



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

ФАКУЛЬТЕТ \_\_\_\_\_ «Информатика и системы управления»

КАФЕДРА \_\_\_\_\_ «Теоретическая информатика и компьютерные технологии»

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**  
***К КУРСОВОЙ РАБОТЕ***  
***НА ТЕМУ:***

***«Оценка производительности СУБД Dgraph***  
***при выполнении графовых запросов»***

Студент \_\_\_\_\_  
(Группа)

\_\_\_\_\_  
(Подпись, дата)

\_\_\_\_\_  
(И.О. Фамилия)

Руководитель

\_\_\_\_\_  
(Подпись, дата)

\_\_\_\_\_  
(И.О. Фамилия)

2024 г.

# СОДЕРЖАНИЕ

ВВЕДЕНИЕ . . . . .	4
1 Обзор СУБД Dgraph . . . . .	5
1.1 Графовая модель данных . . . . .	5
1.2 Архитектура . . . . .	5
1.3 Поддержка GraphQL . . . . .	6
1.4 Поддержка DQL . . . . .	6
1.5 Схема DQL . . . . .	7
1.6 DQL и графовые алгоритмы . . . . .	8
1.7 Формат RDF . . . . .	8
1.8 Формат JSON . . . . .	9
1.9 Загрузка данных . . . . .	9
2 Формирование запросов и тестовых данных . . . . .	11
2.1 Набор данных Elliptic++ Transactions . . . . .	11
2.1.1 Описание и структура . . . . .	11
2.1.2 Интерпретация и схема . . . . .	12
2.1.3 Запросы . . . . .	12
2.2 Набор данных MOOC User Actions . . . . .	15
2.2.1 Описание и структура . . . . .	15
2.2.2 Интерпретация и схема . . . . .	16
2.2.3 Запросы . . . . .	17
2.3 Набор данных California Road Network . . . . .	21
2.3.1 Описание и структура . . . . .	21
2.3.2 Интерпретация и схема . . . . .	21
2.3.3 Запросы . . . . .	21
2.4 Набор данных Stablecoin ERC20 Transactions . . . . .	24
2.4.1 Описание и структура . . . . .	24
2.4.2 Интерпретация и схема . . . . .	24
2.4.3 Запросы . . . . .	25
3 Загрузка данных и выполнение запросов . . . . .	29
3.1 Развёртывание кластера Dgraph . . . . .	29

3.2	Преобразование исходных данных . . . . .	29
3.3	Проектирование преобразователя данных . . . . .	30
3.3.1	Формат входных и выходных данных . . . . .	30
3.3.2	Спецификация конфигурационного файла . . . . .	31
3.3.3	Конфигурационные файлы тестовых наборов данных . . . . .	34
3.3.4	Модули преобразователя . . . . .	34
3.3.5	Модуль работы с RDF . . . . .	34
3.3.6	Модуль преобразования . . . . .	35
3.4	Реализация преобразователя данных . . . . .	37
3.4.1	Модуль работы с RDF . . . . .	37
3.4.2	Модуль преобразования . . . . .	38
3.4.3	Использование преобразователя . . . . .	39
3.5	Выполнение запросов . . . . .	39
3.5.1	Реализация . . . . .	39
3.5.2	Использование . . . . .	41
4	Оценка производительности . . . . .	<b>42</b>
4.1	Статистика тестовых наборов данных . . . . .	42
4.1.1	Вершины и рёбра . . . . .	42
4.1.2	Дискковое пространство . . . . .	42
4.2	Выполнение запросов . . . . .	43
4.2.1	Время выполнения . . . . .	43
4.2.2	Оперативная память . . . . .	44
	<b>ЗАКЛЮЧЕНИЕ . . . . .</b>	<b>46</b>
	<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ . . . . .</b>	<b>47</b>
	<b>ПРИЛОЖЕНИЕ А . . . . .</b>	<b>48</b>

# ВВЕДЕНИЕ

Базы данных являются неотъемлемой составляющей многих современных программных продуктов. Существует множество разновидностей баз данных, различающихся способами хранения и обработки информации. В последнее время большое распространение получают NoSQL-решения, среди которых особенно выделяются графовые базы данных. Они хранят информацию в графовой модели, а это означает, что выборка данных относительно их связей с другими данными является базовой операцией, выполняющейся эффективнее, чем в, например, реляционных аналогах. Графовые базы данных успешно применяются при решении различных задач: генерация рекомендаций реального времени, управление идентификацией и доступом, выявление мошенничества и многое другое.

Сегодня существует множество графовых СУБД. Среди наиболее распространённых — Neo4j, ArangoDB, Amazon Neptune, Azure Cosmos DB и другие. Они различаются поддерживаемыми моделями данных, реализацией графовой модели, направленностью на аналитическую обработку или обработку транзакций, возможностями масштабирования. Одной из таких графовых СУБД является Dgraph — проект с открытым исходным кодом, ориентированный на распределённое использование, масштабирование, нативно поддерживающий технологию GraphQL и OLTP-направленный. СУБД Dgraph развивается, и становится актуальным вопрос оценки её производительности для возможности сравнения с аналогичными графовыми СУБД. Важнейший показатель такой оценки — эффективность выполнения графовых запросов.

Таким образом, целью курсового проекта является оценка производительности СУБД Dgraph при выполнении графовых запросов. Курсовой проект выполняется в рамках сравнения производительности графовых СУБД Neo4j, ArangoDB и Dgraph, поэтому результаты оценки также должны быть сравнимы. Для этого, в частности, используются единые, определённые предварительно, наборы данных и графовых запросов. Также результаты оценки должны быть легко воспроизводимы, для чего важно автоматизировать основные этапы работы: загрузку исходных данных и выполнение графовых запросов. При этом необходимо минимизировать дублирование в итоговом программном коде.

# 1 Обзор СУБД Dgraph

## 1.1 Графовая модель данных

Dgraph [1] является *нативной* графовой СУБД, т.е. графовая модель данных в Dgraph — единственная<sup>1</sup>. Современные графовые СУБД поддерживают, как правило, одну из двух следующих реализаций графовой модели:

- *граф свойств* (property graph), хранящий вершины (сущности), рёбра (связи между сущностями) и свойства, описывающие вершины или рёбра. Именно эта модель лежит в основе Dgraph;
- *RDF-граф*, использующий модель субъект-предикат-объект для хранения информации. Dgraph поддерживает формат RDF для импорта и экспорта данных.

Связи между сущностями в Dgraph направлены, что позволяет оптимально совершать обход от одной сущности к другой. Для двунаправленной связи нужно, что естественно, установить связь между сущностями в обе стороны.

## 1.2 Архитектура

Dgraph эффективно масштабируется для работы с большими объёмами данных, поскольку проектируется с самого начала для распределённого использования. В основе его работы — кластер взаимодействующих друг с другом серверных узлов, образующих единое логическое хранилище данных. В Dgraph выделяется два типа серверных узлов:

- *Zero-узлы* содержат метаданные кластера Dgraph, координируют распределённые транзакции и распределяют данные между группами серверов;
- *Alpha-узлы* хранят данные графа и индексы. Важно отметить, что Alpha-узлы хранят и индексируют *предикаты*, представляющие связи между данными. Такой подход позволяет Dgraph выполнять запросы глубины  $N$  к базе данных ровно за  $N$  сетевых переходов.

Каждый кластер Dgraph должен иметь как минимум один Zero- и один Alpha-узел.

---

<sup>1</sup>Существуют также мультимодельные СУБД. Например, в ArangoDB помимо графовой поддерживаются документо-ориентированная модель и модель «ключ-значение».

## 1.3 Поддержка GraphQL

В Dgraph встроена поддержка технологии GraphQL [2]. GraphQL — язык запросов и манипулирования данными для API, а также серверная среда выполнения запросов с открытым исходным кодом. GraphQL позволяет клиенту в декларативной форме описывать точные данные, требуемые от API. Вместо нескольких конечных точек (endpoints), возвращающих отдельные данные, сервер GraphQL предоставляет единственную, возвращающую ровно ту информацию, которую запрашивает клиент. Сервер GraphQL может извлекать данные из различных источников, представляя результат в виде единого графа, то есть GraphQL не привязан к какой-либо базе данных или иному механизму хранения информации.

Для запуска сервиса GraphQL необходимо определить *схему* — типы данных и их поля, и для каждого такого поля — его *разрешающую функцию* (resolver), вычисляющую значение поля. Принимая запрос, GraphQL его валидирует соответственно определённой схеме, и в случае успеха выполняет, вызывая соответствующие полям запроса разрешающие функции. Сервер возвращает данные, в точности отвечающие исходному запросу.

Нативная поддержка GraphQL выделяет Dgraph на фоне его аналогов. Пользователю достаточно описать схему GraphQL, и GraphQL API будет готов к работе. Dgraph автоматически определит разрешающие функции: в основе их реализации — простое следование по связям в графе от вершины к вершине и от вершины к полю, что и обеспечивает нативную графовую производительность.

## 1.4 Поддержка DQL

На практике часто возникает необходимость в выполнении сложных запросов, не поддерживаемых спецификацией GraphQL. Для этого разработан Dgraph Query Language (DQL) — проприетарный язык запросов Dgraph, синтаксически напоминающий GraphQL, но обладающий большей выразительной силой. Кажется, что при наличии DQL нет необходимости в поддержке GraphQL. На деле же языки различаются своим назначением [3] и оба могут обоснованно использоваться в рамках одного проекта.

Сервис GraphQL предоставляет API для контролируемого и защищённого доступа внешних клиентов к базе данных. Dgraph поддерживает набор директив

для применения в схеме GraphQL, посредством которых выражаются правила авторизации доступа к данным и иные ограничения бизнес-логики.

DQL следует использовать как язык «сырых» запросов к базе данных, подобных, например, запросам языка SQL. Предполагается, что Dgraph доверяет клиентам DQL. DQL позволяет выполнять продвинутые запросы к базе данных и лучше подходит для работы с большими данными, допускающими пакетную обработку.

## 1.5 Схема DQL

Схема DQL содержит данные о типах *предикатов*. Предикат — это наименьший фрагмент информации о сущности. Предикат может содержать литеральное значение или связь с другой сущностью. Рассмотрим на примере:

- пусть мы сохраняем имя сущности — ”Петя”. Тогда мы можем использовать некоторый предикат name, значением которого будет строка ”Петя”;
- пусть мы сохраняем отношение сущностей: Петя знает Машу. Тогда мы можем использовать некоторый предикат knows, значением которого будет идентификатор сущности, представляющей Машу.

Объявление предиката в схеме имеет вид

имя\_предиката: тип\_предиката директивы .

где директивы опциональны. Некоторые из поддерживаемых типов предикатов:

- int — целочисленный (32 бита);
- float — вещественнозначный (64 бита);
- string — строковый;
- uid — численный универсальный идентификатор.

Предикат также может содержать список элементов — для этого тип в объявлении обрамляется квадратными скобками.

Посредством директив, в частности, определяются индексы на предикаты (директива @index) или обратные связи для uid (директива @reversed).

## 1.6 DQL и графовые алгоритмы

Хотя язык DQL подобен языку SQL по своей направленности, DQL ограничен в своей выразительности и не пригоден для реализации графовых алгоритмов. Например, запрос с подсчётом всех треугольников в графе не выразим на языке DQL.

DQL поддерживает альтернативу классическому обходу графа в ширину — *рекурсивный запрос*. В запросе указываются глубина и предикаты схемы DQL, по которым совершается обход. В запросе (не только рекурсивном) может быть указана директива `@ignorereflex`: она принудительно удаляет дочерние узлы, которые достижимы из самих себя как родительские.

DQL также поддерживает поиск кратчайшего пути между двумя вершинами. В теле запроса указываются предикаты, по которым совершается обход.

## 1.7 Формат RDF

В Dgraph встроена поддержка формата RDF при создании, импорте и экспорте данных. Resource Description Framework (RDF) [4] — семантический веб-стандарт обмена данными, посредством которого выражаются утверждения о ресурсах. Утверждения описываются в форме троек, оканчивающихся точкой:

`<субъект> <предикат> <объект> .`

Каждая тройка представляет некоторый факт о сущности. В Dgraph `<субъект>` всегда является сущностью и имеет тип `uid`, `<объект>` может быть или другой сущностью, или литеральным значением, а `<предикат>` определяет отношение между субъектом и объектом. Например,

`<0x01> <name> "Петя" .`

`<0x01> <knows> <0x02> .`

При создании сущности могут быть ассоциированы с некоторыми временными именами, что позволит Dgraph самостоятельно назначить им UID. Для этого имена предваряются строчкой «`_:`». Таким образом, все упоминания одного и того же имени, например, `_:Petya`, будут соответствовать одной и той же сущности в рамках операции. Идентификаторы такого формата называются *blank-идентификаторами*.



Dgraph работает с собственным расширением формата RDF, позволяющим назначать свойства не только сущностям (субъектам), но и отношениям (предикатам). Свойства отношений или *фасеты* (facets) описываются парами ключ-значение, перечисляемыми в круглых скобках после указания объекта RDF. Например,

```
_:Petya <car> "A123BC" (since=2020-01-01T12:00:00, first=true) .
```

Ключи фасетов должны быть уникальными. Все фасеты должны указываться в рамках одной тройки RDF – иначе произойдёт перезапись. Допустимы следующие типы фасетов: string, bool, int (32 бита), float (64 бита) и dateTime. Типы фасетов не указываются явно, Dgraph автоматически определяет их по формату значения.

## 1.8 Формат JSON

При вставке, обновлении данных или возврате структур также может использоваться формат JSON. Вложенность в JSON-объектах выражает отношения между сущностями. Например,

```
{"name": "Петя", "homeAddress": {"street": "ул. Пушкина"}}
```

интуитивно соответствует сущностям некоторых типов Person, Address и отношению homeAddress между ними.

Данные в форматах RDF и JSON приводятся к единому внутреннему представлению Dgraph, поэтому их использование одинаково эффективно. Однако RDF, в отличие от JSON, позволяет явно назначать объектам типы данных, если это необходимо.

## 1.9 Загрузка данных

Для загрузки больших объёмов данных целесообразно использовать специализированные для импорта инструменты Dgraph. Таких инструментов два: Live Loader и Bulk Loader.

Live Loader используется для загрузки данных в работающем экземпляре Dgraph, где уже может содержаться некоторая информация. Исходные данные

преобразуются загрузчиком в *мутации* (mutations) Dgraph — запросы на создание или изменение данных — для дальнейшей отправки кластеру. Live Loader позволяет управлять назначением UID добавляемым сущностям, а также обновлять существующие данные. Перед использованием Live Loader необходимо определить схему Dgraph, чтобы создать индексы на предикаты и уменьшить общее время загрузки.

Bulk Loader используется только для загрузки данных в новый кластер и не может быть запущен в работающем экземпляре Dgraph. Bulk Loader работает значительно быстрее Live Loader, и поэтому лучше подходит для импорта больших наборов данных в новый кластер. При работе с Bulk Loader должны быть запущены только Zero-узлы Dgraph; Alpha-узлы запускаются по завершении загрузки данных. При использовании Bulk Loader схема Dgraph может быть передана загрузчику вместе с данными.

И Live Loader, и Bulk Loader принимают данные в формате JSON или в Dgraph-расширении формата RDF.

## 2 Формирование запросов и тестовых данных

Для оценки производительности СУБД Dgraph над тестовыми данными выполняются следующие типы графовых запросов<sup>1</sup>:

- (a) рекурсивный запрос с фильтрацией по вершинам на пути;
- (b) выбор всех вершин с заданным значением поля;
- (c) выбор всех вершин с заданным значением поля с фильтрацией по связям и степени вершины;
- (d) подсчёт для вершин агрегации некоторого параметра по соседним вершинам с ограничением на значение параметра;
- (e) поиск кратчайшего пути между вершинами с фильтрацией по вершинам на пути.

Графовые запросы выполняются над четырьмя разнородными наборами данных в формате CSV. Далее излагаются описание наборов, их структура, графовая интерпретация, а также конкретные схема и запросы DQL.

### 2.1 Набор данных Elliptic++ Transactions

#### 2.1.1 Описание и структура

Набор Elliptic++ Transactions [5] содержит данные о 203 769 транзакциях Bitcoin, их легальности и потоке. Набор состоит из следующих файлов:

- файл `txs_features.csv` (663 МБ), в котором определяются основные атрибуты транзакций: временная метка в целочисленном диапазоне [0, 49], 93 локальных атрибута, описывающих собственную информацию транзакции, 72 агрегированных атрибута, составленных на основе входящих и исходящих транзакций, и 17 дополнительных атрибутов, отражающих статистическую информацию;
- файл `txs_classes.csv` (2.3 МБ), в котором определяется класс транзакции. Множество всех транзакций разбивается на три класса: легальные (42 019), нелегальные (4 545) и неопределённые (157 205);

---

<sup>1</sup>Запросы обозначаются строчными буквами латинского алфавита для удобной ссылки на них в дальнейшем.

- файл `txs_edgelist.csv` (4.3 МБ), в котором хранятся 234 355 упорядоченных пар идентификаторов транзакций, отражающих их поток.

## 2.1.2 Интерпретация и схема

Исходные данные естественным образом представляются ориентированным графом, вершины которого — транзакции, а направленные рёбра соответствуют потоку транзакций. В листинге 1 приводится фрагмент DQL-схемы набора, где определены предикаты, используемые в дальнейшем при выполнении запросов. Исходные идентификаторы транзакций представляются предикатом `id`. Все исходящие транзакции из данной транзакции представляются предикатом `successors`.

Листинг 1: Фрагмент DQL-схемы набора данных Elliptic++ Transactions.

```
1 id: int @index(int) .
2 Time_step: int .
3 Local_feature_1: float .
4 class: int @index(int) .
5 successors: [uid] @reverse .
6 # Остальные 181 предикат...
```

## 2.1.3 Запросы

Выбор вершин графа и предикатов, участвующих в запросах, является важнейшей составляющей оценки. Во многих ситуациях показательные результаты достигаются, когда в запросе участвуют вершины с наибольшей полустепенью захода или исхода — это нужным образом усложняет выполнение запросов. Для обнаружения таких вершин пишутся вспомогательные запросы.

**Запрос (а)** В листинге 2 приводится вспомогательный запрос для определения транзакции с наибольшим числом исходящих транзакций. Предикат `id` этой транзакции равен 2 984 918, а количество исходящих транзакций — 472.

Листинг 2: Запрос для определения транзакции с наибольшим числом исходящих транзакций.

```
1 {  
2   var(func: has(successors)) {  
3     succCount as count(successors)  
4   }  
5  
6   result(func: uid(succCount), orderdesc: val(succCount), first: 1) {  
7     id  
8     val(succCount)  
9   }  
10 }
```

В листинге 3 приводится запрос (а). Рекурсивный обход начинается в транзакции с наибольшим числом исходящих транзакций и завершается на глубине 50. Вершины на пути фильтруются по медианной временной метке.

Листинг 3: Запрос (а) для набора данных Elliptic++ Transactions.

```
1 {  
2   result(func: eq(id, 2984918)) @recurse(depth: 50) {  
3     uid  
4     successors @filter(lt(Time_step, 25))  
5   }  
6 }
```

**Запрос (b)** В листинге 4 приводится запрос (b). В запросе выбираются транзакции, легальность которых не определена — транзакции со значением 3 предиката class, составляющие 77% от всех транзакций.

Листинг 4: Запрос (b) для набора данных Elliptic++ Transactions.

```
1 {  
2   result(func: eq(class, 3)) {  
3     uid  
4   }  
5 }
```

**Запрос (с)** В листинге 5 приводится вспомогательный запрос для определения транзакции с наибольшим числом входящих транзакций. Предикат `id` этой транзакции равен 43 388 675, а количество входящих транзакций — 284.

Листинг 5: Запрос для определения транзакции с наибольшим числом входящих транзакций.

```
1 {  
2   var(func: has(~successors)) {  
3     predCount as count(~successors)  
4   }  
5  
6   result(func: uid(predCount), orderdesc: val(predCount), first: 1) {  
7     id  
8     val(predCount)  
9   }  
10 }
```

В листинге 6 приводится запрос (с). Из транзакций запроса (b) отбираются те, что входят в транзакцию, обладающую наибольшим числом входящих транзакций. Полученные транзакции фильтруются по степени 10.

Листинг 6: Запрос (с) для набора данных Elliptic++ Transactions.

```
1 {  
2   transactions as var(func: eq(class, 3)) @cascade(successors) {  
3     successors @filter(eq(id, 43388675))  
4   }  
5  
6   var(func: uid(transactions)) {  
7     succCount as count(successors)  
8     predCount as count(~successors)  
9     degree as math(succCount + predCount)  
10  }  
11  
12  result(func: ge(val(degree), 10)) {  
13    uid  
14  }  
15 }
```

**Запрос (d)** В листинге 7 приводится запрос (d). Для каждой транзакции подсчитывается среднее значение поля `Local_feature_1` связанных транзакций, которые удовлетворяют фильтру на `Local_feature_1`.

Листинг 7: Запрос (d) для набора данных `Elliptic++ Transactions`.

```
1 {
2   var(func: has(id)) {
3     successors @filter(le(Local_feature_1, -0.15)) {
4       succFeat as Local_feature_1
5     }
6     succAvgFeat as avg(val(succFeat))
7   }
8
9   var(func: has(id)) {
10    ~successors @filter(le(Local_feature_1, -0.15)) {
11      predFeat as Local_feature_1
12    }
13    predAvgFeat as avg(val(predFeat))
14  }
15
16  result(func: has(id)) {
17    uid
18    avgFeat : math((succAvgFeat + predAvgFeat) / 2)
19  }
20 }
```

**Запрос (e)** В листинге 8 приводится запрос (e). Выполняется поиск кратчайшего пути между транзакциями с наибольшим числом исходящих и входящих транзакций. Транзакции на пути фильтруются по временной метке.

## 2.2 Набор данных MOOC User Actions

### 2.2.1 Описание и структура

Набор `MOOC User Actions` [6] содержит данные о 411 749 действиях 7 047 пользователей на платформе онлайн-курсов `MOOC`. Каждое действие ассоциировано с одним из 97 онлайн-курсов.

Набор состоит из следующих файлов:

Листинг 8: Запрос (e) для набора данных Elliptic++ Transactions.

```
1 {  
2   start as var(func: eq(id, 2984918))  
3   end as var(func: eq(id, 43388675))  
4   path as shortest(from: uid(start), to: uid(end)) {  
5     successors @filter(gt(Time_step, 25))  
6   }  
7  
8   result(func: uid(path)) {  
9     uid  
10  }  
11 }
```

- файл `mooc_actions.tsv` (11 МБ), в котором пользователи и курсы связываются отношением многие ко многим посредством действий, и действиям назначаются временные метки;
- файл `mooc_action_features.tsv` (35 МБ), в котором определяются 4 вещественнозначных атрибута действия;
- файл `mooc_action_labels.tsv` (3.5 МБ), в котором действиям назначаются бинарные метки соответственно тому, является ли действие последним действием пользователя на курсе.

## 2.2.2 Интерпретация и схема

Исходные данные представляются ориентированным графом, вершины которого — пользователи, действия и курсы, а рёбра связывают пользователей с действиями и действия с курсами. В листинге 9 приводится DQL-схема набора. Исходные идентификаторы пользователей, действий и курсов представляются предикатами `userId`, `actionId` и `targetId` соответственно. Все действия одного пользователя выражаются предикатом `performs`, а связь действия и курса описывается предикатом `on`.



### Листинг 9: DQL-схема набора данных MOOC User Actions.

```
1  userId: int @index(int) .
2  actionId: int @index(int) .
3  targetId: int @index(int) .
4  performs: [uid] @reverse .
5  on: uid @reverse .
6  label: int @index(int) .
7  feature0: float .
8  feature1: float .
9  feature2: float .
10 feature3: float .
11 timestamp: float .
```

## 2.2.3 Запросы

**Запрос (а)** В листинге 10 приводится вспомогательный запрос для определения двух пользователей, совершивших наибольшее число действий. Ими являются пользователи с `userId` 1181 (505 действий) и `userId` 1686 (470 действий).

Листинг 10: Запрос для определения двух пользователей, совершивших наибольшее число действий.

```
1  {
2    var(func: has(performs)) {
3      actCount as count(performs)
4    }
5
6    result(func: uid(actCount), orderdesc: val(actCount), first: 2) {
7      userId
8      val(actCount)
9    }
10 }
```

В листинге 11 приводится запрос (а). Рекурсивный обход начинается с пользователя, совершившего наибольшее число действий, и завершается на глубине 5. В обходе участвуют предикаты `performs`, `on`, а также их обратные версии — предикаты `~performs` и `~on`. Это позволяет проводить обход графа без учёта направления рёбер. Фильтрация действий на пути проводится по временной метке.

Листинг 11: Запрос (a) для набора данных MOOC User Actions.

```
1 {  
2   result(func: eq(userId, 1181)) @recurse(depth: 5) @ignorereflex {  
3     uid  
4     performs @filter(ge(timestamp, 1_500_000))  
5     on  
6     ~on @filter(ge(timestamp, 1_500_000))  
7     ~performs  
8   }  
9 }
```

**Запрос (b)** В листинге 12 приводится запрос (b). В выборке участвуют действия, не являющиеся последними действиями пользователя на курсе. Значение предиката `class` этих действий равно 0, а их количество составляет 95% от количества всех действий.

Листинг 12: Запрос (b) для набора данных MOOC User Actions.

```
1 {  
2   result(func: eq(label, 0)) {  
3     uid  
4   }  
5 }
```

**Запрос (c)** В листинге 13 приводится вспомогательный запрос для определения курса, с которым ассоциировано наибольшее число действий — курса с `targetId` 8 и 19 474 действиями.

В листинге 14 приводится запрос (c). Из множества действий запроса (b) выбираются действия, относящиеся к курсу с `targetId` 8. Все вершины полученного множества имеют степень 2 (тривиально, по своей структуре); соответствующая проверка добавлена для поддержания единообразия запросов.

**Запрос (d)** В листинге 15 приводится запрос (d). Для каждого пользователя и курса подсчитывается среднее значение поля `feature1` действий, с которыми этот

Листинг 13: Запрос для определения курса, с которым ассоциировано наибольшее число действий.

```
1 {  
2   var(func: has(targetId)) {  
3     actCount as count(~on)  
4   }  
5  
6   result(func: uid(actCount), orderdesc: val(actCount), first: 1) {  
7     targetId  
8     val(actCount)  
9   }  
10 }
```

Листинг 14: Запрос (с) для набора данных MOOC User Actions.

```
1 {  
2   actions as var(func: eq(label, 0)) @cascade(on) {  
3     on @filter(eq(targetId, 8))  
4   }  
5  
6   var(func: uid(actions)) {  
7     targetsCount as count(on)  
8     usersCount as count(~performs)  
9     degree as math(targetsCount + usersCount)  
10  }  
11  
12  result(func: eq(val(degree), 2)) {  
13    uid  
14  }  
15 }
```

пользователь или курс связан. В подсчёте учитываются только неотрицательные значения поля `feature1`.

**Запрос (е)** В листинге 16 приводится запрос (е). Кратчайший путь между пользователями с `userId 1181` и `userId 1686` вычисляется без учёта направления рёбер и с фильтрацией вершин на пути по временной метке.

Листинг 15: Запрос (d) для набора данных MOOC User Actions.

```
1 {
2   var(func: has(userId)) {
3     performs @filter(ge(feature1, 0)) {
4       userFeat as feature1
5     }
6     avgUserFeat as avg(val(userFeat))
7   }
8
9   var(func: has(targetId)) {
10     ~on @filter(ge(feature1, 0)) {
11       targetFeat as feature1
12     }
13     avgTargetFeat as avg(val(targetFeat))
14   }
15
16   users(func: has(userId)) {
17     uid
18     val(avgUserFeat)
19   }
20
21   targets(func: has(targetId)) {
22     uid
23     val(avgTargetFeat)
24   }
25 }
```

Листинг 16: Запрос (e) для набора данных MOOC User Actions.

```
1 {
2   start as var(func: eq(userId, 1181))
3   end as var(func: eq(userId, 1686))
4   path as shortest(from: uid(start), to: uid(end)) @ignorereflex {
5     performs @filter(ge(timestamp, 1_500_000))
6     ~performs
7     on
8     ~on @filter(ge(timestamp, 1_500_000))
9   }
10
11   result(func: uid(path)) {
12     uid
13   }
14 }
```

## 2.3 Набор данных California Road Network

### 2.3.1 Описание и структура

Набор California Road Network [7] содержит данные о дорожной сети Калифорнии. Набор содержит единственный исходный файл `roadNet-CA.txt` (84 МБ), в котором определяются узлы дорожной сети — пункты назначения или пересечения дорог, и двусторонние связи между ними. Количество узлов: 1 965 206, и они не имеют атрибутов; количество связей: 2 766 607.

### 2.3.2 Интерпретация и схема

Исходные данные представляются обыкновенным неориентированным графом. DQL-схема набор приводится в листинге 17. Предикат `id` представляет исходный идентификатор узла, а предикат `successors` — связанные узлы.

Листинг 17: DQL-схема набора данных California Road Network.

```
1 id: int @index(int) .  
2 successors: [uid] .
```

### 2.3.3 Запросы

**Запрос (а)** В листинге 18 приводится вспомогательный запрос для определения двух узлов с наибольшей степенью. Такими являются узлы с `id` 562 818 (степень 12) и `id` 521 168 (степень 10).

В листинге 19 приводится запрос (а). Поскольку узлы не обладают нетривиальными предикатами, для имитации этих предикатов используется `id`. Так, рекурсивный запрос начинается в узле с `id` 562 818, завершается на глубине 50, и вершины на пути фильтруются по значению предиката `id`.

**Запрос (b)** В листинге 20 приводится запрос (b). Выборка запроса тривиальная и совершается по значению 562 818 предиката `id`.

Листинг 18: Запрос для определения двух узлов наибольшей степени.

```
1 {  
2   var(func: has(successors)) {  
3     succCount as count(successors)  
4   }  
5  
6   result(func: uid(succCount), orderdesc: val(succCount), first: 2) {  
7     id  
8     val(succCount)  
9   }  
10 }
```

Листинг 19: Запрос (a) для набора данных California Road Network.

```
1 {  
2   result(func: eq(id, 562818)) @recurse(depth: 50) @ignorereflex {  
3     uid  
4     successors @filter(lt(id, 562818))  
5   }  
6 }
```

Листинг 20: Запрос (b) для набора данных California Road Network.

```
1 {  
2   result(func: eq(id, 562818)) {  
3     uid  
4   }  
5 }
```

**Запрос (c)** В листинге 21 приводится запрос (c). Для узла с id 562 818 проверяется наличие связи с узлом, id которого равен 562 826 (в действительности это непосредственный сосед исходного узла). Далее проверяется, что степень узла совпадает с ожидаемой — с 12.

**Запрос (d)** В листинге 22 приводится запрос (d). Для каждого узла вычисляется среднее арифметическое значений id соседних узлов, удовлетворяющих фильтру.

Листинг 21: Запрос (с) для набора данных California Road Network.

```
1 {  
2   nodes as var(func: eq(id, 562818)) @cascade(successors) {  
3     successors @filter(eq(id, 562826))  
4   }  
5  
6   var(func: uid(nodes)) {  
7     degree as count(successors)  
8   }  
9  
10  result(func: eq(val(degree), 12)) {  
11    uid  
12  }  
13 }
```

Листинг 22: Запрос (d) для набора данных California Road Network.

```
1 {  
2   var(func: has(id)) {  
3     successors @filter(lt(id, 562818)) {  
4       feature as id  
5     }  
6     avgFeature as avg(val(feature))  
7   }  
8  
9   result(func: uid(avgFeature)) {  
10    uid  
11    val(avgFeature)  
12  }  
13 }
```

**Запрос (е)** В листинге 23 приводится запрос (е). Поиск кратчайшего пути выполняется между узлами с наибольшей степенью. Фильтрация производится по предикату `id`.

Листинг 23: Запрос (e) для набора данных California Road Network.

```
1 {  
2   start as var(func: eq(id, 562818))  
3   end as var(func: eq(id, 521168))  
4   path as shortest(from: uid(start), to: uid(end)) {  
5     successors @filter(lt(id, 562818))  
6   }  
7  
8   result(func: uid(path)) {  
9     uid  
10  }  
11 }
```

## 2.4 Набор данных Stablecoin ERC20 Transactions

### 2.4.1 Описание и структура

*Стейблкоинами* называют специальные токены, предназначенные для поддержания фиксированной стоимости в течение долгого времени. Набор Stablecoin ERC20 Transactions [8] содержит данные о более чем 70 миллионах транзакций ERC20 пяти популярных стейблкоинов. Описания транзакций распределены по трём файлам набора, имеющим одинаковую структуру: `token_transfers.csv` (823 МБ), `token_transfers_V2.0.0.csv` (4.4 ГБ) и `token_transfers_V3.0.0.csv` (5.6 ГБ). В файлах определяются *переводы* (transfers), совершаемые в рамках транзакций. Описание каждого перевода включает номер блока транзакции, индекс транзакции, количество переданных стейблкоинов, временную метку, а также адреса отправителя, получателя и контракта, определяющего стейблкоин. Набор также содержит файл с описанием событий, повлиявших на работу сети, файлы с изменением стоимости стейблкоинов, однако объём информации в них незначительный (суммарно, 60 КБ), и этими данными можно пренебречь. Далее работа ведётся только с файлом `token_transfers.csv`.

### 2.4.2 Интерпретация и схема

Исходные данные можно представить ориентированным графом. В отличие от набора данных Elliptic++ Transactions, где транзакции являются единственными



вершинами в графе и непосредственно связаны друг с другом, здесь вершины графа — переводы и адреса отправителей, получателей или контрактов. Таким образом, переводы являются посредниками при выражении отношения многие ко многим между адресами отправителей и получателей. Направленные рёбра в графе связывают

- адрес отправителя и перевод;
- перевод и адрес получателя;
- перевод и адрес контракта.

В листинге 24 приводится DQL-схема набора. Исходные адреса отправителей, получателей и контрактов представляются предикатом `address`. Все переводы некоторого отправителя выражаются предикатом `from`. Связь перевода и получателя описывается предикатом `to`, и связь перевода и контракта — предикатом `contract`.

Листинг 24: DQL-схема набора данных Stablecoin ERC20 Transactions.

```
1 address: string @index(hash) .
2 from: [uid] @reverse .
3 to: uid @reverse .
4 contract: uid .
5 block_number: int .
6 transaction_index: int .
7 time_stamp: int .
8 value: float @index(float) .
```

### 2.4.3 Запросы

**Запрос (а)** В листинге 25 приводится вспомогательный запрос для определения адреса отправителя наибольшего числа переводов. Соответствующий предикат `address` равен `0x74de5d4fc6f63e00296fd95d33236b9794016631`, а количество переводов — `147 437`.

В листинге 26 приводится запрос (а). Рекурсивный обход выполняется от адреса отправителя наибольшего числа переводов и до глубины 3. На пути рассматриваются предикат `from` с фильтрацией по временной метке и предикат `to`.

Листинг 25: Запрос для определения адреса отправителя наибольшего числа переводов.

```
1 {  
2   var(func: has(from)) {  
3     tranCount as count(from)  
4   }  
5  
6   result(func: uid(tranCount), orderdesc: val(tranCount), first: 1) {  
7     address  
8     val(tranCount)  
9   }  
10 }
```

Листинг 26: Запрос (a) для набора данных Stablecoins ERC20 Transactions.

```
1 {  
2   result(func: eq(address, "0x74de5d4fc6f63e00296fd95d33236b9794016631"))  
3     @recurse(depth: 3) {  
4       uid  
5       from @filter(ge(time_stamp, 1653000000))  
6       to  
7     }  
8 }
```

**Запрос (b)** В листинге 27 приводится запрос (b). Переводы отбираются по числу стейблкоинов.

Листинг 27: Запрос (b) для набора данных Stablecoins ERC20 Transactions.

```
1 {  
2   result(func: eq(value, 1000)) {  
3     uid  
4   }  
5 }
```

**Запрос (c)** В листинге 28 приводится вспомогательный запрос для определения адреса контракта с наибольшим числом переводов.

Листинг 28: Запрос для определения адреса контракта с наибольшим числом переводов.

```
1 {  
2   var(func: has(~contract)) {  
3     tranCount as count(~contract)  
4   }  
5  
6   result(func: uid(tranCount), orderdesc: val(tranCount), first: 1) {  
7     address  
8     val(tranCount)  
9   }  
10 }
```

В листинге 29 приводится запрос (с). Переводы запроса (b) фильтруются по адресу контракта с наибольшим числом переводов. Полученные вершины фильтруются по степени 2.

Листинг 29: Запрос (с) для набора данных Stablecoins ERC20 Transactions.

```
1 {  
2   transfers as var(func: eq(value, 1000)) @cascade(contract) {  
3     contract @filter(eq(address,  
4       "0xdac17f958d2ee523a2206206994597c13d831ec7"))  
5   }  
6  
7   var(func: uid(transfers)) {  
8     fromAddresses as count(~from)  
9     toAddresses as count(to)  
10    degree as math(fromAddresses + toAddresses)  
11  }  
12  
13  result(func: eq(val(degree), 2)) {  
14    uid  
15  }  
16 }
```

**Запрос (d)** В листинге 30 приводится запрос (d). Для каждого адреса подсчитывается среднее значение стейблкоинов связанных переводов, которые удовлетворяют фильтру.

Листинг 30: Запрос (d) для набора данных Stablecoins ERC20 Transactions.

```
1 {
2   var(func: has(address)) {
3     from @filter(ge(value, 1000)) {
4       fromValue as value
5     }
6     avgFromValue as avg(val(fromValue))
7   }
8
9   var(func: has(address)) {
10     ~to @filter(ge(value, 1000)) {
11       toValue as value
12     }
13     avgToValue as avg(val(toValue))
14   }
15
16   result(func: has(address)) {
17     uid
18     avgValue : math((avgFromValue + avgToValue) / 2)
19   }
20 }
```

**Запрос (е)** В листинге 31 приводится запрос (е). Выполняется поиск кратчайшего пути между адресом отправителя наибольшего числа контрактов и адресом 0, регулярно участвующим в переводах. Вершины на пути фильтруются по временной метке.

Листинг 31: Запрос (е) для набора данных Stablecoins ERC20 Transactions.

```
1 {
2   start as var(func: eq(address,
3     "0x74de5d4fc6f63e00296fd95d33236b9794016631"))
4   end as var(func: eq(address,
5     "0x0000000000000000000000000000000000000000"))
6   path as shortest(from: uid(start), to: uid(end)) {
7     from @filter(ge(time_stamp, 1653000000))
8     to
9   }
10
11   result(func: uid(path)) {
12     uid
13   }
14 }
```

## 3 Загрузка данных и выполнение запросов

### 3.1 Развёртывание кластера Dgraph

Для развёртывания кластера Dgraph используется инструмент Docker Compose [9], предназначенный для работы с мультиконтейнерными приложениями. Это позволяет развёртывать узлы кластера Dgraph в изолированных процессах — Docker-контейнерах, что не требует выполнения предварительных работ на основном устройстве, а также позволяет единственный раз настроить работу контейнеров и их взаимодействие в конфигурационном файле Docker Compose. В листинге 32 приложения А приводится конфигурационный файл `compose.yml` кластера Dgraph.

Кластер Dgraph составляют один Zero-узел и один Alpha-узел, разделяющие единое хранилище данных (Docker volume). Zero-узел использует следующие порты:

- 5080 для внутрикластерного взаимодействия и работы с инструментами импорта данных Dgraph (Live Loader, Bulk Loader) по протоколу gRPC;
- 6080 для администрирования кластером по протоколу HTTP.

Alpha-узел использует следующие порты:

- 7080 для взаимодействия внутри кластера по протоколу gRPC;
- 8080 для обработки клиентских запросов по протоколу HTTP;
- 9080 для обработки клиентских запросов по протоколу gRPC.

### 3.2 Преобразование исходных данных

Импорт данных в новый кластер будет осуществляться с использованием инструмента Dgraph Bulk Loader. Bulk loader принимает данные в формате JSON или Dgraph-расширении формата RDF, однако исходные данные представлены в формате CSV. Таким образом, возникают задачи:

1. непосредственной конвертации форматов данных;
2. логического преобразование данных из реляционной модели в графовую модель.

В документации Dgraph для конвертации форматов CSV в JSON рекомендуется использовать инструмент `csv2json` [10], что решает первую задачу. Для

решения второй задачи предлагается передавать на вход утилите `jq` [11] результат конвертации, а также текст программы на высокоуровневом функциональном языке программирования JQ, предназначенном для преобразования литералов JSON. Это решает задачу подготовки данных, но требует написания индивидуальных, во многом дублирующих друг друга, программ на языке JQ для каждого набора данных. Более универсальным решением будет реализация собственного инструмента, способного по некоторому описанию исходных файлов и правил их преобразования генерировать данные в нужном виде. Поставим задачу разработки такого инструмента.

## 3.3 Проектирование преобразователя данных

### 3.3.1 Формат входных и выходных данных

Для описания правил преобразования исходных файлов достаточно возможностей формата YAML — требуемое представляется на нём просто и естественно, синтаксис описания нетрудно корректируется, а средства анализа YAML распространены. Обработываемые файлы описываются в одном YAML-файле, который далее мы будем называть *конфигурационным*.

Чтобы упростить работу с определением расположения файлов, целесообразно хранить их в структурированном виде. Определим, что исходные файлы будут располагаться в одной директории с именем `source`, а соответствующий конфигурационный файл `convert.yml` — находиться непосредственно рядом с `source`. Директорию, содержащую `source` и `convert.yml`, будем называть *корневой директорией*.

Целевым форматом преобразования будет Dgraph-расширение формата RDF: оно выразительнее JSON, поскольку позволяет при необходимости явно назначать объектам типы данных. Результат преобразования будет сохраняться в файле `output.rdf` в корневой директории набора.

Таким образом, преобразователю достаточно будет передать путь к корневой директории, где впоследствии будет сгенерирован пригодный для импорта файл `output.rdf`. Удобной будет опция передачи преобразователю пути к директории с несколькими корневыми директориями сразу для параллельной (при возможности) обработки соответствующих наборов данных.

### 3.3.2 Спецификация конфигурационного файла

На верхнем уровне конфигурационного файла `convert.yml` должен быть определён массив `files`, содержащий описания обрабатываемых файлов. Незадействованные файлы описывать в `files` не нужно. Файлы обрабатываются в порядке их следования в массиве `files`. Ниже определяются допустимые поля в описании файлов. Все поля по умолчанию считаются обязательными, если явно не оговорено обратное.

**Поле** `name` Определяет имя обрабатываемого файла.

**Поле** `delimiter` Определяет символ, разделяющий столбцы в файле. Поле опциональное, и по умолчанию разделителем является запятая, что соответствует формату CSV. При работе с форматом TSV разделителем является символ табуляции `\t`. Символы возврата каретки `\r` или перевода строки `\n` не могут быть разделителями.

**Поле** `comment` Определяет символ, являющийся началом однострочного комментария. Поле опциональное. Все строки файла, начинающиеся непосредственно с `comment`, игнорируются. Если `comment` стоит не в начале строки, он считается обычным символом. Символ начала комментария не может совпадать с `delimiter` или быть равным `\r`, `\n`.

**Поле** `declarations` Массив с объявлениями имён и типов столбцов файла, используемых в правилах преобразования. Каждое объявление содержит

- поле `name` — имя столбца;
- поле `type` — тип (данных) столбца. Поддерживаемые типы: строка `string`, целое число `int` (32 бита), число с плавающей запятой `float` (64 бита) и идентификатор `id`. Типы данных `string`, `int` и `float` также будут называться *тривиальными*;
- поле `prefix` (устанавливается, только если `type` — `id`). Если атрибут определён, все значения столбца будут предварены указанным префиксом — это необходимо для обеспечения уникальности `blank-идентификаторов`. Действительно, поскольку идентификаторы в реляци-

онных таблицах обычно целочисленные, blank-идентификаторы с одинаковым номером, но соответствующие разным сущностям, совпадут. Присоединение различных префиксов обеспечит blank-идентификаторам уникальность. Как следствие, если одна и та же сущность используется в различных исходных файлах, префикс её идентификаторов должен совпадать везде (или везде отсутствовать).

Подчеркнём, что все столбцы, используемые в правилах преобразования, должны быть определены в массиве `declarations`. Недействительные столбцы файла определять в `declarations` не следует.

**Поле `artificial_declaration`** Объявление искусственного столбца, данные которого генерируются во время работы преобразователя. В данный момент используется только для генерации искусственных идентификаторов, когда они явно не указываются в исходных файлах (например, в наборе данных 2.4 не предусмотрены идентификаторы переводов в `token_transfers.csv`). Искусственное объявление содержит те же поля, что и обычное объявление: `name`, `type` (только `id`) и `prefix`.

**Поле `entity_facets`** Введём необходимый контекст. Сущность реляционной модели, хранящая информацию о сопоставлении других сущностей, в графовой модели переходит в отношение между этими сущностями. Исходную сущность назовём *соединительной*. Атрибуты соединительной сущности переходят в атрибуты соответствующего отношения, или фасеты, которые в `Dgraph` назначаются ребру все и в рамках одного объявления. Поскольку атрибуты соединительной сущности могут располагаться в разных реляционных таблицах, обрабатывая файлы приходится «накапливать» фасеты, ассоциированные с этой сущностью, до тех пор, пока они не будут собраны все. Только после этого полученные фасеты можно добавить к фасетам выписываемого литерала `RDF`.

Итак, поле `entity_facets` определяет массив правил, по которым в памяти преобразователя сохраняются фасеты, ассоциированные с соединительной сущностью. Каждое такое правило содержит

- поле `id` — имя столбца идентификаторов соединительной сущности;
- поле `key` — имя ключа фасета, которое будет использоваться в `Dgraph`;



- поле `value` — имя столбца значений фасета. Столбец должен быть тривиального типа.

Таким образом, для каждого правила массива `entity_facets` со всяким идентификатором из столбца `id` будет ассоциирован фасет с ключом `key` и соответствующим значением столбца `value`. Сохранение фасетов происходит перед выписыванием литералов RDF.

**Поле `rdfs`** Массив правил, по которым в `output.rdf` выписываются литералы RDF. Каждое такое правило содержит

- поле `subject` — имя столбца субъекта RDF. Столбец должен быть типа `id`;
- поле `predicat` — имя соответствующего предиката схемы Dgraph;
- поле `object` — имя столбца объекта RDF;
- опциональное поле `cast_object_to` — изменённый тип объекта. Используется, если необходимо выполнить преобразование типа объекта RDF (например, чтобы интерпретировать столбец типа `id` как `int` для получения буквального значения идентификатора);
- опциональное поле `entity_facets_id` — имя столбца идентификаторов соединительной сущности. Применяется, если к фасетам данного литерала RDF необходимо добавить фасеты, ассоциированные с соединительной сущностью;
- опциональное поле `facets` — массив описаний фасетов, добавляемых к фасетам данного литерала RDF. Отличается от `entity_facets_id` тем, что фасеты `facets` не ассоциированы с какой-либо сущностью, а их значения берутся непосредственно из обрабатываемого файла. Каждое описание фасета содержит
  - поле `key` — имя ключа фасета, которое будет использоваться в Dgraph;
  - поле `value` — имя столбца значений фасета. Столбец должен быть тривиального типа.

### 3.3.3 Конфигурационные файлы тестовых наборов данных

Приведём конфигурационные файлы `convert.yml` тестовых наборов данных. В листинге 33 приложения А приводится фрагмент конфигурационного файла набора `Elliptic++ Transactions`, в листинге 34 приложения А — фрагмент конфигурационного файла набора `MOOC User Actions`, в листинге 35 приложения А — конфигурационный файл набора `California Road Network` и в листинге 36 приложения А — фрагмент конфигурационного файла набора `Stablecoin ERC20 Transactions`.

### 3.3.4 Модули преобразователя

При проектировании преобразователя естественным образом выделяются два модуля:

1. Модуль работы с RDF. Содержит структуры данных для представления в программе литералов Dgraph-расширения формата RDF, а также необходимые методы работы с ними.
2. Модуль преобразования. Содержит структуры данных для представления конфигурационного файла, методы валидации структур и всю основную функциональность преобразования.

Для описания функциональности модулей частично используется объектно-ориентированная терминология: понятия объекта, метода, конструктора и др. Говоря, что модуль «предоставляет» некоторое определение, подразумевается, что оно публичное, т.е. может использоваться другими модулями.

### 3.3.5 Модуль работы с RDF

Определим представление субъекта, предиката и объекта RDF — термов RDF — в программе. Заметим, что в терме можно выделить его значение, а также «оформление» этого значения, влияющее на контекст использования. Например, `blank-идентификатор` указывается без оформления, `UID` обрамляется треугольными скобками, а литеральное значение берётся в двойные кавычки.

Введём тип перечисления (enumeration) `Decoration` с допустимыми значениями `None`, `AngleBrackets` и `Quotes`. Тогда терм RDF будет представляться структурой `Term` с полями

- `value` — значение терма;
- `decoration` — оформление значения терма типа `Decoration`.

Значения фасетов RDF также могут иметь оформление. Например, целочисленный литерал ничем не обрамляется, а строковый — берётся в двойные кавычки.

Соответственно, фасет RDF представляется структурой `Facet` с полями

- `key` — имя ключа фасета;
- `value` — значение фасета;
- `decoration` — оформление значения фасета типа `Decoration`.

Структура `Rdf` представляет литерал RDF (с фасетами). `Rdf` содержит поля

- `subject` — субъект RDF типа `Term`;
- `predicat` — предикат RDF типа `Term`;
- `object` — объект RDF типа `Term`;
- `facets` — массив фасетов RDF типа `Facet`.

Модуль предоставляет тип `Decoration`, структуры `Term`, `Facet` и `Rdf` и их поля, а также функции создания этих структур и методы их сериализации.

### 3.3.6 Модуль преобразования

Предполагается, что для представления конфигурационного файла сущностями в программе используется некоторый инструмент десериализации `YAML`. Поскольку исходный конфигурационный файл может содержать ошибки, указанные сущности необходимо валидировать: проверять отсутствие необъявленных значений в правилах, соответствие типов используемых столбцов контекстным ограничениям и т.п. Далее при определении структур данных будем подразумевать, что для них определены методы валидации соответственно спецификации конфигурационного файла.

**Представление правил** Правило сохранения фасета, ассоциированного с соединительной сущностью, представляется структурой `EntityFacetRule`. Структура содержит поля

- `id` — имя столбца идентификаторов соединительной сущности;
- `key` — имя ключа фасета;
- `value` — имя столбца значений фасета.

Правило выписывания литерала RDF может содержать описания фасетов, добавляемых к фасетам данного литерала RDF, не ассоциированных с какой-либо соединительной сущностью, и значения которых берутся непосредственно из текущего файла. Они представляются структурой `FacetRule` с полями

- `key` — имя ключа фасета;
- `value` — имя столбца значений фасета.

Правило выписывания литерала RDF представляется структурой `RdfRule`. Структура содержит поля

- `subject` — имя столбца субъекта RDF;
- `predicat` — имя предиката RDF;
- `object` — имя столбца объекта RDF;
- `cast_object_to` — изменённый тип объекта RDF;
- `facets` — массив правил `FacetRule`;
- `entityFacetsId` — имя столбца идентификаторов соединительной сущности.

**Обработка файлов** Для представления допустимых типов столбцов файла вводится перечисление `DataType`, содержащее элементы `String`, `Int`, `Float` и `Id`.

Объявление имени и типа столбца файла, используемое в правилах преобразования `FacetRule`, `EntityFacetRule` и `RdfRule`, представляется структурой `Declaration`. Структура содержит поля

- `name` — имя столбца в файле;
- `type` — тип столбца из `DataType`;
- `prefix` — префикс идентификатора.

Описание исходного файла и правила его преобразования представляется структурой `File`. Структура содержит поля

- `name` — имя файла;
- `delimiter` — символ разделителя столбцов в файле;
- `comment` — символ начала однострочного комментария;
- `declarations` — массив объявлений типа `Declaration`;
- `artificial_declaraiion` — искусственное объявление типа `Declaration`;

- `entityFacets` — массив правил типа `EntityFacet`;
- `rdfs` — массив правил типа `RdfRule`.

Предполагается, что при обработке исходных файлов используется некоторый инструмент десериализации CSV.

Структура `Dataset` содержит данные верхнего уровня конфигурационного файла — единственный массив `files` описаний `File` исходных файлов. При обработке `files` используется общая таблица фасетов `entitiesFacets`, ассоциированных с соединительными сущностями. Результат преобразования всех файлов записывается в единый файл `output.rdf`.

Модуль предоставляет функции `ProcessDataset` и `ProcessDatasets`, принимающие пути к одной корневой директории или директории с несколькими корневыми директориями соответственно и выполняющие необходимые преобразования.

## 3.4 Реализация преобразователя данных

Для реализации преобразователя выбран язык программирования Go версии 1.22 по следующим причинам:

- обладание всеми необходимыми механизмами абстракции и нативная поддержка многопоточности;
- обильная стандартная библиотека, предоставляющая, в частности, удобные инструменты работы с форматами YAML и CSV;
- компилируемость, обеспечивающая бóльшую скорость выполнения программ в сравнении с интерпретируемыми языками. Быстродействие преобразователя существенно, поскольку он рассчитан на работу с большими объёмами данных.

### 3.4.1 Модуль работы с RDF

Модуль работы с RDF реализован в Go-пакете `rdf`. Пакет предоставляет тип `Decoration`, структуры `Term`, `Facet`, `Rdf`, функции `NewTerm`, `NewFacet`, `NewRdf` создания структур и методы `String` их сериализации (листинги 37, 38, 39 приложения А соответственно).

### 3.4.2 Модуль преобразования

Модуль преобразования реализован в Go-пакете `converter`. Опишем основные особенности реализации модуля.

**Обработка ошибок** В реализации большое внимание уделяется валидации структур данных и обработке ошибок. При возникновении ошибки соответствующий объект `error` передаётся среди возвращаемых значений задействованных функций, оборачивая информацию об ошибке необходимым контекстом вплоть до вызова функций `ProcessDataset` и `ProcessDatasets`.

**Публичные функции** Реализация функции `ProcessDatasets` приводится в листинге 40 приложения А. Функция принимает путь к директории с корневыми директориями наборов данных, и для каждой директории вызывает метод `ProcessDataset`. Вызовы происходят в отдельных горутинах с использованием механизма синхронизации `errgroup` стандартного пакета `sync` языка Go. Если при обработке какого-либо набора возникает ошибка, соответствующая горутина завершает свою работу, не влияя на работу остальных горутин, и функция `ProcessDatasets` возвращает объект ошибки. В противном случае функция возвращает `nil`.

Реализация функции `ProcessDataset` приводится в листинге 41 приложения. Функция принимает путь к корневой директории набора данных и средствами стандартного пакета `yaml.v3` языка Go пробует десериализовать конфигурационный файл `convert.yml` во внутренние структуры модуля: `Dataset`, `File`, `Declaration` и пр. В случае успеха функция передаёт управление обработчику (методу обработки) полученной структуры `Dataset`. При возникновении ошибки функция возвращает соответствующий объект, а иначе — `nil`.

**Внутренняя логика** Обработчик структуры `Dataset` (листинг 42 приложения А) создаёт файл `output.rdf`, а также таблицу фасетов `entitiesFacets` на основе стандартной хеш-таблицы `map` языка Go; `output.rdf` и `entitiesFacets` разделяются обработчиками всех объектов `files` набора данных.

Обработчик структуры `File` (листинг 43 приложения А) вначале валидирует `delimiter`, `comment` и преобразует `declarations`, `artificial_declaration` к

удобному для работы виду. Для десериализации исходного файла используется стандартный пакет `encoding/csv` языка Go. Для каждой полученной записи вызываются функции применения правил сохранения фасетов (листинг 44 приложения А) и выписывания литералов RDF (листинг 45 приложения А).

### 3.4.3 Использование преобразователя

Весь исходный код преобразователя находится в Git-репозитории [12]. Для использования преобразователя на Linux можно клонировать указанный репозиторий CLI-утилитой `git` и применить CLI-утилиту `go`, предназначенную для работы с исходным кодом проектов на языке Go. Программа преобразования `convert.go` располагается в директории `cmd/converter`.

Допустимые опции программы:

- `dataset-path` — путь к корневой директории набора данных;
- `datasets-path` — путь к директории с корневыми директориями набора данных;
- `help` — вывод допустимых опций программы.

Одновременно может быть установлена только одна из опций `dataset-path` и `datasets-path`.

## 3.5 Выполнение запросов

Для выполнения запросов Dgraph и оценки ресурсов, затраченных на их выполнение, написана вспомогательная программа. Она значительно проще, чем программа преобразования, поэтому можно сразу перейти к её реализации.

### 3.5.1 Реализация

Языком реализации также является Go, поскольку Dgraph предоставляет официальный Go-клиент `dgo` [13], который взаимодействует с сервером Dgraph по протоколу gRPC. Этот пакет и будет использоваться при выполнении запросов, поскольку предоставляет всю необходимую функциональность.

**Установка соединения** Программа устанавливает соединение с сервером Dgraph и возвращает клиент Dgraph; соответствующая функция приводится в листинге 46 приложения А.

**Выполнение транзакции** В листинге 47 приложения А приводится функция выполнения запроса и оценки времени, оперативной памяти, затраченных на его обработку. Поскольку запросы в рамках поставленной задачи только запрашивают данные и не модифицируют их, для выполнения запросов клиентом Dgraph создаётся транзакция, предназначенная только для чтения (read-only), которая может обрабатываться в обход общего протокола выполнения запроса и повысить скорость чтения.

**Затраченное время** Результат выполнения запроса содержит информацию о времени, затраченном на его обработку. А именно, это время (в наносекундах)

- синтаксического разбора текста запроса;
- непосредственно выполнения запроса;
- сериализации результата выполнения запроса;

Общее время выполнения запроса получается как сумма указанных значений.

**Затраченная оперативная память** Dgraph не предоставляет информации о потреблении оперативной памяти во время выполнения запроса. Чтобы получить эти данные, в отдельном потоке при выполнении запроса с постоянным интервалом во времени отслеживается свободная оперативная память и, в частности, её минимальное значение. Разница между свободной памятью до выполнения запроса и её минимальным значением во время выполнения интерпретируется как потребление оперативной памяти. Разумеется, для получения корректных результатов на основном устройстве не должны одновременно выполняться другие процессы, много и (или) непредсказуемо потребляющие оперативную память.

Для получения информации о свободной оперативной памяти в программе используется внешний пакет `pbnjay/memory` [14].



### 3.5.2 Использование

Исходный код программы `benchmark.go` также находится в git-репозитории [12], в каталоге `cmd/benchmark`. Доступные опции программы:

- `query-path` — путь к файлу, содержащему единственный read-only запрос DQL;
- `host` — хост сервера Dgraph (по умолчанию, `localhost`);
- `port` — gRPC-порт сервера Dgraph (по умолчанию, 9080);
- `duration` — временной интервал между замерами свободной оперативной памяти при выполнении запроса в формате аргумента функции `time.ParseDuration` стандартной библиотеки языка Go (по умолчанию, 100ms);
- `print-respond` — требуется ли печатать результат выполнения запроса в формате JSON;
- `help` — вывод доступных опций программы.

## 4 Оценка производительности

### 4.1 Статистика тестовых наборов данных

#### 4.1.1 Вершины и рёбра

В таблице 1 содержится итоговая информация о числе вершин и рёбер тестовых наборов данных. Для набора Elliptic++ Transactions эти данные остались прежними: число вершин есть число транзакций, число рёбер — число потоков транзакций. Для набора MOOC User Actions число вершин складывается из числа пользователей, действий и онлайн-курсов, а число рёбер равняется удвоенному числу действий, поскольку одно действие соединяет пользователя и онлайн-курс. Для набора California Road Network данные также не изменились: число вершин — число узлов, число рёбер — число связей. Для набора Stablecoin ERC20 Transactions число вершин складывается из числа адресов, полученного вспомогательным запросом, и числа переводов; число рёбер равняется утроенному числу переводов, поскольку один перевод связывает адрес отправителя, адрес получателя и адрес контракта.

Таблица 1 — Число вершин и рёбер тестовых наборов данных.

Набор данных	Число вершин	Число рёбер
Elliptic++ Transactions	203 769	234 355
MOOC User Actions	418 893	823 498
California Road Network	1 965 206	2 766 607
Stablecoin ERC20 Transactions	6 803 466	15 840 393

#### 4.1.2 Дисковое пространство

В таблице 2 содержится информация о дисковом пространстве, занимаемом исходными файлами тестовых наборов и данными кластера Dgraph. Показатели для исходных файлов тривиально получены суммированием размера файлов. Показатели для кластера Dgraph сняты с директории /dgraph/p хранилища Docker, который разделяют Alpha- и Zero-узлы кластера. Директория /dgraph/p содержит загруженные данные, индексы на предикаты и обратные рёбра во внутреннем представлении Dgraph.

Таблица 2 — Дискковое пространство, занимаемое тестовыми наборами данных, МБ.

Набор данных	Исходные файлы	Данные кластера
Elliptic++ Transactions	670	744
MOOC User Actions	48	74
California Road Network	84	122
Stablecoin ERC20 Transactions	823	901

## 4.2 Выполнение запросов

Для получения следующих показателей все запросы выполнялись в одинаковых условиях и по 5 раз; среднее значение результатов выполнения считается итоговым значением. Характеристики устройства, на котором производились замеры:

- объём оперативной памяти: 16 ГБ;
- процессор: Intel Core i5 10300H, 2500 МГц;
- видеокарта: Nvidia GeForce GTX 1650 Ti.

### 4.2.1 Время выполнения

В таблице 3 приводится общее время выполнения запросов. В общем времени учитывается синтаксический разбор текста запроса, непосредственно выполнение и сериализация результатов выполнения.

Таблица 3 — Время выполнения запросов, мс.

Набор данных	Запрос (a)	Запрос (b)	Запрос (c)	Запрос (d)	Запрос (e)
Elliptic++ Transactions	27	85	448	3 072	23
MOOC User Actions	8 030	202	1 235	2989	2 667
California Road Network	8 849	1	1	13 661	30 540
Stablecoin ERC20 Transactions	4 199	1 093	1 099	56 025	5 546

Подчеркнём, что показатели выполнения одного типа запросов на разных наборах тестовых данных не сравнимы между собой. Указанные наборы разнородные, и запросы раздела 2 написаны с учётом индивидуальных особенностей каждого набора так, чтобы отражать наиболее показательные результаты для данного набора, не беря во внимание показатели для других наборов. Так, например, выбор всех вершин с заданным значением поля (запрос (b)) на наборе California

Road Network занимает 1 мс, поскольку выбирается единственная вершина по значению предиката `id` (набор не обладает нетривиальными предикатами), и в то же время выбор всех вершин набора Stablecoin ERC20 Transactions по значению предиката `value` занимает более показательные 1 093 мс.

Отметим некоторые выделяющиеся показатели таблицы 3. Видно, что в среднем наиболее долго выполняются запросы типа (d) — это ожидаемо, исходя из их определения. Самое продолжительное время выполнения такого запроса составляет более 56 секунд и соответствует набору данных ERC20 Stablecoin Transactions. Поиск кратчайшего пути (запрос (e)) и рекурсивный запрос ((a)) наиболее продолжительны на наборе данных California Road Network: более 30 и 8 секунд соответственно. Хотя вершины этого набора не обладают атрибутами, его структура достаточно сложна, чтобы показывать интересные результаты при обходе графа.

## 4.2.2 Оперативная память

В таблице 4 приводятся данные о потреблении оперативной памяти во время выполнения запросов. Показатели получены как разность свободной оперативной памяти до выполнения запроса и её минимального значения во время выполнения.

Таблица 4 — Потребление оперативной памяти при выполнении запросов, МБ.

Набор данных	Запрос (a)	Запрос (b)	Запрос (c)	Запрос (d)	Запрос (e)
Elliptic++ Transactions	20	60	684	1 941	19
MOOC User Actions	4 340	112	1 655	853	1 839
California Road Network	419	4	1	9 266	416
Stablecoin ERC20 Transactions	2 447	351	954	14 293	2 170

По умолчанию кластер Dgraph может использовать всю свободную оперативную память во время выполнения запроса. Это особенно заметно при выполнении запроса (d) на наборе данных Stablecoin ERC20 Transactions: объём потребляемой оперативной памяти составляет около 14 ГБ и не больше лишь по той причине, что объём всей оперативной памяти основного устройства равен 16 ГБ.

Видно, что данные таблицы 4 коррелируют с данными таблицы 3 в среднем, но не всегда. Например, с приблизительно одинаковым временем выполнения запроса (a) на наборах данных MOOC User Actions и California Road Network (около 8 секунд), объёмы затраченной оперативной памяти различаются практически в

10 раз: 4 340 МБ и 419 МБ. Это связано со сложной логикой обхода графа МООС User Actions, где в рекурсивном запросе рассматриваются 4 предиката (performs, ~performs, on и ~on), в то время как при обходе графа California Road Network рассматривается единственный предикат successors.

# ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы поставленные цели во многом были достигнуты.

Важную и большую часть работы занимает программа преобразования, которая по специальному конфигурационному файлу преобразует исходные файлы наборов данных CSV в расширение формата RDF, пригодное для импорта в Dgraph. Разработанный инструмент позволяет унифицировать процесс преобразования форматов и не писать дублированный ad hoc код. Выстроенная архитектура преобразователя позволяет легко добавлять новые возможности, если это необходимо.

Выполнение запросов и их оценивание также удалось автоматизировать и унифицировать путём написания вспомогательной программы. Полученные результаты могут быть воспроизведены и использованы для сравнения Dgraph с другими графовыми СУБД. Ограниченная выразительность языка DQL, с одной стороны, позволяет эффективно обрабатывать определённый класс запросов — OLTP, а с другой — делает СУБД неприменимой, если в работе используются сложные алгоритмы на графах.

Тем не менее, в оценке не учитываются некоторые возможности Dgraph, характеризующие его как распределённую графовую СУБД.

1. Тестовые данные интерпретируются без использования фасетов. Например, для наборов MOOC User Actions или Stablecoin ERC20 Transactions возможно интерпретировать атрибуты пользовательских действий и переводов транзакций соответственно как атрибуты отношений, а не вершин-посредников. Однако такая интерпретация хуже согласуется с определёнными графовыми запросами, поскольку они в большинстве используют информацию о вершинах графа, а не отношениях.
2. Оценка произведена для кластера Dgraph, который составляют один Zero- и один Alpha-узел, что не отражает всех возможностей масштабирования и распределённого использования СУБД. Это сделано также для возможности сравнения Dgraph с другими СУБД в рамках курсового проекта.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Dgraph | Open Source, AI-Ready Graph Database. — URL: <https://dgraph.io/>.
2. GraphQL | A query language for your API. — URL: <https://graphql.org/>.
3. GraphQL vs DQL - Dgraph Blog. — URL: <https://dgraph.io/blog/post/graphql-vs-dql/>.
4. RDF - Semantic Web Standart. — URL: <https://www.w3.org/RDF/>.
5. EllipticPlusPlus/Transactions Dataset at main | git-disl/EllipticPlusPlus. — URL: <https://github.com/git-disl/EllipticPlusPlus>.
6. SNAP: Social network: MOOC User Actions Dataset. — URL: <https://snap.stanford.edu/data/act-mooc.html>.
7. SNAP: Network datasets: California road network. — URL: <https://snap.stanford.edu/data/roadNet-CA.html>.
8. SNAP: ERC20 Stablecoins. — URL: <https://snap.stanford.edu/data/ERC20-stablecoins.html>.
9. Docker Compose overview | Docker Docs. — URL: <https://docs.docker.com/compose/>.
10. csv2json - npm. — URL: <https://www.npmjs.com/package/csv2json>.
11. jq. — URL: <https://jqlang.github.io/jq/>.
12. affeeal/iu9-databases-coursework: "Databases" coursework, BMSTU, IU9, 6th term. — URL: <https://github.com/affeeal/iu9-databases-coursework>.
13. dgraph-io/dgo: Official Dgraph Go client. — URL: <https://github.com/dgraph-io/dgo>.
14. pbnjay/memory: A go function to report total system memory. — URL: <https://github.com/pbnjay/memory>.

## ПРИЛОЖЕНИЕ А

Листинг 32: Конфигурационный файл Docker Compose запускаемого кластера Dgraph.

```
1 services:
2   zero:
3     image: dgraph/dgraph:${DGRAPH_VERSION}
4     volumes:
5       - dgraph:/dgraph
6     ports:
7       - 5080:5080
8       - 6080:6080
9     restart: on-failure
10    command: dgraph zero --my=zero:5080
11
12   alpha:
13     image: dgraph/dgraph:${DGRAPH_VERSION}
14     volumes:
15       - dgraph:/dgraph
16     ports:
17       - 8080:8080
18       - 9080:9080
19     restart: on-failure
20     command: dgraph alpha --my=alpha:7080 --zero=zero:5080 --security \
21               whitelist=0.0.0.0/0 --limit query-edge=10000000
22
23 volumes:
24   dgraph:
```



Листинг 33: Фрагмент конфигурационного файла `convert.yaml` набора данных Elliptic++ Transactions.

```
1  ---
2  files:
3    - name: txs_features.csv
4      declarations:
5        - name: txId
6          type: id
7        - name: Time step
8          type: int
9        - name: Local_feature_1
10         type: float
11        # Остальные объявления...
12      rdfs:
13        - subject: txId
14          predicat: id
15          object: txId
16          cast_object_to: int
17        - subject: txId
18          predicat: Time_step
19          object: Time step
20        - subject: txId
21          predicat: Local_feature_1
22          object: Local_feature_1
23        - subject: txId
24          # Остальные правила...
25    - name: txs_classes.csv
26      declarations:
27        - name: txId
28          type: id
29        - name: class
30          type: int
31      rdfs:
32        - subject: txId
33          predicat: class
34          object: class
35    - name: txs_edgelist.csv
36      declarations:
37        - name: txId1
38          type: id
39        - name: txId2
40          type: id
41      rdfs:
42        - subject: txId1
43          predicat: successors
44          object: txId2
```

Листинг 34: Фрагмент конфигурационного файла `convert.yaml` набора данных MOOC User Actions.

```
1  ---
2  files:
3    - name: mooc_actions.tsv
4      delimiter: "\t"
5      declarations:
6        - name: ACTIONID
7          type: id
8          prefix: a
9        - name: USERID
10         type: id
11         prefix: u
12        - name: TARGETID
13         type: id
14         prefix: t
15        - name: TIMESTAMP
16         type: float
17  rdfs:
18    - subject: ACTIONID
19      predicat: actionId
20      object: ACTIONID
21      cast_object_to: int
22    - subject: USERID
23      predicat: userId
24      object: USERID
25      cast_object_to: int
26    - subject: TARGETID
27      predicat: targetId
28      object: TARGETID
29      cast_object_to: int
30    - subject: ACTIONID
31      predicat: timestamp
32      object: TIMESTAMP
33    - subject: USERID
34      predicat: performs
35      object: ACTIONID
36    - subject: ACTIONID
37      predicat: on
38      object: TARGETID
39  # Остальные файлы...
```

Листинг 35: Конфигурационный файл `convert.yaml` набора данных California Road Network.

```
1 ---
2 files:
3   - name: roadNet-CA.txt
4     delimiter: "\t"
5     comment: "#"
6     declarations:
7       - name: FromNodeId
8         type: id
9       - name: ToNodeId
10        type: id
11   rdfs:
12     - subject: FromNodeId
13       predicat: successors
14       object: ToNodeId
15     - subject: FromNodeId
16       predicat: id
17       object: FromNodeId
18       cast_object_to: int
19     - subject: ToNodeId
20       predicat: id
21       object: ToNodeId
22       cast_object_to: int
```

Листинг 36: Фрагмент конфигурационного файла `convert.yaml` набора данных Stablecoin ERC20 Transactions.

```
1  ---
2  files:
3    - name: token_transfers.csv
4      declarations:
5        - name: from_address
6          type: id
7          prefix: a
8        - name: to_address
9          type: id
10         prefix: a
11        - name: contract_address
12          type: id
13          prefix: a
14        - name: value
15          type: float
16        # Остальные объявления...
17      artificial_declaration:
18        name: transfer
19        type: id
20        prefix: t
21      rdfs:
22        - subject: from_address
23          predicat: address
24          object: from_address
25          cast_object_to: string
26        - subject: to_address
27          predicat: address
28          object: to_address
29          cast_object_to: string
30        - subject: contract_address
31          predicat: address
32          object: contract_address
33          cast_object_to: string
34        - subject: from_address
35          predicat: from
36          object: transfer
37        - subject: transfer
38          predicat: contract
39          object: contract_address
40        - subject: transfer
41          predicat: to
42          object: to_address
43        - subject: transfer
44          predicat: value
45          object: value
46        # Остальные правила...
```

Листинг 37: Структура данных Term пакета rdf программы преобразования и её методы.

```
1  type Decoration uint8
2
3  const (
4      NONE Decoration = iota
5      QUOTES
6      ANGLE_BRACKETS
7  )
8
9  type Term struct {
10     val string
11     dec Decoration
12 }
13
14 func NewTerm(val string, dec Decoration) *Term {
15     return &Term{val: val, dec: dec}
16 }
17
18 func (term *Term) String() string {
19     switch term.dec {
20     case NONE:
21         return term.val
22     case QUOTES:
23         return "`" + term.val + "`"
24     case ANGLE_BRACKETS:
25         return "<" + term.val + ">"
26     }
27
28     return ""
29 }
```

Листинг 38: Структура данных Facet пакета rdf программы преобразования и её методы.

```
1  type Facet struct {
2     key string
3     term *Term
4  }
5
6  func NewFacet(key string, term *Term) *Facet {
7     return &Facet{key: key, term: term}
8  }
```

Листинг 39: Структура данных Rdf пакета rdf программы преобразования и её методы.

```
1 type Rdf struct {
2     subject *Term
3     predicat *Term
4     object *Term
5     facets []*Facet
6 }
7
8 func NewRdf(
9     subject *Term,
10    predicat *Term,
11    object *Term,
12    facets []*Facet,
13 ) *Rdf {
14     return &Rdf{
15         subject: subject,
16         predicat: predicat,
17         object: object,
18         facets: facets,
19     }
20 }
21
22 func (rdf *Rdf) String() string {
23     facets := ""
24     if len(rdf.facets) > 0 {
25         facets = "("
26         for i, facet := range rdf.facets {
27             if i > 0 {
28                 facets += ", "
29             }
30             facets += facet.key + "=" + facet.term.String()
31         }
32         facets += ")"
33     }
34
35     return fmt.Sprintf(
36         "%s %s %s %s.",
37         rdf.subject.String(),
38         rdf.predicat.String(),
39         rdf.object.String(),
40         facets,
41     )
42 }
43
44 func (rdf *Rdf) Stringln() string {
45     return rdf.String() + "\n"
46 }
```

Листинг 40: Функция ProcessDatasets пакета converter программы преобразования.

```
1 func ProcessDatasets(datasetsPath string) error {
2     entires, err := os.ReadDir(datasetsPath)
3     if err != nil {
4         return err
5     }
6
7     g := new(errgroup.Group)
8     for _, entry := range entires {
9         if !entry.IsDir() {
10             continue
11         }
12
13         datasetName := entry.Name()
14         g.Go(func() error {
15             err := errors.Wrap(
16                 ProcessDataset(filepath.Join(datasetsPath,
17                     datasetName)),
18                 datasetName,
19             )
20             if err != nil {
21                 log.Println("Error while processing
22                     dataset", err)
23             } else {
24                 log.Println("Successfully processed
25                     dataset", datasetName)
26             }
27             return err
28         })
29     }
30
31     return g.Wait()
32 }
```

Листинг 41: Функция ProcessDataset пакета converter программы преобразования.

```
1 func ProcessDataset(datasetPath string) error {
2     const configName = "convert.yml"
3
4     config, err := os.Open(filepath.Join(datasetPath, configName))
5     if err != nil {
6         return err
7     }
8     defer config.Close()
9
10    decoder := yaml.NewDecoder(config)
11    decoder.KnownFields(true)
12
13    var ds dataset
14    if err = decoder.Decode(&ds); err != nil {
15        return err
16    }
17
18    return ds.process(datasetPath)
19 }
```



Листинг 42: Метод process структуры dataset пакета converter программы преобразования.

```
1 func (ds *dataset) process(datasetPath string) error {
2     const (
3         outputName = "output.rdf"
4         sourcesName = "sources"
5     )
6
7     output, err := os.Create(filepath.Join(datasetPath, outputName))
8     if err != nil {
9         return err
10    }
11    defer output.Close()
12
13    sourcesPath := filepath.Join(datasetPath, sourcesName)
14    entitiesFacets := make(map[string]entityFacets)
15
16    for _, f := range ds.Files {
17        if err = f.process(entitiesFacets, output, sourcesPath);
18            err != nil {
19                return errors.Wrap(err, f.Name)
20            }
21    }
22
23    return nil
24 }
```

Листинг 43: Метод process структуры file пакета converter программы преобразования.

```
1 func (f *file) process(  
2     entitiesFacets map[string]entityFacets,  
3     output *os.File,  
4     sourcesPath string,  
5 ) error {  
6     schema, err := f.validate()  
7     if err != nil {  
8         return err  
9     }  
10    source, err := os.Open(filepath.Join(sourcesPath, f.Name))  
11    if err != nil {  
12        return err  
13    }  
14    defer source.Close()  
15    reader := csv.NewReader(source)  
16    err = f.adjustReader(reader)  
17    if err != nil {  
18        return err  
19    }  
20    headers, err := reader.Read()  
21    if err != nil {  
22        return err  
23    }  
24    headers = append(headers, f.ArtificialDeclaration.Name)  
25    indices := make(map[string]uint)  
26    for i, header := range headers {  
27        indices[header] = uint(i)  
28    }  
29    for artificialId := 0; true; artificialId++ {  
30        record, err := reader.Read()  
31        if err == io.EOF {  
32            break  
33        }  
34        if err != nil {  
35            return err  
36        }  
37        record = append(record, fmt.Sprintf(artificialId))  
38        f.saveFacets(entitiesFacets, record, schema, indices)  
39        err = f.writeRdfs(output, entitiesFacets, record, schema,  
40                           indices)  
41        if err != nil {  
42            return err  
43        }  
44    }  
45    return nil  
}
```

Листинг 44: Функция saveFacets пакета converter программы преобразования.

```
1 func (file *file) saveFacets(  
2     entitiesFacets map[string]entityFacets,  
3     record []string,  
4     schema map[string]schemaType,  
5     indices map[string]uint,  
6 ) {  
7     for _, rule := range file.EntityFacets {  
8         addFacet(  
9             entitiesFacets,  
10            makeEntityKey(  
11                schema[rule.Id].prefix,  
12                record[indices[rule.Id]],  
13            ),  
14            rule.Key,  
15            rdf.NewTerm(  
16                record[indices[rule.Value]],  
17                facetDecorations[schema[rule.Value].dt],  
18            ),  
19        )  
20    }  
21 }
```

Листинг 45: Фрагмент функции writeRdfs пакета converter программы преобразования.

```
1 func (f *file) writeRdfs(  
2     output *os.File,  
3     entitiesFacets map[string]entityFacets,  
4     record []string,  
5     schema map[string]schemaType,  
6     indices map[string]uint,  
7 ) error {  
8     for _, rule := range f.Rdfs {  
9         objectIndex := indices[rule.Object]  
10        if record[objectIndex] == "" {  
11            continue  
12        }  
13        subject := makeBlankNode(  
14            makeEntityKey(schema[rule.Subject].prefix,  
15                record[indices[rule.Subject]]),  
16        )  
17        objectType := schema[rule.Object]  
18        if rule.CastObjectTo != "" {  
19            objectType.dt = dataTypes[rule.CastObjectTo]  
20        }  
21        var object string  
22        if objectType.dt == idType {  
23            object = makeBlankNode(  
24                makeEntityKey(objectType.prefix,  
25                    record[objectIndex]),  
26            )  
27        } else {  
28            object = record[objectIndex]  
29        }  
30        // Для краткости, опущен код наполнения среза facets  
31        r := rdf.NewRdf(  
32            rdf.NewTerm(subject, rdf.NONE),  
33            rdf.NewTerm(rule.Predicat, rdf.ANGLE_BRACKETS),  
34            rdf.NewTerm(object,  
35                termDecorations[objectType.dt]),  
36            facets,  
37        )  
38        _, err := output.WriteString(r.Stringln())  
39        if err != nil {  
40            return err  
41        }  
42    }  
43    return nil  
44 }
```

Листинг 46: Функция программы `benchmark.go` формирования соединения с сервером Dgraph.

```
1 func getDgraphClient(host string, port uint) (*dgo.Dgraph, cancelFunc) {
2     conn, err := grpc.Dial(
3         formTarget(host, port),
4         grpc.WithDefaultCallOptions(
5             grpc.MaxCallRecvMsgSize(grpcMaxRecieveBytes),
6         ),
7         grpc.WithTransportCredentials(insecure.NewCredentials()),
8     )
9     if err != nil {
10         log.Fatal(err)
11     }
12
13     dc := api.NewDgraphClient(conn)
14     return dgo.NewDgraphClient(dc), func() {
15         if err := conn.Close(); err != nil {
16             log.Fatal(err)
17         }
18     }
19 }
```

Листинг 47: Функция программы benchmark.go выполнения запроса и оценки ресурсов, затраченных на его выполнение.

```
1 func performQuery(  
2     dg *dgo.Dgraph,  
3     query string,  
4     duration time.Duration,  
5     printRespond bool,  
6 ) {  
7     memoryBefore := memory.FreeMemory()  
8     var memoryMinimum uint64 = math.MaxUint64  
9     quit := make(chan bool)  
10    go func() {  
11        for {  
12            select {  
13            case <-quit:  
14                break  
15            default:  
16                freeMemory := memory.FreeMemory()  
17                if freeMemory < memoryMinimum {  
18                    memoryMinimum = freeMemory  
19                }  
20                time.Sleep(duration)  
21            }  
22        }  
23    }()  
24  
25    txn := dg.NewReadOnlyTxn().BestEffort()  
26    resp, err := txn.Query(context.Background(), query)  
27    if err != nil {  
28        log.Fatal(err)  
29    }  
30    quit <- true  
31  
32    log.Printf("Free RAM before the execution: %d bytes.\n",  
33        memoryBefore)  
34    log.Printf("Free RAM minimum during the execution: %d bytes.\n",  
35        memoryMinimum)  
36    log.Printf("Free RAM consumption: %d bytes.\n",  
37        memoryBefore-memoryMinimum)  
38    log.Printf("Request latency: %d nanoseconds.\n",  
39        resp.Latency.GetTotalNs())  
40  
41    if printRespond {  
42        prettyPrintJson(resp.GetJson())  
43    }  
44 }
```