Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

«Агрегатор реляционных баз данных»

Студент ИУ9-82(Б) (Группа)	(Подпись, дата)	Д. С. Чугунов (И.О.Фамилия)
Руководитель ВКР	(Подпись, дата)	А. В. Коновалов (И.О.Фамилия)
Консультант	(Подпись, дата)	(И.О.Фамилия)
Консультант	(Подпись, дата)	(И.О.Фамилия)
Нормоконтролер	(Подпись, дата)	С. Ю. Скоробогатов (И.О.Фамилия)

2020 г.

АННОТАЦИЯ

Темой данной работы является «Агрегатор реляционных баз данных». Объем работы составляет 48 страниц. Для ее написания было использовано 10 источников. В работе содержится 3 рисунка и 12 листингов.

Основной объект исследование — декомпозиция запроса SQL, объединяющего данные из различных баз данных, на подзапросы к этим базам данных.

Цель работы — разработка и реализация инструмента, позволяющего выполнять запросы SQL, которые объединяют данные из различных баз данных.

Дипломная работа состоит из пяти глав. В первой главе рассматриваются основные понятия, используемые в данной работе. Во второй главе дается расширенная постановка задачи. В третьей главе рассматриваются основные подходы для решения поставленной задачи. В четвертой главе приведены особенности реализации, а также руководство пользователя. В пятой главе рассматривается корректность работы программы.

СОДЕРЖАНИЕ

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

СУБД — Система управления базами данных.

РСУБД — Реляционная система управления базами данных.

NoSQL (англ. not only SQL, не только SQL) — ряд подходов, направленных на реализацию баз данных, имеющих существенные отличия в архитектуре по сравнению с РСУБД.

РБНФ — Расширенная форма Бэкуса — Наура.

ДНФ — Дизъюнктивная нормальная форма.

КНФ — Конъюнктивная нормальная форма.

СДНФ — Совершенная дизъюнктивная нормальная форма.

СКНФ — Совершенная конъюнктивная нормальная форма.

ODBC (англ. Open Database Connectivity) — программный интерфейс доступа к базам данных.

ВВЕДЕНИЕ

В современном мире при решении многих прикладных задач используют различные базы данных. Зачастую в больших проектах может использоваться сразу несколько баз данных, которые имеют различия в архитектуре. Причин для этого может быть много, начиная от исторических, когда два отдельных друг от друга проекта сливаются в один, заканчивая техническими, когда для решения различных задач требуются различные по архитектуре базы данных.

Практически во всех таких больших проектах различные базы данных обрабатывают информацию, которую необходимо обрабатывать совместно друг с другом. Например, в одной базе данных храниться информация о розничной торговле, а в другой базе данных хранится информация об оптовой торговле и в обеих базах данных обрабатывается информация об одинаковых товарах. Предположим, для нужд бизнеса, необходимо рассчитать статистику самых востребованных товаров, для этого необходимо объединить данные из различных баз данных. На сегодняшний день существует очень мало баз данных, которые могут работать с данными из внешних источников, это связано в первую очередь с уникальностью архитектур различных СУБД.

Решить поставленную задачу можно несколькими способами:

- Объединить несколько базы данных в одну и использовать одну общую базу данных. Этот способ не всегда возможно применить, так как иногда для решения определенной задачи необходимо использовать конкретную архитектуру базы данных.
- Создание хранилища данных, в которое будет стекаться информация из различных источников. Этот способ является наиболее верным, так как не влияет на существующую архитектуру, но для создания и поддержания хранилища данных необходимы значительные материальные затраты.

- Использование инструмента, который позволяет обрабатывать информацию из различных источников. Этот подход также не влияет на существующую архитектуру и при этом не влечет за собой значительных материальных затрат.

Задачей данной дипломной работы является реализация программного комплекса, предназначенного для локального использования, который способен обрабатывать данные из различных внешних источников.

1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Обзор языка SQL

Различные СУБД отличаются своими архитектурами, поэтому для удобства пользователя практически все из них поддерживают язык SQL для работы с данными. SQL — декларативный язык программирования, применяемый для создания, модификации и управления данными в РСУБД. Хотя язык SQL и предназначен для работы с РУСБД, но некоторые NoSQL базы данных также поддерживают работу с языком SQL. Каждая СУБД работает с определенным диалектом языка SQL, но основные конструкции остаются неизменными.

С точки зрения реализации язык SQL можно представить в виде множества операторов, которые можно разбить на 4 группы:

- Data Definition Language (DDL) — операторы, которые отвечают за управление структурой базы данных.
- Data Manipulation Language (DML) — операторы, которые отвечают за манипулирование данными.
- Data Control Language (DCL) — операторы, которые отвечают за определением доступа к данным.
- Transaction Control Language (TCL) — операторы, которые отвечают за управлением транзакциями.

Группы операторов DDL, DCL, TCL не работают напрямую с данными, поэтому данные операторы не будут рассматриваться в рамках данной работы.

Группа операторов DML включает в себя четыре основных оператора:

- SELECT — отвечает за извлечение данных из базы данных
- INSERT — отвечает за вставку данных в базу данных
- UPDATE — отвечает за обновление данных в базе данных

- DELETE — отвечает за удаление данных из базы данных

Все операторы из группы DML можно расширить на работу с несколькими источниками (базами данных). В данной работе рассматривается оператор SELECT, так как именно этот оператор чаще всего необходим при работе с различными источниками.

1.2 Оператор SELECT

Оператор SELECT — оператор языка SQL из группы DML, возвращающий набор данных из базы данных. Данные оператор представляет наибольший интерес в рамках данной работы, так как именно он отвечает за возврат выборок данных.

Оператор SELECT состоит из нескольких разделов:

1. Список возвращаемых столбцов (как существующих, так и вычисляемых на их основе)
2. Табличное выражение, которое определяет базовый набор данных
3. Ограничение на строки из табличного выражения
4. Объединение строк согласно заданному правилу
5. Ограничения на сгруппированные строки
6. Сортировка результирующей выборки

В рамках данной работы рассматриваются только первые 3 раздела.

1.4 Табличные выражения

Как было сказано ранее, в операторе SELECT участвует табличное представление. В качестве табличного представления может выступать таблица из базы данных, объединенная выборок при помощи оператора UNION, а также соединение выборок при помощи оператора JOIN. В рамках данной работы оператор объединения не рассматривался.

JOIN — оператор языка SQL, который является реализацией операции соединения реляционной алгебры.

Существует несколько типов JOIN, вот самые распространенные:

1) CROSS JOIN — декартовое произведение двух выборок. Каждой строке из первой выборки ставится в соответствие строка из второй выборки. Если размер первой выборки — N, а размер второй — M, то размер результирующей — N * M.

2) INNER JOIN — из декартового произведения двух выборок выбираются только те строки, которые соответствуют условию.

3) LEFT JOIN — объединяет две выборки и извлекает все строки, для которых выполняется логическое выражение плюс строки из первой выборки, которым не соответствует ни одна строка из второй выборки. Минимальный размер результирующей выборки — размер первой выборки.

4) RIGHT JOIN — объединяет две выборки и извлекает все строки, для которых выполняется логическое выражение плюс строки из второй выборки, которым не соответствует ни одна строка из первой выборки. Минимальный размер результирующей выборки — размер второй выборки.

2 РАСШИРЕННАЯ ПОСТАНОВКА ЗАДАЧИ

После краткого описания предметной области, можно формализовать задачу данной дипломной работы.

Необходимо разработать инструмент, в дальнейшем «агрегатор баз данных», который позволяет работать с данными из различных внешних источников, в роли которых выступают различные РСУБД. Инструмент не должен сильно зависеть от аппаратной составляющей компьютера.

На входе агрегатору баз данных подается запрос на выборку данных, описанный на языке SQL (SELECT). На выходе программа выдает требуемую выборку данных, либо сообщение об ошибке.

В конструкции SELECT должны поддерживаться логические и арифметические выражения, ограничения на строки, а также в табличном представлении должны поддерживаться различные типы соединения: CROSS JOIN, INNER JOIN, LEFT JOIN, RIGHT JOIN. Данные из внешних источников могут быть следующих типов: число с плавающей точкой, целое число, логический тип и строка.

3 РАЗРАБОТКА

Агрегатор баз данных можно разделить на несколько частей:

1) Лексический и синтаксический анализы запроса, написанного на языке SQL
2) Семантический анализ дерева разбора, полученного на первом этапе
3) Генерация запросов к различным СУБД
4) Запрос необходимых данных из таблиц
5) Формирование результирующей выборки

В качестве основы для грамматики SQL брался стандарт SQL:1999 (SQL-3) [1]. В данном стандарте описаны основные конструкции языка SQL, а также данный стандарт поддерживается множеством СУБД.

На рисунке 1 представлена общая схема работы агрегатора баз данных.

3.1 Лексический анализатор

В грамматике SQL можно выделить 6 лексических доменов: целое число, число с плавающей точкой, идентификатор, ключевое слово, спецсимвол и пробельный символ.

Целое число — ненулевая последовательность цифр, которая не начинается с нуля или ноль. Пример: «123», «0».

Число с плавающей точкой — два целых числа разделенных символом «.», при этом одно из этих чисел может отсутствовать. Пример: «1.2», «.2», «12.».

Идентификатор — непустая последовательность латинских букв, цифр и символов «_», которая не начинается с числа. В случае если идентификатор совпадает с зарезервированным ключевым словом, его необходимо заключить в кавычки «`».

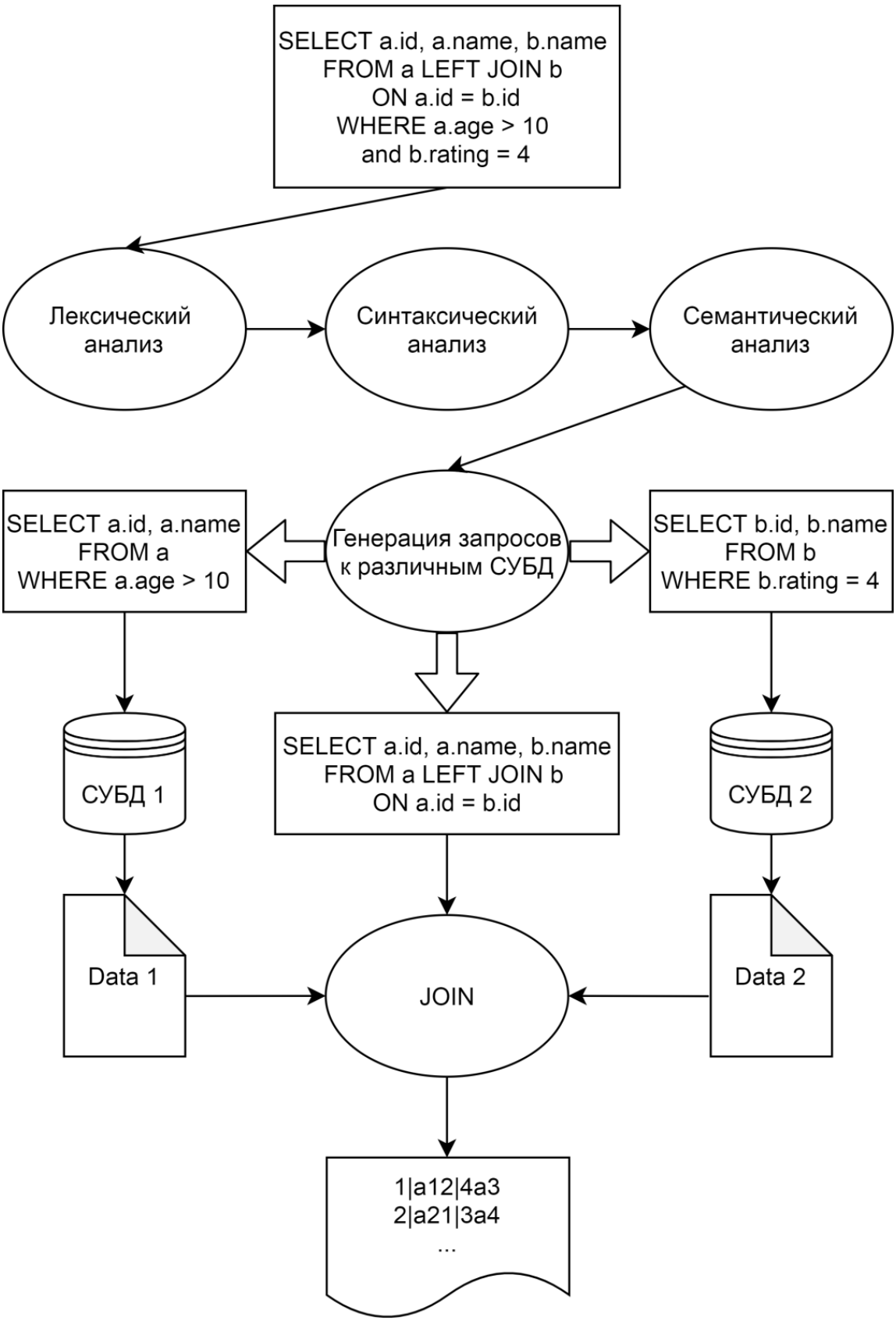

Рисунок 1 — общая схема работы агрегатора баз данных

Ключевое слово — множество зарезервированных идентификаторов. Грамматика SQL не учитывает регистр символов для ключевых слов, поэтому

«SELECT» ≡ «select» (в дальнейшем все ключевые слова будут указываться в верхнем регистре). Всего можно выделить 18 ключевых слов: «SELECT», «AS», «OR», «AND», «NOT», «IS», «TRUE», «FALSE», «NULL», «FROM», «CROSS», «JOIN», «INNER», «OUTER», «LEFT», «RIGHT», «ON», «WHERE».

Спецсимвол — последовательность различных символов. Всего можно выделить 15 спецсимволов. На листинге 1 приведены используемые спецсимволы, а также их названия, которые участвуют в грамматике языка.

Листинг 1 — используемые спецсимволы

```
<greater than or equals operator> ::= >=

<less than or equals operator> ::= <=

<not equals operator> ::= <>

<asterisk> ::= *

<comma> ::= ,

<equals operator> ::= =

<greater than operator> ::= >

<left paren> ::= (

<right paren> :: )

<less than operator> ::= <

<minus sign> ::= -

<plus sign> ::= +

<period> ::= .

<semicolon> ::= ;

<solidus> :: /
```

Пробельный символ — символ пробела либо новой строки. Данные лексемы не участвуют непосредственно в синтаксическом анализе и необходимы только для разделения различных лексем.

3.2 Синтаксический анализатор

Для работы агрегатора баз данных нет необходимости осуществлять полную поддержку грамматики SQL:1999, так как в агрегаторе баз данных будет использоваться только конструкция выборки данных — SELECT.

В приложении А представлена диалект грамматики SQL в формате РБНФ. Стоит отметить, что данная грамматика не является LL(1). Например, нетерминальный символ <value expression> переходит либо в нетерминал <numeric value expression>, либо в нетерминал <boolean value expression>, множество FIRST данных альтернатив пересекается и поэтому нельзя по одному токену однозначно выбрать нужный нетерминальный символ, в который раскрывается нетерминал <value expression>.

Данную особенность грамматики SQL необходимо учесть при реализации синтаксического анализатора, так как не все подходы для построения синтаксических анализаторов поддерживают грамматики отличные от LL(1).

Самым главным отличием используемой грамматики от грамматики SQL:1999 года является возможность использования расширенного имени для таблицы. В стандарте SQL:1999 полное имя таблицы формируется из трех частей: имя базы данных, имя схемы и имя таблицы внутри заданной схемы. Так как агрегатор баз данных позволяет работать с множеством различных СУБД, то в полном имени таблицы должен упоминаться уникальный идентификатор СУБД, поэтому в используемой грамматике полное имя таблицы формируется из четырех частей: имя СУБД, имя базы данных, имя схемы и имя таблицы внутри заданной схемы.

3.3 Тернарная логика

В СУБД вместо стандартной булевой логики чаще всего используется тернарная логика, которая является расширением булевой логики. Помимо двух значений ИСТИНА и ЛОЖЬ добавляется еще одно — НЕИЗВЕСТНО.

Рассмотрим основные операции в тернарной логике при помощи таблиц истинностей, используя следующие обозначения: А, В — свободные переменные, Л — ЛОЖЬ, Н — НЕИЗВЕСТНО, И — ИСТИНА.

Таблица истинности для операции конъюнкции:

A∧B		**A**		
		Л	**Н**	**И**
B	**Л**	Л	Л	Л
	Н	Л	Н	Н
	И	Л	Н	И

Таблица истинности для операции дизъюнкции:

A∨B		**A**		
		Л	**Н**	**И**
B	**Л**	Л	Н	И
	Н	Н	Н	И
	И	И	И	И

Таблица истинности для операции отрицания:

A	$\bar{A}$
Л	И
Н	Н
И	Л

Как видно из таблиц истинности, если не один операнд не равен НЕИЗВЕСТНО, то операции в тернарной логике эквивалентны операциям в булевой логике.

3.4 СДНФ для тернарной логики

Различные формулы в булевой логике могут быть эквивалентными, так, например, $\overline{A \wedge B} \equiv \bar{A} \vee \bar{B}$. Эквивалентность двух формул означает равенство таблиц истинностей этих формул. Для того, чтобы работать не со всем множеством логических формул, а только с логическими формулами заданного вида, существуют нормальные формы. Нормальная форма — канонический

вид, к которому приводится объект при помощи эквивалентных преобразований. Для булевой логики нормальной формой могут служить ДНФ или КНФ [2].

ДНФ — нормальная форма, где булева формула имеет вид дизъюнкции элементарных конъюнкций. Элементарная конъюнкция — конъюнкция переменных или их отрицаний.

КНФ — нормальная форма, где булева формула имеет вид конъюнкции элементарных дизъюнкций. Элементарная дизъюнкция — дизъюнкция переменных или их отрицаний.

Для каждой булевой формулы может быть несколько эквивалентных ДНФ или КНФ, поэтому среди них выделяют совершенные формы.

СДНФ — ДНФ, которая удовлетворяет двум условиям.

- В ней нет одинаковых элементарных конъюнкций
- Каждая элементарная конъюнкция содержит все переменные, причем по одному разу и в одинаковом порядке

Аналогично определяется СКНФ. Для каждой не тождественной ЛЖИ булевой формы существует СДНФ, причем единственная [2]. Аналогично для каждой не тождественной ИСТИНЕ булевой формы существует СКНФ причем единственная [2].

Совершенные нормальные формы подходят для описания логических выражений в запросах SQL, так как приводят логические формы к единому виду, с которым легко работать. В дальнейшем будет рассматриваться СДНФ, хотя аналогичным образом можно использовать СКНФ.

Если две формулы эквивалентны в булевой логике, то в тернарной логике они могут быть не эквивалентны, поэтому стандартный способ сведения формулы к СДНФ [2] не подходит для тернарной логики. Рассмотрим следующую логическую форму $F_1 = A \vee \bar{B}$ с помощью элементарных

преобразований данная форма сводится к следующей СДНФ $F_2 = (\bar{A} \wedge \bar{B}) \vee (A \wedge \bar{B}) \vee (A \wedge B)$.

Таблица истинности для формул F_1 и F_2

A	**B**	$\mathbf{F_1}$	$\mathbf{F_2}$
Л	**Л**	И	И
Л	**Н**	Н	Н
Л	**И**	Л	Л
Н	**Л**	И	Н
Н	**Н**	Н	Н
Н	**И**	Н	Н
И	**Л**	И	И
И	**Н**	И	Н
И	**И**	И	И

Из таблицы истинности видно, что данные формулы эквивалентны в булевой логике, но при этом не эквивалентны в тернарной логике. Алгоритм приведения булевой формы к СДНФ можно модифицировать так, чтобы он подходил для тернарной логики. Стоит заранее отметить, что в базах данных, если конечный результат тернарной операции НЕИЗВЕСТНО, то он расценивается как ЛОЖЬ, поэтому СДНФ будет строиться из расчета на это свойство, другими словами, если при одной и той же комбинации одна формула возвращает ЛОЖЬ, а другая НЕИЗВЕСТНО, то они все равно будут эквивалентными.

Перед тем как рассматривать модификацию алгоритма приведения формулы в СДНФ, необходимо определить новую бинарную операцию «ЯВЛЯЕТСЯ» («IS»). Данная функция принимает ИСТИНУ, если два операнда эквивалентны и ЛОЖЬ в другом случае.

Для тернарной логики также заменяется понятие элементарной конъюнкции (дизъюнкции).

Элементарная конъюнкция (дизъюнкция) в тернарной логике — конъюнкция (дизъюнкция) операций «ЯВЛЯЕТСЯ». Левый операнд операции

«ЯВЛЯЕТСЯ» — свободная переменная, а правый операнд одно из трех элементарных значений: ЛОЖЬ, НЕИЗВЕСТНО, ИСТИНА.

Таблица истинности операции «ЯВЛЯЕТСЯ»

A is B		**A**		
		Л	**Н**	**И**
B	**Л**	И	Л	Л
	Н	Л	И	Л
	И	Л	Л	И

После ввода новой операции можно описать сам алгоритм приведения любой тернарной операции к СДНФ:

1) Для тернарной логической операции выписывается таблица истинности из расчета, что каждый операнд может принимать три значения: ЛОЖЬ, НЕИЗВЕСТНО, ИСТИНА

2) Находятся все такие комбинации, где функция принимает значение ИСТИНА

3) Если таких комбинаций нет, то данную функцию нельзя привести к СДНФ и ее можно считать тождественно равной ЛЖИ (в предположении, что НЕИЗВЕСТНО в конечном результате эквивалентно ЛЖИ). Иначе переводим найденные комбинации в элементарную конъюнкцию, по следующим правилам:

- Если операнд $A = \text{ЛОЖЬ}$, то заменяем его на $A\ is$ ЛОЖЬ;
- Если операнд $A = \text{НЕИЗВЕСТНО}$, то заменяем его на $A\ is$ НЕИЗВЕСТНО;
- Если операнд $A = \text{ИСТИНА}$, то заменяем его на $A\ is$ ИСТИНА;

4) Полученные элементарные конъюнкции объединяются при помощи дизъюнкции;

Стоит отметить, что данный алгоритм продолжает работать и в булевой логике, так как в булевой логике $A\ is$ ИСТИНА $\equiv A$, а $A\ is$ ЛОЖЬ $\equiv \bar{A}$.

Рассмотрим предыдущий пример и применим к нему модифицированный алгоритм получения СДНФ. $F_1 = A \vee \bar{B}$, $F_3 = \text{СДНФ}(F_1) = (A\ is\ \text{Л} \wedge B\ is\ \text{Л}) \vee (A\ is\ \text{Н} \wedge B\ is\ \text{Л}) \vee (A\ is\ \text{И} \wedge B\ is\ \text{Л}) \vee (A\ is\ \text{И} \wedge B\ is\ \text{Н}) \vee (A\ is\ \text{И} \wedge B\ is\ \text{И})$.

Таблица истинности для формул F_1 и F_3

A	**B**	$\mathbf{F_1}$	$\mathbf{F_3}$
Л	**Л**	И	И
Л	**Н**	Н	Л
Л	**И**	Л	Л
Н	**Л**	И	И
Н	**Н**	Н	Л
Н	**И**	Н	Л
И	**Л**	И	И
И	**Н**	И	И
И	**И**	И	И

Как видно из таблицы истинности, данные формулы эквивалентны, если считать, что НЕИЗВЕСТНО эквивалентно ЛОЖИ. Доказательство данного метода приведения формулы тернарной логики в СДНФ аналогично доказательству метода приведения к СДНФ для булевой логики.

3.5 Свертка выражений

В дальнейшем логические условия будут приводиться в терминах языка SQL. ИСТИНА — TRUE, ЛОЖЬ — FALSE, НЕИЗВЕСТНО — NULL.

Часть арифметических выражений можно решить на этапе компиляции. Рассмотрим несколько базовых конструкций, которые можно свернуть (упростить), далее a обозначает колонку:

- $a + 0 \equiv a$
- $a - 0 \equiv a, 0 - a \equiv -a$
- $a * 0 \equiv 0, a * 1 \equiv a$
- $a/0 \equiv NULL, a/1 \equiv a$

- Если хотя бы один из операндов арифметического выражения $NULL$, то все выражение $NULL$
- Если оба операнда не являются колонками, то можно посчитать значение выражения, учитывая, что $TRUE \equiv 1$, $FALSE \equiv 0$

С помощью данных правил можно сворачивать большие арифметические выражения в компактные при помощи рекурсии. Для каждого выражения в начале производится свертка операндов, а потом, если обновленное выражение соответствует какому-либо шаблону свертки, производится свертка данного выражения.

Аналогичным образом можно произвести свертку логических выражений. Рассмотрим шаблоны свертки для логических выражений:

- $not\ Null \equiv Null$
- $a\ and\ FALSE \equiv FALSE$, $a\ and\ TRUE \equiv a$
- $NULL\ and\ FALSE \equiv FALSE$, $NULL\ and\ TRUE \equiv NULL$
- $a\ or\ FALSE \equiv a$, $a\ or\ TRUE \equiv TRUE$
- $NULL\ or\ FALSE \equiv NULL$, $NULL\ or\ TRUE \equiv TRUE$
- Если оба операнда не являются колонками, то можно посчитать значения выражения, учитывая, что любое число эквивалентно $TRUE$, если она не равно 0, иначе число эквивалентно $FALSE$

В логических выражениях используется иной подход к обработке $NULL$, нежели в арифметических выражениях. Если в арифметических выражениях эквивалентность хотя бы одного операнда $NULL$ означало эквивалентность всего выражения $NULL$, то в логических выражениях равенство $NULL$ одного из операндов не всегда влечет за собой равенство $NULL$ всего выражения, например $a\ and\ FALSE \equiv FALSE$. Также можно заметить отсутствие свертки $a\ and\ NULL \equiv NULL$, так как, если значение колонки $a = FALSE$, то результатом выражения будет $FALSE$, а не $NULL$. Аналогично отсутствует сверка $a\ or\ NULL \equiv a$.

3.6 Упрощение СДНФ для тернарной логики

В предыдущем пункте был рассмотрен алгоритм приведение формулы тернарной логики в модифицированную СДНФ. Полученную форму можно упросить и привести в форму, более подходящую для работы в СУБД при помощи следующих правил:

1) Если в каждой элементарной конъюнкции есть одинаковый операнд (операция «ЯВЛЯЕТСЯ» с одинаковыми операндами), то его можно вынести из элементарных конъюнкций, а саму форму поменять на дизъюнкцию данного операнда и исходной СДНФ, где в каждой элементарной конъюнкции отсутствует данный операнд. Пример: $(A\ is\ И \wedge B\ is\ Л) \vee (A\ is\ И \wedge B\ is\ И) \equiv (A\ is\ И) \wedge ((B\ is\ Л) \vee (B\ is\ И))$.

2) Предположим, что во всех элементарных конъюнкциях отсутствует операнд вида $A\ is$ Л, где A — свободная переменная, тогда во всех элементарных конъюнкциях присутствует либо $A\ is$ И, либо $A\ is$ Н. Разделим все элементарные конъюнкции на два множества, в одно попадают элементарные конъюнкции, которые содержат $A\ is$ И, а в другое, которые содержат $A\ is$ Н. Во всех элементарных конъюнкциях исключаем операнд, содержащий свободную переменную A, если при этом два множества становятся равными, то исходную форму можно переписать в виде дизъюнкции $\overline{A\ is\ Л}$ и исходной СДНФ, где в каждой элементарной конъюнкции отсутствуют операнды, содержащие свободную переменную A, и убраны все дубликаты. Аналогичные действия необходимо провести, если во всех элементарных конъюнкциях отсутствует операнд $A\ is$ Н либо $A\ is$ И. Пример: $(A\ is\ Л \wedge B\ is\ И) \vee (A\ is\ И \wedge B\ is\ И) \equiv \overline{(A\ is\ Н)} \wedge (B\ is\ И)$.

3) Предположим, что каждая из операндов $A\ is$ Л, $A\ is$ Н и $A\ is$ И, где A — свободная переменная, встречаются хотя бы в одной элементарной конъюнкции. Разделим все элементарные конъюнкции на три множества, которые содержат $A\ is$ Л, $A\ is$ Н и $A\ is$ И соответственно, а также удалим из всех элементарных конъюнкций данные операнды. Если при этом все три

множества равны между собой, то свободный член A не влияет на значение функции и соответственно во всех элементарных конъюнкциях можно исключить операнды, которые содержат данную переменную. После этого необходимо избавиться от дубликатов. Пример: $(A\ is\ Л \wedge B\ is\ Л) \vee (A\ is\ Н \wedge B\ is\ Л) \vee (A\ is\ И \wedge B\ is\ Л) \equiv (B\ is\ Л)$

3.7 Декомпозиция условия для WHERE

Для эффективного использования ресурсов компьютера необходимо запрашивать из внешних источников только те данные, которые непосредственно участвуют в построении выборки. Другими словами, нет необходимости запрашивать колонки, которые не задействуются в выводе, также нет необходимости запрашивать данные, которые в последствие будут отфильтрованы.

Как было сказано ранее, оператор WHERE задает ограничения на выходную выборку. Все логические условия, которые участвуют в операторе WHERE можно разделить на три группы:

1. Условия, не использующие информацию из таблиц (константы). Пример:

```
SELECT * FROM tbl WHERE True
```

2. Условия, использующие информацию только из одной таблицы. Пример:

```
SELECT * FROM people WHERE age>25
```

3. Условия, использующие информацию из двух и более таблиц. Пример:

```
SELECT * FROM a LEFT JOIN b ON a.id=b.id WHERE a.val>=b.val
```

Наибольший интерес в операторе WHERE представляет группа условий, использующие только одну таблицу, такие условия можно применить на этапе

запроса данных, тем самым уменьшив объем передаваемых данных. В некоторых случаях из условий, которые используют сразу несколько таблиц можно выделить вложенные условия, которые используют только одну таблицу. Рассмотрим следующий запрос:

```
SELECT * FROM a, b WHERE a.val=b.val and a.id=20
```

В чистом виде условие в операторе WHERE использует две таблицы и соответственно невозможно вынести данное условие на этап запроса данных из базы данных. В этом условии можно выделить два вложенных условия: $a.val = b.val$ и $a.id = 20$, которые связаны логической конъюнкцией. Первое вложенное условие использует две таблицы, а второе использует уже одну таблицу, так как два этих вложенных условия связаны конъюнкцией, то истинность всего условия возможна только тогда, когда два операнда истинны, поэтому второе вложенное условие можно вынести на этап запроса данных при этом первое условие станет исходным условием.

В предыдущих параграфах рассматривалась СДНФ для тернарной логики, а также упрощение полученной СДНФ. Так как в СУБД используется тернарная логика, то любые логические условия в операторе WHERE можно представлять в виде СДНФ. Использование колонок таблиц в качестве свободных переменных в логической формуле неэффективно, так как в одном логическом условии может использоваться большое число колонок, которые повлекут за собой большой размер нормальной формы, вычислять которую станет в разы труднее. Поэтому в качестве свободной переменной целесообразно использовать вложенное условие, которое использует только одну таблицу, если условие невозможно «неделимо», то есть его нельзя разделить логически, например, «неделимым» условием считается любая операция сравнения, либо арифметическая операция, и при этом использует более одной таблицы, то такое условие также считается свободной переменной. В качестве примера рассмотрим следующее логическое условие:

```
(a.id=b.id or (a.id=3 or (a.id=1 and a.val=2))) and b.id=1,
```

где a и b — различные таблицы. Для простоты восприятия представим данное выражение в виде дерева разбора (рисунок 2):

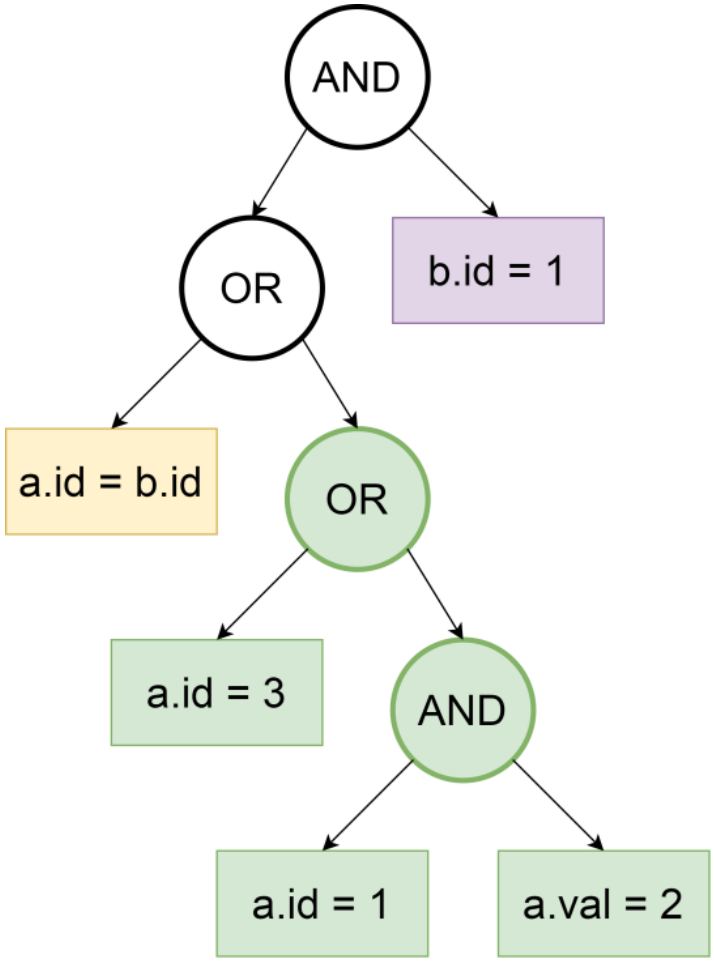

Рисунок 2 — дерево разбора логического выражения

В данном логическом выражении можно выделить три вложенных выражения, которые будут являться свободными переменными при построении СДНФ:

1) A ::= a.id = b.id
2) B ::= a.id = 3 or (a.id = 1 and a.val = 2)
3) C ::= b.id = 1

После замены вложенных условий на свободные переменные, логическое уравнение принимает вид:

```
(A or B) and C.
```

После приведения к СДНФ и упрощения формула принимает вид:

```
   C is True  and (
   A is True  and B is False
or A is True  and B is Null
or A is False and B is True
or A is None  and B is True
or A is True  and B is True).
```

В данной формуле $C\ is\ True\ \equiv b.id = 1\ is\ True$ можно вынести на этап запроса данных из таблицы, тем самым сократив объем запрашиваемых данных. Стоит отметить, что при проверке ограничений свободные переменные A и B вычисляются только один раз для каждой строки и полученные значения подставляются в упрощенную формулу.

3.8 Декомпозиция условия для JOIN

Все действия, описанные для декомпозиции условия WHERE, можно также распространить и на условия для INNER JOIN, так как любой INNER JOIN можно представить в виде CROSS JOIN с дополнительно наложенным условием WHERE.

Для LEFT (RIGHT) JOIN необходимо модифицировать метод декомпозиции условия, так как результирующая выборка может включать в себя строки из левой (правой) таблицы, для которых не найдено соответствие из правой (левой) таблицы согласно заданному условию. Рассмотрим следующий запрос:

```
SELECT * FROM a LEFT JOIN b ON a.id=b.id and a.id=10
```

Из условия данного JOIN можно выделить вложенное условие $a.id = 10$, которое использует только одну таблицу. Если вынести это вложенное условие на этап запроса данных из таблицы, то результирующая выборка стала бы

неполной, так как в ней бы отсутствовали записи, где $a.id = 10$. Поэтому при LEFT (RUGHT) JOIN вложенные условия, которые используют левую (правую) таблицу нельзя выносить на этап запроса данных.

С другой стороны, если из условия LEFT (RIGHT) JOIN можно выделить вложенное условие, которое использует данные только из правой (левой) таблицы, то это вложенное условие можно вынести на этап запроса данных из таблицы при этом результирующая выборка будет полной.

4 РЕАЛИЗАЦИЯ

Реализация агрегатора баз данных производилась на языке Python 3.7. Это позволило сделать данное приложение кроссплатформенным.

4.1 Реализация лексического анализатора

Основные лексические домены грамматики SQL можно описать при помощи регулярных выражений. На листинге 2 представлены регулярные выражения для домена идентификаторов (IDENTIFIER), домена ключевых слов (KEYWORD), домена целых чисел (INTEGER) и домена чисел с плавающей точкой (FLOAT).

Листинг 2 — регулярные выражения для лексических доменов

```
<IDENTIFIER> ::=
    r'[a-zA-Z_][a-zA-Z_0-9]*|`[a-zA-Z_][a-zA-Z_0-9]*`'

<KEYWORD> ::= r'[a-zA-Z_][a-zA-Z_0-9]*'

<INTEGER> ::= r'(?:[1-9]\d*|0)(?![0-9A-Za-z_])'

<FLOAT> ::=
    r'(?:([1-9]\d*|0)?\.\d+|([1-9]\d*|0)\.)(?![0-9A-Za-z_])'
```

Лексема числа (INTEGER или FLOAT) должна быть разделена от лексемы идентификатора пробельным символом, другими словами в исходном тексте программы не допустима следующая подстрока: «123abc», поэтому в регулярных выражениях для чисел производится проверка символа следующего за разобранной подстроки.

Принадлежность к домену спецсимволов определяется при помощи регулярных выражений, которые объявлены в листинге 1.

Для домена ключевых слов производится дополнительная проверка, для того, чтобы отсеять идентификаторы, не являющиеся ключевыми словами, для этого найденное слово приводится к верхнему регистру и проверяется его принадлежность множеству ключевых слов.

В грамматика SQL стандарта 1999 года ключевые слова разделены на два множества: множество зарезервированных ключевых слов и множество незарезервированных ключевых слов. Домен зарезервированных ключевых слов имеет более высокий приоритет, чем домен идентификаторов, поэтому не возникает конфликтов при выборе домена для фрагмента текста программы. В отличие от домена зарезервированных слов домен незарезервированных ключевых слов имеет такой же приоритет, что и домен идентификаторов, поэтому могут возникать ситуации, когда выбор домена для фрагмента теста программы невозможен. В таких ситуациях лексический анализатор возвращает сразу множество всевозможных лексем с максимальной длиной. На данный момент все ключевые слова являются зарезервированными, и данная функциональность необходима для будущего расширения поддерживаемых конструкций языка SQL.

Для того чтобы сократить затраты ресурсов памяти компьютера, фаза лексического анализа совмещена с фазой синтаксического анализа. Данный способ позволяет не только сократить затраты на ресурсы памяти, но также позволяет выводить лексические и синтаксические ошибки в правильном порядке. Для обеспечения псевдопараллельного выполнения фаз лексического и синтаксического анализа, лексический анализатор должен обладать специальным методом, который при вызове будет разбирать следующую лексему в тексте, и возвращать ее. Тем самым в каждый момент работы программы в памяти храниться информация только об одной лексеме, что позволяет сократить затраты памяти. При вызове данного метода первым делом осуществляется пропуск пробельных символов, начиная с позиции, на которой остановился лексический анализатор, после пропуска пробельных символов для каждого лексического домена осуществляется поиск подстроки при помощи заданного регулярного выражения. Если все регулярные выражения не смогли выделить подстроку, то лексический анализатор начинает пропускать символы, до тех пор, пока хотя бы для одного лексического домена не будет найдена подстрока. Если хотя бы для одного лексического домена была разобрана

подстрока, то из всех лексических доменов выбираются те, у которых длина подстроки и приоритет максимальны, для данных лексических доменов формируются лексемы. Возможность продолжать разбор, даже если в тексе программы встретилась подстрока, которая не подходит ни одному лексическому домену, позволяет вывести все ошибки в тесте, а не только первую.

4.2 Реализация синтаксического анализатора

Для реализации синтаксического анализатора использовался алгоритм рекурсивного спуска. Алгоритм рекурсивного спуска обладает простотой реализации, а также синтаксические анализаторы, написанные при помощи рекурсивного спуска, легко расширять путем добавления новых синтаксических конструкций.

Особенностью алгоритма рекурсивного спуска является то, что он поддерживает только LL(1) грамматики, а грамматика SQL используемая в агрегаторе баз данных не является LL(1) грамматикой. Данную проблему можно решить двумя способами: переписать грамматику, чтобы она стала LL(1) грамматикой, либо модифицировать алгоритм рекурсивного спуска, чтобы он смог принимать на вход не только LL(1) грамматики. После приведения в LL(1) форму дерево разбора может стать слишком неудобным для дальнейшего семантического анализа, также дальнейшее расширение грамматики может стать затруднительным, поэтому лучше всего модифицировать алгоритм рекурсивного спуска.

Для того чтобы алгоритм рекурсивного спуска поддерживал не только LL(1) грамматики необходимо реализовать в нем возможность отката, то есть возможность возврата в предыдущее состояние. Общий принцип заключается в проверке всевозможных альтернатив и выборе той альтернативы, для которой разбор оказался успешным и длина разобранной части максимальна. Рассмотрим грамматику, представленную на листинге 3.

Листинг 3 — пример не LL(1) грамматики

```
<A> ::= <B> | <C> | <D>

<B> ::= <INTEGER> + <INTEGER>

<C> ::= <INTEGER> * <INTEGER>

<D> ::= <INTEGER>

<INTEGER> ::= 0 | 1
```

Данная грамматика не является LL(1), так как множества FIRST для нетерминальных символов B, C и D пересекаются и из-за этого по одной лексеме невозможно выбрать нужную альтернативу в нетерминальном символе A. Для примера рассмотрим разбор следующего текста «1 + 0» при помощи синтаксического анализатора данной грамматики, основанной на алгоритме рекурсивного спуска с возможностью откатов. На первом шаге проверяем нетерминальный символ A, для данного нетерминального символа разбор оказался успешным и был разобран текст «1 + 0». На втором шаге возвращаем позицию лексического анализатора на начало и проверяем нетерминальный символ B, для данного нетерминального символа разбор оказался неудачным. На третьем шаге также возвращаем позицию лексического анализатор на начало текста и проверяем нетерминальный символ D, для данного нетерминального символа разбор оказался удачным и бал разобран текст «1». На четвертом шаге производится выбор той альтернативы, для которой длина разобранной части максимальна, в данном случае, это нетерминальный символ A. Если для всех альтернатив разбор оказался неудачным, то синтаксический анализ завершается с ошибкой. Недостатком такого подхода является сложность обработки ошибок, а также восстановления при них. Но при этом данный подход никак не влияет на исходную грамматику языка, а также обладает простой программной реализацией.

В ходе синтаксического анализа вместо построения дерева разбора производится заполнение структуры, которая в дальнейшем проверяется при

помощи семантического анализа. Данный подход позволяет сократить объем кода, так как нет необходимости дополнительно проходиться по дереву разбора.

4.3 Реализация семантического анализатора

Стадия семантического анализа начинается только, если не было ошибок на стадии лексического и синтаксического анализа, иначе производится выход из программы с ошибкой.

На стадии семантического анализа производится проверка всех введенных значений, а также заполнение новой информации необходимой для дальнейшей работы. Семантический анализ можно разделить на 4 части:

1. Проверка табличного выражения. На данном этапе производится проверка существования таблицы, а также запрашивается информация о существующих колонках в данной таблице
2. Проверка условий для всех JOIN. На этом этапе производится проверка существования колонок, используемых логических условиях, а также перевод логической формулы в СДНФ с последующим упрощением. Если из условия можно вынести вложенное условие на этап запроса данных, то данное вложенное условие приписывается к используемой таблице.
3. Проверка списка запрашиваемых колонок, как существующих, так и генерируемых на основе существующих. На этом этапе производится проверка существования колонок, используемых для формирования конечной выборки.
4. Проверка логического условия WHERE. На этом этапе так же, как и на этапе проверки условий для JOIN, производится проверка на существование колонок, а также перевод логической формулы в СДНФ с последующим упрощением. Если из условия можно вынести вложенное условие на этап запроса данных, то данное вложенное условие приписывается к используемой таблице.

Важно соблюдать заданный порядок проверки, так как на некоторых этапах используется информация, полученная на предыдущем этапе.

Для того чтобы запрашивать из таблицы только используемые колонки, на этапе проверки производится подсчет количества использования для каждой колонки. Если условие, использующее какую-либо колонку, переносится на этап запроса данных, то счетчик данной колонки уменьшается на единицу. Таким образом, для каждой колонки мы получаем количество использований, если количество больше нуля, то данная колонка необходима для формирования выборки, иначе данная колонка не используется в выборке.

4.4 Запрос данных из внешних источников

После семантического анализа в случае отсутствия ошибок производится запрос данных из внешних источников.

Для подключения к различным внешним источникам, в роли которых выступают РСУБД, используется ODBC. ODBC позволяет использовать единый интерфейс для подключения к множеству различных РСУБД, поэтому при запросе данных можно не учитывать особенности архитектур различных СУБД.

Для подключения к той или иной базе данных нужно иметь информацию о ней: адрес подключения, используемый драйвер, логин и пароль пользователя. Для удобства пользователя, чтобы он не вводил данную информацию при каждом использовании, информация записывается в конфигурационный файл в формате YAML [3] и в программу передается только путь к данному файлу. На листинге 4 приведен пример конфигурационного файла.

Листинг 4 — пример конфигурационного файла

```
dbms:
  driver: '{PostgreSQL Unicode}'
  server: 127.0.0.1
  port: 5432
  uid: user
  pwd: password
  type: psql
```

В данном примере *dbms* — уникальное имя СУБД, *driver* — название драйвера, *server* — адрес базы данных, *port* — используемый порт, *uid* — имя пользователя, *pwd* — пароль пользователя, *type* — тип СУБД. В рамках данной работы была осуществлена поддержка двух СУБД: psql — PostgreSQL [4] и mysql — MySQL [5].

Реализация поддержка нового типа СУБД происходит при помощи создания нового класса, где будут реализованы методы для получения информации о существующих колонках, а также метод для преобразования типов.

4.5 Формирование результирующей выборки

После получения из внешних источников, данные переносятся во временную базу данных SQLite [6]. По умолчанию база данных создается в оперативной памяти. Использование оперативной памяти позволяет сократить время, которое тратится на запись в файл и чтение из файла. Большинство данных из внешних источников фильтруются на этапе запроса данных, поэтому использование оперативной памяти достаточно для большинства задач.

Перед переносом данных, в базе данных SQLite создаются таблицы согласно структурам данных из внешних источников. Так как различные СУБД могут обладать данными различных типов, то в агрегатор баз данных было внесено ограничение на поддерживаемые типы данных. Поддерживаются следующие типы и/или эквивалентные им: число с плавающей точкой (float), логический тип (boolean), целое число (int), строка (string). Для каждой

поддерживаемой СУБД было задано отображение этих типов в типы SQLite. После этого в данные таблицы заносится информация из внешних источников.

Для формирования результирующей выборки в базе данных SQLite определялось представление (VIEW). Использование представлений позволяет не сохранять результирующую выборку в оперативной памяти, так как ее объем может быть в разы больше чем данные, которые используются при ее построении.

4.6 Установка

Для корректной работы агрегатора баз данных необходим предустановленный интерпретатор языка Python 3.7 (или выше). Возможны два варианта установки:

1. Используя систему управления пакетами pip (Python Package Installer) и систему контроля версий GIT. Для установки агрегатора баз данных необходимо выполнить следующую команду (листинг 5) в командной строке или терминале.

Листинг 5 — установка приложения при помощи системы управления пакетами

```
pip install git+https://github.com/bmstu-iu9/sql-aggregator.git
```

2. Сборка из источников. Для этого необходимо скачать нужную версию на компьютер, а затем произвести установку с помощью Python. На листинге 6 представлены команды, с помощью которых можно установить агрегатор баз данных из исходников.

Во время установки автоматически установятся следующие библиотеки Python: pyodbc [7], PyPika [8], PyQt5 [9], PyYAML [10].

Листинг 6 — установка библиотеки из исходников

```
  curl -LJO https://github.com/bmstu-iu9/sql-aggregator/archive/
master.zip
  unzip sql-aggregator-master.zip
  cd sql-aggregator-master
  python setup.py install
```

4.7 Пользовательский интерфейс

Для удобства пользования в агрегатор баз данных был встроен графический интерфейс пользователя. Графические интерфейс построен с использованием библиотеки PyQt5, поэтому он может быть использован в большинстве современных операционных систем. Для запуска графического пользовательского интерфейса необходимо воспользоваться следующей командой (листинг 7).

Листинг 7 — запуск графического пользовательского интерфейса

```
  python –m multidb.qt
```

Перед запуском программы необходимо убедится, что конфигурационный файл находится в директории, откуда производился запуск пользовательского интерфейса, либо в операционной системе была определена переменная среды MULTIDB_CONFIG, в которой записан путь к конфигурационному файлу.

После выполнения команды открывается окно с пользовательским интерфейсом. На рисунке 3 представлен снимок экрана с запущенным пользовательским интерфейсом в операционной среде Windows 7.

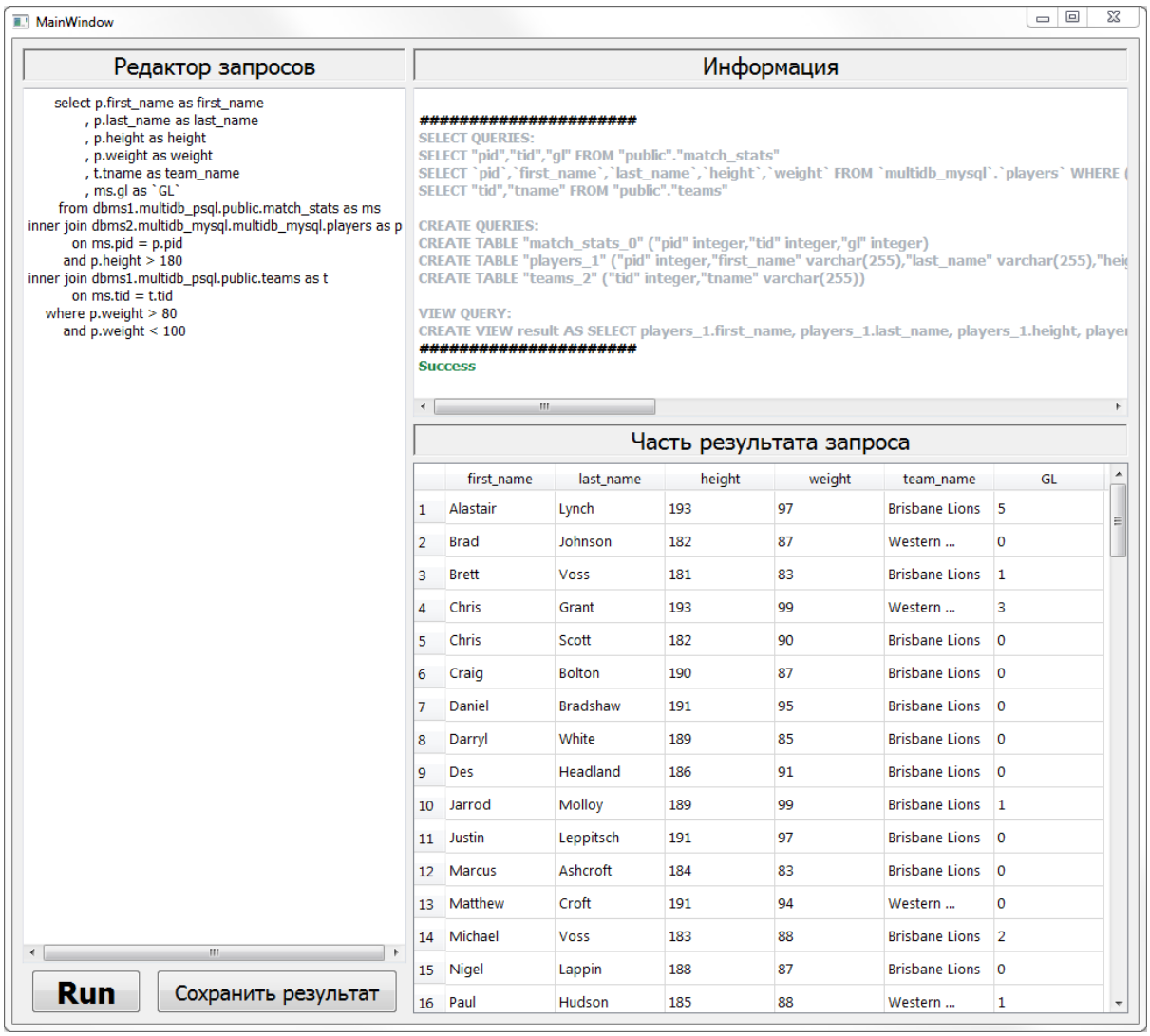

Рисунок 3 — графический пользовательский интерфейс

В левой части окна находится редактор запросов, в данный редактор вводится требуемый запрос на языке SQL. Правая часть окна разделена на две области. В верхнем правом углу выводится отладочная информация: ошибки во время выполнения, запросы для получения данных из внешних источников, запрос для создания представления, а также сообщения об успехе операции. В нижнем правом углу отображается часть результирующей выборки (первые 100 строк). В нижнем левом углу находится две управляющие кнопки. Кнопка «Run» запускает выполнение запроса. Кнопка «Сохранить результат» позволяет сохранить результат в файле базы данных SQLite, после нажатия на данную кнопку открывается окно выбора файла. В сохраненной базе данных будут находиться выборки данных из различных внешних источников, а также

представление с именем «result», которое позволяет получить результирующую выборку.

Агрегатор баз данных можно использовать в качестве библиотеки Python. На листинге 8 представлен пример использования агрегатора баз данных в Python.

Листинг 8 — использование агрегатора баз данных в Python.

```
import os

from multidb.main import ControlCenter

QUERY = '''
    select p.first_name as first_name
         , p.last_name as last_name
         , p.height as height
         , p.weight as weight
         , t.tname as team_name
         , ms.gl as `GL`
      from dbms1.multidb_psql.public.match_stats as ms
inner join dbms2.multidb_mysql.multidb_mysql.players as p
        on ms.pid = p.pid
       and p.height > 180
inner join dbms1.multidb_psql.public.teams as t
        on ms.tid = t.tid
     where p.weight > 80
       and p.weight < 100
'''


def main():
    # Инициализация агрегатора баз данных
    cc = ControlCenter('/path/to/config.yaml')
    # Выполнение запроса
    err, _ = cc.execute(QUERY)
    # Проверка наличия ошибок выполнения
    if err:
        print('Ошибка:', err, sep='\n')
        return
    # Вывод результата
    with cc._sqlite_conn.cursor() as cur:
        cur.execute('select * from result;')
        data = cur.fetchall()
        for row in data:
            print(row)
    # Сохранение результата в файл
    if os.path.isfile('out.db'):
        os.remove('out.db')
```

```
    cc.save_result('out.db')


if __name__ == '__main__':
    main()
```

Класс отвечающий за выполнения запроса находится в модуле *multidb.main* и называется *ControlCenter*. Конструктор класса принимает на вход путь к конфигурационному файлу. Для выполнения запроса необходимо передать строку запроса в метод *execute* объекта класса *ControlCenter*. Данный метод возвращает ошибки во время выполнения и отладочную информацию. Чтобы посмотреть результирующую выборку необходимо выполнить запрос «select * from result» во временной базе данных SQLite. Для сохранения результата необходимо вызвать метод *save_result* объекта класса ControlCenter, передав в него путь к файлу, если такой файл уже существует, то вернется ошибка.

5 ТЕСТИРОВАНИЕ

Для того, чтобы убедится, что агрегатор баз данных запрашивает из внешних источников только необходимые данные, были рассмотрены запросы, включающие в себя различные логические условия. Во всех тестах рассматривалось соединение двух таблиц при помощи INNER JOIN. На листингах 9, 10, 11, 12 представлены исходные запросы SQL, а также запросы к внешним источникам.

Листинг 9 — тестирования элементарной конъюнкции

```
-- Исходный запрос
    select a.val, b.val
      from dbms.db.public.a as a
inner join dbms.db.public.b as b
        on a.id = b.id
       and a.val > 1
       and b.val > 2

-- Запросы к внешним источникам
select a.id, a.val
  from public.a
 where (a.val > 1) is TRUE

select b.id, b.val
  from public.b
 where (b.val > 2) is TRUE
```

Листинг 10 — тестирование отрицания элементарной дизъюнкции

```
-- Исходный запрос
    select a.val, b.val
      from dbms.db.public.a as a
inner join dbms.db.public.b as b
        on not (a.id <> b.id
        or a.val <= 1
```

```
        or b.val <= 2)

-- Запросы к внешним источникам
select a.id, a.val
  from public.a
 where (a.val <= 1) is FALSE

select b.id, b.val
  from public.b
 where (b.val <= 2) is FALSE
```

Листинг 11 — тестирование вложенных конструкций

```
-- Исходный запрос
    select a.val, b.val
      from dbms.db.public.a as a
inner join dbms.db.public.b as b
        on a.id = b.id
       and (a.val > 10 or a.val / 2 < 10)

-- Запросы к внешним источникам
select a.id, a.val
  from public.a
 where (
       a.val > 10
    or a.val / 2 < 10
     ) is TRUE

select b.id, b.val
  from public.b
```

Листинг 12 — тестирования сложного запроса с использованием различных логических конструкций

```
-- Исходный запрос
    select a.val, b.val
      from dbms.db.public.a as a
```

```
inner join dbms.db.public.b as b
        on not (
           not (a.id = b.id and b.val > 1)
        or (a.val <= 10 or b.val = 2)
        )

-- Запросы к внешним источникам
select a.id, a.val
  from public.a
 where (a.val <= 10) is FALSE

select b.id, b.val
  from public.b
 where (b.val > 1) is TRUE
   and (b.val = 2) is FALSE
```

Из результатов тестирования видно, что независимо от формы логического условия, она корректно приводится в СНДФ, что позволяет запрашивать из источников только необходимые данные.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были выполнены все поставленные задачи. А именно, был разработан и реализован инструмент, позволяющий осуществлять запросы к различным внешним источникам, в роли которых выступают различные РСУБД. Кроме того, была осуществлена поддержка логических и арифметических выражений, различных видов соединения в табличном выражении, а также был реализован графический пользовательский интерфейс для работы с инструментом. В рамках данной работы была осуществлена поддержка двух типов внешних источников: РСУБД PostgreSQL и РСУБД MySQL.

Помимо этого, был разработан и реализован алгоритм, способный приводить формулы тернарной логики в СДНФ. Использование данного алгоритма позволяет приводить логические условия в единую форму, которую легко декомпозировать. Тестирование реализованного алгоритма подтвердило корректность его работы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Международный стандарт ISO/IEC 9075-2:1999 (E). Информационные технологии. Языки базы данных. Язык структурированных запросов (SQL). Часть 2. Основа.
2. Новиков Ф. А. Дискретная математика для программистов. — М.: Питер, 2009. — 384 с.
3. YAML [Электронный ресурс] — URL: https://yaml.org/ (дата обращения: 01.06.2020)
4. PostgreSQL [Электронный ресурс] — URL: https://www.postgresql.org/ (дата обращения: 01.06.2020)
5. MySQL [Электронный ресурс] — URL: https://www.mysql.com/ (дата обращения: 01.06.2020)
6. SQLite [Электронный ресурс] — URL: https://www.sqlite.org/ (дата обращения: 01.06.2020)
7. PyODBC [Электронный ресурс] — URL: http://mkleehammer.github.io/pyodbc/ (дата обращения: 01.06.2020)
8. PyPika [Электронный ресурс] — URL: https://pypika.readthedocs.io/ (дата обращения: 01.06.2020)
9. PyQt5 [Электронный ресурс] — URL: https://www.riverbankcomputing.com/software/pyqt/ (дата обращения: 01.06.2020)
10. PyYAML [Электронный ресурс] — URL: https://pyyaml.org/ (дата обращения: 01.06.2020)

ПРИЛОЖЕНИЕ А

```
<IDENTIFIER> ::=
    r'[a-zA-Z_][a-zA-Z_0-9]*|`[a-zA-Z_][a-zA-Z_0-9]*`'

<INTEGER> ::= r'(?:[1-9]\d*|0)(?![0-9A-Za-z_])'

<FLOAT> ::=
    r'(?:([1-9]\d*|0)?\.\d+|([1-9]\d*|0)\.)(?![0-9A-Za-z_])'

<greater than or equals operator> ::= >=

<less than or equals operator> ::= <=

<not equals operator> ::= <>

<asterisk> ::= *

<comma> ::= ,

<equals operator> ::= =

<greater than operator> ::= >

<left paren> ::= (

<right paren> :: )

<less than operator> ::= <

<minus sign> ::= -

<plus sign> ::= +

<period> ::= .

<semicolon> ::= ;

<solidus> :: /

<program> ::= <select>

<select> ::=
    SELECT <select list> <table expression> [ <semicolon> ]

<select list> ::=
      <asterisk>
    | <select sublist> [ { <comma> <select sublist> }... ]

<select sublist> ::= <qualified asterisk> | <derived column>
```

```
<qualified asterisk> ::=
    <asterisked identifier chain> <asterisk>

<asterisked identifier chain> ::=
    <IDENTIFIER> <period> [ { <IDENTIFIER> <period> }... ]

<derived column> ::= <value expression> [ [ AS ] <IDENTIFIER> ]

<value expression> ::=
      <numeric value expression>
    | <boolean value expression>

<numeric value expression> ::=
      <term>
    | <term> <plus sign> <numeric value expression>
    | <term> <minus sign> <numeric value expression>

<term> ::=
      <factor>
    | <factor> <asterisk> <term>
    | <factor> <solidus> <term>

<factor> ::= [ <sign> ] <numeric primary>

<sign> ::= <minus sign> | <plus sign>

<numeric primary> ::= <value expression primary>

<boolean value expression> ::=
      <boolean term>
    | <<boolean term> OR <boolean value expression>

<boolean term> ::=
      <boolean factor>
    | <boolean factor> AND <boolean term>

<boolean factor> ::= [ NOT ] <boolean test>

<boolean test> ::=
    <boolean primary> [ IS [ NOT ] <truth value> ]

<truth value> ::=
      TRUE
    | FALSE
    | NULL

<boolean primary> ::=
      <predicate>
    | <value expression primary>

<predicate> ::= <comparison predicate>
```

```
<comparison predicate> ::=
    <operand comparison> <comp op> <operand comparison>

<operand comparison> ::=
      <not boolean value expression>
    | <parenthesized value expression>

<not boolean value expression> ::= <numeric value expression>

<comp op> ::=
      <equals operator>
    | <not equals operator>
    | <less than operator>
    | <greater than operator>
    | <less than or equals operator>
    | <greater than or equals operator>

<value expression primary> ::=
      <parenthesized value expression>
    | <nonparenthesized value expression primary>

<parenthesized value expression> ::=
    <left paren> <value expression> <right paren>

<nonparenthesized value expression primary> ::=
      <unsigned value specification>
    | <column reference>

<unsigned value specification> ::= <unsigned literal>

<unsigned literal> ::=
      <unsigned numeric literal>
    | <general literal

<unsigned numeric literal> ::= <INTEGER> | <FLOAT>

<general literal> ::= TRUE | FALSE | NULL

<column reference> ::= <basic identifier chain>

<basic identifier chain> ::= <identifier chain>

<identifier chain> ::=
    <IDENTIFIER> [ { <period> <IDENTIFIER> }... ]

<table expression> ::= <from clause> [ <where clause> ]

<from clause> ::= FROM <table reference list>

<table reference list> ::=
    <table reference> [ { <comma> <table reference> }... ]

<table reference> ::= <join factor> [ { <join type> }... ]
```

```
<join factor> ::=
      <table primary>
    | <left paren> <table reference> <right paren>

<table primary> ::=
    <table or query name> [ [ AS ] <IDENTIFIER> ]

<table or query name> ::= <table name>

<table name> ::= <basic identifier chain>

<joined table> ::=
      <cross join>
    | <qualified join>

<cross join> ::= CROSS JOIN <join factor>

<qualified join> ::=
    [ <join type> ] JOIN <join factor> <join specification>

<join type> ::=
      INNER
    | <outer join type> [ OUTER ]

<outer join type> ::= LEFT | RIGHT

<join specification> ::= <join condition>

<join condition> ::= ON <search condition>

<search condition> ::= <boolean value expression>

<where clause> ::= WHERE <search condition>
```