вершин в сообществе. Все ребра определенной кратности проиндексированы в собственном B+ - дереве и доступ к ним осуществляется с помощью следующего объекта:

|  |
| --- |
| ClosedGroupsSTSManager tree\_manager = new ClosedGroupsSTSManager(closed\_groups\_path); |

Выполняется перечисление всех ребер заданной кратности и из них строится граф на списках смежности:

|  |
| --- |
| if (connects\_number >= parameters.MinConnectsNumber &&  connects\_number <= parameters.MaxConnectsNumber)  {  var closed\_groups\_tree = tree\_manager.Get(connects\_number);  foreach (var edge in closed\_groups\_tree)  {  long source = edge.Key.Source;  long dest = edge.Key.Dest;  if (!adjacency\_list.ContainsKey(source))  adjacency\_list.Add(source, new HashSet<long>());  adjacency\_list[source].Add(dest);  if (!adjacency\_list.ContainsKey(dest))  adjacency\_list.Add(dest, new HashSet<long>());  adjacency\_list[dest].Add(source);  }  } |

После чего, в построенном графе выполняется поиск всех компонент связности и остаются только те, размерность которых лежит в заданном промежутке:

|  |
| --- |
| foreach (var pair in adjacency\_list)  {  long source = pair.Key;  if (!used.Contains(source))  {  var final\_graph =  NeighborsWithEdgesClosedGroups(adjacency\_list, source);    if (final\_graph.Key.Count >= parameters.MinMembersNumber &&  final\_graph.Key.Count <= parameters.MaxMembersNumber)  {  ans\_nodes.UnionWith(final\_graph.Key);  ans\_edges.UnionWith(final\_graph.Value);  }  used.UnionWith(final\_graph.Key);  }  } |

## Реализация работы с графовыми операциями

### Диаграмма классов описания аналитических операций

В работе реализовано два класса для импорта данных. Класс ImportOperation загрузить данные из специально подготовленного объекта, а класс BulkImportOperation предназначен для быстрой загрузки графа большого объема из нескольких файлов в пустой экземпляр графового хранидища. Аналитическая операция AnalyzeOperation использует класс MethodologyQuery для описания конкретной аналитической методики.

Для выполнения какой-либо операции, с помощью класса DatabaseManager извлекается объект нужного экземпляра GraphDB и, если это операция анализа, то и соответствующие классы GraphsIO и Analyzer.

UML-диаграмма модуля описания графовых операций представлена на рисунке 2.



**Рисунок 2 ‒ Диаграмма классов, описывающих графовые операции**

### Описание графовых операций

Перейдем к рассмотрению классов, отвечающих за работу с операциями над графом. Сначала рассмотрим члены базового класса ADatabaseOperation, они представлены в таблице 8.

Таблица 8 **‒** Методы класса ADatabaseOperation

|  |  |  |
| --- | --- | --- |
| Тип | Имя | Описание |
| Поле | DateTime time\_of\_loading | Время создания операции |
| Поле | Int32 priority | Приоритет операции |
| Поле | Guid id | Идентификатор операции |
| Поле | String database\_path | Путь к экземпляру хранилища, к которому следует применить операцию |
| Метод | abstract public bool ExecuteOperation() | Абстрактный метод, реализующий логику выполнения операции |

Каждая наследуемая операция от ADatabaseOperation должна реализовать абстрактный метод ExecuteOperation(), который будет выполнять всю логику операции. Кроме того, каждая операция может иметь свой приоритет выполнения priority, чтобы выполниться раньше или позже других операций в очереди выполнения. В случае если операции имеют один приоритет, то они выполняются по мере времени своего создания time\_of\_loading.

За сортировку операций отвечает компаратор DatabaseOperationComparer, который реализует стандартный метод сравнения Compare() следующим образом:

|  |
| --- |
| public int Compare(ADatabaseOperation a, ADatabaseOperation b)  {  if (b.Priority.CompareTo(a.Priority) != 0)  return b.Priority.CompareTo(a.Priority);  else if (a.LoadingTime.CompareTo(b.LoadingTime) != 0)  return a.LoadingTime.CompareTo(b.LoadingTime);  else if (a.Id.CompareTo(b.Id) != 0)  return a.Id.CompareTo(b.Id);  else  return 0;  } |

Рассмотрим поля, которые содержатся в классах, отвечающих за импорт данных в хранилище: ImportOperation и BulkImportOperation. Описание их членов представлено в таблицах 9 и 10 соответственно.

Таблица 9 **‒** Члены класс ImportOperation

|  |  |  |
| --- | --- | --- |
| Тип | Имя | Описание |
| Поле | StorageForImport storage | Объект с данными для импорта |
| Поле | bool import\_closed\_groups | Флаг, указывающий, следует ли осуществлять предподсчет сообществ. |

Реализация метода ExecuteOperation() в классе ImportOperation заключается в получении экземпляра объекта графового хранилища и последующего применения к нему операции импорта:

|  |
| --- |
| GraphDB db =  (GraphDB)DatabaseManager.Instance.GetDatabase(database\_path);  GraphState.Properties.OperationStarted(this);  db.Import(storage.InputDictIn,  storage.InputDictOut,  storage.NumsOfEdges,  import\_closed\_groups); |

Таблица 10 **‒** Члены класса BulkImportOperation

|  |  |  |
| --- | --- | --- |
| Тип | Имя | Описание |
| Поле | string input\_directory | Директория, с файлами для импорта |
| Поле | string ext | Расширение файлов, которые нужно импортировать |
| Поле | ulong ram\_size | Объем оперативной памяти, выделенный для предсортировки исходных файлов |
| Поле | string sorted\_file\_in | Сортированный файл исходных данных по первой вершине |
| Поле | string sorted\_file\_out | Сортированный файл исходных данных по второй вершине |
| Поле | string sorted\_file\_time | Сортированный файл исходных данных по времени совершения связи |

Реализация метода ExecuteOperation() в классе BulkImportOperation заключается в вызове метода внешней сортировки и последующему выполнения метода импорта:

|  |
| --- |
| GraphDB db =  (GraphDB)DatabaseManager.Instance.GetDatabase(database\_path);  GraphState.Properties.OperationStarted(this);  FileSorter.SortFiles(input\_directory,  output\_directory, ext, sort\_types, cfg);  if (!File.Exists(sorted\_file\_in) ||  !File.Exists(sorted\_file\_out) ||  !File.Exists(sorted\_file\_time))  throw new FileNotFoundException(  "One or more sorted files does not exists");  db.BulkImport(sorted\_file\_in, sorted\_file\_out, sorted\_file\_time); |

Перейдем к рассмотрению класса, отвечающего за аналитические операции – AnalyzeOperation. Его члены представлены в таблице 11.

Таблица 11 **‒** Члены класса AnalyzeOperation

|  |  |  |
| --- | --- | --- |
| Тип | Имя | Описание |
| Поле | MethodologyQuery methodology\_query | Описатель аналитической операции, сгенерированный с помощью класса MQFactory |

В методе ExecuteOperation() класса AnalyzeOperation сначала происходит получение необходимого экземпляра класса GraphDB и создание соответствующих классов GraphsIO и Analyzer:

|  |
| --- |
| var db = DatabaseManager.Instance.GetDatabase(database\_path);  Analyzer analyzer = new Analyzer(db);  GraphsIO io = new GraphsIO(db); |

После чего создается директория, в которую будет записан файл результата операции и запускается выполнение анализа с помощью соответствующего метода класса MethodologyQuery:

|  |
| --- |
| string result\_directory =  Path.GetDirectoryName(db.GetDatabasePath()) + "\\results\\";  if (!Directory.Exists(result\_directory))  Directory.CreateDirectory(result\_directory);  metodology\_query.OutputFile = result\_directory + metodology\_query.OutputFile;  metodology\_query.ExecuteAnalyzer(db, analyzer, io);  GraphState.Properties.OperationFinished(); |

### Диаграмма классов подключения аналитических операций

Базовый класс, являющийся описателем аналитической операции – MethodologyQuery. От него наследуются классы, являющиеся описателями конкретных аналитических операций. Параметры каждой аналитической операции задаются в классах, наследуемых от MQParameters. Для того чтобы создать объект класса, наследующегося от MethodologyQuery, нужно воспользоваться фабричными методами, содержащимся в классах типа MQFactory, называемых фабриками.

Список фабрик содержится в классе FactoriesList, который позволяет получить необходимую фабрику по целочисленному или строковому идентификатору. Используя нужную фабрику и передав ее фабричному методу экземпляр объекта MQParameters с заданными параметрами операции, можно получить необходимый описатель аналитической операции, наследуемый от класса MethodologyQuery.

UML-диаграмма модуля подключения аналитических операций представлена на рисунке 3.



**Рисунок 3 ‒ Диаграмма классов подключения аналитических операций**

### Создание аналитических операций

Рассмотрим структуру класса MethodologyQuery, отвечающего за описание аналитической операции. Члены класса MethodologyQuery представлены в таблице 12.

Таблица 12 **‒** Члены класса MetodologyQuery

|  |  |  |
| --- | --- | --- |
| Тип | Имя | Описание |
| Поле | string name | Имя аналитической операции |
| Поле | string output\_file | Имя файла с результатом |
| Поле | int max\_degree | Максимальное количество инцидентных ребер вершины, по которым осуществляется обход |
| Поле | DateTime time\_lower\_bound | Начало периода времени, за который нужно выполнить операцию |
| Поле | DateTime time\_upper\_bound | Конец периода времени, за который нужно выполнить операцию |
| Метод | abstract void ExecuteAnalyzer(IGraphDB db, Analyzer analyzer, GraphsIO io) | Абстрактный метод, реализующий логику выполнения операции |

Поле max\_degree необходимо для оптимизации выполнения обходов графа. Все операции анализа так или иначе используют обход графа в ширину и часто при этом обходе может произойти достижение вершины, у которой очень много инцидентных ребер. В случае биллинговых данных, это могут быть различные короткие номера. Такие вершины в плане анализа интереса не представляют, но при этом сильно ухудшают производительность обхода.

Поля time\_lower\_bound и time\_upper\_bound нужны для выполнения фильтрации результата анализа по времени. Это позволяет извлечь биллинговые данные только за определенный период времени. Основная же логика операции определена в методе ExecuteAnalyzer(), который реализуют все классы, наследующиеся от MethodologyQuery.

Для создания классов MethodologyQuery используются фабричные классы, наследующиеся от MQFactory. Их единственная задача – производить классы MethodologyQuery с помощью фабричного метода CreateMQ(MQParameters parameters), принимающего на вход класс MQParameters, описывающий параметры аналитической операции. Таким образом, члены классов MQFactory и MQParameters описаны в таблице 13.

Таблица 13 **‒** Члены класса MQFactory

|  |  |  |
| --- | --- | --- |
| Тип | Имя | Описание |
| Метод | MethodologyQuery CreateMQ(MQParameters parameters) | Фабричный метод, создающий класс MethodologyQuery по параметрам аналитической операции MQParameters. |

Доступ ко всем фабричным классам осуществляется с помощью вспомогательного класса FactoriesList, предоставляющего доступ к списку фабрик по их идентификаторам и строковым названиям. Члены класса FactoriesList представлены в таблице 14.

Таблица 14 **‒** Члены класса FactoriesList

|  |  |  |
| --- | --- | --- |
| Тип | Имя | Описание |
| Поле | Dictionary<Int32, MQFactory> MQIDToFactory | Словарь, содержащий в себе список фабричных классов, с доступом по целочисленному идентификатору |
| Поле | Dictionary<String, MQFactory> NameToFactory | Словарь, содержащий в себе список фабричных классов, с доступом по имени аналитической операции |
| Метод | MQFactory GetMQFactory(Int32 id) | Метод, возвращающий фабричный класс по целочисленному идентификатору |
| Метод | MQFactory GetMQFactory(String name) | Метод, возвращающий фабричный класс по имени аналитической операции |
| Метод | bool Exists(string name) | Метод, проверяющий существование фабрики по ее имени |
| Метод | bool Exists(int id) | Метод, проверяющий существование фабрики по ее целочисленному идентификатору |

## Реализация службы сетевого доступа

### Диаграмма классов службы сетевого доступа

Класс GraphAPI, реализующий интерфейс IGraphAPI, содержит конкретную реализацию всех методов службы. Кроме того, класс GraphAPI работает с классом SpecialQueue, которые содержит в себе информацию о состоянии всех добавленных в систему операций обработки графа. Таким образом, помимо реализации основных методом службы, класс GraphAPI также выполняет параллельную обработку операций импорта и анализа.

UML-диаграмма модуля службы сетевого доступа представлена на рисунке 4.



**Рисунок 4 ‒ Диаграмма классов службы сетевого доступа**

### Очередь выполнения операций

Очередь выполнения операций Special Queue реализована на основе встроенного в .NET контейнера SortedSet [30]. Члены класса SpecialQueue представлены в таблице 15.

Таблица 15 **‒** Члены класса SpecialQueue

|  |  |  |
| --- | --- | --- |
| Тип | Имя | Описание |
| Поле | SortedSet<ADatabaseOperation> import\_queue | Контейнер, содержащий очередь операций импорта |
| Поле | SortedSet<ADatabaseOperation> analyze\_queue | Контейнер, содержащий очередь операций анализа |
| Поле | HashSet<Guid> finished | Контейнер, содержащий идентификаторы успешно завершенных операций |
| Поле | HashSet<Guid> errors | Контейнер, содержащий идентификаторы операций, завершенных с ошибкой |
| Поле | Dictionary<Guid, String> results | Контейнер, содержащий пути, по которым лежат результаты работы аналитических операций. |
| Метод | Guid PushImport(AImportOperation operation) | Метод, добавляющий операцию импорта в очередь |
| Метод | Guid PushAnalyze(AAnalyzeOperation operation) | Метод, добавляющий операцию анализа в очередь |
| Метод | ADatabaseOperation PopImport() | Метод, возвращающий первую операцию импорта в очереди |
| Метод | ADatabaseOperation PopAnalyze() | Метод, возвращающий первую операцию анализа в очереди |
| Метод | bool Contains(Guid id) | Метод, проверяющий наличие в очереди операции по ее идентификатору |
| Метод | void AddToFinished(Guid id) | Метод, добавляющий идентификатор операции в список успешно завершенных |
| Метод | void AddToErrors(Guid id) | Метод, добавляющий идентификатор операции в список завершенных с ошибкой |

Операции импорта и анализа хранятся в отдельных контейнерах import\_queue и analyze\_queue для возможности последующего распараллеливания выполнения этих операций. Кроме того, отдельно сохраняются идентификаторы завершенных операций с ошибкой в контейнере errors и пути к файлам-результатам аналитических операций в контейнере results.

### Служба сетевого доступа

Перейдем к реализации службы сетевого доступа к графовому хранилища. В качестве технологии создания службы используется Windows Communication Foundation (WCF) [28]. Основным интерфейсом службы является IGraphAPI, его методы представлены в таблице 16.

Таблица 16 **‒** Методы службы сетевого доступа

|  |  |
| --- | --- |
| Имя | Описание |
| void Start() | Начать выполнение операций |
| void Stop() | Остановить выполнение операций |
| Guid ImportFile(  String database\_path,  String import\_file,  bool import\_closed\_groups = true) | Выполнить импорт биллинговых данных из файла |
| Guid ImportBulk(  String database\_path,  string input\_directory,  string ext,  ulong ram\_size) | Выполнить быстрый разовый импорт большого количества биллинговых данных из нескольких файлов |
| Guid Analyze(  String database\_path,  String analyze\_op\_name,  MQParameters parameters) | Выполнить аналитическую операцию. |
| long GetNumberNodes(  String database\_path) | Получить количество вершин в графе |
| long GetNumberEdges(  String database\_path) | Получить количество ребер в графе |
| String GetResultPath(  Guid id) | Получить путь к файлу-результату аналитической операции |
| bool ResultExists(  Guid id) | Проверить, существует ли результат аналитической операции |
| ADatabaseOperation.Status GetOperationStatus(  Guid operation\_id) | Проверить статус выполнения какой-либо операции |
| bool ExistDB(  string database\_path) | Проверить существование экземпляра хранилища данных |
| bool RemoveDB(  string database\_path) | Удалить экземпляр хранилища данных |
| void TruncateDB(  String database\_path) | Очистить все данных в хранилище данных |
| void InsertNode(  String database\_path,  String node) | Вставить вершину в хранилище данных |
| void DeleteNode(  String database\_path,  String node) | Удалить вершину из хранилища данных |
| void DeleteEdge(  String database\_path,  String node\_a,  String node\_b) | Удалить ребро из хранилища данных |

Рассмотрим некоторые особенности реализации класса GraphAPI. Выполнение операций из очереди происходит с помощью двух бесконечных циклов, выполняющихся в параллельных потоках. Поток импорта выполняет следующую последовательность команд:

|  |
| --- |
| if (!queue.EmptyImport())  {  ADatabaseOperation pop;  pop = queue.PopImport();  if (!pop.ExecuteOperation())  {  queue.AddToErrors(pop.Id);  return;  }  queue.AddToFinished(pop.Id);  GC.Collect();  Thread.Sleep(1000);  } |

При наличии операции импорта в очереди, происходит ее выполнение. В случае успешного завершения, она добавляется к списку успешно выполненных операций, иначе – к ошибочно выполненным операциям. Поток анализа выполняет следующую последовательность команд:

|  |
| --- |
| if (!queue.EmptyAnalyze())  {  ADatabaseOperation pop;  pop = queue.PopAnalyze();  if (!pop.ExecuteOperation())  {  queue.AddToErrors(pop.Id);  return;  }  queue.AddToFinished(pop.Id);  AnalyzeOperation analyze\_op = pop as AnalyzeOperation;  queue.AddToResults(analyze\_op.Id, analyze\_op.GetFilePath());  GC.Collect();  Thread.Sleep(1000);  } |

Логика схожая с выполнением операций импорта, но после успешного выполнения операции происходит сохранение пути к файлу, содержащему результат аналитической операции.

Кроме того, для начала выполнения аналитической операции, требуется методу Analyze передать объект класса MQParameters. Если клиент использует доступ к системе с помощью сетевой службы, то у него нет данных о классах, наследующихся от MQParameters. Чтобы предоставить ему эти данные, перед описанием класса MQParameters заданы вспомогательные атрибуты:

|  |
| --- |
| [KnownType(typeof(MQNeighborsParameters))]  [KnownType(typeof(MQIntersectionParameters))]  [KnownType(typeof(MQSurroundNetworkParameters))]  [KnownType(typeof(MQClosedGroupsParameters))]  [KnownType(typeof(MQAllPathsParameters))] |

## Реализация интерпретатора языка запросов

### Диаграмма классов интерпретатора языка запросов

Интерпретация языка запросов к графовому хранилищу осуществляется в 3 этапа: лексический, синтаксический и семантический анализ. На этапе лексического анализа, перебор лексем осуществляет класс Scanner. За описание видов лексем отвечают классы, унаследованные от класса Token: StringToken, KeywordToken, CommentToken, SpecToken, NumberToken и IdentToken.

Кроме того, класс Scanner использует некоторые вспомогательные классы. Класс Position для указания местоположения входной цепочки, класс Fragment для указания фрагмента цепочки и класс Compiler для хранения таблицы имен и идентификаторов.

Этапы синтаксического и семантического анализа объединены в классе SyntaxAnalyzer, который методом рекурсивного спуска осуществляет перебор найденных лексем. UML-диаграмма интерпретатора языка запросов представлена на рисунке 5.



**Рисунок 5 ‒ Диаграмма классов интерпретатора языка запросов**

### Программная реализация лексического анализа

Классы, из которых состоит лексический анализатор, перечислены в таблице 17.

Таблица 17 **‒** Классы лексического анализатора

|  |  |
| --- | --- |
| Класс | Описание |
| Message | хранит какие-либо сообщения, созданные компилятором |
| Position | представляет координаты кодовой точки |
| Fragment | представляет координаты фрагмента программы |
| Token | базовый класс лексического домена, от которого наследуются все остальные классы, представляющие токены, соответствующие разным лексическим доменам |
| DomainTag | перечисление, определяющее соответствие между всеми синтаксическими доменами и целочисленными константами |
| Compiler | класс с изменяемыми таблицами: таблицей имен и таблицей сообщений |
| Parse | класс, содержащий вспомогательные функции для проверки корректности целочисленных констант в разных системах счисления |
| Scanner | класс-итератор, осуществляющий перебор токенов |

В таблице 18 представлены классы, наследующиеся от класса Token и описывающие токены различных лексических доменов.

Таблица 18 **‒** Классы лексических доменов

|  |  |
| --- | --- |
| Класс | Описание |
| IdentToken | токен, описывающий идентификаторы |
| StringToken | токен, описывающий строковые константы |
| CommentToken | токен, описывающий комментарии |
| NumberToken | токен, описывающий целочисленные константы |
| SpecToken | токен, описывающий специальные символы |
| KeywordToken | токен, описывающий ключевые слова |

Основной функцией класса Scanner, да и всего лексического анализатора является NextToken(), которая выполняет поиск следующего токена во входной цепочке символов. Рассмотрим принцип ее работы. В классе Scanner три приватных поля, описание которых дано в таблице 19.

Таблица 19 **‒** Поля класса Scanner

|  |  |  |
| --- | --- | --- |
| Доступ | Поле | Описание |
| private | Compiler compiler | * ссылка на класс Compiler для сохранения в нем сообщений и идентификаторов |
| private | List<Fragment> comments | список комментариев |
| private | Position cur | позиция текущего символа из входной цепочки |

Для того, что получить само значение символа у класса Position есть свойство CP, которое всегда возвращает значение текущего символа или -1, в случае, если достигнут конец программы.

Таким образом, функция NextToken() работает следующим образом: символы входной цепочки поочередно перебираются в цикле до тех пор, пока не будет достигнут конец программы. Все пробельные символы пропускаются. Для каждого непробельного символа происходит поиск наиболее длинной лексемы, которая может начинаться с текущего символа.

Поиск лексем осуществляется по следующему алгоритму:

* проверяются все специальные символы;
* происходит поиск комментариев;
* производится поиск целочисленных констант в разных системах счисления;
* все ключевые слова, целочисленные константы и идентификаторы.

В случае если никакая лексема не была найдена, то выдается ошибка компиляции с указанием позиции начала поиска лексемы.

### Программная реализация синтаксического анализатора

Основной класс синтаксического анализатора – SyntaxAnalyzer, его поля описаны в таблице 20.

Таблица 20 **‒** Поля синтаксического анализатора

|  |  |  |
| --- | --- | --- |
| Доступ | Поле | Описание |
| private | int curr\_token\_ind | индекс текущего токена |
| private | List<Token> tokens | список токенов, которые нужно проанализировать |

Вспомогательные методы синтаксического анализатора описаны в таблице 21.

Таблица 21 **‒** Методы синтаксического анализатора

|  |  |
| --- | --- |
| Метод | Описание |
| NextToken() | инкрементирует индекс текущего токена |
| PrevToken() | возвращает значение предыдущего токена |
| CurrentToken() | возвращает значение текущего токена |
| AfterToken() | возвращает значение следующего токена |
| Check(DomainTag tag) | проверяет, является ли текущий токен токеном типа Tag и если нет, то возвращает ошибку |
| RaiseError(String message) | создает исключение и выводит сообщение, в котором указана ожидаемая лексема |

Начало выполнения синтаксического анализа заключается в вызове функции Start(), которая с помощью метода рекурсивного спуска проверяет корректность синтаксической последовательности. В случае, если ни одна синтаксическая конструкция не подходит, то возвращается ошибка. Рассмотрим последовательность, в которой выполняется рекурсивный вызов функций. В листингах не отображены листовые функции ParseCreateDatabase(), ParseDropDatabase() ParseTruncateDatabase(), ParseInsert(), ParseDelete(), ParseSelect(), ParseImport() и ParseFactor(), т.к. они не содержат ветвления, а только проверяют корректность следования лексем. Кроме того, они содержат семантическую логику, которая будет рассмотрена в следующей главе.

Начало работы программы:

|  |
| --- |
| public void Start()  {  ParseProgram();  } |

Анализ последовательности запросов:

|  |
| --- |
| public void ParseProgram()  {  NextToken();  while (CurrentToken().Tag != DomainTag.END\_OF\_PROGRAM)  {  if (CurrentToken().Tag == DomainTag.SEMICOLON)  NextToken();  try  {  ParseQuery();  }  catch (Exception ex)  {  Console.WriteLine(ex.Message);  NextToken();  }  }  } |

Анализ типа запроса:

|  |
| --- |
| public void ParseQuery()  {  switch (CurrentToken().Tag)  {  case DomainTag.CREATE\_KEYWORD:  ParseCreateDatabase();  break;  case DomainTag.DROP\_KEYWORD:  ParseDropDatabase();  break;  case DomainTag.TRUNCATE\_KEYWORD:  ParseTruncateDatabase();  break;  case DomainTag.INSERT\_KEYWORD:  ParseInsert();  break;  case DomainTag.DELETE\_KEYWORD:  ParseDelete();  break;  case DomainTag.SELECT\_KEYWORD:  ParseSelect();  break;  case DomainTag.IMPORT\_KEYWORD:  ParseImport();  break;  default:  RaiseError("Unexpected query");  break;  }  } |

Анализ запроса SELECT:

|  |
| --- |
| private void ParseSelect()  {  Check(DomainTag.SELECT\_KEYWORD);  if (CurrentToken().Tag == DomainTag.STAR)  Check(DomainTag.STAR);  else  ParseSelectionCond();  Check(DomainTag.FROM\_KEYWORD);  Check(DomainTag.IDENT);  } |

Анализ SELECT-выражения:

|  |
| --- |
| private void ParseSelectionCond()  {  ParseVertexList();  Check(DomainTag.DOT);  ParseFunctions();  } |

Анализ списка исходных вершин:

|  |
| --- |
| private void ParseVertexList()  {  Check(DomainTag.LPARENT);  while (CurrentToken().Tag != DomainTag.RPARENT &&  CurrentToken().Tag != DomainTag.END\_OF\_PROGRAM)  {  Check(DomainTag.STRING);  if (CurrentToken().Tag != DomainTag.RPARENT)  Check(DomainTag.COMMA);  }  if (CurrentToken().Tag == DomainTag.END\_OF\_PROGRAM)  RaiseError("Unexpected end of program");  Check(DomainTag.RPARENT);  } |

Анализ последовательности выполнения аналитических функций:

|  |
| --- |
| private void ParseFunctions()  {  while (CurrentToken().Tag == DomainTag.NEIGHBORS\_KEYWORD ||  CurrentToken().Tag == DomainTag.INTERSECTION\_KEYWORD ||  CurrentToken().Tag == DomainTag.SURRNETWORK\_KEYWORD ||  CurrentToken().Tag == DomainTag.ALLPATHS\_KEYWORD ||  CurrentToken().Tag == DomainTag.COMMUNITIES\_KEYWORD)  {  NextToken();  if (CurrentToken().Tag != DomainTag.DOT)  break;  Check(DomainTag.DOT);  }  } |

### Программная реализация семантического анализа

Класс SyntaxAnalyzer содержит дополнительные приватные поля, отвечающие за семантический анализ. Они описаны в таблице 22.

Таблица 22 **‒** Поля семантического анализатора

|  |  |  |
| --- | --- | --- |
| Доступ | Поле | Описание |
| private | Compiler compiler | ссылка на класс Compiler для доступа к таблице идентификаторов |
| private | GraphAPI graph | ссылка на класс GraphAPI для доступа к интерфейсу графового хранилища |
| private | Dictionary  <string, string> ids | словарь, позволяющий получить путь к экземпляру графового хранилища по его идентификатору |

Кроме того, реализованы вспомогательные функции, которые описаны в таблице 23.

Таблица 23 **‒** Функции семантического анализатора

|  |  |
| --- | --- |
| Метод | Описание |
| String CheckString() | проверяет, является ли текущая лексема строкой. Если да, возвращает эту строку, иначе генерирует сообщение об ошибке |
| Int32 CheckNumber() | проверяет, является ли текущая лексема числом. Если да, возвращает это число, иначе генерирует сообщение об ошибке |
| String CheckIdent(bool check\_existence = false) | проверяет, является ли текущая лексема идентификатором. Если да, проверяет существования базы данных с таким идентификатором и возвращает строковое значение, иначе генерирует сообщение об ошибке. Кроме того, можно передать необязательный параметр true и тогда будет проведена проверка на существование такого идентификатора базы данных |
| void Stop() | сохраняет список словарь идентификаторов на жесткий диск после окончания работы интерпретатора языка запросов |

Для функции создания экземпляра графовой базы нужно добавить идентификатор БД в список идентификаторов, если его там не существует и после этого вызвать метод CreateDB(string database\_path) из класса GraphAPI:

|  |
| --- |
| database\_path = CheckString();  if (ids.ContainsKey(ident))  RaiseError(String.Format("Database id {0} already exists", ident));  ids.Add(ident, database\_path);  graph.CreateDB(database\_path); |

В функции удаления экземпляра базы данных просто вызывается метод RemoveDB(string database) из класса GraphAPI и проверяется его успешное выполнение, иначе генерируется ошибка (такое возможно в случае внешнего воздействия на БД, например, один их файлов удалили):

|  |
| --- |
| string ident = CheckIdent(true);  if (!graph.RemoveDB(ids[ident]))  {  RaiseError("Error while deleting database");  } |

В функции очистки базы данных просто вызывается метод TruncateDB(string database) из класса GraphAPI:

|  |
| --- |
| graph.TruncateDB(ids[ident]); |

В функции вставки вершины вызывается метод InsertNode(string database, string val):

|  |
| --- |
| string ident = CheckIdent(true);  string val = CheckString();  graph.InsertNode(ids[ident], val); |

В функции вставки ребра сначала происходит проверка наличия временного параметра, а затем выполняется процедура, которая связанная с особенностью строения графового хранилища. Создается транзакция импорта данных, внутри этой транзакции вставляется ребро и транзакция отправляется на выполнение:

|  |
| --- |
| string ident = CheckIdent(true);  string a\_node = CheckString();  string b\_node = CheckString();  DateTime time = DateTime.Now;  if (CurrentToken().Tag == DomainTag.TIME\_KEYWORD)  {  NextToken();  string time\_str = CheckString();  if (!DateTime.TryParse(time\_str, out time))  time = DateTime.Now;  }  CGTimeStep step = new CGTimeStep();  graph.CreateImportTransaction(ids[ident], step);  graph.ImportPair(ids[ident], a\_node, b\_node, time);  Guid op\_id = graph.CommitImportTransaction(ids[ident]); |

Функции удаления ребер, вершин и функция импорта данных работают аналогичным образом, так что не будем на них подробно останавливаться. Рассмотрим работу оператора SELECT. В случае, если нужно получить все данные из графового хранилища, то вызывается метод SelectAll(string database) из класса GraphAPI, который возвращает граф в виде списка ребер:

|  |
| --- |
| var edges = graph.SelectAll(ids[ident]); |

Если же применяется последовательность аналитических функций, то сначала функция ParseVertexList() получает список вершин, к которым применяются операции:

|  |
| --- |
| private List<string> ParseVertexList()  {  List<string> nodes = new List<string>();  Check(DomainTag.LPARENT);  while (CurrentToken().Tag != DomainTag.RPARENT &&  CurrentToken().Tag != DomainTag.END\_OF\_PROGRAM)  {  string str = CheckString();  nodes.Add(str);  if (CurrentToken().Tag != DomainTag.RPARENT)  Check(DomainTag.COMMA);  }  if (CurrentToken().Tag == DomainTag.END\_OF\_PROGRAM)  RaiseError("Unexpected end of program");  Check(DomainTag.RPARENT);  return nodes;  } |

После этого функция ParseFunctions (string database, List<string> nodes) разбирает саму последовательность функций с параметрами. Это происходит с помощью цикла:

|  |
| --- |
| while ((CurrentToken().Tag == DomainTag.NEIGHBORS\_KEYWORD ||  CurrentToken().Tag == DomainTag.INTERSECTION\_KEYWORD ||  CurrentToken().Tag == DomainTag.SURRNETWORK\_KEYWORD ||  CurrentToken().Tag == DomainTag.ALLPATHS\_KEYWORD ||  CurrentToken().Tag == DomainTag.COMMUNITIES\_KEYWORD)  && CurrentToken().Tag != DomainTag.END\_OF\_PROGRAM) |

Для каждой функции создается класс, описывающий параметры (MQNeighborsParameters, MQIntersectionParameters или MQSurroundNetworkParameters) и вызываются соответствующие функции анализа из класса GraphAPI:

|  |
| --- |
| if (saved\_tag == DomainTag.NEIGHBORS\_KEYWORD)  {  MQNeighborsParameters parameters = new MQNeighborsParameters();  parameters.Depth = depth;  parameters.Nodes = nodes;  op\_id = graph.Analyze(db, "neighbors", parameters);  }  else if (saved\_tag == DomainTag.INTERSECTION\_KEYWORD)  {  MQIntersectionParameters parameters =  new MQIntersectionParameters();  parameters.Depth = depth;  parameters.Nodes = nodes;  op\_id = graph.Analyze(db, "intersection", parameters);  }  else if (saved\_tag == DomainTag.SURRNETWORK\_KEYWORD)  {  MQSurroundNetworkParameters parameters =  new MQSurroundNetworkParameters();  parameters.Depth = depth;  parameters.Nodes = nodes;  op\_id = graph.Analyze(db, "surround\_network", parameters);  } |

Таким образом, каждая следующая функция применяется к результатам работы предыдущей. Промежуточные результаты сохраняются в файлах, которые находятся там же, где и экземпляр графовой базы данных.

# Тестирование

Тестирование разработанного хранилища биллинговой информации проводилось на двух различных серверах S\_Small и S\_Big. Конфигурация серверов указана в таблице 24.

Таблица 24 **‒** Конфигурация серверов

|  |  |  |
| --- | --- | --- |
| Параметр | Конфигурация | |
| S\_Small | S\_Big |
| CPU | Intel Core i5 3.1 GHz  (4 cores) | Intel Xeon E5 2.4 GHz  (16 cores) |
| RAM | 16GB | 64GB |
| HDD | 500GB, 7200 rpm | 2TB, 7200 rpm |
| OS | Windows Server 2008 R2 Enterprise x64 | Windows Server 2008 R2 Enterprise x64 |

При загрузке данных, сначала проводилась загрузка исходного биллинга, содержащего в себе большой объем данных, после чего начиналась дозагрузка биллинговых данных меньшего объема. Биллинги содержались во множестве небольших текстовых файлов разного размера. Точное количество связей в биллингах могло различаться в пределах нескольких сотен тысяч, но оно точно не было меньше указанного в тестировании значения.

На сервере S\_Small был запущен первоначальный импорт исходного биллинга объемом 500 млн связей, содержащего в себе биллинговые данные за 7 дней. После чего, раз в сутки на протяжении еще 7-ми дней проходила дозагрузка биллинговых данных объемом 100 млн связей. Результаты тестирования импорта данных на сервере S\_Small представлены на рисунке 6.

Аналогично серверу S\_Small, на сервере S\_Big был запущен первоначальный импорт исходного биллинга объемом 7 млрд связей, содержащего в себе 30 дней биллинговой истории. После чего раз в сутки на протяжении 7-ми дней проходила дозагрузка биллинговых данных объемом 200 млн связей. Результаты тестирования импорта данных на сервере S\_Big представлены на рисунке 7.

**Рисунок 6 ‒ Импорт данных S\_Small**

**Рисунок 7 ‒ Импорт данных S\_Big**

После окончания накопления биллинговых данных за 7 дней на серверах S\_Small и S\_Big, на них были запущены все основные графовые аналитические операции:

* Поиск соседей вершины (Q1).
* Поиск попарных пересечений (Q2).
* Поиск окружающей сети (Q3).
* Поиск путей (Q4).
* Поиск сообществ (Q5).

Вызов операций осуществлялся с помощью языка запросов к графовому хранилищу. Описание использования языка запросов приведено в приложении А. Параметры для аналитической операции по поиску сообществ подбирались исходя из конкретной задачи. Общие параметры аналитических операций указаны в таблице 25, а результаты тестирования представлены на рисунке 8.

Таблица 25 **‒** Параметры аналитических операций

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Параметр | Аналитическая операция | | | | |
| Q1 | Q2 | Q3 | Q4 | Q5 |
| Исходное количество номеров | 1 | 10 | 10 | 2 | - |
| Глубина поиска | 1 | 3 | 3 | 6 | - |
| Период времени | 3 дня | 3 дня | 3 дня | 3 дня | 10 дней |

**Рисунок 8 ‒ Время выполнения аналитических операций**

В случае тестирования на серверах S\_Small и S\_Big первоначальный импорт данных занял больше времени, чем последующие, из-за крупного размера исходных данных. При этом, общая скорость импорта остается практически идентичной по мере увеличения объема хранилища и не опускается ниже 25000 тысяч соединений в секунду, что значительно превышает критическую. Можно наблюдать некоторую неравномерность в скорости импорта биллингов одинаковых объемов, это объясняется небольшими различиями в размерах загружаемых данных и различиями в их графовой структуре. Количество занимаемого места растет линейно по мере загрузки. Время выполнения аналитических операций так же полностью вписывается в рамки критических значений.

Результаты тестирования демонстрируют полное соответствие реализованного хранилища с заявленными требованиями объемов и скорости обработки биллинговой информации. Возможно использование данной системы хранения как на обычных пользовательских ПК (S\_Small), так и на промышленных серверных решениях (S\_Big).

# Технико-экономическое обоснование

## Экономическая проблема

За последние годы, в связи с ростом активности абонентов сотовых сетей, все чаще требуется производить сложный анализ биллинговых данных. Объемы этих данных растут с каждым годом, из-за чего на существующих программно-аппаратных комплексах (далее ПАК) любая задача анализа выполняется дольше. Выполнением таких операций и обработкой их результатов занимаются специализированные компании, которым требуется все больше времени на выполнение одних и тех же задач. Доходы таких компаний напрямую зависят от количества поступивших к ним заказов на обработку биллинговых данных. А заказы поступают только в том случае, если компания осуществляет не меньше заданного количество запросов за единицу времени.

В связи с ростом количества данных, производительность (количество обработанных задач за единицу времени) снижается, из-за чего приходится закупать дополнительные ПАК, чтобы распараллелить выполнение задач и устранить падение производительности. Но из-за постоянной закупки таких комплексов и увеличения количества сотрудников, которые их обслуживают, снижаются доходы всей компании, занимающейся анализом биллинговых данных.

В дипломной работе разрабатывается программный продукт (далее ПП) для ПАК, который позволил бы выполнять операции анализа биллинговых данных с одинаковой скоростью, независимо от того, каков объем обрабатываемых данных. Это бы позволило тому же количеству сотрудников выполнять одинаковое количество запросов при постоянном росте количества данных. Таким образом, один раз покупая такой комплекс, компания решает проблему падения доходов из-за роста объема данных.

## Расчет затрат

Проведем вычисление все основных затрат на реализацию проекта. Конечной целью расчетов является вычисление следующих значений:

* Трудоемкость разработки (чел.-дней)
* Количество исполнителей (шт.)
* Заработная плата (руб.)
* Стоимость ПП (руб.)

### Трудоемкость разработки

Разработка ПП разбивается на 5 этапов [31]: техническое задание, эскизный проект, технический проект, рабочий проект, разработка документации (внедрение). Удельный вес каждого этапа и квалификация его исполнителя приведены в таблице 26.

Таблица 26 **‒** Удельный вес каждого этапа разработки ПП

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Наименование этапа | Исполнитель | Удельный вес, % |
| 1 | Техническое задание | Старший инженер | 10 |
| 2 | Эскизный проект | Старший инженер | 10 |
| 3 | Технический проект | Инженер | 25 |
| 4 | Рабочий проект | Инженер | 20 |
| 5 | Разработка документации | Старший инженер | 35 |
| ИТОГО | | | 100 |

Трудоемкость разработки программного продукта может быть определена как сумма величин трудоемкости выполнения отдельных этапов разработки программного продукта из выражения [31]:

+ +,

где - трудоемкость разработки технического задания на создание программного продукта; - трудоемкость разработки эскизного проекта; - трудоемкость разработки технического проекта программного продукта; - трудоемкость разработки рабочего проекта программного продукта; - трудоемкость разработки программной документации. Поскольку техническое задание сформулировано заказчиком, трудоемкость первого этапа можно не учитывать. Учитывая данные таблицы 1, можно записать:

Оценим трудоемкость всех этапов через один, а именно через оценку трудоемкости (п. 5 в таблице 1). Пояснительная записка диплома содержит 100 страниц формата А4. На заполнение одной такой страницы старший инженер тратит в среднем 3.7 часа [33]. Следовательно:

Отсюда получаем, что трудоемкость разработки программного

продукта составляет:

В таблице 27 приведены трудоемкости каждого этапа разработки ПП.

Таблица 27 **‒** Трудоемкость каждого этапа разработки ПП

|  |  |  |
| --- | --- | --- |
| № п/п | Наименование этапа | Трудоемкость, чел.-дни |
| 2 | Эскизный проект | 12 |
| 3 | Технический проект | 30 |
| 4 | Рабочий проект | 31 |
| 5 | Разработка документации | 46 |
| ИТОГО | | 119 |

### Исполнители

Количество исполнителей N, необходимых для выполнения определенного этапа разработки ПП вычисляется с помощью следующей формулы [31]:

где t – трудоемкость этапа, а F – заданное временное требование к этапу.

Разработка ПП велась с 10 марта 2015 года по 31 мая 2015 года. В этом периоде содержится 56 рабочих дней (12 недель). По данным в таблице 26 составляется таблица 28 временных требований (в днях) по каждому из этапов разработки.

Таблица 28 **‒** Временные требования каждого этапа разработки ПП

|  |  |  |
| --- | --- | --- |
| № п/п | Наименование этапа | Разработка этапа, дни |
| 2 | Эскизный проект | 8 |
| 3 | Технический проект | 15 |
| 4 | Рабочий проект | 16 |
| 5 | Разработка документации | 17 |
| ИТОГО | | 56 |

Для реализации проекта потребуются 3 старших инженера и 2 простых инженера. Ленточный график выполнения работ представлен в таблице 29.

Таблица 29 **‒** Ленточный график выполнения работ

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| № этапа | Содержание работ | Продолжительность, раб. дни | Исполнители | | Календарные дни | | | | | | | | | | | |
| Категория | Число | 10.03-15.03 | 16.03-22.03 | 23.03-29.03 | 30.03-05.04 | 06.04-12.04 | 13.04-19.04 | 20.04-26.04 | 27.04-03.05 | 04.05-10.05 | 11.05-17.05 | 18.05-24.05 | 25.05-31.05 |
| Количество рабочих дней | | | | | | | | | | | |
| 4 | 5 | 5 | 5 | 5 | 5 | 5 | 4 | 4 | 4 | 5 | 5 |
| 2 | Эскизный проект | 8 | Старший инженер | 2 | 4 | 4 |  |  |  |  |  |  |  |  |  |  |
| 3 | Технический проект | 15 | Инженер | 2 |  | 1 | 5 | 5 | 4 |  |  |  |  |  |  |  |
| 4 | Рабочий проект | 16 | Инженер | 2 |  |  |  |  | 1 | 5 | 5 | 4 | 1 |  |  |  |
| 5 | Разработка технической документации | 17 | Старший инженер | 3 |  |  |  |  |  |  |  |  | 3 | 4 | 5 | 5 |

### Заработная плата

В общем случае расчет суммы основной заработной платы исполнителей ведется по формуле [31]:

где п - число исполнителей; - однодневный размер оплаты труда 1-го исполнителя, руб./дн.; - количество рабочих дней, отработанных 1-м исполнителем при разработке программного продукта.

При восьмичасовом рабочем дне дневной оклад исполнителя рассчитывается с помощью соотношения [32]:

где - месячный оклад, - месячный фонд рабочего времени.

Месячный фонд рассчитывается по формуле:

где - рабочее время в году, часы. Согласно производственному календарю рабочее время в 2015 году при 40 часовой рабочей неделе равно 1971 часам. Поэтому фонд времени в одном месяце 2015 года составляет:

Дополнительная заработная плата определяется по формуле [31]:

где - коэффициент отчислений на дополнительную заработную плату, = 0.1. В данном случае коэффициент учитывает не только оплату непроработанного времени, но и все компенсационные и стимулирующие надбавки к тарифам.

Заработная плата руководителей и другого персонала определяется по формуле [31]:

где - коэффициент отчислений на заработную плату руководителей и другого персонала, = 0,5.

Исходя из статистических значений средней заработной платы [34], заработные платы исполнителей представлены в таблице 30.

Таблица 30 **‒** Оклады и заработные платы исполнителей

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Исполнители | Месячный оклад, руб | Дневная з/п, руб. | Продолж. работы, дн. | Основная з/п, руб | Доп. з/п, руб |
| Старший инженер | 90000 | 4390 | 25 | 294130 | 29413 |
| Инженер | 60000 | 2930 | 31 | 181660 | 18166 |
| ИТОГО | | | | 523370 | |

Таким образом, суммарно на заработную плату исполнителям необходимо 523370 рублей. Кроме того, отдельно учитывается зарплата руководителям, которая составляет 261685 рубля. В итоге, общие расходы на заработную плату составят 785055 рублей.

### Стоимость продукта

Стоимость программного продукта рассчитывается по следующей формуле [31]:

Где – стоимость ПП, – себестоимость ПП, – желаемая для исполнителя рентабельность ПП. В нашем случае рентабельность составляет 40% и формула будет иметь следующий вид:

Затраты на разработку программного продукта (себестоимость ПП) могут быть определены по формуле [31, 32]:

+ +

где См - материалы и комплектующие, используемые в процессе разработки и производства программного продукта; Сосн - основная заработная плата; Сдоп - дополнительная заработная плата; Срук - заработная плата руководителей и другого персонала; Сос - отчисления на социальные нужды; Сао - амортизация оборудования; Спр - прочие затраты.

Как правило, на стадии принятия управленческого решения о проектировании программного продукта известна только статистика о структуре затрат на создание программного продукта [31]. Рассчитав затраты на оплату труда всех исполнителей при создании программного продукта, можно оценить затраты по остальным статьям сметы, а также и общую сумму затрат и в итоге определить цену проектируемого программного продукта. Структура сметы затрат на разработку программного продукта представлена в таблице 31.

Таблица 31 **‒** Структура сметы затрат на разработку программного продукта

|  |  |  |
| --- | --- | --- |
| № п/п | Наименование статьи | Удельный вес, % |
| 1 | Материальные | 10 |
| 2 | Заработная плата | 35 |
| 3 | Отчисления на социальные нужды | 10 |
| 4 | Амортизационные отчисления | 30 |
| 5 | Прочие затраты | 15 |

Таким образом, можем вычислить себестоимость программного продукта:

Значит, окончательная стоимость программного продукта будет составлять:

## Вывод

В данном разделе дипломной работы было проведено технико-экономическое обоснование проекта. В ходе обоснования была обозначена экономическая выгода решаемой задачи и установлены некоторые экономические характеристики, основные из них следующие:

* Трудоемкость работы
* Количество исполнителей : 3 старших инженера и 2 простых инженера.
* Суммарные расходы на зарплату .
* Общая стоимость программного продукта

Будем считать, что стоимость программно-аппаратного комплекса (как со старым ПП, так и с новым ПП) составляет 500000 рублей, зарплата аналитика, обслуживающего этот комплекс составляет 720000 рублей в год (60000 рублей в месяц), а стоимость переобучения аналитика для работы с новым ПП равна 30000 рублей. Тогда можно рассчитать выгоду использования нового ПП в долгосрочной перспективе. Расходы за год при использовании старого программного продукта и нового представлены в таблице 32.

Таблица 32 **‒** Затраты за год при использовании ПП

|  |  |  |
| --- | --- | --- |
| Год использования | Затраты при использовании нового ПП, руб | Затраты при использовании старого ПП, руб |
| 2015 | 1250000 | 1220000 |
| 2016 | 1250000 | 2440000 |
| 2017 | 1250000 | 3660000 |
| 2018 | 1250000 | 4880000 |
| 2019 | 1250000 | 6100000 |

# Заключение

В дипломной работе спроектирована и реализована графовая база данных, которая позволяет решить задачу хранения и анализа биллинговой информации. База данных позволяет хранить заявленные объемы данных в рамках одного сервера. В систему интегрированы все необходимые аналитические операции, встречающиеся при обработке биллинговых данных.

Система реализована в виде приложения командной строки Windows. Данное приложение предоставляет интерфейс доступа в виде WCF-службы. Кроме того, для удобного доступа к данным пользователем, реализован специализированный язык графовых запросов. Система способна работать со многими экземплярами графовых хранилищ одновременно.

Тестирование показало, что операции импорта выполняются за приемлемое время независимо от объемов загружаемых данных. А время выполнения операций анализа не выходит за рамки заявленных критических значений. Таким образом, на поставленных задачах реализованная графовая база данных способна показывать лучшие результаты, чем уже существующие продукты.

Стоит отметить слабые стороны реализованной базы данных, устранение которых является наиболее приоритетным развитием проекта:

* слабая отказоустойчивость, отсутствие поддержки транзакций;
* отсутствие графического интерфейса, в том числе визуализатора графа.

# Список литературы

1. Прамодкумар Садаладж, Мартин Фаулер. NoSQL. Новая методология разработки нереляционных баз данных, М: «Вильямс». – 2013.
2. R. Angles, C. Gutierrez. Survey of Graph Database Models. ACM Computing Surveys. In *Proceedings of the 2010 international conference on Web-age information management*, WAIM '10. – 2008.
3. Salim Jouili, Valentin Vansteenberghe, An empirical comparison of graph databases. In *Social Computing International Conference*, SocialCom '13. IEEE Computer Society. – 2013.
4. Шериф Сакр. Обработка больших объемов графовых данных: путеводитель по современным технологиям. IBM DeveloperWorks, 2013. URL: http://www.ibm.com/developerworks/ru/library/os-giraph/
5. Neo4J Graph Database. URL: http://www.neo4j.org/
6. Neo4J documentation. URL: http://neo4j.com/docs/2.1.6/
7. D. Dominguez-Sal, P. Urbon-Bayes, A. Gimenez-Vano, S. Gomez-Villamor, N. Martınez-Bazan, J.L. Larriba-Pey. Survey of graph database performance on the HPC scalable graph analysis benchmark. Springer-Verlag. – 2010.
8. Peter Macko, Daniel Margo, Margo Seltzer. Performance Introspection of Graph Databases. In *6th International Systems and Storage Conference*, SYSTOR '13. Harvard University. – 2013.
9. Бартенев М.В., Вишняков И.Э. Использование графовых баз данных в целях оптимизации анализа биллинговой информации. Инженерный журнал: наука и инновации, 2013, вып. 11. URL: http://engjournal.ru/catalog/it/hidden/1058.html
10. Sparksee Graph Database. URL: http://www.sparsity-technologies.com/
11. Sparksee manual. URL: http://www.sparsity-technologies.com/UserManual/
12. Orient Graph Database. URL: http://www.orientechnologies.com/orientdb/
13. Titan Graph Database. URL: http://thinkaurelius.github.io/titan/
14. Titan Provides Real-Time Big Graph Data. URL: http://thinkaurelius.com/2012/08/06/titan-provides-real-time-big-graph-data/
15. Sones Graph Database. URL: https://github.com/sones/sones
16. Cypher Query Language. URL: http://neo4j.com/developer/cypher-query-language/
17. Gremlin Query Language. URL:

https://github.com/thinkaurelius/titan/wiki/Gremlin-Query-Language

1. Blueprints interfaces. URL: http://blueprints.tinkerpop.com/
2. GQL. URL:

http://github.com/downloads/sones/sones/GQL\_cheatsheet\_latest.pdf

1. R. Angles. A comparison of current graph database models. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops*, ICDEW '12. IEEE Computer Society. – 2012.
2. OrientDB graph API manual: URL:

http://www.orientechnologies.com/docs/last/orientdb.wiki/Graph-Database-Tinkerpop.html

1. Titan docs. URL: https://github.com/thinkaurelius/titan/wiki
2. Sones documentation. URL:

https://github.com/sones/sones/tree/master/Documentation

1. Navathe, Ramez Elmasri, Shamkant B. Fundamentals of database systems. – N.J.: «Pearson Education». – 2010.
2. CSharpTest.net BPlusTree 14. URL: http://csharptest.net/projects/bplustree/
3. STSdb 4. URL: http://stssoft.com/
4. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: «Питер». – 2007.
5. MSDN. Windows Comminication Foundation. URL: https://msdn.microsoft.com/ru-ru/library/bb386386.aspx
6. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы: принципы, технологии и инструментарий. – М: «Вильямс». ‒ 2008.
7. MSDN. SortedSet: Электронный ресурс. - Режим доступа: http://msdn.microsoft.com/ru-ru/library/dd412070.aspx
8. Сажин Ю.Б., Самохин С.В. Выполнение организационно­-экономической части дипломного проекта по разработке и использованию программного продукта: Методическое пособие. - М.: Изд-во МГТУ им. Н.Э. Баумана. ‒ 2006.
9. Меняев М.Ф., Бышовец Б.Д., Пряников И.Ф. Организационно­-экономическая часть дипломных проектов, направленных на разработку программного обеспечения. Учебное пособие. - М.: Изд-во МГТУ им. Н.Э. Баумана. ‒ 2007.
10. Процесс создания документации пользователя программного средства. - ГОСТ Р ИСО МЭК 15910-2002.
11. Федеральная служба государственной статистики. Официальная статистика, заработная плата. URL: http://www.gks.ru/

# Приложение А. Руководство пользователя

## A.1. Системные требования

В этом разделе рассмотрены минимальные системные требования, необходимые для запуска графовой базы данных. Системные требования аппаратного обеспечения представлены в таблице А.1.

Таблица А.1 **‒** Мин. требования к аппаратному обеспечению

|  |  |
| --- | --- |
| Характеристика | Минимальное значение |
| Процессор | Intel Core 2 Duo 2х2.0 GHz |
| Оперативная память | 4 GB |
| Жесткий диск | 120 GB |

Системные требования программного обеспечения представлены в таблице А.2.

Таблица А.2 **‒** Мин. требования к программному обеспечению

|  |  |
| --- | --- |
| Характеристика | Минимальное значение |
| Операционная система | Microsoft Windows 7 |
| Установленное ПО | .NET Framework 4.0  C++ Redistributable libs 2012 |

Кроме того, для корректной работы WCF-службы доступа к графовой базе данных, межсетевой экран операционной системы не должен блокировать порт 8732.

## A.2. Составные части

Файлы, из которых состоит графовая база данных, представлены в таблице А.3.

Таблица А.3 **‒** Составные части приложения

|  |  |
| --- | --- |
| Файл | Описание |
| Graph.Client.exe | основной исполняемый файл, запускающий WCF-службу и консольное приложение ввода запросов к графовой базе данных. |
| Graph.Client.exe.config | конфигурационный файл WCF-службы. |
| graph\_analysis.log | файл с логами. |
| NT/NT.GraphStorage.dll | DLL-библиотека, содержащая всю внутреннюю инфраструктуру взаимодействия с данными и алгоритмов анализа. |
| NT/SortFiles.dll | DLL-библиотека, позволяющая сортировать данные во внешней памяти. |
| \_libs/log4net.dll | DLL-библиотека, ответственная за ведение лог-файлов. |

## A.3. Установка и запуск

Установка и запуск графовой базы данных осуществляются в следующей последовательности:

1. Скопировать в нужную директорию все файлы, перечисленные в разделе А.2, сохранив их иерархию.
2. Запустить исполняемый файл Graph.Client.exe от имени администратора.
3. Убедиться, что запустилось консольное окно Windows и в нем отобразился процесс загрузки системы и список конечных точек WCF-службы:

|  |
| --- |
| Loading database manager...  It may takes a few seconds  Service is up and running with followong endpoints:  -> http://localhost:8732/GraphAPI/mex  -> net.tcp://localhost/GraphAPI/  -> http://localhost:8732/GraphAPI/web  Starting operation queue...  Queue started!  GraphDB> |

Последняя строка является командной и в ней можно запускать запросы к графовой базе данных. В случае возникновения ошибки при инициализации службы, описание ошибки будет выведено на экран командной строки.

## A.4. Выполнение запросов к графовой базе данных

Рассмотрим все виды запросов к графовому хранилищу с описанием параметров и примерами каждого запроса.

### A.4.1. CREATE DATABASE

Формат

*CREATE DATABASE database\_id [IN directory\_path]*

Описание

Создание экземпляра графовой базы данных.

Параметры

*database\_id* ‒ идентификатор экземпляра БД;

*directory\_path* ‒ строковое значение (если путь в запросе не указан, директорией по умолчанию считается корневой каталог программы, выполняющей запрос).

Пример

CREATE DATABASE database1 IN "c:\database1\"

Результатом работы будет созданная директорию database1 на диске С: с файлами экземпляра БД.

### A.4.2. DROP DATABASE

Формат

*DROP DATABASE database\_id*

Описание

Удаление экземпляра графовой базы данных.

Параметры

*database\_id* ‒ идентификатор экземпляра БД.

Пример

DROP DATABASE database1

Результатом работы будет удаление директории, соответствующей экземпляру БД database1.

### A.4.3. TRUNCATE

Формат

*TRUNCATE database\_id*

Описание

Удаление всех объектов из экземпляра графовой базы данных.

Параметры

*database\_id* ‒ идентификатор экземпляра БД.

Пример

TRUNCATE database1

Результатом работы будет удаление всех объектов из экземпляра БД database1.

### A.4.4. INSERT NODE

Формат

*INSERT NODE INTO database\_id VALUE node\_value*

Описание

Вставка вершины в базу данных.

Параметры

*database\_id* ‒ идентификатор экземпляра БД;

*node\_value* ‒ строковое значение, идентификатор вставляемой вершины*.*

Пример

INSERT NODE INTO database1 VALUE "0611"

Результатом работы будет добавление вершины «0611» в экземпляр БД database1.

### A.4.5. INSERT EDGE

Формат

*INSERT EDGE INTO database\_id VALUE source\_node, dest\_node [TIME date\_time]*

Описание

Вставка соединения в базу данных (возможно указание дополнительного параметра– временной метки ребра). В случае отсутствия одной из инцидентных вершин, она создается.

Параметры

*database\_id* ‒ идентификатор экземпляра БД;

*source\_node* ‒ строковое значение, идентификатор исходной вершины соединения;

*dest\_node* ‒ строковое значение, идентификатор конечной вершины соединения.

Пример

INSERT EDGE INTO database1 VALUE "0611", "900"

Результатом работы будет добавление соединения из вершины «0611» в вершину «900» в экземпляр БД database1.

### A.4.6. DELETE NODE

Формат

*DELETE NODE FROM database\_id VALUE node\_value*

Описание

Удаление вершины из базы данных.

Параметры

*database\_id* ‒ идентификатор экземпляра БД;

*node\_value* ‒ строковое значение, идентификатор удаляемой вершины*.*

Пример

DELETE NODE FROM database1 VALUE "900"

Результатом работы будет удаление вершины «900», из экземпляра БД database1.

### A.4.7. DELETE EDGE

Формат

*DELETE EDGE FROM database\_id VALUE source\_node, dest\_node* Описание

Удаление соединения из базы данных.

Параметры

*database\_id* ‒ идентификатор экземпляра БД;

*source\_node* ‒ строковое значение, идентификатор исходной вершины соединения;

*dest\_node* ‒ строковое значение, идентификатор конечной вершины соединения.

Пример

DELETE EDGE FROM database1 VALUE "0611", "900"

Результатом работы будет удаление соединения из вершины «0611» в вершину «900» в экземпляре БД database1.

### A.4.8. IMPORT FILE

Формат

*IMPORT FILE file\_path INTO database\_id*

Описание

Импорт файла с данными в базу данных.

Параметры

*file\_path* ‒ строковое значение, путь к файлу импорта;

*database\_id* ‒ идентификатор экземпляра БД.

Пример

IMPORT FILE "c:\test.txt" INTO database1

Результатом работы будет импорт данных из файла test.txt, в экземпляр БД database1.

### A.4.9. SELECT

Формат

*FROM database\_id*

*[DATE\_RANGE start\_date end\_date]*

*SELECT { \* | <selection\_cond> }*

*<selection\_cond> ::= <nodes\_list> . <functions>*

*<nodes\_list> ::= (node\_id [ ,…n ])*

*<functions> ::= <analysis\_function>[.<functions>]*

*<analysis\_function> ::=*

*{ NEIGHBORS(depth) |*

*INTERSECTION(depth) |*

*SURROUND\_NETWORK(depth) |*

*ALLPATHS(source\_node, dest\_node, depth) |*

*COMMUNITIES(min\_connect, max\_connect, min\_members, max\_members)}*

Описание

Извлечение данных из экземпляра графового хранилища по временному диапазону и с помощью аналитических операций.

Параметры общие

*database\_id* ‒ идентификатор экземпляра БД;

*start\_date* ‒ строковое значение, обозначающее начальную дату, с которой выполнить выборку данных;

*end\_date* ‒ строковое значение, обозначающее конечную дату, с которой выполнить выборку данных;

*node\_id* ‒ строковое значение, обозначающее идентификатор вершины.

Параметры операции NEIGHBORS

*depth* ‒ числовое значение, глубина поиска.

Параметры операции INTERSECTION

*depth* ‒ числовое значение, глубина поиска.

Параметры операции SURROUND\_NETWORK

*depth* ‒ числовое значение, глубина поиска.

Параметры операции ALLPATHS

*source\_node* ‒ строковое значение, обозначающее идентификатор начальной вершины;

*dest\_node* ‒ строковое значение, обозначающее идентификатор конечной вершины;

*depth* ‒ числовое значение, глубина поиска.

Параметры операции COMMUNITIES

*min\_connect* ‒ числовое значение, минимальная кратность ребер в сообществе;

*max\_connect* ‒ числовое значение, максимальная кратность ребер в сообществе;

*min\_members* ‒ числовое значение, минимальное количество членов в сообществе;

*max\_members* ‒ числовое значение, максимальное количество членов в сообществе.

Пример 1

FROM database1 DATE\_RANGE("01.01.2010", "01.01.2012") SELECT \*

Результатом работы будут все данные из экземпляра БД database1 в период с 01.01.2010 по 01.01.2012.

Пример 2

FROM database1 SELECT ().COMMUNITIES(5, 10, 30, 100)

Результатом работы будут все сообщества с заданными параметрами.

Пример 3

FROM database1 SELECT ("1", "2", "3").NEIGHBORS(2).ALLPATHS("4", "5", 3)

В ходе данного запроса будет выполнен поиск соседей до глубины два от исходных вершин "1" и "2", после чего в результате будет выполнен поиск путей между вершинами "4" и "5" длины не более трех.