Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Терехов Арсений Константинович

Разработка матричного алгоритма поиска путей с контекстно-свободными ограничениями для RedisGraph

Выпускная квалификационная работа бакалавра

Научный руководитель:
к.ф.-м.н., доцент кафедры информатики СПбГУ Григорьев С. В.

Рецензент:
к.ф.-м.н., Инженер-программист ООО "ИнтеллиДжей Лабс" Березун Д. А.

Санкт-Петербург
2020

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Software Engineering

Terekhov Arseniy

Development of matrix-based context-free path querying algorithm for RedisGraph

Bachelor's Thesis

Scientific supervisor:
PhD, Assistant Professor Semyon Grigorev

Reviewer:
PhD, Software engineer LCC "IntelliJ Labs" Daniil Berezun

Saint-Petersburg
2020

Оглавление

Введение

Поиск путей в графе с ограничением в виде формальных языков [?] — это задача, в которой формальные языки используются для задания множества искомых путей. В таком подходе каждый путь соответствует слову, состоящему из меток его рёбер, а ограничением на путь является принадлежность соответствующего ему слова некоторому заданному формальному языку.

В качестве класса формальных языков по иерархии Хомского наибольший интерес представляют контекстно-свободные языки. В отличие от регулярных они обладают большей выразительностью. Поэтому в задаче поиска путей контекстно-свободные ограничения позволяют задавать более сложные отношения между вершинами. Так, например, важный класс запросов поиска вершин, лежащих на одном уровне иерархии [?], задаётся только контекстно-свободными, но не регулярными ограничениями. Запросы такого вида, как и другие запросы с контекстно-свободными ограничениями имеют широкое применение в биоинформатике [?] и при обработке rdf-файлов [?].

Наиболее удобным и подходящим инструментом для работы с граф-структурированными данными являются графовые базы данных. Так же как и реляционные, графовые базы данных поддерживают свой язык запросов. С его помощью графовые базы данных позволяют решать вышеупомянутую задачу поиска путей. Но ограничения на пути, которые поддерживается в наиболее распространённых базах данных, являются в лучшем случае регулярными.

Отсутствие поддержки контекстно-свободных ограничений в графовых базах данных, во-первых, сильно ограничивает выразительность языка запросов. Во-вторых, при необходимости в более сложных запросах разработчикам приходится самим писать алгоритмы, решающие задачу контекстно-свободной достижимости для их частного случая. Так, например, Хуэй Мяо и др. [?] разработали систему хранения и отслеживания версий артефактов, возникающих при научных работах. Вся информация про артефакты хранилась в графовой базе данных.

При этом при разработке возникла потребность в выполнении запросов с контекстно-свободными ограничениями для выявления взаимоотношений между различными версиями различных артефактов. Это и послужило началом статьи [?], в которой приводятся алгоритмы решения частных запросов.

В недавнем исследовании Йохем Куйперс и др. [?] произвели сравнительный анализ наиболее известных алгоритмов поиска путей с контекстно-свободными ограничениями. Алгоритмы запускались на графах, находящихся в хранилище графовой базы данных Neo4j. По результатам исследования было показано, что в контексте Neo4j алгоритмы обладают большим временем работы, и поэтому дальнейшая работа по расширению языка запросов прекратилась. При этом Рустам Азимов [?] предоставил матричный алгоритм и его реализацию, которая работает за разумное время на реальных данных. Но, так как алгоритм был реализован вне контекста базы данных, его результат приняли недостаточно показательным. Поэтому вопрос о реализуемости запросов с контекстно-свободными ограничениями в графовых базах данных, а соответственно и о возможности расширения языка запросов для их поддержки остаётся открытым.

1. Постановка задачи

Целью данной работы является полная поддержка запросов с контекстно-свободными ограничениями для графовой базы данных. А именно, необходимо предоставить пользователю возможность формулировать запросы с контекстно-свободными ограничениями в терминах одного из существующих стандартных языков запросов и исполнять их в графовой базе данных за приемлемое время. Для достижения этой цели были поставлены следующие задачи.

- Выполнить обзор существующих реализаций поддержки запросов с контекстно-свободными ограничениями в графовых базах данных. В результате обзора необходимо выбрать наиболее перспективный с точки зрения производительности алгоритм решения задачи контекстно-свободной достижимости и подходящую для его интеграции базу данных. При выборе базы данных необходимо учитывать как возможность интеграции выбранного алгоритма, так и возможность поддержки одного из стандартных языков запросов, позволяющего выражать контекстно-свободные ограничения.

- Интегрировать выбранный на предыдущем шаге алгоритм в выбранную графовую базу данных.

- Расширить язык запросов выбранной базы данных конструкциями, необходимыми для выражения контекстно-свободных ограничений.

- Произвести замеры производительности полученного решения и сравнить его с существующими решениями.

2. Обзор

2.1. Терминология

Контекстно-свободной грамматикой называется $G = (\Sigma, N, P, S)$, где Σ — алфавит терминальных символов, N — алфавит нетерминальных символов, P — множество правил вида $A \to \alpha$, где $A \in N$, $\alpha \in (\Sigma \cup N)^*$, а $S \in N$ — выделенный стартовый нетерминал.

Языком L над алфавитом Σ называется любое подмножество 2^{Σ^*}. Языком, порождаемой грамматикой G, является множество $L(G) = \{S \overset{*}{\Rightarrow} \beta, \beta \in \Sigma^*\}$, где $S \overset{*}{\Rightarrow} \beta$ означает, что из нетереминала S путём последовательного применения правил грамматики выводится β.

Контекстно-свободная грамматика $G = (\Sigma, N, P, S)$ находится в ослабленной нормальной форме Хомского, если любое её правило имеет вид $A \to BC$, где $A, B, C \in N$, либо $A \to a$, где $A \in N, a \in \Sigma$. В отличие от нормальной формы Хомского в ослабленной, во-первых, допускается присутствие стартового нетерминала S в правых частях правил грамматики, во-вторых, запрещаются правила вида $S \to \epsilon$, где ϵ — пустая строка.

В задаче поиска путей с ограничениями в виде формальных языков дан граф (V, E), разметка его рёбер $l : E \to \Sigma$ и язык L над алфавитом Σ. Требуется найти множество всех пар вершин, между которыми существует путь, метки на рёбрах которого образуют слово в заданном языке. То есть требуется найти следующее множество:

$$\{(v, to) : \exists p = (e_1, ..., e_n) \in E^* : l(e_1)...l(e_n) \in L,\ src(e_1) = v,\ dst(e_n) = to\}$$

Здесь $src(e)$ и $dst(e)$ для $e \in E$ означают начальную и конечную вершину ребра e. В данном контексте язык L называется языком ограничений.

Задача поиска путей с контекстно-свободными ограничениями — это задача поиска путей в виде формальных языков, в которой язык задаётся контекстно-свободной грамматикой.

2.2. Графовые базы данных

Графовые СУБД[1] (далее просто графовые базы данных) — это разновидность СУБД, в которой данные хранятся в виде графов. В отличие от других разновидностей, в графовых базах данных отношения между объектами так же важны, как и сами объекты.

Основной моделью представления графов в таких базах данных является gpraph property model [?]. В ней каждая сущность может содержать набор свойств в формате ключ-значение. Основными сущностями являются узлы и отношения. Узлы соответствуют вершинам графа и помимо свойств могут иметь несколько меток. Отношения соответствуют рёбрам и имеют ровно одну метку, которая называется типом отношения. На рисунке ?? показан небольшой пример графа в такой модели.

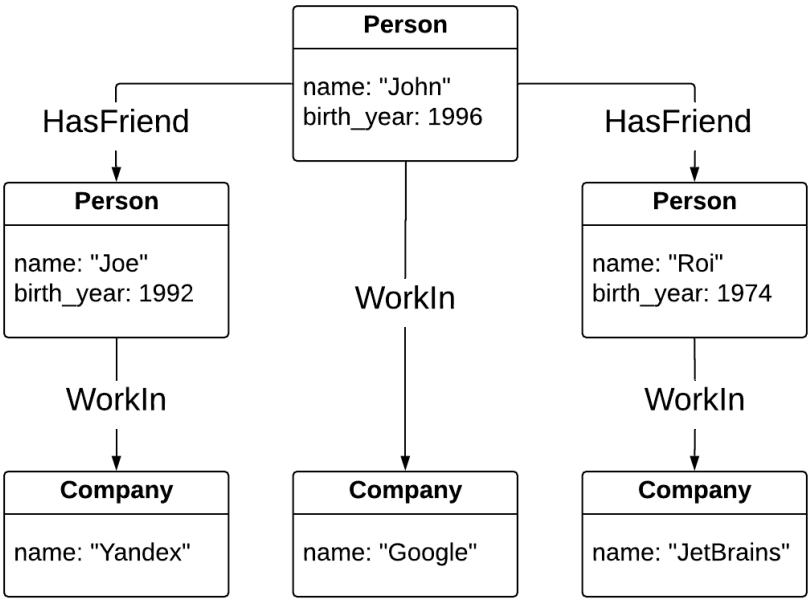

Рис. 1: Пример социального графа

Для работы с графами графовые базы данных предоставляют язык запросов, самым популярным из которых является Cypher [?]. В нём главный интерес представляют запросы вида сопоставления с образцом. Они позволяют задавать интересующие пути или подграфы в ви-

[1]СУБД — Система управления базами данных

де шаблонов и описывать информацию, которую нужно извлечь после удачного сопоставления. На рисунке **??** приведён пример такого запроса. Шаблон пути описывается в выражении MATCH. В нём между круглыми скобками задаются шаблоны вершин, а между квадратными шаблоны рёбер. Таким образом в данном примере задаются следующие ограничения: путь должен начинаться из вершины с меткой Person и именем John и состоять из двух рёбер, первое из которых должно иметь тип HasFriend, а второе WorkIn. В выражении RETURN задаётся информация, которую нужно извлечь. В данном примере это имя последней в пути вершины. В итоге ответом на такой запрос являются имена компаний, в которых работают друзья Джона. Результатом работы этого запроса на графе из рисунка **??** является множество {"Yandex", "JetBrains"}.

```
MATCH (p:Person)-[:HasFriend]->()-[:WorkIn]->(to)
WHERE p.name = ''John''
RETURN to.name
```

Рис. 2: Пример конечного запроса на языке Cypher

С формальной точки зрения шаблоны пути в выражении MATCH позволяют поставить задачу поиска путей с ограничениями в виде формальных языков. Так в запросе на рисунке **??** языком ограничений является конечный язык $\{(HasFriend, WorkIn)\}$. Для запроса на рисунке **??** ограничением является регулярный язык $\{A, B\}^*$.

```
MATCH (v)-[:A | :B *]->(to)
RETURN to.name
```

Рис. 3: Пример регулярного запроса на языке Cypher

При этом регулярные ограничения поддерживаются лишь частично и позволяют искать только пути произвольной длины с заданными метками на рёбрах, а более глубокие регулярные выражения не поддерживаются. Поэтому на текущий момент язык запросов довольно ограничен.

2.3. Существующие решения

Как было упомянуто раннее, ни одна графовая база данных не поддерживает запросов с контекстно-свободными ограничениями. Тем не менее существуют альтернативные решения поддержки таких запросов.

2.3.1. Парсер-комбинаторы для Neo4j

В 2018 году группой исследователей из JetBrains Research на основе библиотеки Meerkat была разработана библиотека для поддержки запросов с контекстно-свободными ограничениями [**?**]. Она использует графовую базу данных Neo4j [**?**] как хранилище графов и позволяет задавать запросы в виде парсер-комбинаторов. Основным достоинством данной работы является то, что с помощью этой библиотеки кроме контекстно-свободных запросов можно выразить базовую часть языка Cypher. Но так как конкурировать с оригинальной реализацией выполнения запросов Neo4j очень сложно, это достигается вместе с сильной потерей производительности. Кроме этого контекстно-свободные запросы обрабатываются также достаточно медленно.

Таким образом данное решение является альтернативой языка запросов Neo4j, а не его расширением. Из-за медленного времени работы такое решение подходит только для работы с небольшими графами.

2.3.2. Расширение языка запросов SPARQL

В 2016 Сяованг Чжан предоставил язык cfSPARQL [**?**] — расширение языка SPARQL, который способен выразить запросы с контекстно-свободными ограничениями. Также он привёл алгоритм для вычисления таких запросов и замеры производительности. Но, во-первых, работа была сделана вне контекста графовой базы данных, а во-вторых, время работы предложенного алгоритма было больше, чем время работы парсер-комбинаторов.

2.3.3. Существующие реализации алгоритмов решения задачи контекстно-свободной достижимости

Основной сложностью расширения языка запросов для поддержки запросов с контекстно-свободными ограничениями является долгое время работы соответствующих алгоритмов. Так, например, в 2019 году Йохем Куйперс и другие исследователи с целью попытки расширения языка запросов Cypher для графовой базы данных Neo4j произвели сравнительный анализ производительности наиболее известных алгоритмов решения задачи контекстно-свободной достижимости.

В данном исследовании были рассмотрены и произведены замеры времени работы алгоритма Элле Хелингса [**?**], основанного на атрибутных грамматиках, восходящего алгоритма Фреда Сантоса [**?**], матричного алгоритма Рустама Азимова [**?**] и алгоритма Петтери Севона [**?**]. Все алгоритмы были интегрированы в Neo4j и запускались на графах, находящихся в её хранилище. Алгоритмы были написаны на языке Java, при этом их реализация являлась однопоточной.

По результатам замеров производительности было показано, что время работы алгоритмов является слишком большим и неприемлемым для широкого практического использования. Поэтому дальнейшая работа по интеграции и расширению языка запросов была приостановлена.

Тем не менее, матричный алгоритм Рустама Азимова в сравнительном анализе Йохема Куйперса был реализован без необходимых матричных библиотек, которые могут сильно уменьшить время его работы. Так, например, в исследовании Никиты Мишина и др. [**?**] был произведён сравнительный анализ времени работы нескольких реализаций алгоритма Рустама Азимова, основанных на различных специализированных матричных библиотеках. Графы и запросы к ним были взяты из объемлющего набора данных CFPQ_Data [**?**], предоставленного лабораторией языковых инструментов JetBrains Research.

Результаты замеров Никиты Мишина и др. показали, что при грамотной реализации алгоритма Рустама Азимова и использовании под-

ходящих матричных библиотек можно добиться очень высокой производительности. Поэтому, так как основной проблемой применимости запросов с контекстно-свободными ограничениями является долгое время работы соответствующих алгоритмов, в качестве алгоритма решения задачи контекстно-свободной достижимости был выбран матричный алгоритм Рустама Азимова.

2.4. Матричный алгоритм Рустама Азимова

Выбранный в предыдущей главе алгоритм Рустама Азимова [**?**], в отличие от других алгоритмов решения задачи контекстно-свободной достижимости [**?**, **?**, **?**], работает с графами в виде разреженных матриц смежности. Данный алгоритм состоит из последовательности операций над разреженными матрицами, время работы которых зависит не от размеров матричных операндов, а от количества их ненулевых элементов.

На вход алгоритму (см. алгоритм 1) поступает помеченный граф $D = (V, E)$ и контекстно-свободная грамматика $G = (\Sigma, N, P, S)$ в ослабленной нормальной форме Хомского. Для каждого нетерминала A в ассоциативном массиве T хранится соответствующая ему булева матрица $T[A]$. На всём этапе алгоритма поддерживается следующий инвариант: $T[A]_{i,j} = 1$ равносильно существует пути, метки на рёбрах которого образуют слово, выводящееся из нетерминала A. На первом этапе происходит инициализация матриц с помощью простых правил грамматики, после чего инвариант выполняется для всех путей единичной длины. На втором этапе происходит транзитивное замыкание, после чего этот инвариант верен для всех путей. Результатом данного алгоритма является матрица, соответствующая стартовому нетерминалу S.

Практическое время работы алгоритма Рустама Азимова сильно зависит от производительности используемой матричной библиотеки. Это накладывает некоторые ограничения на выбор подходящей графовой базы данных, так же как и возможность представления графов в матричном виде.

Algorithm 1 Матричный алгоритм Рустама Азимова

```
1:  function contextFreePathQuerying(D, G)
2:      n = getNodeCount(D)
3:      N = getAllNonterms(G)
4:      E = getEdges(D)
5:      P = getRules(G)
6:      S = getStartNonterm(G)
7:      T = { A → ∅_{n×n} : A ∈ N }
8:      for all (v, to, label) ∈ E do          ▷ Инициализация матриц
9:          for all A → label ∈ P do
10:             T[A]_{i,j} = 1
11:     while ∃A : T[A] is changing do          ▷ Вычисление замыкания
12:         for all A → BC ∈ P do
13:             T[A] ⊕= T[B] ⊗ T[C]
14:     return T[S]
```

2.5. RedisGraph

RedisGraph [**?**] — это высокопроизводительная графовая база данных, поддерживающая язык запросов Cypher. В отличие от наиболее распространённой графовой базы данных Neo4j [**?**], RedisGraph написан на языке Си и для работы с данными использует Redis [**?**], основным достоинством которого является возможность хранить данные прямо в оперативной памяти. Это позволяет RedisGraph быстро обрабатывать пользовательские запросы.

Также RedisGraph является единственной графовой базой данных, которая работает с графами в виде разреженных матриц смежности и транслирует запросы языка Cypher в матричные выражения. Для представления графов в таком виде и работы с ними в терминах линейной алгебры используется мощный матричный фреймворк GraphBlas [**?**]. Его реализация SuiteSparse [**?**] является многопоточной и сильно оптимизирована, что позволяет RedisGraph добиться высокой производительности.

Из всего этого следует, что RedisGraph идеально подходит для интеграции матричного алгоритма. Во-первых, графы представляются в необходимом алгоритму виде, что позволит избежать издержек на кон-

вертацию форматов. Во-вторых, использование SuiteSparse для вычисления матричных операций позволит добиться высокой производительности. Поэтому RedisGraph был выбран в качестве графовой базы данных для интеграции матричного алгоритма и расширения языка запросов.

2.6. Расширение языка Cypher

На текущий момент оригинальная версия языка Cypher, используемая в том числе и в RedisGraph, не поддерживает запросов с контекстно-свободными ограничениями. Но тем не менее в 2017 году был разработан черновой вариант спецификации расширения Cypher [**?**], которая вводит в язык шаблоны путей. Они позволяют выразить более сложные запросы, в том числе запросы с контекстно-свободными ограничениями.

Шаблоны путей являются альтернативой шаблонам рёбер, которые есть в оригинальном Cypher. Они, как и шаблоны рёбер, могут встречаться в выражении MATCH и иметь своё направление. Кроме этого в глобальной области запроса им можно задавать имя, на которое потом можно ссылаться внутри других шаблонов.

Шаблон пути представляет из себя регулярное выражение над некоторыми примитивами. В качестве таких примитивов могут выступать шаблоны рёбер, шаблоны вершин и ссылки на именованные шаблоны путей. Также любым подвыражениям можно задавать своё направление. Основная часть конкретного синтаксиса данного расширения приведена на рисунке **??**.

Каждый шаблон пути задаёт отношение на множестве вершин. Поэтому семантикой языка шаблонов путей L_P в контексте графа $G(V, E)$ является отображение $[\![\cdot]\!]_G : L_P \to V \times V$, которое каждому шаблону $p \in L_P$ сопоставляет множество пар вершин, между которыми существует путь, удовлетворяющий данному шаблону p.

Подробное описание данной семантики приводится в таблице **??**. В ней наибольший интерес представляют именованные шаблоны путей,

$$
\begin{aligned}
PathPattern &= [\text{"}<\text{"}],\ \text{"}-/\text{"},\ PathExpression,\ \text{"}/-\text{"},\ [\text{"}>\text{"}] \\
PathExpression &= \{PathAlternative\} \\
PathAlternative &= PathRepetition,\ \{\text{"}|\text{"}, PathRepetition\} \\
PathRepetition &= [\text{"}<\text{"}],\ PathBase,\ [\text{"}>\text{"}],\ (\text{"}*\text{"}) \\
PathBase &= PathEdge \\
&\mid PathNode \\
&\mid PathReference \\
&\mid \text{"}[\text{"},\ PathExpression,\ \text{"}]\text{"} \\
PathEdge &= Label \\
PathNode &= \text{"}(\text{"},\ [Label,\ \{\text{"}|\text{"},\ Label\}],\ \text{"})\text{"} \\
PathReference &= \text{"}\sim\text{"},\ SymbolicName; \\
Label &= \text{"}:\text{"},\ LabelName
\end{aligned}
$$

Рис. 4: Расширение конкретного синтаксиса Cypher

так как именно с помощью них можно выразить запросы с контекстно-свободными ограничениями. Все именованные шаблоны путей $S_i = p_j$ можно рассматривать как правила контекстно свободной грамматики с алфавитом нетерминалов $\{S_i\}_{i=1}^n$. Тогда каждый нетерминал S_j порождает язык $L_{S_j} \subset L_p$, а семантикой соответствующего именованного шаблона пути $S_j = p_j$ является множество $\bigcup\limits_{p \in L_{s_j}} [\![p]\!]_G$.

На рисунке **??** приведён пример запроса в расширенном синтаксисе. В нём декларируется именованный шаблон S, который задаёт множество правильных скобочных последовательностей над ребрами с типом L и R. Далее в выражении MATCH задаётся шаблон пути, состоящий из ссылки на шаблон S. Таким образом результатом обработки запроса является множество всех пар вершин, между которыми существует путь, метки на рёбрах которого образуют правильную скобочную последовательность.

Данная спецификация расширения Cypher была представлена официальными разработчиками и сильно расширяет выразительность языка, предоставляя удобную возможность выражать запросы как с регу-

$p \in L_P$	$[\![p]\!]_G$	Описание шаблона пути
()	$\{(v, v) : v \in V\}$	Пустой путь, состоящий из одной произвольной вершины
:a	$\{e = (v, to) : e \in E, type(e) = a\}$	Путь единичной длины, состоящий из ребра с типом a
(:b)	$\{(v, v) : v \in V, label(v) = b\}$	Пустой путь, состоящий из одной вершины, помеченной меткой b
$\alpha\ \beta$	$[\![\alpha]\!]_G \circ [\![\beta]\!]_G$	Конкатенация путей α и β
$\alpha \mid \beta$	$[\![\alpha]\!]_G \cup [\![\beta]\!]_G$	Альтренатива между путями α и β
$[\alpha]$	$[\![\alpha]\!]_G$	Квадратные скобки позволяют группировать выражения для задания ассоциативности
$<\alpha$	$\{(to, v) : (v, to) \in [\![\alpha]\!]_G\}$	Путь, обратный к пути α
$<\alpha>$	$[\![\alpha \mid <\alpha]\!]_G$	Альтернатива между путём α и обратным к нему
α^*	$[\![\alpha]\!]^*_G$	Путь, состоящий из конкатенации 0 или более путей α
$\{S_i = p_i\}_{i=1}^n$ – named path patterns	$P = \{S_i \to p_i\}_{i=1}^n$ $Gram_j = (\Sigma, \{S_i\}_{i=1}^n, P, S_j)$ $[\![S_j]\!]_G = \bigcup_{p \in L(G)} [\![p]\!]_G$	Именнованые шалоны путей
$\sim S$	$[\![S]\!]_G$	Ссылка на именованный шаблон пути

Таблица 1: Семантика языка шаблонов путей

```
PATH PATTERN S = ()−/ [:L ~S :R] | [~S ~S] | () /−()
MATCH (v)−/ ~S /−(to)
RETURN v, to
```

Рис. 5: Пример запроса в расширенном синтаксисе Cypher

лярными, так и с контекстно-свободными ограничениями. Поэтому в моей работе приводится поддержка выполнения запросов именно для этого расширения языка.

3. Реализация

По результатам обзора было решено реализовать поддержку расширения языка Cypher, представленную в главе **??**, для графовой базы данных RedisGraph. За основу алгоритма, решающего задачу поиска путей с контекстно-свободными ограничениями, был взят матричный алгоритм Рустама Азимова, описанный в главе **??**.

3.1. План выполнения запроса

В RedisGraph основной частью обработки запроса является построение плана его выполнения. Её часть, которая относится к шаблонам путей приведена на рисунке **??**. В ней зелёным цветом выделено то, что было добавлено или расширено.

В самом начале, после получения запроса строится его абстрактное синтаксическое дерево *AST*. Далее, из него извлекаются именованные и неименованные шаблоны путей *PathPattern* и *NamedPathPatterns*, после чего они преобразуются в более удобное промежуточное представление *PathExpr*. При этом, для дальнейшего связывания ссылок, именованные шаблоны сохраняются в глобальном контексте запроса *PathPatternCtx*.

На следующем этапе происходит трансляция промежуточных представлений *PathExpr* в матричные выражения *AlgebraicExpression*. В них операндами являются либо матрицы, полученные из указанного в запросе графа *GraphCtx*, либо ссылки на именованные шаблоны путей из *PathPatternCtx*. Основной идеей трансляции является то, что после вычисления матричного выражения получается матрица, которая задаёт то же самое отношение на множестве вершин, что и исходный шаблон пути.

Далее каждое такое выражение формирует новую операцию плана выполнения запроса *CfpqTraverseOp*. При её вычислении сначала происходит запуск расширенной версии матричного алгоритма Рустама Азимова. Он решает задачу контекстно-свободной достижимости, заданной именованными шаблонами путей. После этого все ссылки в матричном выражении заменяются на полученные в ходе алгоритма матрицы и

происходит вычисление матричного выражения. Каждая такая операция добавляется в план выполнения запроса.

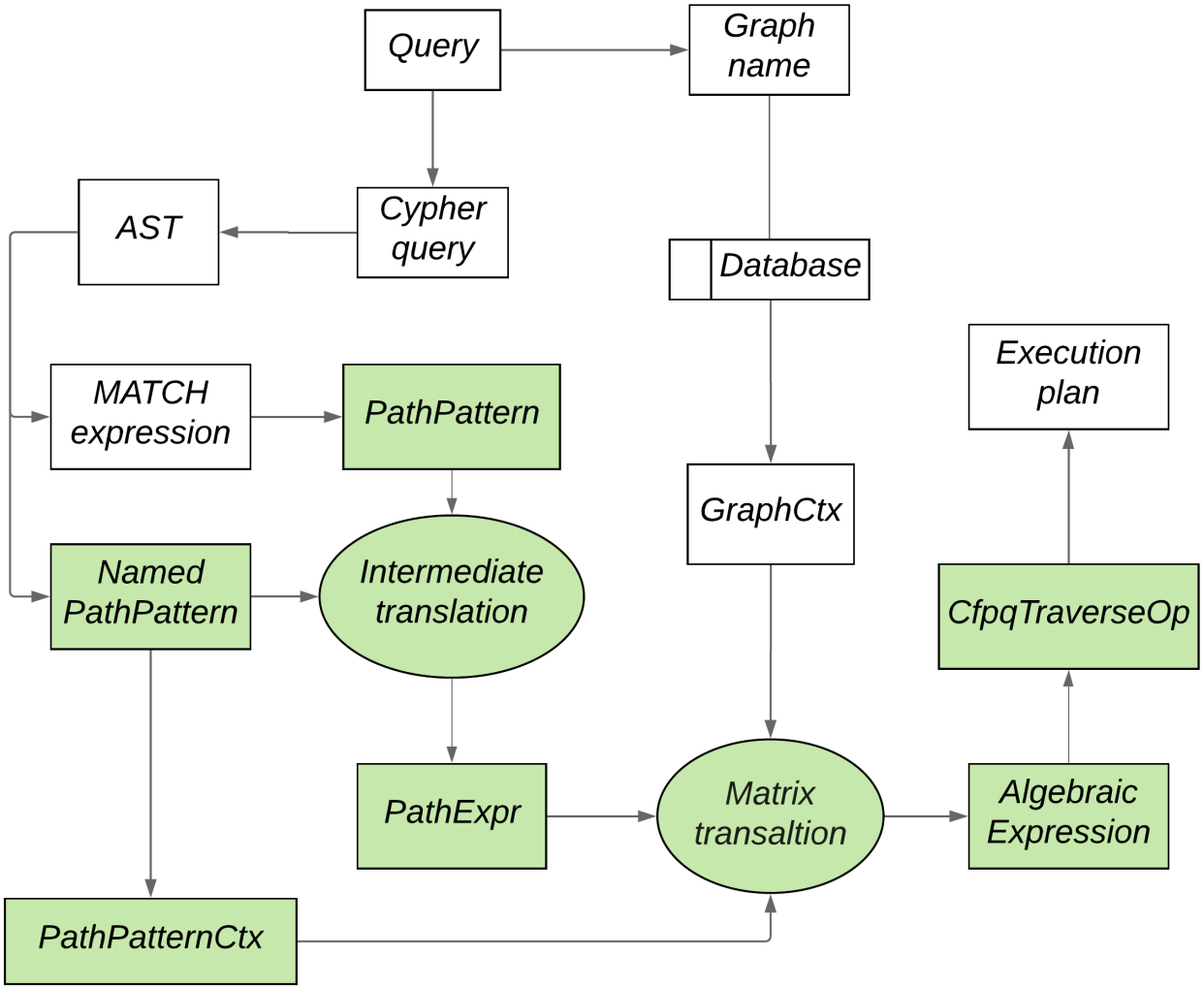

Рис. 6: Расширение построения плана выполнения запроса

3.2. Промежуточное представление

Шаблоны путей, полученные из *AST*, транслируются в промежуточное представление *PathExpr*. Оно позволяет задавать более простой абстрактный синтаксис, который описан на рисунке **??**.

Таким образом, альтернативе и конкатенации шаблонов соответствуют *PathAlt* и *PathSeq*, *PathGroup* позволяет задавать направление пути и наличие замыкания, а *PathBasic* соответствует либо примитивным шаблонам *PathNode*, *PathEdge* и *PathRef*, либо целому выражению

PathExpr. Примеры промежуточного представления шаблонов приведены в таблице **??**.

$$
\begin{aligned}
PathExpr = {} & PathSeq(PathExpr,\ PathExpr)\ | \\
& PathAlt(PathExpr,\ PathExpr)\ | \\
& PathGroup(PathBasic, direction, range) \\
PathBasic = {} & PathNode(label)\ | \\
& PathEdge(type)\ \ | \\
& PathRef(name)\ \ | \\
& PathExpr \\
direction \in {} & \{inbound,\ outbound,\ bidirectional\} \\
range \in {} & \{*, \varnothing\}
\end{aligned}
$$

Рис. 7: Промежуточное представление PathExpr

L_p	PathExpr
[:A :B] | (:C)	*PathAlt*(*PathSeq*(*PathEdge*("*A*"), *PathEdge*("*B*")) *PathNode*("*C*"))
<[:A ∼S]*	*PathGroup*(*PathSeq*(*PathEdge*("*A*"), *PathRef*("*S*")), *inbound*, *,)

Таблица 2: Примеры промежуточного представления запросов

3.3. Трансляция в матричные выражения

Как было упомянуто ранее, RedisGraph представляет графы в виде разреженных матриц. А именно каждый граф G задаётся следующей тройкой $(A \in M_{n\times n},\ lab \in Labels \to Diag_n,\ rel \in RelTypes \to M_{n\times n})_G$. Здесь $M_{n\times n}$ означает полукольцо булевых матриц, а $Diag_n$ полукольцо диагональных булевых матриц. Матрица A служит матрицей смежности графа, а отображения lab и rel сопоставляют меткам вершин и типам рёбер соответствующие булевы матрицы. Таким образом, ребро (v, to) графа G имеет тип a тогда и только тогда, когда $rel(a)_{v,to} = 1$. Таким же образом, принадлежность метки l вершине v равносильно $lab(l)_{v,v} = 1$.

Любую булеву матрицу M, участвующую в задании графа $G(V, E)$, можно рассматривать как отношение на множестве вершин $R(M) = \{(v, to) :\ M_{v,to} = 1\}$. Операциями сложения и умножения в булевом полукольце являются дизъюнкция и конъюнкция. Поэтому умножению матриц $A * B$ соответствует композиция отношений $R(A) \circ R(B)$, сложению $A + B$ соответствует объединение отношений $R(A) \cup R(B)$, а транспонированная матрица A^T соответствует обратному к R(A) отношению $R(A)^{-1}$. Такая взаимосвязь между матричными операциями и отношениями лежит в основе алгоритма трансляции, приведённом на рисунке **??**.

Данный алгоритм является рекурсивным и принимает на вход промежуточное представление шаблона пути $expr$, представление графа g и контекст именованных шаблонов путей $pathCtx$. Целевым языком трансляции является простой язык матричных выражений, приведённый на рисунке **??**.

Базовым случаем рекурсии являются примитивные шаблоны *PathNode*, *PathEdge* и *PathReference*, которые транслируются в операнды матричного выражения. Для первых двух соответствующие матрицы извлекаются из графа с помощью функций *GetLabelMatrix* и *GetRelationMatrix*. При этом случай $label = \varnothing$ соответствует шаблону пути, состоящему из одной произвольной вершины. Поэтому такой путь задаёт-

ся тождественным отношением $R(I)$, где I — единичная матрица. Для *PathReference* создаётся ссылка на матрицу именованного шаблона, которая будет вычислена на следующем этапе при выполнении алгоритма контекстно-свободной достижимости.

Трансляция для шаблонов *PathSeq* и *PathAlt* происходит одинаковым образом — сначала происходит трансляция дочерних шаблонов, а потом из полученного результата образуются операции умножения или сложения. Такая трансляция обосновывается семантикой шаблонов альтернативы и конкатенации, приведенной в главе **??**, и связью отношений с матричными операциями, описанными раннее.

Наиболее интересным случаем является трансляция *PathGroup*, так как в некоторых случаях контекст именованных шаблонов $pathCtx$ расширяется. В начале происходит трансляция дочернего шаблона. Далее, если заданное направление является обратным, к полученной матрице применяется операция транспонирования. Если же направление является произвольным, то формируется операция сложения из полученной матрицы и транспонированной к ней. Это соответствует альтернативе между прямым путём и обратным к нему. После этого при отсутствии замыкания результат возвращается. Иначе происходит трансляция замыкания полученного выражения. Так как его нельзя выразить через имеющиеся матричные операции, создаётся новый именованный шаблон, а замыкание заменяется ссылкой на него. Нетрудно показать, что регулярное выражение R^* и контекстно-свободная грамматика с одним правилом $S \to R\ S \mid \epsilon$ равносильны. Поэтому правая часть этого правила тривиальным образом сразу же транслируется в выражение *Add(Mul(R, MatrixRef(S)), I)* и добавляется в *pathCtx* вместе с полученным новым именем.

Таким образом, после работы данного алгоритма из промежуточного представления шаблона пути получается выражение над матрицами. При дальнейшем его вычислении получается матрица, которая соответствует такому же отношению на множестве вершин, как и семантика изначального шаблона.

Algorithm 3 Алгоритм трансляции в матричные выражения

```
1: function TRANSLATE(PathExpr expr, GraphCtx g, PathPatternCtx pathCtx)
2:     switch expr
3:         case PathNode(label):
4:             if label == ∅ then
5:                 return GetIdentityMatrix(g)
6:             else
7:                 return GetLabelMatrix(g, label)
8:         case PathEdge(type):
9:             return GetRelationMatrix(g, type)
10:        case PathRef(name):
11:            return MatrixRef(name)
12:        case PathSeq(left, right):
13:            return Add(TRANSLATE(left), TRANSLATE(right))
14:        case PathAlt(left, right):
15:            return Mul(TRANSLATE(left), TRANSLATE(right))
16:        case PathGroup(basic, dir, range):
17:            res = TRANSLTE(basic)
18:            switch dir
19:                case inbound:
20:                    res = Transpose(res)
21:                case bidirectional:
22:                    res = Add(res, Transpose(res))
23:            switch range
24:                case ∅:
25:                    return res
26:                case ∗:
27:                    name = AllocateNewPathPattern(ctx)
28:                    res = Mul(res, MatrixRef(name))
29:                    res = Add(res, GetIdentityMatrix(g))
30:                    SetPathPetternExpression(p, name, res)
31:                    return MatrixRef(name)
```

$$
\begin{aligned}
AlgExpr = \; & Add(AlgExpr, AlgExpr) \mid \\
& Mul(AlgExpr, AlgExpr) \mid \\
& Transpose(AlgExpr) \mid \\
& Matrix \mid \\
& MatrixRef(ref)
\end{aligned}
$$

Рис. 8: Алгебраическое выражение над матрицами

3.4. Формирование и вычисление операции плана выполнения

После этапа трансляции в RedisGraph происходит построение плана выполнения запроса. Он формируется из последовательности операций, которые выполняют базовые вычисления. Для поддержки шаблонов путей была добавлена операция *CfpqTraverseOp*. Она создаётся для каждого матричного выражения, полученного на предыдущем шаге из неименованого шаблона пути, и отвечает за его вычисление.

На этапе инициализации новой операции *CfpqTraverseOp* из соответствующего матричного выражения рекурсивно извлекаются все ссылки на именованные шаблоны путей, от которых зависит данное выражение. После этого они поступают на вход алгоритма, решающего задачу контекстно-свободной достижимости (см. алгоритм 4).

Этот алгоритм является расширенной версией матричного алгоритма Рустама Азимова, приведенного в главе **??**. В данном алгоритме, в отличие от алгоритма Рустама Азимова, не требуется задавать правила грамматики в ослабленной нормальной форме Хомского. Вместо этого правая часть правила задаётся с помощью промежуточного представления *PathExpr*. При этом подразумевается, что для любого именованного шаблона p из глобального контекста *pathCtx* его промежуточное представление уже транслировано в матричное выражение и записано в *pathCtx[p].algExpr*.

Принцип работы алгоритма остаётся прежним. На каждой итерации для всех невычисленных шаблонов происходит вычисление матричного выражения. Если результирующая матрица не изменяется, то она является окончательной для данного именованного шаблона и он больше не участвует в обновлении. Иначе соответствующая матрица перезаписывается.

После работы данного алгоритма все ссылки на именованные шаблоны путей в матричном выражении заменяются на подсчитанные алгоритмом матрицы, после чего происходит вычисление матричного выражения. Результат сохраняется в операции $CfpqTraverseOp$ и участвует в вычислении плана выполнения запроса наряду с результатами других операций.

Algorithm 4 Расширенный матричный алгоритм

```
1: function CfpqTraverseNew(String[] patterns, PathPatternCtx pathCtx)
2:     while ∃ p ∈ patterns: !pathCtx[p].isEvaluated do
3:         for all p ∈ patterns do
4:             if !pathCtx[p].isEvaluated then
5:                 new_matrix = EvalueteAlgExpr(pathCtx[p].expr)
6:                 if new_matrix == pathCtx[p].matrix then
7:                     pathCtx[p].isEvaluated = true
8:                 else
9:                     pathCtx[p].matrix = new_matrix
```

4. Замеры производительности

После реализации поддержки нового синтаксиса шаблонов путей были произведены замеры производительности.

4.1. Сравнение с парсер-комбинаторами

Сравнительный анализ времени работы полученного решения (колонка RedisGraph) и библиотеки парсер-комбинаторов (колонка Meerkat), описанной в главе **??**, приведён в таблице **??**. Запросы были взяты из эксперимента оригинальной статьи про парсер-комбинаторы [**?**]. Эквивалентные им запросы, написанные в расширенном синтаксисе Cypher, приведены на рисунках **??**, **??** и представляют из себя частный случай запросов поиска объектов, лежащих на одном уровне иерархии. Набор графов также был взят из вышеупомянутого эксперимента и впервые был представлен в статье Сяованга Чжана [**?**].

Замеры обоих решений производились локально на оборудовании со следующими характеристиками: Intel Core i7 4×1.8GHz, 8 GB RAM. Каждый запрос запускался 20 раз и время его работы усреднялось. Время работы указано в миллисекундах. Также в колонках $|V|$ и $|E|$ указано количество вершин и рёбер графа, а в колонке $\#result$ количество найденных соответствующим запросом пар вершин.

G	$\|V\|$	$\|E\|$	Query_1			Query_2		
			#result	Meerkat time (ms)	RedisGraph time (ms)	#result	Meerkat time (ms)	RedisGraph time (ms)
wine	773	2450	66572	541	31	133	6	3
pizza	671	2604	56195	476	24	1262	30	4
measure-primitive	341	771	15156	158	11	2871	39	5
funding	778	1480	17634	99	14	1158	14	6
atom-primitive	291	685	15454	102	10	122	53	3
people-pets	337	834	9472	55	7	37	3	3
travel	131	397	2449	21	3	63	2	2

Таблица 3: Сравнение Meerkat и полученного решения

```
PATH PATTERN S = ()−/ [<:Type      [~S | ()] :Type] |
                      [<:SubClass [~S | ()] :SubClass] /−()
MATCH (v)−/ ~S /−>(to)
RETURN COUNT(*)
```

Рис. 9: Query_1

```
PATH PATTERN S = ()−/ :SubClass | [<:SubClass ~S :SubClass] /−()
MATCH (v)−/ ~S /−>(to)
RETURN COUNT(*)
```

Рис. 10: Query_2

По результатам замеров видно, что даже на небольших графах время работы Meerkat сильно больше, чем время работы полученного решения. При этом в большинстве случаев оно отличатся на порядок. Также стоит отметить, что запросы, указанные на рисунках **??**, **??**, помимо расширенного синтаксиса используют и оригинальную часть языка Cypher, а конкретно функцию COUNT. Это является небольшим примером того, что расширение языка запросов является полностью совместимым с его оригинальной частью.

4.2. Сравнение с матричным алгоритмом

Графы, приведенные в предыдущих замерах являются достаточно маленькими, поэтому также были произведены замеры на более больших графах. Они были взяты из набора данных CFPQ_Data [**?**], собранного исследователями лаборатории языков инструментов JetBrains Research. Замеры производились таким же образом и на том же оборудовании, что и в главе **??**.

Кроме этого, для анализа издержек выполнения запроса в таблице **??** приводится время работы оригинального алгоритма Рустама Азимова (колонка Matrix algorithm), описанного в главе **??**. Данный алгоритм был интегрирован в RedisGraph и запускался на графах, находящихся в его хранилище. Для этого была разработана отдельная команда, принимающая на вход название графа и путь до файла с грамматикой, написанной в нормальной форме Хомского. Для вычисления

матричных операций также использовалась библиотека SuiteSparse.

Таким образом, во-первых, время работы оригинального алгоритма не включает в себя издержки, возникающие при выполнении запроса внутри графовой базы данных. Во-вторых, оригинальный алгоритм отличается от алгоритма, используемого при выполнении запроса в расширенном синтаксисе. Тем не менее время работы обоих решений отличается не сильно и является достаточно небольшим для применения на практике.

G	$\|V\|$	$\|E\|$	Query_1			Query_2		
			#result	Matrix algorithm time (ms)	RedisGraph time (ms)	#result	Matrix algorithm time (ms)	RedisGraph time (ms)
go	272770	1068622	304070	1272	1236	334850	662	683
go-hierarchy	45007	1960436	588976	271	276	738937	193	290
eclass-514	48815	219390	90994	198	304	96163	121	241
enzyme	239111	1047454	396	103	47	8163	68	37

Таблица 4: Сравнение матричного алгоритма и полученного решения

4.3. Сравнение с анализом Йохема Куйперса

Также был произведён замер времени работы на очень большом графе geospeices [**?**]. Этот граф является довольно важным, потому что он участвовал в сравнительном анализе алгоритмов, проведенным Йохемом Куйперсом. Именно из-за колоссального времени работы запроса на данном графе дальнейшее расширение языка запросов Йохемом Куйперсом и др. было приостановлено.

Повторить эксперимент не предоставилось возможным, так как в статье не приводились ссылки на реализацию алгоритмов. Поэтому в таблице **??** приводится замер из оригинальной статьи алгоритма с наилучшим временем работы (колонка Neo4j). Время указано в секундах. Эквивалентный запрос в расширенном синтаксисе приводится на рисунке **??**. Характеристики оборудования, на которых выполнялись запросы, следующие:

- Neo4j: Intel Xeon E5-4610 v2, 8×2.30GHz, 400 GB RAM

- RedisGraph: Intel Core i7-6700 CPU, 64 GB RAM 4×3.4GHz

```
PATH PATTERN S = ()−/ [:broaderTransitive [~S | ()] <:broaderTransitive] /−()
MATCH (v)−/ ~S /−>(to)
RETURN COUNT(*)
```

Рис. 11: Query

G	$\|V\|$	$\|E\|$	#result	Neo4j time (s)	RedisGraph time (s)
geospeices	225 000	1 550 000	226 669 749	6 953.9	26.1

Таблица 5: Сравнение с замером Йохема Куйперса

По результатам замеров видно, что удалось достичь времени работы в десятки секунд. Такое время было обозначено Куйперсом как приемлемое время работы для практического применения.

4.4. Выводы

По результатам замеров времени выполнения можно говорить о том, что полученное решение делает запросы с контекстно-свободными ограничениями доступными для практического применения. При этом новый синтаксис языка сильно расширяет его возможности и является полностью совместимым с его оригинальной версией.

Заключение

В ходе работы были получены следующие результаты:

- Выполнен обзор текущих решений поддержки запросов с контекстно-свободными ограничениями, по результатам которого было решено интегрировать матричный алгоритм Рустама Азимова в графовую базу данных RedisGraph с последующим расширением языка запросов Cypher.
- Интегрирован матричный алгоритм Рустама Азимова в RedisGraph.
- Разработана поддержка расширения языка запросов Cypher для RedisGraph, позволяющая задавать запросы с контекстно-свободными ограничениями. Исходный код находится в репозитории на github [**?**]. Также для удобства и возможности позапускать запросы без процесса установки необходимого программного обеспечения предоставляется docker контейнер [**?**].
- Произведены замеры производительности полученного решения и сравнение времени работы с текущими аналогами.
- Результаты работы изложены в статье, принятой на конференцию GRADES-NDA 2020.

В будущем планируется разработать подробную пользовательскую документацию запросов в расширенном синтаксисе, так как черновой вариант официальной спецификации рассчитан больше на разработчиков. Также планируется отправить запрос на принятие изменений в официальный репозиторий RedisGraph.