

КАФЕДРА Системы обработки информации и управления

Система автоматического сбора информации о работе NoSQL баз данных

Нормоконтролер _____ Ю.Н.Кротов
(Подпись, дата) (И.О.Фамилия)

2025 г.

Аннотация

Расчётно-пояснительная записка квалификационной работы бакалавра содержит 82 страницы. С приложениями объем составляет 89 страниц. Работа включает в себя 5 таблиц и 38 иллюстраций. В процессе выполнения было использовано 36 источников.

Цель работы заключается в создании системы для взаимодействия с различными БД, созданными разными СУБД и анализа их работы, что позволяет сделать процесс получения и обработки информации из разных БД более удобным для пользователей.

В работе представлено три раздела: исследовательская часть, конструкторско-технологическая часть, тестирование.

В первой главе исследуются и подвергаются анализу СУБД, использующиеся в системе. Рассматривается их синтаксис запросов и особенности каждой СУБД, рассматривается общее строение озёр данных и обзор имеющихся, после чего производится сравнение с разрабатываемой системой. При сравнении отмечаются достоинства и недостатки, рассматриваются различные системы, которые решают аналогичную задачу и приведено их описание. Все найденные системы распределены на группы.

Во второй главе описывается строение разработанной системы, грамматика и описание команд на разработанном языке запросов к системе, описана работа парсера для данного языка, представлены разработанные UML диаграммы системы.

В заключительной главе представлены методы, способы, параметры и результаты проведённого тестирования.

Результатом работы являются работающая система для взаимодействия с различными БД и анализа их работы.

Пояснительная записка содержит 2 приложения.

Содержание

Обозначения и сокращения.....	5
Введение.....	6
1 ИССЛЕДОВАТЕЛЬСКАЯ ЧАСТЬ	7
1.1 Обзор СУБД, которые включены в систему	7
1.1.1 Виды СУБД.....	7
1.1.2 MongoDB.....	8
1.1.3 Neo4j	10
1.1.4 Cassandra	11
1.2 Сравнение с озером данных	13
1.2.1 Основная идея построения	13
1.2.2 Недостатки Data lake.....	15
1.2.3 Обзор платформ для создания и управления озёр данных	16
1.3. Группы методов взаимодействия различных СУБД	20
1.3.1 Узконаправленные системы.....	20
1.3.2 Ручная интеграция СУБД.....	23
1.3.3 Создание новой СУБД	28
1.3.4 Унифицировать имеющиеся модели БД.....	33
1.3.5 Взаимодействие в виде графа	38
2 КОНСТРУКТОРСКО-ТЕХНОЛОГИЧЕСКАЯ ЧАСТЬ	41
2.1 Описание взаимодействия модулей	41
2.2 Синтаксис и грамматика созданного языка запросов	43
2.3 Классы СУБД.....	46
2.4 Парсер запросов	47
2.5 UML диаграмма системы	51
3 ТЕСТИРОВАНИЕ	72
3.1 Функциональное тестирование.....	72
3.2 Модульное тестирование.....	74
3.3 Веб-тестирование производительности	75

Заключение	78
Список использованных источников	79
ПРИЛОЖЕНИЕ А	83
ПРИЛОЖЕНИЕ В ТЕХНИЧЕСКОЕ ЗАДАНИЕ	86

Обозначения и сокращения

В настоящей ВКР применяют следующие сокращения и обозначения:

СУБД – система управления базами данных.

БД – база данных.

Введение

В настоящее время наблюдается продолжающийся рост систем, поддерживающих огромный объем реляционных и нереляционных форм данных. Примерами моделей данных, которые поддерживают многомодельные базы данных, являются документные, графические, реляционные модели и модели ключ-значение [1]. Наличие единой системы данных для анализа и управления, как хорошо структурированными данными, так и данными NoSQL выгодно пользователям, поскольку для каждой конкретной задачи существуют более предпочтительные варианты хранения данных в зависимости от типа СУБД, таким образом, предоставление разных структур хранения данных в одной системе позволяет сделать информацию более доступной и понятной для пользователей. Такая система также улучшает визуализацию и понимание данных.

Запрос на систему для взаимодействия нескольких баз данных между собой выражен в следующей статье [2]. Опосредовано нужна в данной системе упоминается в самых различных сферах, например, при исследовании биологии [3, 4] или политологии [5]. Попытка создания аналогичной системы уже были, однако она была представлена для специализированной сферы, что представлено в статье [6], также уже была попытка создать систему связывающую различные БД через JSON запросы [7], а рассматриваемая в данной работе система также является универсальной, что выражается в поддержке нескольких СУБД и возможности подключать множество баз данных и брать информацию из нескольких источников одновременно, однако работает по другому принципу. В описываемой системе не производится сбор данных из разных БД, а идёт обращение непосредственно к СУБД.

Система разработана на языке Python с использованием фреймворка Flask.

1 ИССЛЕДОВАТЕЛЬСКАЯ ЧАСТЬ

1.1 Обзор СУБД, которые включены в систему

1.1.1 Виды СУБД

Для разработки системы необходимо решить следующие задачи:

1. Произвести анализ средств извлечения информации из Nosql СУБД
2. Исследовать методы и технологии извлечения информации из нескольких СУБД
3. Разработать структуру и архитектуру системы
4. Разработать грамматику языка для запросов к системе
5. Создать метод и алгоритм разбиения запроса к нескольким СУБД на подзапросы для каждой
6. Разработать макет интерфейса системы
7. Произвести тестирование разработанной системы

В настоящее время существует четыре основные модели баз данных: ключ-значение, семейство столбцов, документальные и графовые [8].

Базы данных ключ-значения имеют хранилище ключ-значение. Они позволяют разработчику приложения хранить данные без схемы. Эти данные обычно состоят из строки, которая представляет ключ, и фактических данных, которые считаются значением в отношениях «ключ-значение». Сами данные обычно представляют собой своего рода примитив языка программирования (строка, целое число, массив) или объект, который создаётся привязками языков программирования к хранилищу ключ-значение. Это заменяет необходимость в фиксированной модели данных и делает строгими требования к правильному форматированию данных без данных [9].

Графовая модель базы данных — это модель, в которой структуры данных для схемы и/или экземпляров моделируются как направленный, возможно, помеченный граф или обобщение структуры данных графа, где манипулирование

данными выражается с помощью графо-ориентированных операций и конструкторов типов, а соответствующие ограничения целостности могут быть определены в структуре графа [10].

База данных семейства столбцов — это база данных NoSQL, которая хранит данные с использованием столбцового подхода, в отличие от реляционных, которые упорядочивают данные по строкам. Данные, хранящиеся в базе данных семейства столбцов, выбираются вертикально, что делает частичное чтение более эффективным, поскольку загружается только набор атрибутов строки [11].

Вместо хранения данных в фиксированных строках и столбцах базы данных документов используют гибкие документы. Документ — это запись в базе данных документов. Документ обычно хранит информацию об одном объекте и любых связанных с ним метаданных. Документы хранят данные в парах поле-значение. Значения могут быть различных типов и структур, включая строки, числа, даты, массивы или объекты [12].

1.1.2 MongoDB

MongoDB — это не реляционная, а документно-ориентированная система управления базами данных. Документно-ориентированная СУБД заменяет концепцию «строки» более гибкой моделью, «документом». Позволяя использовать вложенные документы и массивы, документно-ориентированный подход дает возможность представлять сложные иерархические отношения с помощью одной записи. Также нет предопределенных схем: ключи и значения документа не имеют фиксированных типов или размеров. Когда нет фиксированной схемы, добавлять или удалять поля по мере необходимости становится проще. MongoDB — СУБД общего назначения, поэтому помимо создания, чтения, обновления и удаления данных она предоставляет большинство тех функций, которые можно ожидать от системы управления базами данных. Специальные типы коллекций и индексов MongoDB поддерживает коллекции данных TTL (time-to-live), срок

действия которых должен истечь в определенное время, такие как сеансы и коллекции фиксированного размера, для хранения недавно полученных данных, например, журналов. MongoDB также поддерживает частичные индексы, ограниченные только теми документами, которые соответствуют фильтру критериев, чтобы повысить эффективность и уменьшить необходимый объем дискового пространства [13].

Каждый запрос к БД начинается с “db”, после чего идёт символ точки, а затем название коллекции, к которой необходимо обратиться.

Для простого добавления элемента в коллекцию используется метод insert:

```
db.collection.insert({"elem_name":"elem_value"})
```

Для добавления множества элементов необходимо через запятую указать нужные:

```
db.collection.insert({"arr":[{a:1,b:1},{a:2,b:2}]})
```

Всю информацию из коллекции можно удалить через метод remove (сама коллекция при этом не удаляется):

```
db.collection.remove()
```

Удалить объект по какому-то параметру необходимо следующим образом:

```
db.collection.remove({"elem_name":"elem_value"})
```

Для обновления коллекции используется метод update:

```
db.collection.update({"elem_name":"elem_value"})
```

Но тогда будет произведена замена всего документа, а, чтобы такого не произошло необходимо использовать модификатор “\$set”:

```
db.collection.update({$set:{"elem_name":"value"}})
```

Так же с помощью модификатора “\$unset” можно удалить ключ:

```
db.collection.update({$unset:{"elem_name":"value"}})
```

Имеется возможность использования Upsert. При его использовании, если документ по запрашиваемую критерию не найден, то он будет создан, если же найден, то он будет обновлён, как обычно. Чтоб использовать upsert нужно просто в команде update добавить третий параметр равный true:

```
db.collection.update({$set:"elem_name":"value"},true)
```

Для поиска используется `find`. Возвращает массив документов в виде коллекции, если документов нет — пустую коллекцию.

```
db.collection.find({ "elem_name" : "elem_value" })
```

Для добавления условий поиска используются следующие операторы: `$lt` — меньше, `$lte` — меньше или равно, `$gt` — больше, `$gte` — больше или равно, `$ne` — не равно. Так же имеется возможность использовать для поиска регулярные выражения [14].

Для получения индексов коллекции используется команда:

```
db.collection.getIndexes()
```

Для добавления индексов коллекции используется команда:

```
db.collection.createIndex({ "name1":1, "name2":-1 })
```

Для удаления индексов коллекции используется команда:

```
db.collection.dropIndex("index_name")
```

Для вывода всех коллекций используется команда:

```
db.getCollectionNames()
```

Для вывода строения коллекции используется команда:

```
db.collection.aggregate([{"$project":{"arrayofkeyvalue":{"$objectToArray":"$$ROOT"}}}, {"$unwind":"$arrayofkeyvalue"}, {"$group":{"_id":None, "allkeys":{"$addToSet":"$arrayofkeyvalue.k"}}}])
```

Для вывода количества записей в коллекции, используется команда:

```
db.collection.countDocuments()
```

1.1.3 Neo4j

Neo4j — это собственная графовая база данных, а это означает, что она реализует настоящую графовую модель вплоть до уровня хранения. Вместо использования «абстракции графика» поверх другой технологии данные хранятся в Neo4j так же, как вы можете доносить свои идеи на доску. Помимо основного графа, Neo4j также обеспечивает транзакции ACID, поддержку кластеров и аварийное переключение во время выполнения [15].

Для простого добавления узла используется метод create:

```
CREATE (node:label { key1: value1, key2: value2})
```

Узел можно удалить через метод remove:

```
MATCH (node {attribute1: 'value1'}) REMOVE
node.attribute2
```

Для обновления атрибута узла используется метод update:

```
MATCH (node {attribute1: 'value1'}) SET
node.attribute2='value2' RETURN node.attribute1
```

Для поиска значений необходимо указать в конце строки return N.

```
MATCH (n) WHERE (n.id = 0) RETURN n;
```

Для получения индексов меток используется команда:

```
SHOW INDEXES WHERE "name" in labelsOrTypes
```

Для добавления индексов меток используется команда:

```
CREATE INDEX name_index FOR (n:name_label) ON
(n.prop1, n.prop2)
```

Для удаления индексов таблицы используется команда:

```
DROP INDEX name_index
```

Для вывода всех меток используется команда:

```
MATCH (n) RETURN DISTINCT labels(n) AS labels
```

Для вывода строения узлов с конкретными метками используется команда:

```
MATCH (n:name_label) UNWIND keys(n) AS key RETURN key
```

Для вывода количества записей с определёнными метками, используется команда:

```
MATCH (n:name_label) RETURN count(n) AS count
```

1.1.4 Cassandra

Apache Cassandra — это распределенная база данных NoSQL с открытым исходным кодом. Линейная масштабируемость и проверенная отказоустойчивость стандартного оборудования или облачной инфраструктуры

делают его идеальной платформой для хранения критически важных данных [16].

Для простого добавления строки в таблицу используется метод insert:

```
INSERT INTO table_name (id, attribute1) VALUES (now(),  
'value1');
```

Запись можно удалить через метод delete:

```
DELETE FROM table_name WHERE id=54daf810-9aeb-11ea-  
b1d1-3148925e06e7;
```

Для обновления строки используется метод update:

```
UPDATE table_name SET attribute1 = 'value1', WHERE  
id=54daf810-9aeb-11ea-b1d1-3148925e06e7;
```

Для поиска используется команда select:

```
SELECT attribute1, MAX(attribute2) FROM table_name  
GROUP BY attribute1;
```

Для получения индексов таблицы используется команда:

```
SELECT index_name FROM system_schema.indexes WHERE  
table_name = 'table_name' ALLOW FILTERING
```

Для добавления индексов таблицы используется команда:

```
CREATE CUSTOM INDEX name_index ON name_table  
(field_index) USING 'type_index'
```

Для удаления индексов таблицы используется команда:

```
DROP INDEX name_index
```

Для вывода всех таблиц используется команда:

```
SELECT table_name FROM system_schema.tables
```

Для вывода строения таблицы используется команда:

```
SELECT column_name FROM system_schema.columns WHERE  
table_name = 'table_name' ALLOW FILTERING
```

Для вывода количества записей в таблице, используется команда:

```
SELECT COUNT(*) FROM table_name
```

1.2 Сравнение с озером данных

Основная идея системы может показаться похожей на озера данных. Для определения различий необходимо рассмотреть подробнее.

Озеро данных (Data lake)— это система или хранилище данных, которые хранятся в необработанном формате. Data lake обычно представляет собой единое хранилище данных, включающее необработанные копии данных исходной системы. Data lake может включать структурированные данные из реляционных баз данных (строки и столбцы), полуструктурированные данные (CSV, журналы, XML, JSON), неструктурированные данные (электронные письма, документы, PDF-файлы) и двоичные данные (изображения, аудио, видео) [17].

Озёра данных предназначены для того, чтобы собирать, хранить и обрабатывать большое количество информации, поступающей практически непрерывным потоком. Такую информацию называют Big Data, или большими данными. Data Lake полезны всем компаниям, которые планируют анализировать большие данные любой области. Само по себе озеро данных бесполезно, потому что это просто хранилище. Чтобы с ним работать, нужны инструменты для очистки, структурирования, извлечения и анализа данных, и специалисты для работы с этими инструментами [18].

Часто Data lake используют для хранения важной информации, которая пока не используется в аналитике. Или даже для данных, которые кажутся бесполезными, но, вероятно, пригодятся компании в будущем. Data lake позволяет накапливать данные «про запас», а не под конкретный запрос бизнеса. За счет того, что данные всегда «под рукой», компания может быстро проверить любую гипотезу или использовать данные для своих целей [19].

1.2.1 Основная идея построения

Основная идея Data lake заключается в следующем: все данные, отправляемые организацией, будут храниться в единой структуре данных, называемой

Data Lake. Данные будут храниться в озере в исходном формате. Будет исключена сложная предварительная обработка и преобразование данных при загрузке в Data lake. Как только данные помещены в озеро, они доступны для анализа всем сотрудникам организации.

Озеро представляет собой файловое хранилище на нескольких серверах, в котором лежат данные. Как правило данные распределены между серверами, чтобы хранилище можно было быстро масштабировать — подключить новые серверы для расширения места.

К серверам настраивают подключение разных источников данных, доступных компании. Каналы поставки данных называют пайплайнами, а всю схему подключения — ETL-процессом. Обычно всё настроено так, чтобы данные загружались автоматически.

Хотя Data lake и неструктурированное, порядок в нём всё-таки должен быть, иначе спустя время накопится огромное количество данных, в которых невозможно будет разобраться. Поэтому перед добавлением в озеро данные размечают и запоминают, откуда и в каком формате они поступили. В итоге внутри озера данных хранятся не только сами объекты, но и метаданные, то есть информация об объектах. Это облегчает поиск, извлечение и анализ данных в будущем.

В архитектуре озера данных должны быть предусмотрены инструменты резервного копирования, чтобы информация не терялась.

Общий алгоритм работы выглядит следующим образом:

1. В одном из источников формируются данные.
2. По заранее настроенному маршруту данные с серверов отправляются в Data lake.
3. При поступлении данные размечаются: записывается их источник, время поступления, формат и структура.
4. Данные помещаются в озеро и хранятся там. Как правило, срок хранения не ограничен, хотя иногда данные удаляют по мере устаревания или использования в аналитике.

При необходимости данные извлекают из хранилища по определённым критериям и используются [19, 20].

1.2.2 Недостатки Data lake

Озера данных оптимизированы для высокой пропускной способности, но ради этого приходится жертвовать качеством данных [21]:

1. В Data lake не требуется структурировать данные, поэтому их сложнее анализировать.
2. Data lake не имеет инструментов для целостного получения всех данных.
3. Без квалифицированного контроля за озерами данных трудно гарантировать конфиденциальность и безопасность хранилища.
4. Если управление озером организовано плохо, в нем быстро накапливаются большие объемы неконтролируемых, и, возможно, бесполезных данных. Для эффективной фильтрации данных и отсеечения недостоверных источников требуется высокая квалификация.

Если в Data lake хранится слишком много данных, которые плохо организованы, без надлежащего управления метаданными и надежного управления данными, найти соответствующие данные становится все труднее. Через определенное время данные теряют свою актуальность и, если данные все еще остаются в хранилище данных, в течение длительных периодов времени накапливается все больше и больше неактуальных данных. Неправильные временные метки набора данных также приводят к тому, что информацию невозможно найти или оценить. И в таком случае образуется то, что называется болотом данных (data swamp).

Существуют типичные характеристики болота данных, на наличие которых вы можете проверить свое озеро данных и затем от них избавиться:

1. Большие данные без какой-либо организации и документации, например, через каталог данных или концепцию ролей.
2. Отсутствует метаинформация структурированных или неструктурированных данных.

3. Устаревшие и неверные данные.
4. Нет директора по данным или владельца продукта, который управляет платформой.
5. Отсутствующие или нарушенные связи между информацией.

Для очистки данных при замусоривании данных могут оказаться полезны такие роли, как владелец продукта или директор по цифровым технологиям, которые организуют и развивают Data lake. Кроме того, необходимо создать каталог данных, который обеспечит ясность данных. Вместе с концепцией ролей это гарантирует, что данные дойдут до нужных людей. Неверные и старые данные должны быть удалены или заархивированы, поскольку это в любом случае часто требуется нормативными актами и может также привести к снижению затрат. Требованиями к записи данных являются, например, маркировка источника данных, маркировка метаданных и содержательная номенклатура [22].

1.2.3 Обзор платформ для создания и управления озёр данных

GCP предлагает набор услуг автоматического масштабирования, которые позволяют создать озеро данных, которое интегрируется с существующими приложениями. К ним относятся Dataflow и Cloud Data Fusion для поглощения данных, Cloud Storage для хранения, а также Dataproc и BigQuery для обработки данных и аналитики. Google Cloud предоставляет инструменты и рабочие процессы для управления озерами данных на протяжении всего их жизненного цикла. Google структурирует свои услуги озера данных по четырем ключевым этапам жизненного цикла озера данных [23]:

1. Приём — позволяет данным из многочисленных источников, таких как потоки данных о событиях, журналы и устройства IoT, хранилища исторических данных, данные из транзакционных приложений, поступать в озеро данных.
2. Хранение — хранение данных в надёжном и легкодоступном формате.
3. Обработка — преобразование данных из исходного формата в формат, позволяющий использовать и анализировать.

4. Исследование и визуализация — анализ данных и представление их в виде визуализаций или отчетов, предоставляющих ценную информацию бизнес-пользователям.

Так же на этой платформе имеется возможность интеграции уже существующих Data lake из некоторых других платформ.

Hadoop Azure Data Lake - является платформой, в которой и создавалась концепция Data lake [24]. В озерах данных данные чаще всего хранятся в распределенной файловой системе Hadoop (HDFS). Эта система позволяет осуществлять одновременную обработку данных. Это связано с тем, что при приеме данные разбиваются на сегменты и распределяются по разным узлам кластера.

HDFS обладает рядом отличительных свойств [25]:

1. Большой размер блока по сравнению с другими файловыми системами (>64MB), поскольку HDFS предназначена для хранения большого количества огромных (>10GB) файлов;

2. Ориентация на недорогие и, поэтому не самые надежные сервера — отказоустойчивость всего кластера обеспечивается за счет репликации данных;

3. Зеркалирование и репликация осуществляются на уровне кластера, а не на уровне узлов данных;

4. Репликация происходит в асинхронном режиме — информация распределяется по нескольким серверам прямо во время загрузки, поэтому выход из строя отдельных узлов данных не повлечет за собой полную пропажу данных;

5. HDFS оптимизирована для потоковых считываний файлов, поэтому применять ее для нерегулярных и произвольных считываний нецелесообразно;

6. Клиенты могут считывать и писать файлы HDFS напрямую через программный интерфейс Java;

7. Файлы пишутся однократно, что исключает внесение в них любых произвольных изменений;

8. Принцип WORM (Write-once and read-many, один раз записать — много раз прочитать) полностью освобождает систему от блокировок типа «запись-

чение». Запись в файл в одно время доступен только одному процессу, что исключает конфликты множественной записи.

9. HDFS оптимизирована под потоковую передачу данных;

10. Сжатие данных и рациональное использование дискового пространства позволило снизить нагрузку на каналы передачи данных, которые чаще всего являются узким местом в распределенных средах;

11. Самодиагностика — каждый узел данных через определенные интервалы времени отправляет диагностические сообщения узлу имен, который записывает логи операций над файлами в специальный журнал;

12. Все метаданные сервера имен хранятся в оперативной памяти.

AWS EMR — это сервис, предоставляемый Amazon Web Services (AWS). Он объединяет в себе возможности сервиса EMR (Elastic MapReduce) для обработки и анализа больших данных с функциональностью Data Lake, обеспечивая пользователям удобный и масштабируемый способ работы с данными. Особенность AWS EMR, что она использует продукты от AWS.

Для обработки используется AWS Lake Formation, его особенности [26]:

1. Импорт данных из существующих баз данных. Данные сканируются, когда пользователь предоставляет AWS Lake Formation местоположение текущих баз данных и свои данные для входа.

2. Организация и маркировка данных. Lake Formation предлагает коллекцию технических метаданных, извлеченных из источников данных, для потребителей, которые ищут наборы данных.

3. Преобразование данных. Такие преобразования, как перезапись форматов дат для обеспечения единообразия, возможны с помощью Lake Formation. Amazon Data Lake Formation создает шаблоны преобразований и организует процессы, которые будут их выполнять.

4. Принудительное шифрование. Пользовательское Data lake зашифровано с помощью шифрования Amazon S3 через Lake Formation. Чтобы предот-

вратить удаление вредоносных данных при передаче, можно использовать отдельные учетные записи для исходного и целевого регионов при использовании S3.

5. Управление контролем доступа. Lake Formation позволяет управлять разрешениями на доступ к данным в Data lake из одного места. Доступ к данным можно ограничить на уровне базы данных, таблицы, столбца, строки и ячейки с помощью правил безопасности. Эти политики применяются к пользователям и ролям, а также к пользователям и группам, объединенным через внешнего поставщика удостоверений.

6. Настройте ведение журнала аудита. Мониторинг доступа к данным на платформах аналитики и машинного обучения.

7. Метатеги данных для бизнеса. В Data Lake на Amazon можно определить соответствующие варианты использования и уровни конфиденциальности данных, используя безопасность формирования и ограничения доступа.

8. Поиск данные для анализа. Пользователи Lake Formation имеют доступ к текстовому поиску, выполняемому онлайн, для поиска и фильтрации наборов данных, хранящихся в общей библиотеке данных.

В Azure Data Lake представлены все возможности, упрощающие хранение данных любых объема, формата и скорости передачи, а также выполнение любых видов обработки и анализа на разных платформах и языках для разработчиков, специалистов по обработке и анализу данных и аналитиков. Azure Data Lake упрощает получение и хранение данных, одновременно ускоряя работу пакетной, потоковой и интерактивной аналитики.

Особенность заключается в заранее собранном наборе инструментов, таких как [27]:

1. Azure HDInsight — это управляемая комплексная облачная служба аналитики с открытым кодом, предназначенная для предприятий. С помощью HDInsight можно использовать платформы с открытым кодом, такие как Apache Spark, Apache Hive, LLAP, Apache Kafka, Hadoop и т. д. в среде Azure. Azure HDInsight можно применять в различных сценариях обработки больших данных.

Это могут быть исторические данные (данные, которые уже собираются и хранятся) или данные в режиме реального времени (данные, которые передаются непосредственно из источника).

2. Data Lake Store — это высокомасштабируемое облачное озеро данных, предназначенное для предприятий, создано в соответствии с открытыми стандартами HDFS. В нём отсутствии ограничений на размер данных, есть возможностью выполнять огромное количество параллельных аналитических задач и имеется единая платформа хранения данных. Так же имеется проверка подлинности данных с помощью Microsoft Entra ID и управления доступом на основе ролей.

3. Data Lake Analytics — служба заданий аналитики. Это облачная служба аналитики, в которой можно с легкостью разрабатывать и выполнять программы обработки и программы массовых параллельных операций преобразования данных на U-SQL, R, Python и .NET. В ней нет инфраструктуры, так как нет серверов, виртуальных машин или кластеров, которые нужно ждать, настраивать и которыми нужно управлять. Можно масштабировать вычислительную мощность, измеряемую в единицах Azure Data Lake Analytics (AU).

Исходя из написанного выше, можно выделить следующие значимые отличия разрабатываемой системы от озера данных:

1. Данные не хранятся на сервере;
2. Возможность простого подключения БД к уже имеющийся СУБД
3. Иной формат доступа к данным.

1.3. Группы методов взаимодействия различных СУБД

1.3.1 Узконаправленные системы

Проблема взаимодействия с различными СУБД и БД существует уже давно [28]. Далее представлены некоторые способы её решения, поделённые на группы.

В данной группе системы специально создаются системы, которые предназначены для решения проблемы в конкретной среде. Примеры таких систем представлены далее.

Общая архитектура DiscoveryLink является общей для многих гетерогенных систем баз данных, включая TSIMMIS, DISCO, Pegasus, DIOM, HERMES и Garlic. Приложения подключаются к серверу DiscoveryLink с помощью любого из множества стандартных клиентских интерфейсов базы данных, таких как OpenDatabase Connectivity (ODBC) или Java DatabaseConnectivity (JDBC**), и отправляют запросы в DiscoveryLink на стандартном языке SQL. Информация, необходимая для ответа на запрос, поступает из одного или нескольких источников данных, которые были идентифицированы в DiscoveryLink через процесс, называемый регистрацией. Источники данных, представляющие интерес для наук о жизни, варьируются от простых файлов данных до сложных доменно-специфичных систем, которые не только хранят данные, но и включают специализированные алгоритмы для поиска или обработки данных. Возможность использования этих специализированных возможностей не должна быть утрачена при доступе к данным через DiscoveryLink.

Когда приложение отправляет запрос на сервер DiscoveryLink, сервер определяет соответствующие источники данных и разрабатывает план выполнения запроса для получения запрошенных данных. План обычно разбивает исходный запрос на фрагменты, которые представляют работу, которая должна быть делегирована отдельным источникам данных, плюс дополнительную обработку, которая должна быть выполнена сервером DiscoveryLink для дальнейшей фильтрации, агрегации или слияния данных. Способность сервера DiscoveryLink дополнительно обрабатывать данные, полученные из источников, позволяет приложениям использовать всю мощь языка SQL, даже если часть запрашиваемой ими информации поступает из источников данных с небольшими или отсутствующими собственными возможностями обработки запросов, такими как файлы.

Сервер DiscoveryLink взаимодействует с источником данных с помощью оболочки, программного модуля, адаптированного для определенного семейства источников данных. Оболочка для источника данных отвечает за четыре задачи:

1. Отображение информации, хранящейся в источнике данных, в реляционную модель данных DiscoveryLink
2. Информирование DiscoveryLink о возможностях обработки запросов источников данных
3. Отображение фрагментов запроса, отправленных в оболочку, в запросы, которые могут быть обработаны с использованием собственного языка запросов или программного интерфейса источника данных
4. Выдача таких запросов и возврат результатов после их выполнения

Поскольку оболочки являются ключом к расширяемости в DiscoveryLink, одной из основных целей для архитектуры wrapper было обеспечение реализации оболочек для максимально широкого спектра источников данных с минимальными усилиями. Чтобы сделать диапазон источников данных, к которым можно получить доступ с помощью DiscoveryLink, максимально широким, требуется только, чтобы источник данных (или приложение) имел некоторую форму программного интерфейса, который может отвечать на запросы и, как минимум, мог возвращать неотфильтрованные данные, смоделированные как строки таблицы. Автору оболочки не нужно реализовывать стандартный интерфейс запроса, который может быть слишком высокоуровневым или слишком низкоуровневым для базового источника данных. Вместо этого оболочка предоставляет информацию о возможностях обработки запросов источника данных и специализированных средствах поиска серверу DiscoveryLink, который динамически определяет, какую часть данного запроса источник данных способен обработать. Этот подход позволяет быстро создавать оболочки для простых источников данных, сохраняя при этом возможность использовать уникальные возможности обработки запросов нетрадиционных источников данных, таких как поисковые системы для химических структур или изображений. Оболочка может быть написана с минимальным знанием внутренней структуры DiscoveryLink. В результате стоимость

написания базовой оболочки невелика. Оболочка, которая просто делает данные из нового источника доступными для DiscoveryLink, не пытаясь использовать большую часть собственных возможностей обработки запросов источника данных, может быть написана за считанные дни.

Поскольку сервер DiscoveryLink может компенсировать отсутствующую функциональность в источниках данных, даже этот вид простой оболочки позволяет приложениям применять всю мощь SQL для извлечения новых данных и интеграции данных с информацией из других источников, хотя, возможно, и с производительностью ниже оптимальной. После написания базовой оболочки ее можно постепенно улучшать, чтобы использовать больше возможностей обработки запросов источника данных, что приводит к повышению производительности и повышению функциональности, поскольку раскрываются специализированные алгоритмы поиска или другие новые возможности обработки запросов источника данных.

Оболочка DiscoveryLink — это программа на языке C++, упакованная как общая библиотека, которая может динамически загружаться сервером DiscoveryLink при необходимости. Обычно одна оболочка способна получать доступ к нескольким источникам данных, если они используют общий или похожий интерфейс прикладного программирования (API). Это происходит потому, что оболочка не кодирует информацию о схеме, используемой в источнике данных. Таким образом, схемы могут развиваться без необходимости внесения изменений в оболочку, пока API источника остается неизменным. Например, оболочка Oracle, предоставляемая DiscoveryLink, может использоваться для доступа к любому количеству баз данных Oracle, каждая из которых имеет свою схему [6].

1.3.2 Ручная интеграция СУБД

Данная группа пытается вручную объединить несколько разных СУБД. Для этого имеется несколько подходов.

Первый подход. Получить все имеющиеся схемы БД и, сравнивая и редактируя их, можно добиться слияния в единую схему БД.

Первым и обязательным условием является выбор СУБД для проектируемой гетерогенной системы баз данных. Этот выбор показывает решающее влияние на последующие процессы разработки и оценки систем для объединения данных SQL и NoSQL. Например, в качестве СУБД SQL можно выбрать PostgreSQL. PostgreSQL — это объектно-реляционная система на основе данных с открытым исходным кодом, которая использует расширяемый язык SQL в сочетании с некоторыми функциями, которые позволяют безопасно хранить и масштабировать сложные рабочие данные. К основным преимуществам PostgreSQL можно отнести высокую степень развития: в этой СУБД есть поддержка главных объектов и их поведений, включая типы данных, операции, функции, индексы и домены. По этой причине PostgreSQL можно назвать действительно гибким. Кроме того, эта СУБД предоставляет возможность создавать, хранить и использовать сложные структуры данных. Также стоит отметить, что PostgreSQL поддерживает вложенные и составные конструкции, которые не применяются и основаны на существующих стандартных реляционных базах данных. В качестве нереляционной СУБД можно выбрать MongoDB — система управления базами данных NoSQL, которая набирает всю большую популярность на рынке и выделяет среди конкурентов своей технологией масштабировать потребление. К преимуществам MongoDB можно отнести гибкость, масштабируемость, доступность и высокую производительность.

При проектировании базы данных для управления проектами необходимо учитывать множество аспектов, связанных с проектами управления. Следует учитывать, что разрабатываемая система может использовать команды, которые могут входить в состав различных отдельных компаний, а также просто обычных людей, некоторые из которых разделились в команде для разработки продукта.

В проектах часто могут использоваться различные факторы, основанные на различных типах баз данных. Причин для такого подключения может быть

несколько: добавление новых баз данных для балансировки нагрузки, обеспечение хранения данных или использование ресурсов.

Для доступа к внешним данным (из одной базы данных в другую) используются оболочки (wrapper) внешних данных (Foreign Data Wrappers). Wrapper — это библиотека, предназначенная для взаимодействия с внешним источником и загрузки данных из него. В качестве внешних источников могут выступать репозитории NoSQL или сторонние серверы Postgres.

В настоящее время доступно множество оберток внешних данных (FDW), которые позволяют серверу PostgreSQL работать с различными удаленными хранилищами данных.

Среди множества оберток внешних данных для баз данных NoSQL можно также найти FDW для MongoDB. Обертка данных MongoDB выполняет функцию соединения между сервером MongoDB и PostgreSQL, транслируя операторы PostgreSQL в запросы, понятные базе данных MongoDB. Для этого соединения поддерживаются операторы SELECT, INSERT, DELETE и UPDATE.

Чтобы настроить соединение между PostgreSQL и MongoDB для отправки запросов, потребуется установить расширение `mongo_fdw`. Для компиляции `mongo_fdw` требуются следующие библиотеки:

- `libbson`;
- `libmongoc`;
- `json-c`.

Библиотеки `libbson` и `libmongoc` необходимы для корректной работы `mongo_fdw`, поскольку это расширение использует для работы драйвер языка C [29].

Второй подход — разработка интегрированной схемы. Этапы разработки интегрированной схемы:

- Предварительная интеграция, где входные схемы преобразуются, чтобы сделать их более однородными (как синтаксически, так и семантически);
- Идентификация соответствия, посвященная идентификации и описанию межсхемных отношений;

- Интеграция, заключительный этап, который разрешает межсхемные конфликты и объединяет соответствующие элементы в интегрированную схему.

Предварительный этап интеграции. Установление общего понимания существующих данных является предпосылкой успешной интеграции баз данных. Для этой цели входные схемы обычно преобразуются, чтобы сделать их максимально однородными. Исследователи в области интеграции баз данных обычно предполагают, что все входные схемы выражены в одной и той же модели данных, так называемой «общей» модели данных (CDM). Этап преобразования становится предпосылкой интеграции и рассматривается как отдельная проблема.

К сожалению, современное состояние перевода моделей данных не располагает инструментами для автоматического перевода. Текущие разработки сосредоточены на переводах между объектно-ориентированными и реляционными моделями.

Большинство исследователей отдают предпочтение объектно-ориентированной модели. Аргумент заключается в том, что она содержит все концепции других моделей и что можно использовать методы для реализации определенных правил отображения. Но чем богаче модель, тем сложнее будет процесс интеграции, так как возникнет много несоответствий из-за разных выборов моделирования разными дизайнерами. Для упрощения интеграции альтернативой является CDM с минимальной семантикой, где представления данных сводятся к элементарным фактам, для которых нет альтернативы моделирования, как в моделях с бинарными отношениями. Вместо того, чтобы иметь дело с конкретными переводами, например, из модели X в модель Y, в последних работах ищутся общие методы. Общие трансляторы используют метамодель, т. е. модель данных, способную описывать модели данных, для получения знаний о моделях данных. Затем переводы выполняются как последовательность процесса реструктуризации данных в базе данных метамодели (например, вложение плоских кортежей для получения вложенного кортежа). Модели данных не могут выразить всю семантику реального мира. Неполнота их спецификаций приводит к неоднозначностям в интерпретации схемы. Семантическое обогащение — это процесс получения

дополнительной информации для разрешения таких неоднозначностей. Самый сложный случай — когда данные находятся в файлах, но понимание ненормализованных и плохо документированных реляционных баз данных также представляет собой серьезную проблему.

Шаг идентификации соответствия. Следующий шаг — это выявление общих черт. Базы данных содержат представления фактов реального мира (объектов, связей или свойств). Интеграция баз данных выходит за рамки представлений, чтобы сначала искать то, что представлено, а не то, как оно представлено. Поэтому говорят, что две базы данных имеют что-то общее, если факты реального мира, которые они представляют, имеют некоторые общие элементы или иным образом взаимосвязаны. Примером последнего является: две отдельные библиотеки, одна из которых специализируется на научных дисциплинах, а другая — на социальных науках, желающие создать интегрированную базу данных.

Было бы целесообразно создать интегрированные типы объектов, такие как Автор (соответственно Статья), представляющие всех авторов (соответственно все статьи) в любой библиотеке. Мы говорим, что два элемента (вхождение, значение, кортеж, ...) из двух баз данных соответствуют друг другу, если они описывают один и тот же факт реального мира (объект, связь или свойство).

Вместо того, чтобы определять соответствия экстенционально, между экземплярами, соответствия определяются интенционально, между типами. Пример: каждая статья S2 соответствует статье S1, так что значение заголовка в S1 равно значению заголовка в S2. Интенциональное определение называется утверждением соответствия между схемами (ICA). Процесс интеграции состоит в определении этих ICA и предоставлении для каждого из них федеративным пользователям глобального описания со всеми доступными данными по связанным элементам. Федеративная система хранит глобальное описание в интегрированной схеме (IS) и определение отображений между IS и локальными схемами. Полная интеграция существующих баз данных требует исчерпывающей

идентификации и обработки всех соответствующих ICA. Тем не менее, возможно принять инкрементальный подход, при котором IS плавно обогащается по мере постепенного выявления новых соответствий [30].

1.3.3 Создание новой СУБД

Для решения задачи связи нескольких СУБД можно разработать совершенно новые СУБД.

Менеджер хранения в СУБД обычно является интерфейсом для преобразования запроса данных СУБД в запрос ввода-вывода. В процессе преобразования вся семантическая информация удаляется, оставляя только информацию о физической компоновке запроса: логический адрес блока, направление (чтение/запись), размер и фактические данные, если это запись.

Это, по сути, создает семантический разрыв между СУБД и системами хранения. В hStorage-DB преодолевается семантический разрыв, предоставляя выбранную и важную семантическую информацию менеджеру хранения, который, следовательно, может классифицировать запросы на разные типы.

С набором предопределенных правил каждый тип ассоциируется с политикой качества обслуживания (QoS), которую может поддерживать базовая система хранения.

Во время выполнения, используя протокол дифференцированных услуг хранения, связанная политика запроса передается в систему хранения вместе с самим запросом. Получив запрос, система хранения сначала извлекает связанную политику QoS, а затем использует соответствующий механизм для обслуживания запроса в соответствии с требованиями политики QoS.

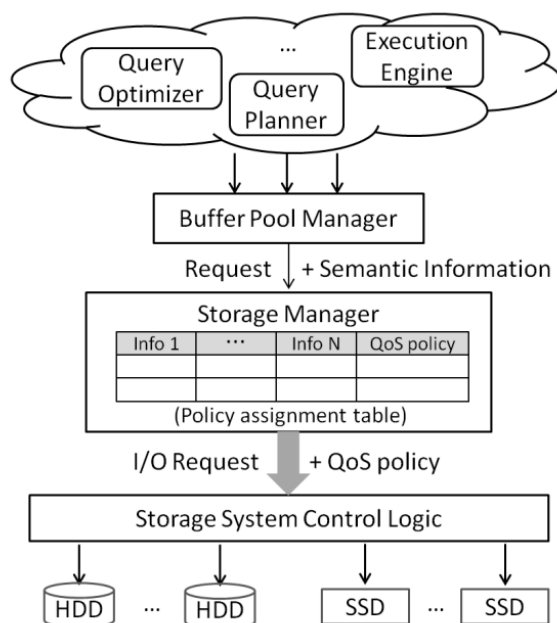


Рисунок 1 – Архитектура hStorage-DB

Рисунок 1 показывает архитектуру hStorage-DB. Когда менеджер буферного пула отправляет запрос менеджеру хранилища, также передается связанная семантическая информация. hStorage-DB расширяет менеджер хранилища «таблицей назначения политик», которая хранит правила для назначения каждому запросу надлежащей политики QoS в соответствии с его семантической информацией. Политика QoS встроена в исходный запрос ввода-вывода и доставляется в систему хранения через блочный интерфейс. hStorage-DB с использованием протокола Differentiated Storage Services от Intel Labs для доставки запроса и связанной с ним политики в гибридную систему хранения.

После получения запроса система хранения сначала извлекает политику и вызывает механизм для обслуживания этого запроса.

hStorage-DB основана на PostgreSQL 9.0.4. В основном это касается трех вопросов: (1) Она оснащена оптимизатором запросов и механизмом выполнения для извлечения семантической информации, встроенной в деревья планов запросов и в запросы пула буферов. (2) Имеет расширенную структуру данных пула буферов для хранения собранной семантической информации. Менеджер хранилища также был расширен для включения «таблицы назначения политик». (3) Наконец, поскольку PostgreSQL является многопроцессорной СУБД, для работы

с параллелизмом был выделен небольшой регион общей памяти для глобальных структур данных, к которым должны иметь доступ все процессы.

Ключом к эффективности hStorage-DB является связывание каждого запроса с надлежащей политикой QoS [31].

Авторы в статье [32] представляют другой двухуровневый подход, чтобы преодолеть гетерогенность и учитывать автономию баз данных насколько это возможно, и при этом концептуально построить Гиперраспределенную Базу Данных на основе существующих баз данных. Пользователи постепенно и динамически обучаются доступному информационному пространству, не будучи перегруженными всей доступной информацией. Двухуровневая структура предоставляет участвующим базам данных гибкое средство для обмена информацией. Система, которая реализует этот подход, называется FINDIT.

Двухуровневый подход, который предлагается соответствует коалициям (первый уровень) и сервисным ссылкам (второй уровень). Коалиции являются средством для баз данных, чтобы быть сильно связанными, в то время как сервисные ссылки являются средством для них, чтобы быть слабо связанными. Со-базы данных вводятся как средство для реализации этих концепций и как помощь для межсайтового обмена данными.

Коалиции являются группировками баз данных, которые разделяют некоторый общий интерес. Это состоит в интересе к информационному мета-типу (например, Рестораны). В этом контексте базы данных будут делиться описаниями этого типа информации.

Во многих странах политические партии, придерживающиеся разных идеологий, согласны на минимальный набор целей на ограниченный период времени. Результатом является коалиция. Таким образом, коалиции основаны на краткосрочном интересе. В политических коалициях члены сохраняют свою автономию, оставаясь при этом приверженными набору правил, которые были согласованы. Наша концепция коалиций поэтому очень близка к концепции политических коалиций. С другой стороны, концепция федерации больше похожа на

федерацию штатов, где набор правил является долгосрочным и где штаты пользуются разумным, хотя и ограниченным, объемом автономии.

Другим примером группировки, близким к концепции коалиций, является Интернет. Интернет — это компьютерная сеть, которая состоит из подсетей, которые соединены друг с другом. Каждый подсеть имеет свой собственный набор стандартных протоколов для общения. Хотя, в отличие от FINDIT, все подсети предоставляют почти одинаковый набор информации, идея кооперативной среды существует. В каждой подсети участвующие сайты подчиняются набору правил, которые регулируют общение между собой. Подсети обычно создаются для выполнения определенной цели в рамках некоторых географических границ. Например, NSFnet — это сеть, целью которой является связывание крупных исследовательских учреждений в США (географическая граница) для проведения исследований с использованием суперкомпьютеров (цель). Другим примером подсети является DECnet, где границей является компания (организационная граница), а целью является предоставление структуры для их исследователей для обмена информацией. Каждая сеть связана с другими сетями, в основном для обмена электронной почтой.

Сервисные ссылки — это упрощенный способ для баз данных делиться информацией. Они позволяют обмениваться с низкими накладными расходами. Сервисные ссылки обычно предполагают некоторые обязательства, которые похожи на контракты. В терминах обмена данными, объем, ожидаемый для обмена в сервисной ссылке, обычно включает минимальное количество информации. Например, ожидается, что будет поделено только имя информации с несколькими синонимами и информация о базе данных, которая экспортирует эту информацию. В этом отношении сервисные ссылки несут низкие накладные расходы по сравнению с использованием коалиций.

Сервисные ссылки могут возникать между любыми двумя сущностями. Сервисные ссылки могут быть только одного из трех типов. Первый тип включает сервисную ссылку между двумя коалициями для обмена информацией. Второй тип включает сервисную ссылку между двумя базами данных. Третий тип

включает сервисную ссылку между коалицией и базой данных. Сервисная ссылка между двумя коалициями включает предоставление общего описания информации, которая будет делиться. Аналогично, сервисная ссылка между двумя базами данных также включает предоставление общего описания информации, которую базы данных хотели бы поделить. Третья альтернатива — это сервисная ссылка между коалицией и базой данных. В этом случае база данных (или коалиция) предоставляет общее описание информации, которую она готова поделить с коалицией (или базой данных). Разница между этими тремя альтернативами заключается в том, как разрешаются запросы. В первой и третьей альтернативе (когда поставщик информации — это коалиция) предоставляющая коалиция берет на себя дальнейшее разрешение запроса. Во втором случае, однако, пользователь отвечает за контакт с администратором предоставляющей базы данных, чтобы узнать больше о информации.

Когда базы данных неохотно показывают слишком много деталей о типе информации, которую они содержат, у них есть выбор присоединиться к сервисной ссылке с другими базами данных или коалициями. По сути, сервисные ссылки — это средства для баз данных быть слабо связанными с другими базами данных (или коалициями). Это предоставляет структуру, в которой базы данных обмениваются минимальным количеством данных о информации, которую они хотели бы поделить. Обмениваются только общая информация о фактической информации, которую нужно поделить, и информация о обслуживающих базах данных. Следует подчеркнуть, что, как и в любой сервисной ссылке, существует услуга, которую должна предоставить одна из вовлеченных сущностей, т.е. коалиции или базы данных.

Со-базы данных — это объектно-ориентированные базы данных, прикрепленные к каждой участвующей базе данных. Объектно-ориентированность зарекомендовала себя как отличная парадигма моделирования для сложной структуры и поведения. Наследование и инкапсуляция оказались критически важными при проектировании надежных и легко поддерживаемых программных систем. Схема со-базы данных состоит из двух подсхем. Каждая подсхема представляет

либо коалицию, либо сервис. Каждая подсхема состоит из решетки классов. Каждый класс представляет набор баз данных, которые могут отвечать на запросы о конкретном типе информации (например, запросы о ресторанах). Подсхема сервиса состоит из двух подсхем: первая — это подсхема услуг, которые коалиции, членом которых она является, имеет с другими базами данных и коалициями, а вторая — это подсхема услуг, которые база данных имеет с другими базами данных и коалициями. Каждая из этих подсхем, в свою очередь, состоит из двух подклассов, которые соответственно описывают услуги с базами данных и услуги с другими коалициями. Подсхема коалиции состоит из одной или нескольких подсхем, где каждая из них представляет коалицию.

Описание о коалиционных сервисах включает информацию о точках входа и контакте с этими коалициями. Другие описания предоставляют информацию местным базам данных, чтобы можно было выбрать лучшую точку контакта. Следует отметить, что подсхема, представляющая набор коалиционных сервисов, будет одинаковой для всех баз данных, которые являются членами обслуживающей коалиции.

1.3.4 Унифицировать имеющиеся модели БД

Можно каким-либо образом унифицировать модели БД и тогда с ними можно будет удобно взаимодействовать.

Фундаментальную модель унифицированной системы управления данными можно проследить до системы управления объектно-реляционными базами данных (ORDBMS), которая интегрирует объектно-ориентированные функции в реляционную модель. Система ORDBMS может управлять различными типами данных, такими как реляционные, объектные, текстовые и пространственные, подключая доменно-специфические типы данных, функции и реализации индексов в ядра СУБД. Например, PostgreSQL поддерживает реляционные, пространственные и XML-данные.

Единая модель данных. В чистой реляционной модели столбец таблицы должен быть встроенным скалярным типом. Таким образом, чистая модель RDB представляет собой набор элементов, где каждый элемент имеет встроенные скалярные типы. С другой стороны, модель объектной базы данных в ORDBMS по-прежнему имеет концепцию набора верхнего уровня (которая обычно известна как коллекция объектов). Коллекция объектов представляет собой набор, содержащий элементы произвольного сложного объекта. Сам объект представляет собой еще один набор своих элементов, каждый из которых может быть другим набором. Однако текущая проблема в управлении многомодельными данными заключается в том, что каждый объект, такой как XML, JSON и граф, моделирует свои собственные данные домена и имеет определенный язык запросов (например, SQL для реляционных данных, XQuery для XML и SPARQL для RDF). Таким образом, унифицированная многомодельная база данных должна предоставлять новую (логически) унифицированную модель данных, которая действует как глобальное представление для различных типов данных.

Такая абстракция может скрыть детали реализации данных от пользователей и облегчить глобальный доступ и запрос для различных типов данных. Текущие усилия авторов статьи [33] в этом направлении направлены на объединение пяти типов данных, включая отношение, ключ-значение, JSON, XML и граф. Эта цель может быть достигнута в два этапа. На первом этапе представляем гибкий способ представления графа, JSON, XML и моделей ключ-значение в виде унифицированной модели NoSQL (UNM) логически. На втором этапе будут исследованы новые подходы для объединения UNM и моделей отношений. В конечном итоге унифицированная модель может поддерживать все пять типов данных.

Эта модель определит глобальные представления и операции для пяти типов данных. Эта унифицированная модель данных заложит общие основы для доступа к многомодельным данным и управления ими. Гибкое управление схемами. Оригинальная ORDBMS предполагает идеальный мир, основанный на схемах. Полуструктурированные данные и неструктурированные данные бросают вызов ORDBMS с дизайном без схем.

Улучшение обнаружения схемы для всех видов данных является проблемой, и это еще один интерфейс, которого не хватает в исходной ORDBMS. В ORDBMS нет индексирующего интерфейса обнаружения схемы.

Эволюция модели. С ростом зрелости баз данных NoSQL многие приложения переходят на хранение данных с помощью документов JSON или представлений «ключ-значение». Но их устаревшие данные по-прежнему хранятся в традиционной ORDBMS. Таким образом, изменение модели может повлиять на удобство использования запросов и приложений, разработанных в ORDBMS. Поэтому в унифицированной многомодельной базе данных исследовательская задача заключается в том, как выполнить отображение модели и переписывание запроса для автоматической обработки эволюции модели. Необходимо обратить внимание, что эволюция модели является более сложным процессом, чем эволюция схемы в ORDBMS, поскольку она включает в себя как изменение атрибутов, так и изменение структуры.

Другой способ представлен в статье [34] для решения рассматриваемой задачи предлагается 4-уровневой архитектуре клиент-сервер с использованием системы с несколькими базами данных можно визуализировать как систему клиент-сервер, которая позволяет клиентам одновременно получать доступ и обновлять данные, хранящиеся на нескольких серверах распределенных баз данных:

Уровень 1 – это клиентский графический пользовательский интерфейс или веб-интерфейс, который находится на вершине клиентской прикладной программы или сервера приложений

Уровень 2 – это сервер приложений, который содержит клиентскую программу, бизнес-логику, API и доступ к серверу системы с несколькими базами данных.

Уровень 3 – это система с несколькими базами данных, которая контролирует и поддерживает глобальную схему и глобальный каталог, а также доступ к различным удаленным серверам баз данных на основе запросов пользователей.

Уровень 4 – это удаленные гетерогенные локальные серверы компонентных баз данных.

Существует клиентская программа, которая находится на верхнем уровне глобальной системы управления базами данных и глобальной схемы сервера многобазовой системы. Глобальная схема создается с набором виртуальных глобальных классов и хранится в глобальной базе данных (GDB). Пользователь будет отправлять запрос на глобальную схему, используя веб-интерфейс или графический пользовательский интерфейс программы приложения, запрос будет разбит на набор подзапросов и будет отправлен на соответствующие удаленные локальные компонентные серверы баз данных и будет выполняться локально.

Примером реализации данной архитектуры может являться MOMIS [35].

Так же можно использовать JSON. В настоящее время многие основные реляционные базы данных, такие как Oracle, Microsoft SQL Server, MySQL, PostgreSQL и TeraData, активно изучаются для выявления способов оптимизации производительности базы данных для адаптации к эпохе больших данных. Таким образом, были предприняты попытки интегрировать хранилище текста JSON в реляционные базы данных для совместимости с базами данных NoSQL, тем самым достигая эффективного гибридного облачного хранилища.

Тем не менее, характеристики самих реляционных баз данных привели к их неотъемлемой неспособности выполнять обработку JSON. Этот недостаток также заставил разработчиков неохотно использовать единую реляционную базу данных для одновременной обработки высокопроизводительных данных и сложных логических реляционных данных в современной гибридной облачной системе хранения.

Более того, исследователи предложили несколько методов хранения текста JSON в реляционных базах данных. Обсуждалось хранение собственных данных JSON в коммерческих базах данных и использование SQL для расширенных запросов.

Авторами [4] предложен гибридный язык запросов JSON на основе SQL. Были предложены и сравнены два различных метода отображения, которые использовались для хранения данных JSON в реляционных базах данных.

Модель данных сущность-атрибут-значение использовалась для обсуждения поддержки двух реляционных баз данных с открытым исходным кодом и двух коммерческих реляционных баз данных для документов JSON.

В определенной степени интеграция текста JSON, хранящегося в реляционной базе данных, решила проблему взаимодействия различных типов данных в гибридном облачном хранилище. Однако реляционная база данных не подходит для текстового хранилища JSON, поскольку она в первую очередь предназначена для хранения реляционных структур данных. Более того, совместимость текста JSON, хранящегося в реляционных базах данных, зависит от его операторов SQL из-за ограничений его структуры. Поэтому многие исследователи пытались рассмотреть взаимодействие между различными типами данных в гибридном облачном хранилище с другой точки зрения, то есть достичь взаимодействия посредством взаимного сопоставления между JSON и реляционной базой данных для реализации единого управления хранилищем текста JSON и реляционной базой данных.

С точки зрения исследования сопоставления данных JSON с реляционными данными был предложен алгоритм сопоставления из JSON в реляционную базу данных, и данные JSON были сохранены в реляционной базе данных. JSON был определен в веб-запросе данных. Была предложена формальная модель данных JSON и был определен легкий язык запросов. Был предложен формат обмена данными между службами RESTful, который больше склонен хранить данные сетевых атрибутов.

На основе существующих моделей сопоставления между JSON и реляционными данными и нереляционными данными в статье были объединены эти две модели сопоставления и предложена новая модель JSON под названием XYJSON model. Используя сопоставление модели управления XYJSON, эта модель данных достигла унифицированного управления различными типами баз данных,

помогая заполнить пробел в модели управления приложениями для гибридных облачных баз данных и продвигая исследования по унифицированному управлению для гибридных облачных баз данных.

1.3.5 Взаимодействие в виде графа

В статье [36] авторы предлагают глобальную систему, в которой все схемы отображаются в виде графа и доступны для взаимодействия через графический интерфейс, со следующими основными характеристиками:

- Графовая модель (GM) для представления и запроса баз данных. Этот модель подходит для придания точной семантики сложным визуальным представлениям и является достаточно общим для формализации, в принципе, базы данных, выраженной в любой из наиболее распространенных моделей данных. Прimitives запроса модели, хотя и состоят исключительно из двух элементарных графических действий, а именно выбора узла и рисования ребра, по крайней мере столь же выразительны, как реляционная алгебра.

- Адаптивный визуальный интерфейс, построенный на основе вышеуказанной модели, предоставляющий пользователю различные визуальные представления и механизмы взаимодействия вместе с возможностью переключения между ними.

- Определение трех подходящих наборов алгоритмов перевода, один для перевода базы данных, выраженной в любой из наиболее распространенных моделей данных, во внутреннюю системную модель, один для перевода запроса GM в терминах языков запросов базовых моделей данных и один, предназначенный для реализации последовательного переключения между различными визуальными представлениями во время формулировки запроса.

- Построение и управление эффективной пользовательской моделью, которая позволяет системе предоставлять пользователю наиболее подходящее визуальное представление в соответствии с его навыками и потребностями.

Система состоит из диспетчера визуального интерфейса, диспетчера пользовательской модели, диспетчера G MDB и запросов и одной или нескольких СУБД. Диспетчер визуального интерфейса способен поддерживать несколько представлений (на основе форм, иконических, диаграммных и гибридных) баз данных и соответствующих модальностей взаимодействия. Таким образом, пользователю предоставляется язык многопарадигматических запросов, основанный на наборе визуальных языков запросов, каждый из которых взаимодействует с различным визуальным представлением ГМ и все они разделяют одну и ту же выразительную силу.

Представления на основе форм являются первой попыткой предоставить пользователю дружественные интерфейсы для манипулирования данными; они обычно предлагаются в рамках реляционной модели, где формы на самом деле являются таблицами. Их основная характеристика состоит в визуализации прототипических форм, в которых запросы формулируются путем заполнения соответствующих полей. В предшествующих системах, принявших представление на основе форм, таких как QBE, отображается только интенциональная часть отношений: экстенциональная часть заполняется пользователем, чтобы предоставить пример запрошенного результата.

В более поздних предложениях интенциональная и экстенциональная части базы данных сосуществуют. Диаграммные представления являются наиболее используемыми в существующих системах. Обычно они представляют с помощью различных визуальных элементов различные типы концепций, доступных в модели. Соответствие между визуальными элементами и связанными типами концепций требует эстетических критериев для размещения визуальных элементов и связей. Например, иерархические структуры для обобщения и агрегации объектов диктуют вертикальное размещение задействованных элементов. Диаграммные представления принимают в качестве типичных операторов запроса выбор элементов, обход смежных элементов и создание моста между разъединенными элементами.

Иконическое представление использует наборы иконок для обозначения как объектов базы данных, так и операций, которые должны быть выполнены с ними. Запрос выражается в первую очередь путем объединения иконок. Например, иконки могут быть объединены вертикально для обозначения конъюнкции (логическое И) и горизонтально для обозначения дизъюнкции (логическое ИЛИ). Чтобы быть эффективным, предлагаемый набор иконок должен быть легко понятен большинству людей.

Однако во многих случаях трудно или даже невозможно найти общепринятый набор иконок. В качестве альтернативы иконки могут быть определены пользователем для адаптации к конкретным потребностям пользователя и его/ее собственному ментальному представлению задач, которые он/она хочет выполнить. Гибридные представления используют произвольную комбинацию вышеуказанных подходов, либо предлагая пользователю различные альтернативные представления баз данных и запросов, либо объединяя различные визуальные структуры в единое представление. Диаграммы часто используются для описания схемы базы данных, в то время как значки используются либо для представления конкретных прототипических объектов, либо для указания действий, которые необходимо выполнить. Формы в основном используются для отображения результата запроса.

Можно выделить недостатки для каждой группы:

- Узконаправленные системы. Узконаправленное применение, доступны только СУБД для конкретной тематики.
- Ручная интеграция СУБД. Необходимо продумывать действия при каждой новой БД, ошибки из-за человеческого фактора.
- Создание новой СУБД. Отсутствие обмена опытом из-за малой распространённости.
- Унифицировать все имеющиеся модели БД. Необходимо продумывать действия при каждой новой БД.
- Взаимодействие в виде графа. Ограничение взаимодействия из-за формы отображения в виде графа

2 КОНСТРУКТОРСКО-ТЕХНОЛОГИЧЕСКАЯ ЧАСТЬ

2.1 Описание взаимодействия модулей

В данной работе разрабатывается системы для анализа и взаимодействия пользователя с множеством баз данных, находящимися в различных системах управления базами данных. Первоначально пользователю необходимо выбрать нужные ему базы данных и соответствующие СУБД, из которых он планирует извлекать данные. Список доступных для выбора СУБД заранее определён разработчиком. Для извлечения данных нужно писать запросы по разработанному синтаксису. Пользователь имеет возможность получать информацию из нескольких баз данных и СУБД одновременно. Кроме того, данная система обладает гибкостью, так как имеет возможность использовать потенциально любую СУБД, при условии, что для этого предварительно добавлен необходимый функционал, соответствующий указанному шаблону.

В системе используется две PostgreSQL базы данных. Первая будет хранить данные о пользователях, пример с одной записью представлен в таблице 1. Вторая будет хранить данные о подключениях, пример с одной записью представлен в таблице 2.

Таблица 1 – Схема БД пользователей

id	login	password
0	log	pass

Таблица 2 – Схема БД подключений

id	login	password	ip	port	conn_name	db_name	special	user_id
4	neo4j	87654321	localhost	7687	neo	neo4j		0

Всего в системе можно выделить 3 части - ввод запроса пользователем, парсер запроса для каждой СУБД, исполнение запроса каждой СУБД. Визуально их взаимодействие показано на рис. 2, текстовое описание представлено далее.

Первая часть - ввод запроса пользователем. Пользователь в браузере вводит запрос на получение нужных ему данных в соответствии с созданным языком запросов, строение которого описано далее.

Вторая часть - парсер запросов. В этом элементе системы запрос пользователя разбивается на части для каждой БД. Эти части запроса в нужной для корректного исполнения последовательности подаются в соответствующие классы определённой СУБД, после чего возвращают результат, который уже обрабатывается и выводится пользователю или добавляется в запрос для следующего класса СУБД, если это не конечный результат.

Третья часть, которую можно выделить, это - классы СУБД. Каждый такой класс представляет собой возможность взаимодействия с СУБД для выполнения запроса, а его объекты представляют каждую БД в этой СУБД. Каждый объект класса должен преобразовывать часть запроса, пришедшего к нему от парсера, в запрос на язык, который является командой для СУБД, к классу которой он относится. После преобразования он обращается с этой командой к БД, к которой он относится, и возвращает результат в парсер.

При взаимодействии с СУБД имеются следующие возможности:

1. Взаимодействие с данными (добавление, удаление, обновление, чтение) в различных БД из одной системы с помощью созданного языка запросов;
2. Взаимодействие с индексами, а именно создание, удаление, просмотр созданных;
3. Получение списка всех таблиц из БД;
4. Показ структуры выбранной таблицы из любой БД;
5. Показ количества записей в выбранной таблице;
6. Показ времени выполнения запроса;
7. Показ загрузки оперативной памяти при исполнении запроса;
8. Показ плана запроса;
9. Запуск запроса на языке СУБД.

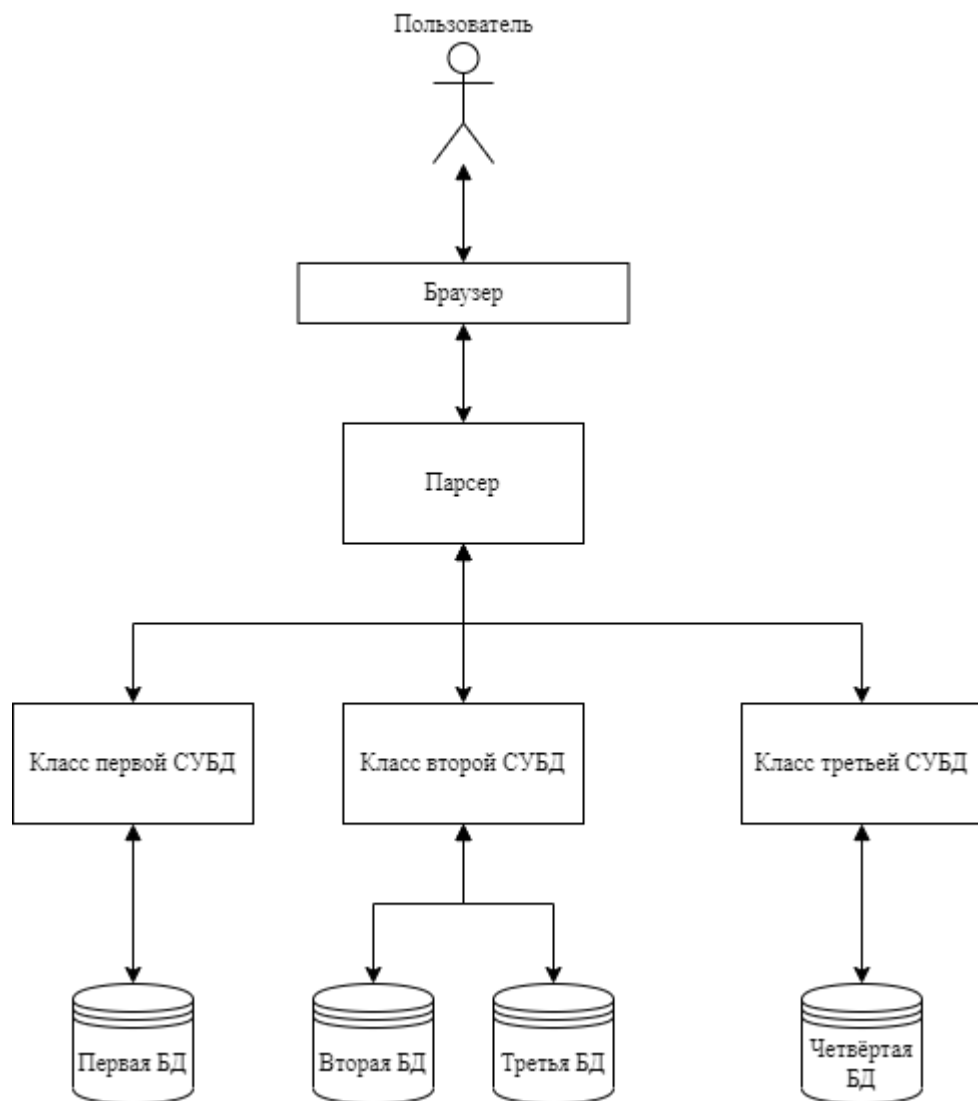


Рисунок 2 – Взаимодействие между элементами

2.2 Синтаксис и грамматика созданного языка запросов

Чтобы была возможность делить запросы для исполнения на разных СУБД необходимо сделать свой синтаксис запросов. Далее представлено описание синтаксис основных запросов.

Запрос будет записываться одной строкой, в которой будут писаться название БД, путь к данным из БД и операторы с разделителями в виде точек.

Пример запроса:

```

dbcity.read(dbcity.building.address).where(dbpeople.read(
dbpeople.personal_data.name).where(dbpeople.personal_data.id=1)=dbcity.building.address_owner)
  
```

Сначала пишется название БД. После этого через разделитель - точку, пишется оператор, применяемый к указанной БД.

В системе будут представлены следующие операторы:

- **x.y.z**, где x – название БД, y – название сущности, являющийся аналогом таблицы из реляционных БД, из ранее обозначенной базы данных, z – название сущности, являющийся аналогом столбцов из реляционных БД, из ранее обозначенной таблицы. Таким образом из БД можно получить необходимые данные.

- **where** – оператор, в котором в скобках описывается фильтр, по которому выбираются данные из БД. В условии используются оператор доступа к данным и операторы сравнения. Сравниваемыми элементами могут являться данные из БД или константы. Условия могут вложенными. Так же возможно использование операторов AND и OR.

- **create** – вставка данных в БД. У оператора в скобках сначала через запятую указываются операторы доступа к данным, которые означают новые поля, после чего в конце через запятую указываются вставляемые данные в виде массива списков. Например,

```
dbcity.create(dbcity.building.address,  
dbcity.building.owner, [[“Wall Street”, “Stan Smith”],  
[“Broadway”, “John Doe”]]).
```

- **read** – чтение данных из БД. У оператора в скобках указываются операторы доступа к данным, которые должны быть прочитаны. Например,

```
dbcity.read(dbcity.building.address).
```

- **update** – изменение данных в БД. У оператора в скобках сначала через запятую указываются операторы доступа к данным, которые означают поля, в которых будут производиться изменения, после чего в конце через запятую указываются вставляемые данные в виде списка. Например,

```
dbcity.update(dbcity.building.address,  
dbcity.building.owner, [“Broadway”, “John Doe”]).
```

- **delete** – удаление данных из БД. После данного оператора указывается оператор `where`. Например, `dbcity.delete.where(dbcity.building.owner="John Doe").`

- **index** – получение индексов для конкретной таблицы. Выводит список всех индексов для указанной в аргументе таблицы. Например, `mg.index(human).`

- **all** – получить список всех таблиц. Выводит все таблицы, имеющиеся в БД. Например, `mg.all()`.

- **show** – показ структуры выбранной таблицы. Для таблицы, указанной в аргументе, выводит её строение. Например, `mg.show(human).`

- **count** – количество записей в таблице. Выводит количество записей для указанной в аргументе таблицы. Например, `mg.count(human).`

- **create_index** – добавление индекса. Для добавления необходимо указать сначала название индекса, затем тип индекса, после чего в квадратных скобках указать столбцы, для которых будет применяться индекс. Например, `mg.create_index("name", 1, [mg.human.id, mg.human.name]).`

- **delete_index** – удаление индекса. Удаляется индекс, название которого указано в аргументе. Например, `neo.delete_index("name").`

- **exec** – исполнение кода на языке СУБД. Исполняет код, который указан в аргументе, на языке запросов СУБД. Например, `cas.exec("SELECT country_name FROM country WHERE city_name='Msk' ALLOW FILTERING").`

Грамматика описывается формулой в приложении А.1. Пример преобразования запроса на разработанном языке в запрос для каждой из доступных СУБД представлен на рисунке 3.

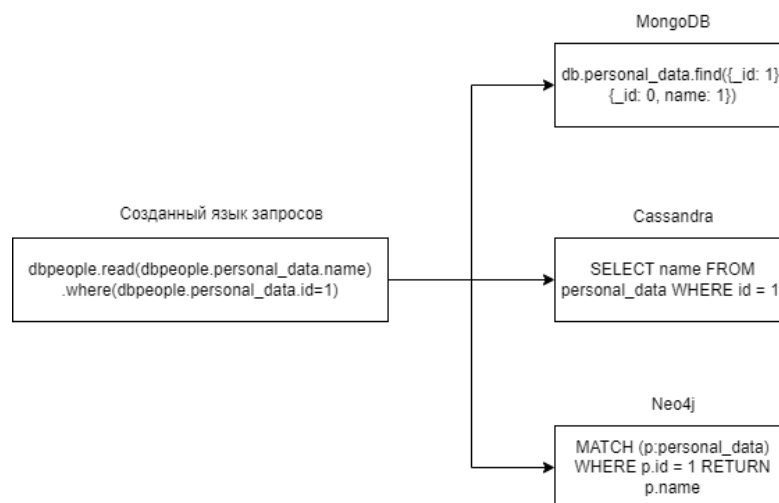


Рисунок 3 – Пример преобразования запроса на разработанном языке в запрос для каждой из доступных СУБД

2.3 Классы СУБД

В системе для каждой СУБД создаётся собственный класс. Для каждой БД, которую подключает пользователь создаётся собственный объект соответствующего класса СУБД. Все классы наследуются от шаблона, пример для Neo4j, Cassandra и MongoDB представлен на рисунке 4.

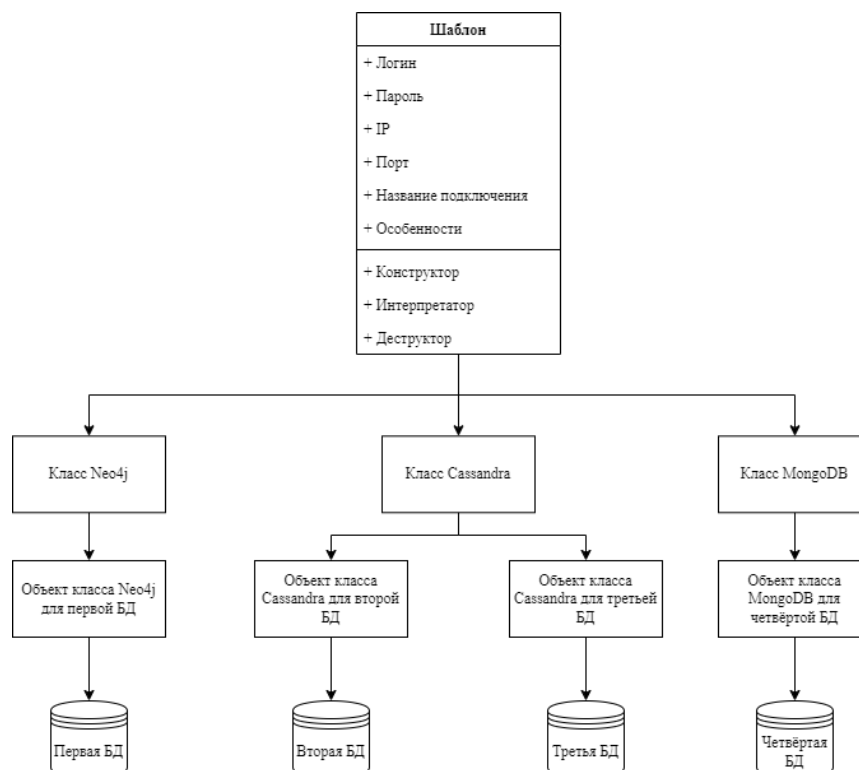


Рисунок 4 – Схема работы классов СУБД

Сам шаблон должен иметь следующую структуру полей:

- Логин – логин, который нужен для подключения к базе данных.
- Пароль – пароль, который нужен для подключения к базе данных.
- IP – ip для подключения к базе данных.
- Порт – порт для подключения к базе данных.
- Название подключения - название, которое необходимо для различия между разными БД одной СУБД.
- Особенности – текстовая строка с необходимой информации для корректного выполнения запросов.

Так же шаблон должен содержать следующие методы:

- Конструктор. Метод, которому на вход подаётся - адрес базы данных; порт базы данных; логин для авторизации; пароль для авторизации; название базы данных. Данный метод создаёт и хранит подключение и название базы данных.
- Интерпретатор. Метод, которому на вход подаётся - запрос на синтаксисе, описанном в предыдущей главе. Данный метод преобразует полученный запрос в формат, нужный для конкретной СУБД и исполняет его. В конце возвращает массив кортежей с полученным результатом.
- Деструктор. Метод, который закрывает подключение для текущей БД.

2.4 Парсер запросов

В описываемой в данной работе системе запрос от пользователя необходимо разделить на подзапросы. После этого разбитые подзапросы конвертируются в запросы, которые могут воспринимать СУБД, для которых эти запросы предназначены и исполняются.

Сначала запрос читается слева направо и разбивается на подзапросы. Далее подзапросы конвертируются в читаемый для СУБД вид и исполняются справа налево с передачей результата запроса. Данный процесс подробно изображен на рисунке 5.

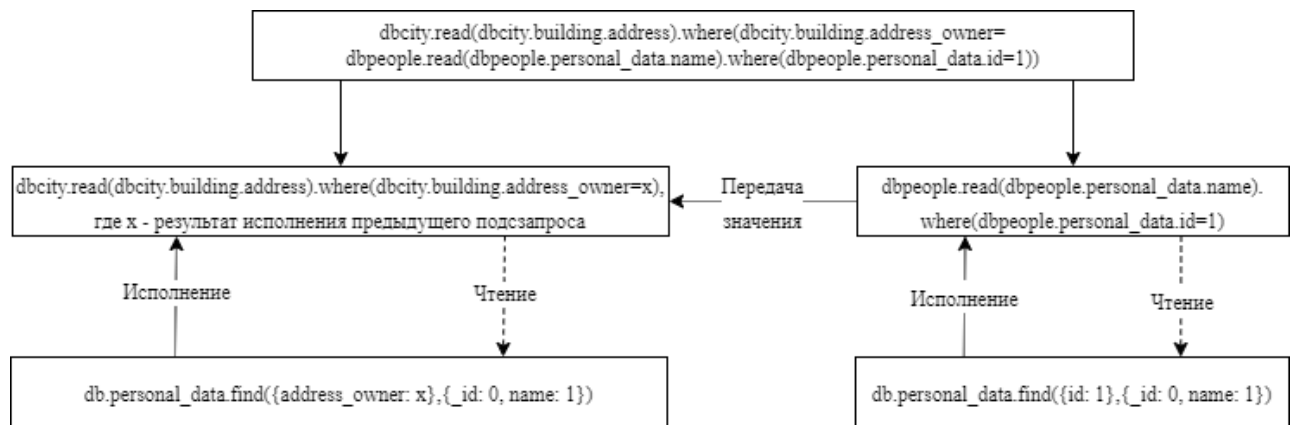


Рисунок 5 – Деление основного запроса на подзапросы для каждой БД

Конвертацию подзапросов осуществляет общий для всех классов БД метод Интерпретатор. Рассмотрим работу одного из них, а именно интерпретатора класса MongoDB.

Был выделен следующий подзапрос:

```
dbpeople.read(dbpeople.personal_data.name).where(dbpeople.personal_data.id=1)
```

Сначала идёт подключение к базе данных dbpeople. Парсер начинает выполнять с наиболее вложенного подзапроса:

```
(dbpeople.personal_data.name).where(dbpeople.personal_data.id=1)
```

В этом запросе dbpeople – БД в СУБД MongoDB.

Часть запроса dbpeople.personal_data.name преобразуется в {_id: 0, name: 1}, where(dbpeople.personal_data.id=1) преобразуется в {_id: 1}.

Далее конвертируется часть запроса dbpeople.read() и выполняется в виде следующего запроса:

```
db.personal_data.find({_id: 1},{_id: 0, name: 1})
```

Для создания очереди на исполнения создаётся стек, со следующим строением:

1. Название подключения – какое подключение необходимо выполнить для исполнения запроса.

2. Команда – название команды, которая будет исполняться
3. Аргументы команды – аргументы, которые были указаны при вызове команды.
4. Where список – список, в котором хранятся символы, которые необходимо учитывать при анализе условия – “(”, “)”, “AND”, “OR”, “=”, “!=”, “<”, “>”, “<=”, “>=”
5. Номер итерации – положение элемента в стеке, в который необходимо вставить результат
6. Позиция в where списке – в какой порядковый номер необходимо вставить результат в where списке

Сначала в запросе ищется первая из команд, после чего в элемент стека заносится подключение для которого была вызвана команда и затем сама команда. После ищется конец аргумента, начиная с первого символа, если количество открывающих скобок равно -1 (так как первая уже учтена) и встречается закрывающая, то это конце аргумента. Весь аргумент заносится в элемент стека. Далее проверяется имеется ли у команды оператор where. Если он отсутствует, то заносится пустой список и на место координат вставки результата записывается число -1. В случае наличия оператора where ищется конец его аргумента. И после система пытается найти есть ли в аргументе операторы AND или OR. Для этого ищутся координаты начала и конца первого названия путём посимвольного прохождения всей строки, пока не будет встречен символ. После этого всё что до первого аргумента вносится отдельно в список where и вместо аргумента ставится @. Данный процесс продолжается пока не закончится проверяемая строка. Все найденные аргументы заносятся в отдельный список. После для каждого аргумента в списке ищется разделитель аналогично поиску конца аргумента. Далее каждый аргумент разделителя, если они простые заносятся вместо @. Если же они составные, то для них процесс начинается сначала, но уже заносят результат в итерацию и место, где оно стоит. Место помечается знаком %.

Последовательно из стека берётся запись, которая передаётся в соответствующий интерпретатор, который определяется по первому значению в этой записи. После выполнения запроса берётся результат и заносится вместо символа % в where список нужного элемента стека. Место для вставки указано последними 2 элементами в записи стека. И данный процесс продолжается пока не закончатся записи в стеке.

Описание метода для деления на подзапросы и приведения к универсальному списку для каждой СУБД:

1. В строке ищется первая из возможных команд.
2. В элемент стека заносится подключение, для которого была вызвана команда, и затем сама команда.
3. С начала оставшейся строки ищется конец аргумента.
4. Весь аргумент заносится в элемент стека.
5. Далее проверяется, имеется ли у команды оператор where. Если он отсутствует, то в элемент стека заносится пустой список и на место координат вставки результата записывается число -1.
6. В случае наличия оператора where ищется конец его аргумента.
7. Начинается посимвольное прохождение всей строки, пока не будет встречен оператор AND или OR.
8. Символы до первого аргумента, вносятся отдельно в список where, и вместо аргумента ставится @. Найденные аргументы заносятся в отдельный список. Затем заносится сам оператор AND или OR.
9. Если была пройдена не вся строка, то возврат к пункту 7.
10. Для каждого аргумента в списке ищется оператор сравнения.
11. Каждый аргумент разделителя, если они простые, заносится вместо @ и сам разделитель между ними.
12. Если же они составные, то для каждой части в элемент стека заносится итерация и место, которое помечается символом % и возврат к пункту 1.

Блок-схема алгоритма для деления на подзапросы и приведения к универсальному списку для каждой СУБД, представлен на рисунке 6.

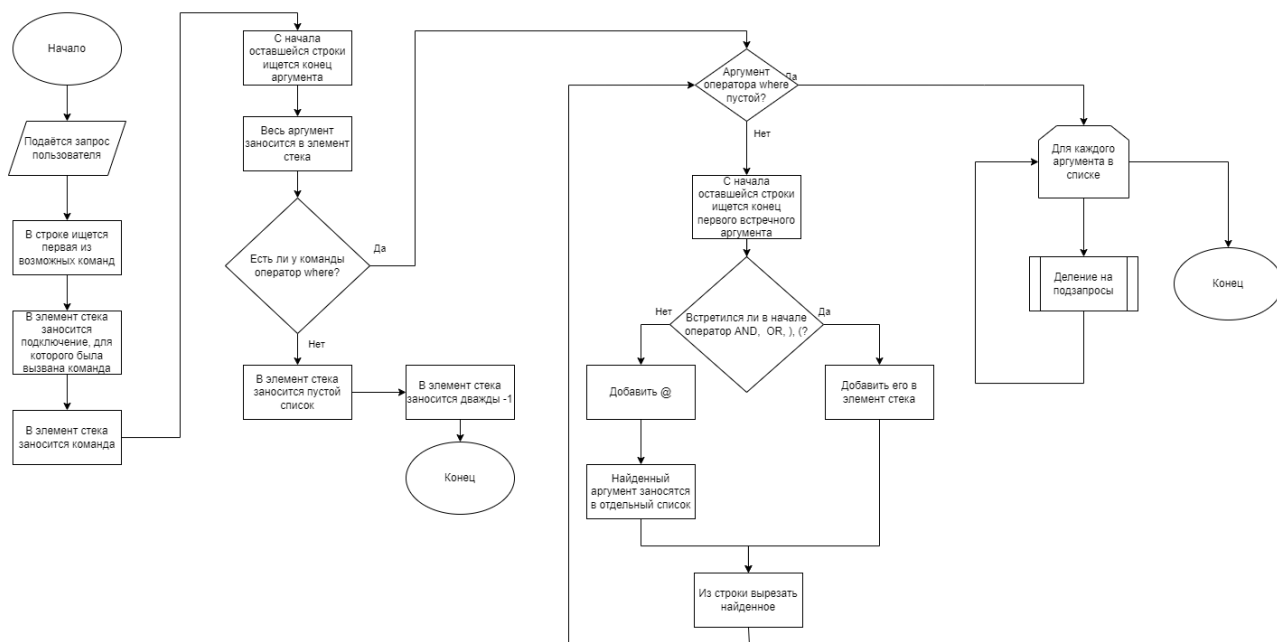


Рисунок 6 – Блок-схема алгоритма для деления на подзапросы и приведения к универсальному списку для каждой СУБД

2.5 UML диаграмма системы

Диаграмма прецедентов представлена на рисунке 7.

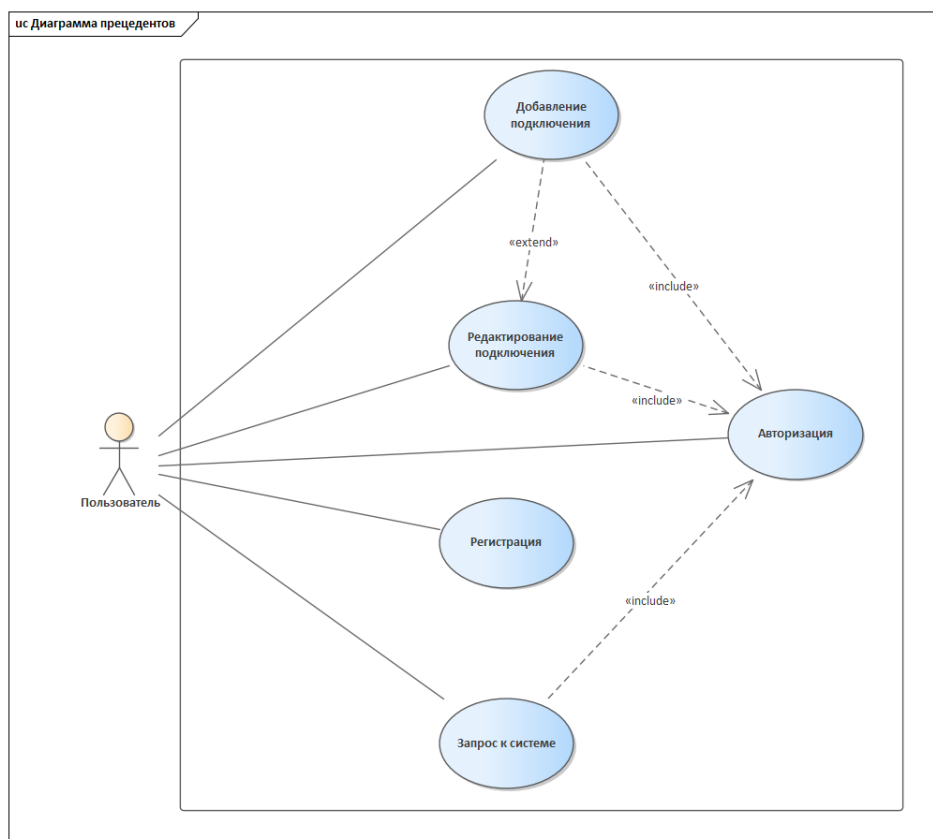
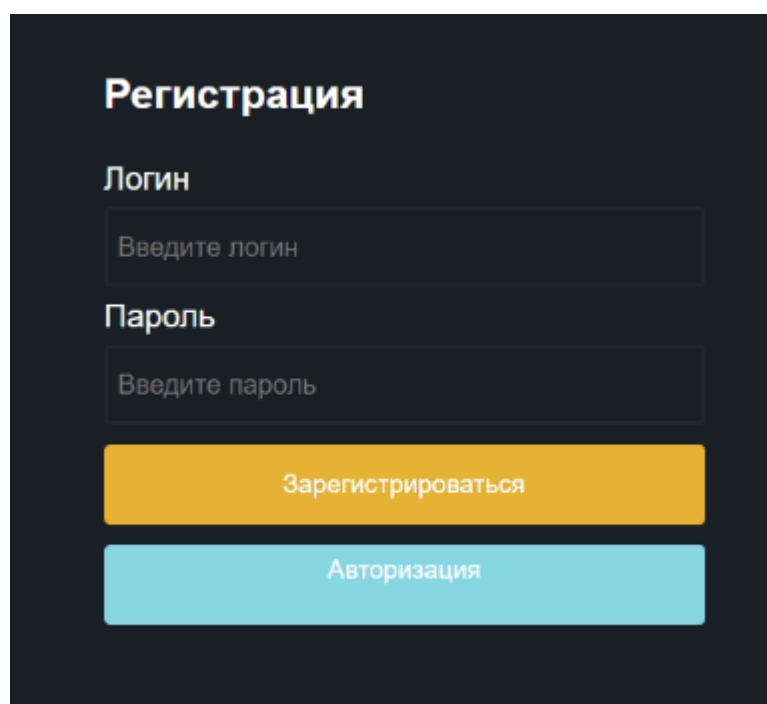


Рисунок 7 – Диаграмма прецедентов

Разработанный интерфейс страницы для авторизации представлен на рисунке 8.

Рисунок 8 – Интерфейс страницы для авторизации

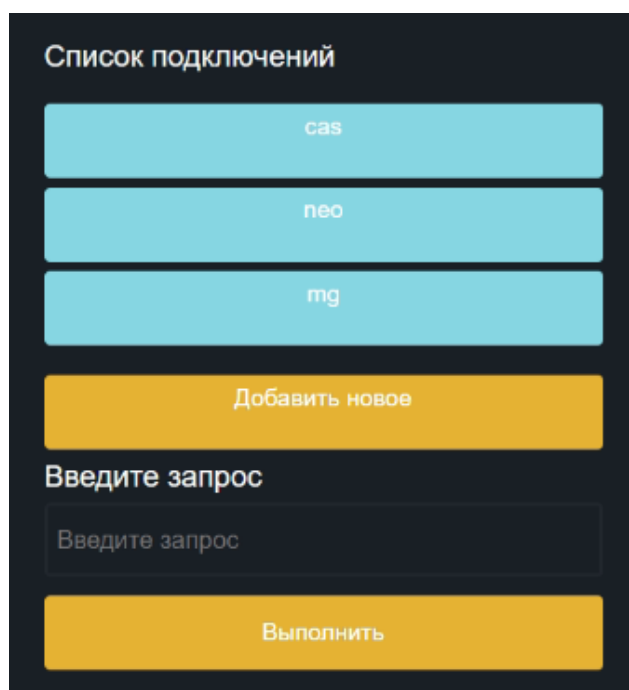
Интерфейс страницы регистрации представлен на рисунке 9.



The registration form is displayed on a dark background. It features a title 'Регистрация' in white. Below the title are two input fields: 'Логин' (Login) and 'Пароль' (Password), both with placeholder text 'Введите логин' and 'Введите пароль' respectively. Below the password field are two buttons: a yellow 'Зарегистрироваться' (Register) button and a light blue 'Авторизация' (Authorization) button.

Рисунок 9 – Интерфейс страницы для регистрации

Разработанный интерфейс основной страницы представлен на рисунке 10.



The main page interface is displayed on a dark background. It features a title 'Список подключений' (List of connections) in white. Below the title are three light blue buttons labeled 'cas', 'neo', and 'mg'. Below these buttons is a yellow button labeled 'Добавить новое' (Add new). Below the yellow button is a text input field with the placeholder 'Введите запрос' (Enter query). Below the input field is a yellow button labeled 'Выполнить' (Execute).

Рисунок 10 – Интерфейс основной страницы

Разработанный интерфейс добавления нового подключения представлен на рисунке 11.

Создание подключения

Введите название подключения

Введите IP

Введите PORT

Логин

Пароль

Особенности

СУБД
MongoDB ▾

Создать

Назад

Рисунок 11 – Интерфейс страницы для добавления нового подключения

Интерфейс редактирования подключения представлен на рисунке 12.

The image shows a dark-themed web interface for editing a database connection. At the top, the title is "Подключение: нео" (Connection: neo). Below it, the database name is "СУБД: Neo4j" (DB: Neo4j). There are four input fields: "Введите IP" (Enter IP) with the value "localhost", "Введите PORT" (Enter PORT) with the value "7687", "Логин" (Login) with the value "neo4j", and "Пароль" (Password) with masked characters ".....". Below these is a section titled "Особенности" (Features) with a text input field containing "Введите новые дополнительные параметры" (Enter new additional parameters). At the bottom, there are three buttons: a yellow "Сохранить" (Save) button, a red "Удалить" (Delete) button, and a light blue "Назад" (Back) button.

Подключение: нео

СУБД: Neo4j

Введите IP

localhost

Введите PORT

7687

Логин

neo4j

Пароль

.....

Особенности

Введите новые дополнительные параметры

Сохранить

Удалить

Назад

Рисунок 12 – Интерфейс страницы для редактирования подключения

Интерфейсы для результата выполнения запроса к системе представлен на рисунке 13.

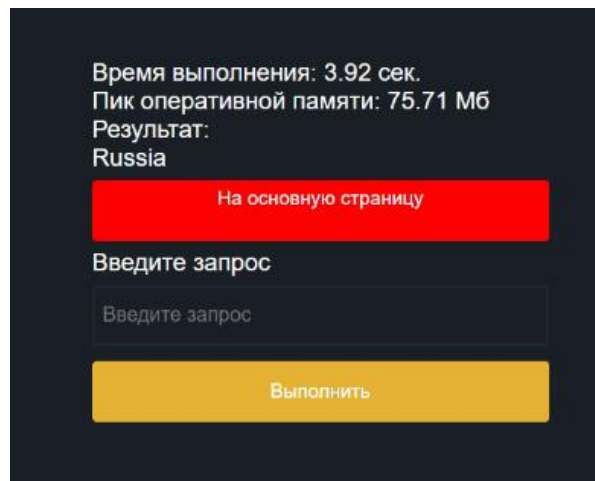


Рисунок 13 – Интерфейс страницы для результата выполнения запроса

В результате разработки получилась следующая диаграмма классов сущностей представлена на рисунке 14.

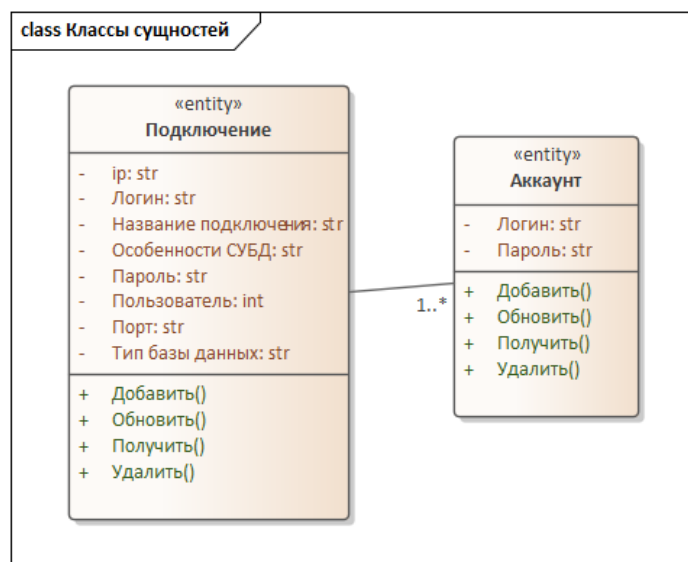


Рисунок 14 – Диаграмма классов сущностей

На рисунке 15 изображена разработанная диаграмма управляющих классов.

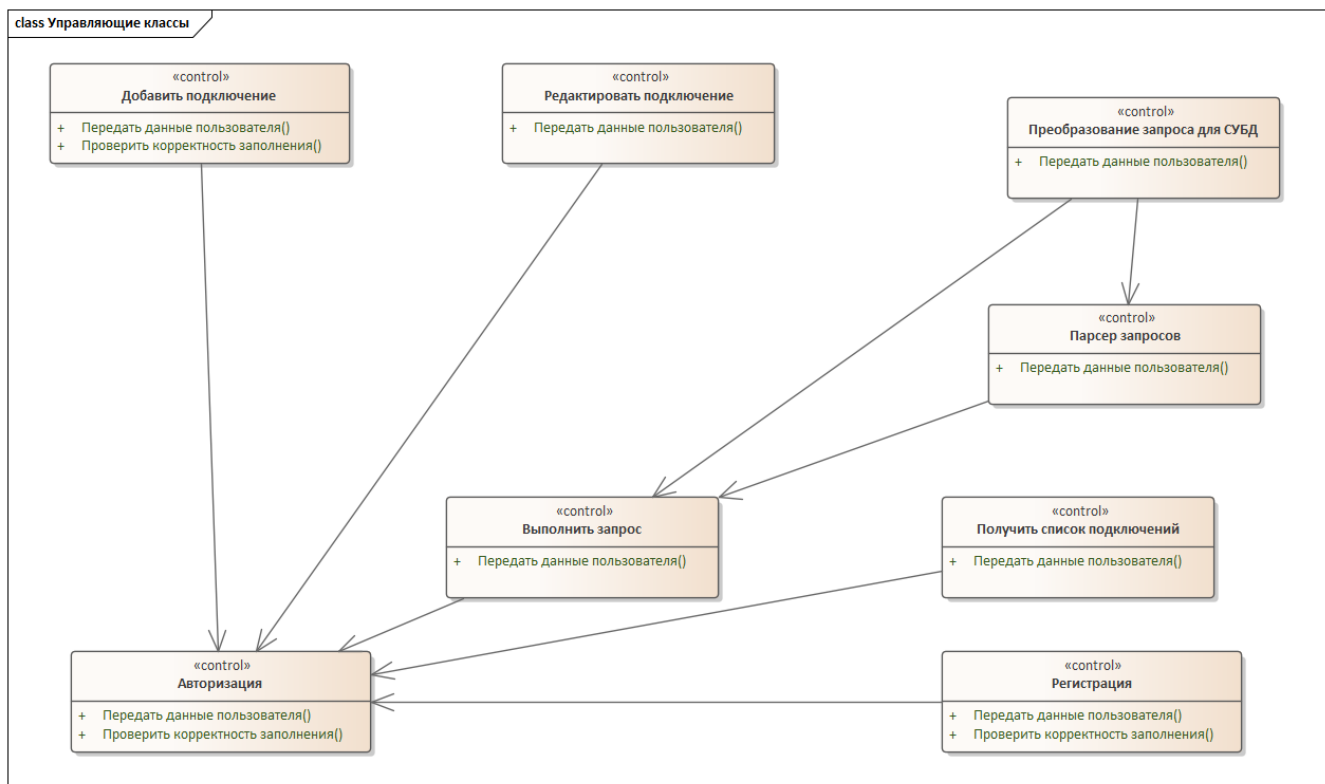


Рисунок 15 – Диаграмма управляющих классов

Созданная диаграмма граничных классов представлена на рисунке 16.

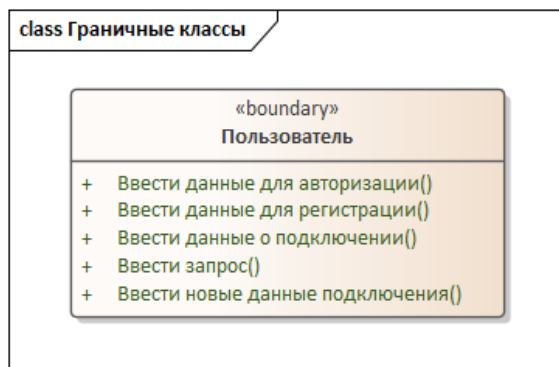


Рисунок 16 – Диаграмма граничных классов

Диаграмма классов для прецедента авторизация представлена на рисунке 17.

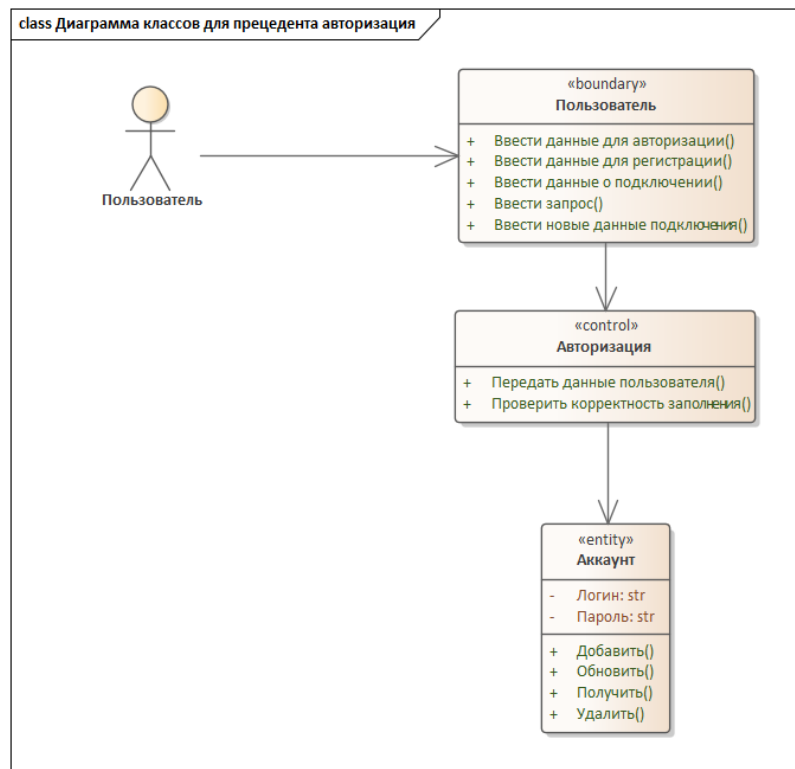


Рисунок 17 – Диаграмма классов для прецедента авторизация

В результате разработки получилось следующее строение диаграммы последовательностей для прецедента авторизация, которая представлена на рисунке 18.

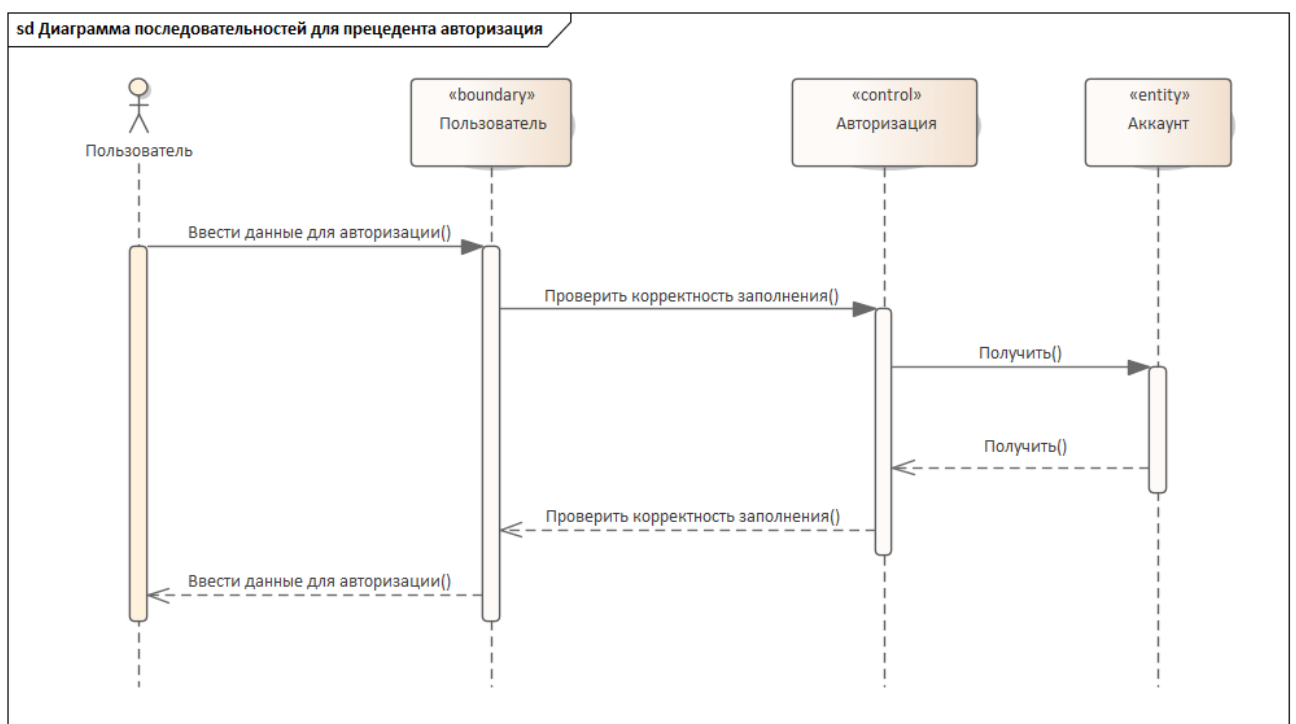


Рисунок 18 – Диаграмма последовательностей для прецедента авторизация

На рисунке 19 изображена разработанная диаграмма классов для прецедента регистрация.

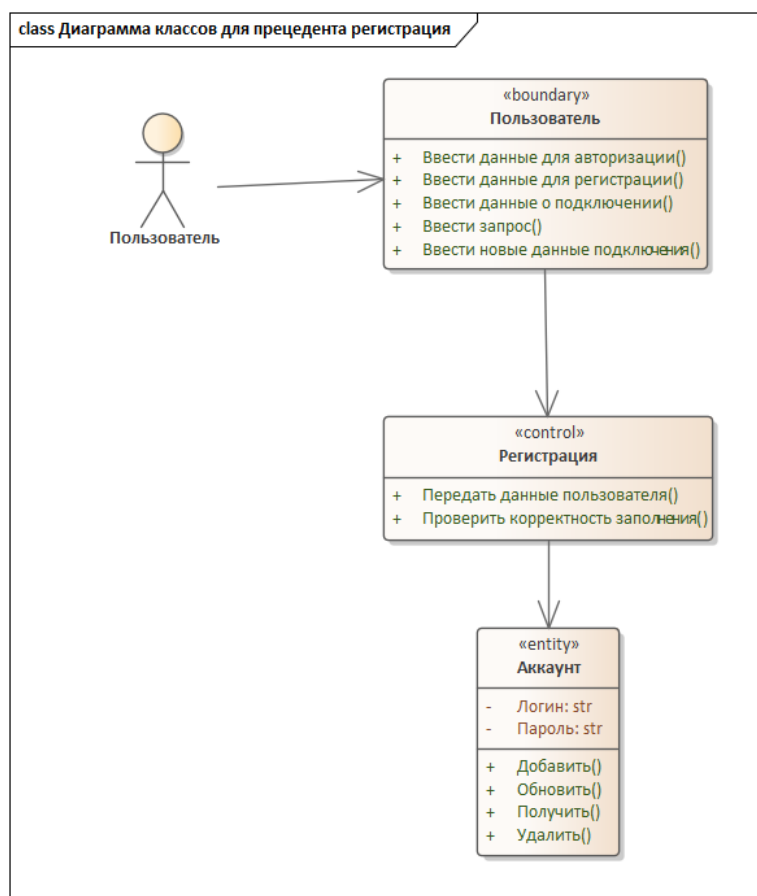


Рисунок 19 – Диаграмма классов для прецедента регистрация

Созданная диаграмма последовательностей для прецедента регистрация представлена на рисунке 20.

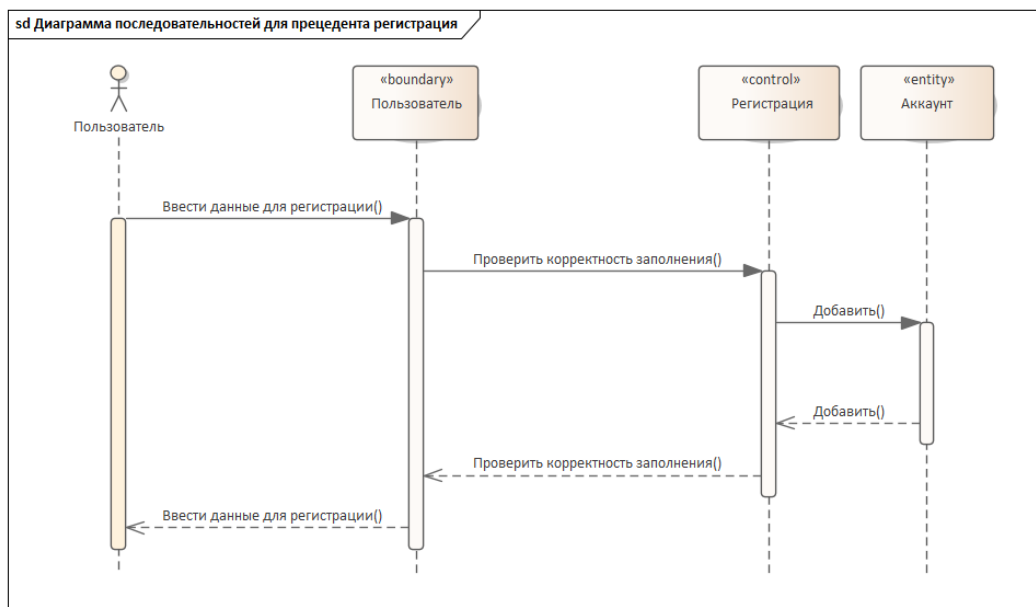


Рисунок 20 – Диаграмма последовательностей для прецедента регистрация

Диаграмма классов для прецедента добавления подключения представлена на рисунке 21.

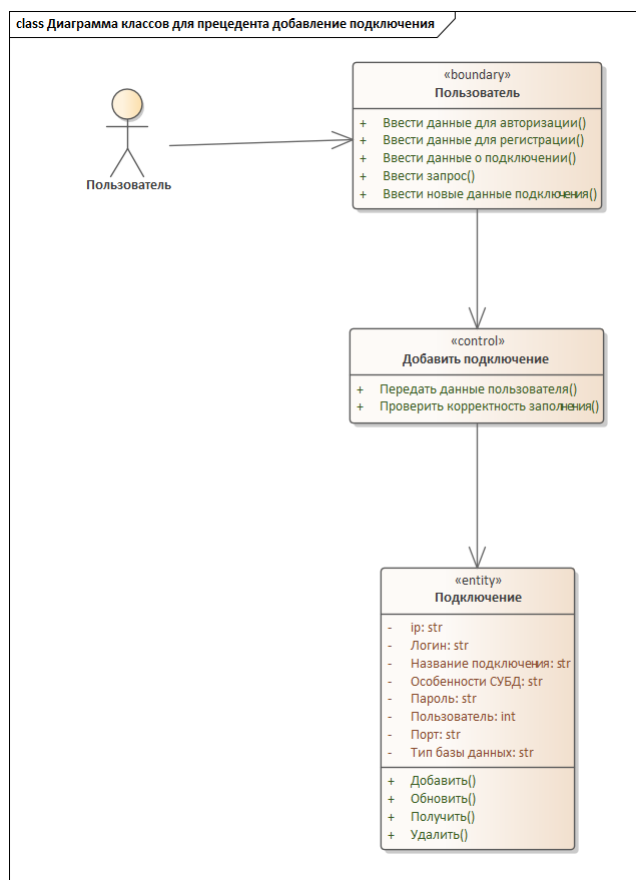


Рисунок 21 – Диаграмма классов для прецедента добавление подключения

В результате разработки получилось следующее строение диаграммы последовательностей для прецедента добавление подключения, которая представлена на рисунке 22.

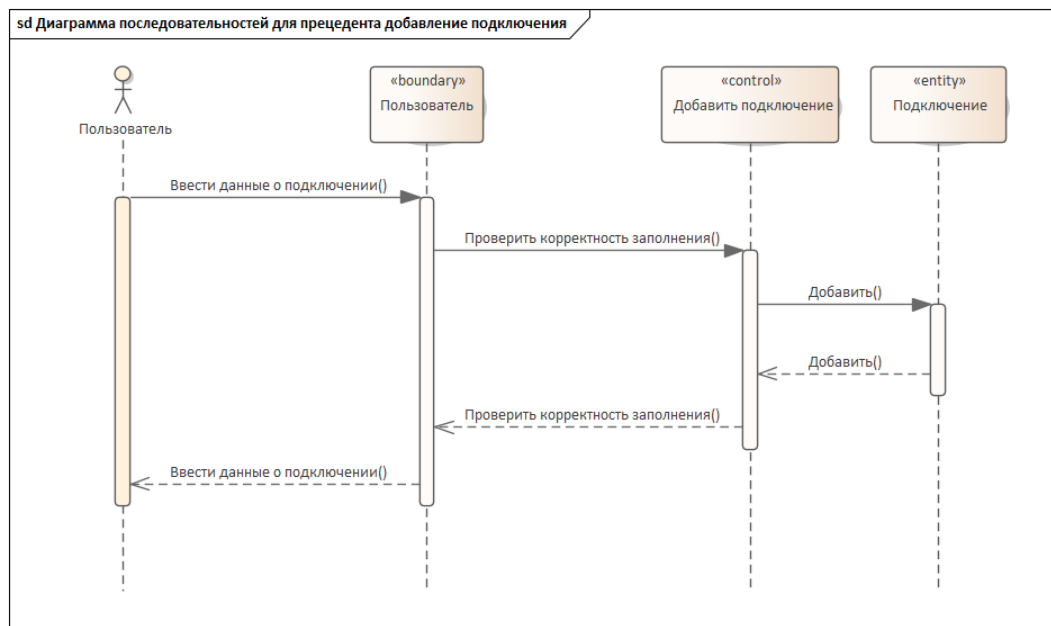


Рисунок 22 – Диаграмма последовательностей для прецедента добавление подключения

На рисунке 23 изображена разработанная диаграмма классов для прецедента редактирование подключения.

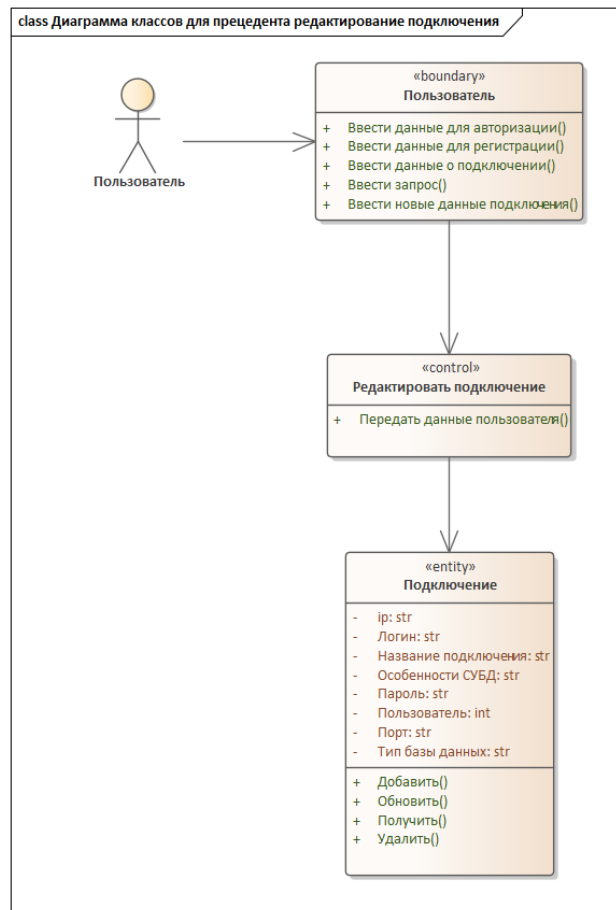


Рисунок 23 – Диаграмма классов для прецедента редактирование подключения

Созданная диаграмма последовательностей для прецедента редактирование подключения представлена на рисунке 24.

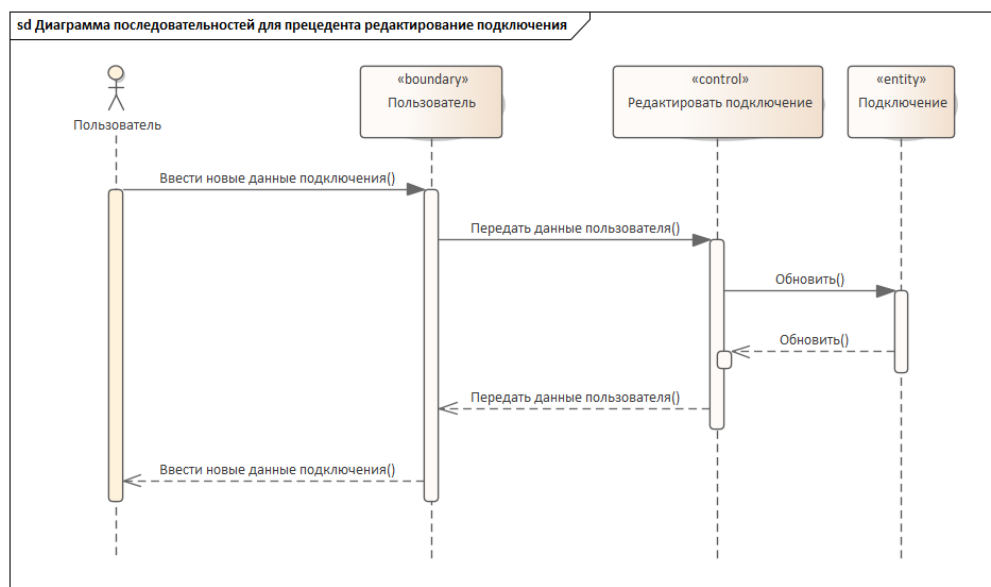


Рисунок 24 – Диаграмма последовательностей для прецедента редактирование подключения

Диаграмма классов для прецедента редактирование подключения представлена на рисунке 25.

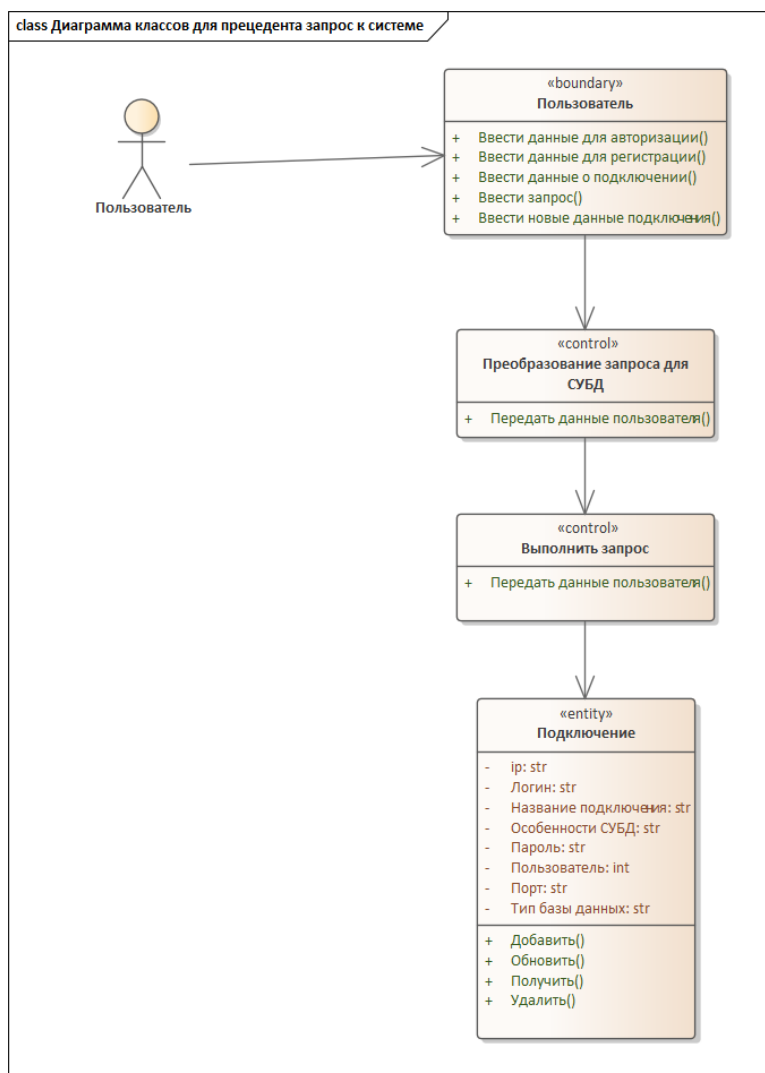


Рисунок 25 – Диаграмма классов для прецедента запрос к системе

В результате разработки получилось следующее строение диаграммы последовательностей для прецедента запрос к системе, которая представлена на рисунке 26.

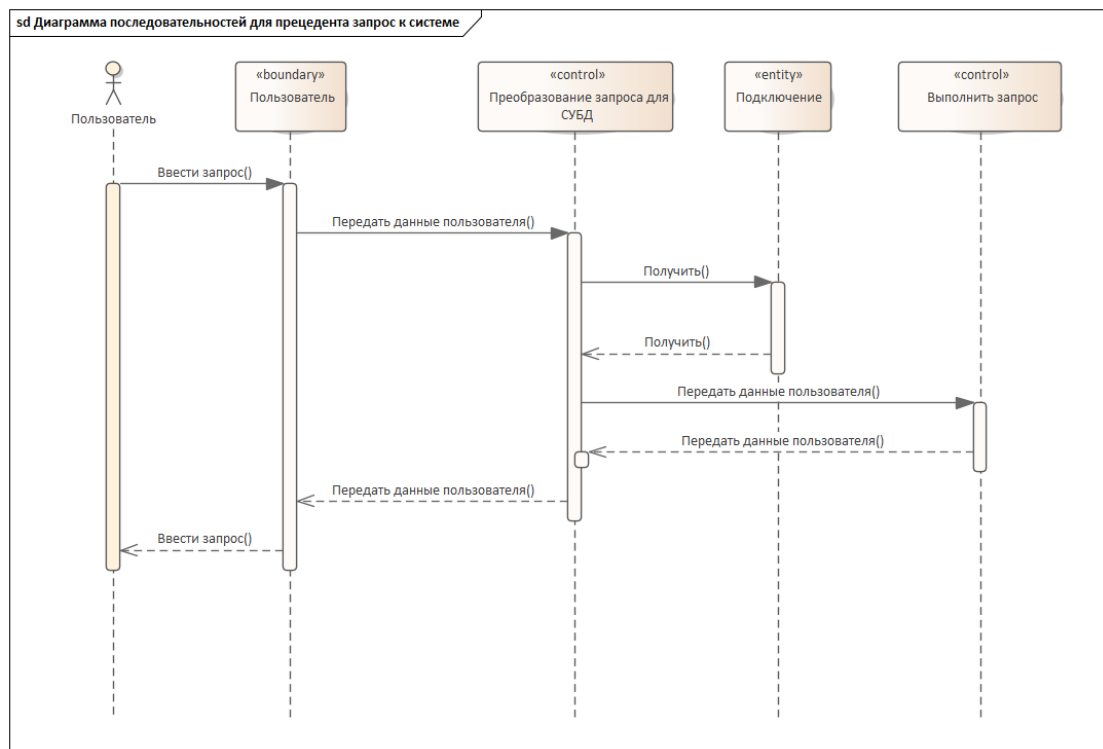


Рисунок 26 – Диаграмма последовательностей для прецедента запрос к системе

Диаграммах, показывающая пакеты анализа и сервисные пакеты в форме обобщенной диаграммы классов, представлена на рисунке 27.

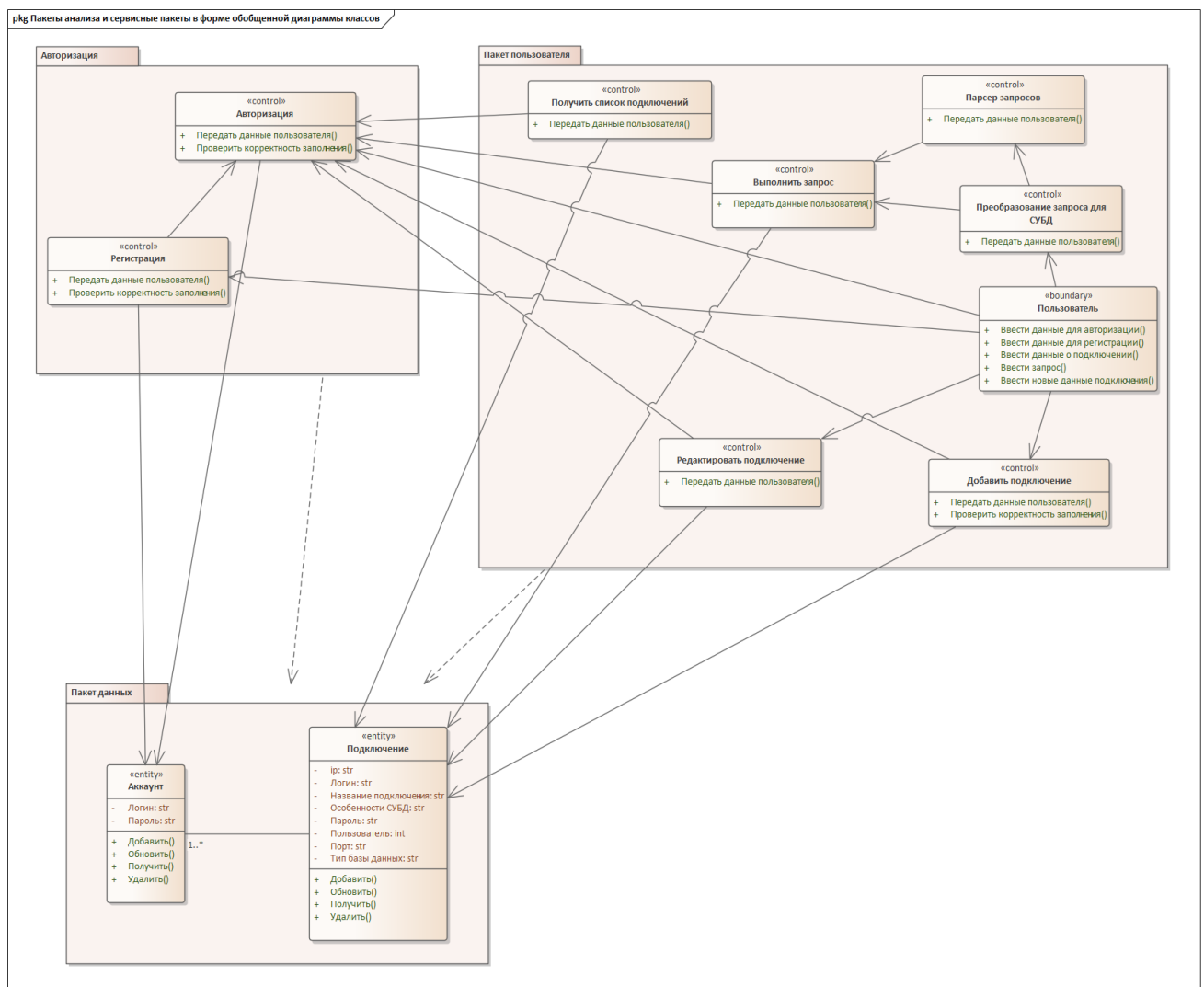


Рисунок 27 – Пакеты анализа и сервисные пакеты в форме обобщенной диаграммы классов

На рисунке 28 изображена разработанная диаграмма распределения классов проектирования по подсистемам.

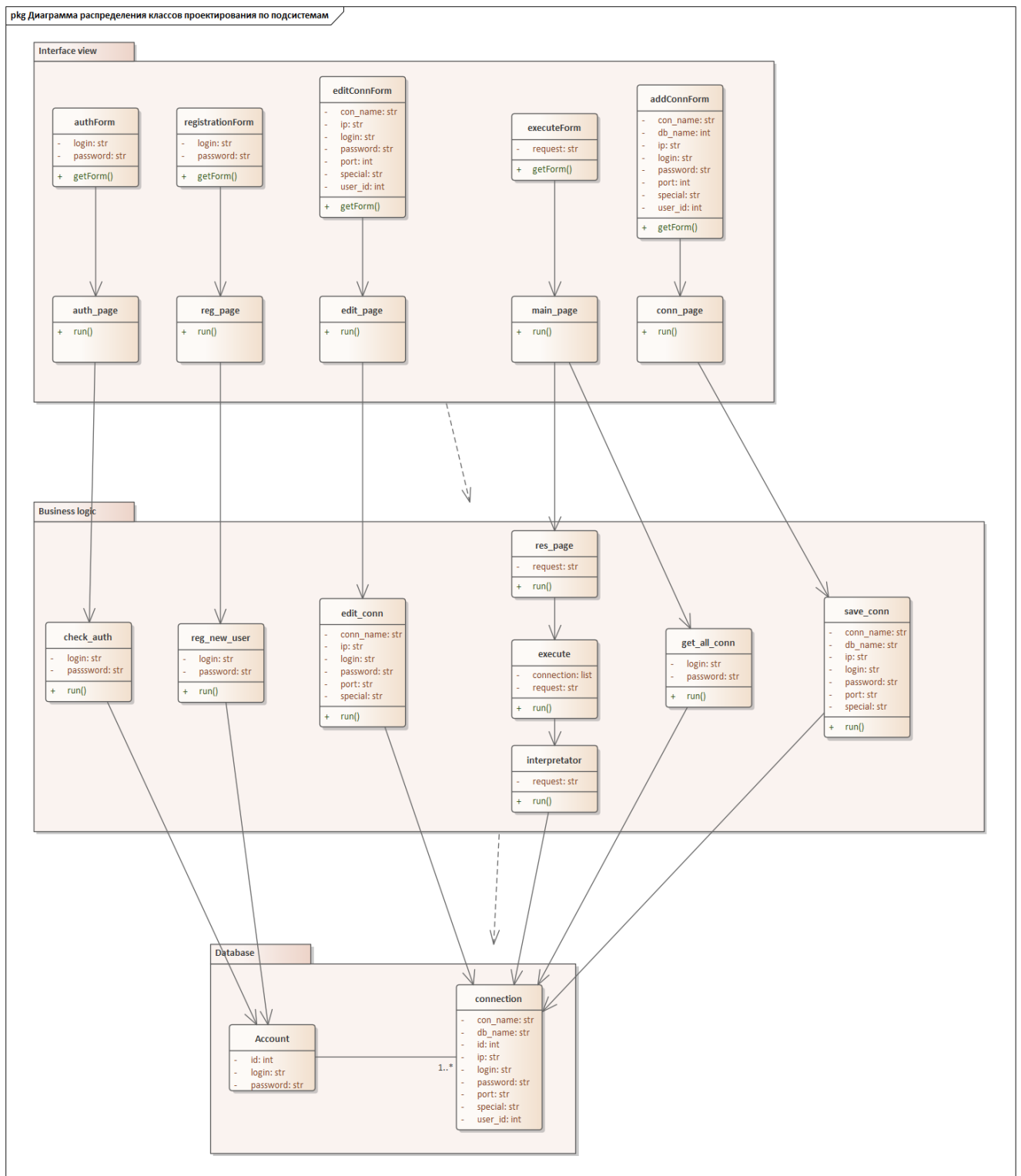


Рисунок 28 – Диаграмма распределения классов проектирования по подсистемам

Диаграмма, показывающая трассировку классов анализа в подсистемы, представлена на рисунке 29.

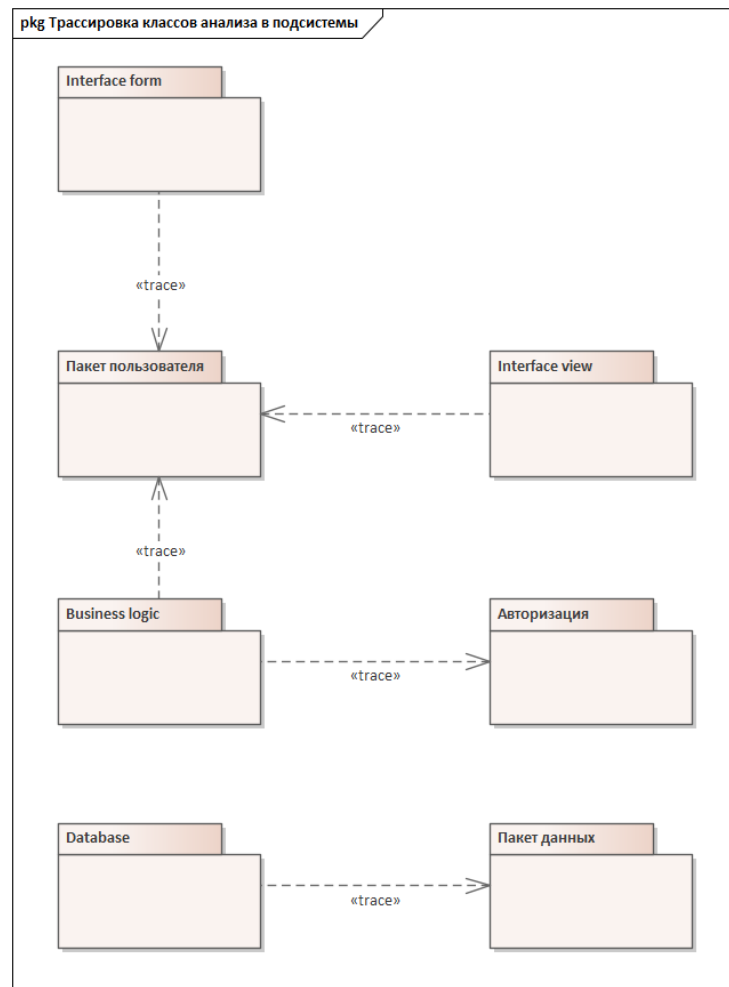


Рисунок 29 – Диаграмма трассировки классов анализа в подсистеме

Трассировка классов анализа в классы проектирования представлена на рисунке 30.

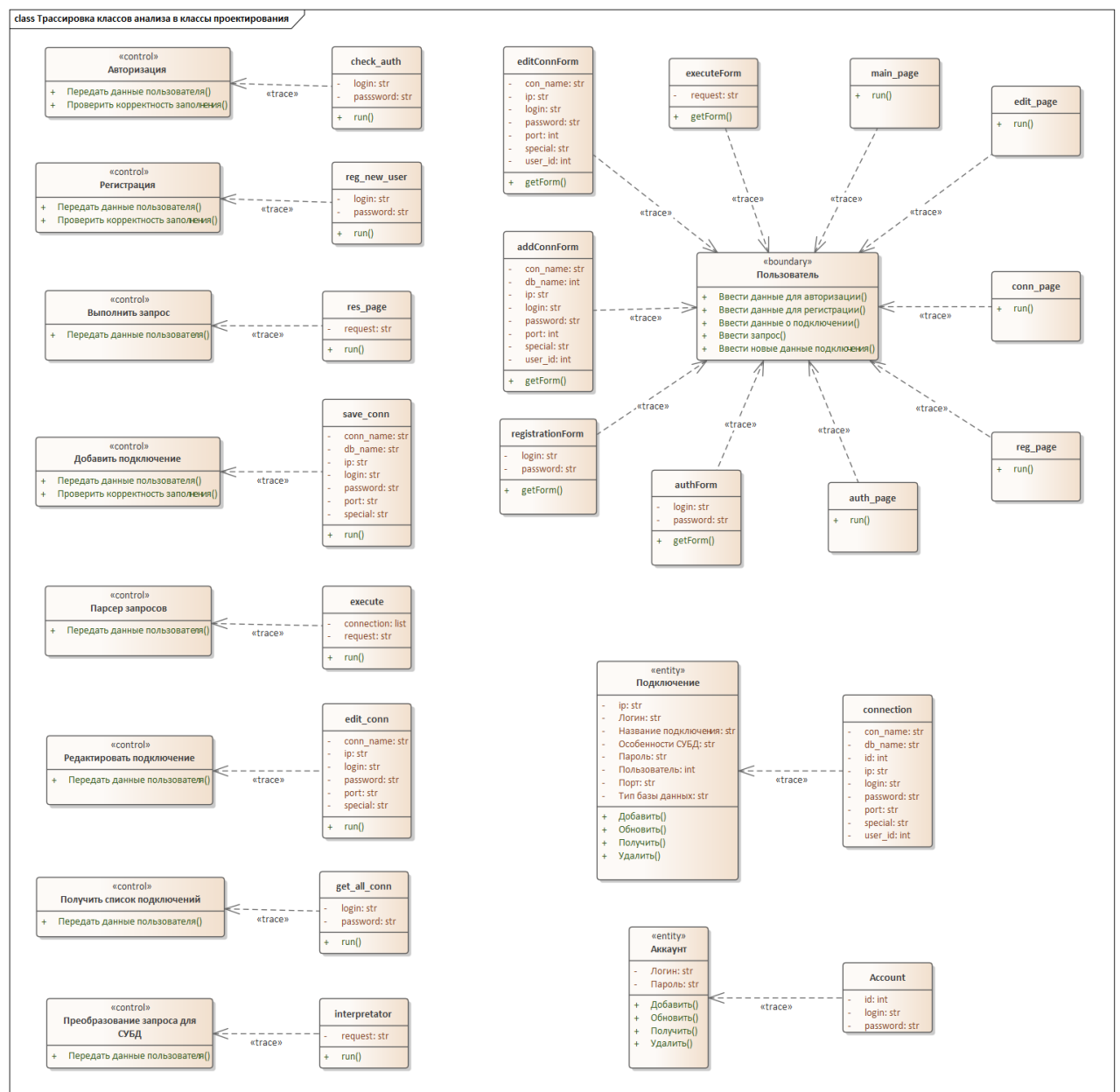


Рисунок 30 – Трассировка классов анализа в классы проектирования

В результате разработки получилось следующие строение диаграммы трассировки подсистем в компоненты, которая представлена на рисунке 31.

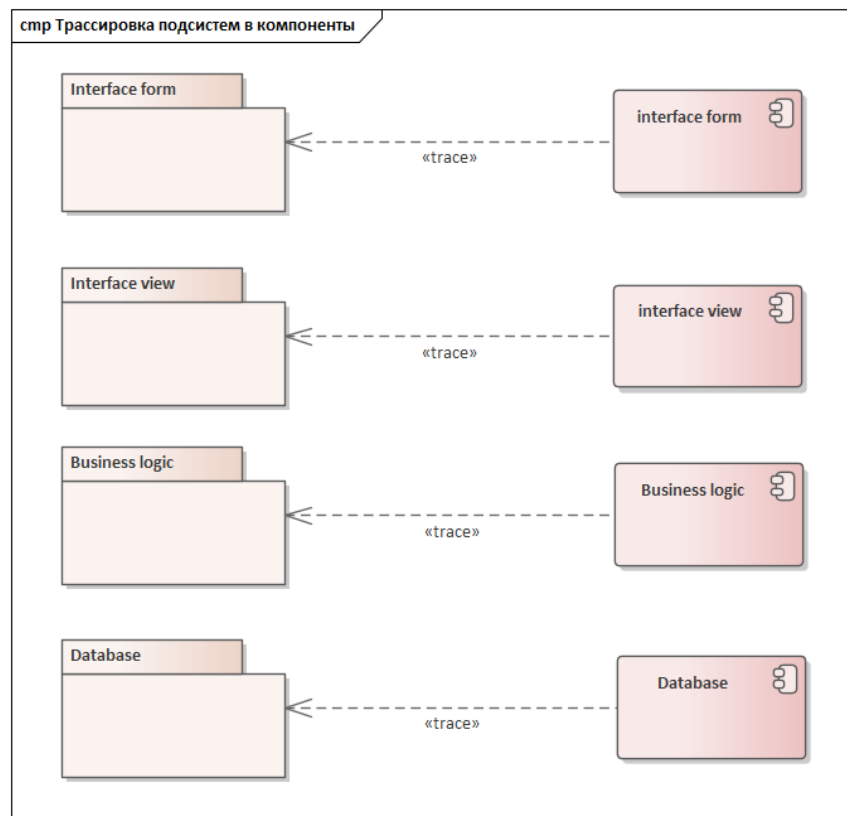


Рисунок 31 – Трассировка подсистем в компоненты

На рисунке 32 изображена трассировка классов проектирования в исходные файлы.

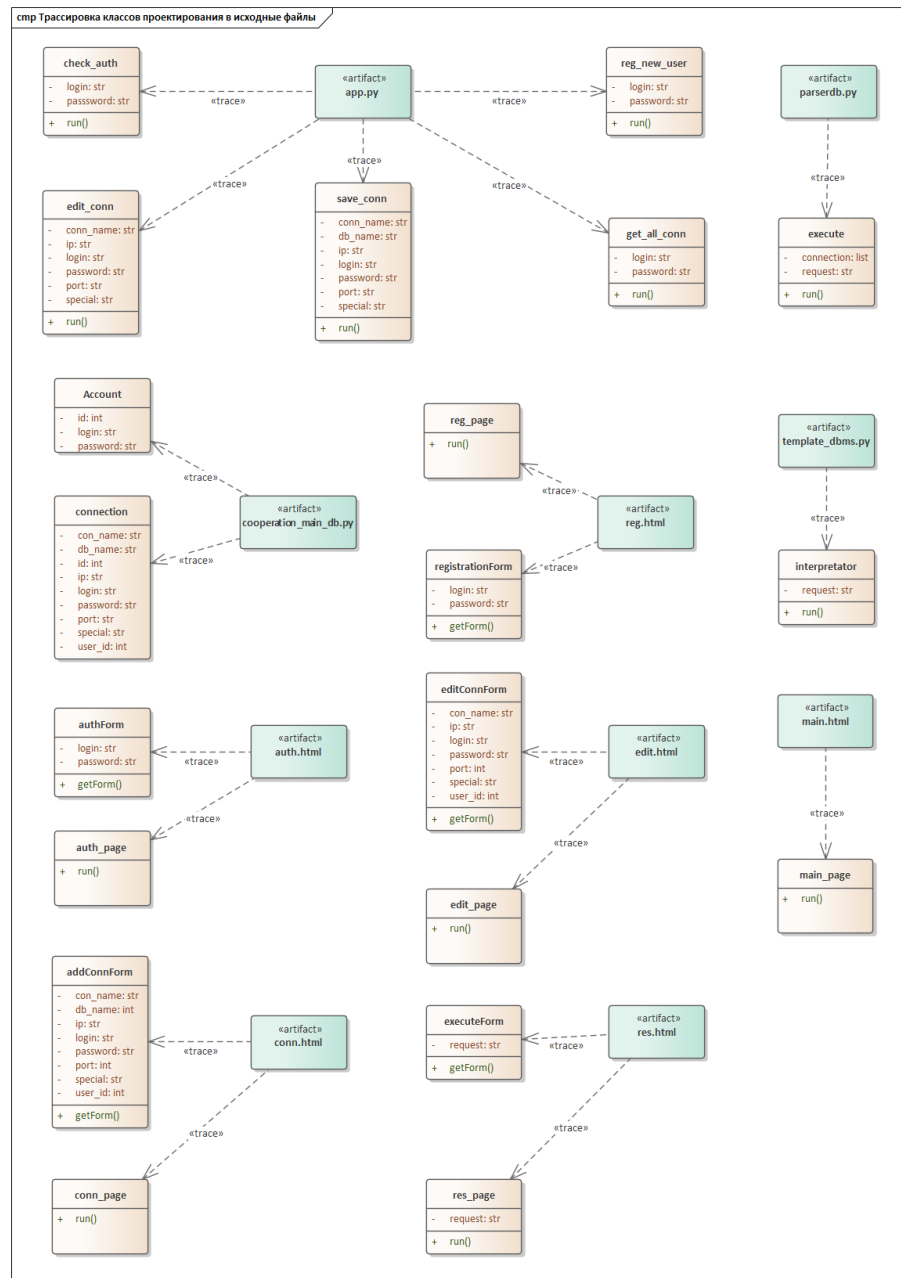


Рисунок 32 – Трассировка классов проектирования в исходные файлы

Созданная диаграмма зависимости компонентов от исходных файлов представлена на рисунке 33.

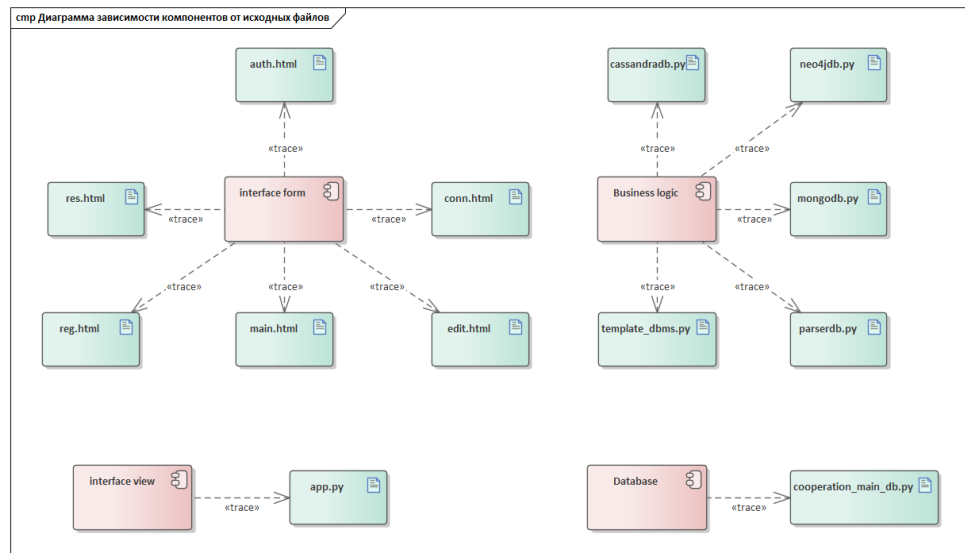


Рисунок 33 – Диаграмма зависимости компонентов от исходных файлов

Общая диаграмма классов реализации представлена на рисунке 34.

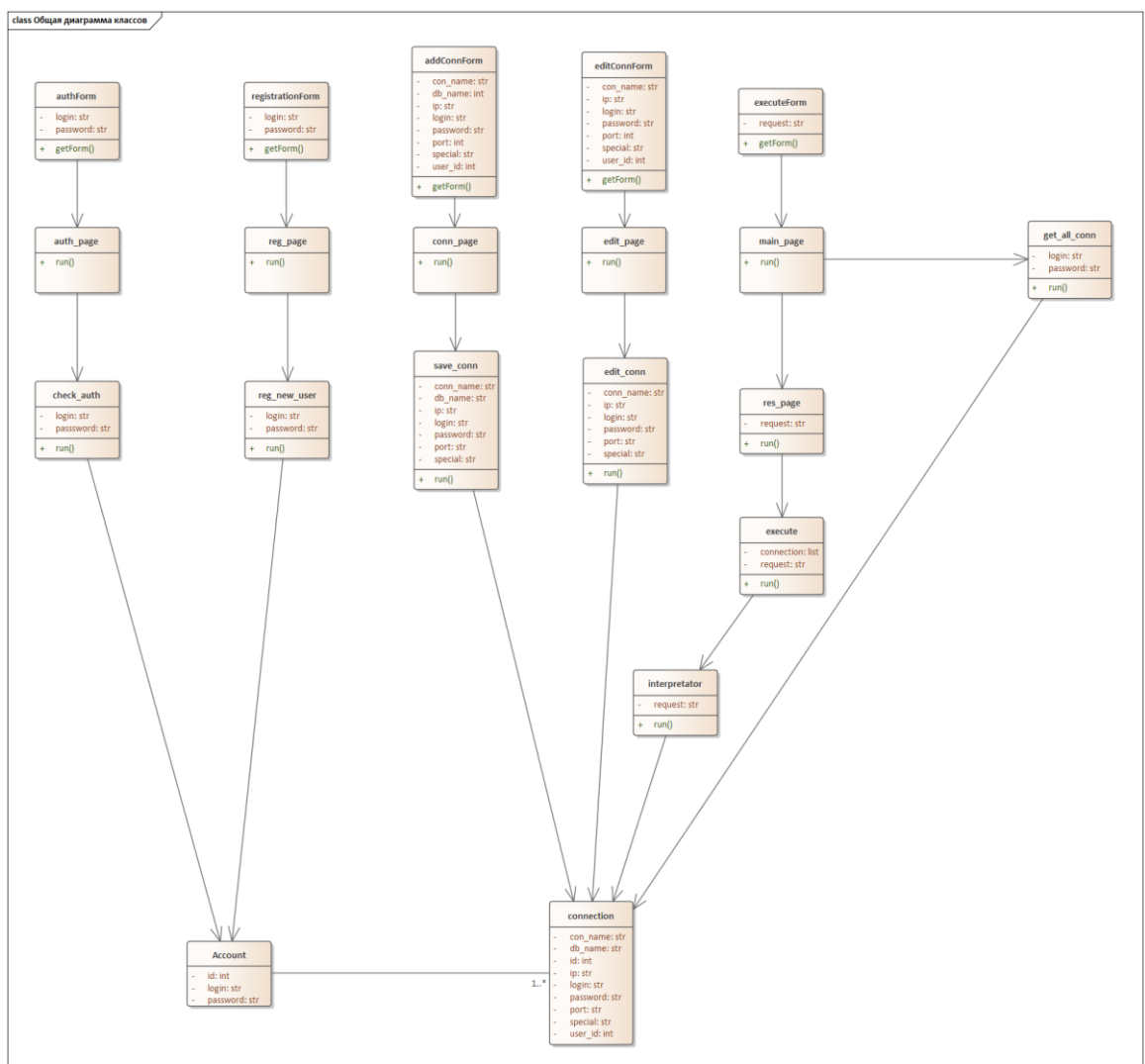


Рисунок 34 – Общая диаграмма классов

3 ТЕСТИРОВАНИЕ

3.1 Функциональное тестирование

Было проведено 3 вида тестирования – модульное, функциональное и веб-тестирование производительности.

Для функционального тестирования были выполнены необходимые запросы к таблице с данными, представленными в таблицах 3, 4, 5.

Таблица 3 – Исходные данные в таблицу для MongoDB

id	name
0	IvanovII

Таблица 4 – Исходные данные в таблицу для Neo4j

id	city_name	human_name
0	Msk	IvanovII

Таблица 5 – Исходные данные в таблицу для Cassandra

id	country_name	city_name
0	Russia	Msk

Цель функционального тестирования - проверка корректности и полноты выполняемых системой функций. Метод тестирования - Причина / Следствие (Cause/Effect — CE).

Запрос для получить данных, проходя через все СУБД:

```
cas.read(cas.country.country_name).where(cas.country.city_name=neo.read(neo.city.city_name).where(neo.city.human_name=mg.read(mg.human.name).where(mg.human.id=0)))
```

Запросы для чтения из СУБД:

- `mg.read(mg.human.name).where(mg.human.id=0)`


```

- neo.read(neo.city.city_name).where(neo.city.human_
name="IvanovII")
- cas.read(cas.country.country_name).where(cas.countr
y.city_name="Msk")

```

Запросы для добавления одного значения:

```

- mg.create(mg.human.id, mg.human.name, [[1, "Stan
Smith"]])
- neo.create(neo.city.city_name, neo.city.human_name,
[["Sp", "Stan Smith"]])
- cas.create(cas.country.id, cas.country.country_name
, cas.country.city_name, [[now(), "RF", "Sp"]])

```

Запросы для удаления данных:

```

- mg.delete.where(mg.human.name="Stan Smith")
- neo.delete.where(neo.city.human_name="Stan Smith")
- cas.delete.where(cas.country.id="Sp")

```

Запросы для множественного добавления:

```

- mg.create(mg.human.id, mg.human.name, [[1, "Stan
Smith"], [2, "John Doe"]])
- neo.create(neo.city.city_name,
neo.city.human_name, [["Sp", "Stan Smith"], ["S-p", "John
Doe"]])
- cas.create(cas.country.id, cas.country.country_name
, cas.country.city_name, [[now(), "RF", "Sp"], [now(), "R",
"S-p"]])

```

Запросы для обновления данных:

```

- mg.update(mg.human.id, mg.human.name, [3, "No
name"]).where(mg.human.name= "John Doe")
- neo.update(neo.city.city_name, neo.city.human_name,
["Nn", "No name"]).where(neo.city.human_name="John Doe"))

```

```
- cas.update(cas.country.country_name, cas.country.city_name, ["RFF", "SpS"]).where(cas.country.id="S-p"))
```

3.2 Модульное тестирование

В рамках мыли протестированы все модули и классы СУБД. Тесты реализованы с помощью библиотеки `pytestcoverage`. Цель тестирования - проверить функциональность и найти ошибки в частях приложения, которые доступны и могут быть протестированы по-отдельности. Метод тестирования - Причина / Следствие.

Пример тестирования преобразования из запроса на созданном языке в универсальный вид, который после будет интерпретироваться каждой СУБД. Сам тест приведён на рисунке 35.

```
class TestParserDB(unittest.TestCase):
    def setUp(self):
        self.mock_conn1 = MagicMock()
        self.mock_conn1.conn_name = "conn1"
        self.mock_conn1.interpreter.return_value = ("result1", "", {"time": 0.1, "pik_memory": 100})

        self.mock_conn2 = MagicMock()
        self.mock_conn2.conn_name = "conn2"
        self.mock_conn2.interpreter.return_value = ("result2", "", {"time": 0.2, "pik_memory": 200})

        self.connections = [self.mock_conn1, self.mock_conn2]

    def test_execute_simple_request(self):
        parser = ParserDB("conn1.read(conn1.table1.field1)", self.connections)
        result, err, stat = parser.execute()

        self.assertEqual(result, "result1")
        self.assertEqual(err, "")
        self.assertEqual(stat["time"], 0.1)
        self.assertEqual(stat["pik_memory"], 100)
        self.mock_conn1.interpreter.assert_called_with(['conn1', 'read', 'conn1.table1.field1', -1, -1])
```

Рисунок 35 – Тестирование преобразования из запроса на созданном языке в универсальный вид

Входными данными является запрос:

```
conn1.read(conn1.table1.field1
```

Результат тестирования примера, представлен на рисунке 36.

```

===== test session starts =====
collecting ... collected 1 item

test_parser.py::TestParserDB::test_execute_simple_request PASSED [100%]

===== 1 passed in 0.04s =====

```

Рисунок 36 – Результат тестирования

Покрытие проекта тестами с помощью ранее указанной библиотеки. Покрытие кода тестами представлено на рисунке 37 и составляет 80%.

Module ↑	statements	missing	excluded	coverage
app.py	195	58	0	70%
cassandradb.py	258	67	0	74%
cooperation_main_db.py	50	0	0	100%
mongodb.py	333	109	0	67%
neo4jdb.py	302	96	0	68%
parserdb.py	212	54	0	75%
template_dbms.py	7	3	0	57%
test_app.py	121	1	0	99%
test_cassandra.py	149	1	0	99%
test_main_db.py	71	1	0	99%
test_mongo.py	113	1	0	99%
test_neo4j.py	122	1	0	99%
test_parser.py	58	1	0	98%
Total	1991	393	0	80%

Рисунок 37 – Покрытия кода тестами

3.3 Веб-тестирование производительности

Было произведено веб-тестирование производительности, цель которого – оценка скорости, стабильности и масштабируемости веб-приложения. Метод тестирования - Причина / Следствие. Пример результата тестирования выполнения запроса представлен на рисунке 38.

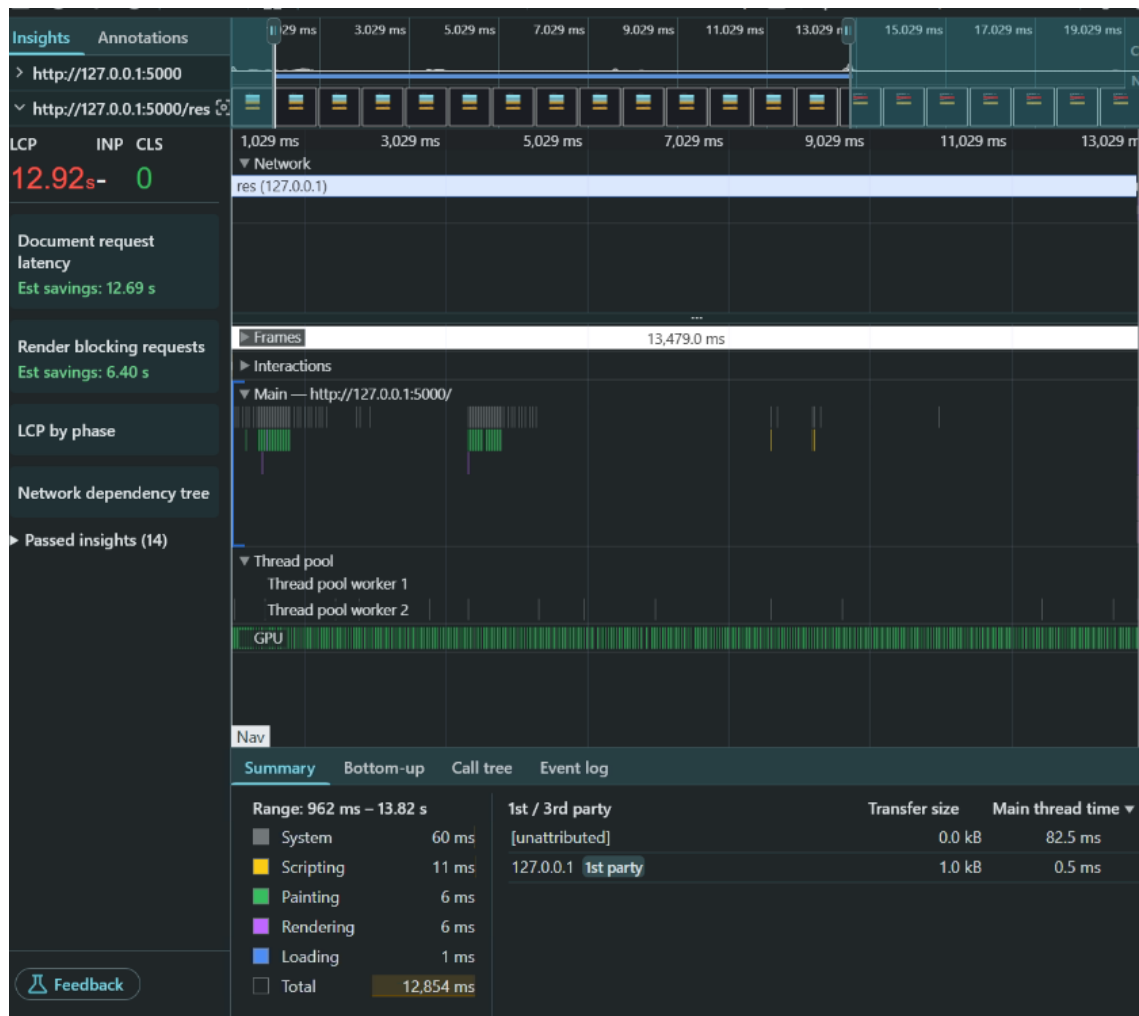


Рисунок 38 – Результат веб-тестирования производительности выполнения запроса

Для тестирования был выбран следующий запрос:

```
cas.read(cas.country.country_name).where(cas.country.city_name=neo.read(neo.city.city_name).where(neo.city.human_name=mg.read(mg.human.name).where(mg.human.id=0)))
```

Полученные результаты тестирования выполнения запроса:

- Загрузка (Loading) страницы заняла 1 мс.
- Выполнение и загрузка скрипта (Scripting) страницы заняло 11 мс.
- Рендеринг, а именно отрисовка страницы (Rendering) страницы занял 6 мс.
- Погрузка css - стилей страницы (Painting) страницы заняла 6 мс.

- Погрузка системных браузерных настроек (System) страницы заняла 60 мс.

- Общая загрузка (Total) заняла 12,854 мс.

Так же было выполнение тестирования загрузки основной страницы. При загрузке основной страницы в качестве входных данных должны иметься 3 подключения к разным БД, которые получает браузер пользователя.

Полученные результаты загрузки основной страницы:

- Загрузка (Loading) страницы заняла 1 мс.
- Выполнение и загрузка скрипта (Scripting) страницы заняло 2 мс.
- Рендеринг, а именно отрисовка страницы (Rendering) страницы занял 6 мс.

- Погрузка css - стилей страницы (Painting) страницы заняла 5 мс.

- Погрузка системных браузерных настроек (System) страницы заняла 52 мс.

- Общая загрузка (Total) заняла 4,512 мс.

Заключение

В данной работе для эффективного взаимодействия с СУБД были изучены синтаксисы для СУБД, входящих в систему. Для каждой СУБД были изучены запросы, которые необходимы для запросов системы при взаимодействии с ними.

Для выделения преимуществ и недостатков системы были изучены аналогичные работы по данной тематике, разные озёра данных и их строение в целом. Была разработана грамматика языка для запросов к системе, алгоритм деления этого запроса на подзапросы для каждой СУБД и функции преобразования в запрос на язык СУБД, для которой он предназначен.

В итоге, в данной работе была разработана и протестирована с помощью 3 видов тестирования система, которая позволяет пользователю получать и анализировать информацию из разных БД, основанных на разных СУБД.

Список использованных источников

1. Бердыбаев Р.Ш., Гнатюк С.О., Тынимбаев С., Азаров И.С. Анализ современных баз данных для использования в системах Siem // Вестник Алматинского университета энергетики и связи. — 2021. — Т. 54, № 3. — С. 33—47.
2. Blinova O.V., Pankratova E.V., Farkhadov M.P. Principles of construction and analysis of architectures for modern scientific information systems // 7th International Scientific Conference. — 2023. — P. 111—113.
3. Schäfer R.A., Rabsch D., Scholz G.E., Stadler P.F., Hess W.R., Backofen R., Fallmann J., Voß B. RNA interaction format: a general data format for RNA interactions // Bioinformatics. — 2023. — Vol. 39, № 11. — P. 1—3.
4. Martha V.S., Liu Z.H., Guo L., Su Zh., Ye Ya., Fang H., Ding D., Tong W., Xu X. Constructing a robust protein-protein interaction network by integrating multiple public databases // Bioinformatics. — 2011. — Vol. 12, № 10. — P. 1—10.
5. Панов П.В. База данных «субнациональный регионализм и многоуровневая политика (reg-mlg)» // Вестник Пермского университета. ПОЛИТОЛОГИЯ. — 2021. — Т. 15, № 4. — С. 111—120.
6. Haas L.M., Rice J.E., Schwarz P.M., Swope W.C. DISCOVERYLINK: a system for integrated access to life sciences data sources // IBM Systems Journal. — 2001. — Vol. 40, № 2. — P. 489—511.
7. Zhang L., Pang K., Xu J., Niu B. JSON-based control model for SQL and NoSQL data conversion in hybrid cloud database // Journal of Cloud Computing. — 2022. — Vol. 11, № 1. — P. 1—12.
8. Płuciennik E., Zgorzałek K. The Multi-model Databases – A Review // Communications in Computer and Information Science. — 2017. — Vol. 716. — P. 141—152.
9. Seeger M. Key-Value stores: a practical overview. — Stuttgart, Germany, 2009.

10. Angles R., Gutierrez C. Survey of graph database models // ACM Computing Surveys. — 2008. — Vol. 40. — P. 1—39.
11. Khalil A., Belaissaoui M. An Approach for Implementing Online Analytical Processing Systems under Column-Family Databases // IAENG International Journal of Applied Mathematics. — 2023. — Vol. 53. — P. 31—39.
12. What is a Document Database? [Электронный ресурс]. — URL: <https://www.mongodb.com/document-databases> (дата обращения: 17.12.2023).
13. Bradshaw S. MongoDB: The Definitive Guide. / Brazil E., Chodorow K. — Boston: O'Reilly Media, Inc., 2019. — 511 p.
14. Query Documents [Электронный ресурс]. — URL: <https://www.mongodb.com/docs/manual/tutorial/query-documents/> (дата обращения: 17.12.2024).
15. What is Neo4j? [Электронный ресурс]. — URL: <https://neo4j.com/docs/getting-started/whats-neo4j> (дата обращения: 17.12.2023).
16. What is Apache Cassandra? [Электронный ресурс]. — URL: https://cassandra.apache.org/_/index.html (дата обращения: 17.12.2023).
17. Data lake [Электронный ресурс]. — URL: https://en.wikipedia.org/wiki/Data_lake (дата обращения: 05.06.2024).
18. Чем озеро данных отличается от базы и зачем оно нужно аналитикам [Электронный ресурс]. — URL: <https://practicum.yandex.ru/blog/chto-takoe-ozera-dannyh> (дата обращения: 05.06.2024).
19. Что такое озера данных и почему в них дешевле хранить big data [Электронный ресурс]. — URL: <https://cloud.vk.com/blog/chto-takoe-ozera-dannyh-i-zachem-tam-hranyat-big-data> (дата обращения: 05.06.2024).
20. Pwint Phyu Khine, Zhao Shun Wang. Data Lake: A New Ideology in Big Data Era. — Wuhan, China, 2017.
21. Data Lake [Электронный ресурс]. — URL: <https://yandex.cloud.ru/docs/glossary/datalake> (дата обращения: 05.06.2024).

22. What is a Data Swamp? [Электронный ресурс]. — URL: <https://medium.com/codex/what-is-a-data-swamp-38b1aed54dc6> (дата обращения: 05.06.2024).
23. Google Cloud Data Lake: 4 Phases of the Data Lake Lifecycle [Электронный ресурс]. — URL: <https://bluexp.netapp.com/blog/gcp-cvo-blg-google-cloud-data-lake-4-phases-of-the-data-lake-lifecycle> (дата обращения: 05.06.2024).
24. Pentaho, Hadoop, and Data Lakes [Электронный ресурс]. — URL: <https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/> (дата обращения: 05.06.2024).
25. HDFS [Электронный ресурс]. — URL: <https://bigdataschool.ru/wiki/hdfs> (дата обращения: 05.06.2024).
26. AWS Lake Formation: Overview, Architecture & Functionality [Электронный ресурс]. — URL: <https://k21academy.com/amazon-web-services/aws-data/aws-lake-formation/> (дата обращения: 05.06.2024).
27. Azure Data Lake [Электронный ресурс]. — URL: <https://azure.microsoft.com/ru-ru/solutions/data-lake> (дата обращения: 05.06.2024).
28. Sheth A.P. When will we have true heterogeneous database systems // Proceedings of the 1987 Fall Joint Computer Conference (ACM '87). — IEEE Computer Society Press, 1987. — P. 747—748.
29. Shalina O.V., Andrianova E.E. The Study of Combining SQL and NoSQL Databases in a Heterogeneous System for the Development of a Project Management Database // Theoretical & Applied Science. — 2024. — № 5 (133). — P. 14—17.
30. Parent C., Spaccapietra S. Issues and Approaches of Database Integration // Communications of the ACM. — 1998. — Vol. 41, № 5es. — P. 166—178.
31. Luo T., Lee R., Mesnier M., Chen F., Zhang X. hStorage-DB: Heterogeneity-aware Data Management to Exploit the Full Capability of Hybrid Storage Systems // Proceedings of the VLDB Endowment. — 2012. — Vol. 5.
32. Papazoglou M., Bouguettaya A. On Building a Hyperdistributed Database // Information Systems. — 1995. — Vol. 20, № 5.

33. Lu J., Liu Z., Xu P., Zhang C. UDBMS: Road to Unification for Multi-model Data Management // Advances in Databases and Information Systems (ADBIS). — 2018. — P. 330—344.
34. Ghulam A. A Multidatabase System as 4-Tiered Client-Server Distributed Heterogeneous Database System // International Journal of Computer Science and Information Security. — 2009. — Vol. 6. — P. 10—14.
35. Beneventano D., Bergamaschi S., Gelati G., Guerra F., Vincini M. MIKS: An Agent Framework Supporting Information Access and Integration // Dipartimento Ingegneria dell'Informazione, Università di Modena. — 2002.
36. Catarci T., Santucci G., Cardiff J. Graphical Interaction with Heterogeneous Databases // The VLDB Journal. — 1997. — Vol. 6.

ПРИЛОЖЕНИЕ А

В графическую часть выпускной квалификационной работы входят:

А.1. Формула для описания грамматики созданного языка запросов.

А.1 Формула для описания грамматики созданного языка запросов

$$\Gamma = \{V_T, V_A, \langle I \rangle, R\} \quad (1)$$

$$V_T = \{a, b, \dots, z, A, B, \dots, Z, 0, 1, \dots, 9, ., (,), ,, =, !, <, >, <=, >=, AND, OR, where, create, read, update, delete, ", [,], create_index, delete_index, exec, index, all, show, count\}$$

$$V_A = \{\langle I \rangle, \langle \text{операция чтения} \rangle, \langle \text{доступ} \rangle, \langle \text{условие} \rangle, \langle \text{операция} \rangle, \langle \text{подключение} \rangle, \langle \text{таблица} \rangle, \langle \text{столбец} \rangle, \langle \text{сравнение} \rangle, \langle \text{константа} \rangle, \langle \text{строка} \rangle, \langle \text{буква} \rangle, \langle \text{цифра} \rangle, \langle \text{число} \rangle, \langle \text{список} \rangle, \langle \text{элемент списка} \rangle, \langle \text{список списков} \rangle\}$$

$$R = \{$$

$$\langle I \rangle \rightarrow \langle \text{подключение} \rangle. \langle \text{операция} \rangle,$$

$$\langle \text{операция чтения} \rangle \rightarrow read(\langle \text{доступ} \rangle) | read(\langle \text{доступ} \rangle). where(\langle \text{условие} \rangle),$$

$$\begin{aligned} \langle \text{операция} \rangle \rightarrow & create(\langle \text{доступ} \rangle, \langle \text{доступ} \rangle, [\langle \text{список списков} \rangle]) | update(\langle \text{доступ} \rangle, \langle \text{доступ} \rangle, [\langle \text{список} \rangle]), | update(\langle \text{доступ} \rangle, \langle \text{доступ} \rangle, \langle \text{список} \rangle). where(\langle \text{условие} \rangle) | \\ & delete. where(\langle \text{условие} \rangle) | create_index(\langle \text{константа} \rangle, \langle \text{константа} \rangle, [\langle \text{список} \rangle]), delete_index(\langle \text{константа} \rangle) | \\ & exec(\langle \text{константа} \rangle) | index(\langle \text{константа} \rangle) | all() | show(\langle \text{константа} \rangle) | count(\langle \text{константа} \rangle), \end{aligned}$$

$$\langle \text{доступ} \rangle \rightarrow \langle \text{подключение} \rangle. \langle \text{таблица} \rangle. \langle \text{столбец} \rangle,$$

$$\begin{aligned} \langle \text{условие} \rangle \rightarrow & \langle \text{доступ} \rangle \langle \text{сравнение} \rangle \langle \text{константа} \rangle | \langle \text{условие} \rangle AND \langle \text{условие} \rangle | \langle \text{условие} \rangle OR \langle \text{условие} \rangle | \\ & \langle \text{доступ} \rangle \langle \text{сравнение} \rangle \langle \text{операция чтения} \rangle, \end{aligned}$$

$$\langle \text{сравнение} \rangle \rightarrow = | ! = | < | > | \leq | \geq,$$

$$\langle \text{константа} \rangle \rightarrow " \langle \text{строка} \rangle " | \langle \text{число} \rangle,$$

$$\langle \text{подключение} \rangle \rightarrow \langle \text{строка} \rangle,$$

$$\langle \text{таблица} \rangle \rightarrow \langle \text{строка} \rangle,$$

$$\langle \text{столбец} \rangle \rightarrow \langle \text{строка} \rangle,$$

$\langle \text{строка} \rangle \rightarrow \langle \text{строка} \rangle \langle \text{буква} \rangle \mid \langle \text{строка} \rangle \langle \text{цифра} \rangle \mid \$,$
 $\langle \text{буква} \rangle \rightarrow a \mid b \mid \dots \mid z \mid A \mid B \mid \dots \mid Z \mid \text{а} \mid \text{б} \mid \dots \mid \text{я} \mid \text{А} \mid \text{Б} \mid \dots \mid \text{Я},$
 $\langle \text{цифра} \rangle \rightarrow 0 \mid 1 \mid \dots \mid 9,$
 $\langle \text{число} \rangle \rightarrow \langle \text{число} \rangle \langle \text{цифра} \rangle \mid \$,$
 $\langle \text{список} \rangle \rightarrow \langle \text{список} \rangle, \langle \text{элемент списка} \rangle \mid \langle \text{элемент списка} \rangle,$
 $\langle \text{элемент списка} \rangle \rightarrow \langle \text{константа} \rangle \mid \langle \text{элемент списка} \rangle, \langle \text{константа} \rangle,$
 $\langle \text{список списков} \rangle \rightarrow \langle \text{список} \rangle \mid \langle \text{список списков} \rangle, \langle \text{список} \rangle \}$

ПРИЛОЖЕНИЕ В ТЕХНИЧЕСКОЕ ЗАДАНИЕ

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. Н.Э. Баумана

Кафедра «Системы обработки информации и управления»

Утверждаю
Заведующий кафедрой ИУ-5
_____ В.И.Терехов
"__" _____ 2025 г.

Согласовано
Научный руководитель
_____ А.Э. Самохвалов
"__" _____ 2025 г.

Система автоматического сбора информации о работе NoSQL баз данных

Техническое задание
(вид документа)

писчая бумага
(вид носителя)

4
(количество листов)

ИСПОЛНИТЕЛЬ:

_____ Журавлев Николай Вадимович
"__" _____ 2025 г.

Москва - 2025

1. Введение

Система автоматического сбора информации о работе NoSQL баз данных, которая может применяться для анализа данных из различных СУБД.

2. Основания для разработки

Основанием для разработки является задание на выпускную квалификационную работу, подписанное руководителем выпускной работы и утверждённое заведующим кафедрой ИУ5 МГТУ им. Н.Э. Баумана 15 декабря 2024 года.

3. Назначение разработки

Назначением работы является разработка системы для взаимодействия пользователя с множеством баз данных, находящимися в различных системах управления базами данных.

4. Требования к программе или программному изделию

4.1 Требования к функциональным характеристикам;

Программа должна выполнять следующие функции:

- подключение различные СУБД;
- выполнять запросы на созданном языке запросов;
- просматривать результат исполнения в web-браузере;
- авторизоваться под своей учётной записью в системе.

Требования к входным данным:

- Запрос, составленный на разработанном языке для взаимодействия между СУБД.

Требования к выходным данным:

- Выводить результат исполнения написанного запроса в читаемом для человека виде;
- Результат должен быть типа, который возможно отобразить в формате строки.

5. Требования к документации

По окончании работы предъявляется следующая техническая документация:

1. Техническое задание;
2. Расчётно-пояснительная записка;
3. Графический материал по проекту в формате презентации.

6. Техничко-экономические показатели

Требования к данному разделу не предъявляются

7. Стадии и этапы разработки

График выполнения отдельных этапов работ приведен в соответствии с приказом об организации учебного процесса в 2024/2025 учебном году.

Таблица 1 – Этапы разработки

№ п/п	Наименование этапа	Содержание работы	Дата исполнения
1.	Анализ ПО	Анализ задач, подлежащих реализации	<u>15.02.2025</u>
2.	Составление ТЗ	Формулировка требований, составление технического задания	<u>01.03.2025</u>
3.	Создание программного изделия	Разработка алгоритмов работы программы и архитектуры	<u>20.04.2025</u>
4.	Тестирование программного изделия	Устранение ошибок в программе	<u>30.04.2025</u>
5.	Отладка программного изделия	Завершение тестирования и отладка системы	<u>10.05.2025</u>
6.	Оформление технической документации	Описание технической части работы	<u>11.05.2025</u>

№ п/п	Наименование этапа	Содержание работы	Дата исполнения
7.	Оформление графической документации		<u>15.05.2025</u>
8.	Заключение руководителя	Завершающий этап подготовки ВКРМ	<u>27.05.2025</u>
9.	Защита работы	Защита	<u>09.06.2025</u>

8. Порядок контроля и приёмки

Приём и контроль работы осуществляется на защите выпускной квалификационной работы.