Минобрнауки России
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий

Работа допущена к защите
И.О. заведующего кафедрой КИТ
__________________ А.В. Щукин
«_____» _______________2019 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

СОЗДАНИЕ СИСТЕМЫ ТРАНСЛЯЦИИ ЕСТЕСТВЕННОГО ЯЗЫКА В ЗАПРОСЫ НА ЯЗЫКЕ SQL

по направлению 02.04.03 Математическое обеспечение и администрирование информационных систем
по образовательной программе
02.04.03_01 Математическое обеспечение и администрирование корпоративных информационных систем

Выполнил
студент гр.23546/1 Н.В. Горбатов

Руководитель
доцент каф. КИТ, к.т.н. О.Ю. Сабинин

Консультант
по нормоконтролю О.В. Колосова

Санкт-Петербург

2019

САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО
ИНСТИТУТ КОМПЬЮТЕРНЫХ НАУК И ТЕХНОЛОГИЙ

УТВЕРЖДАЮ

И.О. заведующего кафедрой КИТ

А.В. Щукин

«   »   20   г.

ЗАДАНИЕ
по выполнению выпускной квалификационной работы

студенту Горбатову Никите Владимировичу, гр. 23546/1

1. Тема работы: Создание системы трансляции естественного языка в запросы на языке SQL.
2. Срок сдачи студентом законченной работы: 22.05.2019.
3. Исходные данные по работе: алгоритмы трансляции естественного языка, принципы перевода на формальные языки.
4. Содержание работы (перечень подлежащих разработке вопросов):

- введение и постановка задачи;
- изучение принципов и алгоритмов трансляции естественного языка;
- проектирование алгоритма трансляции предложения естественного языка в язык SQL;
- реализация транслятора на основании разработанного алгоритма;
- анализ эффективности реализованного алгоритма;
- выводы и перспективы.

5. Перечень графического материала (с указанием обязательных чертежей): отсутствует.
6. Консультанты по работе профессор каф. КИТ, д.т.н. О.В. Колосова.
7. Дата выдачи задания 11.02.2019.

Руководитель ВКР ______________________________ / Сабинин О.Ю. /
(подпись)

Задание принял к исполнению 11.02.2019.
(дата)

Студент ______________________________ / Горбатов Н.В. /
(подпись)

РЕФЕРАТ

49 с., 20 рисунков, 1 таблица, 1 приложение.

МАШИННАЯ ФОРМАЛИЗАЦИЯ ТЕКСТА, БАЗЫ ДАННЫХ

В данной работе рассматривается проблема трансляции естественного текста в язык SQL, а так же предлагается решение, путем создания и реализации алгоритма формализации и трансляции предложений естественного языка. Полученный алгоритм исследуется на предмет работоспособности, а так же производится его анализ по заранее определенным критериям.

THE ABSTRACT

49p., 20 figures, 1 table, 1 appendix.

MACHINE TEXT FORMALIZATION, DATABASES

This paper discusses the problem of translating natural text into SQL queries, also a solution is proposed by creating and implementing an algorithm for formalizing and translating natural language sentences. The resulting algorithm is examined for efficiency, as well as analysis of this algorithm is performed on predetermined criteria.

СОДЕРЖАНИЕ

Введение.. 5
Глава 1. Аналитический обзор методов.. 8
1.1. Особенности неформализованных языков 8
1.2. Базовые принципы перевода текста с естественного языка......... 9
1.3. Особенности перевода в SQL 12
1.3.1. Apache OpenNLP 13
1.3.2. GATE 14
1.3.3. ETAP-3 14
1.4.Оптимизация процесса формализации 14
Глава 2. Общее описание алгоритма.. 17
2.1. Процесс формализации 18
2.2. Сценарий взаимодейтсвия модулей разбора.............................. 21
Глава 3. Реализация транслятора... 24
3.1. Основные инструменты разбора текста...................................... 25
3.2. Инструменты обработки запросов выборки 28
3.3. Прочие элементы взаимодействия с запросами 30
3.4. Модуль шаблонизации запросов... 34
Глава 4. Анализ эффективности реализованного алгоритма 35
4.1. Базовые виды запросов.. 36
4.2. Усложненные версии запросов.. 38
4.3. Сравнение эффективности реализованных функций.................. 42
Заключение 45
Список сокращение и условных обозначений 46
Список терминов.. 47
Список использованных источников 48
Приложение 50

ВВЕДЕНИЕ

С ростом и развитием технологий, во многих сферах нашей жизни нам всё чаще приходится сталкиваться с различными информационными системами для взаимодействия, с которыми требуются определённые навыки и знания. Такие технологии проникают всё в большее количество сфер, и как следствие, возникает необходимость в повышении квалификации некоторых сотрудников, или же найма дополнительных, обладающих нужными навыками. В связи с этим возникает вопрос, можно ли найти менее затратный и автоматизированный способ решения данной проблемы?

Одна из самых очевидных идей – разработка «интерфейса» или прослойки, которая позволит людям без нужных навыков взаимодействовать, к примеру, с системами управления базами данных, (далее -СУБД). Но возникают две проблемы. Первая - чем больший функционал включает в себя промежуточный интерфейс, тем чаще всего он менее интуитивно понятен, и тем сложнее с ним работать не знакомому с ним пользователю. Вторая - для каждого функционального элемента потребуется разрабатывать или модифицировать интерфейс.

И возникает вопрос, нет ли более удобного способа? Самым лёгким способом взаимодействия с любым не знакомым сервисом для любого пользователя, вне зависимости от его навыков и образования, является взаимодействие через привычный пользователю язык, со знакомыми ему грамматическими конструкциями. Есть несколько способов достичь данного взаимодействия, либо обучить машину полностью понимать естественный язык, либо использовать прослойку, которая могла бы переводить с одного языка на другой, сохраняя смысл и переформатируя конструкцию в пригодный для конечного языка, с точки зрения грамматики, вид. Так как, преобразовывать машинную логику в естественный язык существенно сложнее, чем проводить обратное преобразование (в связи с более широким спектром эмоциональной окраски и большого количества синонимов в естественном языке).

И именно данную функцию берёт на себя машинный формальный перевод. Он берёт свои корни из обычного машинного перевода, который впервые упоминался ещё в 1627г. С того момента прошло много времени, и с развитием вычислительной техники, метод многие годы эволюционировал. Больше всего нас интересует первое появление интеллектуальной обработки текстов, которое берёт своё начало в 60х годах 20го века, что в свою очередь, с увеличением вычислительной мощности в последующие года, привело к нескольким базовым способам интерпретации естественного языка. Это такие способы как синтаксический и семантический анализ шаблонов. Синтаксический строится на основе разбора фразы с учётом частей предложений, семантический – использует информацию из предыдущего метода и дополняет его информацией из тезаурусов.

На основе вышеперечисленных способов были построены различные формализаторы естественного языка различной степени сложности и эффективности. Некоторые из них были нацелены на помощь неопытным пользователям с языками программирования, некоторые на постепенное приобщение малоопытных сотрудниках к новым технологиям.

Большинство из реализованных трансляторов естественного языка (когда речь заходит об русском языке) основаны на словарном методе, то есть использовании некоторого количества больших тезаурусов, и массивного поиска по ним. К сожалению данный подход обладает существенным минусом – словари должен кто-то дополнять и модифицировать по мере надобности.

Из более современных способов преобразования естественного языка хочется отметить использование нейронных сетей и онтологий. Трансляторы основанные на использовании нейронных сетей довольно эффективны в трансляции и формализации обычной речи в команды различных языков, если они не содержат в себе не выбивающихся из обученной модели полей и ключевых слов.

Возникает идея, что если совместить сильные стороны упомянутых выше методов, можно создать транслятор, позволяющий проводить формализацию

естественного текста, содержащего в себе выбивающиеся из естественного языка элементы (которые часто присутствуют в описании схем баз данных), который не будет сильно зависим от ручного дополнения словарей и будет полезен в системах, в которые используются людьми без навыков работы с СУБД - системами управления базами данных. Учитывая, вышеперечисленные методы.

Целью данной работы является разработка и реализация системы трансляции предложений естественного языка в запросы на языке SQL. Для решение этой цели должны быть выполнены следующие задачи:

— изучить существующие методы анализа и формализации естественного языка;
— разработать алгоритм преобразования предложения на естественном языке в запрос на языке SQL;
— реализовать разработанный алгоритм на выбранном языке в связке с выбранной СУБД;
— исследовать реализованный алгоритм и провести анализ его эффективности.

.

ГЛАВА 1. АНАЛИТИЧЕСКИЙ ОБЗОР МЕТОДОВ

1.1. Особенности неформализованных языков

Для того, чтобы начать разговор о переводе, в начале следует определить, чем формальные языки отличаются от естественных.

Что такое естественный язык, читателю должно быть понятно — это язык, на котором разговаривают люди (в данной работе рассматривается русский), содержащий некоторое количество слов в различных позициях и формах. Формальные же языки, созданы человеком для выполнения определённой функции. К примеру, ещё задолго до изобретения ЭВМ люди использовали математическую, химическую, физическую и другие нотации.

Для формальных языков характерно наличие синтаксиса, к примеру выражение 2+2=4 для математики корректно, а 2+=4$ уже нет. Синтаксические особенности формального языка можно разделить на две большие составляющие - символы и структуру. Символами называются базовые элементы языка, которыми могут быть слова, числа, специальные символы. Структура - определенный порядок в позиции символов. Пример приведённый выше был не корректен и с символической точки зрения (в математике не существует символа $) и со структурной (= не может идти сразу после +).

Можно отметить, что и символы, и структура присущи как формальным, так и естественным языкам, но, к сожалению, существует ряд отличий, который усложняет процедуру перевода.

- **Неоднозначность**. Естественный язык полон неоднозначности, и часто для понимания смысла фразы требуется дополнительный контекст. К тому же, естественным языкам свойственно иметь большое количество синонимов. Как следствие одно и то же действие можно выразить некоторым количеством фраз.
- **Избыточность.** Фразы, написанные на естественном языке, пестрят большим количеством слов, которые не влияют на понимание смысла фразы.

С другой стороны, формальные языки лишены этих недостатков и фразы на таких языках всегда однозначны, и почти не содержат в себе излишней информации [1].

1.2. Базовые принципы перевода текста с естественного языка

Прежде чем приступать к анализу методов преобразования естественного языка в формализованные языки стоит определить, из каких этапов оно состоит. Формально, преобразование можно разделить на несколько этапов.

— **Формализацию** или семантический анализ, которая представляет собой декомпозицию, определение смысла и контекста каждого слова, чтобы определить, является конкретное слово командой, частью команды, операндом или же операцией.

— **Трансляцию** в конкретный формализованный язык. Трансляция представляет собой преобразование формально понятных сущностей в конкретную конструкцию языка. Довольно часто, к данному разделу так же прибавляют грамматический модуль, который позволяет соблюсти все правила конечного языка. Так же иногда грамматический модуль включается в формализацию [2].

Для удобства модуль, выполняющий формализацию принято называть анализатором. В зависимости от поставленной задачи анализатор создаётся с использованием разбора предложения разными методами, о которых поговорим далее.

Опорный пункт формализации - определение свойств каждого отдельного слова. Довольно часто, определение происходит с использованием разных тезаурусов, а также небольших программных модулей, осуществляющих анализ. Его можно разделить на несколько ключевых составляющих.

— **Морфологический.** На этом этапе учитываются части речи, падеж, род, число и другие признаки, с целью как определить зависимость между

словами, так и выявить изначальные формы слов, которые могут потребоваться для поиска, создания или любых других действий.

— **Синтаксический.** Данный этап включает в себя анализ всего предложения, определяющий зависимости между его частями, смысловые части предложения, а также включает в себя оценку грамматических зависимостей внутри его.

— **Семантический.** Этап весьма условен, и не является отдельным модулем, а просто означает конец процедуры анализа. Целью всей процедуры анализа является выявление общего семантического смысла предложения, а также установление зависимостей внутри него, что поможет в дальнейшем пере конвертировать данную смысловую фразу на формальный язык.

Далее подробнее рассмотрим этапы и их возможные реализации. Стоит сразу же отметить, что, ещё начиная с первых попыток реализации машинного обучения, использовались словари, заранее созданные человеком.

Морфологический этап претерпел много изменений, за свою историю. Изначально использовался чисто словарный подход, как следствие анализ сводился к поисковой оптимизации. Но из-за банального перебора, требовалось, во-первых, иметь довольно большой словарь, что приводило к ухудшению скорости поиска, а во-вторых, для того, чтобы поддерживать словарь в актуальном состоянии, часто требовалось человеческое вмешательство, в связи с чем данным метод был модернизирован.

Модернизация, которая нас интересует - применение словарного метода в сочетании с алгоритмом процедурного разбора, который в свою очередь представляет собой разделение слова на составные части (основу, суффикс, окончание и т. д.). Нужно заметить, что использование только процедурного анализатора применимо к русскому языку не оптимально, в связи с большим количеством слов с нестандартной структурой. Комбинированный анализ,

использующий оба метода, заключается в их последовательном применении. В начале применяется процедурный анализ, и если же определить свойства слова не удалось, в дело вступает словарный.

Синтаксический анализ, основан на использовании созвучно названных синтаксических словарей, содержащих информацию не о конкретных словах, а о типичных зависимостях, а также не редки случаи, когда синтаксический анализатор основывается на результатах морфологического. В большинстве реализаций разбор основывается на построении древовидных структур. К примеру, базовое дерево при работе с русским языком, делит предложение на именную и глагольные части, что позволяет определять зависимые и главные слова в предложении. После построения оного следует этап составления дерева фразы. Дерево фразы представляет собой базовую структуру предложения, листьями которого являются семантически связанные между собой множества слов. Для наглядности - на рис.1.1 изображен разбор простого предложения.

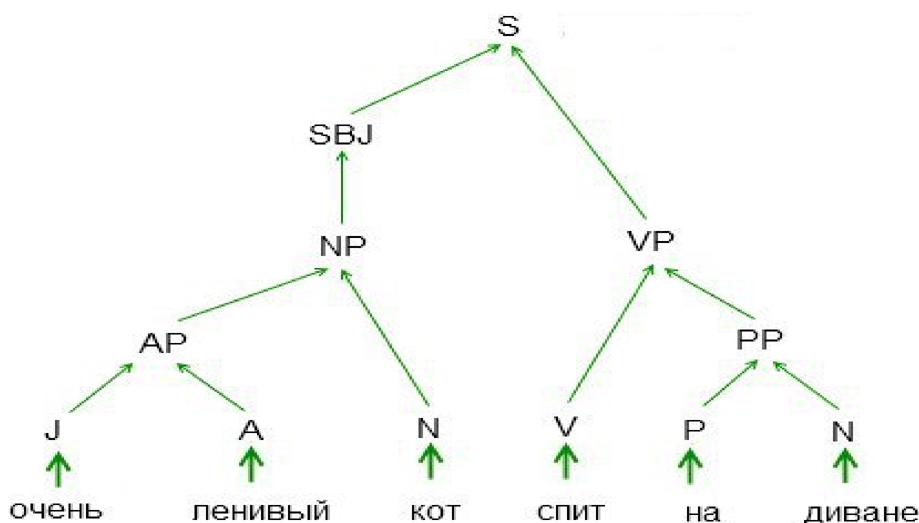

Рис.1.1. Древовидный разбор

Источник: [3].

Базовое дерево фразы имеет один минус - в нём могут возникнуть циклы, если в предложении была сочинительная связь. Чтобы такого не происходило, дерево можно усовершенствовать. К примеру, можно создать древо подчинительной и сочинительной связи. Ещё один способ усовершенствования

- составить дерево непосредственных составляющих. Его отличие от базовой версии - дерево учитывает близость слов, в том числе определение порядка применения прилагательных.

Рассмотрим процесс трансляции. Чтобы понять, какой функционал должен содержать модуль, стоит понять, что чаще всего возвращает анализатор. Формат вывода может быть самым различным. К примеру, в некоторые реализациях это вектор параметров, где-то список слов с некоторым приоритетом, где-то древовидная структура и другие похожие структуры. Следовательно, модуль трансляции должен проводить преобразование некоего формализованного набора данных в конечный (тоже формализованный) язык, учитывая его грамматические правила. Что фактически означает, подстановку данных в грамматические конструкции итогового языка, учитывая грамматику. Как следствие, данный этап не является особо алгоритмически сложным и не сильно требователен к ресурсам.

1.3. Особенности перевода в SQL

Основные особенности перевода в SQL - завязаны на его строгой структурированности и формализованности. Более строгая структурированность характеризуется малым количеством типовых команд и их форм. Язык SQL используется для обработки и отображения информации, хранящейся в реляционной базе данных.

Основную задачу перевода можно упростить до выявления типа команды, определения полей и дополнение их грамматическими конструкциями языка. Но такой алгоритм применим только на базовом уровне, т. е. с запросами обычной структуры, без использования соединений или вложенных запросов. К тому же возникают сложности, если предложение на естественном языке плохо сформулировано, или содержит большое количество лишней информации, но такая проблема характера не только для SQL, но и для большинства формализованных языков [4].

На втором аспекте хочется остановиться поподробнее. При попытке перевода связанных запросов, с использованием вышеописанных методов формализации возникают сложности. Требуется не только определить какую и откуда информацию нужно вывести, но и какую последовательность связей меж таблицами нужно построить для создания запроса. Аналогичная проблема возникает - когда требуется использовать вложенные запросы. Решить данную проблему используя вышеописанные подходы довольно затруднительно и хочется взглянуть на способы, которые помогут в разработке специального анализатора.

1.3.1. Apache OpenNLP

Библиотека для языка Java, созданная Apache, основанная на машинном обучении и приспособлена для анализа естественного языка. Упомянуть её хотелось прежде всего потому, что она поддерживает ряд функций, который требуется для реализации анализатора. Во-первых, стоит отметить малый синтаксический анализ — это процесс построения древовидной зависимости, аналогичной построению простого дерева фразы. Во-вторых - механизм именного распознавания. На этом механизме стоит остановится подробнее, так как он достаточно важен для нашей реализации. В механизме используется заранее заданный список сущностей, в случае взаимодействия с реляционной базой, в качестве списка можно установить именование столбцов конкретных таблиц. Используя данный список, механизм анализирует фразу и сопоставляет отдельные слова сущностям. Применяя данный метод, удобно определять используемые столбцы для запросов.

Последний интересный метод - распознавание частей речи, который в сочетании с малым синтаксическим анализом позволяет расширить функциональность анализатора.

1.3.2. GATE

General Architecture for Text Engineering - набор расширений для Java направленный на обработку естественного языка. В основном, он используется в задачах, связанных с добавлением и работой с выделением смыслового содержания текста, а также имеет встроенный набор утилит для создания и работы с аннотациями. Кроме того, набор включает в себя большое количество утилит, среди которых: инструмент разработки поддерживающий редактирование текстов, поддержка облачных технологий, позволяющая работать с крупными лингвистическими проектами и много других утилит. И самая интересная из утилит, это ANNIE. Расшифровывается как “A Nearly-New Information Extraction System”, и содержит в себе довольно обширный функционал по обработке естественного языка, в том числе механизм именования сущностей, малый синтаксический разбор, токенизацию - разделение частей текста на осознанные компоненты и различные способы выделения отдельных высказываний.

1.3.3. ETAP-3

Система, разработанная в Институте проблем передачи информации имени А.А. Харкевича для перевода английского на русский язык и обратно. Основана система на своде правил и включает в себя модуль машинного перевода, в том числе псевдодревовидную структуру зависимостей русского языка. Последняя представляется особо интересной. Данная структура является частью национального корпуса русского языка, и содержит в себе информацию о более чем 600 тыс. слов, с полной морфологической и синтаксической информацией о них. Данная структура представляет интерес, для разработки анализатора в качестве тезауруса.

1.4 .Оптимизация процесса формализации

В последнее время, благодаря многочисленным исследованиям в области машинного обучения, в том числе связанных с нейронными сетями, появилось

ещё несколько способов анализа естественного языка. Всё чаще и чаще в качестве анализатора используются рекуррентные и нейронные сети свёртки. Существует несколько исследований, которые рассматривают данные анализаторы с точки зрения эффективности и точности, в применении в данном вопросе.

Отдельно стоит отметить, что упоминание двух типов нейронных сетей выше не случайно. Хотя чаще всего, бытует мнение, что рекуррентные нейронные сети лучше справляются с задачей обработки естественного языка и разбития предложений на осмысленные операторы, однако нейронные сети свёртки могут быть эффективнее, в случае если их обучение проходит на специальных графических платах, оптимизированных под работу с нейронными сетями.

При использовании сетей свёртки обычно выделяют два «вида» обработки. Семантический парсинг и логическую связку. Семантический парсинг по сути является задачей интерпретации, которая требуется от анализатора в данной задаче, а вот логическое связывание может быть избыточным. Логическое связывание представляет собой прямое соединение семантической интерпретации естественного языка и соответствующих ему записей из базы данных. Хотя это и кажется прогрессивным и полезным функционалом, в задаче конвертации и трансляции естественного языка данный метод имеет малую полезность, т. к. этот функционал скорее используется для обучения нейросетей и других вопросов, не связанных напрямую с тематикой данной работы.

Очевидный плюс, связанный с данным типом анализатора при решении данной задачи – возможность анализировать вопросы со сложной структурой. К примеру, запрос, содержащий некоторое количество соединительных операций, но логически выстроенный из 4-6 слов (к примеру, если мы рассмотрим схему HR, вывод всех сотрудников, работающих в каком-то конкретном городе) при использовании анализатора, основанного на словарях, довольно сложно автоматизировано обработать и преобразовать.

Чаще всего, при работе с анализатором основанном на нейронных сетях, для обучения анализатора на какой-то конкретной базе данных генерируются перекрестные запросы по случайным полям, между одной или некоторым количеством таблиц. Затем, из данных запросов исключаются логически не корректные (что требует некоторых затрат человеческих ресурсов), а следом запускается обучение нейросети на данных экземплярах.

Одна из интересных разработок, основанных на технологии нейронных сетей, является система [10].

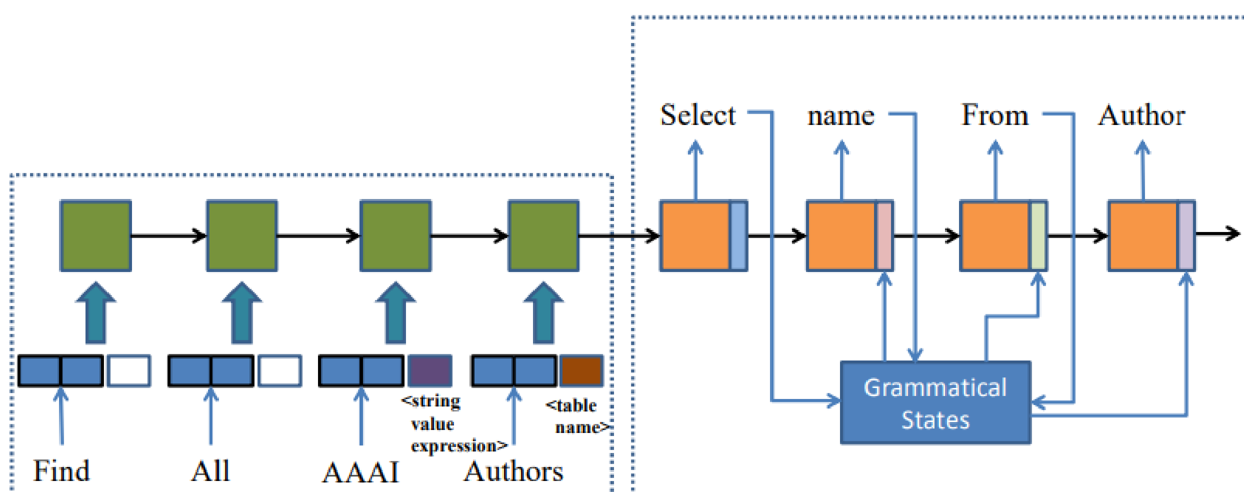

Рис.1.2. Модель реализации на основе нейросети

Источник: [10].

Алгоритм работы, представленной в [10] изображен на рис.1.2, сам же процесс и результат работы авторов можно сформулировать следующим образом. С помощью некоторого количества волонтёров была создана база запросов на естественном языке, большая часть которых использовалась для обучения нейронной сети, вкупе со специально сгенерированными запросами (которые фактически являются вручную преобразованной версией запросов на естественном языке). Кроме этого, для проверки результата были сгенерированы запросы содержащие сочетания случайных таблиц (от 2х и более), далее часть запросов, которая являлась некорректной была исключены вручную, затем проводилась проверка правильности конвертации запросов естественного языка

в язык СУБД. Стоит отметить, что нейросеть работала по следующим правилам – во-первых использовался вектор, представляющий собой ряд флагов, определяющий чем, является конкретное слово (именем таблицы, частью выводимого кортежа, операндом и т. д.), во-вторых, для формирования грамматически верных SQL запросов использовался написанный авторами сторонний Фреймворк, чтобы не нагружать нейросеть дополнительными нагрузками.

Результаты данного исследования показали высокую эффективность перевода подобным образом (корректность на схеме IMDB была в районе 90%), и что не мало важно, хотя данный перевод был нацелен только на запросы выборки (т. е. запросы, содержащие в себе обновление, удаление или создание не рассматривались авторами), стоит отметить интересную особенность. Обученная ими нейросеть была способна отличить не только запросы, адресованные к одной таблице, но и формировать перевод, который в итоге содержал соединение 4х таблиц между собой.

Рассмотрев анализаторы, использующие в своей основе нейронные сети, хочется отметить одну закономерность – хоть анализатор, требующийся для преобразования в SQL и должен быть специфическим, и нейронные сети можно применить и при анализе простых запросов, в которых требуется просто разобрать предложение и выделить нужные параметры, а также тип команды – для решения этого вопроса намного проще применять анализ, разобранный в предыдущем разделе. А вот при разборе сложных запросов – нейронные сети подойдут идеально, и это самый лучший из разобранных вариантов.

ГЛАВА 2. ОБЩЕЕ ОПИСАНИЕ АЛГОРИТМА

Чтобы приступить к решению задач, требуется для начала определить общее направление алгоритма и основные его элементы. Первым элементом, который нам потребуется очевидно будет лингвистический преобразователь. Необходимость данного модуля понятна – т.к. естественный язык имеет довольно большой спектр различных форм построения одного и того же

высказывания. И так же стоит отметить, что данный модуль будет необходим ещё и из-за большого синонимичного набора слов (одно и то же действие/объект можно обозначить по-разному) и каждое такое слово может быть не только в базовой форме.

Следующий необходимый элемент – транслятор, преобразующий упрощенную лингвистическим анализатором команду в SQL. Исходя из ранее полученных знаний, стоит сразу же отметить что транслятора нам нужно два, для простых и сложных SQL запросов. Так же понадобится как минимум ещё один модуль, позволяющий организовывать взаимодействие с СУБД. Как следствие, вырисовывается стретегия, которую можно увидеть на рис.2.1.

После ввода команды от пользователя, пользовательский интерфейс передаёт её в лингвистический анализатор, задача которого определить важные атрибуты (такие как тип команды, объект с которым требуется выполнить действие и различные параметры), после чего преобразовать их в базовую форму, а так же заменить синонимами если требуется. После этого из команды атрибутов и параметров (в изменённой форме) создаётся вектор, который передаётся в модуль распределения.

2.1. Процесс формализации

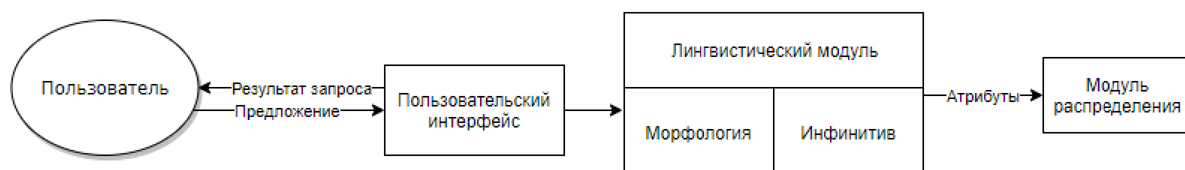

Рис.2.1. Процедура разбора

Основная идея алгоритма состоит в первую очередь во всевозможной формализации конкретного предложения. Для того чтобы этого достичь следует избавиться от всевозможных словоформ определяя их инфинитив, но при этом сохраняя информацию о первоначальной формулировке. Изначальные

формулировки могут помочь в определении логических взаимосвязей между операндами в предложении, так что знать их характеристики тоже полезно.

Задача модуля распределения – выяснить, какому из трансляторов передавать данный вектор. Решение принимается следующим образом – мы имеем информацию о конкретной схеме в СУБД, полях, хранящихся в таблицах, ключах и ещё ряде параметров. По ним модуль распределения сможет определить, хранится ли вся нужная нам информация в одной таблице, или же для получения/обновления/создания нам потребуется соединять несколько таблиц. Если первый вариант, то вектор и управление перейдёт в статический модуль формализации, иначе в модуль используйющий нейронную сеть.

Касательно работы модулей формализации, хотелось бы начать разговор со статического модуля. Планируется организовать работу по следующему принципу – в зависимости от типа команды, выбирается шаблон, в который подставляются параметры из вектора. Стоит указать, что шаблоны имеют вложенную структуру (т.к. одна и та же команда может включать различные виды параметров. К примеру выборка может быть как в базовом виде, так и содержать элементы сортировки, группировки и т.д.).

Второй модуль формализации, так же как и первый, принимает вектор, содержащий команду и ряд параметров. Но для получения результата дополнительно используется общая карта путей, полученная с помощью нейросети, что в свою очередь позволяет получать SQL запросы, испольщующие операцию соединения.

Касательно способов реализации лингвистического модуля, хотелось бы отметить несколько возможных инструментов, которые могу тпригодиться при реализации.

— **Apache OpenNLP**. Набор программных методов, основанных на машинном обучении, основной целью которых является анализ естественного языка. Из возможностей хотелось бы упомянуть синтаксический анализ, метод распознавания сущностей, который

можно связать с сущностями/параметрами из схемы БД, а также метод распознавания частей речи, которое может помочь с токенизацией.

— **RCO**. Интересный модуль анализа текста, позволяющий выявить и описать связи между сущностями в тексте, что может помочь в распознавании зависимых сущностей. Но, к сожалению, у данного продукта не особо удобная модель распространения, и есть вероятность, что протестировать данный продукт не получится.
— **Apache stanbol**. Система, основанная на внедрении семантических сервисов, анализирующая естественный язык. Основные возможности данной системы – определение языка, определение выражений, токенизация, определение части речи, вычленение связности, определение именованных сущностей и стемминг (определение изначальной формы слова). Что интересно, данная система позволяет расширять функционал за счёт включения дополнительных сервисов, которые могут помочь в реализации данного алгоритма. Нам интересны будут два модуля, позволяющие использовать stanbol совместно с Freeling и CELI.
— **Freeling -** это система обработки естественного языка (что важно для нас - с GPL лицензией), разработанная на языке C, которая позволяет определять часть речи и именованные сущности на нескольких языках, в том числе на русском.
— **CELI**. это ещё одна система, работающая с естественным русским языком. Из важного функционала – поддержка определения именованных сущностей, лексический анализ и стемминг.
— **Mystem**. Продукт компании Яндекс, который позволяет определяет все нужные нам части речи: глаголы для определения команды, существительные, для определения таблиц/колонок, а также существительные, не представленные в интересующей нас схеме базы данных - с большой вероятностью это параметры, нужные нам для фильтрации, предлоги, которые позволяют нам связать вместе

несколько токенов. И стоит отметить, что данная утилита позволяет отображать другие формы слова (в том числе начальную форму и единственное/множественное число. К сожалению, данная утилита не имеет удобного API, так что для работы с ней придется использовать файловый обмен).

2.2. Сценарий взаимодейтсвия модулей разбора

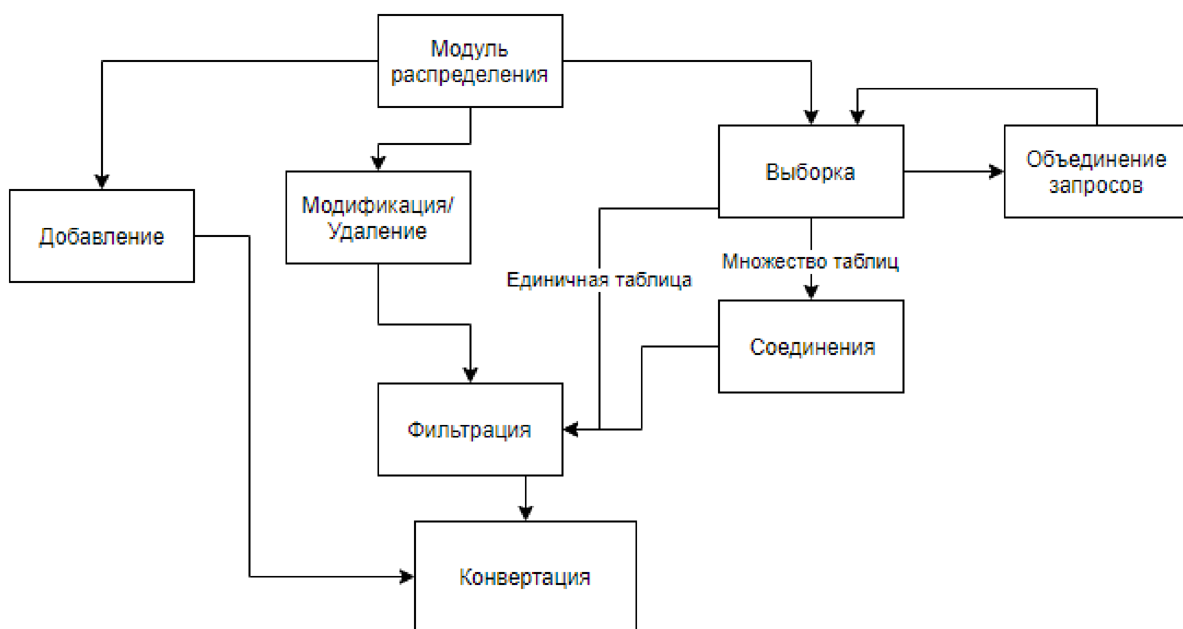

Рис.2.2 Схема выбора вида запроса

Конкретно рассматривая некоторые сценарии разбора запросов (рис.2.2) для начала хочется обговорить механизм обработки запросов, содержащих в себе множественное соединение таблиц. Как говорилось выше - требуется не только определить какую и откуда информацию нужно вывести, но и какую последовательность связей между таблицами нужно построить для создания запроса. Существует несколько способов решения данной проблемы. Один из давно известных – ручное составление словаря-схемы базы данных, с указанием полного пути соединения для каждых двух и более таблиц. Более современный подход – обучение нейронной сети, с целью создания преобразователя [1; 2; 6]. В данной работе хотелось бы взять лучшее из двух подходов, а именно

отсутствие необходимости вручную корректировать решение для других схем базы данных, а также скорость решения конкретной задачи.

Идея такого решения довольно проста - из базы данных получаем набор таблиц, информацию о соединениях по ключам, и эту информацию пускаем в нейросеть, с целью получения полной карты соединений, а также весовой информации. После получения полной карты мы можем использовать её для анализа и составления запроса с соединениями без нужды переделывать карту, пока структура схемы не будет изменена.

Дополнительно, к работе с запросами выборки можно отнести их подвид, состоящий из двух компонентов, соединенных командами – union, minus, intersect. Честно говоря, обработка такого вида запроса мало чем отличается от обработки простого запроса выборки, за тем исключением, что благодаря атрибутам позиции и конкретным ключевым словам, мы разделяем входное предложение на несколько частей, обрабатывая их отдельно, как будто это просто два разных запроса. И после этого объединяем их при помощи нужного оператора.

Самым часто используемым элементом алгоритма (кроме лингвистического модуля разобранного выше) является модуль фильтрации, который используется как в самых разных видах выборок, так и в запросах модификации и удаления. Работа алгоритма фильтрации основывается на целом ряде факторов - во-первых на информации полученной из схемы базы данных, во-вторых на логических и позиционных связях, в-третьих на морфологических характеристиках. Обработчик фильтрации всегда вступает в работу одним из последних, для того чтобы убрать из его поля зрения, все лишние аргументы, которые используются в других частях транслятора.

Конечно обработка запросов выборки это хорошо, но пользователю может понадобиться возможность какой-либо базовой модификации информации в схеме, с которой он работает. Обработка и изменение самой структуры, в данной работе не рассматриваются, по причине излишне нагруженных и не нужен для такого пользователя, на которого нацелен транслятор. Основная сложность при

обработке запросов модификации состоит в определении точного действия, с помощью которого можно определить, что конкретно требуется сделать с изменяемыми данными. Используя информацию из лингвистического модуля, такую как морфологические характеристики конкретных слов, наличие ключевых или указывающих слов, итоговый запрос может представлять собой как полную замену данных, так и изменение с использованием входных значений. Кстати важно будет заметить, что модуль так же обладает возможностью обнаруживать и обрабатывать условия модификации заданные неявно, т.е. с использованием словесных конструкций, а не математических операторов.

Намного проще организована работа модуля обработки запросов удаления. Основная часть запроса состоит просто из определения объекта базы данных, из которого требуется удалить данные. Затем остаётся только определить какие конкретно данные требуется удалить, для этого данный модуль передает управление в модуль обработки фильтрации, который способен проанализировать остальную часть предложения, с целью поиска маркеров-фильтров и ключевых слов.

Следующий модуль обрабатывающий запросы, о котором хотелось бы поговорить - модуль добавления данных. Алгоритмически он довольно прост, но представляет некоторую сложность для пользователя с пониженным уровнем понимания происходящего. Дело в том, что для добавления информации в таблицу базы данных, следует во-первых указать все ключевые поля, в которые следует вставить значения (возможен пропуск некоторых столбцов базы данных, если это позволяет структура таблицы), во-вторых нужно указать значения, которые требуется добавить. Но проблема даже не в том, что пользователь может забыть какое-то конкретное поля, а в том, что порядок входа полей и их значений должен быть одинаковым. К большому сожалению, чаще всего алгоритмически сложно формализовать предложение настолько, чтобы стало однозначно понятно, какое значение относится к какому. И существует два возможных исхода, не корректного указания порядка добавляемых полей. Во-первых

пользователь может получить ошибку, к примеру в случае, если типы данных не совпадают (пользователь может попытаться вставить в поле численного типа текстовое значение). Во-вторых – всё может пройти нормально, но не совсем так, как этого ожидал пользователь. Если например поменять местами значения полей «имя» и «фамилия», то данные прекрасно добавятся, только вот значения будут логически не корректными. Как следствие, при вводе новых данных, на пользователя возлагается дополнительная ответственность за логически корректную формулировку предложения.

Последний модуль - модуль конвертации, принимающий на вход различные векторизованные собрания данных, и конвертирующий их с помощью шаблонизации в корректные для языка SQL запросы. После этого, остаётся только исполнить созданный запрос (или отобразить его, в зависимости от настроек подключения), что будет происходить при помощи модуля взаимодействия с СУБД.

ГЛАВА 3. РЕАЛИЗАЦИЯ ТРАНСЛЯТОРА

Основываясь на разработанном ранее алгоритме, хотелось бы начать разговор о практической реализации с определения и обозначения общей структуры взаимодействия основных классов в трансляторе. Схему можно увидеть на рис.3.1. Так же, для удобства понимания на схеме отображены способы общения классов между собой, а именно какую информацию каждый из них требует, и из какого источника получает.

Основной класс выполняющий общую логику и отвечающий за основные взаимодействия с методами разбора текста - ParserMain. Именно в нем располагаются основные управляющие элементы, принимающие решения об определении дальнейшего хода алгоритма. Подробнее рассмотрим их ниже.

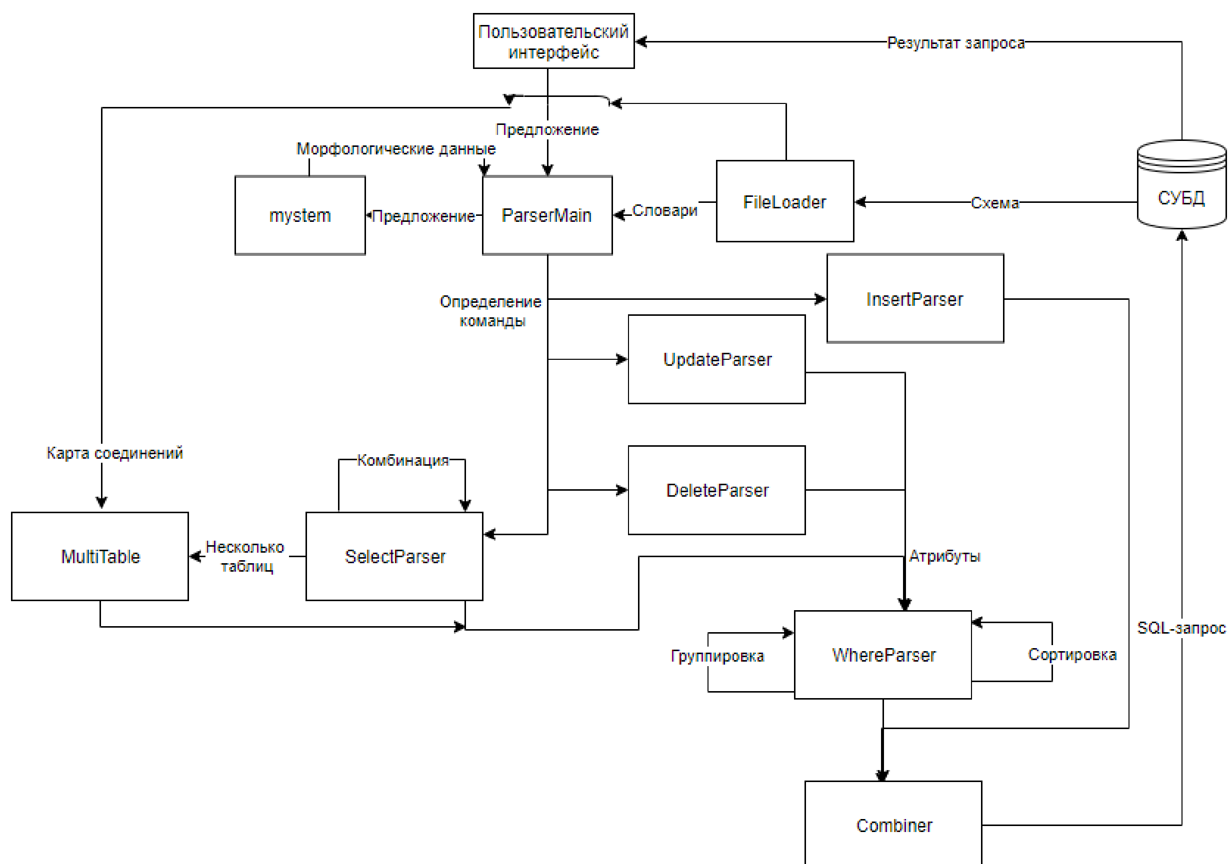

Рис.3.1. Структура классов транслятора

3.1. Основные инструменты разбора текста

Первый этап анализа любого текста, поданного в транслятор это разбор его компонентов, основанный на морфологической информации, сведений из системы управления базами данных и ключевых словах. Начнём с элементов класса ParserMain.

— **Decide**.Базовая процедура управления которая решает каком порядке и какие манипуляции с входным выражением нам предстоит сделать. Именно в данной процедуре происходит последовательный анализ текста на естественном языке.

— **TableField**. Один из вспомогательных классов, используемый для хранения и манипуляций с данными полученными из СУБД. В том числе, в нем хранится информация о выясненной принадлежности поля к одному из объектов схемы.

- **FindDBObjects**. Функция обработки и нахождения используемых в пользовательском выражении таблиц и других объектов СУБД. Используется для анализа входного выражения на предмет явного указания конкретных объектов, с которыми точно предстоит работать.
- **FindFields**. Функция определения вхождений полей таблиц схемы, вне зависимости от их принадлежности к определенным таблицам. Работает в паре с функцией CheckFields. Функция находит все вхождения полей в исходное выражение и сохраняет их для дальнейшего определения принадлежности к объектам базы данных
- **CheckFields**. Дочерний инструмент функции FindFields и довольно важный участник разбора фактически любого текстового выражения. Так как мы разбираем поля вне зависимости от того, к какому объекту они относятся, у нас возникает ряд случаев, когда поля определяются неоднозначно (могут относиться к нескольким таблицам сразу). В таком случае мы не можем однозначно определить, из какой конкретно таблицы их хотел взять пользователь, но по правде говоря чаще всего нет разницы, когда мы можем определить таблицу/таблицы по другим значениям (из условия where или по другим полям, которые хочет отобразить пользователь). Для того чтобы это сделать, нам для начала требуется определить - возможно ли выполнить запрос пользователя обращаясь к одной таблице, или нам потребуется использовать соединения. Для того, чтобы это сделать проводим следующую операцию - перебираем таблицы, в попытке “собрать” наибольшее кол-во полей в одной таблице и убираем объекты, поля которых дублируются в других таблицах. К примеру поля: “имя” входит в сотрудник, отдел и регион. “телефон” входит в отдел, сотрудник и менеджеры, “номер” в регион и сотрудник. Путем нескольких итераций мы понимаем, что все 3 поля присутствуют в таблице сотрудник и рассматривать другие смысла нет. В случае же, если описанным выше

алгоритмом объединить поля не получается, управление перейдёт в класс MultiTable для обработки соединений.

— **CheckCommand**. Основная функция определения режима. Используя словари и морфологическую информацию полученную с помощью yandex mystem, определяет основной тип команды, и в случае будущей обработки запросов модификации так же уточняет действие, которое нужно будет сделать с данными.

— **StringArguments**. Вспомогательный класс который используется как основная структура хранения обработанных данных. Класс отвечает за определение варианта использования данных, такие как - какую форму слова стоит использовать. В базе данных могут таблицы/поля могут хранится как в произвольном падеже/форме, так и в первой/инфинитиве. Также благодаря этому классу удобно взаимодействовать и использовать псевдонимы, когда они необходимы. К примеру псевдонимы автоматически генерируются и добавляются перед нужными полями в случае, когда запрос включает обработку соединений таблицы самой с собой.

— **AnotherObjects**. Одна из важных функций, необходимая для обработки условий и операций ввода и модификации данных. В обычном текстовом предложении часто присутствует некоторое количество предлогов и связующих слов, не относящихся к смысловой составляющей предложения. Нам важно их отделить от потенциальных объектов для фильтрации и модификации данных. В данной функции рассматриваются токены не относящиеся к структуре базы данных и к элементам управления. Нас интересуют прежде всего существительные, прилагательные, числительные которые могут составлять часть выражения. Конечно некоторые предлоги/частицы также могут относиться к условию (намекать на способ модификации данных. К примеру - увеличить НА значений), но их удобнее будет проанализировать при рассмотрении условий/основных частей

операций модификации, т.к. на данном этапе нас не интересует их позиционная составляющая, а важны лишь морфологические/синтаксические характеристики.

3.2. Инструменты обработки запросов выборки

Выборка – это самый часто используемый имеющий наибольший функционал в данном трансляторе тип запроса. Хотелось бы поподробнее рассмотреть элементы класса SelectParser, так как они представляют наибольшую алгоритмическую ценность.

— **FindMainArguments**. Основная операция класса SelectParser. Используя ранее выделенные объекты базы данных и морфологическую информацию слов, определяем основные объекты для вывода. Данный разбор строится прежде всего на позиционных составляющих и пунктуации. Первым делом определяем поле базы данных, располагающееся ближе всего к команде управления (одной из вариации слова “выведи”, “отобрази” и т.д.).

  После определения поля у нас есть целых 3 пути. Во-первых, у нас может быть больше одного поля, которое следует вывести. В этом случае рассматриваем располагающиеся рядом поля, с целью определить, требуется ли их вывести, или же их следует отнести к другим частям запроса. Во-вторых возможно поле требуется каким-то образом преобразовать. Анализируем относящиеся к этому полю ключевые слова или операции модификации и смотрим не требуется ли сделать что-либо ещё. Ну и в-третьих, после вышеперечисленных операций перейти к разбору следующих частей предложения, а также пометить использованные поля, чтобы они по ошибке не включились в другие части запроса.

— **FindGroupByHaving**. Одна из функций парсера фильтрации, непосредственно относящаяся к парсеру выборки. В случае, если во время первоначального разбора строки, был замечен маркер,

указывающий на включение раздела group by, вместо функции обнаружения фильтров для раздела where, запускается данная. Первым делом определяем поля, по которым следует группировать. А затем, уже из этой функции вызывается базовая функцию определения фильтров. Если в условие where попал один из аргументов, по которым мы группируем, просто выносим их в условие having и отдаём управление обратно управляющему классу.

— **FindOrder**. Ещё одна интересная функция парсера условий, позволяющая определить функционал сортировки выводимых данных. Алгоритм работы функции основан на ключевых словах в связке с позиционной информацией. С помощью морфологических данных, определяем есть ли во входной строке команда, которая может указать на сортировку, затем позиционным способом выделяем поля, по которым требуется сортировать (порядок сортировки зависит от порядка указанного пользователем во введенном тексте). Функция имеет 2 алгоритмических пути, один из которых добавлен с оглядкой на возможные ошибки пользователя. Первый случай обрабатывает логически связанные с ключевым словом поля таблиц, второй же напоминает формализованное представление оператора сортировки, и если в формулировке естественного языка ключевой маркер помещён в конец, и после него нет других операторов, указывающих на фильтрацию или операции с данными, все поля указанные после принимаются в качестве сортируемых.

— **CombineofSelect**. Отличающаяся от остальных (по крайней мере алгоритмически) функция взаимодействующая с комбинациями выборок. Реализовано несколько вариантов работы, исходя из формата ввода и команды для соединения. Существует два формата ввода: разделяемый и совмещённый.

Разделяемый подразумевает фактически раздельный ввод команд, соединенный оператором комбинации. И алгоритмически представляет

собой рекурсивный запуск обработки запроса выборки, для каждой из выделенных частей предложения. Совмещённый же подразумевает дублирование вывода, иными словами - и в первом и втором запросе выводятся одинаковые поля и используется одинаковая фильтрация, но могут быть различные таблицы. Из команд ввода поддерживается работа с minus, union, intersect, обработка которых основана на морфологически-словарном методе.

— **SelfConnect**. Последняя из любопытных функций SelectParser. В случае, если во время первоначального разбора основных команд, обнаруживается маркер соединения таблицы самой, с собой, управление передаётся в данную функцию. Для формализации в данной ситуации строка разделяется на 2 части, фактически определяя для каждой из таблиц, какие данные следует вывести, так как для каждого из выводимых параметров необходимо будет указывать псевдоним таблицы, из которой его следует выводить. Псевдонимы генерируются внутри данной процедуры, от пользователя не требуется их указывать. Так же определяется условие соединения таблиц, подразумевая что взгляд на него соответствует стандартной логике естественного языка – то есть или условие позиционно явно относится к одной из таблиц, либо соблюдается левосторонняя логика построения.

3.3. Прочие элементы взаимодействия с запросами

Остальные операторы обработки, которые не хотелось бы выносить в отдельные разделы. Наиболее важный и пригождающийся в очень большом количестве запросов – WhereParser, занимающийся как обработкой условий для нескольких типов запросов, так и принимающий относительно участие в обработке некоторых функций запросов выборки.

— **FilterParameters**. Одна из вспомогательных структур класса WhereParser. Удобный инструмент для построения единичного условия (формата A=B,A>B, A>B+10 и т.д.), грамотной конвертации и удобного

взаимодействия с другими элементами парсера. Так же благодаря данному классу происходит удобная операция соединения нескольких условий, путём добавления соединительной частицы с помощью переписанной операции сложения.

— **FindCombinationOfFilters**. Основная функция анализа парсера условий. Логика работы представлена следующим образом - анализируем первый из не используемых в других операциях элемент базы данных, проверяем соединенные с ним элементы. Во-первых смотрим нет ли управляющих конструкций (к примеру имя=Петров, конструкция которую фактически не нужно анализировать, требуется просто вставить в запрос), проверяем можно ли найти соседний операнд, и определить операцию. Операнд может быть представлен как и другим полем базы данных, так и числительным, существительным или просто другим словом. Также операция может быть задана неявно, и чтобы её конвертировать, мы обращаем внимания во-первых на формулировку соседних указательных/важных токенов, а также на предлоги, которые могут указать какую операцию нам нужно сделать с данными.

— **FindFilters**. Если предыдущая функция отвечала за базовые алгоритмы обнаружения и вычленения элементов условия, в данной функции мы рассматриваем полученную на предыдущем этапе информацию и конвертируем её в пригодный для СУБД вид. Так же именно здесь происходит анализ и конвертация неявно заданных условий.

— **CheckSecondInLine**. Данная функция обязана своим существованием одной занятной особенности работы mystem. Дело в том, что в связи с тем, что это текстовый анализатор, он начисто игнорирует числа и операторы сравнения. Так что приходится проходится впрямую по строке, с целью обнаружения числовых значений и операторов сравнения/равенства. Плюс также важно не забывать про конвертацию литералов в условии и объединения базовых математических операций.

— **UpdateFindArguments**. Управляющая функция класса UpdateParser. Именно здесь следует определить, какое действие мы делаем с данными. Основной упор в разборе делаем на словесные маркеры, указывающие на желаемый способ модификации и с какими именно данными нам следует работать. Важно отметить, что в данной ситуации нас интересуют не столько расположение ключевых маркеров, сколько формулировка управляющих конструкций и их взаимосвязей. К примеру, в зависимости от запроса, даже сама основная управляющая фраза (позволяющая определить тип запроса) будет относиться к факторам которые нам следует рассмотреть. В зависимости от ее формулировки мы можем определить, следует ли нам заменить существующие данные, увеличить их (если это возможно) или наоборот уменьшить. Ну и каким способом это следует сделать. К тому же в данной функции мы определяем какие поля следует модифицировать, с помощью позиционного способа и морфологической информации.

— **UpdateFindParams**. После определения принципа модификации, нам во-первых нужно транслировать эти инструкции в более формальный вид, а также определить - чем (или при помощи чего) требуется модифицировать данные. При наличии, определяем литералы, при необходимости заключаем их в кавычки, или же определяем числовые значения, похожим способом как делали это в разделе условий. Кроме того, в зависимости от указанных (функцией выше) инструкций подставляем шаблоны, подходящие под конкретную ситуацию.

— **InsertArguments**. Основной функционал класса InsertParser, использующий уже упоминались выше списки объектов базы и потенциально подходящие нам литералы. Реализовано 2 подхода в работе данной функции. Существует простой вариант, когда пользователь через запятую указывать последовательность полей а следом последовательность литералов, которые соответствуют данным полям. И способ прямого соответствия, когда во входной строке указано

конкретное перечисления с указанием соответствия. Так как физически нет возможности определить отношение полей, имеющих одинаковый тип, корректная последовательность введенных данных остается на совести пользователя. (К примеру и имя и фамилия будут иметь строковый тип, и определить к чему относится какой пункт, кроме как по порядку не представляется возможным).

— **CalculatedPath**. Класс используемый для хранения и обработки значений, полученных на выходе из нейронной сети. Для нас важны следующие данные.

1. Вектор содержащий способ возможного соединения таблиц, в котором как минимум 2 таблицы, но также сюда включаются пути любой длины, которые могут нам пригодиться.
2. Способ соединения таблиц. Используется “старый” способ соединения, т.е. условия соединения таблиц помещаются в раздел where, а порядок соединения таблиц для нас не важен.
3. Вес конкретного пути. Довольно интересная и важная характеристика, которая при модернизации нейронной сети может позволить указывать предпочитаемые пути соединения, к примеру исключать слишком нагруженные таблицы или же выбирать более длинные пути, если того требует структура схемы.

Так же хочется обговорить не упоминаемый на схеме процесс получения полной карты схемы базы данных. Реализован он следующим образом. С помощью класса FileLoader, получаем полный список таблиц схемы и список ключей, с указанием полей и таблиц. Затем загружаем его с указанием дополнительных параметров (к примеру таких, как степень загруженности таблицы, иными словами факторов, которые могут повлияет на её выбор как промежуточного звена при множественном соединении) в модифицированную нейронную сеть от Octavian-ai, приспособленную для нахождения пути. На выходе получаем супер-карту, содержащую веса и всевозможные кратчайшие

пути внутри нашей схемы. Что позволяет нам не пересчитывать их каждый раз для нового запроса, а просто брать информацию из файла.

— **FromAndWhereWithManyTables**. Данная функция анализирует данные, полученные с помощью нейросети. Основная задача у алгоритма следующая - по введенным таблицам (две и более) требуется определить способ их соединения (базово с минимальным количеством соединений, но возможны и другие условия в зависимости от заданных в нейросеть данных). Алгоритм выбран был следующий - на выходе из нейросети мы получаем коллекцию различных путей, со всевозможными соединениями внутри конкретной схемы базы данных. Коллекция упорядочена по первой из списка таблиц, для более быстрого анализа. Начинаем искать путь с первой таблицы, по весам находим минимальный путь в указанную (следующую) таблицу. На следующей итерации проверяем, нет ли у нас более быстрого пути охватывающего все перечисленные ранее таблицы, а так же следующую по порядку. В результате имеем готовый раздел From для запроса, и дополнительную информацию для раздела where.

3.4. Модуль шаблонизации запросов

Модуль включает функции комбинирования векторизованных данных с помощью заранее заданных шаблонов форм запросов:

— CombineSelect;

— CombineDelete;

— CombineInsert;

— CombineUpdate.

Модуль собирает воедино обработанную до этого информацию, с помощью циклического соединения аргументов, преобразования в нужный. В зависимости от типа команды, в класс подаются разные аргументы требующиеся для конкретной команды. Так же именно данный класс оперирует методами

выполнения составленных запросов и отправляет их результаты в основной управляющий модуль.

Каждая из вышеуказанных функций построена похожим образом, из списка выделяется только обработчик запросов выборки, так как имеет довольно большое количество отличающихся элементов. Кроме стандартного набора областей, которые встречаются чаще всего: основная содержащая информацию которую нужно вывести, раздел таблиц и раздел фильтрации, в зависимости от формулировки входного предложения, могут добавляться разделы группировки, сортировки и фильтрации по группам. Кроме того, важно отметить, что шаблон в некоторых случаях вызывается несколько раз, к примеру при использовании операторов соединения запросов.

Кроме того, именно данный класс содержит в себе дополнительные функции позволяющие исполнять полученные запросы:

— ExecuteSelect;
— ExecuteInsert;
— ExecuteDelete;
— ExecuteUpdate.

После выполнения какой-либо из перечисленных функций, транслятор анализирует полученный ответ от системы управления базами данных. Варианта ответа всего два – либо выводится список полученных данных, который может состоять из выборки нужных данных или вывода информации о успешном изменении данных, либо сообщение об ошибке, которое с небольшим изменением (без отображения номера ошибки, и с пояснением) выводится пользователю.

ГЛАВА 4. АНАЛИЗ ЭФФЕКТИВНОСТИ РЕАЛИЗОВАННОГО АЛГОРИТМА

Для того, чтобы проанализировать работу разработанного транслятора следует рассмотреть некоторые виды запросов и их возможные формулировки.

4.1. Базовые виды запросов

Начнём с простого запроса выборки, не содержащего никаких условий и усложнений. (рис.4.1).

```
Выведи зарплату сотрудников
Select зарплата  From сотрудник
```

Рис.4.1. Базовый запрос выборки

Рассмотрит чуть более усложненную версию, требующую не только определения условия, но и его конвертацию в формальный вид (рис.4.2)

```
Удали сотрудников с зарплатой меньше 100
Delete  From сотрудник  Where зарплата<100
```

Рис.4.2. Запрос на удаление с условием

Хочется заметить, что вне зависимости от наличия и формулировок лишней, не относящейся к основному запросу информации (союзы и другие соединительные элементы) транслятор отрабатывает задачу хорошо (рис.4.3)

```
Увеличь зарплату на 100 сотруднику по имени Иван
Update сотрудник  Set  зарплата=зарплата + 100  Where   имя='иван'
```

Рис.4.3.- Запрос на увеличение численных полей

Касательно запросов добавления данных, у пользователя есть два варианта доступного (и понятного) синтаксиса. (рис.4.3 и 4.4).

```
Добавь в отделы номеротдела, название и геолокацию, значения 10, десятый, 10
Insert into отдел (номеротдела, название, геолокация) Values (10,'десятый',10)
```

Рис.4.4. Запрос добавления

В случае, если пользователь попытается ввести недопустимые значения (к примеру текстовое значение в поле, в котором хранится число), транслятор отработает верно, и даже составить корректный запрос, но система СУБД вернёт ошибку, и мы вернём её пользователю.

```
Добавь в отделы номеротдела=десять, название=10, геолокация=10
Недопустимое значения для добавляемого поля
```

Рис.4.5. Запрос добавления с ошибкой ввода

В зависимости от формулировки, введённой пользователем, неявное условие (которые транслятор самостоятельно формализует условия, которые заданы в текстовом варианте) по умолчанию (при отсутствии ключевых маркеров, которые могут указывать на условие включающее в себя условия неравенства или отрицания) фильтр формализуется при условии равенства. Так же, литералы помечаются кавычками для корректной работы (рис.4.6)

```
Выведи имена сотрудников у которых фамилия Петров
Select имя  From сотрудник  Where  фамилия='петров'
```

Рис.4.6. Запрос выборки с простой фильтрацией

Так же у пользователя есть возможность как ввести условие в формализованном числовом виде, в таком случае транслятор просто подставить эти значения во время парсинга раздела фильтрации, так и ввести его в более сложном словесном виде (рис.4.7)

```
Покажи фамилии сотрудников зарплата которых на 100 больше, чем их номеротдела
Select фамилия From сотрудник  Where (зарплата+100)>номеротдела
Покажи фамилии сотрудников зарплата>(номеротдела-100)
Select фамилия From сотрудник  Where зарплата>(номеротдела-100)
```

Рис.4.7. Запрос выборки с вариациями фильтрации

При использовании операторов сортировки, начинают возникать небольшие сложности или особенности данного транслятора. Так как он основан на морфологическом и позиционном разборе, в некоторых ситуациях запрос будет не полностью соответствовать желанию пользователя. В простом варианте, когда оператор сортировки находится в конце введённой команды, пользователь может в любом виде указать порядок и поля по которым следует её производить (рис.4.8).

```
Выведи зарплаты сотрудников и отсортируй по зарплате
Select зарплата  From сотрудник  Order by зарплата
```

Рис.4.8. Запрос выборки с упорядочиванием

В случае же, когда формулировка запроса пользователя немного не стандартна, и правилами пунктуации пользователь не пользуется, возникает следующая проблема. Анализатор сортировки действует следующим путём –

после определения команды сортировки и поля, по которому следует отсортировать, проходит проверка, нет ли ещё полей, по которым нам следует тоже отсортировать. Но если правило сортировки указано не в конце строки (т.е. после оператора сортировки указаны ещё команды), и как в данном случае (рис. 9) отсутствует оператор перечисления (запятая или соединительный союз), транслятор опускает поля указанные далее, и сортирует только по первому полю.

```
Выведи зарплаты сотрудников, отсортируй их по зарплатам фамилиям и зарплаты должны быть больше 300
Select зарплата From сотрудник  Where зарплата>300  Order by зарплата
```

Рис.4.9. Запрос с упорядочиванием и сложной формулировкой

Если транслятор, в процессе анализа обнаруживает оператор группировки, аналогичным образом, как это происходит с другими операторами фильтрации, добавляется раздел group by.(рис.4.10) Но, если в момент разбора части предложения обнаруживается, что в него включено условие, и это условие основано на поле, упомянутом в разделе группировки, данное условие выносится в раздел having. Остальные же условия остаются.

```
Отобрази зарплаты сгруппированные по номеротдела, и номеротдела>100
Select зарплата  From сотрудник  Group by номеротдела  Having номеротдела>100
```

Рис.4.10. Запрос содержащий группировку

4.2. Усложненные версии запросов

Стоит упомянуть, что транслятор поддерживает добавление псевдонимов таблиц, но данная возможность немного скрыта от пользователя. По причине того, что в обычном запросе, чаще всего не требуется принудительное добавление псевдонимов (особенно учитывая, что названия таблиц в некоторых запросах пользовать может просто не указывать). Однако в ситуациях, когда псевдонимы нужны для корректной работы запросов, к примеру в случае соединения таблицы самой с собой (рис.4.11), для таблиц генерируются временные псевдонимы, и они заменяют названия соответствующих таблиц во всех разделах кроме from. Так же хочется отметить автоматическое подставление имени таблицы перед полем, в ситуациях, когда поля определены неоднозначно.

```
Отобрази фамилии сотрудников, у которых зарплата больше чем у других сотрудников
Select temp1.фамилия  From сотрудник temp1,сотрудник temp2  Where temp1.зарплата>temp2.зарплата
```

Рис.4.11. Запрос выборки с соединением таблицы самой с собой

Ещё один немного не однозначный, и рассчитанный на пользователя, который корректным образом использует правила русского языка элемент работы транслятора. В случае, когда запрос подразумевает работу с операторами соединения (union, minus, intersect), очень многое зависит от правильно построенного предложения. Дело в том, что ключевые маркеры, намекающие транслятору, что требуется использовать несколько запросов и соединить их каким-либо образом, сильно зависят от позиции и логических связей, в которые они поставлены (рис.4.12). Если предложение для разбора состоит из двух логически разделяемых запросов, и в нем присутствуют ключевые маркеры никаких проблем не будет. Если же предложение сформулировано так, что логически разделить запрос на два довольно проблематично, и характерные операторы находятся среди перечисления полей или условий, транслятор попытается формализовать запрос в виде выборки с соединением (если участвует несколько таблиц), либо в виде обычной выборки, если это возможно.

```
Покажи имена сотрудников вычти из них имена сотрудников, у которых зарплата больше 100
Select имя  From сотрудник MINUS Select имя  From сотрудник  Where зарплата>100
```

Рис.4.12. Комбинирование запросов выборки

Отдельно хочется проанализировать работу в отношении запросов, которые не решить без единичного или множественного соединения. Для более гибкой возможности построения запроса, пользователь может указать конкретную таблицу, от которой следует строить связи (рис.4.13), что может помочь сформировать более грамотную цепочку соединений. В случае, когда пользователь хочет вывести поля, которые можно однозначно определить (т.е. они не встречаются в разных таблицах использующихся в запросе), в построенном запросе они будут указаны без привязки к таблице (см рис.4.13).

```
Выведи фамилие и город сотрудника
Select фамилия, город From сотрудник, отдел, местоположение Where сотрудник.номеротдела=отдел.номеротдела

and отдел.геолокация=местоположение.геолокация
```

Рис.4.13. Выборка из нескольких таблиц

Отдельно хочется отметить, что от пользователя не требуется указывать все таблицы, которые требуются для выполнения корректного соединения. Транслятор основываясь на полной карте полученной с помощью нейронной сети, приступает к определению кратчайшего пути, между всеми полями, указанными в запросе. Для удобства, в трансляторе реализовано использование старого диалекта соединения, чтобы была возможность более гибко строить последовательность таблиц, не концентрируясь на порядке таблиц. По умолчанию, алгоритм рассматривает в качестве основного критерия количество соединений, и пытается выбрать такой путь, чтобы их было как можно меньше. Но при надобности, можно заново обработать полную карту, и в качестве веса дополнительно указать такие характеристики как степень желательности использования конкретной таблицы при построении соединительных запросов.

Так же хочется упомянуть о том, что учитывая направленность транслятора на людей слабо разбирающихся в структуре и операциях работы с системами управления базами данных, является возможным просто указание параметров, которые пользователь хочет отобразить, и алгоритм полностью подберёт все нужные таблицы и соединит их в требуемом для получения корректной информации виде (рис.4.14). И если информация, которую требуется вывести (а конкретно некоторые поля) определяются не однозначно, они будут приписаны к первой из таблиц для которой они подходят (считая от начала соединения, т.е. слева).

```
Покажи номеротдела и фамилию
Select отдел.номеротдела, фамилия From отдел, сотрудник Where сотрудник.номеротдела=отдел.номеротдела;
```

Рис.4.14. Выборка из нескольких таблиц без указания базовой таблицы

В заключении хотелось бы рассмотреть случай формулировки предложения без явного указания таблиц, и его потенциальные слабые стороны. В подобных запросах могут быть использованы фактически все рассмотренные выше

элементы (от базовой фильтрации, до комбинаций запросов), и опять же указания на таблицы не требуется (рис.4.15). Только существует одна проблема. В случае, если пользователь указывает только поля, и некоторые из них созданы в разных таблицах, которые никак не связанны между собой (к примеру поле фамилия может быть как в таблице сотрудники, так и в таблице покупатели, и совпадение фамилий ничего не гарантирует). В такой ситуации запрос будет построен, нет стопроцентной гарантии, что он будет таким, каким подразумевал его пользователь, и одной из целью соединения может быть не та таблица, из которой информация нам требуется. Как следствие результат запроса будет не тем, который требовался пользователю.

```
Покажи город, в котором работает человек по фамилии Петров
Select город From местоположение, отдел, сотрудник Where сотрудник.номеротдела=отдел.номеротдела
and отдел.геолокация=местоположение.геолокация and фамилия='петров'
```

Рис.4.15. Выборка из нескольких таблиц с неявным условием

Но нужно отметить, что данную проблему вполне просто решить, если пользователь видит структуру схемы, с которой он работает, и указывает конкретно название, нужной таблицы, содержащей «неоднозначную» колонку.

Так же хочется отметить, что эффективность данного транслятора имеет ограничения, и при обработке запросов, содержащих большое количество структур (к примеру запрос содержащий в себе операцию вычитания двух запросов, каждый из которых представляет из себя соединение нескольких таблиц), запрос получаемый с помощью транслятора будет далеко не самым эффективным с точки зрения времени выполнения и затраченных ресурсов. Транслятор прежде всего нацелен на перевод команды пользователя, и только после этого на оптимизацию. Соответственно, если предложение на естественном языке составлено так, что его прямая интерпретация, представляет собой не самый эффективный способ вывести информацию, запрос транслятора будет проигрывать в эффективности, запросу, который был написан человеком с опытом работы с системами управления базами данных. Но так как данный транслятор прежде всего нацелен на малоопытных пользователей, данный аспект не может являться решающим, в оценке эффективности работы транслятора.

После анализа реализованного транслятора хотелось бы подвезти итоги сравнить полученную реализацию с аналогами. К сожалению, транслятора поддерживающего формализацию русского языка, с похожим набором функций, найдено не было. Но это не помешает нам сравнить его с англоязычными аналогами. В связи с тем, что языки существенно отличаются по строению, вполне логично сравнить не по конкретным предложениям, а по ключевым возможностям и существенным показателям которые логично требовать от подобного типа трансляторов.

4.3. Сравнение эффективности реализованных функций

Для того чтобы начать сравнение требуется сначала определить критерии, по которым будем сравнивать. Основная задача подобных трансляторов – работа с пользователями без особых навыков и знаний по работе с системами управления базами данных, без знания формальных языков, обладающими базовыми понятиями о том, с какой конкретно информацией (схемой) им предстоит работать. Как следствие, задачи относящиеся к построению структуры и модифицирования схемы или структуры базы данных нас не интересуют.

- **Поддержка DQL** – возможность проводить выборку из базы данных.
- **Адаптируемость к изменению схемы** – транслятор должен продолжать работу, после изменения некоторых структурных элементов базы данных.
- **Независимость от конкретной схемы** – при добавлении новой схемы транслятор должен без существенных действий со стороны администратора подстраиваться под новую схему базы данных.
- **Поддержка DML** – довольно часто, кроме работы с выводом данных, пользователям требуется возможность их добавления и модификации.
- **Оптимальность построения запросов** – кроме задачи формализации, транслятор так же принимает решения о использовании конкретных способов вывода данных.

Таблица 4.1

Сравнение ключевых особенностей подходов

Подходы / Критерии	Словарный	Нейросеть	Разработанный транслятор
Поддержка DQL	+	+	+
Адаптируемость к изменению схемы	-	+	+
Независимость от конкретной схемы	-	+/-	+
Поддержка DML	+	-	+
Оптимальность построения запросов	+/-	Зависит от реализации	+/-
Независимость от конкретного диалекта	-	+/-	+/-

Источники: [2;7;10–15].

Как можно видеть из табл.4.1, полученный транслятор имеет ряд преимуществ относительно других подходов к реализации. Благодаря тому, что конечно формализация и шаблонизация запроса происходит на основании токенизированных данных, транслятор легко можно модифицировать для работы с другими системами управления базами данных. Так же исходя из подхода к обработке схемы базы данных, отсутствует нужда в ручной корректировке транслятора, в случае её модификации и по той же причине, возможна работа с несколькими схемами баз данных, для разных сессий. Для каждой из таких сессий, предстоит сгенерировать полную карту путей, но этот процесс легко автоматизируется. В данный транслятор так же включена поддержка работы с языком DML, т.е. возможность модификации табличных данных.

Упоминания вопрос об оптимальности построения запросов, после рассмотрения их эффективности ранее в этой главе, хотелось бы заключить

следующее. Трансляция базовых версий запросов не вызывает никаких нареканий, так же обработка более комплексных версий, включающих в себя множественные соединения, группировку, сортировку и операции фильтрация так же преобразовываются без особых проблем. Но в случае, когда логика запроса неоднозначна или существует более эффективное решение, основанное на вложенных запроса ресурсоемкость полученных транслятором запросов может уступать тем, что создал опытный человек с пакетом знаний в области работы с базами данных. Но учитывая, что данный транслятор направлен прежде всего на взаимодействие с пользователем, без нужного пакета знаний, свою роль он выполняет полностью.

ЗАКЛЮЧЕНИЕ

В заключении, получаются следующие выводы. Результат разработки транслятора вполне удовлетворителен, и его функционал соответствует потребностям средне-статистического пользователя, не имеющего особых навыков в работе с СУБД. Но так же важно отметить, что в случае работы с большими, комплексными запросами, результат созданный транслятором будет уступать работе обученного человека, который обладает навыками написания вложенных и сложно-соединенных запросов, в связи с тем, что транслятор в первую очередь разработан как инструмент-прослойка, помогающий не очень грамотному пользователю взаимодействовать с базой данных, и превращать его мысли в формализованные запросы.

Целью данной работы является разработка и реализация системы трансляции предложений естественного языка в запросы на языке SQL. Для решение этой цели были выполнены следующие задачи:

— изучены существующие методы анализа и формализации естественного языка;

— разработан алгоритм преобразования предложения на естественном языке в запрос на языке SQL;

— реализован разработанный алгоритм на выбранном языке в связке с выбранной СУБД;

— исследован реализованный алгоритм и проведен анализ его эффективности.

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

СУБД – система управления базами данных.

API – программный интерфейс приложения.

NLP – Natural Language Processing, обработка естественного языка.

DQL – Data Query Language, язык вывода данных.

DML – Data Manipulation Language, язык изменения данных.

СПИСОК ТЕРМИНОВ

В данной пояснительной записке к записке к выпускной квалификационной работе применяют следующие термины с соответствующими определениями.

— Формализация – процесс представления содержательной области в виде формальной системы.
— Токенизация – процесс выделения ключевых элементов.
— Семантический анализ – применение методов морфологического и тезаурусного анализа.
— Позиционно-логический анализ – процесс определения каких-либо свойств объекта исходя из его морфологических свойств и расположения относительно других объектов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. «Natural language interface for database: A brief review» // Neelu Nihalani, Sanjay Silakari, Mahesh Motwani. 2011. 9 с. [Электронный документ] URL:(https://pdfs.semanticscholar.org/00e2/6133551f152f4c8913c5442238d30a63ef79.pdf). Проверено 10.05.2019.

2. «Neural machine translation by jointly learning to align and translate»// Bahdanau, D.; Cho, K.; 2014. [Электронный документ] URL: (https://arxiv.org/abs/1409.0473) ICLR 2015. 16 с. Проверено 21.05.2019.

3. «Методы и алгоритмы трансляции естественно-языковых запросов к базе данных в SQL-запросы» // Л.В. Найханова, издательство ВСГТУ Улан-Удэ 2004, 148с.

4. Документация deeplearning4j. URL: www.deeplearning4j.org. Проверено 9.05.2019.

5. «Опыт теории лингвистических моделей “смысл-текст”» // Мельчук И.А. Наука 1982. 345 с.

6. Документация Keras. URL: www.keras.io. Проверено 4.05.2019.

7. «Vocabulary Selection Strategies for Neural Machine Translation» // Gurvan L'Hostis. 02.11.2016 [Электронный документ] URL: https://arxiv.org/abs/1610.00072. Проверено 7.05.2019.

8. Документация Stanbol. URL: www.stanbol.apache.org. Проверено 15.05.2019.

9. Документация API Яндекс. URL: www.tech.yandex.ru. Проверено 29.04.2019.

10. «An Encoder-Decoder Framework Translating Natural Language to Database Queries» // Ruichu Cai, Boyan Xu , Xiaoyan Yang , Zhenjie Zhang , Zijian Li. Advanced Digital Sciences Center, Illinois at Singapore Pte. Ltd, Singapore 2017г. 6с.

11. Text/Relation Database Management Systems: Harmonizing SQL and SGML // G.E. Black and others. 1994 [Электронный документ]. URL:

https://link.springer.com/chapter/10.1007%2F3-540-58183-9_54. Проверено 24.04.2019.

12. TypeSQL: Knowledge-based Type-Aware Neural Text-to-SQL Generation // Tao Yu и другие. 7 с. [Электронный документ] URL: https://arxiv.org/abs/1804.09769. Проверено 23.04.2019

13. Natural language to SQL conversion system // Anil M. BhadGale 2013. [Электронный документ] URL: https://www.academia.edu/3311986/Natural_Language_to_SQL_Conversion_System. Проверено 22.02.2019.

14. SQL generation and execution from natural language processing // Saravjeet Kaur. 2012. 9 с. [Электронный документ] URL: http://www.researchmanuscripts.com/isociety2012/54.pdf. Проверено 20.04.2019.

15. Automatic SQL Query Formation from Natural Language Query //Subharata Sengupta. 2014. [Электронный документ] URL: https://www.researchgate.net/publication/296561401_Automatic_SQL_Query_Formation_from_Natural_Language_Query. Проверено 14.05.2019.

16. Модули Neuroph [Электронный документ] URL: http://neuroph.sourceforge.net. Провеерно 21.03.2019.

17. Документация ApacheOpenNLP [Электронный документ] URL:https://opennlp.apache.org/docs/1.9.1/manual/opennlp.html Проверено 23.04.2019.

18. Natural Language Interface for Java Programming: Survey // Archana R. Shinde 10.2017. 20 с. [Электронный документ] URL: http://www.ijritcc.org/download/browse/Volume_5_Issues/November_17_Volume_5_Issue_11/1510221503_09-11-2017.pdf. Проверено 11.05.2019.

19. Документация Яндекс mystem [Электронный документ] URL: https://tech.yandex.ru/mystem/doc/. Проверено 10.05.2019.

20. Документация Oracle [Электронный документ]. URL:https://docs.oracle.com/en/database/. Проверено 5.05.2019.

Приложение

Исходный код программы

```
public static void  Decide ()
 {
    try {
        String Result;
        String Input = null;
        java.io.BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        Input = in.readLine();

        ToFile(Input);
        String[] cmd = {"/bin/sh", "-c", "cd //home/nik/Documents/mystem; ./run.sh;"};
        Process p = Runtime.getRuntime().exec(cmd);
        try {
            Thread.sleep(100);
        } catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
        }
        String POS = FromFile("");
        String[] Params = POS.split("\\{|\\}");
        ArgumentsInfos=new StringArguments[Math.floorDiv(POS.length(),2)];
        int k=0;
        Pattern pat=Pattern.compile("");
        for (int i=1;i<Params.length;i+=2)
        {
             if (Params[i].contains("=S")) {
               String TempFind = Params[i].substring(0, Params[i].indexOf("=S,", 0));
               TempFind = TempFind.substring(TempFind.lastIndexOf("|") + 1, TempFind.length());
               TempFind = TempFind.replace("?", "");
               ArgumentsInfos[k] = new StringArguments(Params[i - 1], TempFind, "", "S", Input.indexOf(Params[i
- 1]), false);
            }
        }
        Result="";
        String Command=CheckCommand(Params);
        String OrderPart="";
        String OrderClause="";
        String[] OrderByAnalogs={"отсортируй","сортируй", "упорядочи","отсортировывать"};
        if (Command.equals("Select"))
            for (int i=0;i<OrderByAnalogs.length;i++)
                for (int j=1;j<Params.length;j+=2)
                    if (Params[j].indexOf(OrderByAnalogs[i])!=-1)
                    {
                        OrderPart=Input.substring(Input.indexOf(Params[j-1]),Input.length());
                        Input=Input.substring(0,Input.indexOf(Params[j-1]));
                        break;
                    }
        if (!OrderPart.equals(""))
        {
            Vector<TableField> Fields= FindFields(Params);
            Vector<String> OnlyFields=new Vector<String>();
            for (int i=0;i<Fields.size();i++)
                OnlyFields.add(Fields.elementAt(i).FieldName);
             OrderClause=FindOrder(OrderPart,OnlyFields,Params);
            String[] Count=OrderPart.split(" ");
            String[] Temp=new String[Params.length-(2*Count.length)];
            for (int i=0;i<Temp.length;i++)
                Temp[i]=Params[i];
            Params=new String[Temp.length];
```

```
            for (int i=0;i<Temp.length;i++)
                Params[i]=Temp[i];
        }
        Vector DBObjects = FindDbObjects(Params);
         Vector<TableField> Fields = FindFields(Params);

    Vector<String> DbFields=new Vector<String>();
        for (int i=0;i<Fields.size();i++)
            DbFields.add(Fields.elementAt(i).FieldName);
        Vector<String> Another = AnotherObjects(Params, DBObjects, Fields);
        for (int i=0;i<Fields.size();i++){
            if (!DBObjects.contains(Fields.elementAt(i).TableName))
                DBObjects.add(Fields.elementAt(i).TableName);
        }


        if (Command.equals("Select") ||
                Command.equals("Update") ||
                Command.equals("Insert") ||
                Command.equals("Delete")){
            if (Command.equals("Select")) {
                Vector<String> MainArgs = FindMainArguments(Input, Params, Another, DbFields,
CheckCommand(Params));

    Vector<FilterParameters> Filters = FindFilters(Input, Params, Another, DbFields, MainArgs);

                Vector<String> FilterToString = new Vector<String>();
                for (int i = 0; i < Filters.size(); i++)
                    FilterToString.add(Filters.elementAt(i).toString());
                  if (DBObjects.size()>1){
                    Result=FromAndWhereWithManyTables(MainArgs,FilterToString,FullPath,DBObjects);
                  }else{
                Result = CombineSelect(MainArgs, (String) DBObjects.elementAt(0), FilterToString);
                  }
                  if (!OrderPart.equals(""))
             Result+=" " + OrderClause;
            }else if (Command.equals("Update")){
                Vector<FilterParameters> UpdateArguments=UpdateFindArguments(Input,Params,DbFields,
                    Another);
                Vector<String> UpdateMain=new Vector<String>();
                for (int i=0;i<UpdateArguments.size();i++)
                    UpdateMain.add(UpdateArguments.elementAt(i).toString());
                Vector<String> UpdateArgsForFilters=new Vector<String>();
                for (int i=0;i<UpdateArguments.size();i++){
                    UpdateArgsForFilters.add(UpdateArguments.elementAt(i).DbObject);
                    if (!isNumeric(UpdateArguments.elementAt(i).Filter))
                        for (int j=0;j<DbFields.size();j++)
                            if (UpdateArguments.elementAt(i).Filter.equals(DbFields.elementAt(j)))
                                UpdateArgsForFilters.add(UpdateArguments.elementAt(i).Filter);
                }
                Vector<FilterParameters> UpdateFilter=FindFilters(Input,Params,
                    Another,DbFields,UpdateArgsForFilters);
                Vector<String> UpdateStringFilter=new Vector<String>();
                for (int i=0;i<UpdateFilter.size();i++)
                    UpdateStringFilter.add(UpdateFilter.elementAt(i).toString());
               Result=CombineUpdate((String)DBObjects.elementAt(0),
                    UpdateMain,UpdateStringFilter);
                                    }

        else if (Command.equals("Delete")) {
                Vector<FilterParameters> Filters = FindFilters(Input, Params, Another, DbFields, new
Vector<String>());
               Vector<String> FilterToString = new Vector<String>();
```

```
            if (Debug)
               for (int i = 0; i < Filters.size(); i++)
                  FilterToString.add(Filters.elementAt(i).toString());

            Result = CombineDelete((String) DBObjects.elementAt(0), FilterToString);
         }
       else if (Command.equals("Insert"))
          {

            Result=CombineInsert((String) DBObjects.elementAt(0),
                  InsertArguments(DbFields,Another));
        }
       }else
       {
          Report("Не корректный тип команды");
       }

 }
 public static class StringArguments{
  private static  String Original;
  private static String FirstForm;
  private static String Pseudonim;
  private static String partOfSpeach;
  private static String PartOfQuery;
  private static int PositionInString;
  boolean IsOriginalOrFirstForm=false;//false - first form, true original
  public  StringArguments(){
     Original="";
     FirstForm="";
     Pseudonim="";
     partOfSpeach="";
     PartOfQuery="";
     PositionInString=-1;

  }
  public  StringArguments(String NewOrigin,String NewFirstForm,
                   String NewPseyd, String NewPartofSpeach,
                   int NewPosition, boolean IsOriginalCorrect){
     Original=NewOrigin;
     FirstForm=NewFirstForm;
     Pseudonim=NewPseyd;
     partOfSpeach=NewPartofSpeach;
     PositionInString=NewPosition;
     IsOriginalOrFirstForm=IsOriginalCorrect;
  }
  public String GetCorrectName(){
     if (Pseudonim!="")
         return Pseudonim;
     if (IsOriginalOrFirstForm)
        return Original;
     return FirstForm;
  }
  public String GetPartOfSpeach(){
     return partOfSpeach;
  }
  public String GetOriginal(){
     return Original;
  }
  public  String GetPartOfQuerty(){
     return PartOfQuery;
  }
  public void SetPartOfQuqery(String NewPartOfQuery){
   PartOfQuery=NewPartOfQuery;
```

```
 }
}

public static Vector FindDbObjects(String[] Params) {

      Vector VTables = new Vector();
   for (int i = 1; i < Params.length; i += 2) {
      if (Params[i].indexOf("=S,", 0) != -1) {
         String TempFind = Params[i].substring(0, Params[i].indexOf("=S,", 0));
         TempFind=TempFind.substring(TempFind.lastIndexOf("|")+1,TempFind.length());
         TempFind=TempFind.replace("?","");
         for (String s : DbTables)
            if (s.equals(TempFind)) VTables.add(TempFind);
      }
            }
   return VTables;

}
public static class CalculatedPath {
    Vector<String> Tables;
    String LabelsToConnect;
    int Weight;
   CalculatedPath(Vector<String> Tabs, String Labels,int W){
      Tables=Tabs;
      LabelsToConnect=Labels;
      Weight=W;
   }
   CalculatedPath()
   {
      Tables=new Vector<String>();
      LabelsToConnect="";
      Weight=0;
   }
}
public static class TableField {
   String TableName;
   String FieldName;

   TableField(String TName, String FName) {
      TableName = TName;
      FieldName = FName;
   }

   TableField() {
      TableName = "";
      FieldName = "";
   }

   public String GetTableName() {
      return TableName;
   }

   public String GetFieldName() {
      return FieldName;
   }

   public boolean equals(TableField TFNew) {
      if (TFNew.FieldName.equals(FieldName) &&
            TFNew.TableName.equals((TableName)))
         return true;
      return false;
   }
}
```

```
public static Vector<TableField> FindFields(String[] Params)throws IOException {
    Vector<TableField> Result = new Vector<TableField>();
    Vector<TableField> DBInfo = DBVector();
   for (int i = 1; i < Params.length; i += 2) {
       if (Params[i].indexOf("=S,", 0) != -1) {
          String TempFind = Params[i].substring(0, Params[i].indexOf("=S,", 0));
          TempFind=TempFind.substring(TempFind.lastIndexOf("|")+1,TempFind.length());
          TempFind=TempFind.replace("?","");
          for (int j = 0; j < DBInfo.size(); j++)
             if (TempFind.equals(DBInfo.elementAt(j).FieldName))
                Result.add(DBInfo.elementAt(j));
       }
    }
    CheckFields(Result);
    return Result;
}

public static Vector<TableField> CheckFields(Vector<TableField> InVect) {
    boolean Check = false;
   for (int i = 0; i < InVect.size() - 1; i++) {
       int Count = 0;
       for (int j = i + 1; j < InVect.size(); j++) {
          if (InVect.elementAt(i).equals(InVect.elementAt(j)))
             Count++;
       }
       if (Count > 1) Check = true;
    }
    if (Check) {
       Vector<String> Tables = new Vector<String>();
       Vector<Integer> CountOfTables = new Vector<Integer>();
       for (int i = 0; i < InVect.size(); i++) {
          if (Tables.indexOf(InVect.elementAt(i).TableName) != -1) {
             Tables.add(InVect.elementAt(i).TableName);
             CountOfTables.insertElementAt(1,
                   Tables.indexOf(InVect.elementAt(i).TableName));
          } else
             CountOfTables.set(Tables.indexOf(InVect.elementAt(i).TableName),
                   CountOfTables.elementAt(Tables.indexOf(InVect.elementAt(i).TableName)));
       }
       int LoopCount = Tables.size() - 1;
       while (LoopCount > 0) {
          int Max = -1;
          int MaxTable = -1;
          for (int i = 0; i < Tables.size(); i++)
             if (CountOfTables.elementAt(i) < Max)
                MaxTable = i;
          String MaxTableName = Tables.elementAt(Max);
          Vector<String> MaxTableFields = new Vector<String>();
          for (int i = 0; i < InVect.size(); i++)
             if (InVect.elementAt(i).TableName.equals(MaxTableName))
                MaxTableFields.add(InVect.elementAt(i).FieldName);
          for (int i = 0; i < InVect.size(); i++)
             for (int j = 0; j < MaxTableFields.size(); j++)
                if (InVect.elementAt(i).FieldName.equals(MaxTableFields.elementAt(j))
                      && !(InVect.elementAt(i).TableName.equals(MaxTableName)))
                   InVect.remove(i);
          LoopCount -= 1;
       }

    }
    return InVect;
```

```
}
public static String CheckCommand(String[] Params) {
     for (int i = 1; i < Params.length; i += 2) {
    if (Params[i].indexOf("=V,", 0) != -1) {
        String TempFind = Params[i].substring(0, Params[i].indexOf("=V,", 0));
        TempFind=TempFind.substring(TempFind.lastIndexOf("|")+1,TempFind.length());
        TempFind=TempFind.replace("?","");
       for (String s : UpdateAnalogs)
            if (s.indexOf(TempFind)!=-1) return "Update";
        for (String s : SelectAnalogs)
            if (s.indexOf(TempFind)!=-1) return "Select";
        for (String s : DeleteAnalogs)
            if (s.indexOf(TempFind)!=-1) return "Delete";
        for (String s : InsertAnalogs)
            if (s.indexOf(TempFind)!=-1) return "Insert";
    }
  }
  return "";
}

public static Vector<String> AnotherObjects(String[] Params, Vector DbObjects, Vector<TableField> Fields) {
  Vector<String> Result = new Vector<String>();
  for (int i = 1; i < Params.length; i += 2) {
    boolean NeedsToBeChecked = true;
    String TempFind = "";
    if (Params[i].indexOf("=S,", 0) != -1){
         TempFind=Params[i].substring(0, Params[i].indexOf("=S,", 0));
        TempFind=TempFind.substring(TempFind.lastIndexOf("|")+1,TempFind.length());
        TempFind=TempFind.replace("?","");}
    else if (Params[i].indexOf("=ADV,", 0) != -1){
        TempFind = Params[i].substring(0, Params[i].indexOf("=ADV,", 0));
        TempFind=TempFind.substring(TempFind.lastIndexOf("|")+1,TempFind.length());
        TempFind=TempFind.replace("?","");
    }
    else if (Params[i].indexOf("=A,", 0) != -1){
        TempFind = Params[i].substring(0, Params[i].indexOf("=A,", 0));
        TempFind=TempFind.substring(TempFind.lastIndexOf("|")+1,TempFind.length());
        TempFind=TempFind.replace("?","");
    }
    else if (Params[i].indexOf("=NUM,", 0) != -1)
            {
            TempFind = Params[i].substring(0, Params[i].indexOf("=NUM,", 0));
            TempFind=TempFind.substring(TempFind.lastIndexOf("|")+1,TempFind.length());
            TempFind=TempFind.replace("?","");

        }          else {
        NeedsToBeChecked = false;
    }

    if (NeedsToBeChecked) {
        for (int j = 0; j < DbObjects.size(); j++) {
            if (TempFind.equals(DbObjects.elementAt((j)))) {
                NeedsToBeChecked = false;
                break;
            }
        }
    }
    if (NeedsToBeChecked) {
        for (int k = 0; k < Fields.size(); k++) {
            if (TempFind.equals(Fields.elementAt(k).FieldName)) {
                NeedsToBeChecked = false;
                break;
            }
```

```
            }
        }
        if (NeedsToBeChecked && TempFind.indexOf("|", 0) == -1)
            Result.add(TempFind);
    }
    return Result;
}

public static Vector<String> FindMainArguments(String MainString,
                                          String[] Params,
                                          Vector<String> AnotherObjs,
                                          Vector<String> FilterObjects,
                                          String MainCommad) {
    Vector<String> Result = new Vector<String>();
   for (int i = 0; i < AnotherObjs.size(); i++)
        for (int j = 0; j < StarAnalogs.length; j++)
            if (AnotherObjs.elementAt(i).equals(StarAnalogs[j])) {
                Result.add("*");
                return Result;
            }
    int CommandPosition = -1;
    switch (MainCommad) {
        case "Select":
            for (int i = 1; i < Params.length; i += 2) {
                if (Params[i].indexOf("=V,", 0) != -1) {
                    String TempFind = Params[i].substring(0, Params[i].indexOf("=V,", 0));
                    TempFind=TempFind.substring(TempFind.lastIndexOf("|")+1,TempFind.length());
                    TempFind=TempFind.replace("?","");
                    for (String s : SelectAnalogs)
                        if (s.equals(TempFind)) CommandPosition = MainString.indexOf(Params[i - 1]);
                }
            }
            break;
        case "Update":
            for (int i = 1; i < Params.length; i += 2) {
                if (Params[i].indexOf("=V,", 0) != -1) {
                    String TempFind = Params[i].substring(0, Params[i].indexOf("=V,", 0));
                    TempFind=TempFind.substring(TempFind.lastIndexOf("|")+1,TempFind.length());
                    TempFind=TempFind.replace("?","");
                    for (String s : UpdateAnalogs)
                        if (s.equals(TempFind)) CommandPosition = MainString.indexOf(Params[i - 1]);
                }
            }
            break;
       case "Insert":
            for (int i = 1; i < Params.length; i += 2) {
                if (Params[i].indexOf("=V,", 0) != -1) {
                    String TempFind = Params[i].substring(0, Params[i].indexOf("=V,", 0));
                    for (String s : InsertAnalogs)
                        if (s.equals(TempFind)) CommandPosition = MainString.indexOf(Params[i - 1]);
                }
            }
            break;
        default:
Report("Не корректный тип комманды");
            break;
    }
    int FilterPosition = -1;
    boolean Stop=true;
    int StartToFilterObjs=0;
    String Filter = "";
    while(Stop) {
        int Distance = 99999;
```

```
        FilterPosition=-1;
        for (int i = StartToFilterObjs; i < FilterObjects.size(); i++) {
            for (int j = 1; j < Params.length; j += 2)
                if (Params[j].indexOf(FilterObjects.elementAt(i)) != -1) {
                    if (Math.abs(MainString.indexOf(Params[j - 1]) - CommandPosition) < Distance) {
                        FilterPosition = MainString.indexOf(Params[j - 1]);
                        Distance = Math.abs(FilterPosition - CommandPosition);
                        Filter = FilterObjects.elementAt(i);
                        StartToFilterObjs=i+1;
                    }
                }
        }
        Stop=false;
        Result.add(Filter);
        CommandPosition = FilterPosition + Filter.length();
        if (MainString.length()>CommandPosition){
        String temp=MainString.substring(CommandPosition , CommandPosition + 1);
        if (MainString.substring(CommandPosition , CommandPosition + 1).equals(","))
            Stop = true;}
    }
    try{
    if (MainString.replace(" ", "").substring(FilterPosition - CommandPosition + 1) != "," &&
            MainString.replace(" ", "").substring(FilterPosition + CommandPosition + 1) != ",") {
        return Result;
    }else{
        Vector<String> AlsoArguments= FindMany(MainString,Params,FilterPosition,Filter,true);
        for (int i=0;i<AlsoArguments.size();i++)
            Result.add(AlsoArguments.elementAt(i));
        return Result;
    }}
    catch (Exception e)
    {
        Report("");
    }
   return Result;
}

public static Vector<String> FindMany(String MainString,
                            String[] Params,
                            int ArgumentPosition,
                            String Argument,
                            boolean Forward){
    Vector<String> Result=new Vector<String>();
    return Result;

}
    public  static String FindClosestToFilter(String MainString,
                                          String[] Params,
                                          Vector<String> AnotherObjs,
                                          String Field)
    {
        String Result="";
        int FieldPosition=-1;
        for (int i=1;i<Params.length;i+=2)
            if (Params[i].indexOf(Field)!=-1)
                FieldPosition=MainString.indexOf(Params[i-1]);
        int Distance=999999;
        for (int i=0;i<AnotherObjs.size();i++)
            for(int j=1;j<Params.length;j+=2)
            {
                if (Params[j].indexOf(AnotherObjs.elementAt(i))!=-1)
                    if(Math.abs(FieldPosition-MainString.indexOf(Params[j-1]))
                            <Distance){
```

```
                Distance=Math.abs(FieldPosition-MainString.indexOf(Params[j-1]));
                Result=AnotherObjs.elementAt(i);
              }
         }
      return Result;
    }
    public static class FilterParameters{
    public String DbObject;
    public String Operand;
    public String Filter;
    public FilterParameters(String DbObj, String Oper, String Filt)
    {
      DbObject=DbObj;
      Operand=Oper;
      Filter=Filt;
    }
    @Override public  String toString() {
      return " "+DbObject+Operand+Filter+" ";
    }

  }
    public  static Vector<FilterParameters> FindFilters(String MainString,
                                          String[] Params,
                                          Vector<String> AnotherObjects,
                                          Vector<String> DbObjects,
                                          Vector<String> MainObjects){
    Vector<FilterParameters> Result=new  Vector<FilterParameters>();
       if (MainObjects.size()>0) {

      for (int i = 0; i < DbObjects.size(); i++){
         boolean NeedToProcessArg = true;
         for (int j = 0; j < MainObjects.size(); j++){
            if ((DbObjects.elementAt(i).equals(MainObjects.elementAt(j))))
            { NeedToProcessArg=false;
            break;
            }}
            if (NeedToProcessArg){
            Result.add(FindCombinationOfFilters(
                   MainString, Params,
                   DbObjects.elementAt(i), AnotherObjects,DbObjects));}

         }}else{
      for (int i=0;i<DbObjects.size();i++)
                   Result.add(FindCombinationOfFilters(
                      MainString,Params,
                      DbObjects.elementAt(i),AnotherObjects,DbObjects));

            }
    return Result;
    }

    public  static FilterParameters FindCombinationOfFilters(String MainString,
                                            String[] Params,
                                            String DbField,
                                            Vector<String> AnotherObjects,
                                            Vector<String> DbObjects)
    {
      FilterParameters Result=new FilterParameters(DbField,"","");
     String DbFieldInStringName="";
      String[] MainStringElements= MainString.split(" ");
      int DbFileElementPosition=-1;
      int DbFieldPosition=-1;
      for (int i=1;i<Params.length;i+=2)
```

```
    if (Params[i].indexOf(DbField)!=-1){
        DbFieldInStringName=Params[i-1];
        DbFieldPosition=MainString.indexOf(DbFieldInStringName);
        for (int j=0;j<MainStringElements.length;j++)
            if (MainStringElements[j].equals(DbFieldInStringName)){
                DbFileElementPosition=j;
                break;
                }
        break;
    }
        boolean NeedToFindOperand=true;
Result.Filter=CheckSecondInLine(Params,
        MainStringElements[DbFileElementPosition+1],
        AnotherObjects);
if (Result.Filter.equals(""))
{
    NeedToFindOperand=true;
}
else
{
    NeedToFindOperand=false;
    Result.Operand="=";
}
if (NeedToFindOperand)
    switch (MainStringElements[DbFileElementPosition+1]) {
    case "=":
        Result.Filter=CheckSecondInLine(Params,
                MainStringElements[DbFileElementPosition+2],
                AnotherObjects);
        if (!Result.Filter.equals("")){
            Result.Operand="=";
            NeedToFindOperand=false;}
        break;
    case "<":
        Result.Filter=CheckSecondInLine(Params,
                MainStringElements[DbFileElementPosition+2],
                AnotherObjects);
        if (!Result.Filter.equals("")){
            Result.Operand="<";
            NeedToFindOperand=false;}
        break;
    case ">":
        Result.Filter=CheckSecondInLine(Params,
                MainStringElements[DbFileElementPosition+2],
                AnotherObjects);
        if (!Result.Filter.equals("")){
            Result.Operand=">";
            NeedToFindOperand=false;}
        break;
    case "!=":
        Result.Filter=CheckSecondInLine(Params,
                MainStringElements[DbFileElementPosition+2],
                AnotherObjects);
        if (!Result.Filter.equals("")){
            Result.Operand="!=";
            NeedToFindOperand=false;}
        break;
    default:
        for (int i=0;i<TextEqual.length;i++){
            if (TextEqual[i].equals(MainStringElements[DbFileElementPosition+1])) {
                Result.Filter=CheckSecondInLine(Params,
                        MainStringElements[DbFileElementPosition+2],
                        AnotherObjects);
```

```
                if (!Result.Filter.equals("")){
                    Result.Operand="=";
                    NeedToFindOperand=false;}
            }}
        for (int i=0;i<TextBigger.length;i++){
            if (TextBigger[i].equals(MainStringElements[DbFileElementPosition+1])) {
                Result.Filter=CheckSecondInLine(Params,
                        MainStringElements[DbFileElementPosition+2],
                        AnotherObjects);
                if (!Result.Filter.equals("")){
                    Result.Operand=">";
                    NeedToFindOperand=false;}
            }}
        for (int i=0;i<TextLower.length;i++){
            if (TextLower[i].equals(MainStringElements[DbFileElementPosition+1])) {
                Result.Filter=CheckSecondInLine(Params,
                        MainStringElements[DbFileElementPosition+2],
                        AnotherObjects);
                if (!Result.Filter.equals("")){
                    Result.Operand="<";
                    NeedToFindOperand=false;}
            }}
                    break;
    }
     if (NeedToFindOperand==false){
      if (!isNumeric(Result.Filter))
        Result.Filter="'"+Result.Filter+"'";
      String ElementInAnotherObjs="";
      for (int i=0;i<Params.length;i+=2)
        if (Params[i].indexOf(Result.Filter)!=-1)
        {
            ElementInAnotherObjs=Params[i+1].substring(0,Params[i+1].indexOf("="));
            break;
        }
      for (int i=0;i<AnotherObjects.size();i++)
        if (AnotherObjects.elementAt(i).equals(ElementInAnotherObjs))
        {
            AnotherObjects.remove(i);
            break;
        }

    }
    return Result;
}

public static String CheckSecondInLine(String[] Params,
                                String NeedElement,
                                Vector<String> AnotherObjects)
{
    if (isNumeric(NeedElement)) return NeedElement;
    String BaseElement="";
    for (int i=0;i<Params.length;i+=2)
      if (Params[i].indexOf(NeedElement)!=-1)
      {
        BaseElement=Params[i+1].substring(0,Params[i+1].indexOf("="));
        break;
      }
    for (int i=0;i<AnotherObjects.size();i++)
      if (AnotherObjects.elementAt(i).equals(BaseElement)){
        return BaseElement;
      }
    return "";
} public static Vector<FilterParameters> UpdateFindArguments(String MainString,
```

```
                                String[] Params,
                                Vector<String> DbObjects,
                                Vector<String> AnotherObjects)
{
   Vector<FilterParameters> UpdateArguments=new Vector<FilterParameters>();
   String[] MainSeparate=MainStringCorrect.split(" ");
   Vector<FilterParameters> Result=new Vector<FilterParameters>();
   String[] Ussrs={"и"};
   String[] UpScale={"увеличить","увеличивать"};
   String[] DownScale={"уменьшить","уменьшать"};
   int Up=0;
   for (int i=1;i<Params.length;i++)
   {
      for (int j=0;j<UpScale.length;j++)
      if (Params[i].indexOf(UpScale[j])!=-1)
         Up=-1;
      for (int j=0;j<DownScale.length;j++)
         if (Params[i].indexOf(DownScale[j])!=-1)
            Up=1;
   }
   int i=0;
   boolean Continue=true;
   Vector<String> DbObjInString=DbObjectsOriginal(Params,DbObjects);
   do{
      for (int j=0;j<DbObjInString.size();j++)
         if (MainSeparate[i].equals(DbObjInString.elementAt(j)))
         {
            FilterParameters Temp=UpdateFindParams(MainSeparate,Params,DbObjects,AnotherObjects,
                  i,Up,DbObjects.elementAt(j));
            Continue=false;
            }
            for (int k=0;k<DbObjInString.size();k++)
               if(Temp.Filter.indexOf(DbObjInString.elementAt(k))!=-1){
                  Temp.Filter=Temp.Filter.replace(DbObjInString.elementAt(k),
                        DbObjects.elementAt(k));
                  break;
               }
            Result.add(Temp);
            if ((i+3)<MainSeparate.length)
            {
               if (MainSeparate[i].equals(",")){
                  i+=3;
               }
               else{
                  for (int k=0;k<Ussrs.length;k++)
                     if (MainSeparate[i+3].equals(Ussrs[k])){
                        i+=3;
                        break;
                     }}
            }
        }
      i+=1;
      if (!Continue)
         break;
   }while(i<MainSeparate.length);
   return Result;
         }
public static Vector<String> DbObjectsOriginal(String[] Params, Vector<String> DbObjects)
{
   Vector<String> Result=new Vector<String>();
   for (int i=0;i<DbObjects.size();i++)
      for (int j=1;j<Params.length;j+=2)
         if (Params[j].indexOf(DbObjects.elementAt(i))!=-1)
```

```
            Result.add(Params[j-1]);

    return Result;
}
public static FilterParameters UpdateFindParams(String[] MainStringSeparate,
                    String[] Params,
                    Vector<String> DbObjects,
                    Vector<String> AnotherObjects,
                    int NumInMainString,
                              int Up,
                              String LeftArgument)
{
    String[] TextEqual={"равен","с"};
    String[] PretextSum={"на"};
    String[] PretextScale={"в"};
    FilterParameters Result= new FilterParameters(LeftArgument,
            "="," ");
    try {
        if (MainStringSeparate[NumInMainString + 1].equals("=")) {
            Result.Filter = MainStringSeparate[NumInMainString + 2];
            return  Result;
        }
        for (int i = 0; i < Params.length; i += 2)
            if (Params[i].equals(MainStringSeparate[NumInMainString + 1])) {
                //Равенство
                for (int j = 0; j < TextEqual.length; j++)
                    if (TextEqual[j].equals(Params[i + 1].substring(0, Params[i + 1].indexOf("=")))) {
                        Result.Filter = MainStringSeparate[NumInMainString + 2];
                        return Result;
                    }
                for (int j = 0; j < PretextSum.length; j++){
                    if (PretextSum[j].equals(Params[i + 1]
                            .substring(0, Params[i + 1].indexOf("=")))){
                        if (Up==1)
                        {
                         Result.Filter=MainStringSeparate[NumInMainString]+" + " +
                                MainStringSeparate[NumInMainString+2];
                        }else if (Up==-1)
                        {
                            Result.Filter=MainStringSeparate[NumInMainString]+" - " +
                                    MainStringSeparate[NumInMainString+2];
                        }else{
                            Result.Filter=MainStringSeparate[NumInMainString+2];
                        }

                    return Result;
                    }

                }
                for (int j = 0; j < PretextScale.length; j++){
                    if (PretextScale[j].equals(Params[i + 1]
                            .substring(0, Params[i + 1].indexOf("=")))) {
                        if (Up==1) {
                            Result.Filter = MainStringSeparate[NumInMainString] + " * " +
                                    MainStringSeparate[NumInMainString + 2];
                        } else if (Up==-1) {
                            Result.Filter = MainStringSeparate[NumInMainString] + " / " +
                                    MainStringSeparate[NumInMainString + 2];
                        }
                        else{
                            Result.Filter=MainStringSeparate[NumInMainString+2];
                        }
                    return Result;
```

```
                }
              }

            }
    return  Result;

    } catch (ArrayIndexOutOfBoundsException ex)
    {
       return Result;
    }
}
public  static Vector<FilterParameters> InsertArguments(Vector<String> DbObjects,
                                            Vector<String> AnotherObjects)
{
 Vector<FilterParameters> Result=new Vector<FilterParameters>();
 if (DbObjects.size()!=AnotherObjects.size())
    throw new IndexOutOfBoundsException();
 for (int i=0;i<DbObjects.size();i++)
    Result.add(new FilterParameters(DbObjects.elementAt(i),"=",
          AnotherObjects.elementAt(i)));
 return Result;
}
public static String FindOrder(String PartOfMainString,Vector<String> DbObjs,String Params[] )
{
    String Result="ORDER BY ";
    String[] Temp=PartOfMainString.split(" ");
    for (int i=0;i<Temp.length;i++)
       for (int j=0;j<Params.length;j+=2)
          if (Params[j].indexOf(Temp[i])!=-1)
             for (int k=0;k<DbObjs.size();k++)
                if (Params[j+1].indexOf(DbObjs.elementAt(k))!=-1)
                   Result+=" "+DbObjs.elementAt(k)+ " ,";
    Result=Result.substring(0,Result.length()-2);
    return Result;

}
public  static String CombineSelect(Vector<String> MainObjects,
                              String Tables,
                              Vector<String> Filters)
{
    String Result="Select ";
    for (int i=0;i<MainObjects.size();i++)
       Result+=MainObjects.elementAt(i) +" , ";
    Result=Result.substring(0,Result.length()-2);
    Result+=" From " + Tables + " ";
    if (Filters.size()>0){
       Result+=" Where ";
       for (int i=0;i<Filters.size();i++)
       {
          Result+= Filters.elementAt(i) + " and ";
       }
       Result=Result.substring(0,Result.length()-4);
    }
    return Result;
}
public  static String CombineDelete(String Tables,
                              Vector<String> Filters)
{
    String Result=" Delete From ";
    Result+= Tables + " ";
    if (Filters.size()>0)
    {
       Result+= "WHERE ";
```

```
        for (int i=0;i<Filters.size();i++)
            Result+=Filters.elementAt(i) + " ";
        }
        return Result;
    }
    public  static String CombineUpdate(String Tables,
                                Vector<String> FieldsAndValues,
                                Vector<String> Filters)
    {
        String Result=" Update ";
        Result+= Tables + " ";

            Result+=" Set ";
            for (int i=0;i<FieldsAndValues.size();i++)
                Result+= FieldsAndValues.elementAt(i) +" , ";
            Result=Result.substring(0,Result.length()-2);

        if (Filters.size()>0){
            Result+= " WHERE ";
            for (int i=0;i< Filters.size();i++)
                Result+=" " + Filters.elementAt(i)+ " ";
        }
            Result+=" ;";
        return Result;
    }
    public static String CombineInsert(String Table,
                                Vector<FilterParameters> Arguments)
    {
        String Result="";
        Result+= "Insert  into ";
        Result+= Table+ " ";
        for (int i=0;i<Arguments.size();i++)
            Result+=Arguments.elementAt(i).DbObject+ " , ";
        Result=Result.substring(0,Result.length()-2);
        Result+= " VALUES ";
        for (int i=0;i<Arguments.size();i++)
            Result+=Arguments.elementAt(i).Filter+ " , ";
        Result=Result.substring(0,Result.length()-2);
        return Result;

    }

public  static Vector<String> ExecSelect(ResultSet Rs,
                                Statement St,
                                String Query,
                                Vector<String> OutFields)
{
    Vector<String> Result=new Vector<String>();
    try {
        Rs=St.executeQuery(Query);
        while (Rs.next())
        {
            String TempString="";
            for (int i=0;i<OutFields.size();i++)
            {
                TempString+=Rs.getString(OutFields.elementAt(i));
            }
            Result.add(TempString);
        }
    }
    catch (Exception e)
    {
Report(e.getMessage());
```

```
    }
    return Result;
  }
  public static String ExecuteUpdate(Statement St,
                                  String Query) {
    String Result = "";
    try {
      int CountOfUpdates=St.executeUpdate(Query);
      Result="Кол-во обновленных строк = " + CountOfUpdates;

    } catch (Exception e)
    {
      Report (e.getMessage());
    }
    return Result;

  }
  public static String FromAndWhereWithManyTables(Vector<String> MainObjects,
                                          Vector<String> Filters,
                                        CalculatedPath[] FullPath,
                                        Vector<String> Tables){
    Vector<String> ResultTables=new Vector<String>();
    Vector<String> ResultWhere=new Vector<String>();
    ResultTables.add(Tables.elementAt(0));
    for (int i=1;i<Tables.size();i++) {
      int MinWeight=9999;
      int BestPosition=0;
      for (int k = 0; k < ResultTables.size(); k++){
        for (int j = 0; j < FullPath.length; j++){
          if (FullPath[j].Tables.elementAt(0).equals(ResultTables.elementAt(k)) &&
          FullPath[j].Tables.elementAt(FullPath[j].Tables.size()-1).equals(Tables.elementAt(i)))
            if (FullPath[j].Weight<MinWeight){
              BestPosition=j;
              MinWeight=FullPath[j].Weight;
              }

    }}
      ResultTables.add(Tables.elementAt(i));
      for (int p=1;p<FullPath[BestPosition].Tables.size()-1;p++){
        ResultTables.add(FullPath[BestPosition].Tables.elementAt(p));
      }
      ResultWhere.add(FullPath[BestPosition].LabelsToConnect);

    }
    for (int i=0;i<ResultTables.size();i++)
      for (int j=0;j<ResultTables.size();j++){
        if (i!=j)
          if (ResultTables.elementAt(i).equals(ResultTables.elementAt(j)))
            ResultTables.remove(j);
      }

    String Result="Select ";
    for (int i=0;i<MainObjects.size();i++)
      Result+=MainObjects.elementAt(i) +" ,";
    Result=Result.substring(0,Result.length()-2);
    Result+=" From ";
    for (int i=0;i<ResultTables.size();i++)
      Result+=" "+ ResultTables.elementAt(i)+ " , ";
    Result=Result.substring(0,Result.length()-2);
    Result+=" Where ";
    if (Filters.size()>0){

      for (int i=0;i<Filters.size();i++)
```

```
        {
          Result+= Filters.elementAt(i) + " and ";
        }
      }
      for (int i=0;i<ResultWhere.size();i++)
        Result+=" "+ ResultWhere.elementAt(i)+ " and ";
      Result=Result.substring(0,Result.length()-4);
      return Result;

  }
 }
 public  static Vector<Test.TableField> LoadDbFields()
        throws  IOException
  {
     String data = "";
      data=new String(Files.readAllBytes(Paths.get(Folder+"DbObjs.txt")));
      String[] Result=data.split("\n");
      Vector<Test.TableField> ResultVec=new Vector<Test.TableField>();
      for (int i=0;i<Result.length;i++){
      String[] Temp=Result[i].split(" ");
      Test.TableField Field=new Test.TableField();
      for (int j=1;j<Temp.length;j++)
        ResultVec.add(new Test.TableField(Temp[0],Temp[j]));
      }
      return ResultVec;

  }
     public static Test.CalculatedPath[] LoadCalculatedPath()
  throws  IOException{
      String Folder="//Libr/";
      String data = "";
      data=new String(Files.readAllBytes(Paths.get(Folder+"Path.txt")));
      String[] Temp=data.split("\n");
      Test.CalculatedPath[] Result=new Test.CalculatedPath[Temp.length];
      for (int i=0;i<Temp.length;i++){
        Result[i]=new Test.CalculatedPath();
       Vector<String> TempVector=new Vector<String>();
        String[] TempTemp=Temp[i].split(";");
        Result[i].Tables=new Vector<String>();
        for (String s:TempTemp[0].split(" ")
             ) {
          Result[i].Tables.add(s);
          //TempVector.add(s);
        }

        Result[i].LabelsToConnect=TempTemp[1];
        Result[i].Weight= Integer.parseInt(TempTemp[2].trim());
      }
      return Result;

  }

}
```