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

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[1]. 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 направленные, 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.

[1] 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 раздельные данные, сервер GraphQL предоставляет единственнную, 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 валидидирует 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 даных 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 ассоцированы 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[1]:

(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: временна́я 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);

[1] 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, использующиеся 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.

```
id: int @index(int) .
Time_step: int .
Local_feature_1: float .
class: int @index(int) .
successors: [uid] @reverse .
# 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 (a) 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.

```
{
  var(func: has(successors)) {
    succCount as count(successors)
  }

  result(func: uid(succCount), orderdesc: val(succCount), first: 1) {
    id
    val(succCount)
  }
}
```

В листинге 3 приводится запрос (a). 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 (a) для набора данных Elliptic++ Transactions.

```
{
  result(func: eq(id, 2984918)) @recurse(depth: 50) {
    uid
    successors @filter(lt(Time_step, 25))
  }
}
```

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.

```
{
  result(func: eq(class, 3)) {
    uid
  }
}
```

Запрос (c) 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.

```
{
  var(func: has(~successors)) {
    predCount as count(~successors)
  }

  result(func: uid(predCount), orderdesc: val(predCount), first: 1) {
    id
    val(predCount)
  }
}
```

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

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

```
{
  transactions as var(func: eq(class, 3)) @cascade(successors) {
    successors @filter(eq(id, 43388675))
  }

  var(func: uid(transactions)) {
    succCount as count(successors)
    predCount as count(~successors)
    degree as math(succCount + predCount)
  }

  result(func: ge(val(degree), 10)) {
    uid
  }
}
```

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.

```
{
  var(func: has(id)) {
    successors @filter(le(Local_feature_1, -0.15)) {
      succFeat as Local_feature_1
    }
    succAvgFeat as avg(val(succFeat))
  }

  var(func: has(id)) {
    ~successors @filter(le(Local_feature_1, -0.15)) {
      predFeat as Local_feature_1
    }
    predAvgFeat as avg(val(predFeat))
  }

  result(func: has(id)) {
    uid
    avgFeat : math((succAvgFeat + predAvgFeat) / 2)
  }
}
```

Запрос (e) В листинге 8 приводится запрос (e). Выполняется поик 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.

```
{
  start as var(func: eq(id, 2984918))
  end as var(func: eq(id, 43388675))
  path as shortest(from: uid(start), to: uid(end)) {
    successors @filter(gt(Time_step, 25))
  }

  result(func: uid(path)) {
    uid
  }
}
```

— 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.

```
userId: int @index(int) .
actionId: int @index(int) .
targetId: int @index(int) .
performs: [uid] @reverse .
on: uid @reverse .
label: int @index(int) .
feature0: float .
feature1: float .
feature2: float .
feature3: float .
timestamp: float .
```

2.2.3 Запросы

Запрос (a) 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.

```
{
  var(func: has(performs)) {
    actCount as count(performs)
  }

  result(func: uid(actCount), orderdesc: val(actCount), first: 2) {
    userId
    val(actCount)
  }
}
```

В листинге 11 приводится запрос (a). 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.

```
{
  result(func: eq(userId, 1181)) @recurse(depth: 5) @ignorereflex {
    uid
    performs @filter(ge(timestamp, 1_500_000))
    on
    ~on @filter(ge(timestamp, 1_500_000))
    ~performs
  }
}
```

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.

```
{
  result(func: eq(label, 0)) {
    uid
  }
}
```

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.

```
{
  var(func: has(targetId)) {
    actCount as count(~on)
  }

  result(func: uid(actCount), orderdesc: val(actCount), first: 1) {
    targetId
    val(actCount)
  }
}
```

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

```
{
  actions as var(func: eq(label, 0)) @cascade(on) {
    on @filter(eq(targetId, 8))
  }

  var(func: uid(actions)) {
    targetsCount as count(on)
    usersCount as count(~performs)
    degree as math(targetsCount + usersCount)
  }

  result(func: eq(val(degree), 2)) {
    uid
  }
}
```

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

Запрос (e) В листинге 16 приводится запрос (e). 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.

```
{
  var(func: has(userId)) {
    performs @filter(ge(feature1, 0)) {
      userFeat as feature1
    }
    avgUserFeat as avg(val(userFeat))
  }

  var(func: has(targetId)) {
    ~on @filter(ge(feature1, 0)) {
      targetFeat as feature1
    }
    avgTargetFeat as avg(val(targetFeat))
  }

  users(func: has(userId)) {
    uid
    val(avgUserFeat)
  }

  targets(func: has(targetId)) {
    uid
    val(avgTargetFeat)
  }
}
```

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

```
{
  start as var(func: eq(userId, 1181))
  end as var(func: eq(userId, 1686))
  path as shortest(from: uid(start), to: uid(end)) @ignorereflex {
    performs @filter(ge(timestamp, 1_500_000))
    ~performs
    on
    ~on @filter(ge(timestamp, 1_500_000))
  }

  result(func: uid(path)) {
    uid
  }
}
```

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.

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

2.3.3 Запросы

Запрос (a) 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 (a). 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.

```
{
  var(func: has(successors)) {
    succCount as count(successors)
  }

  result(func: uid(succCount), orderdesc: val(succCount), first: 2) {
    id
    val(succCount)
  }
}
```

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

```
{
  result(func: eq(id, 562818)) @recurse(depth: 50) @ignorereflex {
    uid
    successors @filter(lt(id, 562818))
  }
}
```

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

```
{
  result(func: eq(id, 562818)) {
    uid
  }
}
```

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 (c) для набора данных California Road Network.

```
{
  nodes as var(func: eq(id, 562818)) @cascade(successors) {
    successors @filter(eq(id, 562826))
  }

  var(func: uid(nodes)) {
    degree as count(successors)
  }

  result(func: eq(val(degree), 12)) {
    uid
  }
}
```

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

```
{
  var(func: has(id)) {
    successors @filter(lt(id, 562818)) {
      feature as id
    }
    avgFeature as avg(val(feature))
  }

  result(func: uid(avgFeature)) {
    uid
    val(avgFeature)
  }
}
```

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

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

```
{
  start as var(func: eq(id, 562818))
  end as var(func: eq(id, 521168))
  path as shortest(from: uid(start), to: uid(end)) {
    successors @filter(lt(id, 562818))
  }

  result(func: uid(path)) {
    uid
  }
}
```

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.

```
address: string @index(hash) .
from: [uid] @reverse .
to: uid @reverse .
contract: uid .
block_number: int .
transaction_index: int .
time_stamp: int .
value: float @index(float) .
```

2.4.3 Запросы

Запрос (a) В листинге 25 приводится вспомогательный запрос для определения адреса отправителя наибольшего числа переводов. Соответствующий предикат `address` равен `0x74de5d4fcbf63e00296fd95d33236b9794016631`, а количество переводов — 147 437.

В листинге 26 приводится запрос (a). 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.

```
{
  var(func: has(from)) {
    tranCount as count(from)
  }

  result(func: uid(tranCount), orderdesc: val(tranCount), first: 1) {
    address
    val(tranCount)
  }
}
```

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

```
{
  result(func: eq(address, "0x74de5d4fcbf63e00296fd95d33236b9794016631"))
      @recurse(depth: 3) {
    uid
    from @filter(ge(time_stamp, 1653000000))
    to
  }
}
```

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

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

```
{
  result(func: eq(value, 1000)) {
    uid
  }
}
```

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.

```
{
  var(func: has(~contract)) {
    tranCount as count(~contract)
  }

  result(func: uid(tranCount), orderdesc: val(tranCount), first: 1) {
    address
    val(tranCount)
  }
}
```

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

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

```
{
  transfers as var(func: eq(value, 1000)) @cascade(contract) {
    contract @filter(eq(address,
        "0xdac17f958d2ee523a2206206994597c13d831ec7"))
  }

  var(func: uid(transfers)) {
    fromAddresses as count(~from)
    toAddresses as count(to)
    degree as math(fromAddresses + toAddresses)
  }

  result(func: eq(val(degree), 2)) {
    uid
  }
}
```

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.

```
{
  var(func: has(address)) {
    from @filter(ge(value, 1000)) {
      fromValue as value
    }
    avgFromValue as avg(val(fromValue))
  }

  var(func: has(address)) {
    ~to @filter(ge(value, 1000)) {
      toValue as value
    }
    avgToValue as avg(val(toValue))
  }

  result(func: has(address)) {
    uid
    avgValue : math((avgFromValue + avgToValue) / 2)
  }
}
```

Запрос (e) В листинге 31 приводится запрос (e). 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 (e) для набора данных Stablecoins ERC20 Transactions.

```
{
  start as var(func: eq(address,
      "0x74de5d4fcbf63e00296fd95d33236b9794016631"))
  end as var(func: eq(address,
      "0x0000000000000000000000000000000000000000"))
  path as shortest(from: uid(start), to: uid(end)) {
    from @filter(ge(time_stamp, 1653000000))
    to
  }

  result(func: uid(path)) {
    uid
  }
}
```

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. Обрабатываемые 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 корневым директориям 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) стобца. 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`. Незадействованные 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 можны 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 `enitity_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 42 приложения А приводится фрагмент конфигурационного файла набора Elliptic++ Transactions, в листинге 43 приложения А — фрагмент конфигурационного файла набора MOOC User Actions, в листинге 44 приложения А — конфигурационный файл набора California Road Network и в листинге 45 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`. Стуктура 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_declaraion` — 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 33, 34, 35 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 36 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 37 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 соответветствующий объект, а иначе — `nil`.

Внутренняя логика Обработчик структуры `Dataset` (листинг 38 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 39 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 40 приложения А) и выписывания литералов RDF (листинг 41 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 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 умолчнанию, `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 датасетов

Прежде чем представить результаты выполнения запросов, приведём сводную статистику по датасетам.

4.1.1 Вершины и рёбра

В таблице 1 содержится итоговая информация о вершинах и рёберах датасетов. Для датасета Elliptic++ Transactions эти данные остались прежними: число вершин есть число транзакций, число рёбер — число потоков транзакций. Для датасета MOOC User Actions число вершин складывается из числа пользователей, действий и онлайн-курсов, а число рёбер равняется числу действий, домноженному на 2, поскольку каждое действие соединяет одного пользователя и один курс. Для датасета California Road Network данные также не изменились: число вершин — число узлов, число рёбер — число связей. Для датасета Stablecoin ERC20 Transactions число вершин складывается из числа адресов, полученного вспомогательным запросом, и числа передач; число рёбер есть трижды число передач, поскольку одна передача связывает адрес отправителя, адрес получателя и адрес контракта.

Таблица 1 — Вершины и рёбра датасетов.

Датасет	Вершины	Рёбра
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 содержатся данные (в мегабайтах) о занимаемом дисковом пространстве исходными файлами датасетов и данными кластера Dgraph после импорта датасетов. Данные для исходных файлов тривиально получены суммированием размера файлов. Данные кластера сняты с директории `dgraph/p` Docker-контейнера, который разделяют Alpha- и Zero-узлы кластера.

Таблица 2 — Занимаемое датасетами дисковое пространство, МБ.

Датасет	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 показателей следующих разделов все запросы выполнялись в одинаковых условиях и по 5 раз; среднее значение результатов выполнения считается итоговым значением.

Характеристики устройства, на котором производилось оценивание:

— 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 содержатся данные о времени выполнения запросов (в миллисекундах). Под временем понимается общее время выполнения запроса, включающее синтаксический разбор текста запроса, непосредственно выполнение и сериализацию результатов выполнения.

Таблица 3 — Время выполнения запросов, мс.

Запрос / Датасет	(a)	(b)	(c)	(d)	(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

4.2.2 Оперативная память

В таблице 4 содержатся данные о потреблении оперативной памяти (в мегабайтах) во время выполнения запросов. Показатели получены, как разность сво-

бодной оперативной памяти до выполнения запроса и её минимального значения во время выполнения.

Таблица 4 — Потребление ОЗУ при выполнении запросов, МБ.

Датасет \ Запрос	(a)	(b)	(c)	(d)	(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

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы поставленные цели были достигнуты.

Важную и большую часть работы занимает программа преобразования, которая по специальному конфигурационному файлу преобразует исходные файлы CSV-датасетов 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 применяются сложные алгоритмы на графах.

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 кластера Dgraph

```
services:
  zero:
    image: dgraph/dgraph:${DGRAPH_VERSION}
    volumes:
      - dgraph:/dgraph
    ports:
      - 5080:5080
      - 6080:6080
    restart: on-failure
    command: dgraph zero --my=zero:5080

  alpha:
    image: dgraph/dgraph:${DGRAPH_VERSION}
    volumes:
      - dgraph:/dgraph
    ports:
      - 8080:8080
      - 9080:9080
    restart: on-failure
    command: dgraph alpha --my=alpha:7080 --zero=zero:5080 --security \
             whitelist=0.0.0.0/0 --limit query-edge=10000000

volumes:
  dgraph:
```

Листинг 33: Структура данных `Term` пакета `rdf` и её методы.

```
type Decoration uint8

const (
        NONE Decoration = iota
        QUOTES
        ANGLE_BRACKETS
)

type Term struct {
        val string
        dec Decoration
}

func NewTerm(val string, dec Decoration) *Term {
        return &Term{val: val, dec: dec}
}

func (term *Term) String() string {
        switch term.dec {
        case NONE:
                return term.val
        case QUOTES:
                return `"` + term.val + `"`
        case ANGLE_BRACKETS:
                return "<" + term.val + ">"
        }

        return ""
}
```

Листинг 34: Структура данных `Facet` пакета `rdf` и её методы.

```
type Facet struct {
        key  string
        term *Term
}

func NewFacet(key string, term *Term) *Facet {
        return &Facet{key: key, term: term}
}
```

Листинг 35: Структура данных Rdf пакета rdf и её методы.

```
type Rdf struct {
        subject  *Term
        predicat *Term
        object   *Term
        facets   []*Facet
}

func NewRdf(
        subject *Term,
        predicat *Term,
        object *Term,
        facets []*Facet,
) *Rdf {
        return &Rdf{
                subject:  subject,
                predicat: predicat,
                object:   object,
                facets:   facets,
        }
}

func (rdf *Rdf) String() string {
        facets := ""
        if len(rdf.facets) > 0 {
                facets = "("
                for i, facet := range rdf.facets {
                        if i > 0 {
                                facets += ", "
                        }
                        facets += facet.key + "=" + facet.term.String()
                }
                facets += ") "
        }

        return fmt.Sprintf(
                "%s %s %s %s.",
                rdf.subject.String(),
                rdf.predicat.String(),
                rdf.object.String(),
                facets,
        )
}

func (rdf *Rdf) Stringln() string {
        return rdf.String() + "\n"
}
```

Листинг 36: Функция ProcessDatasets пакета converter.

```
func ProcessDatasets(datasetsPath string) error {
        entires, err := os.ReadDir(datasetsPath)
        if err != nil {
                return err
        }

        g := new(errgroup.Group)
        for _, entry := range entires {
                if !entry.IsDir() {
                        continue
                }

                datasetName := entry.Name()
                g.Go(func() error {
                        err := errors.Wrap(
                                ProcessDataset(filepath.Join(datasetsPath,
                                    datasetName)),
                                datasetName,
                        )
                        if err != nil {
                                log.Println("Error while processing
                                    dataset", err)
                        } else {
                                log.Println("Successfully processed
                                    dataset", datasetName)
                        }
                        return err
                })
        }

        return g.Wait()
}
```

Листинг 37: Функция `ProcessDataset` пакета `converter`.

```
func ProcessDataset(datasetPath string) error {
	const configName = "convert.yml"

	config, err := os.Open(filepath.Join(datasetPath, configName))
	if err != nil {
		return err
	}
	defer config.Close()

	decoder := yaml.NewDecoder(config)
	decoder.KnownFields(true)

	var ds dataset
	if err = decoder.Decode(&ds); err != nil {
		return err
	}

	return ds.process(datasetPath)
}
```

Листинг 38: Метод process структуры dataset пакета converter.

```
func (ds *dataset) process(datasetPath string) error {
        const (
                outputName  = "output.rdf"
                sourcesName = "sources"
        )

        output, err := os.Create(filepath.Join(datasetPath, outputName))
        if err != nil {
                return err
        }
        defer output.Close()

        sourcesPath := filepath.Join(datasetPath, sourcesName)
        entitiesFacets := make(map[string]entityFacets)

        for _, f := range ds.Files {
                if err = f.process(entitiesFacets, output, sourcesPath);
                    err != nil {
                        return errors.Wrap(err, f.Name)
                }
        }

        return nil
}
```

Листинг 39: Метод `process` структуры `file` пакета `converter`.

```
func (f *file) process(
        entitiesFacets map[string]entityFacets,
        output *os.File,
        sourcesPath string,
) error {
        schema, err := f.validate()
        if err != nil {
                return err
        }
        source, err := os.Open(filepath.Join(sourcesPath, f.Name))
        if err != nil {
                return err
        }
        defer source.Close()
        reader := csv.NewReader(source)
        err = f.adjustReader(reader)
        if err != nil {
                return err
        }
        headers, err := reader.Read()
        if err != nil {
                return err
        }
        headers = append(headers, f.ArtificialDeclaration.Name)
        indices := make(map[string]uint)
        for i, header := range headers {
                indices[header] = uint(i)
        }
        for artificialId := 0; true; artificialId++ {
                record, err := reader.Read()
                if err == io.EOF {
                        break
                }
                if err != nil {
                        return err
                }
                record = append(record, fmt.Sprint(artificialId))
                f.saveFacets(entitiesFacets, record, schema, indices)
                err = f.writeRdfs(output, entitiesFacets, record, schema,
                    indices)
                if err != nil {
                        return err
                }
        }
        return nil
}
```

Листинг 40: Функция saveFacets пакета converter.

```
func (file *file) saveFacets(
        entitiesFacets map[string]entityFacets,
        record []string,
        schema map[string]schemaType,
        indices map[string]uint,
) {
        for _, rule := range file.EntityFacets {
                addFacet(
                        entitiesFacets,
                        makeEntityKey(
                                schema[rule.Id].prefix,
                                record[indices[rule.Id]],
                        ),
                        rule.Key,
                        rdf.NewTerm(
                                record[indices[rule.Value]],
                                facetDecorations[schema[rule.Value].dt],
                        ),
                )
        }
}
```

Листинг 41: Фрагмент функции `writeRdfs` пакета `converter`.

```
func (f *file) writeRdfs(
        output *os.File,
        entitiesFacets map[string]entityFacets,
        record []string,
        schema map[string]schemaType,
        indices map[string]uint,
) error {
        for _, rule := range f.Rdfs {
                objectIndex := indices[rule.Object]
                if record[objectIndex] == "" {
                        continue
                }
                subject := makeBlankNode(
                        makeEntityKey(schema[rule.Subject].prefix,
                            record[indices[rule.Subject]]),
                )
                objectType := schema[rule.Object]
                if rule.CastObjectTo != "" {
                        objectType.dt = dataTypes[rule.CastObjectTo]
                }
                var object string
                if objectType.dt == idType {
                        object = makeBlankNode(
                                makeEntityKey(objectType.prefix,
                                    record[objectIndex]),
                        )
                } else {
                        object = record[objectIndex]
                }
                // Для краткости, опущен код наполнения среза facets
                r := rdf.NewRdf(
                        rdf.NewTerm(subject, rdf.NONE),
                        rdf.NewTerm(rule.Predicat, rdf.ANGLE_BRACKETS),
                        rdf.NewTerm(object,
                            termDecorations[objectType.dt]),
                        facets,
                )
                _, err := output.WriteString(r.Stringln())
                if err != nil {
                        return err
                }
        }
        return nil
}
```

Листинг 42: Фрагмент конфигурационного файла датасета Elliptic++ Transactions.

```yaml
---
files:
  - name: txs_features.csv
    declarations:
      - name: txId
        type: id
      - name: Time step
        type: int
      - name: Local_feature_1
        type: float
      # Остальные объявления...
    rdfs:
      - subject: txId
        predicat: id
        object: txId
        cast_object_to: int
      - subject: txId
        predicat: Time_step
        object: Time step
      - subject: txId
        predicat: Local_feature_1
        object: Local_feature_1
      - subject: txId
      # Остальные правила...
  - name: txs_classes.csv
    declarations:
      - name: txId
        type: id
      - name: class
        type: int
    rdfs:
      - subject: txId
        predicat: class
        object: class
  - name: txs_edgelist.csv
    declarations:
      - name: txId1
        type: id
      - name: txId2
        type: id
    rdfs:
      - subject: txId1
        predicat: successors
        object: txId2
```

Листинг 43: Фрагмент конфигурационного файла датасета MOOC User Actions.

```yaml
---
files:
  - name: mooc_actions.tsv
    delimiter: "\t"
    declarations:
      - name: ACTIONID
        type: id
        prefix: a
      - name: USERID
        type: id
        prefix: u
      - name: TARGETID
        type: id
        prefix: t
      - name: TIMESTAMP
        type: float
    rdfs:
      - subject: ACTIONID
        predicat: actionId
        object: ACTIONID
        cast_object_to: int
      - subject: USERID
        predicat: userId
        object: USERID
        cast_object_to: int
      - subject: TARGETID
        predicat: targetId
        object: TARGETID
        cast_object_to: int
      - subject: ACTIONID
        predicat: timestamp
        object: TIMESTAMP
      - subject: USERID
        predicat: performs
        object: ACTIONID
      - subject: ACTIONID
        predicat: on
        object: TARGETID
    # Остальные файлы...
```

Листинг 44: Конфигурационный файл датасета California Road Network.

```
---
files:
  - name: roadNet-CA.txt
    delimiter: "\t"
    comment: "#"
    declarations:
      - name: FromNodeId
        type: id
      - name: ToNodeId
        type: id
    rdfs:
      - subject: FromNodeId
        predicat: successors
        object: ToNodeId
      - subject: FromNodeId
        predicat: id
        object: FromNodeId
        cast_object_to: int
      - subject: ToNodeId
        predicat: id
        object: ToNodeId
        cast_object_to: int
```

Листинг 45: Фрагмент конфигурационного файла датасета Stablecoin ERC20 Transactions.

```yaml
---
files:
  - name: token_transfers.csv
    declarations:
      - name: from_address
        type: id
        prefix: a
      - name: to_address
        type: id
        prefix: a
      - name: contract_address
        type: id
        prefix: a
      - name: value
        type: float
      # Остальные объявления...
    artificial_declaration:
      name: transfer
      type: id
      prefix: t
    rdfs:
      - subject: from_address
        predicat: address
        object: from_address
        cast_object_to: string
      - subject: to_address
        predicat: address
        object: to_address
        cast_object_to: string
      - subject: contract_address
        predicat: address
        object: contract_address
        cast_object_to: string
      - subject: from_address
        predicat: from
        object: transfer
      - subject: transfer
        predicat: contract
        object: contract_address
      - subject: transfer
        predicat: to
        object: to_address
      - subject: transfer
        predicat: value
        object: value
      # Остальные правила...
```

Листинг 46: Функция формирования соединения с сервером Dgraph и получения клиента.

```
func getDgraphClient(host string, port uint) (*dgo.Dgraph, cancelFunc) {
        conn, err := grpc.Dial(
                formTarget(host, port),
                grpc.WithDefaultCallOptions(
                        grpc.MaxCallRecvMsgSize(grpcMaxRecieveBytes),
                ),
                grpc.WithTransportCredentials(insecure.NewCredentials()),
        )
        if err != nil {
                log.Fatal(err)
        }

        dc := api.NewDgraphClient(conn)
        return dgo.NewDgraphClient(dc), func() {
                if err := conn.Close(); err != nil {
                        log.Fatal(err)
                }
        }
}
```

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

```
func performQuery(
        dg *dgo.Dgraph,
        query string,
        duration time.Duration,
        printRespond bool,
) {
        memoryBefore := memory.FreeMemory()
        var memoryMinimum uint64 = math.MaxUint64
        quit := make(chan bool)
        go func() {
                for {
                        select {
                        case <-quit:
                                break
                        default:
                                freeMemory := memory.FreeMemory()
                                if freeMemory < memoryMinimum {
                                        memoryMinimum = freeMemory
                                }
                                time.Sleep(duration)
                        }
                }
        }()

        txn := dg.NewReadOnlyTxn().BestEffort()
        resp, err := txn.Query(context.Background(), query)
        if err != nil {
                log.Fatal(err)
        }
        quit <- true

        log.Printf("Free RAM before the execution: %d bytes.\n",
            memoryBefore)
        log.Printf("Free RAM minimum during the execution: %d bytes.\n",
            memoryMinimum)
        log.Printf("Free RAM consumption: %d bytes.\n",
            memoryBefore-memoryMinimum)
        log.Printf("Request latency: %d nanoseconds.\n",
            resp.Latency.GetTotalNs())

        if printRespond {
                prettyPrintJson(resp.GetJson())
        }
}
```