Санкт-Петербургский государственный университет

Кафедра системного программирования

Кириллов Илья Олегович

Парсер-комбинаторы для запросов к графовым базам данных

Курсовая работа

Научный руководитель:
к. ф.-м. н., доц. Григорьев С. В.

Санкт-Петербург
2018

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering

Ilya Kirillov

Parser Combinators for Graph Database Querying

Coursework

Scientific supervisor:
associate professor Semyon Grigorev

Saint-Petersburg
2018

Оглавление

Введение

В двадцать первом веке люди оперируют огромными объёмами информации. Для её хранения принято использовать базы данных (БД), в которых данные представлены в структурируем виде. Самая популярная структура БД — реляционная, в рамках которой информация организована в виде набора связанных таблиц. Но это не всегда удобно, особенно, если исходная структура данных далека от табличной. Часто она оказывается близка к структуре графа. Для таких данных существуют графовые БД. В такой БД вершины — это сущности (аналог записей в реляционных базах данных), соединённые рёбрами (аналог отношений между записями). Одной из самых популярных графовых БД является Neo4j [11]. Хорошим примером применения графовых баз данных являются социальные сети, в графовом представлении которых вершинами являются участники этой соцсети, а рёбрами — отношения между ними. В ситуации, когда графовая модель данных является естественной для предметной области, графовые БД могут значительно превосходить реляционные по производительности и, ко всему прочему, имеют более наглядное представление данных [14].

Существуют множество способов для работы с графовым базам данных. Одним из таких способов является использование строкового языка запросов (в случае с Neo4j это язык Cypher — SQL-подобный декларативный язык). Слабой строной такого подхода является отстуствие проверки корректности запросов на этапе компиляции и, соответственно, большая вероятность возникновения труднобнаружимых ошибок. Одним из возможных решений этой проблемы является использование ORM-технологий (Object-Relational Mapping), позволяющих работать с данными на специальном DSL (Domain Specific Language). Недостатком данного подхода является сложность работы с составлением и переиспользование подзапросов, а так же отсутствие контроля над результирующим запросом.

В некоторых случаях запрос к графовым БД можно представить

как путь в графе с некоторыми ограничениями. Такие ограничения могут быть описаны с помощью контекстно-свободной грамматики. Контекстно-свободная грамматика, в свою очередь, может быть представлена с помощью технологии парсер-комбинаторов. В этом случае грамматика описывается в терминах функций и операций над ними. Особенно этот подход популярен в функциональных языках, как Haskell или Scala. Такой подход обладает следующими преимуществами: возможность описать запрос к графовой базе данных, не прибегая к помощи других языков программирования, проверка типов на этапе компиляции, композициональность (возможность составлять сложные запросы из подзапросов). В результате и логика программы, и запросы к БД создаются на одном языке, с использованием одних и тех же подходов и инструментов.

Для языка программирования Scala существует большое количество библиотек для работы с парсер-комбинаторами. Для данной работы была выбрана библиотека Meerkat, так как большинство библиотек предназначены в основном для работы с грамматиками языков программирования, а Meerkat гарантировано работает с произвольными контекстно-свободными грамматиками, которые могут потребоваться [1]. Так же в Meerkat есть базовая поддержка работы с графами [19], которая была улучшена в рамках данной работы.

В рамках данной работы создан инструмент, позволяющий описывать запросы к графовым БД с помощью парсер-комбинаторов.

1. Постановка задачи

Целью данной работы является поддержка графовой базы данных Neo4j в библиотеке парсер-комбинаторов Meerkat для языка программирования Scala. Для её достижения были поставлены следующие задачи:

- реализовать удобный интерфейс для представления входных данных в виде графа;
- реализовать поддержку работы с вершинами;
- реализовать интерфейс для графа, представленного в виде БД Neo4j;
- провести экспериментальное исследование.

2. Обзор

2.1. Графовые базы данных

Графовая база данных — это удобный способ хранения данных, структура которых близка к структуре графа. В качестве примеров таких данных можно привести: граф отношений между участниками социальной сети, граф ссылок в научных статьях, структуру локальной сети.

Основными понятиями, которыми оперируют графовые БД, являются:

- сущности — вершины в графе, основная концепция графовых БД. Обычно связаны с другими сущностями, а так же имеют список свойств, хранящих информацию о конкретной сущности;
- отношения — рёбра в графе. Соединяют две сущности. Так же могут иметь свойства;
- свойства — именованные значения, по которым может осуществляться поиск в БД. Привязанны к сущностям или отношениям;
- метки — значения, группирующие сущности или отношения. У сущности может быть несколько меток.

Neo4j одна из самых популярных графовых СУБД на данный момент. Она имеет поддержку JVM языков, к каковым относится Scala, а так же у Neo4j есть богатый Java API, позволяющий работать с БД на низком уровне.

Для работы с графовой базой данных Neo4j для языка Scala существуют следующие решения [16]:

- язык Cypher — декларативный язык запросов, похожий на SQL для реляционных БД;
- низкоуровневое API для работы со встроенными базами данных;

- Neo4j Java Driver — Java-драйвер для подключения к Neo4j серверу;
- Neo4j HTTP API — интерфейс для подключения к серверу Neo4j.

2.2. Парсер-комбинаторы

Технология парсер-комбинаторов [6] позволяет описывать контекстно-свободные грамматики через исполняемый код, выраженный функциями высших порядков (то есть такими функциями, которые в качестве аргументов или возвращаемого значения могут иметь другие функции).

Основная идея парсер-комбинаторов заключается в получении новых парсеров из уже существующих благодаря использованию функций над парсерами. Парсер представляет из себя функцию, которая принимает текущую позицию во входных данных и возвращает либо успешный результат и новую позицию для продолжения синтаксического разбора, либо ошибку. Для языка программирования Scala тип парсера и простейший парсер, принимающий только определённый токен, могут быть описаны следующим образом:

```
type Parser = Position => Option[Position]
def token(l: String) =
    (pos: Position) =>
        if (input(pos) == l) Some(pos + 1)
        else None
```

Здесь `Parser` — это функция, которая принимает на вход позицию для разбора и возвращает результат, упакованный в контейнер `Option[Position]`. Данный контейнер является типом-суммой, принимающий либо значение типа `Position`, либо `None`. `input` — это список лексем, принимаемый парсером на вход, а `token` — функция, возвращающая парсер, который принимает на вход только лексему `l`, а для других возвращает ошибку.

Парсер-комбинаторы — это функции, принимающие существующие парсеры и возвращающие парсер, полученный нужной комбинацией

исходных. Для соответствующей композиции парсеров класс парсер обычно объявляется монадой [7] и композиция осуществляется с помощью функции `>>=` (`bind`). В языке Scala для этих целей служит метод `flatMap`. Для контейнера `Option[A]` он будет иметь вид `flatMap[B](f: A =>Option[B]): Option[B]`. `flatMap`, в случае успеха, применяет переданную функцию к значению, хранящемуся в контейнере и возвращает полученный результат, а в случае неудачи возвращает значение `None`.

В качестве примера рассмотрим конкатенирующий комбинатор:

```
def concat(first: Parser, second: Parser): Parser =
    (pos: Position) => first(pos) flatMap second
```

Здесь описана функция, принимающая в качестве входных данных два парсера и возвращающая новый парсер, который сначала запускает первый парсер, а затем, в случае его успешного завершения, передаёт управление второму. Если во время работы какой-то из парсеров сообщил об ошибке, то и результирующий парсер выдаст ошибку.

Парсер-комбинаторы, в отличие от классических решений для описания грамматик, например, как Yacc [8], не требуют кодогенерации, что значительно упрощает отладку кода и позволяет переиспользовать уже существующие парсеры.

Контекстно-свободную грамматику, представленную в EBNF-нотации [17], очень просто описать с помощью парсер-комбинаторов. Нетерминалы — это парсеры. Последовательности терминалов/нетерминалов описываются с помощью комбинатора конкатенации. Операция ”или” из EBNF-нотации так же представляет из себя комбинатор, называющийся комбинатором альтернации.

2.3. Библиотека Meerkat

Библиотека Meerkat реализует парсер-комбинаторы на языке программирования Scala и обладает следующими преимуществами [1]:

- Meerkat работает с произвольными контекстно-свободными грамматиками (в том числе и с леворекурсивными);

- Meerkat в худшем случае работает за $O(n^3)$ от длины входных данных;
- Meerkat позволяет получить всевозможные деревья разбора, которые требуются для задачи поиска пути в графе.

Meerkat по умолчанию поддерживает следующие стандартные комбинаторы (Табл. 1).

Комбинатор	Описание
`a ~ b`	сначала `a`, потом `b`
`a | b`	`a` или `b`
`a.?`	`a` или ничего
`a.*`	ноль или больше `a`
`a.+`	один или больше `a`

Таблица 1: Стандартные комбинаторы библиотеки Meerkat

В Meerkat уже существует поддержка работы с графами, реализованная в рамках работы [19], но она обладает следующими недостатками: слишком большое время работы на графах, переусложнённый интерфейс, представляющий входные данные, невозможность работы с вершинами графа, невозможность работы с метками, отличными от строк, а так же невозможность создавать запросы, начинающиеся только с произвольной вершины графа.

2.4. Синтаксический анализ графов

Проблема Context-Free Language Reachability (CFL-R) были впервые сформулирована в [18] и заключается в следующем. Возьмём контекстно-свободную грамматику G с множеством терминалов T и нетерминалов N. Обозначим за $L(G, S),\ S \in N$ язык, порождённый грамматикой G со стартовым нетерминалом S. Также рассмотрим ориентированный граф D, рёбра которого помечены терминалами из множества T. Возьмём теперь произвольный путь p в графе D и сопоставим каждому ребру в данном пути его метку из множества терминалов G. Получим слово из алфавита T^*. Обозначим это слово

как $l(p)$. Если $l(p) \in L(G, S)$ для некого $S \in T$, то назовём такой путь S-путём. Context-Free Language Reachability включает в себя следующие проблемы [10]:

- проблема поиска всех пар вершин (v_1, v_2) в графе D, таких, что существует S-путь в D из v_1 в v_2;
- проблема поиска всех вершин v_2 в графе D, таких, что существует S-путь в D из v_1 в v_2 для фиксированной вершины v_1;
- проблема поиска всех вершин v_1 в графе D, таких, что существует S-путь в D из v_1 в v_2 для фиксированной вершины v_2;
- проблема, заключающаяся в ответе на вопрос: существует ли S-путь в D из v_1 в v_2 для фиксированных вершин v_1 и v_2.

2.5. Запросы к графовой БД с помощью парсер-комбинаторов

CFL-R может быть применён для запросов к графовой базе данных. Для этого запрос представляется в виде контекстно-свободной грамматики, в которой терминалы — это метки отношений в графовой БД. Данная грамматика выражается в виде набора парсеров, где каждый нетерминал — это отдельный парсер. Затем для нужного стартового нетерминала S выполняется поиск всех S-путей в графовой базе данных.

2.6. Существующие решения

Рассмотрим некоторые решения для работы с графовыми БД.

Библиотека Trails [4] — библиотека комбинаторов для обхода графа, аналог парсер-комбинаторов, позволяющие описывать с помощью комбинаторов пути в графах. Как и Meerkat, он позволяет удобно переиспользовать существующие запросы запросы и имеет проверку запросов на этапе компиляции. Недостатком данной библиотеки

является невозможность работать с лево-рукурсивными грамматиками и правильно обрабатывать циклы.

OpenCypher [3] — это SQL-подобный декларативный язык запросов для графой БД Neo4j. Поддерживает класс контекстно-свободных запросов, но, как и все встроенные языки, обладает отсутствием копозициональности и проверки корректности запросов на этапе компиляции.

Алгоритм на базе GLL [5], реализованный для языка программирования F#, позволяет описывать запросы к графам в виде КС грамматик в EBNF-нотации, Недостатками является необходимость кодогенерации и отсутствие поддержки IDE.

3. Реализация

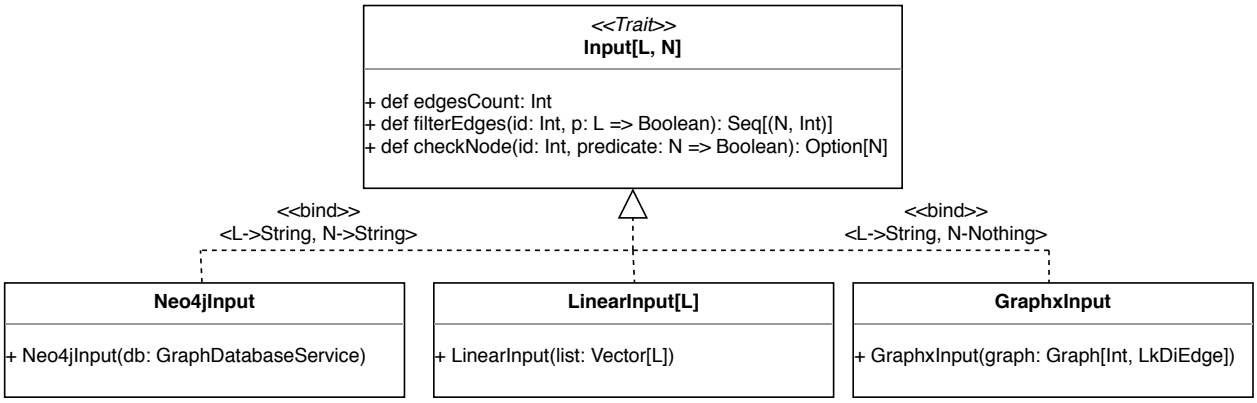

Рис. 1: Диаграмма классов, представляющих входные данные

В библиотеке Meerkat уже была реализация для работы с графами, добавленная в рамках работы [19]. Основные проблемы данный реализации: неудобство расширения, долгое время работы запросов и отсутствие поддержки вершин и предикатов. Она была упрощена до реализации, которая основана на следующей идее. Изначально библиотека Meerkat предназначена для синтаксического анализа строк. Строку можно считать линейным графом, в котором рёбра последовательно помечены символами из исходной строки. Что позволяет обобщить интерфейс входных данных до функции, возвращающей по переданной текущей вершине и предикату все исходящие из данной вершины рёбра, удовлетворяющие данному предикату. В случае со строкой эта функция будет просто возвращать для каждой позиции в строке соответствующий ей символ. Для работы с произвольными графами так же была добавлена поддержка вершин. Для этого в интерфейс для входных данных была добавлена функция, позволяющая узнать, удовлетворяет ли нужная вершина переданному предикату. Всё это было инкапсулировано в трейт (Scala аналог интерфейса) `Input` (Рис. 3), где-то `L` — это тип метки ребра, `N` — тип метки вершины.

На данный момент этот интерфейс реализован для трёх источников входных данных:

- `Neo4jInput` — входные данные, представленные в виде графовой БД Neo4j;
- `LinearInput` — входные данные, представляющие из себя линейную последовательность нетерминалов;
- `GrapxInput` — входные данные, представленные в виде графа, хранящегося в оперативной памяти.

Описанный выше интерфейс позволяет получать информацию о рассматриваемом графе внутри библиотеки Meerkat. Для составления запросов к графу были добавлены комбинаторы для работы вершинами и рёбрами:

- Комбинатор `E[L](p: L =>Boolean)` для работы с рёбрами, возвращающий парсер, принимающий только рёбра, для меток которых предикат `p` истинен;
- Комбинатор `V[N](p: N =>Boolean)` для работы с вершинами, возвращающий парсер, принимающий только вершины, для меток которых предикат `p` истинен.

Если вернуться к примеру про граф социальной сети, то, например, запрос для поиска всех пар друзей, с именами, начинающимся на букву ”A”, будет выглядеть следующим образом:

```
val startsWithA = _.name.startsWith(''A'')
val query = V(startsWithA) ~ E(''frend_of'') ~ V(startsWithA)
```

В случае, когда от рёбер требуется только их метка, комбинатор `E` можно опускать, как, например, показано на Рис. 3.

Библиотека Meerkat к началу работы с ней умела искать пути, начинающиеся только со специальной помеченной вершиной. Для поиска всех S-путей для некого нетерминала S реализован алгоритм, который заключается в следующем (Рис.3). Процесс синтаксического анализа запускается, начиная с каждой из вершин графовой базы данных. Все результаты сохраняются в структуру `SPPFLookup`,

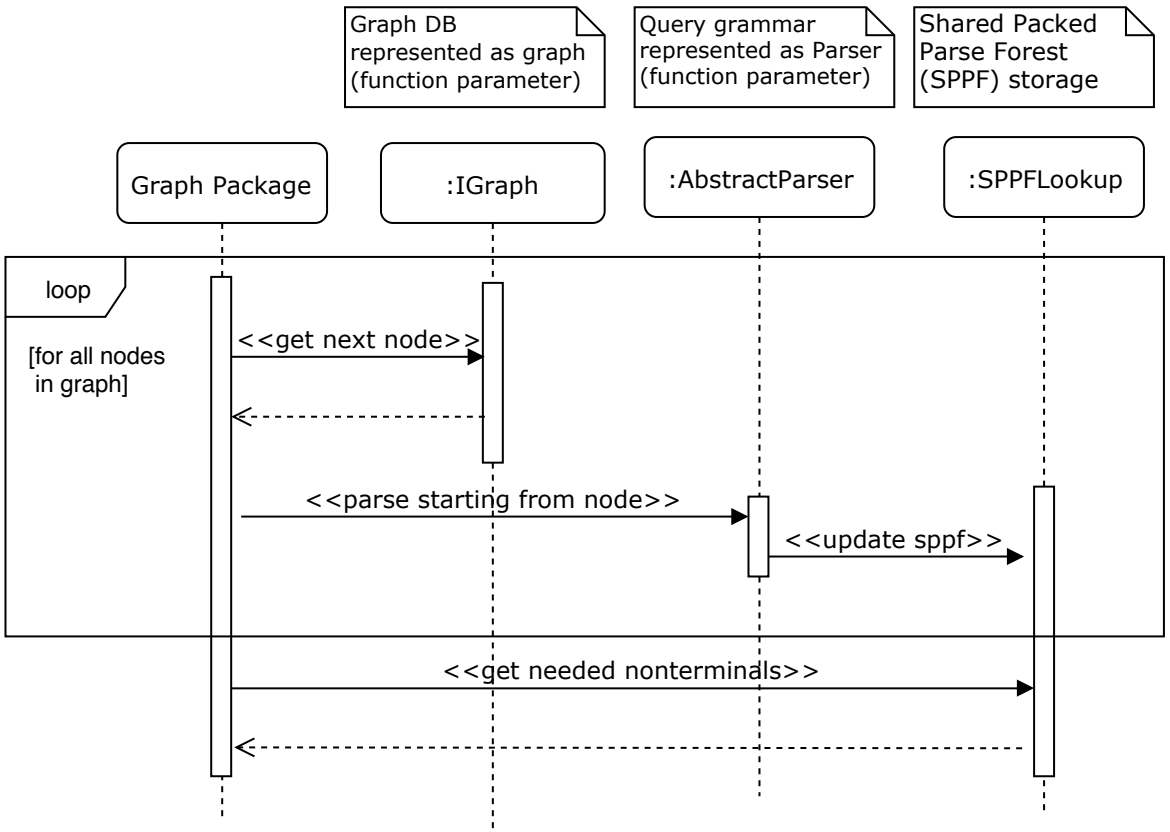

Рис. 2: Алгоритм работы функции для получения всех S-путей

представляющую из себя хранилище для SPPF [13] (Shared Packed Parse Forest) — сжатое представление леса разбора. Затем из этой структуры извлекаются нужные нетерминалы в виде троек (`name: Any, leftExtent: Int, rightExtent: Int`), которые представлют из себя искомые пути в графе, где `name` — имя нетерминала, а `leftExtent` и `rightExtent` — это номера вершин, являющиеся началом и концом пути, соответственно.

Онтология	Тройки	Запрос 1					Запрос 2				
		Кол-во результатов	Граф в памяти (мс)	Neo4j (мс)	Trails (мс)	GLL (мс)	Кол-во результатов	Граф в памяти (мс)	Neo4j (мс)	Trails (мс)	GLL (мс)
atom-primitive	425	15454	63	27	2849	232	122	67	27	453	19
biomedical-measure-primitive	459	15156	110	18	3715	482	2871	111	18	60	26
foaf	631	4118	12	0	432	29	10	11	1	1	1
funding	1086	17634	69	9	367	179	1158	71	9	76	13
generations	273	2164	4	1	9	12	0	4	1	0	0
people_pets	640	9472	38	1	75	80	37	38	1	2	1
pizza	1980	56195	326	17	7764	793	1262	332	17	905	50
skos	252	810	1	1	6	6	1	1	2	0	0
travel	277	2499	12	1	34	21	63	11	1	1	2
univ-bench	293	2540	9	1	31	24	81	10	1	2	1
wine	1839	66572	399	2	3156	606	133	423	6	4	5

Таблица 2: Сравнение Meerkat, Trails и GLL на онтологиях

4. Экспериментальное исследование

В этом разделе приведено эксперементальное исследование полученного результата. В качестве набора данных был взят классический набор онтологий [2] и проведено сравнение времени работы на двух запросах из [5]. Сравнение произведено с существующими аналогами: библиотекой Trails [9]и алгоритмом GLL [5].

Все тесты проведены на машине под управлением операционной системой Fedora 27, четырёхъядерным Intel Core i7 2.5 GHz, и восьмью гигабайтами памяти.

Набор онтологий из [2] — это известный подход для тестирования запросов к графам. В рамках данной работы было проведено тестирование производительности на данном наборе онтологий, каждая из которых представленна в виде RDF [2] файла. В начале RDF файлы были преобразованы в помеченные орграфы следующим образом: для каждой тройки вида ($object$, $predicate^{-1}$, $subject$) созданы два ребра ($subject$, $predicate$, $object$) и ($object$, $predicate^{-1}$, $subject$). Затем полученные графы были либо сразу загруженны в память, либо помещены в базу данных Neo4j. После чего к каждому из полученных графов были применены два запроса из [5]. Грамматики данных запросов представленны на Рис. 4, а их представление в виде комбинаторов на Рис. 3. Для тестирования была использована библиотека ScalaMeter [15].

```
val Q1 = syn(
  ''subclassof-1'' ~ Q1.? ~ ''subclassof'' | ''type-1'' ~ Q1.? ~ ''type'')

val S = syn(''subclassof-1'' ~ S.? ~ ''subclassof'')
val Q2 = syn(S ~ ''subclassof'')
```

Рис. 3: Запросы Q1 и Q2 на языке комбинаторов

$$
\begin{aligned}
Q1 &\to subclassof^{-1}\ Q1?\ subclassof \mid type^{-1}\ Q1?\ type \\
S &\to subclassof^{-1}\ Q1?\ subclassof \\
Q2 &\to S\ subclassof
\end{aligned}
$$

Рис. 4: Грамматика запросов Q1 и Q2 в EBNF-нотации

Результаты работы представленны в таблице 2, где столбец "тройки" показывает размер соответствующего RDF файла, а "Кол-во результатов" — количество пар вершин для которых существует по крайней мере один S-путь между ними.

Алгоритм GLL и решение на базе комбинаторов показывает одни и те же результаты (столбец "Кол-во результатов"), в то время как решение на базе Meerkat на некотрых графах примерно в два раза быстрее, чем решение на базе алгоритма GLL для запросе Q1. На запросе Q2 Meerkat уступает по производительности алгоритму GLL. Если сравнивать производительность запросов для графа, хранящегося в памяти, и графа, помещённого в базу данных, то, что естественно, производительность первого быстрее примерно в $2-4$ раза. И, наконец, библиотека Trails значительно уступает обоим решениям по времени работы.

Итого, полученное решение превосходит рассмотренные аналоги по производительности.

5. Заключение

В ходе работы были достигнуты следующие результаты:

- реализован интерфейс для представления входных данных в виде графа, который позволяет удобно создавать новые источники входных данных;
- реализована поддержка работы с вершинами, а так же добавлена поддержка предикатов;
- реализована поддержка работы с графовой базой данных Neo4j для библиотеки Meerkat;
- проведено экспериментальное исследование, а именно сравнение с существующими решениями: библиотекой парсер-комбинаторов Trails и алгоритмом GLL, которое показало, что реализованное решение на базе комбинаторов быстрее, чем рассматриваемые аналоги.

Дальнейшим возможным развитием работы является:

- поддержка конъюнктивных грамматик [12], которые могут требоваться для некоторых запросов из сферы статического анализа;
- распараллеливание процесса синтаксического анализа, которое может потребоваться в случае работы с БД большого объёма.

Список литературы

[1] Anastasia Izmaylova Ali Afroozeh Tijs van der Storm. Practical, General Parser Combinators // PEPM '16 Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. — 2016. — P. 1–12.

[2] Context-free path queries on RDF graphs / Xiaowang Zhang, Zhiyong Feng, Xin Wang et al. // International Semantic Web Conference. — Springer, 2016. — P. 632–648.

[3] Cypher Query Language. — `https://neo4j.com/developer/cypher-query-language/`.

[4] Daniel Kröni Raphael Schweizer. Parsing Graphs: Applying Parser Combinators to Graph Traversals // Scala '13, Montpellier, France. — 2013.

[5] Grigorev Semyon, Ragozina Anastasiya. Context-free Path Querying with Structural Representation of Result // Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia. — CEE-SECR '17. — New York, NY, USA : ACM, 2017. — P. 10:1–10:7. — URL: `http://doi.acm.org/10.1145/3166094.3166104`.

[6] Hutton Graham. Higher-Order Functions for Parsing // Journal of Functional Programming. — 1992. — Vol. 2. — P. 323–343.

[7] Hutton Graham, Meijer Erik. Functional pearl: Monadic parsing in Haskell. — 1998. — 07. — Vol. 8.

[8] Johnson S. C. YACC - Yet Another Compiler-Compiler. — 1975. — Vol. 32.

[9] Kröni Daniel, Schweizer Raphael. Parsing Graphs: Applying Parser Combinators to Graph Traversals // Proceedings of the 4th Workshop

on Scala. — SCALA '13. — New York, NY, USA : ACM, 2013. — P. 7:1–7:4. — URL: http://doi.acm.org/10.1145/2489837.2489844.

[10] Melski D. Reps T. Interconvertibility of a class of set constraints and context-free-language reachability // Theoretical Computer Science. — 2000. — Vol. 248. — P. 29–98.

[11] Neo4j official website. — https://neo4j.com/product/.

[12] Okhotin Alexander. Conjunctive and Boolean grammars: The true general case of the context-free grammars // Computer Science Review. — 2013. — Vol. 9. — P. 27 – 59. — URL: http://www.sciencedirect.com/science/article/pii/S157401371300018X.

[13] Rekers Joan Gerard. Parser generation for interactive environments : Ph. D. thesis / Joan Gerard Rekers ; Universiteit van Amsterdam. — 1992.

[14] Robinson Ian, Webber Jim, Eifrem Emil. Graph Databases. — O'Reilly Media, Inc., 2013. — ISBN: 1449356265, 9781449356262.

[15] ScalaMeter main site. — https://scalameter.github.io/.

[16] Using Neo4j from Java. — https://neo4j.com/developer/java/.

[17] Wirth Niklaus. Extended Backus-Naur Form (EBNF) // ISO/IEC. — 1996. — Vol. 14977. — P. 2996.

[18] Yannakakis M. Graph-theoretic methods in database theory // Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, POD. — 1990.

[19] С.К. Смолина. Реализация синтаксического анализатора графов с помощью парсер-комбинаторов. — 2017.